UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

UACPyP DEL CCH - IIMAS

LOGICA TEMPORAL Y COMPUTACION

T E S I S

QUE PARA OBTENER EL GRADO DE:

MAESTRO EN CIENCIAS DE LA COMPUTACION

P R E S E N T A :

MIGUEL CARRILLO BARAJAS

MEXICO, D. F. FEBRERO DE 1993

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

I N D I C E

INTRODUCCION

0.1 Antecedentes

La lógica ha tenido un papel importante en el nacimiento y desarrollo de la computación. Como ejemplo basta mencionar que en el área de los lenguajes de programación la lógica ha resultado ser una herramienta muy útil y ha llegado a conformar una tendencia denominada programación lógica.

La programación lógica y la programación funcional se ubican dentro de la corriente de la programación declarativa que surge como alternativa diferente de la programación imperativa. La programación lógica tiene como paradigma usar un lenguaje lógico para programar. El éxito mas conocido de este enfoque es el lenguaje Prolog del cual existen versiones comerciales. Prolog permite escribir programas usando un lenguaje basado en la lógica tradicional. Es por esto que su capacidad expresiva queda limitada, en cierta forma, por el poder expresivo de la lógica tradicional.

El poder expresivo de la lógica tradicional puede ser enriquecido agregándole operadores que permitan describir hechos o conceptos que dependan del tiempo tales como *siempre*, *eventualmente (a-veces)* y *después*. El resultado de esto se conoce como lógica temporal.

Recientemente han surgido aplicaciones de la lógica temporal en diversas áreas de computación. Un tema que ha recibido atención especial en este sentido es la aplicación de la lógica temporal en la especificación y verificación de programas (principalmente programas concurrentes). Las ventajas de la lógica temporal en esta tarea son grandes debido a que el concepto de programa (si se entiende como una *sucesión de instrucciones*) contiene aspectos temporales. De tal forma que usando lógica temporal es

Proyecto Académico:

Maestría en Ciencias de la Computación.
UACPyP del CCH, UNAM.

Alumno: Miguel Carrillo Barajas (7951441-1, expediente 048356).

Título de la tesis: Lógica Temporal y Computación.

En esta tesis se analiza el lenguaje de programación Tempura como un producto de la interacción de la lógica matemática temporal con la computación. Se describe la sintaxis y la semántica de la lógica temporal de intervalos y se propone un sistema axiomático para la misma. Por otro lado, se presenta la sintaxis, la semántica, el uso y la implementación de Tempura. Esto incluye la propuesta de una semántica formal para las instrucciones que definen predicados y funciones y para las instrucciones de entrada y salida de datos. También se analizan algunas de las características de Tempura comparándolo con otros lenguajes de programación y se presenta una semántica denotacional para este lenguaje. Se concluye con algunas observaciones acerca del trabajo realizado y se mencionan opciones para continuar el mismo. Como parte experimental de la tesis se reporta la programación de un intérprete de Tempura ejecutable en computadoras personales.

Vo.Bo.

Dr. Felipe Bracho
Director de tesis.

relativamente fácil expresar, por ejemplo, que un programa *eventualmente* terminará o que *siempre* producirá un resultado de características determinadas. El lado negativo de este punto de vista es que en la tarea de programación se requiere del conocimiento de no de uno sino de dos formalismos: el lenguaje que se utiliza en la programación y el lenguaje lógico-temporal que se usa en el estudio de las propiedades de un programa. Es claro que el manejo de estos dos formalismos establece una brecha incómoda entre *programas* y *verificaciones* (o *especificaciones*) y una forma de evitar esto es haciendo que la lógica temporal interprete los dos papeles: el de formalismo de especificación o verificación y el de lenguaje de programación. Tal es el caso del lenguaje Tempura diseñado por B. Moszkowski.

0.2 Organización y Presentación

En este trabajo se describe el lenguaje Tempura como un producto de la interacción de la lógica temporal con la computación. Esta descripción tiene algunas características propias en cuanto a forma y contenido. En lo que se refiere a la forma, hemos detallado la relación entre Tempura y la lógica temporal resaltando sus semejanzas y diferencias. Hemos preferido dar las definiciones principales en forma total (y no gradualmente) con la intención de que las ideas iniciales sean globales. Además, el paso de la lógica temporal a Tempura se ha suavizado resaltando la existencia de una teoría temporal que constituye un puente entre ambos. Respecto al contenido, hemos considerado importante incluir el tema de la axiomatización de la lógica temporal de intervalos tocando los puntos de correctez y de incompletez esencial. En la definición de la semántica formal de Tempura hemos considerado el caso de las instrucciones que definen predicados y funciones y el caso de las instrucciones de entrada y salida de datos. Y como parte de la descripción de la implementación hemos incluido una sección con la semántica de Tempura al estilo de la semántica denotacional. Cabe mencionar

que existen algunas diferencias menores entre el tratamiento que aquí se da a algunas fórmulas (como es el caso de los condicionales if e then w) y el tratamiento que les da B. Moszkowski.

La presentación se divide principalmente en tres partes: *Lógica Temporal de Intervalos*, *Teoría Temporal de Intervalos* y *Tempura*.

La introducción a la parte teórica se encuentra en el capítulo 1 que presenta la sintaxis y la semántica de una lógica temporal con tres operadores de tiempo: *siempre, después* y *corta* que se denotan con los símbolos □, O y ; respectivamente. Este capítulo concluye con algunos resultados sobre la incompletez de la lógica temporal y con la propuesta de un sistema axiomático.

El capítulo 2 contiene la definición de una teoría temporal sobre un dominio que abarca a los números racionales, valores de verdad, cadenas de caracteres y listas finitas. Esta teoría sirve de conexión entre la lógica temporal de intervalos y Tempura.

La parte de aplicación se encuentra en el capítulo 3. Aquí se presenta a Tempura como un lenguaje de programación imperativo que incorpora una parte de la teoría temporal ya mencionada. La presentación incluye la sintaxis, la semántica, el uso y la implementación de Tempura. En el aspecto experimental, reportamos la realización de un intérprete que ha servido para investigar, aclarar y concretar algunas de las ideas que se exponen aquí. Finalmente, se analizan algunas de las características de Tempura comparándolo con lenguajes de programación imperativa, lógica y funcional.

El capítulo 4 concluye con algunas observaciones acerca de posibles extensiones de Tempura y el apéndice A contiene un resumen de la sintaxis y la semántica de este lenguaje.

El marco de exposición que empleamos es el de la teoría intuitiva de conjuntos. Usamos indistintamente los términos *conjunto* y *clase* para referirnos a un grupo de objetos y con {x/P(x)} nos referimos a la clase de los objetos que tienen la propiedad P.

Recurrimos a varios símbolos de uso común en matemáticas: $\forall$ (para todo), $\exists$ (existe), $\supset$ (implica), $\wedge$ (y), $\vee$ (o), $\neg$ (no), sii (si y sólo si), $\in$ (pertenece), $\subseteq$ (está contenido), Φ (conjunto vacío), $\cap$ (intersección), $\cup$ (unión), $\mathbb{N}$ (números naturales), $\mathbb{Q}$ (números racionales), $\mathbb{B}$ (valores de verdad), etc. A varios de estos símbolos se les asignará, aparte de su significado intuitivo, un significado formal por medio de ciertas definiciones. En tal caso, el significado (intuitivo o formal) de estos símbolos podrá determinarse por el contexto en que se usen.

Usamos ampliamente los mecanismos matemáticos de inducción y recursión en algunas definiciones y demostraciones. Las definiciones se resaltan con letra *script*.

Febrero de 1993.

LOGICA TEMPORAL DE INTERVALOS

En este capítulo se describen la sintaxis, la semántica y la axiomatización de un formalismo llamado Lógica Temporal de Intervalos (LTI). Más adelante usaremos esta lógica para definir una Teoría Temporal de Intervalos (TTI) sobre un dominio específico. El interés por esta teoría radica en que proporciona un marco formal al lenguaje de programación Tempura.

1.1 Introducción

La lógica modal es una extensión de la lógica clásica que resulta de agregar los operadores modales de necesidad y de posibilidad ($\Box$ y $\Diamond$ respectivamente). La lógica temporal surge como una variante de la lógica modal agregando el operador O (*después*) e interpretando $\Box$ y $\Diamond$ como los operadores de tiempo *siempre* y *a-veces*. En este trabajo nos ocuparemos solamente de la lógica temporal de intervalos sin entrar en más detalles acerca de la lógica clásica[1], la lógica modal[2] y la lógica temporal[3].

La lógica temporal de intervalos es un formalismo que resulta al considerar una variación semántica en la lógica temporal. La semántica de la lógica temporal se define mediante una función que en cada estado (de tiempo) le asigna un valor de verdad a cada fórmula. La LTI considera intervalos (sucesiones de estados) en lugar de estados y su semántica se define mediante una función que le asigna un valor de verdad a cada fórmula en cada intervalo. Además de esta variación semántica, la LTI tiene el operador de composición secuencial *corta*, denotado con ';', que permite expresar que dos fórmulas son respectivamente ciertas en dos subintervalos de tiempo que se traslapan de tal forma que el

---

[1] [Ebbinghaus-Flum-Thomas 84].

[2] [Hughes-Cresswell 73].

[3] [Rescher-Urquhart 71].

estado final del primero es idéntico al estado inicial del segundo.

La lógica clásica, el cálculo de predicados de primer orden, es un lenguaje apropiado para expresar algunas propiedades. Por ejemplo, para expresar 'X es igual a 1 y Y es igual a 2' podemos emplear la fórmula

$$(X = 1) \wedge (Y = 2),$$

sin embargo, es necesario notar que la expresión de este ejemplo es atemporal, es decir, no indica en qué momento suceden los hechos que afirma. Si queremos expresar 'X es igual a 1 en un instante y X es igual a 2 en un instante posterior' sería necesario emplear una fórmula parecida a

$$\exists\, t,t'(t \leq t' \ \wedge \ X(t) = 1 \ \wedge \ X(t') = 2),$$

Las variables de tiempo, t y t', sirven para indicar que el valor de X depende de ellas. Sin embargo, hay varios inconvenientes en el uso de variables de tiempo. Por ejemplo, se debe indicar siempre qué relación existe entre ellas, de qué manera están cuantificadas y qué variables dependen de ellas. En el peor de los casos, el uso de variables de tiempo, además de resultar complicado o tedioso con fórmulas grandes, puede reducir la claridad de lo que se desea expresar y entorpecer el desarrollo del conocimiento del objeto de estudio.

Una alternativa al uso de variables de tiempo es la lógica temporal de intervalos. Como una muestra del poder expresivo de la LTI, damos algunos ejemplos de cómo expresar hechos temporales. Los ejemplos incluyen el uso de los operadores $\Box$ (siempre), $\Diamond$ (a-veces, eventualmente), O (después) y ; (corta).

a) 'X es igual a 1 inicialmente y X es igual a 2 en el siguiente instante'

$$(X = 1) \wedge \bigcirc (X = 2)$$

b) 'X es igual a 1 inicialmente y el valor de X se duplica en cada instante siguiente':

$$(X = 1) \wedge \Box (\bigcirc X = 2*X)$$

c) 'Si X siempre es igual a 3 y eventualmente Y será igual a 3 entonces eventualmente X-Y será igual a cero':

$$((\Box X=3) \wedge (\Diamond Y=3)) \supset \Diamond (X-Y=0)$$

d) 'Inicialmente X = 1 y Y = 2, en el siguiente instante X = Y+1 y Y = X+2 ; tomando los últimos valores de X y de Y, ahora Z = X+Y y en el siguiente instante Z = Z-1 ':

$$(X=1 \wedge Y=2 \wedge \bigcirc X=Y+1 \wedge \bigcirc Y=X+2) \; ; \; (Z=X+Y \wedge \bigcirc Z=Z-1)$$ [1]

Hasta aquí, se han descrito en forma intuitiva los conceptos básicos de la Lógica Temporal de Intervalos. Las secciones siguientes formalizan estas ideas.

---

[1] El significado exacto de A;B se dará más adelante.

1.2 Sintaxis

El *alfabeto* de la lógica temporal de intervalos (LTI) está constituido por los siguientes símbolos:

- Los símbolos de constante global c_i donde $i \in \mathbb{N}$.
- El símbolo de la constante local *empty*.
- Los símbolos de variable local[1] v_i donde $i \in \mathbb{N}$.
- Los símbolos de variable global u_i donde $i \in \mathbb{N}$.
- Los símbolos de función f_i^j donde $i, j \in \mathbb{N}$.
- Los símbolos de predicado p_i^j donde $i, j \in \mathbb{N}$.
- Los símbolos $\neg \wedge \forall = \Box \bigcirc ;$ *if then else*
- Los símbolos ( ) , :

Algunas veces nos referiremos a los símbolos omitiendo la frase "símbolo de", por ejemplo, diremos "constante" en lugar de "símbolo de constante".

Si f_i^j es un símbolo de función y p_i^j es un símbolo de predicado, decimos que su *aridad* es j.

De los símbolos de la LTI distinguimos las siguientes clases:

OBG	$= \{f_i^j\} \cup \{p_i^j\} \cup \{u_i\} \cup \{c_i\}$	*Objetos Globales*
OBL	$= \{v_i\} \cup \{empty\}$	*Objetos Locales*
OPT	$= \{\Box, \bigcirc, ;\}$	*Operadores Temporales*
VARL	$= \{v_i\}$	*Variables Locales*
VARG	$= \{u_i\}$	*Variables Globales*

Definiremos la sintaxis de la Lógica Temporal de Intervalos mediante dos categorías sintácticas: expresiones y fórmulas. Las expresiones denotan objetos del dominio de discurso y las fórmulas servirán para establecer predicados acerca de éstos y

---

[1] [Moszkowski 86] se refiere a las variables locales como *state variables* y a las globales como *static variables*.

denotan un valor de verdad. Seguimos un enfoque clásico donde las fórmulas tendrán en cada modelo un valor de *falso* o *verdadero* que denotamos con *false* y *true* respectivamente.

1.2.1 Expresiones

Definimos la clase de *variables*, VAR, como la clase que tiene como elementos los símbolos de variable es decir,

$$VAR = \{v_i / \ i \in \mathbb{N}\} \cup \{u_i / \ i \in \mathbb{N}\} = VARL \cup VARG.$$

La clase de *expresiones* de la LTI, EXP, es la clase más pequeña que cumple:

a) $\{c_1\} \cup VAR \subseteq EXP$.

b) Si $e_1, e_2, \ldots, e_k \in EXP$ ($k \geq 0$) y f es un símbolo de función de aridad k entonces $f(e_1, e_2, \ldots, e_k) \in EXP$[2].

c) *empty* $\in EXP$.

d) Si $e \in EXP$ entonces $(O\ e) \in EXP$.

e) Si $e_1, e_2, e_3 \in EXP$, $(\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3) \in EXP$.

La clase de *expresiones clásicas* de la LTI, EXPC, es la clase más pequeña que cumple:

a) $\{c_1\} \cup VARG \subseteq EXPC$.

b) Si $e_1, e_2, \ldots, e_k \in EXPC$ ($k \geq 0$) y f es un símbolo de función de aridad k entonces $f(e_1, e_2, \ldots, e_k) \in EXPC$.

Claramente, $EXPC \subseteq EXP$.

Si c es una constante, v una variable, f un símbolo de función y las e_i son expresiones, la *lectura de expresiones* se hace de la siguiente manera:

---

[2] Los símbolos de función de aridad cero también se consideran símbolos de constante.

Expresión	Lectura
- *empty*, c, v	vacío, c, v.
- $f(e_1,e_2,\ldots,e_k)$	f de $e_1,e_2,\ldots,e_k$.
- $(O\ e_1)$	el valor próximo de e_1.
- $(if\ e_1\ then\ e_2\ else\ e_3)$	si e_1 entonces e_2 si no e_3.

1.2.2 Fórmulas

La clase de *fórmulas* de la LTI, FORM, es la clase más pequeña que cumple:

a) Si w ∈ FORM entonces (¬ w) ∈ FORM.
b) Si r,s ∈ FORM entonces (r ∧ s) ∈ FORM.
c) Si $e_1,e_2,\ldots,e_k \in$ EXP (k≥0) y p es un símbolo de predicado de aridad k entonces $p(e_1,e_2,\ldots,e_k) \in$ FORM.[3]

d) Si w ∈ FORM y v ∈ VAR entonces (∀ v: w) ∈ FORM.
e) Si e_2, $e_2 \in$ EXP entonces $(e_1 = e_2) \in$ FORM.

f) *empty* ∈ FORM.
g) Si w ∈ FORM entonces (□ w) ∈ FORM.
h) Si w ∈ FORM entonces (O w) ∈ FORM.

i) Si r,s ∈ FORM entonces (r ; s) ∈ FORM.
j) Si e ∈ EXP y $w_1,w_2 \in$ FORM, entonces
(if e *then* w_1 *else* w_2) ∈ FORM

La clase de *fórmulas clásicas* de la LTI, FORMC, es la clase más pequeña que cumple:

a) Si w ∈ FORMC entonces (¬ w) ∈ FORMC.
b) Si r,s ∈ FORMC entonces (r ∧ s) ∈ FORMC.
c) Si $e_1,e_2,\ldots,e_k \in$ EXPC (k≥0) y p es un símbolo de predicado de aridad k entonces $p(e_1,e_2,\ldots,e_k) \in$ FORMC.
d) Si w ∈ FORMC y v ∈ VARG entonces (∀ v: w) ∈ FORMC.
e) Si e_2, $e_2 \in$ EXPC entonces $(e_1 = e_2) \in$ FORMC.

