Llamado de procedimientos a distancia y abstracción de tipos

María Virginia Aponte
Conservatoire National des Arts et Métiers

Département d'Informatique, CNAM,
292 rue Saint Martin, 75003 Paris, France
E-mail: aponte@margaux.inria.fr

Xavier Leroy
INRIA

Projet Cristal, B.P. 105,
78153 Le Chesnay, France.
E-mail: Xavier.Leroy@inria.fr

Resumen

En este artículo estudiamos la relación entre el llamado de procedimientos a distancia (RPC) y los lenguajes con tipaje estático y abstracción de tipos. En particular, mostramos como explotar la información de tipos afin de reducir el tiempo de transmisión de datos a través de la red. Con este propósito, desarrollamos una formalisación simple que describe la generación automática de interfaces eficientes de comunicación. Terminamos nuestro estudio con una prueba de corrección que muestra la equivalencia entre la evaluación local y la evaluación distribuída de todo programa.

1 Introducción

El llamado de procedimientos a distancia o RPC [12, 5] es un modelo de descripción para un cálculo de procesos distribuídos [2]. En RPC, el principal instrumento de comunicación entre procesos es el llamado de procedimientos y un llamado remoto no difiere de un llamado local, es decir, el carácter remoto de los llamados es transparente para el usuario.

La figura 1 muestra un esquema de comunicación entre dos procesadores A y B por medio de un llamado remoto a la función f que reside en B. Si desde A se hace el llamado remoto $f(a)$, éste se efectúa en realidad sobre una *función de interface del cliente* que llamamos f_{clien} y que es local a A (en inglés, *stub code*). El papel de f_{clien} es en primer lugar, de establecer una conexión a través de la red y de transmitir el argumento a a B. B utilisa una *función de interface del servidor* f_{serv} que aplica localmente f a a y envía el resultado r del llamado de regreso a A. La transmisión de a y del resultado r a través de la red requiere un codaje (en inglés, *marshaling*): la adopción de un formato común para los enteros y flotantes, y la descomposición de las estructuras de datos en secuencias formadas por sus componentes. Así, la función f_{clien} codifica el argumento a antes de enviarlo a B, f_{serv} lo decodifica antes de aplicarlo a f, y simétricamante, f_{serv} codifica el resultado r a ser transmitido y f_{clien} lo decodifica antes de retornarlo como resultado final del llamado.

El modelo RPC es bastante utilisado en las aplicaciones distribuídas. Dos ejemplos mayores son el sistema de distribución de archivos a distancia NFS, y NIS, el sistema de distribución de palabras claves y de otras informaciones de sistemas opérativos [18, 20]. La razón de esta popularidad radica en su facilidad de utilisación: programar la comunicación se resume a la escritura de la función de interface, sin necesidad de utilisar otras primitivas de comunicación.

En segundo lugar, la función de interface puede generarse automáticamente [19] pues sólo depende de la información de tipos de los argumentos y del resultado del llamado distante. Esta generación automática aumenta la transparencia de la comunicación en el modelo RPC, haciendo aún más agradable su utilisación.

RPC y tipaje estático

El modelo RPC no requiere la utilisación de lenguajes con tipaje estático. Sin embargo las ventajas y la generalidad obtenidas son mucho más grandes si se le utilisa en un contexto de tipaje fuerte y estático.

Por ejemplo, el tipaje estático garantiza que la operación de decodificación de los datos transmitidos reconstruye correctamente el dato codificado originalmente. Igualmente, dificultades para transmitir estructuras de datos se presentan si las dos entidades comunicantes no hacen las mismas hipótesis de tipos y el resultado de la comunicación es entonces imprevisible. Por ejemplo, si el receptor espera un flujo de datos más largo que el envíado por el transmisor, la comunicación permecerá bloqueada. Sin tipaje estático, es necesario realizar verificaciones de tipos dinámicamente, lo que resulta costoso en tiempo; o es necesario resconstruir sin verificar, lo cual no asegura la corrección del resultado.

Por otro parte, la utilisación de sistemas de tipo avanzados puede disminuír el tamaño de los datos trasmitidos, aumentado así la eficiencia de los llamados. Un ejemplo de sistemas de tipos con estas ventajas son los sistemas que ofrecen una noción de abstracción de tipos [6, 15, 10].

Abstracción de tipos

Sea t un tipo abstracto definido localmente al programa P. La abstracción de t significa que su representación in-

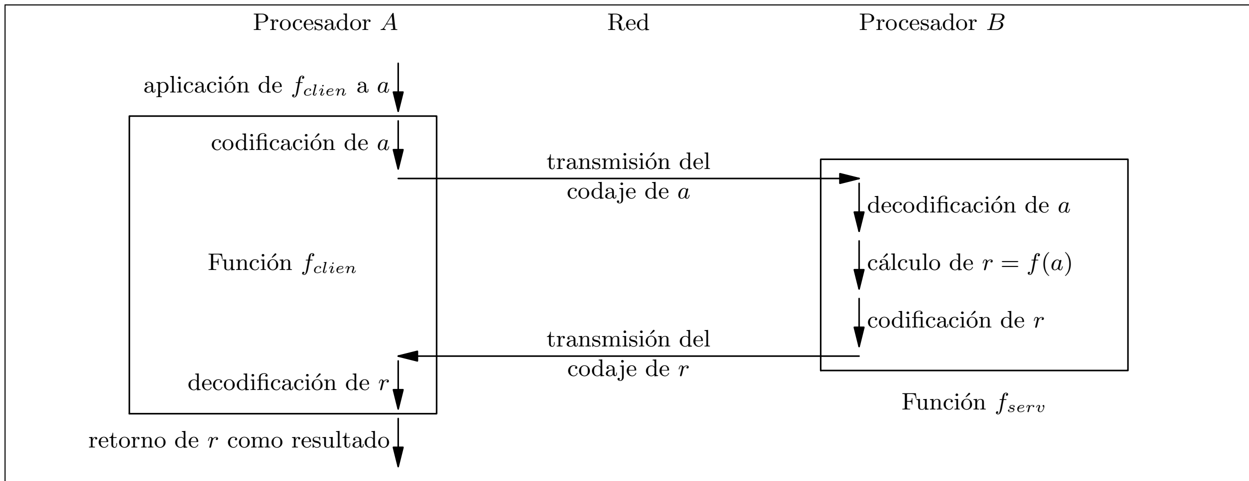


Figura 1: Llamado $f(a)$ desde A de la función distante f de B

terna es visible sólo desde P e inaccesible para cualquier otro programa. En particular, para construir o examinar valores de t, es necesario utilisar las operaciones definidas por P y, de hecho, sólo estas operaciones tienen acceso a la representación de los valores de t [9].

Los programas escritos en este tipo de lenguajes verifican propiedades de independencia con respecto a la representación de los datos extremadamente agradables en programación [16, 11]. Intuitivamente, un programa Q sólo puede manipular valores de t utilisando las funciones de P. Es fácil ver que en Q no puede haber ningún recorrido de una estructura de tipo t, ni ninguna construcción directa de un valor de ese tipo. Así, el programa Q es independiente de la estructura escogida para representar el tipo t en P. De manera general, esta propiedad, conocida también bajo el nombre de *parametricidad* [16, 11, 21, 1], aumenta la modularidad y la seguridad de los programas. En el contexto de los llamados a distancia, nosotros la utilisaremos para disminuír el costo de las transmisiones.