---

[3] Si la aridad de p es cero, p se denomina variable proposicional.

Claramente, FORMC ⊆ FORM .

Si w, w_1, w_2, r, s son fórmulas, v es una variable y las e_i son expresiones la *lectura de fórmulas* se define como sigue:

Fórmula	Lectura
- *empty*	vacío.
- (¬ w)	no w.
- (r ∧ s)	r y s.
- p(e_1,e_2,...,e_k)	p de e_1,e_2,...,e_k.
- (□ w)	Siempre w.
- (○ w)	Después w.
- (r ; s)	r corta s[4].
- (∀ v: w)	Para toda v: w.
- $e_1 = e_2$	e_1 igual a e_2
- (*if* e *then* w_1 *else* w_2)	Si e entonces w_1, si no w_2

Es importante observar que el operador ○ y el símbolo *empty* aparecen tanto en expresiones como en fórmulas. Distinguiremos su uso de acuerdo al contexto en que aparezcan.

1.2.3 Abreviaciones

En lugar de incluir todos los conectivos y operadores dentro de la definición básica, consideraremos que algunos son abreviaciones de otros.

Si $w, w_1, w_2, w_3 \in$ FORM, e,d ∈ EXP y v ∈ VAR, entonces:

- $w_1 \vee w_2$	$\stackrel{=}{}_{def} \neg (\neg w_1 \wedge \neg w_2)$.
- $w_1 \supset w_2$	$\stackrel{=}{}_{def} \neg w_1 \vee w_2$.
- $w_1 \equiv w_2$	$\stackrel{=}{}_{def} (w_1 \supset w_2) \wedge (w_2 \supset w_1)$.
- ∃ v: w	$\stackrel{=}{}_{def} \neg \forall$ v: ¬ w
- ∃ v_1,v_2,...,v_n: w	$\stackrel{=}{}_{def} \exists v_1 : (\exists v_2 : (\ldots (\exists v_n :$ w)...))

[4] Más informalmente, 's después que r'. [Moszkowski 86] se refiere a ';' como el operador *chop*.

- *more*	$\equiv_{def}$ ¬ *empty* .
- *true*	$\equiv_{def}$ *empty* ∨ ¬ *empty* .
- *false*	$\equiv_{def}$ ¬ *true* .
- ◇ w	$\equiv_{def}$ ¬ □ ¬ w .
- *if* e *then* w	$\equiv_{def}$ *if* e *then* w *else true*.
- *halt* e	$\equiv_{def}$ *if* e *then empty else more* .
- ⊖ w	$\equiv_{def}$ *empty* ∨ ○ w .
- *skip*	$\equiv_{def}$ ○ *empty* .
- r U s	$\equiv_{def}$ ((□ r) ; (○ s)).[5]
- r wU s	$\equiv_{def}$ ((□ r) ; (○ s)) ∨ □ r.[6]

La *lectura* de estas abreviaciones se hace como sigue:

Fórmula	Lectura
- $w_1 \vee w_2$	w_1 o w_2
- $w_1 \supset w_2$	w_1 implica w_2
- $w_1 \equiv w_2$	w_1 equivale a w_2
- ∃ v: w	Existe v tal que w.
- ∃ $v_1, v_2, \ldots, v_n$: w	Existen $v_1, v_2, \ldots, v_n$ tal que w.
- *more*	más
- *true*	cierto
- *false*	falso
- ◇ w	a veces w, eventualmente w.
- *if* e *then* w	si e entonces w
- *halt* e	alto si e
- ⊖ w	después w (débilmente)
- *skip*	salta
- r U s	r hasta s
- r wU s	r hasta s (débilmente)

[5] El operador U (*until*) es importante en la lógica temporal (de estados): □ y ○ pueden definirse en términos de U, [Kröger 84].

[6] wU (*weak until*), no requiere que el segundo operando se cumpla.

1.2.4 Sustituciones

Es posible definir fórmulas que se obtienen de otra fórmula mediante la sustitución de una variable por una expresión. Sin embargo, dicha sustitución debe hacerse con cierto cuidado para no obtener una fórmula con un significado indeseado. En esta sección introducimos definiciones que precisan los conceptos de *ocurrencia libre*, *enunciado* y *sustitución*.

El *alcance* de una ocurrencia de los símbolos □, ○, o ∀v en una fórmula de la forma ...(□ w)..., ...(○ w)... o ...(∀ v: w)... respectivamente es w. El *alcance* de una ocurrencia de ; en una fórmula de la forma ...(r ; s)... es rs. El *alcance* de una ocurrencia de ○ en fórmulas de la forma ...(○ e)... es e (e ∈ EXP).

Si v ∈ VAR y w ∈ FORM, decimos que una ocurrencia de v en w es una *ocurrencia ligada* sii ocurre inmediatamente después del símbolo ∀ o si está en el alcance de una ocurrencia de ∀v en w. Si una ocurrencia de v en w no es ligada decimos que es una *ocurrencia libre*.

Si w ∈ FORM y X es un objeto local, decimos que una ocurrencia de X en w es una *ocurrencia temporalmente ligada* sii dicha ocurrencia está en el alcance de una ocurrencia de un operador temporal en w. Si una ocurrencia de X en w no es temporalmente ligada decimos que es una *ocurrencia temporalmente libre*.

Si w ∈ FORM, w es un *enunciado* de la LTI sii w no tiene ocurrencias libres de ninguna variable. El *conjunto de enunciados* de la LTI, ENUN, es

ENUN = {w ∈ FORM / w es un enunciado}.

La clase de *enunciados clásicos* de la LTI, ENUNC, es:

ENUNC = ENUN ∩ FORMC .

Si e, d son expresiones y x es una variable, definimos e *con* d *en lugar de* x, e[d/x], como la expresión que resulta de e al hacer una *sustitución apropiada* en e poniendo d en lugar de x. Una sustitución en expresiones es *apropiada* si no introduce nuevos objetos locales temporalmente ligados. La definición de e[d/x] se precisa mediante inducción sobre la estructura de e: (v ∈ VAR, f es un simbolo de función y las e_1 son expresiones)

- e[d/x] = e si e ∈ {*empty*} ∪ {c_1}
- v[d/x] = v si v ≠ x,
  = d si v = x.
- (O e)[d/x] = (O e) si algún objeto local ocurre en d,
  = O (e[d/x]) si no.
- $f(e_1,e_2,\ldots,e_k)[d/x] = f(e_1[d/x],e_2[d/x],\ldots,e_k[d/x])$
- (*if* e_1 *then* e_2 *else* e_3)[d/x] =
  (*if* e_1[d/x] *then* e_2[d/x] *else* e_3[d/x])

Si e es una expresión, x es una variable y w es una fórmula, definimos w *con* e *en lugar de* x, w[e/x], como la fórmula que resulta de w al hacer una *sustitución apropiada* en w poniendo e en lugar de x. Una sustitución en fórmulas es apropiada si no introduce nuevas variables ligadas ni nuevos objetos locales temporalmente ligados. La definición de w[e/x] se precisa mediante inducción sobre la estructura de w: (r, s y las w_1 son fórmulas; las e_1 son expresiones; p es un predicado)

- w[e/x] = w si w ∈ {*empty*}
- (¬ w)[e/x] = ¬ w[e/x]
- (r ∧ s)[e/x] = r[e/x] ∧ s[e/x]
- $p(e_1,e_2,\ldots,e_k)[e/x] = p(e_1[e/x],e_2[e/x],\ldots,e_k[e/x])$
- $(e_1 = e_2)[e/x] = (e_1[e/x] = e_2[e/x])$
- (∀ v: w)[e/x]
  = ∀ v: w si x=v,
  = ∀ v: (w[e/x]) si x≠v y v no ocurre en e.
  = ∀ z: w[z/v][e/x] si x≠v y v ocurre en e.
  donde z es una variable del mismo tipo que v (global o local) distinta de x y distinta de las variables que ocurren en w y e .

- $(\Box\, w)[e/x]$ = $(\Box\, w)$ si algún objeto local ocurre en e,
  = $\Box\,(w[e/x])$ si no.
- $(O\, w)[e/x]$ = $(O\, w)$ si algún objeto local ocurre en e,
  = $O\,(w[e/x])$ si no.
- $(r;s)[e/x]$ = $(r;s)$ si algún objeto local ocurre en e,
  = $r[e/x]$; $s[e/x]$ si no.
- $(\mathit{if}\ e_0\ \mathit{then}\ w_1\ \mathit{else}\ w_2)[e/x]$ =
  $\mathit{if}\ e_0[e/x]\ \mathit{then}\ w_1[e/x]\ \mathit{else}\ w_2[e/x]$

1.3 Semántica

Para definir la semántica de la LTI utilizaremos una variante de las nociones clásicas de interpretación y modelo que incluye los aspectos temporales de la LTI.

1.3.1 Interpretaciones

Si D es una clase no vacía, decimos que s es una *sucesión finita* sobre D sii $\exists\, k \in \mathbb{N}$ tal que s: $\{i \in \mathbb{N} \,/\, 1 \leq k\} \longrightarrow D$.

Si D es una clase no vacía, decimos que s es una *sucesión infinita* sobre D sii s: $\mathbb{N} \longrightarrow D$.

Si D es una clase no vacía, decimos que s es una *sucesión* sobre D sii s es una sucesión finita o una sucesión infinita sobre D. En tal caso escribimos s_i en lugar de s(i) y en lugar de s escribimos $\langle s_j \rangle$, o $\langle s_0 s_1 \ldots s_k \rangle$ si s es finita, o $\langle s_0 s_1 \ldots s_k \ldots \rangle$ si s es infinita.

Si D es una clase no vacía, el *conjunto de estados* sobre D, Σ_D, es el conjunto de todas las funciones que le asignan a cada variable un elemento de D:

$\Sigma_D = \{s \,/\, s\colon VAR \longrightarrow D\}$.

Los elementos de Σ_D se denominan *estados* sobre D[1].

Si D es una clase no vacía, el *conjunto de intervalos* sobre D, Σ_D^+, es el conjunto de las sucesiones sobre Σ_D (finitas e infinitas) que mantienen fijos los valores de las variables globales:

---

[1]Habría sido más natural definir un estado como una función s: VARL $\longrightarrow$ D pero el enfoque que usamos resulta más cómodo en las definiciones.

$\Sigma_D^+ =$ { σ / (σ es una sucesión sobre Σ_D) $\wedge$
$\forall$ i $\forall$ u: ((u $\in$ VARG) $\supset$ σ_1(u) = σ_0(u))}.

Los elementos de Σ_D^+ se denominan *intervalos* sobre D.

Si σ es un intervalo decimos que σ es un *intervalo finito* sii σ es una sucesión finita. Decimos que σ es un *intervalo infinito* sii σ es una sucesión infinita.

Si $\sigma \in \Sigma_D^+$ es un intervalo, definimos la *longitud* de σ, $|\sigma|$, como el número de estados de σ menos 1, es decir:

$|\sigma| = k$ si $\sigma = \langle\sigma_0\sigma_1\ldots\sigma_k\rangle$ (σ finito)
$= \aleph_0$ si $\sigma = \langle\sigma_0\sigma_1\ldots\sigma_k\ldots\rangle$ (σ infinito)

Normalmente denotaremos a un intervalo σ mediante
$\langle\sigma_0\sigma_1\ldots\sigma_{|\sigma|}\rangle$.

Si $\sigma \in \Sigma_D^+$ es un intervalo y $k \in \mathbb{N}$, entonces el *sufijo* k-esimo de σ, σ^k, y el *prefijo* k-esimo de σ, ${}^k\sigma$, son los siguientes intervalos:

$\sigma^k = \langle\sigma_k\sigma_{k+1}\ldots\sigma_{|\sigma|}\rangle$ si $0 \leq k \leq |\sigma|$,
$= \sigma$ si no.
${}^k\sigma = \langle\sigma_0\sigma_1\ldots\sigma_k\rangle$ si $0 \leq k \leq |\sigma|$,
$= \sigma$ si no.

Si σ, $\sigma' \in \Sigma_D^+$ son dos intervalos, decimos que σ' es *subintervalo de* σ, $\sigma' \leq \sigma$, sii
$|\sigma'| \leq |\sigma|$ y $\exists$ k (Vi (($0 \leq i \leq |\sigma'|$) $\supset$ $\sigma'_i = \sigma_{i+k}$).
Claramente, $\leq$ es reflexiva, transitiva y antisimétrica. También se puede ver fácilmente que $\forall j(\sigma^j \leq \sigma \wedge {}^j\sigma \leq \sigma)$.

Si σ, $\sigma' \in \Sigma_D^+$ y v $\in$ VAR, decimos que σ' y σ son *intervalos equivalentes* módulo v, $\sigma' \simeq_v \sigma$, sii
$|\sigma'| = |\sigma|$ y Vi$\forall$x: (($0 \leq i \leq |\sigma|$ y x $\neq$ v) $\supset$ $\sigma'_i(x) = \sigma_i(x)$)
Se puede ver fácilmente que $\simeq_v$ es una relación de equivalencia.

Si D es una clase no vacía, una *valuación global* de la LTI sobre D, γ, es una función que le asigna un valor a las constantes, funciones y predicados de la LTI cumpliendo lo siguiente:

- para toda constante global c_i: $\gamma(c_i) \in D$
- para toda función f_i^J: $\gamma(f_i^J) : D^J \longrightarrow D$
- para todo predicado p_i^J: $\gamma(p_i^J) : D^J \longrightarrow \{true, false\}$

Es importante notar que γ no le asigna un valor a la constante local *empty*.

Una *interpretación* de la LTI es una tercia $\langle D, \gamma, \sigma \rangle$, donde:

- D, la *base o dominio de la interpretación*, es una clase tal que $\{true, false\} \subseteq D$,
- γ es una valuación global,
- $\sigma \in \Sigma_D^+$.

La condición que se impone a D es necesaria porque más adelante, al definir la semántica asociada a una interpretación, le asignaremos a cada expresión un elemento de D y es necesario para nuestros propósitos que la expresión *empty* tenga asignado un valor de verdad.

1.3.2 Modelos

Tomando como base una interpretación $\langle D,\gamma,\sigma\rangle$ de la LTI es posible definir una función que le asigne a cada expresión un elemento de D y a cada fórmula un valor de verdad. Esta función determina una relación de satisfactibilidad que fija la semántica de la LTI.

Si $I=\langle D,\gamma,\sigma\rangle$ es una interpretación de la LTI, la *semántica asociada* a I es la función

M_I: (EXP ∪ FORM) ⟶ D

que cumple las siguientes reglas[2]:

(Escribimos $M_\sigma[x]$ en lugar de $M_I(x)$ considerando que D y γ están fijos. c es una constante, v es una variable, f es una función, p es un predicado, las e_i son expresiones y las w_i son fórmulas)

- $M_\sigma[c] = \gamma(c)$.
- $M_\sigma[empty] = $ true sii $|\sigma|=0$. ($empty \in$ EXP ∪ FORM).
- $M_\sigma[v] = \sigma_0[v]$.
- $M_\sigma[f(e_1,\ldots,e_k)] = \gamma(f)(M_\sigma[e_1],\ldots,M_\sigma[e_k])$.
- $M_\sigma[\bigcirc e] = M_{\sigma^1}[e]$.
- $M_\sigma[$*if* e_1 *then* e_2 *else* $e_3]$ $= M_\sigma[e_2]$ si $M_\sigma[e_1]$ = true, $= M_\sigma[e_3]$ si $M_\sigma[e_1] \neq$ true.
- $M_\sigma[p(e_1,\ldots,e_k)] = \gamma(p)(M_\sigma[e_1],\ldots,M_\sigma[e_k])$.
- $M_\sigma[e_1 = e_2]$ = true sii $M_\sigma[e_1] = M_\sigma[e_2]$.
- $M_\sigma[\neg w]$ = true sii $M_\sigma[w]$ = false.
- $M_\sigma[w_1 \wedge w_2]$ = true sii $M_\sigma[w_1]$=true y $M_\sigma[w_2]$=true
- $M_\sigma[\bigcirc w]$ = true sii $M_{\sigma^1}[w]$=true.
- $M_\sigma[\forall$ v: w] = true sii
  $M_{\sigma'}[w]$ =true para toda $\sigma' \in \Sigma_D^+$ tal que $\sigma' \sim_v \sigma$.
- $M_\sigma[\Box w]$ = true sii $\forall\, i \in \mathbb{N}$: $i \leq |\sigma| \supset M_{\sigma^i}[w]$=true.

---

[2]Esta definición es esencialmente la que da [Moszkowski 86].

- $M_\sigma[w_1;\ w_2]$ = true sii existe $i \in \mathbb{N}$ tal que
  $i \le |\sigma|$, $M_{\sigma_1}[w_1]$=true y $M_{\sigma^i}[w_2]$=true.
- $M_\sigma[$*if* e *then* w_1 *else* $w_2]$ $= M_\sigma[w_1]$ si $M_\sigma[e]$ = true,
  $= M_\sigma[w_2]$ si $M_\sigma[e] \neq$ true.

Si $I=\langle D,\gamma,\sigma\rangle$ es una interpretación de la LTI y $w \in$ FORM, decimos que I *realiza* a w o que I es un *modelo* para w, $I \models w$, sii $M_\sigma[w]$=true.

Decimos que una interpretación $I=\langle D,\gamma,\sigma\rangle$ es un *modelo* para un conjunto de fórmulas F, $I \models F$, sii $\forall f: (f \in F \supset I \models f)$.

Una fórmula w es *consecuencia semántica* de un conjunto de fórmulas F, $F \models w$, sii para toda interpretación I: $I \models F \supset I \models w$.