Continuando el ejemplo anterior, supongamos que P contiene una función f que retorna un resultado de tipo t. El programa Q hace un llamado distante a f, de manera que esta función es ejecutada en la máquina donde reside P. Su resultado debe ser trasmitido de regreso a Q, quién lo almacena en la variable v. La observación importante aquí es que, debido a la abstracción de t, Q no puede utilisar el contenido de v directamente: toda utilisación requiere un “retorno” de v hacia P. En consecuencia, es inútil transmitir hacia Q el resultado del llamado remoto (es decir v). No solamente esta información no podrá ser explotada por Q, sino que además la transmisión — que puede ser complicada y que exige una etapa de codificación y otra de decodificación — tomará un tiempo inútil.

En este artículo

Proponemos como alternativa a la transmisión clásica, el considerar que todos los valores del tipo abstracto t son locales a P, y que son representados por apuntadores a estos valores. La transmisión de datos de tipo t entre P y Q se resume entonces en un intercambio de apuntadores locales a P, reduciéndose de esta manera al mínimo el tiempo de transmisión. Más aún, la seguridad aumenta gracias a este mécanismo. En efecto, como los datos no transitan la red, éstos no pueden ser accesados o construídos artificialmente, y en particular el pirataje de la parte de una máquina externa a la comunicación es más difícil.

Estas ideas ya han sido utilisadas en la implementación de sistemas RPC, como por ejemplo en Flume [17] y en diversos sistemas distribuidos orientados por objetos [8, 4]. Sin embargo, hasta ahora ninguna descripción formal de este mécanismo ha sido ofrecida. En particular, las propiedades de parametricidad antes mencionadas han sido ampliamente estudiadas en el contexto de los lenguajes funcionales, sin que ninguna relación entre estos problemas y los modelos de RPC haya sido establecida.

En este artículo, formalizamos y mostramos la coreción del tratamiento de datos abstractos antes descrito en presencia de RPC. A este fin, desarrollamos un formalismo simple para describir la generación automática de interfaces de comunicación y su relación con un sistema de tipos abstracto.

Otros trabajos

La mayoría de los artículos que estudian los mécanismos de RPC se concentran en los aspectos de bajo nivel o relacionados con las redes de comunicación [5, 17, 3]. La interacción entre abstracción de tipos y RPC es estudiada por Herlihy y Liskov [7] pero en su solución, el programador debe proporcionar manualmente funciones

de traducción entre un tipo abstracto y una representación transmisible. Nuestro mecanismo automatiza totalmente esta etapa de traducción.

El estudio más próximo al nuestro ha sido realizado por Ohori y Kato [13], quienes consideran el caso simétrico de lenguajes con polimorfismo. Sin embargo, su solución no explota las propiedades de parametricidad del polimorfismo. En el caso de un llamado de función polimorfa, Ohori y Kato transmiten un codaje del tipo de datos, e igualmente, el valor del dato. Es fácil ver que, no solamente el volumén de datos transmitidos es más grande, sino que además una etapa de decodificación dinámica es nécasaria para efectuar el llamado remoto.

La idea de no transmitir la representación de los valores de tipos abstractos es empleadda igualmente en los sistemas distribuídos orientados a objetos. En efecto, un objeto distribuído no transmite su estado interno sino solo un apuntador a este [4]. En realidad, es posible ver un modelo simple de objetos como un caso particular de tipos abstractos donde el tipo representa el estado interno de un objeto, y las operaciones del tipo corresponden a los métodos [14]. Sin embargo, algunos de estos sistemas permiten la migración de objetos [8], mientras que esta posibilidad no puede expresarse en nuestro formalismo.

2 El lenguaje fuente

En el modelo RPC, distinguimos dos tipos de lenguajes: los lenguajes fuente y objeto. El lenguaje fuente sirve a programar cada entidad distribuída. Estos programas son traducidos automáticamente en lenguaje objeto con el fin de introducir el código necesario a la comunicación.

En esta parte presentamos el lenguaje fuente de la traducción. Este consiste básicamente en un λ-cálculo enriquecido con constantes correspondientes a los valores predefinidos del lenguaje (enteros, boleanos, etc), con pares de valores, y con módulos.

Además, el lenguaje utilisa la noción de módulo para formalizar las entidades de cálculo comunicantes. Un módulo es una colección de declaraciones de tipos y de valores, siguiendo el estilo de Modula-2 [22]. Las componentes de un módulo sont accesibles calificando el identificador de componente con el nombre del módulo: la notación $S.x$ permite accesar la componente x del módulo S.

2.1 Sintáxis

En la gramática de la figura 2, x representa a los identificadores de valores, S los nombres de módulos, y t los identificadores de tipos.

El sub-lenguaje de las expresiones está constituído por un identificador o una constante; una definición de función ($\lambda x.\, e$ representa la función de parámetro x y de cuerpo e); una aplicación de función ($f(e)$ representa la aplicación de la función f al parámetro e); una construcción de un par (e_1, e_2); una aplicación de operación primitiva (por ejemplo, $\texttt{fst}(e)$ para extraer la primera componente del par e, y $\texttt{snd}(e)$ para extraer la segunda componente); o finalemente, de un acceso $S.x$ a la componente x de un módulo S.

Enfin, un programa del lenguaje fuente es una colección de módulos seguido de una expresion que los utilisa.

Siguiendo las ideas desarrolladas en la introducción, usamos anotaciones de tipos en las declaraciones de componentes de las entidades de comunicación (módulos).

El sub-lenguaje de tipos comprende varias categorías. Los tipos predefinidos `int` y `bool`; los tipos funcionales, donde $\sigma \to \tau$ es el tipo de las funciones de argumento de tipo σ y de resultado de tipo τ; y los tipos producto, dónde $\sigma \times \tau$ es el tipo de los pares de un valor de tipo σ y de un valor de tipo τ.

Toda definición de tipo en un módulo es abstracta. Una componente de tipo t de un módulo S se utilisa con calificación ($S.t$) al exterior de S, y directamente (t) de manera local a S. Un tipo abstracto t declarado en un módulo S prohíbe el acceso de su representación interna a todo módulo distinto de S.

2.2 Semántica

En esta parte definimos la semántica dinámica del lenguaje fuente, que llamamos también *sémantica estandard*. Utilisamos un conjunto de reglas de reducción o de reescritura (ver figura 3). Notamos: $P \vdash e \xrightarrow{*} v$ la relación de reducción de una expresión e de un programa P del lenguaje fuente en un valor v. La definimos con los axiomas y las reglas de inferencia de la figura 3. Notamos las reglas de inferencia de la manera siguiente:

$$\frac{H_1 \quad \dots \quad H_n}{C}$$

que leemos: "si las hipótesis $H_1, \dots, H_n$ se cumplen, podemos concluír C".

El resultado de una evaluación en el lenguaje fuente es una expresión de forma particular, que llamamos valor. Definimos los valores, que notamos v, como el subconjunto de las expresiones, dado por la gramática siguiente:

Valores: $v ::= c \mid \lambda x.\, e \mid (v_1, v_2)$

En la regla de evaluación correspondiente al acceso $S.x$, notamos $\mathrm{Def}(x, S, P)$ la expresión asociada a $S.x$ en el programa P. Si P es de la forma

$$\begin{array}{ll} \texttt{let} \ \dots; S = [& \texttt{val}\ x_1 : \tau_1 = e_1; \\ & \dots; \\ & \texttt{val}\ x_n : \tau_n = e_n; \\ & \texttt{val}\ x : \tau = e; \\ & \dots \\ ] & \end{array}$$

(donde $x_1 \dots x_n$ son los identificadores de valores definidos antes de x en S), definimos la expresión $\mathrm{Def}(x, S, P)$ como

$$\mathrm{Def}(x, S, P) = e\{x_n \leftarrow e_n\} \dots \{x_2 \leftarrow e_2\}\{x_1 \leftarrow e_1\}.$$

Expresiones:	$e ::= x$	identificador
	$\mid c$	constante
	$\mid \lambda x.\, e$	definición de función
	$\mid e_1\ e_2$	aplicación de función
	$\mid S.x$	acceso a un identificador de otro módulo
	$\mid op(e)$	aplicación de una opéración primitiva
	$\mid (e_1, e_2)$	construcción de un par
Constantes:	$c ::= 0 \mid 1 \mid \ldots$	constantes enteras
	$\mid$ `false` $\mid$ `true`	constantes booléanas
Operaciones primitivas:	$op ::=$ `fst` $\mid$ `snd`	acceso a las componentes de un par
Tipos:	$\tau ::=$ `int` $\mid$ `bool`	tipos de base (enteros, booléanos)
	$\mid \tau_1 \rightarrow \tau_2$	tipo de función
	$\mid \tau_1 \times \tau_2$	tipo de par
	$\mid t$	nombre de tipo declarado localmente
	$\mid S.t$	nombre de tipo declarado en otro módulo
Módulos:	$M ::= \varepsilon$	módulo vacío
	$\mid$ `val` $x : \tau = e;\ M$	definición de valor
	$\mid$ `type` $t = \tau;\ M$	definición de tipo
Programas:	$P ::=$ `let` $S_1 = [M_1];\ \ldots;\ S_n = [M_n]$ `in` e	

Figura 2: Gramática del lenguaje fuente

La expresión e puede depender de identificadores de valores definidos previamente en el mismo módulo. Sería incorrecto utilisar e sin modificación al exterior de S. Así, substituímos los identificadores de los valores $x_1 \ldots x_n$ por sus definiciones.

2.3 Tipaje estático

Las reglas de tipaje estático establecen las verificaciones de tipo que efectúa un compilador del lenguaje fuente. Ellas definen la relación $P, E \vdash e : \tau$, que leemos, "en el programa P y bajo las hipótesis de E, la expresión e es tipable, y posee el tipo τ". Las hipótesis de E son asignaciones de tipos a identificadores. Si E contiene la hipótesis $x : \tau$, notamos $E(x)$ el tipo τ de x según E. Igualmente, notamos $E + x : \tau$ la adición de una hipótesis $x : \tau$ a E.

Las reglas de tipaje de módulos definen la relación $P, E \vdash S$ `ok`, que leemos "en el programa P y bajo las hipótesis de E, el módulo S es tipable". De manera análoga, establecemos una relación de tipaje para los programas. Las reglas de tipaje de las expresiones, módulos y programas son definidas en la figura 4.

De manera análoga al caso de los valores, en la regla de tipaje correspondiente al acceso en un módulo, notamos $\mathrm{Decl}(x, S, P)$ el tipo de declaración de x en el módulo S del programa P. Si P es de la forma

$$\texttt{let}\ \ldots; S = [M_1; \texttt{val}\ x : \tau = e; M_2]; \ldots\ \texttt{in}\ \ldots,$$

definimos el tipo $\mathrm{Decl}(x, S, P)$ como

$$\mathrm{Decl}(x, S, P) = \tau\{t \leftarrow S.t \mid t \text{ tipo definido en } M_1\}.$$

Así, substituímos los identificadores de tipo t definidos antes de x en S por su denominación al exterior de S, es decir $S.t$. No sería correcto de substituír t por su definición pues esto violaría su carácter abstracto.

Ejemplo:

```
S = [ type t = int;
      type u = bool;
      val x : t × u = (1, true) ]
```

El tipo externo de `S.x` es `t` × `u`{`t` ← `S.t`, `u` ← `S.u`}, es decir, `S.t` × `S.u`. □

3 Generación automática de interfaces

En esta sección presentamos las tranformaciónes de programas necesarias a la distribución de los cálculos. En nuestra presentación asumimos que la ejecución de los módulos pueden repartirse sobre máquinas diferentes, pero que éstos comunican entre si solamente a través de la red.

3.1 Los tipos `stream` y `ref`

El lenguaje objeto para la transformación incorpora dos nuevos tipos de datos: los flujos (tipo `stream`) y los apuntadores (tipo `ref`). Un flujo es una secuencia de octetos representando los mensajes que transitan la red. Un apuntador es una manera abstracta de hacer referencia a un objeto de una máquina distante; concretamente, se trata de la dirección de la máquina distante junto con la dirección que contiene el objeto en su memoria local.

Las operaciones primitivas sobre los tipos `stream` y `ref` son definidas en la figura 5.

Evaluación de expresiones:

$$P \vdash c \xrightarrow{*} c \qquad P \vdash \lambda x.\, e \xrightarrow{*} \lambda x.\, e \qquad \frac{P \vdash e_1 \xrightarrow{*} \lambda x.\, e \quad P \vdash e_2 \xrightarrow{*} v_2 \quad P \vdash e\{x \leftarrow v_2\} \xrightarrow{*} v}{P \vdash e_1(e_2) \xrightarrow{*} v}$$

$$\frac{P \vdash \mathrm{Def}(x, S, P) \xrightarrow{*} v}{P \vdash S.x \xrightarrow{*} v} \qquad \frac{P \vdash e_1 \xrightarrow{*} v_1 \quad P \vdash e_2 \xrightarrow{*} v_2}{P \vdash (e_1, e_2) \xrightarrow{*} (v_1, v_2)} \qquad \frac{P \vdash e \xrightarrow{*} (v_1, v_2)}{P \vdash \mathtt{fst}(e) \xrightarrow{*} v_1} \qquad \frac{P \vdash e \xrightarrow{*} (v_1, v_2)}{P \vdash \mathtt{snd}(e) \xrightarrow{*} v_2}$$

Evaluación de programas:

$$\frac{(\mathtt{let}\ S_1 = [M_1];\ \ldots;\ S_n = [M_n]\ \mathtt{in}\ e) \vdash e \xrightarrow{*} v}{\vdash (\mathtt{let}\ S_1 = [M_1];\ \ldots;\ S_n = [M_n]\ \mathtt{in}\ e) \xrightarrow{*} v}$$

Figura 3: Reglas de reducción del lenguaje fuente (semántica estandard)

Tipaje de constantes:

$$C(0) = C(1) = \cdots = \mathtt{int} \qquad C(\mathtt{false}) = C(\mathtt{true}) = \mathtt{bool}$$

Tipaje de expresiones:

$$P, E \vdash x : E(x) \qquad P, E \vdash c : C(c) \qquad P, E \vdash S.x : \mathrm{Decl}(x, S, P)$$

$$\frac{P, (E + x : \sigma) \vdash e : \tau}{P, E \vdash \lambda x.\, e : \sigma \to \tau} \qquad \frac{P, E \vdash e_1 : \sigma \to \tau \quad P, E \vdash e_2 : \sigma}{P, E \vdash e_1(e_2) : \tau}$$

$$\frac{P, E \vdash e_1 : \tau_1 \quad P, E \vdash e_2 : \tau_2}{P, E \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{P, E \vdash e : \tau_1 \times \tau_2}{P, E \vdash \mathtt{fst}(e) : \tau_1} \qquad \frac{P, E \vdash e : \tau_1 \times \tau_2}{P, E \vdash \mathtt{snd}(e) : \tau_2}$$