Decimos que w es una *fórmula válida* (tautología), $\models w$, sii para toda interpretación I: $I \models w$ (equivalentemente, si Φ es el conjunto vacío: $\Phi \models w$).

Es conveniente recalcar algunas características sutiles de la fórmulas de la LTI con cuantificadores. En las lógicas temporales es común que sólo se permita la cuantificación sobre variables globales. La LTI permite cuantificación sobre cualquier variable individual (global o local). A continuación damos un ejemplo para aclarar las diferencias sutiles que existen entre estos dos tipos de cuantificación:

Supongamos que $u, t \in$ VARG, $u \neq t$, $v \in$ VARL, $v \neq t$ y consideremos las fórmulas:

a) $\forall u: \exists t: \Box u = t$

b) $\forall v: \exists t: \Box v = t$

Estas fórmulas sólo difieren en que la primera cuantifica globalmente y la segunda cuantifica localmente. Se puede demostrar que la primera es válida y que la segunda no lo es:

Demostraremos $\models \forall u: \exists t: \Box u = t$.

Sup. que $I=\langle D,\gamma,\sigma\rangle$ es una interpretación.

Sea σ' tal que $\sigma' \approx_u \sigma$

como $u \in VARG$, tenemos que $\forall i(\sigma'_1(u)=\sigma'_0(u))$.
$\therefore$ podemos definir $\tau \in \Sigma_D^+$ como sigue:

$\tau_1(x) = \sigma'_1(u)$ si $x=t$
$= \sigma'_1(x)$ si $x \neq t$

$\therefore \forall i(\ \tau_1(t)=\sigma'_1(u)=\tau_1(u)\ )$
$\therefore \forall i\ M_{\tau^i}[u=t]=true$,
$\therefore M_\tau[\Box\, u=t]=true$ y $\tau \approx_t \sigma'$,
$\therefore M_\sigma,[\exists\, t: \Box\, u=t]$
$\therefore M_\sigma[\forall\, u: \exists\, t: \Box\, u=t]$, $\therefore I \models \forall\, u: \exists\, t: \Box\, u=t$
$\therefore \models \forall\, u: \exists\, t: \Box\, u=t$. $\Box$

Ahora demostraremos que $\forall v: \exists\, t: \Box\, v = t$ no es válida:
Tomamos $I=<D,\gamma,\sigma>$ con $D=\{true,false\}$, γ una valuación global, y $\sigma \in \Sigma_D^+$ definido como sigue:

$\sigma = <\sigma_0\ \sigma_1>$
$\sigma_0(x)$ = true si $x=v$
= false si no
$\sigma_1(x)$ = false si $x=v$
$= \sigma_0(x)$ si no

Sea $\tau \approx_t \sigma$, entonces:
$t_1(v)=\sigma_1(v)$ y $\tau_0(t) \in D=\{true,false\}$.
si $\tau_0(t)=true$ ent. $\tau_1(t)=true \neq \tau_1(v)$, $\therefore M_{\tau^1}[v=t]=false$
si $\tau_0(t)=false$ ent. $\tau_0(t) \neq \tau_0(v)$, $\therefore M_{\tau^0}[v=t]=false$
$\therefore M_\tau[\Box\, v=t]=$ false
$\therefore \forall\, \tau(\tau \approx_t \sigma \supset M_\tau[\Box\, v=t]=$ false)
$\therefore M_\sigma[\exists\, t: \Box\, v=t]=$ false y $\sigma \approx_v \sigma$
$\therefore M_\sigma[\forall\, v: \exists\, t: \Box\, v=t]=$ false
$\therefore I$ no realiza a $\forall\, v: \exists\, t: \Box\, v=t$
$\therefore \forall\, v: \exists\, t: \Box\, v=t$ no es válida. $\Box$

Terminamos esta sección con más definiciones que se usarán más adelante.

Si $I=<D,\mathcal{I},\sigma>$ es una interpretación, $t \in EXP$ y $x \in VAR$, definimos σ *con* t *en lugar de* x, $\sigma[t/x]$, como el intervalo que cumple:

- $|\sigma[t/x]|=|\sigma|$
- $\sigma[t/x]_I(h) = M_{\sigma I}[t]$ si $h=x$
  $= \sigma_I(h)$ si $h \neq x$

Claramente, $\sigma[t/x] \approx_x \sigma$.

Una fórmula $w \in FORM$ (o $F \subseteq FORM$) es *satisfacible* sii existe una interpretación I tal que $I \models w$ (o $I \models F$) .

Un conjunto de enunciados, $F \subseteq ENUN$, es una *teoría sii* F es satisfacible y F es *cerrada baja consecuencia semántica*, es decir, $\forall w$ ($(w \in ENUN$ y $F \models w) \supset w \in F$).

Si $F \subseteq ENUN$, las *consecuencias semánticas* de F, *Con*(F), son el conjunto $Con(F) = \{w \in ENUN \ / \ F \models w\}$. Claramente, si F es satisfacible *Con*(F) es una teoría.

La *lógica temporal de intervalos*, LTI, es la teoría *Con*(Φ), es decir : $LTI = Con(\Phi) = \{w \in ENUN \ / \models w\}$

1.4 Axiomatización

En esta sección presentamos algunos conceptos y resultados en relación a los sistemas axiomáticos de la LTI.

Comenzamos con una definición de sistema axiomático para la LTI y continuamos la presentación de algunos resultados acerca de la completez de dichos sistemas y de los sistemas axiomáticos de las lógicas temporales en general. Finalmente, presentamos un sistema axiomático para la LTI acompañado de demostraciones que prueban la correctez de algunos de sus axiomas.

1.4.1 Sistemas axiomáticos

En esta sección presentamos brevemente los conceptos de deducción ($\vdash$) y de sistema axiomático como contraparte sintáctica de los conceptos de consecuencia semántica ($\models$) y de modelo.

Decimos que un conjunto X es *decidible* sii existe un algoritmo P[1] tal que

P(e) =1 si e $\in$ X.
=0 si e $\notin$ X.

Decimos que un conjunto X es *recursivamente enumerable* (r.e.) o *semi-decidible* sii existe un semi-algoritmo P[2] tal que:

P(e) =1 si e$\in$ X
=0 o P no termina si e$\notin$ X

Decimos que J es una *regla de inferencia* de la LTI sii existe n $\in$ $\mathbb{N}$ tal que J $\subseteq$ FORM^n x FORM y J es un conjunto decidible.

---

[1] P puede ser un programa que con cualquier entrada termina.

[2] P puede ser un programa que no siempre termina.

Si J es una regla de inferencia y $(F,w) \in J$, decimos que los elementos de F son las *premisas* de la regla y w es el *resultado de aplicar* J a F.

Las reglas de inferencia que usaremos aquí son:

a) *modus ponens* (MP):
$MP=\{((f,g),h) \in FORM^2 \times FORM \mid \exists A,B\ (f=A \supset B \text{ y } g=A \text{ y } h=B)\}$

b) *generalización* (GEN):
$GEN=\{(f,g) \in FORM \times FORM \mid \exists A,x\ (f=A \text{ y } g=\forall x:A)\}$

c) *necesitación* (NEC):
$NEC=\{(f,g) \in FORM \times FORM \mid \exists A\ (f=A \text{ y } g=\Box A)\}$

Decimos que $A \subseteq FORM$ es un *conjunto de axiomas* de la LTI sii A es decidible.

Un *sistema axiomático* (finitista) de la LTI es una pareja <A,R> donde A es un conjunto de axiomas y R es un conjunto decidible (generalmente finito) de reglas de inferencia.

Si S=<A,R> es un sistema axiomático, $w \in FORM$ y $F \subseteq FORM$, decimos que w *se deduce de* F en S, $F \vdash_S w$, sii existe una *demostración* de w a partir de F, es decir, una sucesión finita de fórmulas, $s_0, s_1, \ldots, s_n$, tal que:

- $s_n = w$ y
- $\forall i: [\ (s_i \in A) \vee (s_i \in F) \vee$
  $(\exists J \in R: (\ ((g_1, g_2, \ldots, g_n), s_i) \in J \wedge$
  $\{g_j / 1 \leq j \leq n\} \subseteq \{s_h / 0 \leq h < i\}\ )\ ]$

Si S es un sistema axiomático y $w \in FORM$, decimos que w es un *teorema* en S, $\vdash w$, sii $\Phi \vdash_S w$.

Si S=<A,R> es un sistema axiomático, decimos que:
S es *correcta* sii $\forall w: \vdash w \supset \models w$
S es *completa* sii $\forall w: \models w \supset \vdash w$.

1.4.2 Incompletez

En esta sección las lógicas temporales a las que hagamos referencia serán lógicas temporales de estados si no se hace explícito que se trata de lógicas temporales de intervalos.

Un aspecto importante de un sistema axiomático es la relación que pueda establecerse entre la validez de una fórmula y su demostrabilidad en el sistema. La relación ideal es la que se da cuando un sistema es correcto y completo. En tal caso se cuenta con una relación de equivalencia ($\models f$ sii $\vdash f$) que permite pasar del aspecto semántico de una fórmula al aspecto sintáctico y viceversa.

Desafortunadamente, la relación de equivalencia entre validez y demostrabilidad no es posible para todas las lógicas temporales y no se da en el caso de la LTI. A continuación presentamos algunos resultados al respecto que ubican el caso de la LTI.

En [Rescher-Urquhart 71] se encuentran varios sistemas axiomáticos para diversas variaciones semánticas de la lógica temporal. Estas variaciones incluyen lógicas para tiempo lineal, tiempo arborescente (branching-time) y lógicas temporales con y sin símbolos de cuantificación universal. Resalta el hecho de que todas ellas cuentan con un sistema axiomático correcto y completo. Sin embargo, ninguna de estas lógicas contiene operadores similares a los operadores *después*, *corta* y *hasta* (O, ;, U).

La lógica temporal proposicional que incluye a los operadores $\Box$ (*siempre*), O (*después*) y U (*hasta*, *until*), también cuenta con un sistema axiomático correcto y completo. De hecho, el sistema axiomático que se presenta en [Thayse 89] cumple con esto y está constituido por los siguientes esquemas axiomáticos y reglas de inferencia: (A,B y C son fórmulas proposicionales)

Cálculo de proposiciones:

A1) $(A \supset (B \supset A))$

A2) $((A \supset (B \supset C)) \supset ((A \supset B) \supset (A \supset C)))$

A3) $((\neg A \supset \neg B) \supset ((\neg A \supset B) \supset A))$

Lógica temporal de proposiciones de tiempo lineal:

A4) $\bigcirc (A \supset B) \supset (\bigcirc A \supset \bigcirc B)$

A5) $\neg\bigcirc A \equiv \bigcirc \neg A$

A6) $\Box (A \supset B) \supset (\Box A \supset \Box B)$

A7) $\Box A \supset (A \wedge \bigcirc A \wedge \bigcirc\Box A)$

A8) $\Box (A \supset \bigcirc A) \supset (A \supset \Box A)$

A9) $A \mathbf{U} B \supset (B \vee (A \wedge \bigcirc (A \mathbf{U} B)))$

A10) $(C \wedge \Box (C \supset B \vee (A \wedge \bigcirc C)))) \supset A \mathbf{U} B$

Reglas de inferencia:

R1) Modus Ponens (MP):
B es resultado de aplicar MP a A y $A \supset B$.

R2) Necesitación (NEC): $\Box A$ es resultado de aplicar NEC a A.

Hemos enfocado este trabajo al estudio de la LTI de primer orden sin profundizar en las propiedades de la LTI proposicional. Una referencia interesante a trabajo en esta dirección es la de [Abadi-Manna 85].

La situación, en el aspecto axiomático, para la lógica temporal de primer orden con operadores $\Box$ y $\bigcirc$ es más complicada que la del caso proposicional. El estudio de la lógica temporal de primer orden mediante sistemas axiomáticos está limitado por resultados importantes acerca de la completez de este tipo de lógicas. Presentamos algunos resultados en esta dirección precedidos por algunas definiciones de tipo general:

Decimos que una *lógica*[3] es una tercia formada por
-un conjunto de fórmulas definidas sobre un alfabeto,
-una clase de interpretaciones admisibles,
-una relación de satisfacción que le asigna un valor de verdad a cada fórmula en una interpretación admisible.

---

[3] [Ebbinghaus-Flum-Thomas 84]

La *signatura* de un lenguaje temporal (de primer orden) es la clase formada por:

- los símbolos de predicado (globales y locales)
- los símbolos de función (globales y locales)
- los símbolos de variable (globales y locales).

La LTI está parametrizada por las signaturas de su lenguaje. Generalmente nos referimos a la LTI considerando una signatura arbitraria, aunque es posible referirse a la LTI con diferentes signaturas particulares.

Decimos que una lógica es *débilmente incompleta* sii su conjunto de tautologías (sobre una signatura arbitraria) no es recursivamente enumerable o, equivalentemente, sii no hay un sistema axiomático (finitista) que sea correcto y completo para esta lógica.

Decimos que una lógica es *fuertemente incompleta* sii no existe ninguna signatura tal que el conjunto de tautologías sobre dicha signatura es recursivamente enumerable o, equivalentemente[4], sii el conjunto de tautologías sobre la signatura vacía no es recursivamente enumerable.

Los siguientes resultados son de gran importancia para la lógica temporal (de estados):

I) Si el lenguaje de la lógica temporal con operadores $\Box$ y $\bigcirc$ contiene un símbolo de constante 0, un símbolo de función de aridad uno s y dos símbolos de función de aridad dos + y *, entonces no existe ningún sistema axiomático finitista para dicha lógica que sea correcto y completo[5].

---

[4] usando que: si A es r.e. y $B \subseteq A$ entonces B es r.e.

[5] [Szalas 86] .

II) La lógica temporal de primer orden con operador until es débilmente incompleta.[6]

III) La lógica temporal de primer orden con operador until es fuertemente incompleta.[7]

Como un ejemplo de las técnicas empleadas en la demostración de este tipo de afirmaciones, incluimos más adelante la adaptación de la demostración que dan A. Szalas y L. Holenderski de la afirmación del inciso (II) al caso de la LTI. Respecto a las otras dos afirmaciones solamente mencionaremos un par de hechos interesantes acerca de las demostraciones que dan los autores mencionados:

- La demostración de la afirmación (I) se basa en la incompletez de la aritmética de primer orden.

- La demostración de la afirmación (II) se basa en que la propiedad de finitud es representable con una fórmula de la lógica temporal de primer orden con operador *until*.

- La demostración de la afirmación (III) consiste en reducir el complemento del problema de la detención (halting problem) al problema de determinar si una fórmula temporal es válida. La reducción se logra mostrando cómo codificar la computación de una máquina de Turing con una fórmula de la lógica temporal de primer orden con operador *until*.

Procedemos a demostrar que la lógica temporal de primer orden es débilmente incompleta:

Definimos la *fórmula de finitud*, f_{fin}, como la fórmula
$\exists u_1$: $((\Box \Diamond v_1=u_1) \wedge \Box (v_1=u_1 \supset \bigcirc \forall u_2$: $(\neg(v_1=u_1)\ U\ v_1=u_2)))$.

---

[6] [Szalas-Holenderski 88] p. 320.

[7] [Szalas-Holenderski 88] p. 325.

La fórmula de finitud dice intuitivamente dos cosas respecto a la sucesión de valores de v_1:

1) Que siempre aparecerá, en algún instante, el valor u_1.
2) Que entre dos valores consecutivos iguales a u_1 aparecerán todos los valores del dominio.

Es decir, si la sucesión de valores de v_1 es $\{d_i\}$, gráficamente tenemos:

$$d_0 d_1 \cdots \underset{\uparrow u_1}{d_{i_1} d_{i_1+1}} \cdots \underset{\uparrow u_1}{d_{i_2} d_{i_2+1}} \cdots \underset{\uparrow u_1}{d_{i_3} d_{i_3+1}} \cdots$$

$$D \subseteq \{d_{i_1}, d_{i_1+1}, \ldots, d_{i_2-1}\}$$

Esto se formaliza en el siguiente lema:

Lema 2.1.- Si $J=<D,\gamma,\sigma>$ es una interpretación y $J \models f_{fin}$ entonces D es finito.

Demostración

Consideremos dos subfórmulas de f_{fin} :

$f_1(u_1)= ((\Box \Diamond v_1=u_1)$ y

$f_2(u_1)= \Box (v_1=u_1 \supset \bigcirc \forall u_2\colon (\neg(v_1=u_1)\ U\ v_1=u_2))$,

de tal forma que $f_{fin}= \exists u_1 : (f_1(u_1) \wedge f_2(u_1))$

Sup. $J \models f_{fin}$, entonces:

$\exists\ \tau \in \Sigma_D^+$ tal que $\tau \approx_{u_1} \sigma$ y $M_\tau[f_1(u_1) \wedge f_2(u_1)]$= true,

sea $d=\tau_0(u_1)$, $u_1 \in VARG$, $\therefore\ \tau = \sigma[d/u_1]$ ..... (1)

$\therefore\ M_{\sigma[d/u_1]}[f_1(u_1)]$= true y $M_{\sigma[d/u_1]}[f_2(u_1)]$= true,

$M_\tau[\Box \Diamond v_1=u_1]$= true,

$\therefore\ \forall\ i \in \mathbb{N}\ M_{\tau^i}[\Diamond v_1=u_1]$= true, ... (2)

$\therefore\ M_\tau[\Diamond v_1=u_1]$=true,

$\therefore\ \exists\ i \in \mathbb{N}$ tal que $M_{\tau^i}[v_1=u_1]$=true,

Sea $i \in \mathbb{N}$ la mínima que cumple

$M_{\tau^i}[v_1=u_1]$= true, ... (3)

$\therefore\ \sigma[d/u_1]_1(v_1) = \tau^1{}_0(v_1) = \tau^i{}_0(u_1) = \tau_0(u_1) = d$.