Tipaje de módulos:

$$P, E \vdash \varepsilon\ \mathtt{ok} \qquad \frac{P, E \vdash e : \tau \quad P, (E + x : \tau) \vdash M\ \mathtt{ok}}{P, E \vdash (\mathtt{val}\ x : \tau = e;\ M)\ \mathtt{ok}} \qquad \frac{P, E \vdash M\{t \leftarrow \tau\}\ \mathtt{ok}}{P, E \vdash (\mathtt{type}\ t = \tau;\ M)\ \mathtt{ok}}$$

Tipaje de programas:

$$\frac{P = (\mathtt{let}\ S_1 = [M_1];\ \ldots;\ S_n = [M_n]\ \mathtt{in}\ e) \quad P, \emptyset \vdash M_i\ \mathtt{ok} \quad (i = 1, \ldots, n) \quad P, \emptyset \vdash e : \mathtt{int}}{\vdash P\ \mathtt{ok}}$$

Figura 4: Reglas de tipaje del lenguaje fuente

newref	:	$\tau \rightarrow$ ref	crea un apuntador hacia un objeto de tipo τ arbitrario
deref	:	ref $\rightarrow \tau$	extrae el objeto apuntado
encode_int	:	int $\rightarrow$ stream	codifica un entero a transmitir
decode_int	:	stream $\rightarrow$ int	decodifica la representación de un entero
encode_bool	:	bool $\rightarrow$ stream	codifica un booléano a transmitir
decode_bool	:	stream $\rightarrow$ bool	decodifica la representación de un booléano
encode_ref	:	ref $\rightarrow$ stream	codifica un apuntador a transmitir
decode_ref	:	stream $\rightarrow$ ref	decodifica la representación de un apuntador
concat	:	stream $\times$ stream $\rightarrow$ stream	concatena dos flujos
$\texttt{head}_n$	:	stream $\rightarrow$ stream	extrae los n primeros octetos de un flujo
$\texttt{tail}_n$	:	stream $\rightarrow$ stream	salta los n primeros octetos de un flujo
remote_apply	:	ref $\times$ stream $\rightarrow$ stream	llamado remoto de función

Figura 5: Primitivas sobre los tipos `stream` y `ref`

La primitiva `newref` reserva una célula de la memoria donde almacena su argumento y devuelve en resultado un apuntador con su dirección. La primitiva `deref` lee el contenido de un apuntador. La lectura de un apuntador sólo puede realizarse localemente a la máquina donde fue creado. En otras palabras, no suponemos la distribución de la memoria sobre la red.

Las operaciones `encode_int`, `encode_bool` y `encode_ref` tranforman respectivamente un entero, un booléano o un apuntador en el flujo compuesto de la codificación en binario de estos valores. En el caso de una red con máquinas heterogéneas, es necesario adoptar un convención única de representación de los enteros (en inglés, *endianness*).

Las estructuras de datos compuestas son transmitidas utilisando la operación `concat`. Por ejemplo, para transmitir `(1,true)` construímos el flujo `s = concat(encode_int 1, encode_bool true)`.

La decodificación se hace con las primitivas `decode_int`, `decode_bool` y `decode_ref`. Ellas extraen respectivamente un entero, un booleano o un apuntador del comienzo de un flujo. La primitiva `head` permite la extracción de la parte de un flujo que va del comienzo hasta una posición dada; `tail` permite la extracción comenzando de una posición arbitraria. La primera componente de `s` (el flujo definido en el párrafo anterior) se obtiene con $\texttt{decode_int}(\texttt{head}_4\ s)$, y la segunda con $\texttt{decode_bool}(\texttt{tail}_4\ s)$, si suponemos que las representaciones transmisibles de enteros ocupan 4 octetos.

La operación `remote_apply` es la primitiva que permite la aplicación remota de funciones. Su primer argumento es un apuntador a una función distante, y el segundo es un flujo que codifica el argumento a envíar a la función. Por ejemplo, consideremos el programa siguiente:

```
let S =
 [ val f: ref =
     newref (λs. encode_int(1 + decode_int s))]
in
  decode_int(remote_apply(S.f, encode_int 3))
```

La función `remote_apply` envía el flujo resultado de `encode_int 3` a la máquina `S`. La máquina `S` lo pasa a la función apuntada por `S.f`, es decir, a λ`s. encode_int(1 + decode_int s)`, obteniendo como resultado `encode_int 4`, valor que es enviado como resultado de regreso a la máquina que realizó el llamado distante.

La primitiva `remote_apply` abstrae todos los detalles de bajo nivel del mécanismo de llamados remotos (establecimiento de conexiones de redes, etc). Sin embargo, su aplicación es limitada y difícil manualmente puesto que el argumento y el resultado de la función distante deben ser flujos. Así, para efectuar cálculos distantes sobre objetos de otros tipos es necesario introducir (como en el ejemplo anterior) varias étapas de codificación y de decodificación. En el resto de la sección mostraremos como automatizar este proceso de traducción.

3.2 El lenguaje objeto

Definimos el lenguaje objeto como el lenguaje fuente aumentado con las primitivas de la figura 5. Si las sintáxis de los lenguajes fuente y objeto son muy parecidas, la semántica del lenguaje objeto es más limitada afin de permitir la distribución (ver figura 6). En efecto, puesto que los módulos pueden residir en procesadores diferentes, los valores de cada módulo deben poder transitar la red, y deben ser entonces objetos de tipo flujo.

Por cada ocurrencia del acceso $S.x$, la regla de evaluación correspondiente requiere la lectura del valor remoto x de S a través de la red. En realidad, puesto que en nuestro sistema los valores no son modificables, no es necesario hacer un acceso al procesador distante S por cada ocurrencia de $S.x$, sino que basta con almacenar su valor en la mémoria del procesador local (en inglés, *cacheing*).

La semántica de reducción del lenguaje objeto, que llamamos también *semántica de comunicación*, es definida en la figura 6. La relación de reducción es de la forma: $P \vdash e \xrightarrow{S} v$, donde S es el procesador (módulo)

sobre el cual se efectúa la reducción. La mención del procesador de reducción permite evitar el acceso a un apuntador creado por otro procesador.

Añadimos al conjunto de valores definidos anteriormente, valores de apuntadores y de flujos. Los valores apuntadores son de la forma $\mathtt{ref}_S(v)$ donde S es el procesador donde el apuntador fue creado.

$$
\begin{aligned}
\text{Valores:}\quad v ::= {} & c \\
\mid {} & \lambda x.\, e \\
\mid {} & (v_1, v_2) \\
\mid {} & \mathtt{ref}_S(v) \\
\mid {} & \mathtt{encode_int}(v) \\
\mid {} & \mathtt{encode_bool}(v) \\
\mid {} & \mathtt{encode_ref}(v) \\
\mid {} & \mathtt{concat}(v)
\end{aligned}
$$

De manera análoga, definimos los valores φ de flujos como un sub-conjunto de los valores siguientes.

$$
\begin{aligned}
\text{Valores de flujo:}\quad \varphi ::= {} & \mathtt{empty} \\
\mid {} & \mathtt{concat}(\mathtt{encode_int}(v), \varphi) \\
\mid {} & \mathtt{concat}(\mathtt{encode_bool}(v), \varphi) \\
\mid {} & \mathtt{concat}(\mathtt{encode_ref}(v), \varphi)
\end{aligned}
$$

3.3 Funciones de codificación y de decodificación

En esta parte presentamos dos esquemas de funciones, E de codificación, y D de decodificación de datos. La función de codaje, $E_\tau(e)$, construye un flujo que codifica el valor de la expresion e vista como de tipo τ. La función de decoficación, $D_\tau(s)$, extrae del flujo s un valor de tipo τ. Definimos estas funciones inductivamente sobre la estructura del tipo τ.

3.3.1 Tipos predefinidos

En el caso de los tipos predefinidos `int` y `bool`, basta aplicar directamente las primitivas de codaje y de decodaje correspondientes.

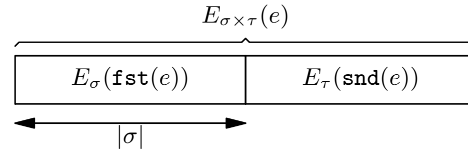
$$
\begin{aligned}
E_{\mathtt{int}}(e) &= \mathtt{encode_int}(e) \\
D_{\mathtt{int}}(s) &= \mathtt{decode_int}(s) \\
E_{\mathtt{bool}}(e) &= \mathtt{encode_bool}(e) \\
D_{\mathtt{bool}}(s) &= \mathtt{decode_bool}(s)
\end{aligned}
$$

3.3.2 Pares

En el caso de los pares, se codifica recursivamente las componentes del par, concaténandose los resultados. La primera componente se obtiene decofificando el comienzo del flujo; la segunda decoficando el final del flujo.

$$
\begin{aligned}
E_{\sigma\times\tau}(e) &= \mathtt{concat}(E_\sigma(\mathtt{fst}(e)), E_\tau(\mathtt{snd}(e))) \\
D_{\sigma\times\tau}(s) &= (D_\sigma(\mathtt{head}_{|\sigma|}(s)), D_\tau(\mathtt{tail}_{|\sigma|}(s)))
\end{aligned}
$$

La posición de lectura de la segunda componente — que notamos mas arriba $|\sigma|$ — es calculada a partir del tipo de la primera componente, y representa el tamaño que ocupa un dato una vez codificado.



El tamaño $|\sigma|$ se define recursivamente sobre la estructura de σ.

$$
\begin{aligned}
|\mathtt{int}| &= L_{\mathtt{int}} & |\mathtt{bool}| &= L_{\mathtt{bool}} \\
|\sigma\times\tau| &= |\sigma| + |\tau| & |\sigma\to\tau| &= L_{\mathtt{ref}} \\
|t| &= L_{\mathtt{ref}} & |S.t| &= L_{\mathtt{ref}}
\end{aligned}
$$

Definimos la constante $L_{\mathtt{int}}$ como la longitud en octetos de un entero una vez codificado, y de manera análoga, constantes de longitud para los booléanos ($L_{\mathtt{bool}}$) y para los apuntadores ($L_{\mathtt{ref}}$). Por ejemplo, en una arquitectura de 32 bits, tendríamos $L_{\mathtt{int}} = 4$, $L_{\mathtt{bool}} = 1$, y $L_{\mathtt{ref}} = 8$.

3.3.3 Valores funcionales

La traducción de valores funcionales es una de las más delicadas. El objetivo de la traducción es de poder aplicar a distancia una función f utilisando la primitiva `remote_apply`. Como vimos en la introducción (ver figura 1), es necesario tranformar f en dos funciones de interface: f_{clien} para el cliente del llamado y f_{serv} para el servidor del llamado.

Notemos $\sigma \to \tau$ el tipo de f. El argumento a pasar a f_{serv} es un flujo a decodificar en un valor de tipo σ. Aplicando f a este valor, obtenemos un resultado de tipo τ, que debe a su vez ser codificado para su transmisión de regreso al llamador. De esta manera, obtenemos el código de f_{serv}:

$$f_{serv} = \lambda s.\, E_\tau(f(D_\sigma(s))).$$

Por otra parte, f_{serv} debe transmitirse al cliente bajo la forma de un apuntador codificado, lo cual resulta en la traducción siguiente.

$$
\begin{aligned}
& E_{\sigma\to\tau}(f) \\
&\quad = \mathtt{encode_ref}(\mathtt{newref}(\lambda s.\, E_\tau(f(D_\sigma(s))))) \\
& D_{\sigma\to\tau}(s) \\
&\quad = \lambda x.\, D_\tau(\mathtt{remote_apply}(\mathtt{decode_ref}(s), E_\sigma(x)))
\end{aligned}
$$

En nuestro caso, la decodificación se inicia decodificando el apuntador a la función f_{serv} (leído a través de la conexión), obteniéndose el apuntador a_{serv} que es pasado en argumento a `remote_apply`. La función f_{serv} recibe por intermedio de `remote_apply` un argumento de tipo flujo y regresa un resultado del mismo tipo. Así, de manera simétrica a la codificación, es necesario componer la aplicación remota de f_{serv} con la codificación de su argumento y la decodificación de su resultado. En otras palabras, el resultado de la decodificación es la función

$$f_{clien} = \lambda x.\, D_\tau(\mathtt{remote_apply}(a_{serv}, E_\sigma(x))).$$

Evaluación de expresiones:

$$P \vdash c \xrightarrow{S} c \qquad P \vdash \lambda x.\, e \xrightarrow{S} \lambda x.\, e \qquad \frac{P \vdash e_1 \xrightarrow{S} \lambda x.\, e \quad P \vdash e_2 \xrightarrow{S} v_2 \quad P \vdash e\{x \leftarrow v_2\} \xrightarrow{S} v}{P \vdash e_1(e_2) \xrightarrow{S} v}$$

$$\frac{P \vdash \mathrm{Def}(x, S', P) \xrightarrow{S'} \varphi}{P \vdash S'.x \xrightarrow{S} \varphi} \qquad \frac{P \vdash e_1 \xrightarrow{S} v_1 \quad P \vdash e_2 \xrightarrow{S} v_2}{P \vdash (e_1, e_2) \xrightarrow{S} (v_1, v_2)} \qquad \frac{P \vdash e \xrightarrow{S} (v_1, v_2)}{P \vdash \mathtt{fst}(e) \xrightarrow{S} v_1} \qquad \frac{P \vdash e \xrightarrow{S} (v_1, v_2)}{P \vdash \mathtt{snd}(e) \xrightarrow{S} v_2}$$

$$\frac{P \vdash e \xrightarrow{S} v}{P \vdash \mathtt{newref}(e) \xrightarrow{S} \mathtt{ref}_S(v)} \qquad \frac{P \vdash e \xrightarrow{S} \mathtt{ref}_S(v)}{P \vdash \mathtt{deref}(e) \xrightarrow{S} v} \qquad \frac{P \vdash e \xrightarrow{S} (\mathtt{ref}_T(v), \varphi) \quad P \vdash v(\varphi) \xrightarrow{T} \varphi'}{P \vdash \mathtt{remote_apply}(e) \xrightarrow{S} \varphi'}$$