$\therefore \sigma[d/u_1]_1(v_1) = d$ ..... (4)

$\therefore$ por otro lado, $M_\tau[f_2(u_1)]$= true,

$\therefore \forall h\in N\ (M_{\tau^h}[v_1=u_1 \supset \bigcirc \forall u_2\colon (\neg(v_1=u_1)\ U\ v_1=u_2)]$= true),

$\therefore M_{\tau^1}[v_1=u_1 \supset \bigcirc \forall u_2\colon (\neg(v_1=u_1)\ U\ v_1=u_2)]$= true,

pero por (3) tenemos que $M_{\tau^1}[v_1=u_1]$= true,

$\therefore M_{\tau^1}[\bigcirc \forall u_2\colon (\neg(v_1=u_1)\ U\ v_1=u_2)]$= true,

$\therefore M_{\tau^{i+1}}[\forall u_2\colon (\neg(v_1=u_1)\ U\ v_1=u_2)]$= true,

Sea $\tau'=\tau^{i+1}$ ($\tau'=(\sigma[d/u_1])^{i+1}$) . Entonces :

$\forall \rho\in\Sigma_D^+\ (\rho^\alpha{}_{u_2}\tau' \supset M_\rho[\neg(v_1=u_1)\ U\ v_1=u_2]$= true),

$\therefore \forall e\in D\ (M_{\tau'[e/u_2]}[\neg(v_1=u_1)\ U\ v_1=u_2]$= true),

$\therefore \forall e\in D\ \exists j\in N\ \ (M_{(\tau'[e/u_2])^j}[v_1=u_2]$= true $\wedge$

$\forall k<j\ (M_{(\tau'[e/u_2])^k}[\neg(v_1=u_1)]$= true)),

$\therefore \forall e\in D\ \exists j\in N\ \forall k<j$

$(\tau'[e/u_2]^j{}_0(v_1) = \tau'[e/u_2]^j{}_0(u_2) \wedge$

$\tau'[e/u_2]^k{}_0(v_1) \neq \tau'[e/u_2]^k{}_0(u_1)$)

$\therefore \forall e\in D\ \exists j\in N\ \forall k<j$

$(\sigma[d/u_1][e/u_2]_{i+1+j}(v_1) = \sigma[d/u_1][e/u_2]_{i+1+j}(u_2) = e \wedge$

$\sigma[d/u_1][e/u_2]_{i+1+k}(v_1) \neq \sigma[d/u_1][e/u_2]_{i+1+k}(u_1) = d$)

pero $\sigma_{i+1+k}(v_1) = \sigma[d/u_1][e/u_2]_{i+1+k}(v_1)$,

$\therefore \forall e\in D\ \exists j\in N\ \forall k<j$

$(\sigma_{i+1+j}(v_1) = e \wedge \sigma_{i+1+k}(v_1) \neq d)$ ..... (a)

usando (2) y $\tau'=\tau^{i+1}$, tenemos que $M_{\tau'}[\Diamond v_1=u_1]$= true,

$\therefore \exists\, h \in \mathbb{N}\ (M_{\tau,h}[v_1=u_1] = \text{true})$,

sea h la mínima que cumple $(M_{\tau,h}[v_1=u_1] = \text{true})$, ent.

$\sigma[d/u_1]_{1+1+h}(v_1) = \tau,^{h}_{0}(v_1) = \tau,^{h}_{0}(u_1) = \tau_{1+1+h}(u_1) = d$

$\therefore \sigma[d/u_1]_{1+1+h}(v_1) = d,$

$\therefore \sigma_{1+1+h}(v_1) = d$ ..... (b)

y de (4) tenemos $\sigma_1(v_1) = d$ ..... (c)

de (a), (b) y (c) se concluye que

$D \subseteq \{\sigma_1(v_1), \sigma_{1+1}(v_1), \ldots, \sigma_{1+1+h}(v_1)\}$

$\therefore$ D es finito. □

Lema 2.2.- Si D es finito y γ es una valuación global, entonces existe $\sigma \in \Sigma^{+}_{D}$ tal que $\langle D, \gamma, \sigma\rangle \models f_{fin}$.

Demostración:

Sup. $n \in \mathbb{N}$, $n \geq 1 \wedge D=\{d_0, d_1, \ldots, d_{n-1}\}$.

Sea $\sigma \in \Sigma^{+}_{D}$ tal que para i, $k \in \mathbb{N}$:

$\sigma_{i+kn}(x) = d_i$ si $x=v_1$

$= d_0$ si no.

Se verifica fácilmente que

$M_{\sigma[d_0/u_1]}[\Box \Diamond v_1=u_1] = \text{true}$, y que

$M_{\sigma[d_0/u_1]}[\Box\, (v_1=u_1 \supset \bigcirc \forall u_2\colon\ (\neg(v_1=u_1)\ \mathcal{U}\ v_1=u_2))] = \text{true}$

$\therefore M_{\sigma} \models f_{fin}$.□

Enunciamos sin demostración los siguientes resultados de la lógica clásica que se trasladan a los enunciados clásicos de la LTI[8]:

---

[8] Ver [Ebbinghaus-Flum-Thomas 84].

Lema 2.3.- Si $f \in ENUNC$ y $J = \langle D,\gamma,\sigma\rangle$ es una interpretación, entonces:

a) $J \models f \vee J \models \neg f$

b) $J \models f \supset \forall \tau \in \Sigma_D^+ \langle D,\gamma,\tau\rangle \models f$.

Teorema de Trahktenbrot- El conjunto de los enunciados clásicos satisfacibles por todas las interpretaciones con dominio finito no es recursivamente enumerable, es decir,

Si $F_{fin} = \{ f \in ENUNC / \forall D,\gamma,\sigma\ (D \text{ finito} \supset \langle D,\gamma,\sigma\rangle \models f) \}$, entonces F_{fin} no es recursivamente enumerable.

Lema 2.4.- Si f es un enunciado clásico de la LTI ($f \in ENUNC$), entonces $\models (f_{fin} \supset f)$ sii

$\forall D,\gamma,\sigma\ (D \text{ finito} \supset \langle D,\gamma,\sigma\rangle \models f)$.

Demostración:

($\longrightarrow$) Sup. $\models (f_{fin} \supset f)$

Sup. $\exists\ D,\gamma,\sigma$ tales que D finito y $\langle D,\gamma,\sigma\rangle$ no realiza a f,

$\therefore \langle D,\gamma,\sigma\rangle \models \neg f$ (lema 2.3 (a)).

Por el lema 2.2, existe $\tau \in \Sigma_D^+$ tal que $\langle D,\gamma,\tau\rangle \models f_{fin}$.

$\therefore \langle D,\gamma,\tau\rangle \models \neg f$ (lema 2.3 (b))

$\therefore \langle D,\gamma,\tau\rangle$ no realiza a f,

$\therefore (f_{fin} \supset f)$ no es válida. Esto contradice la hipótesis original,

$\therefore \forall D,\gamma,\sigma\ (D \text{ finito} \supset \langle D,\gamma,\sigma\rangle \models f)$.

($\longleftarrow$) Sup. $\forall D,\gamma,\sigma\ (D \text{ finito} \supset \langle D,\gamma,\sigma\rangle \models f)$.

Sea $J=\langle D,\gamma,\sigma\rangle$ una interpretación,

Si $J \models f_{fin}$ entonces:

D es finito (lema 2.1),

$\therefore J \models f$,

$\therefore \forall J\ (J \models f_{fin} \supset J \models f)$

$\therefore \models (f_{fin} \supset f)$

Teorema 2.5.- La LTI es débilmente incompleta.

Demostración:

Sup. que la LTI es completa, entonces:

Existe un sistema axiomático para la LTI tal que si $f \in FORM$ entonces: $\models f$ sii $\vdash f$.

Sea G= {f∈ENUNC/ ⊨ f_{fin}⊃f}, entonces
G= {f∈ENUNC/ ⊢ f_{fin}⊃f}, ∴ G es recursivamente enumerable.
por el lema 2.4, tenemos que G=F_{fin},
∴ F_{fin} es recursivamente enumerable, pero esto contradice al teorema de Trahktenbrot,
∴ la LTI es incompleta. Esta incompletez es débil porque la demostración del teorema de Trahktenbrot requiere de una signatura particular. □

1.4.3 Un sistema axiomático.

A pesar de los resultados de incompletez mencionados anteriormente, en esta sección proponemos un sistema axiomático que se aproxima a la LTI. Este sistema tiene como base el sistema que describe [Thayse 89] para el cálculo de predicados temporal. Hemos agregado el operador *corta*, y la constante *empty*; también hemos suprimido el operador temporal *hasta* (*until*) y hemos modificado ligeramente la definición de objeto local[9].

A continuación presentamos un sistema axiomático que se aproxima a la LTI y que denominamos SLTI, *Sistema axiomática para la Lógica Temporal de Intervalos*. Los esquemas axiomáticos y reglas de inferencia de este sistema son los siguiente: (A,B,C y w son fórmulas; t es una expresión; x z son variables).

Cálculo de proposiciones:
A1) (A ⊃ (B ⊃ A))
A2) ((A ⊃ (B ⊃ C)) ⊃ ((A ⊃ B) ⊃ (A ⊃ C)))
A3) ((¬A ⊃ ¬B) ⊃ ((¬A ⊃ B) ⊃ A))

Lógica temporal de proposiciones de tiempo lineal:
A4) O (A ⊃ B) ⊃ (O A ⊃ O B)

---

[9]De acuerdo con la definición de OBL, nuestros objetos locales son las variables locales y la constante *empty*.

A5) $\neg \bigcirc A \equiv \bigcirc \neg A$
A6) $\Box (A \supset B) \supset (\Box A \supset \Box B)$
A7) $\Box A \supset (A \wedge \bigcirc A \wedge \bigcirc \Box A)$
A8) $\Box (A \supset \bigcirc A) \supset (A \supset \Box A)$

Lógica temporal de predicados de tiempo lineal:
A9) $(A \supset \bigcirc A)$ si A no tiene objetos locales
A10) $(\forall x{:}A) \supset A[t/x]$ si t no tiene objetos locales
A11) $(\forall x{:}(A \supset B)) \supset (A \supset \forall x{:}B)$ si x no ocurre libre en A
A12) $(\forall x{:}\bigcirc A) \supset \bigcirc \forall x{:}\ A$

A13) $x = x$
A14) $x{=}z \supset (A \supset A[z/x])$ si A no tiene operadores temporales

Lógica temporal de intervalos:
A15) $(empty \wedge A) \supset \bigcirc A$
A16) $A;\ empty \equiv A$
A17) $empty;\ A \equiv A$
A18) $(A\ ;\ (B\ ;\ C)) \equiv ((A\ ;\ B)\ ;\ C)$
A19) $((A;B) \wedge empty) \equiv (A \wedge B \wedge empty)$

Reglas de inferencia:
R1) Modus Ponens (MP):
B es resultado de aplicar MP a A y $A \supset B$.
R2) Generalización (GEN):
$(\forall x{:}\ A)$ es resultado de aplicar GEN a A.
R3) Necesitación (NEC):
$\Box A$ es resultado de aplicar NEC a A.

En vista de los resultados mencionados anteriormente, el SLTI no puede ser completo, sin embargo sí es correcto: $\vdash w \supset \models w$. Esto se puede probar usando inducción sobre la longitud de la demostración de w. Para esto es necesario probar que los axiomas son válidos y que las reglas de inferencia preservan la validez.

En la literatura del tema es más común encontrar trabajos sobre la lógica temporal de primer orden de tiempo lineal con semántica de estados y cuantificación sobre variables globales. Pensando

que los rasgos que distinguen a la LTI de la lógica temporal (la semántica de intervalos, la cuantificación sobre variables locales, el operador *corta*, etc.) pudieran provocar cierta inquietud, demostramos a continuación que las reglas de inferencia del SLTI preservan la validez y que los axiomas del mismo son válidos.

Teorema 3.1- Las reglas de inferencia del SLTI preservan la validez:

Demostración:

a) Modus Ponens preserva la validez:

Si $\models A$ y $\models A \supset B$ entonces $\models B$.

Sup. $\models A$ y $\models A \supset B$, ent. $\forall I$: $I \models A \wedge I \models A \supset B$,

Sea $I=\langle D, \gamma, \sigma\rangle$ una interpretación, entonces:

$M_I \models A$ y $M_I \models A \supset B$, $\therefore M_\sigma[A]=$ true y $M_\sigma[A \supset B]=$ true,

$\therefore M_\sigma[A]=$ true y ($M_\sigma[A]=$false o $M_\sigma[B]=$ true),

$\therefore M_\sigma[B]=$ true, $\therefore M_I \models B$, $\therefore \models B$. □

b) Generalización preserva la validez:

Si $\models A$ entonces $\models \forall x: A$.

Sup. $\models A$, entonces $\forall I$: $I \models A$,

Sea $I=\langle D, \gamma, \sigma\rangle$ una interpretación, entonces:

Si $\sigma' \alpha_x \sigma$ entonces

$\langle D, \gamma, \sigma'\rangle \models A$, $\therefore M_{\sigma'}[A]=$ true,

$\therefore \forall \sigma' (\sigma' \alpha_x \sigma \supset M_{\sigma'}[A]=$ true), $\therefore M_\sigma[\forall x: A]=$ true, $\therefore I \models \forall x: A$

$\therefore \forall I \quad I \models \forall x: A$, $\therefore \models \forall x: A$. □

c) Necesitación preserva la validez:

Si $\models A$ entonces $\models \Box A$.

Sup. $\models A$ entonces $\forall I$: $I \models A$,

Sea $I=\langle D, \gamma, \sigma\rangle$ una interpretación, entonces:

Si $0 \leq i \leq |\sigma|$ entonces

$\langle D, \gamma, \sigma^i\rangle \models A$, $\therefore M_{\sigma^i}[A]=$ true,

$\therefore \forall i$ ($0 \leq i \leq |\sigma| \supset M_{\sigma^i}[A]=$ true), $\therefore M_\sigma[\Box A]=$ true, $\therefore I \models \Box A$

$\therefore \forall I \quad I \models \Box A$, $\therefore \models \Box A$. □

Teorema 3.2- Los axiomas del SLTI son válidos.
Demostración:

a) Los axiomas A1, A2 y A3 son válidos.

Sea I=<D,γ,σ> una interpretación:

Si $M_\sigma[A]$= true entonces:

si $M_\sigma[B]$= true entonces $M_\sigma[A]$= true

$\therefore M_\sigma[(A \supset (B \supset A)]$= true, $\therefore$ I $\models$ (A$\supset$(B$\supset$A)

$\therefore \forall$I I $\models$ (A$\supset$(B$\supset$A), $\therefore \models$ (A$\supset$(B$\supset$A), $\therefore$ A1 es válido.

Las demostraciones para A2 y A3 son análogas a la de A1. □

b) Los axiomas A4 a A8 son válidos.

Sea I=<D,γ,σ> una interpretación:

Si $M_\sigma[\bigcirc (A \supset B)]$= true entonces

$M_{\sigma 1}[A]$=false o $M_{\sigma 1}[B]$= true

$\therefore$ Si $M_\sigma[\bigcirc A]$= true entonces $M_{\sigma 1}[A]$= true, $\therefore M_{\sigma 1}[B]$= true

$\therefore M_\sigma[\bigcirc B]$= true

$\therefore M_\sigma[\bigcirc A \supset \bigcirc B]$= true

$\therefore M_\sigma[\bigcirc (A \supset B) \supset (\bigcirc A \supset \bigcirc B)]$= true, $\therefore$ I $\models \bigcirc (A \supset B) \supset (\bigcirc A \supset \bigcirc B)$,

$\therefore \forall$I I $\models \bigcirc (A \supset B) \supset (\bigcirc A \supset \bigcirc B)$, $\therefore$ A4 es válido.

Las demostraciones para A5 a A8 son análogas a la de A4. □

Para demostrar que el axioma A9 es válido necesitaremos los siguientes lemas:

Lema 3.3.- Si I= <D,γ,σ> es una interpretación y e $\in$ EXP no tiene objetos locales entonces:

$\forall\tau$ ($\tau \leq \sigma \supset M_\tau[e]=M_\sigma[e]$).

Demostración: (ind. sobre la estructura de e)

Sea $\tau \leq \sigma$.

1) e=*empty* porque e no tiene objetos locales.

Si e $\in \{c_i\}$ entonces

$M_\tau[e] = \gamma(e) = M_\sigma[e]$.

2) Si e $\in$ VAR entonces e $\in$ VARG.

$\therefore (\forall i\ \sigma_1(e)=\sigma_0(e))$ y $(\exists j\ \tau_0 = \sigma_j)$, $\therefore \tau_0(e)=\sigma_0(e)$,

$M_\tau[e] = \tau_0(e) = \sigma_0(e) = M_\sigma[e]$.