$$\frac{P \vdash e \xrightarrow{S} v}{P \vdash \mathtt{encode_int}(e) \xrightarrow{S} \mathtt{concat}(\mathtt{encode_int}(v), \mathtt{empty})} \qquad \frac{P \vdash e \xrightarrow{S} \mathtt{concat}(\mathtt{encode_int}(v), \varphi)}{P \vdash \mathtt{decode_int}(e) \xrightarrow{S} v}$$

$$\frac{P \vdash e \xrightarrow{S} v}{P \vdash \mathtt{encode_bool}(e) \xrightarrow{S} \mathtt{concat}(\mathtt{encode_bool}(v), \mathtt{empty})} \qquad \frac{P \vdash e \xrightarrow{S} \mathtt{concat}(\mathtt{encode_bool}(v), \varphi)}{P \vdash \mathtt{decode_bool}(e) \xrightarrow{S} v}$$

$$\frac{P \vdash e \xrightarrow{S} v}{P \vdash \mathtt{encode_ref}(e) \xrightarrow{S} \mathtt{concat}(\mathtt{encode_ref}(v), \mathtt{empty})} \qquad \frac{P \vdash e \xrightarrow{S} \mathtt{concat}(\mathtt{encode_ref}(v), \varphi)}{P \vdash \mathtt{decode_ref}(e) \xrightarrow{S} v}$$

$$\frac{P \vdash e \xrightarrow{S} (\varphi_1, \varphi_2) \quad \varphi_1 + \varphi_2 = \varphi}{P \vdash \mathtt{concat}(e) \xrightarrow{S} \varphi} \qquad \frac{P \vdash e \xrightarrow{S} \varphi \quad C(\varphi, n) = \varphi'}{P \vdash \mathtt{head}_n(\varphi) \xrightarrow{S} \varphi'} \qquad \frac{P \vdash e \xrightarrow{S} \varphi \quad F(\varphi, n) = \varphi'}{P \vdash \mathtt{tail}_n(\varphi) \xrightarrow{S} \varphi'}$$

Funciones auxiliares sobre los flujos:

Concatenación de valores de flujo:

$$\begin{array}{rcl} \mathtt{empty} + \varphi & = & \varphi \\ \mathtt{concat}(v, \varphi) + \varphi' & = & \mathtt{concat}(v, (\varphi + \varphi')) \end{array}$$

Comienzo de un flujo:

$$\begin{array}{rcl} C(\varphi, 0) & = & \mathtt{empty} \\ C(\mathtt{concat}(\mathtt{encode_int}(v), \varphi), n + L_{\mathtt{int}}) & = & \mathtt{concat}(\mathtt{encode_int}(v), C(\varphi, n)) \\ C(\mathtt{concat}(\mathtt{encode_bool}(v), \varphi), n + L_{\mathtt{bool}}) & = & \mathtt{concat}(\mathtt{encode_bool}(v), C(\varphi, n)) \\ C(\mathtt{concat}(\mathtt{encode_ref}(v), \varphi), n + L_{\mathtt{ref}}) & = & \mathtt{concat}(\mathtt{encode_ref}(v), C(\varphi, n)) \end{array}$$

Final de un flujo:

$$\begin{array}{rcl} F(\varphi, 0) & = & \varphi \\ F(\mathtt{concat}(\mathtt{encode_int}(v), \varphi), n + L_{\mathtt{int}}) & = & F(\varphi, n) \\ F(\mathtt{concat}(\mathtt{encode_bool}(v), \varphi), n + L_{\mathtt{bool}}) & = & F(\varphi, n) \\ F(\mathtt{concat}(\mathtt{encode_ref}(v), \varphi), n + L_{\mathtt{ref}}) & = & F(\varphi, n) \end{array}$$

Figura 6: Reglas de reducción del lenguaje objeto (semántica de comunicación)

3.3.4 Tipos abstractos

Como lo mencionamos en la introdución, los valores de tipos abstractos no son transmitidos a través de la red; sólo son transmitidos apuntadores a ellos. Un apuntador es creado y accesado únicamente de manera local al procesador de su definición. Hay dos casos. Si una expresion e es de tipo t, el valor de e es local al módulo de definición de t y puede solamente exportarse bajo la forma de un apuntador creado localmente y codificado para su transmisión. Si e es de tipo $S.t$, su representación es la de un apuntador que debe tratarse de manera abstracta. En particular, no es posible accesarlo, sólo propagarlo a través de la red.

$$\begin{aligned} E_t(e) &= \texttt{encode_ref}(\texttt{newref}(e)) \\ D_t(s) &= \texttt{deref}(\texttt{decode_ref}(s)) \\ E_{S.t}(e) &= \texttt{encode_ref}(e) \\ D_{S.t}(s) &= \texttt{decode_ref}(s) \end{aligned}$$

3.4 Inserción del código de interfaces

Todos los valores exportados por un módulo son susceptibles de transmisión, y en consecuencia, deben ser traducidos en flujos. Por ende, es necesario introducir código de interface para realizar la tradución, por un lado, al momento de la définición de los valores (codificación del valor en un flujo), y por otro, al momento de su utilisación (decodificación de un flujo en un valor).

3.4.1 Traducción de expresiones

La traducción de una expresión e de un programa P, que notamos $T_P(e)$, consiste en el remplazo de todos los accesos a módulos $S.x$ por $D_\tau(S.x)$, donde τ es el tipo de declaración de x en el programa P. De esta manera, introducimos el código de interface correspondiente al cliente de una comunicación.

$$\begin{aligned} T_P(x) &= x \\ T_P(c) &= c \\ T_P(\lambda x.\, e) &= \lambda x.\, T_P(e) \\ T_P(e_1(e_2)) &= T_P(e_1)(T_P(e_2)) \\ T_P(S.x) &= D_{\mathrm{Decl}(x,S,P)}(S.x) \end{aligned}$$

3.4.2 Traducción de módulos

La traducción de un módulo necesita la codificación de todos los valores que éste exporta. Una declaración $\texttt{val}\ x : \tau = e$ debe transformase en

$$\texttt{val}\ x : \texttt{stream} = E_\tau(e)$$

lo que corresponde a la introducción del código de interface del servidor de la comunicación. Adicionalmente, es necesario considerar el caso de un módulo que declara y hace referencia a x localmente, y cuyo valor no debe ser codificado. Es por ello que el codaje de x comprendre un paso suplementario de definición una nueva variable x', que contiene el valor de la expresión e de declaración de x sin codificación alguna.

$$\begin{aligned} & T_P(\varepsilon) = \varepsilon \\ & T_P(\texttt{type}\ t = \tau;\ M) = \\ & \qquad \texttt{type}\ t = \tau;\ T_P(M) \\ & T_P(\texttt{val}\ x : \tau = e;\ M) = \\ & \qquad \texttt{val}\ x' : \tau = T_P(e); \\ & \qquad \texttt{val}\ x : \texttt{stream} = E_\tau(x'); \\ & \qquad T_P(M\{x \leftarrow x'\}) \end{aligned}$$

En la última igualdad, el identificador x' es escogido distinto de todo identificador presente o generado por la traducción en el módulo.

3.4.3 Traducción de programas

Para finalizar, la tradución $T(P)$ de un programa completo $P = (\texttt{let}\ S_1 = [M_1];\ \ldots;\ S_n = [M_n]\ \texttt{in}\ e)$, es definida por:

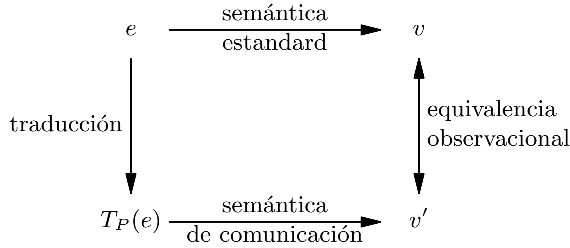
$$\texttt{let}\ S_1 = [T_P(M_1)];\ \ldots;\ S_n = [T_P(M_n)]\ \texttt{in}\ T_P(e)$$

4 Prueba de equivalencia

En esta parte mostramos la equivalencia para los fines de la comunicación entre la reducción de un programa y la reducción de su traducción. En particular, establecemos la equivalencia de comportamiento entre el resultado de un llamado calculado localmente y el resultado de ese llamado calculado a distancia y utilisando el código de interface de nuestra traducción. Partiendo de este resultado obtenemos que el mecanismo de RPC descrito es transparente con respecto al comportamiento de los programas. La proposición siguiente expresa formalmente ese resultado:

Proposición 1 *Sea P un programa tipable ($\vdash P$ `ok`) y v un valor entero. Si $\vdash P \xrightarrow{*} v$ usando la semántica estandard, entonces $\vdash T(P) \xrightarrow{*} v$ con el mismo valor v y usando la semántica de comunicación.*

La demostración de esta proposición se hace por inducción sobre las etapas de la reducción durante la evaluación. En la prueba utilisamos un resultado análogo para el caso de las expresiones. Se trata de un resultado de conmutación entre la evaluación y la traducción de una expresión. Concretamente, dada una expresión e, es posible evaluarla con la semántica estandard, o de traducir e primero, evaluándola luego con la semántica de comunicación. Mostraremos que ambos métodos conducen a resultados que sin ser iguales, son equivalentes observacionalmente.



Más adelante definiremos la relación de equivalencia observacional, que notamos $\approx$. Ella corresponde a la igualdad en el caso de valores enteros y booléanos. En el caso de valores funcionales, la equivalencia no se reduce a la igualdad sintáctica, puesto que dos funciones sintácticamente diferentes pueden comportarse igualmente. Diremos que dos funciones son equivalentes si envían argumentos equivalentes hacia resultados equivalentes. A continuación un ejemplo de este caso.

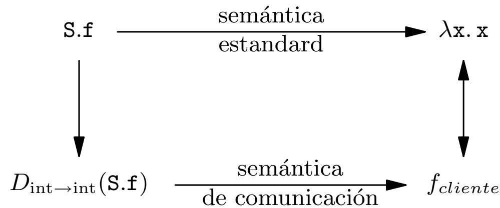
Ejemplo: Consideremos el programa P siguiente:

```
let S = [ val f : int → int = λx.x ]
in S.f
```

Su traducción $T(P)$ es:

```
let S =
  [ val f' : int→int = λx.x;
    val f : stream =
      encode_ref(newref(
        λs. encode_int(decode_int(s)))) ]
in λx. decode_int(
         remote_apply(decode_ref(S.f),
                      encode_int(x)))
```

Consideremos la evaluación de la expresión `S.f`.



En este diagrama, notamos $f_{cliente}$ la función

```
λx. decode_int(remote_apply
         (decode_ref(S.f), encode_int(x)))
```

La función $f_{cliente}$ no es idéntica a $\lambda\texttt{x}.\,\texttt{x}$. Sin embargo, es fácil ver que para todo valor entero i y toda máquina R, se cumple

$$P \vdash (\lambda\texttt{x}.\,\texttt{x})(i) \xrightarrow{*} i \text{ y } T(P) \vdash f_{cliente}(i) \xrightarrow{R} i.$$

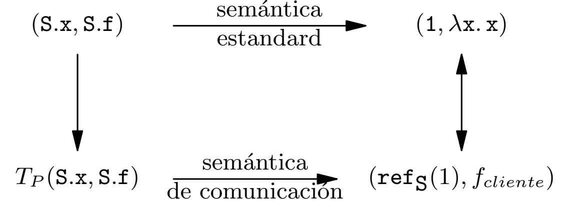
Así, estas dos funciones son equivalentes observacionalmente. □

El caso de los valores de tipos abstractos presenta un problema análogo: la representación de un valor abstracto visto desde el procesador de su definición es diferente de su representación para los otros procesadores. Los procesadores externos lo ven como un apuntador.

Ejemplo: Consideremos el programa P siguiente:

```
let S = [ type t = int;
          val x : t = 1
          val f : t→t = λx.x ]
in (S.x, S.f)
```

La expresión `(S.x, S.f)` se evalúa de la manera siguiente:



En la semántica de comunicación, la primera componente del resultado es un apuntador al entero 1, y no el entero 1 directamente. La segunda componente es una función $f_{cliente}$ cuyo tipo va de los apuntadores en los apuntadores, y no de los enteros en los enteros. No obstante, la función $f_{cliente}$ es equivalente a $\lambda\texttt{x}.\,\texttt{x}$ en el sentido que si $\lambda\texttt{x}.\,\texttt{x}$ envía un entero i en un entero j, entonces $f_{cliente}$ envía un apuntador a i en un apuntador a j. □

Definimos ahora la relación de equivalencia observacional entre dos valores v y v' observados con respecto a un tipo τ y desde una máquina S, que notamos $P, S \models v \approx v' : \tau$. v es un valor obtenido por reducción a partir de una expresión del lenguaje fuente usando la semántica estandard. El valor v' es obtenido por reducción de una expresión del lenguaje objeto utilisando la semántica de comunicación. Esta relación de equivalencia es la más pequeña que satisface las condiciones siguientes:

- $P, S \models i \approx i : \texttt{int}$ para toda constante entera i
- $P, S \models \texttt{false} \approx \texttt{false} : \texttt{bool}$ y $P, S \models \texttt{true} \approx \texttt{true} : \texttt{bool}$.
- $P, S \models (v_1, v_2) \approx (v_1', v_2') : \tau_1 \times \tau_2$ si y solo si $P, S \models v_1 \approx v_1' : \tau_1$ y $P, S \models v_2 \approx v_2' : \tau_2$.
- $P, S \models \lambda x.\, e \approx \lambda x'.\, e' : \sigma \to \tau$ si y solo si para todos los valores v, v', r tales que $P, S \models v \approx v' : \sigma$ y $P \vdash (\lambda x.\, e)(v) \xrightarrow{*} r$, existe un valor r' tal que $T(P) \vdash (\lambda x'.\, e')(v') \xrightarrow{S} r'$ y $P, S \models r \approx r' : \tau$.
- $P, S \models v \approx v' : t$ si y solo si $P, S \models v \approx v' : \tau$, donde τ es el tipo de definición de t. Es decir, P es de la forma $P = (\texttt{let} \ \ldots S = [M_1;\ \texttt{type}\ t = \tau;\ M_2] \ldots)$.
- $P, S \models v \approx \texttt{ref}_R(v') : R.t$ si y solo si $P, R \models v \approx v' : \sigma$, donde σ es definido como

  $$\sigma = \tau\{t \leftarrow R.t \mid t \text{ tipo definido en } M_1\}$$

  y τ como

  $$P = (\texttt{let} \ \ldots R = [M_1;\ \texttt{type}\ t = \tau;\ M_2] \ldots).$$

Podemos ahora enunciar formalmente el resultado de conmutación entre traducción y evaluación.