3) Si $e=f(e_1,e_2,\ldots,e_n)$, entonces:
$M_\tau[e] =$
$= \gamma(f)(M_\tau[e_1],M_\tau[e_2],\ldots,M_\tau[e_n]) =$
$= \gamma(f)(M_\sigma[e_1],M_\sigma[e_2],\ldots,M_\sigma[e_n]) =$
$= M_\sigma[f(e_1,e_2,\ldots,e_n)] = M_\sigma[e].$

4) Si $e=O\,d$, entonces
$M_\tau[e]= M_{\tau_1}[d]= M_\tau[d]= M_\sigma[d]= M_{\sigma^1}[d]= M_\sigma[e]$

5) Si $e=$ *if* e_1 *then* e_2 *else* e_3, entonces
$M_\tau[e_1] = M_\sigma[e_1]$
$\therefore M_\tau[$*if* e_1 *then* e_2 *else* $e_3] =$
$M_\sigma[$*if* e_1 *then* e_2 *else* $e_3]$. □

Lema 3.4.- Si $I=\langle D,\gamma,\sigma\rangle$ es una interpretación y $w \in FORM$ no tiene objetos locales entonces:
$\forall\tau\ (\tau\leq\sigma \supset M_\tau[w]=M_\sigma[w])$.
Demostración: (ind.sobre la estructura de w)
Sea $\tau \leq \sigma$.
1) $w\neq$*empty* porque w no tiene objetos locales.
Si $w=(e_1=e_2)$ o $w=p(e_1,e_2,\ldots,e_n)$ entonces:
las e_1 tampoco tienen objetos loc., $\therefore M_\tau[e_1] = M_\sigma[e_1]$
$\therefore M_\tau[(e_1=e_2)] = M_\sigma[(e_1=e_2)]$ y
$M_\tau[p(e_1,e_2,\ldots,e_n)] = M_\sigma[p(e_1,e_2,\ldots,e_n)]$.

2) Si $w= r \wedge s$ o $w= \neg r$ entonces
$M_\tau[r]= M_\sigma[r]$ y $M_\tau[s]= M_\sigma[s]$,
$\therefore M_\tau[r\wedge s]= M_\sigma[r\wedge s]$ y $M_\tau[\neg r]= M_\sigma[\neg r]$

3) Si $w= \forall x{:}A$ entonces
Si $M_\sigma[\forall x{:}A]=$ true ent. $\forall\alpha(\alpha =_x \sigma \supset M_\alpha[A]=$ true)
sea $\beta=_x\tau$,
$\tau\leq\sigma$, $\therefore \exists k\forall i\ \tau_i= \sigma_{k+i}$,

definimos β' tal que

- $|\beta'|=|\sigma|$, y
- $\beta'_i = \beta_i$ si $0 \le i-k \le |\beta|$
  $= \sigma_i$ si $0 > i-k$ o $i-k > |\beta|$.

$\beta' \ \alpha_X \sigma$ y $\beta \le \beta'$,

$\therefore M_{\beta'}[A]=$ true, $\therefore M_\beta[A]=$ true,

$\therefore \forall\beta(\beta \ \alpha_X \tau \supset M_\beta[A]=$ true), $\therefore M_\tau[\forall x:A]=$ true .

Si $M_\sigma[\forall x:A]=$ false ent. $\exists\alpha(\alpha \ \alpha_X \sigma$ y $M_\alpha[A]=$ false)

$\tau \le \sigma$, $\therefore \exists k \forall i \ \tau_i = \sigma_{k+i}$,

sea β tal que $|\beta|=|\tau|$ y $\beta_i=\alpha_{k+i}$, ent. $\beta \ \alpha_X \tau$ y $\beta \le \alpha$,

$\therefore \beta \ \alpha_X \tau$ y $M_\beta[A]=$ false, $\therefore M_\tau[\forall x:A]=$ false.

4) Si $w = \Box r$ entonces

Si $M_\sigma[\Box r]=$ true ent. $\forall i(0 \le i \le |\sigma| \supset M_{\sigma^i}[r]=$ true)

$\tau \le \sigma$, $\therefore \exists k \forall i \ \tau^i = \sigma^{k+i}$,

$\therefore \forall i(0 \le i \le |\tau| \supset M_{\tau^i}[r]=$ true), $\therefore M_\tau[\Box r]=$ true

Si $M_\sigma[\Box r]=$ false ent. $\exists i(0 \le i \le |\sigma|$ y $M_{\sigma^i}[r]=$ false),

$\therefore M_\sigma[r]=$ false, de lo contrario $M_{\sigma^i}[r]=$ true,

$\therefore M_{\tau^0}[r]=$ false porque $\tau^0 = \tau \le \sigma$,

$\therefore M_\tau[\Box r]=$ false

5) Si $w = \bigcirc r$ entonces

$\tau^1 \le \tau \le \sigma$ y $\sigma^1 \le \sigma$,

$\therefore M_\sigma[\bigcirc r] = M_{\sigma^1}[r] = M_\sigma[r] = M_{\tau^1}[r] = M_\tau[\bigcirc r]$

6) Si $w = r;s$ entonces

Si $M_\sigma[r;s]=$ true entonces

$\exists i \ (M_{(^i\sigma)}[r] =$ true y $M_{\sigma^i}[s] =$ true),

$^i\tau \le \tau \le \sigma$, $\tau^i \le \tau \le \sigma$, $^i\sigma \le \sigma$, y $\sigma^i \le \sigma$,

$M_{(^i\tau)}[r] = M_{(^i\sigma)}[r] =$ true y $M_{\tau^i}[r] = M_{\sigma^i}[r] =$ true,

$\therefore M_\tau[r;s]=$ true

Si $M_\sigma[r;s]$= false entonces

$\forall i\ (M_{(i_\sigma)}[r] = \text{false o } M_{\sigma^1}[s] = \text{false})$,

$\therefore (M_{(0_\sigma)}[r] = \text{false o } M_{\sigma^0}[s] = \text{false})$,

$\therefore (M_\tau[r] = \text{false o } M_\tau[s] = \text{false})$,

$\therefore \forall i\ (M_{(i_\tau)}[r] = \text{false o } M_{\tau^1}[s] = \text{false})$,

$\therefore M_\tau[r;s]$ = false .

7) Si w= *if* e *then* w_1 *else* w_2, entonces

$M_\tau[e] = M_\sigma[e]$, $M_\tau[w_1] = M_\sigma[w_1]$, $M_\tau[w_2] = M_\sigma[w_2]$,

$\therefore M_\tau[$*if* e *then* w_1 *else* $w_2] =$

$= M_\sigma[$*if* e *then* w_1 *else* $w_2]$. □

c) El axioma A9 es válido:

Si A no tiene objetos locales entonces $\models A \supset \bigcirc A$.

Sea I=<D,γ,σ> una interpretación:

Si $M_\sigma[A]$= true entonces

$M_{\sigma^1}[A]$= true porque $\sigma^1 \preceq \sigma$ y A no tiene objetos locales,

$\therefore M_\sigma[\bigcirc A]=M_{\sigma^1}[A]$= true,

$\therefore M_\sigma[A \supset \bigcirc A]$= true, $\therefore$ I $\models A \supset \bigcirc A$

$\therefore \forall$I I$\models A \supset \bigcirc A$, $\therefore \models A \supset \bigcirc A$, $\therefore$ A9 es válido. □

Para demostrar que el axioma A10 es válido necesitaremos de los siguientes lemas:

Lema 3.5.- Si I= <D,γ,σ> es una interpretación, x $\in$ VAR, e,t $\in$ EXP y t no tiene objetos locales, entonces

$\forall\tau\ (\tau \preceq \sigma \supset M_{\tau[t/x]}[e] = M_\tau[e[t/x]])$

Demostración: (ind.sobre la estructura de e)

Sea $\tau \preceq \sigma$.

1) Si e $\in$ {*empty*}$\cup${c_i} entonces e[t/x] = e

además, $|\tau[t/x]| = |\tau|$

$\therefore M_{\tau[t/x]}[e] = \gamma(e) = M_\tau[e[t/x]]$ si e $\in$ {c_i}

y $M_{\tau[t/x]}[e] = M_\tau[e[t/x]]$ si e $\in$ {*empty*}.

2) Si e ∈ VAR entonces
si e≠x entonces $e[t/x] = e$ y $\tau[t/x]_0(e) = \tau_0(e)$,
$\therefore M_{\tau[t/x]}[e] = \tau[t/x]_0(e) = \tau_0(e) = M_\tau[e[t/x]]$.
si e=x entonces $e[t/x] = t$ y $\tau[t/x]_0(e) = M_{\tau_0}[t]$
$\therefore M_{\tau[t/x]}[e] = \tau[t/x]_0(e) = M_{\tau_0}[t] = M_\tau[t] = M_\tau[e[t/x]]$.

3) Si $e=f(e_1,e_2,\ldots,e_n)$, entonces:
$M_{\tau[t/x]}[e] =$
$\gamma(f)(M_{\tau[t/x]}[e_1], M_{\tau[t/x]}[e_2], \ldots, M_{\tau[t/x]}[e_n]) =$
$\gamma(f)(M_\tau[e_1[t/x]], M_\tau[e_2[t/x]], \ldots, M_\tau[e_n[t/x]]) =$
$M_\tau[f(e_1[t/x], e_2[t/x], \ldots, e_n[t/x])] = M_\tau[e[t/x]]$.

4) Si e=O d, entonces
$e[t/x] = O\,(d[t/x])$ porque t no tiene objetos locales.
$\therefore M_{\tau[t/x]}[e] = M_{\tau[t/x]}[O\ d] = M_{\tau[t/x]^1}[d] =$
$= M_{\tau^1[t/x]}[d] = M_{\tau_1}[d[t/x]] =$
$= M_\tau[O\,(d[t/x])] = M_\tau[e[t/x]]$.

5) Si e= *if* e_1 *then* e_2 *else* e_3, entonces
$M_{\tau[t/x]}[e_1] = M_\tau[e_1[t/x]]$
$\therefore M_{\tau[t/x]}[$*if* e_1 *then* e_2 *else* $e_3] =$
$= M_\tau[($*if* e_1 *then* e_2 *else* $e_3)[t/x]]$. □

Lema 3.6.- Si $I=<D,\gamma,\sigma>$ es una interpretación, $w \in FORM$, $x \in VAR$, $t \in EXP$ y t no tiene objetos locales, entonces
$\forall\tau\ (\tau\leq\sigma \supset M_{\tau[t/x]}[w] = M_\tau[w[t/x]])$
Demostración: (ind.sobre la estructura de w)
Sea $\tau \leq \sigma$.
1) Si w= *empty* entonces $|\tau[t/x]| = |\tau|$,
$\therefore M_{\tau[t/x]}[$*empty*$] = M_\tau[$*empty*$[t/x]]$.
Si $w= (e_1=e_2)$ o $w= p(e_1,e_2,\ldots,e_n)$ entonces:
$M_{\tau[t/x]}[e_1] = M_\tau[e_1[t/x]]$ (t no tiene obj. locales),
$\therefore M_{\tau[t/x]}[(e_1=e_2)] = M_\tau[(e_1=e_2)[t/x]]$ y
$M_{\tau[t/x]}[p(e_1,e_2,\ldots,e_n)] = M_\tau[p(e_1,e_2,\ldots,e_n)[t/x]]$.

2) Si $w = r \wedge s$ o $w = \neg r$ entonces
$M_\tau[t/x][r] = M_\tau[r[t/x]]$ y $M_\tau[t/x][s] = M_\tau[s[t/x]]$,
$\therefore M_\tau[t/x][r \wedge s] = M_\tau[r \wedge s[t/x]]$ y
$M_\tau[t/x][\neg r] = M_\tau[\neg r[t/x]]$.

3) Si $w = \forall v:A$ entonces
si $x=v$ ent. $\forall v:A\ [t/x] = \forall v:A$, $\tau[t/x] = \tau[t/v] \sim_v \tau$,
$\therefore \xi \sim_v \tau[t/x]$ sii $\xi \sim_v \tau$,
$\therefore M_{\tau[t/x]}[\forall v:A] =$ true sii
$\forall \xi(\xi \sim_v \tau[t/x] \supset M_\xi[A] =$ true) sii
$\forall \xi(\xi \sim_v \tau \supset M_\xi[A] =$ true) sii
$M_\tau[\forall v:A] =$ true sii $M_\tau[\forall v:A[t/x]] =$ true.
si $x \neq v$ y (v no ocurre en t) entonces
$(\forall v:A)[t/x] = \forall v:(A[t/x])$,
$\therefore M_\tau[(\forall v:A)[t/x]] =$ true sii
$\forall \xi(\xi \sim_v \tau \supset M_\xi[A[t/x]] =$ true) sii (h. inducción)
$\forall \xi(\xi \sim_v \tau \supset M_{\xi[t/x]}[A] =$ true). sii (ver abajo)
$\forall \xi(\xi \sim_v \tau[t/x] \supset M_\xi[A] =$ true). sii
$\therefore M_{\tau[t/x]}[\forall v:A] =$ true.

probaremos que
$\forall \xi(\xi \sim_v \tau \supset M_{\xi[t/x]}[A] =$ true) sii
$\forall \xi(\xi \sim_v \tau[t/x] \supset M_\xi[A] =$ true).
$\longrightarrow$) si $\xi \sim_v \tau[t/x]$ entonces
definimos ξ' tal que $|\xi'| = |\xi|$ y
$\xi_1'(h) = \tau_1(h)$ si $h=x$,
$= \xi_1(h)$ si $h \neq x$.
$\xi' \sim_v \tau$ porque
si $h \neq v$ ent.
$\xi_1(h) = \tau_1(h)$ si $h=x$
$= \xi_1(h) = \tau[t/x]_1(h) = \tau_1(h)$ si $h \neq x$.
$\xi'[t/x] = \xi$ porque
v no ocurre en t y $\xi' \sim_v \tau$,
$\therefore M_{\xi'_1}[t] = M_{\tau_1}[t]$,
$\therefore$ si $h=x$, $\xi'[t/x]_1(h) = M_{\xi'_1}[t] = M_{\tau_1}[t]$
$= \tau[t/x]_1(h) = \xi_1(h)$
si $h \neq x$ $\xi'[t/x]_1(h) = \xi'_1(h) = \xi_1(h)$.

$\therefore M_{\xi}[A] = M_{\xi'[t/x]}[A] = \text{true}.$

$\longleftarrow$) si $\xi \approx_v \tau$ entonces

v no ocurre en t, $\therefore M_{\xi_1}[t] = M_{\tau_1}[t]$.

$\therefore \xi[t/x] \approx_v \tau[t/x]$, porque

si $h \neq v$ entonces:

si $h \neq x$, $\xi[t/x]_1(h) = \xi_1(h) = \tau_1(h) = \tau[t/x]_1(h)$.

si $h = x$, $\xi[t/x]_1(h) = M_{\xi_1}[t] = M_{\tau_1}[t] = \tau[t/x]_1(h)$.

$\therefore M_{\xi[t/x]}[A] = \text{true}.$

4) Si $w \models \Box r$ entonces

$(\Box r)[t/x] = \Box (r[t/x])$ (t no tiene objetos locales).

$M_{\tau[t/x]}[\Box r] = \text{true}$ sii $\forall i (M_{\tau[t/x]_1}[r] = \text{true})$ sii

$\forall i (M_{\tau_1[t/x]}[r] = \text{true})$ sii $\forall i (M_{\tau_1}[r[t/x]] = \text{true})$ sii

$M_{\tau}[\Box (r[t/x])] = \text{true}.$

5) Si $w \models \bigcirc r$ entonces

$(\bigcirc r)[t/x] = \bigcirc (r[t/x])$ (t no tiene objetos locales).

$M_{\tau[t/x]}[\bigcirc r] = \text{true}$ sii $M_{\tau[t/x]_1}[r] = \text{true}$ sii

$M_{\tau_1[t/x]}[r] = \text{true}$ sii $M_{\tau_1}[r[t/x]] = \text{true}$ sii

$M_{\tau}[\bigcirc (r[t/x])] = \text{true}.$

6) Si $w \models r;s$ entonces

$(r;s)[t/x] = (r[t/x]);(s[t/x])$ (t no tiene obj. locales.)

$M_{\tau[t/x]}[r;s] = \text{true}$ sii

$\exists i (M_{i_{(\tau[t/x])}}[r] = \text{true}$ y $M_{\tau[t/x]_1}[s] = \text{true})$ sii

$\exists i (M_{i_{(\tau)}}[t/x][r] = \text{true}$ y $M_{\tau_1[t/x]}[s] = \text{true})$ sii

$\exists i (M_{i_{\tau}}[r[t/x]] = \text{true}$ y $M_{\tau_1}[s[t/x]] = \text{true})$ sii

$M_{\tau}[(r[t/x]);(s[t/x])] = \text{true}.$

7) Si w= *if* e *then* w_1 *else* w_2, entonces
$M_{\tau[t/x]}[e] = M_\tau[e[t/x]]$ y $M_{\tau[t/x]}[w_1] = M_\tau[w_1[t/x]]$,
$\therefore M_{\tau[t/x]}[$*if* e *then* w_1 *else* $w_2] =$
$M_\tau[($*if* e *then* w_1 *else* $w_2)[t/x]]$. □

d) El axioma A10 es válido:

Si t no tiene objetos locales entonces $\models \forall x{:}A \supset A[t/x]$.

Sup. que t no tiene objetos locales y sea $I=<D,\mathcal{I},\sigma>$ una interpretación:

Si $I\models \forall x{:}A$, entonces $M_\sigma[\forall x{:}A]$= true,
$\therefore \forall \tau\ (\tau \approx_x \sigma \supset M_\tau[A]$= true)
$\therefore M_{\sigma[t/x]}[A]$= true porque $\sigma[t/x] \approx_x \sigma$
como t no tiene objetos locales,

$M_\sigma[A[t/x]] = M_{\sigma[t/x]}[A]$= true,

$\therefore I\models A[t/x]$

$\therefore$ si $I\models \forall x{:}A$ entonces $I\models A[t/x]$,

$\therefore I\models (\forall x{:}A) \supset (A[t/x])$,

$\therefore \forall I\ (I\models (\forall x{:}A) \supset (A[t/x]))$, $\therefore \models (\forall x{:}A) \supset (A[t/x])$. □

Es importante observar que el axioma A10 en general no es válido si t tiene objetos locales. Para esto consideremos un ejemplo:

Sean b ∈ VARG y B ∈ VARL, tomemos
t = B, y A = (b=1 ⊃ ○ b = 1).
Sea w una instancia de A10 tal que
w = (∀b: A) ⊃ (A [t/x]) =
= (∀b: (b=1 ⊃ ○ b =1)) ⊃ ((b=1 ⊃ ○ b =1)[t/x]) =
= (∀b: (b=1 ⊃ ○ b =1)) ⊃ ((B=1 ⊃ ○ b =1))
si σ es el intervalo tal que
$\sigma_i(x)$ = 1 si i= 0 ∧ x= B
= 2 si no
entonces $M_\sigma[(B=1 \supset \bigcirc b =1)]$ = false y
$\forall\ \tau \approx_b \sigma\ M_\tau[(b=1 \supset \bigcirc b =1)]$ = true,
$\therefore M_\sigma[\forall b: (b=1 \supset \bigcirc b =1)]$ = true.
$\therefore M_\sigma[w]$ = false

e) El axioma A11 es válido:

Si x no ocurre libre en A entonces

$\models (\forall x:(A \supset B)) \supset (A \supset \forall x:B)$.

Sup. que x no ocurre libre en A y sea $I=<D,\gamma,\sigma>$ una interpretación:

Si $M_\sigma[\forall x:(A \supset B)]$= true entonces

$\forall\sigma'(\sigma' \approx_x \sigma \supset M_{\sigma'}[A \supset B]$= true).

Si $M_\sigma[A]$= true entonces

si $\tau \approx_x \sigma$ entonces $M_\tau[A \supset B]$= true,

$\therefore M_\tau[A]$=false o $M_\tau[B]$= true,

pero $M_\sigma[A]$= true, $\tau \approx_x \sigma$ y x no ocurre libre en A,

$\therefore M_\tau[A]$= true,

$\therefore M_\tau[B]$= true,

$\therefore \forall\tau\ (\tau \approx_x \sigma \supset M_\tau[B]$= true), $\therefore M_\sigma[\forall x: B]$= true,

$\therefore M_\tau[A \supset \forall x: B]$= true,

$\therefore M_\sigma[(\forall x:(A \supset B)) \supset (A \supset \forall x:B)]$= true,

$\therefore \forall I\ (I \models (\forall x:(A \supset B)) \supset (A \supset \forall x:B))$,

$\therefore \models (\forall x:(A \supset B)) \supset (A \supset \forall x:B)$. □

f) El axioma A12 es válido:

$\models (\forall x: \bigcirc A) \supset (\bigcirc \forall x: A)$.

Sea $I=<D,\gamma,\sigma>$ una interpretación:

Si $M_\sigma[\forall x: \bigcirc A]$= true entonces $\forall\sigma'\ (\sigma' \approx_x \sigma \supset M_{\sigma'}[\bigcirc A]$= true),

sea $\tau \approx_x \sigma^1$, definimos $\tau'=<\sigma_0\tau_0\tau_1\ldots\tau_{|\tau|}>$. Entonces:

claramente $(\tau')^1=\tau$ y $\tau' \approx_x \sigma$,

$\therefore M_{\tau'}[\bigcirc A]$= true, $\therefore M_\tau[A]= M_{(\tau')^1}[A]$= true,

$\therefore \forall\tau\ (\tau \approx_x \sigma^1 \supset M_\tau[A]$= true),

$\therefore M_{\sigma^1}[\forall x: A]$= true, $\therefore M_\sigma[\bigcirc \forall x: A]$= true,

$\therefore M_\sigma[(\forall x: \bigcirc A) \supset (\bigcirc \forall x: A)]$= true,

$\therefore \forall I\ (I \models (\forall x: \bigcirc A) \supset (\bigcirc \forall x: A))$,

$\therefore \models (\forall x: \bigcirc A) \supset (\bigcirc \forall x: A)$. □

g) El axioma A13 es válido.

Sea $I=<D,\gamma,\sigma>$ una interpretación:

$M_\sigma[x] = M_\sigma[x]$, $\therefore M_\sigma[x=x] = true$,
$\therefore \forall I \ (I \models x = x)$,
$\therefore \models x = x$. □

Para demostrar la validez del axioma A14 necesitaremos el siguiente resultado:

Lema 3.7.- Si $I = \langle D, \gamma, \sigma \rangle$ es una interpretación, $\psi \in EXP \cup FORM$ no tiene operadores temporales, $x,z,v \in VAR$, $x \neq v$, $z \neq v$ y $M_\sigma[x] = M_\sigma[z]$, entonces

$\forall \alpha \ (\alpha \approx_v \sigma \supset M_\alpha[\psi] = M_\alpha[\psi\ [z/x]])$.

Omitimos la demostración de este lema que es similar a las demostraciones de los lemas 3.3 y 3.4.

h) El axioma A14 es válido.

Sup. que A no tiene operadores temporales,
Sea $I = \langle D, \gamma, \sigma \rangle$ una interpretación:
Si $M_\sigma[x=z] = true$ entonces
$M_\sigma[x] = M_\sigma[z]$,
Sea $v \in VAR$ tal que $x \neq v$ y $z \neq v$,
$\therefore$, por el lema 3.7, $\forall \alpha \ (\alpha \approx_v \sigma \supset M_\alpha[A] = M_\alpha[A[z/x]])$,
$\therefore M_\sigma[A] = M_\sigma[A[z/x]]$
$\therefore M_\sigma[\ A \supset A[z/x]\ ] = true$
$\therefore M_\sigma[\ x=z \supset (A \supset A[z/x])\ ] = true$
$\forall I \ (I \models x=z \supset (A \supset A[z/x]))$
$\therefore$ Si A no tiene operadores temporales entonces
$\models x=z \supset (A \supset A[z/x])$. □

i) Los axiomas A15 a A19 son válidos.

Sea $I = \langle D, \gamma, \sigma \rangle$ una interpretación:
Si $M_\sigma[empty \wedge A] = true$ entonces:
$M_\sigma[empty] = true$ y $M_\sigma[A] = true$,
$\therefore |\sigma| = 0$ y $M_\sigma[\bigcirc A] = M_\sigma[A] = true$,
$\therefore M_\sigma[(empty \wedge A) \supset \bigcirc A] = true$, $\therefore I \models (empty \wedge A) \supset \bigcirc A$
$\therefore \forall I \ I \models (empty \wedge A) \supset \bigcirc A$,
$\therefore \models (empty \wedge A) \supset \bigcirc A$, $\therefore$ A15 es válido.

Las demostraciones para los axiomas A16 a A19 son análogas a la de A15. □

TEORIA TEMPORAL DE INTERVALOS

En este capítulo presentamos una teoría temporal de intervalos sobre un dominio específico. Esta teoría servirá más adelante como base para la definición del lenguaje Tempura. La definición del dominio se hace a través del concepto de estructura relacional y la teoría temporal se define como una extensión de la LTI distinguiendo algunos símbolos y agregando algunas expresiones y fórmulas. Estas ideas se precisan a continuación.

2.1 Alfabeto

En la definición del alfabeto de la teoría temporal de intervalos será necesario distinguir varios símbolos de constante y símbolos de función. Se requiere distinguir un símbolo de constante global por cada elemento del dominio que nos interesa y un símbolo de función por cada una de ciertas funciones sobre este dominio. Para lograr este objetivo, se define a continuación una estructura relacional que contiene el dominio, las constantes, las funciones y los predicados que nos interesan.

Una *estructura relacional* es una cuarteta ordenada de la forma
$<B, \{f_i\}_{i\in I}, \{p_j\}_{j\in J}, \{c_k\}_{k\in K}>$
con funciones asociadas $\lambda: I \longrightarrow \mathbb{N}$ y $\mu: J \longrightarrow \mathbb{N}$ tales que:

- B, *la base o dominio de la estructura*, es un conjunto no vacío,
- I es un conjunto tal que $\forall\, i\in I$ f_i es una función de aridad $\lambda(i)$, $f_i: B^{\lambda(i)} \longrightarrow B$,
- J es un conjunto tal que $\forall\, j\in J$ p_j es un predicado de aridad $\mu(j)$, $p_j: B^{\mu(j)} \longrightarrow \{true, false\}$ y
- K es un conjunto tal que $\forall\, k\in K$ $c_k \in B$.

Si A y B son dos conjuntos, la *suma* de A y B, A+B, es la clase $A+B = \{(a,1)/\ a\in A\} \cup \{(b,2)/\ b\in B\}$.

Observación: Abusando de la notación, identificaremos al

conjunto A con el subconjunto de A+B $\{(a,1)/\ a\in A\}$ de tal forma que si $x\in A+B$, escribiremos "$x\in A$" en lugar de "$x\in\{(a,1)/\ a\in A\}$". Más aún, si $x\in(A+B)^n$, escribiremos "$x\in A^n$" en lugar de "$x\in(\{(a,1)/\ a\in A\})^n$". Esta situación se extiende en forma análoga para B.

Si *A* y *B* son dos estructuras,
$A=\langle A,\{f_l\}_{l\in I},\{p_j\}_{j\in J},\{c_k\}_{k\in K}\rangle$ y $B=\langle B,\{g_l\}_{l\in L},\{q_m\}_{m\in M},\{d_n\}_{n\in N}\rangle$
con funciones asociadas λ_A, μ_A, y λ_B, μ_B respectivamente,
la *suma* de *A* y *B*, *A+B*, es la estructura
$A+B=\langle A+B\cup\{\perp\},\{h_l\}_{l\in I+L},\{r_j\}_{j\in J+M},\{e_k\}_{k\in K+N}\rangle$
(con funciones asociadas $\lambda: I+L \longrightarrow \mathbb{N}$, y $\mu: J+M \longrightarrow \mathbb{N}$) que cumple lo que sigue:

- $\lambda(x) = \lambda_A(x)$ si $x\in A$
  $= \lambda_B(x)$ si $x\in B$
  $= \perp$ si no.
- $\mu(x) = \mu_A(x)$ si $x\in A$
  $= \mu_B(x)$ si $x\in B$
  $= \perp$ si no.
- $h_l: (A+B\cup\{\perp\})^{\lambda(l)} \longrightarrow A+B\cup\{\perp\}$

  $h_l(x) = f_l(x)$ si $l\in I$ y $x\in A^{\lambda_A(l)}$
  $= g_l(x)$ si $l\in L$ y $x\in B^{\lambda_B(l)}$
  $= \perp$ si no
- $r_l: (A+B\cup\{\perp\})^{\mu(l)} \longrightarrow \{true, false\}$

  $r_l(x) = p_l(x)$ si $l\in J$ y $x\in A^{\mu_A(l)}$
  $= q_l(x)$ si $l\in M$ y $x\in B^{\mu_B(l)}$
  $=$ false si no

Las estructuras que describimos a continuación sirven para determinar el dominio que permitirá definir, con base en la LTI, la teoría temporal de intervalos que dará el respaldo semántico a Tempura:

- **Números racionales**
  $Q = \langle \mathbb{Q}, \{+,-,*,/,\hat{},\text{mod},\text{div}\}, \{=,\leq,<,\geq,>\}, \{c/\ c\in\mathbb{Q}\}\rangle$

donde

- $\mathbb{Q}$ es el conjunto de los números racionales;
- $+,-,*,/,\hat{},\text{mod},\text{div}$ son respectivamente las operaciones de suma, diferencia, producto, división, exponenciación, residuo entero y cociente entero;
- $=,\leq,<,\geq,>$ son las relaciones de igualdad y orden en $\mathbb{Q}$.

- **Valores de verdad**

$\mathcal{B} = < \mathbb{B}, \{\text{and, or, not, imp}\}, \{=\}, \{c/\ c \in \mathbb{B}\} >$

donde

- $\mathbb{B} = \{\text{True, False}\}$ es el conjunto de valores de verdad;
- and, or, not, imp, son respectivamente las funciones de conjunción, disyunción, negación e implicación.
- $=$ es la relación de igualdad en $\mathbb{B}$.

- **Cadenas (strings)**

$\mathcal{S} = < S, \{+\}, \{=,\leq,<,\geq,>\}, \{c/\ c \in S\} >$

donde

- S es el conjunto de secuencias finitas de caracteres imprimibles del código ASCII, $S \subseteq \text{ASCII}^*$;
- $+$ es la operación de concatenación de cadenas.
- $=,\leq,<,\geq,>$ son las relaciones de igualdad y orden en S.

- **Tipos**

$\mathcal{T} = < \mathbb{T}, \{\}, (=), \{c/\ c \in \mathbb{T}\} >$

donde

- $\mathbb{T} = \{$ TNumber, TBoolean, TString, TType, TList $\}$.
- $=$ es la relación de igualdad en $\mathbb{T}$.

- **Listas**

- $\mathcal{L} = < \mathbb{L}, \{\#, \text{Car}, \text{Cdr}\}, \{\}, \{c/\ c \in \mathbb{L}\} >$

donde

- $\mathbb{L}$, el conjunto de listas finitas ordenadas, está definido como la clase más pequeña que cumple:

a) $\{\} \in \mathbb{L}$.

b) Si $n \geq 0$ y $e_0, e_1, \ldots e_{n-1} \in \mathbb{L} \cup \mathbb{Q} \cup \mathbb{B} \cup S \cup \mathbb{T}$ entonces $\{e_0, e_1, \ldots, e_{n-1}\} \in \mathbb{L}$.

(si $n<k$ consideramos $\{e_k, e_{k+1}, \ldots, e_n\} = \{\}$.)

- # es la función de longitud (cardinalidad) de listas:
  # {} = 0;
  si n ≥ 0 entonces # $\{e_0, e_1, \ldots, e_{n-1}\}$ = n.

- Car es la función que entrega el primer elemento de una lista:
  $Car(\{e_0, e_1, \ldots, e_{n-1}\}) = e_0$ si n>0

- Cdr es la función que entrega la lista que resulta de eliminar el primer elemento de una lista:
  $Cdr(\{e_0, e_1, \ldots, e_{n-1}\}) = \{e_1, \ldots, e_{n-1}\}$.

Definimos la *estructura de los valores básicos*, *Bv*, como la estructura que resulta al modificar $Q \cup B \cup S \cup T \cup L$ de la siguiente manera:

- La *base* de *Bv* es la misma que la de $Q \cup B \cup S \cup T \cup L$ y la llamamos conjunto de *valores básicos*, Bv, es decir,
  Bv = $\mathbb{Q} \cup \mathbb{B} \cup \mathbb{S} \cup \mathbb{T} \cup \mathbb{L}$.
- Las constantes de *Bv* son las mismas que las de $Q \cup B \cup S \cup T \cup L$
- Agregamos a la lista de funciones todos los predicados de $Q \cup B \cup S \cup T \cup L$. Esto es posible porque $\mathbb{B} \subseteq$ Bv.
- Eliminamos todos los predicados de $Q \cup B \cup S \cup T \cup L$ excepto el (los) de igualdad.
- Agregamos el predicado List y las funciones TypeOf, Elem, SubL y Cons que se definen a continuación:

TypeOf es la función que entrega el tipo de valor de una lista:

TypeOf(x)	= TNumber	si $x \in \mathbb{Q}$,
	= TBoolean	si $x \in \mathbb{B}$,
	= TString	si $x \in \mathbb{S}$,
	= TList	si $x \in \mathbb{L}$,
	= TType	si $x \in \mathbb{T}$.

Elem es la función que entrega el elemento i-esimo de una lista:

Elem(x,i)= e_i si $x \in \mathbb{L}$, $i \in \mathbb{N}$ y
$\exists$ n>0 ($x = \{e_0, e_1, \ldots, e_{n-1}\}$ y $0 \leq i \leq n-1$)
usamos e[d] en lugar de elem(e,d).

SubL es la función que entrega una sublista de una lista :
SubL(x,i,j)= $\{x[i], x[i+1], \ldots, x[j-1]\}$ si $x \in \mathbb{L}$ y $i, j \in \mathbb{N}$.
usamos x[i:j] en lugar de SubL(x,i,j)

Cons es la función que agrega un elemento a una lista :
Cons(i,x)= $\{i, e_0, e_1, \ldots, e_{n-1}\}$ si $x=\{e_0, e_1, \ldots, e_{n-1}\} \in \mathbb{L}$

List es la relación que indica si x es una lista de longitud e:
List(x,e) = true si $x \in \mathbb{L}$, $e \in \mathbb{N}$ y (# x) = e,
= false si no.

La estructura de los valores básicos contiene las constantes, funciones y predicados que nos interesa distinguir para definir la teoría temporal que servirá de respaldo a Tempura. A continuación definimos esta teoría especificando su alfabeto, sus expresiones y sus fórmulas:

Definimos el *alfabeto* de la *Teoría Temporal de Intervalos*, TTI, como el conjunto que contiene los siguientes símbolos:
- Los símbolos del alfabeto de la LTI
- Los símbolos { / } *len*

Es fácil ver que en la estructura de los valores básicos se cumple que:
- El conjunto de constantes es numerable,
- El conjunto de funciones es finito,
- El conjunto de predicados es finito.

Debido a esto, es posible *distinguir* los siguientes símbolos de la TTI:
- Por cada constante c_k de *Bv distinguimos* un símbolo de constante c_1.
- Por cada función f_i de aridad j de *Bv distinguimos* un símbolo de función f_i^j.