Proposición 2 *Sea e una expresion sin variables libres y S una máquina. Si* $\emptyset \vdash e : \tau$ *y* $P \vdash e \xrightarrow{*} v$ *con la semántica estandard, entonces existe un valor* v' *tal que* $T(P) \vdash T_P(e) \xrightarrow{S} v'$ *con la semántica de comunicación, y además* $P, S \models v \approx v' : \tau$.

Esta proposición implica la proposición 1 puesto que la relación $\approx$ es la igualdad en los enteros.

Omitimos la prueba completa de la proposición 2. El lema principal establece la propiedad de anulación entre las operaciones de codaje y de decodaje.

Proposición 3 *Sea* `val` $x : \tau = e$ *la definición de un valor del módulo R en el programa P. Notemos* $\sigma = \mathrm{Decl}(x, R, P)$. *Para todos los valores* v, v' *tales que* $P, R \models v \approx v' : \tau$ *y todos los módulos S, obtenemos*

$$P, S \models v \approx D_\sigma(E_\tau(v')) : \sigma.$$

Utilisamos este lema en el caso delicado de la prueba de la proposición 2: la reducción de un acceso $R.x$ a un identificador distante. Mostramos el esquema de la demostración. Supongamos P de la forma

$$\texttt{let} \;\ldots;\; R = [M_1;\; \texttt{val}\; x : \tau = e;\; M_2];\; \ldots$$

y notemos $\sigma = \mathrm{Decl}(x, R, P)$ y $e = \mathrm{Def}(x, R, P)$. Siguiendo las reglas de reducción del lenguaje fuente, $\vdash R.x \xrightarrow{*} v$ si y solo si $P \vdash e' \xrightarrow{*} v$. Aplicando inductivamente la proposición 2, obtenemos un valor v' tal que

$$T(P) \vdash T_P(e') \xrightarrow{R} v' \text{ y } P, R \models v \approx v' : \tau.$$

Aplicando una etapa de codificación, obtenemos

$$T(P) \vdash E_\tau(e') \xrightarrow{R} E_\tau(v').$$

La definición de $R.x$ en $T(P)$ es $E_\tau(x')$ donde x' es igual a $T_P(e)$. A partir de ésto y sabiendo que la substitución conmuta con la traducción, obtenemos

$$\mathrm{Def}(x, R, T(P))) = E_\tau(e').$$

Combinando los dos últimos resultados, obtenemos

$$T(P) \vdash \mathrm{Def}(x, R, T(P))) \xrightarrow{R} E_\tau(v').$$

Es fácil ver que $E_\tau(v')$ es un valor de flujo, y entonces podemos aplicar la regla de reducción de la semántica de comunicación, obteniendo

$$T(P) \vdash R.x \xrightarrow{S} E_\tau(v').$$

Por definición de la traducción, $T_P(R.x) = D_\sigma(R.x)$. Aplicando una etapa de decodificación, obtenemos

$$T(P) \vdash T_P(R.x) \xrightarrow{S} D_\sigma(E_\tau(v')).$$

Sea $v'' = D_\sigma(E_\tau(v'))$. Hemos demostrado que $T(P) \vdash T_P(R.x) \xrightarrow{S} v''$, y por la proposición 3, deducimos $P, S \models v \approx D_\sigma(E_\tau(v')) : \sigma$. Y este es el resultado esperado.

5 Conclusión

En este artículo establecemos la relación entre el problema de la parametricidad bien estudiada teóricamente y una técnica de transmisión de datos en los sistemas distribuídos importante en la práctica. Este trabajo puede verse como una aplicación de estos trabajos teóricos o como una justificación formal de esta práctica.

Un caso similar al estudiado en este artículo se presenta en los sistemas de tipos con polimorfismo paramétrico. Por ejemplo, un llamado distante a una función generica de inversión de listas que tiene el tipo polimorfo: $\forall t.\, \texttt{list}(t) \to \texttt{list}(t)$, no necesita transmitir los elementos de la lista sino solamente la lista de los apuntadores a los elementos. Nuestro formalismo se extiende facilmente a este caso de la parametricidad que es simétrico al caso de tipos abstractos.

Bibliografía

[1] Martín Abadi, Luca Cardelli, y Pierre-Louis Curien. Formal parametric polymorphism. *Theoretical Computer Science*, 121(1):9–58, 1993.

[2] Henri E. Bal, Jennifer G. Steiner, y Andrew S. Tanenbaum. Programming languages for distributed computing systems. *Computing Surveys*, 21(3):261–322, 1989.

[3] Brian Bershad, Thomas Anderson, Edward Lazowska, y Henry Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, 1990.

[4] Andrew Birrell, Greg Nelson, Susan Owicki, y Edward Wobbler. Network objects. Reporte de investigación 115, DEC Systems Research Center, 1994.

[5] Andrew D. Birrell y Bruce J. Nelson. Implementing remote procedure call. *ACM Transactions on Computer Systems*, 2(1):39–59, 1984.

[6] Luca Cardelli y Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing surveys*, 17(4):471–522, 1985.

[7] Maurice Herlihy y Barbara Liskov. A value transmission method for abstract data types. *ACM Transactions on Programming Languages and Systems*, 4(4):527–551, 1982.

[8] Eric Jul, Henry Levy, Norman Hutchinson, y Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, 1988.

[9] Barbara Liskov y John Guttag. *Abstraction and specification in program development.* MIT Press, 1986.

[10] John C. Mitchell. Type systems for programming languages. En Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, volume B*, páginas 367–458. The MIT Press/Elsevier, 1990.

[11] John C. Mitchell. On the equivalence of data representations. En V. Lifschitz, editor, *Artificial intelligence and mathematical theory of computation*, páginas 305–330. Academic Press, 1991.

[12] Bruce J. Nelson. Remote procedure call. Reporte técnico CSL-81-9, Xerox PARC, 1981.

[13] Atsushi Ohori y Kazuhiko Kato. Semantics for communication primitives in a polymorphic language. En *20th symposium Principles of Programming Languages*, páginas 99–112. ACM Press, 1993.

[14] Benjamin C. Pierce y David N. Turner. Object-oriented programming without recursive types. En *20th symposium Principles of Programming Languages*, páginas 299–312. ACM Press, 1993.

[15] John C. Reynolds. Toward a theory of type structure. En *Programming Symposium, Paris, 1974*, volumen 19 de *Lecture Notes in Computer Science*, páginas 408–425. Springer-Verlag, 1974.

[16] John C. Reynolds. Types, abstraction and parametric polymorphism. En *Information Processing '83*, páginas 513–523. North-Holland, 1983.

[17] Michael Schroeder y Michael Burrows. Performance of Firefly RPC. *ACM Transactions on Computer Systems*, 8(1):1–17, 1990.

[18] Sun Microsystems, Inc. RPC: Remote procedure call protocol specification: version 2. Request for comment 1057, Network Information Center, 1988.

[19] Sun Microsystems, Inc. `rpcgen` programming guide. En *Sun OS Network Programming Manual*. Sun Microsystems, Inc., 1988.

[20] Sun Microsystems, Inc. NFS: Network file system protocol specification. Request for comment 1094, Network Information Center, 1989.

[21] Philip Wadler. Theorems for free! En *Functional Programming Languages and Computer Architecture 1989*. ACM Press, 1989.

[22] Niklaus Wirth. *Programming in Modula-2*. Springer-Verlag, 1983.