Observación: El único predicado de *Bv* es el de igualdad y este predicado no necesita de ningún símbolo de predicado que lo distinga porque ya se cuenta con el símbolo = que se interpreta como la igualdad.

Para simplificar la notación y hacer que las fórmulas sean más legibles, identificaremos cada símbolo distinguido con el elemento (constante, función) al cual distingue y usaremos formalmente al elemento mismo en lugar del símbolo que lo distingue. Más aún, en caso de que alguna función se use informalmente con notación infija, también se usará formalmente con dicha notación. Por ejemplo:

- Si el símbolo c distingue a la constante 3, escribiremos 3 en lugar de c.
- Si el símbolo c distingue a la constante $\{1,\{7,8\}\}$, escribiremos $\{1,\{7,8\}\}$ en lugar de c.
- Si el símbolo f distingue a la función +, escribiremos e_1+e_2 en lugar de $f(e_1,e_2)$.
- Si el símbolo f distingue a la función $\leq$, escribiremos $e_1 \leq e_2$ en lugar de $f(e_1,e_2)$.

Hasta ahora, la interpretación de los símbolos de la LTI (constantes, variables, predicados y funciones) sólo necesita cumplir algunas condiciones generales. Las constantes y las variables se interpretan como elementos del dominio y los predicados y funciones se interpretan como predicados y funciones de aridad apropiada. Para la TTI nos interesan las interpretaciones en las que los símbolos distinguidos se interpretan como el elemento (constante, función) al cual distinguen.

Decimos que $J=\langle B,\gamma,\sigma\rangle$ es una *interpretación de la TTI*, si J es una interpretación que cumple:

- $Bv \subseteq B$ y
- $\gamma(s) = s$ para cada símbolo distinguido s.

2.2 Expresiones y fórmulas

En esta sección mostramos de que manera se extiende la LTI para dar lugar a la TTI. A las definiciones formales les precede una explicación intuitiva.

La clase de expresiones se extiende con expresiones que permiten denotar listas finitas ($\{e_1: e_2 \leq u < e_3\}$) cuyos elementos se obtienen mediante el valor de una expresión (e_1) parametrizada por los valores enteros de una variable global (u) en un rango dado ($[e_2,e_3)$).

La clase de fórmulas se extiende con fórmulas que permiten expresar que una fórmula debe ejecutarse (ser cierta) *consecutivamente* un número finito de veces:

- La fórmula (*for* e *times do* w) expresa que w debe ejecutarse consecutivamente e veces.
- La fórmula (*for* $e_1 \leq u < e_2$ *do* w) expresa que deben ejecutarse consecutivamente las fórmulas obtenidas de w al usar como parámetro a la variable u que toma en forma creciente los valores enteros del rango $[e_1,e_2)$.
- La fórmula (*for* u *in* e *do* w) expresa que deben ejecutarse consecutivamente las fórmulas obtenidas de w al usar como parámetro a la variable u que toma como valor a los elementos de la lista e en forma creciente.

Además de esto, la clase de fórmulas se extiende con la fórmula *len*(e) que permite expresar que la longitud del intervalo debe ser e.

La clase de *expresiones de la TTI*, EXPD, es la mínima clase que cumple:

a) $\{c_i\} \cup VAR \subseteq EXPD$.
b) Si $e_1,e_2,\ldots,e_k \in EXPD$ ($k \geq 0$) y f es un símbolo de función de aridad k entonces $f(e_1,e_2,\ldots,e_k) \in EXPD$.
c) *empty* $\in$ EXPD.

d) Si e ∈ EXPD entonces (○ e) ∈ EXPD.
e) Si e_1, e_2, e_3 ∈ EXPD, (*if* e_1 *then* e_2 *else* e_3) ∈ EXPD.

f) Si e_1, e_2, e_3 ∈ EXPD y u ∈ VARG entonces:
{e_1: $e_2 \leq u < e_3$} ∈ EXPD.

Claramente, EXP ⊆ EXPD.

La clase de *fórmulas de la TTT*, FORMD, es la mínima clase que cumple:

a) Si w ∈ FORMD entonces (¬ w) ∈ FORMD.
b) Si r,s ∈ FORMD entonces (r ∧ s) ∈ FORMD.
c) Si $e_1, e_2, \ldots, e_k$ ∈ EXPD (k≥0) y p es un símbolo de predicado de aridad k entonces $p(e_1, e_2, \ldots, e_k)$ ∈ FORMD.

d) Si w ∈ FORMD y v ∈ VAR entonces (∀ v: w) ∈ FORMD.
e) Si e_2, e_2 ∈ EXPD entonces ($e_1 = e_2$) ∈ FORMD.

f) *empty* ∈ FORMD.
g) Si w ∈ FORMD entonces (□ w) ∈ FORMD.
h) Si w ∈ FORMD entonces (○ w) ∈ FORMD.

i) Si r,s ∈ FORMD entonces (r ; s) ∈ FORMD.
j) Si e ∈ EXPD y w_1, w_2 ∈ FORMD, entonces
(*if* e *then* w_1 *else* w_2) ∈ FORMD

k) Si e ∈ EXPD entonces *len*(e) ∈ FORMD.
l) Si e ∈ EXPD y w ∈ FORMD entonces:
(*for* e *times do* w) ∈ FORMD.
m) Si e_1, e_2 ∈ EXPD, u ∈ VARG y w ∈ FORMD entonces:
(*for* $e_1 \leq u < e_2$ *do* w) ∈ FORMD.
n) Si e ∈ EXPD, u ∈ VARG y w ∈ FORMD entonces:
(*for* u *in* e *do* w) ∈ FORMD.

Claramente, FORM ⊆ FORMD

La lectura de estas nuevas expresiones y fórmulas de la TTI se hace como sigue:

Elemento	Lectura
- $\{e_1: e_2 \leq u < e_3\}$	la lista de las e_1 tales que u está entre e_2 y e_3.
- *len*(e)	longitud e.
- (*for* e *times do* w)	por e veces haz w.
- (*for* $u_1 \leq u < e_2$ *do* w)	para u entre e_1 y e_2 haz w.
- (*for* u *in* e *do* w)	para u en la lista e haz w.

La semántica de la LTI se extiende a la TTI como sigue:

Si I=<D,γ,σ> es una interpretación de la TTI, la *semántica asociada* a I es la función

M_I: (EXPD $\cup$ FORMD) $\longrightarrow$ D

que cumple las siguientes reglas:

(Escribimos $M_\sigma[x]$ en lugar de $M_I(x)$ considerando que D y γ están fijos. las c_1 son constantes, v es una variable, u es una variable local, f es una función, p es un predicado, las e_1 son expresiones y las w_1 son fórmulas)

- $M_\sigma[c] = \gamma(c)$.
- $M_\sigma[empty] =$ true sii $|\sigma|=0$. ($empty \in$ EXPD $\cup$ FORMD).
- $M_\sigma[v] = \sigma_0[v]$.
- $M_\sigma[f(e_1,\ldots,e_k)] = \gamma(f)(M_\sigma[e_1],\ldots,M_\sigma[e_k])$.
- $M_\sigma[O\ e] = M_{\sigma^1}[e]$.
- $M_\sigma[$*if* e_1 *then* e_2 *else* $e_3]$ $= M_\sigma[e_2]$ si $M_\sigma[e_1]$ = true, $= M_\sigma[e_3]$ si $M_\sigma[e_1] \neq$ true.
- $M_\sigma[p(e_1,\ldots,e_k)] = \gamma(p)(M_\sigma[e_1],\ldots,M_\sigma[e_k])$.
- $M_\sigma[e_1 = e_2] =$ true sii $M_\sigma[e_1] = M_\sigma[e_2]$.
- $M_\sigma[\neg\ w] =$ true sii $M_\sigma[w] =$ false.
- $M_\sigma[w_1 \wedge w_2] =$ true sii $M_\sigma[w_1]$=true y $M_\sigma[w_2]$=true.
- $M_\sigma[O\ w] =$ true sii $M_{\sigma^1}[w]$=true.
- $M_\sigma[\forall$ v: w] = true sii
  $M_{\sigma'}[w]$ =true para toda $\sigma' \in \Sigma_D^+$ tal que $\sigma' \approx_v \sigma$.

- $M_\sigma[\Box w]$ = true sii $\forall i \in \mathbb{N}$: $i \leq |\sigma| \supset M_{\sigma_i}[w]$=true
- $M_\sigma[w_1; w_2]$ = true sii existe $i \in \mathbb{N}$ tal que $i \leq |\sigma|$, $M_{\sigma_i}[w_1]$=true y $M_{\sigma_i}[w_2]$=true.
- $M_\sigma[$*if* e *then* w_1 *else* $w_2]$ = $M_\sigma[w_1]$ si $M_\sigma[e]$ = true,
  = $M_\sigma[w_2]$ si $M_\sigma[e] \neq$ true.

- $M_\sigma[\{e_1: e_2 \leq u < e_3\}]$ =
  = nil si $c_3 - c_2 \leq 0$
  = Cons(c_1, $M_\sigma[\{e_1: c_2+1 \leq u < c_3\}]$) si no
  donde $c_1 = M_\sigma[e_1[c_2/u]]$, $c_2 = M_\sigma[e_2]$, $c_3 = M_\sigma[e_3]$.
- $M_\sigma[$*len*(e)$]$ = true sii $M_\sigma[e] \in \mathbb{N}$ y $|\sigma| = M_\sigma[e]$
- $M_\sigma[$*for* e *times do* w$]$ =
  = $M_\sigma[$*empty*$]$ si $c \leq 0$.
  = $M_\sigma[w$; *for* c-1 *times do* w$]$ si no.
  donde $c = M_\sigma[e]$.
- $M_\sigma[$*for* $e_1 \leq u < e_2$ *do* w$]$=
  = $M_\sigma[$*empty*$]$ si $c_1 \geq c_2$,
  = $M_\sigma[(\exists u: (u=c_1 \wedge w))$;
  (*for* $c_1+1 \leq u < c_2$ *do* w)$]$ si $c_1 < c_2$.
  donde $c_1 = M_\sigma[e_1]$ y $c_2 = M_\sigma[e_2]$.
- $M_\sigma[$*for* u *in* e *do* w$]$ =
  = $M_\sigma[$*empty*$]$ si $c = \{\}$
  = $M_\sigma[(\exists u: (u=$Car(c) $\wedge$ w));
  (*for* u *in* Cdr(c) *do* w)$]$ si $c \neq \{\}$.
  donde $c = M_\sigma[e]$.

Cabe hacer notar que la construcción *len*(e) no puede incluirse mediante la distinción de un símbolo de predicado. Si $J=<D, \gamma, \sigma>$ es una interpretación, el significado de los predicados, como símbolos globales, se obtiene mediante γ que no depende ni de σ ni de los estados de σ. Contrariamente, *len* requiere que su significado dependa de σ y por lo tanto su significado no puede definirse mediante γ. Puede decirse que de alguna manera *len* es un predicado local.

2.3 Abreviaciones

Las nuevas construcciones de TTI permiten definir las siguientes abreviaciones:

Si $e, e_1 \in$ EXPD, $u \in$ VARG y $w, w_1 \in$ FORMD entonces:

- $\{e_1: u < e_2\}$ $\equiv_{def}$ $\{e_1: 0 \leq u < e_2\}$.
- $(\forall\, e_1 \leq u < e_2:\ w)$ $\equiv_{def}$ $\forall u: (if\ (e_1 \leq u \text{ AND } u < e_2)\ then\ w)$.
- $(\forall\, u < e:\ w)$ $\equiv_{def}$ $\forall u: (if\ (0 \leq u \text{ AND } u < e)\ then\ w)$.
- e_1 *gets* e_2 $\equiv_{def}$ $\Box\ (if$ more $then\ ((O\ e_1) = e_2))$.
- *stable* e $\equiv_{def}$ e *gets* e.
- $e_1 \approx e_2$ $\equiv_{def}$ $\Box\ (e_1 = e_2)$.
- $e_1 \longleftarrow e_2$ $\equiv_{def}$ $\exists\ z: ((z = e_2) \wedge fin\ (e_1 = z))$.
  donde:
  $z \in$ VARG y z no ocurre en e_1 ni en e_2

- *while* e *do* w $\equiv_{def}$ *if* e *then* (w;(*while* e *do* w))
  else empty
- *repeat* w *until* e $\equiv_{def}$ w;(*while* (NOT e) *do* w).
- *loop* w_1 *exit when* e *otherwise* w_2 $\equiv_{def}$
  $\equiv_{def}$ w_1;(*while* (NOT e) *do* $(w_2; w_1)$) .
- *fin* w $\equiv_{def}$ $\Box\ (if\ empty\ then\ w)$.

La lectura de estas nuevos elementos se hace como sigue:

Elemento	Lectura
- $\{e_1: u < e_2\}$	la lista de las e_1 tales u es menor que e_2.
- $(\forall\ e_1 \leq u < e_2: w)$	para u entre e_1 y e_2: w.
- $(\forall\ u < e: w)$	para u entre 0 y e: w.
- e_1 *gets* e_2	e_1 obtiene e_2.
- *stable* e	estable e.
- $e_1 \approx e_2$	e_1 temporalmente igual a e_2.
- $e_1 \longleftarrow e_2$	asigna temporalmente e_2 a e_1.
- *while* e *do* w	mientras e haz w.
- *repeat* w *until* e	repite w hasta e.
- *loop* w_1 *exit when* e *otherwise* w_2	repite w_1, (en cada ciclo) termina si e, si no haz w_2.
- *fin* w	al final w

LENGUAJE TEMPURA

En este capítulo, presentamos una descripción del lenguaje de programación Tempura tomando como base a la teoría temporal de intervalos (TTI). Tempura fue definido por B.C. Moszkowski[1] después de realizar algunas aplicaciones de la LTI a la especificación y verificación de componentes de hardware[2]. Presentamos a Tempura como una variación de la TTI abordando los puntos que se refieren a su sintaxis, semántica, uso, ubicación e implementación.

3.1 Sintaxis

En esta sección se define la sintaxis de Tempura. Por razones prácticas relacionadas con la evaluación de las fórmulas, la sintaxis de Tempura se obtiene mediante la aplicación de algunas restricciones, extensiones y convenciones a la sintaxis de la TTI.

3.1.1 Restricciones

a) Con el objetivo de facilitar la evaluación, se restringe el uso del operador O en las expresiones. Este operador sólo podrá ocurrir en una expresión si ésta no tiene símbolos de función y aparece en una fórmula a la izquierda de un símbolo de igualdad. El uso de O en fórmulas no se restringe. Debido a esta restricción tenemos que:

- $(O\ X)=Y$, $(O\ O\ X)=Y$ pertenecen a Tempura, pero
- $X = O\ Y$, $O\ X \leq Y$, $X = Y+O\ Z$, $1+(O\ X) = Y$, $p(X, O\ Y)$ no pertenecen a Tempura aunque pertenecen a la TTI.

---

[1] [Moszkowski 86]

[2] [Halpern-Manna-Moszkowski 83], [Moszkowski 83,85].

Esta restricción al uso de O en expresiones está inducida por la estrategia de evaluación. El objetivo de esta estrategia es encontrar un intervalo que satisfaga a la fórmula en cuestión. Este intervalo se determina construyendo consecutivamente (comenzando por el estado cero) cada uno de los estados que lo forman. Cada estado se determina con el valor de las variables en el instante correspondiente. En cada instante de la construcción se conoce solamente el valor de las variables del estado en turno (no se conoce el valor de una variable en los estados pasados o futuros).

Si el valor de una variable depende del valor de una expresión cuyo valor depende a la vez del valor de variables que aparecen en ella, entonces estas variables no pueden depender de valores pasados o futuros. Por ejemplo:

- Si el valor de Y en el estado s_i es c, es posible evaluar (O X) = Y transformándola a O (X = c), lo cual pospone su evaluación al estado s_{i+1}.
- Por el contrario, la evaluación de X = Y+O Y en s_i no es posible (disponiendo sólo de un valor para cada variable) porque se necesitaría saber el valor de Y en s_i y s_{i+1}.

b) Se excluyen de Tempura las fórmulas que tengan alguna ocurrencia de alguno de los símbolos v, ◇ y ¬. Por ejemplo,

- X=1 v X=2, ◇ X=3, ¬ X=4 no pertenecen a Tempura.

Esta restricción al uso de los símbolos v, ◇, ¬ obedece a que actualmente, por razones de simplicidad, la estrategia de evaluación de Tempura es determinista. La estrategia supone que la fórmula a evaluar proporciona información no ambigua del comportamiento de las variables en todo el intervalo. Cada uno de los operadores v ◇ y ¬ tienen diferentes características no-deterministas que ilustramos con los siguientes ejemplos:

- X=1 v X=2 no determina el valor de X en el estado actual, sólo indica que es 1 o 2.
- ◇ X=3 no indica en qué estado tomará X el valor 3.

- ¬ X=4 no determina el valor de X, sólo indica que es distinto de 4. Otra razón para excluir ¬ es que mediante ∧, □ y ¬ se pueden definir ∨ y ◇.

c) Se excluyen de Tempura las fórmulas que tengan alguna ocurrencia de ∀ que no sea de la forma ∀ $e_1 \leq u < e_2$: w, u ∈ VARG. Se requiere que el valor de u esté en un intervalo para evitar evaluar una cantidad infinita de instancias de w; se pide que u sea global para que el valor de u se conserve a través de los estados al evaluar cada instancia de w.

3.1.2 Extensiones y convenciones

a) Para conservar dentro de Tempura a las fórmulas *more*, *true*, *false* y ∃ v:w, agregamos a la definición de las fórmulas de Tempura cláusulas que las incluyan (no como abreviaciones). Anteriormente estas fórmulas se definieron como abreviaciones que usan los operadores ¬ y ∨. Al eliminar de Tempura de las fórmulas con operadores ¬ y ∨ se excluyen también estas abreviaciones.

Es importante observar que Tempura sólo puede evaluar una fórmula de la forma ∃ v:w cuando ésta expresa una propiedad determinista, es decir, cuando la evaluación de w conduce tarde o temprano a la evaluación de una fórmula de la forma v=e. Los casos en que esto no sucede la evaluación fracasa sin poder terminar. Por ejemplo, Tempura puede evaluar ∃ v:(v = e ∧ w) pero no puede evaluar ∃ v:(y = v).

b) Para realizar entrada y salida de datos se agregan las fórmulas *request*($l_1, l_2, \ldots, l_n$) y *display*($e_1, e_2, \ldots, e_n$). La instrucción (fórmula) *request* permite que los valores de sus parámetros sean proporcionados por el usuario. La instrucción *display* despliega el valor de sus parámetros en la salida. El valor de verdad de estas fórmulas es normalmente *true* (si la entrada o salida de datos es satisfactoria).

c) Para *definir* predicados y funciones se agregan dos tipos de fórmula:

- *predicate* p($v_1, v_2, \ldots, v_n$):w . Esta fórmula expresa que el

predicado p está definido por la fórmula w parametrizada por los valores de las variables v_i.
- *function* $f(v_1,v_2,\ldots,v_n)$:e . Esta fórmula expresa que la función f está definida por la expresión e parametrizada por los valores de las variables v_i.

El valor de verdad de estas fórmulas es normalmente *true* (si la definición es satisfactoria). El paso de parámetros es por referencia.

d) Se agregan las siguientes fórmulas de carácter completamente práctico: *quit*, *execute* e_1 y *execute* e_1 e_2. El valor de estas fórmulas es siempre *true* pero realizan una acción lateralmente:

- *quit* abandona el intérprete de Tempura.
- *execute* e_1 redirecciona la entrada del intérprete al archivo cuyo nombre es el valor de e_1.
- *execute* e_1 e_2 redirecciona la entrada y la salida del intérprete a los archivos cuyos nombres son los valores de e_1 y de e_2 respectivamente.

e) Se agrega también por razones prácticas, la fórmula *process* w. La implementación que describe B. Moszkowski[3] utiliza una *variable* interna (*DoneVar*) que indica cuándo termina el intervalo de ejecución, además, asume que todas variables deben tener exactamente un valor en cada estado. Bajo estas condiciones, las fórmulas que expresan redundantemente la longitud del intervalo (por ejemplo *empty* & *empty*) generan un error de ejecución (dos valores para una variable). Para evitar esta situación indeseable Moszkowski incorpora *process* a Tempura. Por ejemplo, (*process empty*) & *empty* no produce un error de ejecución[4].

---

[3] [Moszkowski 86]

[4] Nuestro intérprete evalúa fórmulas con redundancia en *empty* (como *empty* & *empty*). También acepta la instrucción *process*.

f) Para evitar subíndices, se utilizan identificadores en lugar de los símbolos de variable, de función y de predicado. Para distinguir el tipo de cada identificador adoptamos convenciones de notación. Los identificadores que comienzan con letra mayúscula denotan variables locales y los que comienzan con letra minúscula denotan variables globales. Se consideran predicados o funciones aquellos identificadores usados en alguna definición de un predicado o una función mediante las construcciones *predicate* y *function*. Algunos identificadores se usan en forma reservada en lugar de algunos símbolos de Tempura, por ejemplo, 'empty' para el símbolo *empty*, 'always' para $\Box$, 'all' para $\forall$, etc.

A continuación presentamos las definiciones que formalizan las ideas anteriores:

3.2.3 Expresiones

La clase de *locatidades*, LOC, es la clase más pequeña que cumple:

- VAR $\subseteq$ LOC.
- Si l $\in$ LOC entonces $\bigcirc$ l $\in$ LOC.

Observación: LOC $\subseteq$ EXP $\subseteq$ EXPD

La clase de *expresiones de Tempura*, EXPT, es la clase más pequeña que cumple:

a) $\{c_1\} \cup$ VAR $\subseteq$ EXPT.

b) Si $e_1, e_2, \ldots, e_k \in$ EXPT ($k \geq 0$) y f es un símbolo de función de aridad k entonces $f(e_1, e_2, \ldots, e_k) \in$ EXPT.

c) $\{empty, more\} \subseteq$ EXPT.

d) Si $e_1, e_2, e_3 \in$ EXPT, ($if\ e_1\ then\ e_2\ else\ e_3$) $\in$ EXPT.

e) Si $e_1, e_2, e_3 \in$ EXPT y u $\in$ VARG entonces:
$\{e_1 : e_2 \leq u < e_3\} \in$ EXPT.

Observación: VARL $\subseteq$ EXPT $\subseteq$ EXPD

3.1.4 Fórmulas

La clase de *fórmulas de Templura*, FORMT, es la clase más pequeña que cumple:

a) Si r,s ∈ FORMT entonces (r ∧ s) ∈ FORMT.
b) Si $e_1, e_2, \ldots, e_k \in$ EXPT (k≥0) y p es un símbolo de predicado de aridad k entonces $p(e_1, e_2, \ldots, e_k) \in$ FORMT.

c) Si w ∈ FORMT $e_1, e_2 \in$ EXPT y u ∈ VARG entonces
   $\forall\, e_1 \leq u < e_2$: w ∈ FORMT
d) Si w ∈ FORMT y v ∈ VAR entonces (∃ v:w) ∈ FORMT
e) Si l ∈ LOC y e ∈ EXPT entonces (l = e) ∈ FORMT.

f) {*empty*, *more*, *true*, *false*, *quit*} ⊆ FORMT.
g) Si w ∈ FORMT entonces (□ w) ∈ FORMT.
h) Si w ∈ FORMT entonces (○ w) ∈ FORMT.

i) Si r,s ∈ FORMT entonces (r ; s) ∈ FORMT.
j) Si e ∈ EXPT y $w_1, w_2 \in$ FORMT entonces
   (*if* e *then* w_1 *else* w_2) ∈ FORMT

k) Si $l_i \in$ LOC entonces *request*$(l_1, l_2, \ldots, l_n) \in$ FORMT
l) Si $e_i \in$ EXPT entonces *display*$(e_1, e_2, \ldots, e_n) \in$ FORMT

m) Si p es un predicado, $v_i \in$ VAR y w ∈ FORMT entonces
   predicate $p(v_1, v_2, \ldots, v_n)$:w ∈ FORMT
n) Si f es una función, $v_i \in$ VAR y e ∈ EXPT entonces
   function $f(v_1, v_2, \ldots, v_n)$:e ∈ FORMT

o) Si $e_1, e_2 \in$ EXPT entonces
   execute $e_1 \in$ FORMT y *execute* $e_1\ e_2 \in$ FORMT.
p) Si w ∈ FORMT entonces *process* w ∈ FORMT

Observación: No es cierto que FORMD ⊆ FORMT y tampoco es cierto que FORMT ⊆ FORMD, pero FORMT ∩ FORMD ≠ Φ.

3.1.5 Abreviaciones

Las abreviaciones que se conservan en Tempura son las siguientes:

Si $e_1 \in EXPT$, $l_1 \in LOC$, $w_1 \in FORMT$, $v_1 \in VAR$, $u \in VARG$ y $V \in VARL$ entonces:

- $\exists\ v_1, v_2, \ldots, v_n$: w $\equiv_{def} \exists v_1 : (\exists v_2 : (\ldots (\exists v_n : w) \ldots))$
- *if* e *then* w $\equiv_{def}$ *if* e *then* w *else true.*
- *halt* e $\equiv_{def}$ *if* e *then empty else more .*
- *skip* $\equiv_{def} \bigcirc$ *empty* .
- w_1 U w_2 $\equiv_{def} ((\Box w_1)\ ;\ (\bigcirc w_2))$.

- $\{e_1 : u < e_2\}$ $\equiv_{def} \{e_1 : 0 \leq u < e_2\}$.
- $(\forall\ e_1 \leq u < e_2 :$ w) $\equiv_{def} \forall u : (if\ (e_1 \leq u$ AND $u < e_2)$ *then* w)
- $(\forall\ u < e :$ w) $\equiv_{def} \forall u : (if\ (0 \leq u$ AND $u < e)$ *then* w)
- l_1 *gets* e_2 $\equiv_{def} \Box$ (*if* more *then* $((\bigcirc l_1) = e_2)$)
- *stable* V $\equiv_{def}$ V *gets* V.
- $l_1 \approx e_2$ $\equiv_{def} \Box (l_1 = e_2)$.
- $l_1 \longleftarrow e_2$ $\equiv_{def} \exists z : ((z = e_2) \wedge fin\ (l_1 = z))$, donde: z $\in$ VARG y z no ocurre en l_1 ni en e_2

- *while* e *do* w $\equiv_{def}$ *if* e *then* (w;(*while* e *do* w)) *else empty*
- *repeat* w *until* e $\equiv_{def}$ w;(*while* (NOT e) *do* w).
- *loop* w_1 *exit when* e *otherwise* w_2 $\equiv_{def} w_1$;(*while* (NOT e) *do* $(w_2; w_1)$) .
- *fin* w $\equiv_{def} \Box$ (*if empty then* w) .

3.2 Semántica

En esta sección se resume la semántica de Tempura. En apartados independientes se comentan las fórmulas que permiten definir funciones y predicados y la fórmulas que permiten realizar entrada y salida de datos.

3.2.1 Fórmulas y expresiones

La semántica de las fórmulas y expresiones de Tempura se define como sigue:

Si $J=\langle D, \gamma, \sigma \rangle$ es una interpretación de Tempura, la *semántica de Tempura* asociada a J es la función

M_J: EXPT ∪ FORMT ⟶ D

que cumple las siguientes reglas:

(Escribimos $M_\sigma(x)$ en lugar de $M_J(x)$ considerando que D, γ están fijos. p es un predicado, las e_i son expresiones, las l_i son localidades y las w_i son fórmulas.)

Expresiones

- $M_\sigma[c] = \gamma(c)$
- $M_\sigma[v] = \sigma_0(v)$
- $M_\sigma[\bigcirc l] = M_{\sigma_1}[l]$
- $M_\sigma[f(e_1, e_2, \ldots, e_k)] = \gamma(f)(M_\sigma[e_1], M_\sigma[e_2], \ldots, M_\sigma[e_k])$
- $M_\sigma[empty] =$ true sii $|\sigma|=0$ (*empty* ∈ EXPT ∪ FORMT)
- $M_\sigma[more] =$ true sii $|\sigma|\neq 0$ (*more* ∈ EXPT ∪ FORMT).
- $M_\sigma[$*if* e_1 *then* e_2 *else* $e_3]$ = $M_\sigma[e_2]$ si $M_\sigma[e_1]$=true; = $M_\sigma[e_3]$ si $M_\sigma[e_1]\neq$true
- $M_\sigma[\{e_1: e_2 \leq u < e_3\}]$ =
  - = nil si $c_3 - c_2 \leq 0$
  - = Cons$(c_1, M_\sigma[\{e_1: c_2+1 \leq u < c_3\}])$ si no

  donde $c_1 = M_\sigma[e_1[c_2/u]]$, $c_2 = M_\sigma[e_2]$, $c_3 = M_\sigma[e_3]$.

Fórmulas

- $M_\sigma[w_1 \wedge w_2] =$ true sii $M_\sigma[w_1]$=true y $M_\sigma[w_2]$=true.
- $M_\sigma[p(e_1, \ldots, e_k)] = \gamma(p)(M_\sigma[e_1], \ldots, M_\sigma[e_k])$.
- $M_\sigma[\forall v: w] =$ true sii

  $M_{\sigma'}[w]$ =true para toda $\sigma' \in \Sigma_D^+$ tal que $\sigma' \approx_v \sigma$.

$M_\sigma[\exists\ v{:}w]$ = true sii
existe $\sigma' \in \Sigma_D^+$ tal que $\sigma' \propto_v \sigma$ y $M_{\sigma'}[w]$ =true .

- $M_\sigma[l = e]$ = true sii $M_\sigma[l] = M_\sigma[e]$.
- $M_\sigma[true]$ = true
- $M_\sigma[false]$ = false
- $M_\sigma[quit]$ = true
- $M_\sigma[\Box\ w]$ = true sii
para toda $i \in \mathbb{N}$: $i \le |\sigma| \supset M_{\sigma^i}[w]$=true.
- $M_\sigma[O\ w]$ = true sii $M_{\sigma^1}[w]$=true.
- $M_\sigma[w_1;\ w_2]$ = true sii existe $i \in \mathbb{N}$ tal que
$i \le |\sigma|$, $M_{1_\sigma}[w_1]$=true y $M_{\sigma^1}[w_2]$=true.
- $M_\sigma[if\ e\ then\ w_1\ else\ w_2]$ = $M_\sigma[w_1]$ si $M_\sigma[e]$ = true,
= $M_\sigma[w_2]$ si $M_\sigma[e] \neq$ true.
- $M_\sigma[len(e)]$ = true sii $M_\sigma[e] \in \mathbb{N}$ y $|\sigma| = M_\sigma[e]$.
- $M_\sigma[request(l_1, l_2, \ldots, l_n)]$ = true
- $M_\sigma[display(e, e_2, \ldots, e_n)]$ = true
- $M_\sigma[process\ w]$ = $M_\sigma[w]$
- $M_\sigma[for\ e\ times\ do\ w]$ =
= $M_\sigma[empty]$ si $c \le 0$.
= $M_\sigma[w\ ;\ for\ c\text{-}1\ times\ do\ w]$ si no.
donde $c = M_\sigma[e]$.
- $M_\sigma[for\ e_1 \le u < e_2\ do\ w]$=
= $M_\sigma[empty]$ si $c_1 \ge c_2$,
= $M_\sigma[(\exists u{:}(u = c_1 \wedge w));$
$(for\ c_1{+}1 \le u < c_2\ do\ w)]$ si $c_1 < c_2$.
donde $c_1 = M_\sigma[e_1]$ y $c_2 = M_\sigma[e_2]$.

- $M_\sigma[for\ u\ in\ e\ do\ w]$ =
= $M_\sigma[empty]$ si $c = \{\}$
= $M_\sigma[(\exists u{:}(u = Car(c) \wedge w));$
$(for\ u\ in\ Cdr(c)\ do\ w)]$ si $c \neq \{\}$.
donde $c = M_\sigma[e]$.
- $M_\sigma[execute\ e]$ = true
- $M_\sigma[execute\ e_1, e_2]$ = true

- M_σ[*predicate* $p(x_1,x_2,\ldots,x_n)$: w] = true sii $\gamma(p) = w_1$

  donde

  $l=(x_1,x_2,\ldots,x_n) \in VAR^n$, y

  w_1, el *predicado asociado* a w respecto a l, se define como:

  $w_1 : D^n \longrightarrow \{true, false\}$

  $w_1(d_1,d_2,\ldots,d_n) = M_\sigma[w[(d_1,d_2,\ldots,d_n)/(x_1,x_2,\ldots,x_n)]]$

- M_σ[*function* $f(x_1,x_2,\ldots,x_n)$: e] = true sii $\gamma(f) = e_1$

  donde

  $l=(x_1,x_2,\ldots,x_n) \in VAR^n$, y

  e_1, la *función asociada* a e respecto a l se define como:

  $e_1: D^n \longrightarrow D$

  $e_1(d_1,d_2,\ldots,d_n) = M_\sigma[e[(d_1,d_2,\ldots,d_n)/(x_1,x_2,\ldots,x_n)]]$

3.2.2 Predicados y funciones

Las fórmulas de Tempura

predicate $p(v_1,v_2,\ldots,v_n)$:w y *function* $f(v_1,v_2,\ldots,v_n)$:e, permiten darle a los símbolos de predicados y de función un significado cercano al significado de procedimientos y funciones de los lenguajes de programación. Antes de la introducción de estas fórmulas, el significado de los predicados y funciones quedaba determinado por la valuación global (γ) de la interpretación. Ahora, dicho significado puede ser controlado por medios sintácticos. Veamos un ejemplo:

Supongamos que (p es un predicado, f una función, X,Y,A son variables) w es la fórmula

(*predicate* p(X,Y): X=Y+1) $\wedge$ p(A,3) $\wedge$ *empty*.

Supongamos además que J=<Bv,γ,σ> es una interpretación y calculemos M_σ[w]:

M_σ[w] = true sii

a) M_σ[*predicate* p(X,Y): X=Y+1] = true, y

b) M_σ[p(A,3)] = true, y

c) M_σ[*empty*] = true

Esto se cumple sii

a) $\gamma(p) = w_{(X,Y)}$

$w_{(X,Y)} : Bv^2 \longrightarrow \{true, false\}$

$w_{(X,Y)}(a,b) = M_\sigma[w[(a,b)/(X,Y)]]$

$= M_\sigma[a=b+1]$.

$\therefore w_{(X,Y)}(a,b) = true$ sii

$M_\sigma[a=b+1]=true$ sii $M_\sigma[a] = M_\sigma[b+1]$ sii $a=b+1$.

b) $\gamma(p)(M_\sigma[A], M_\sigma[3]) = true$

c) $|\sigma|=0$

Esto se cumple sii

a) $\gamma(p) : Bv^2 \longrightarrow \{true, false\}$

$\gamma(p)(a,b) = true$ sii $a=b+1$.

b) $\sigma_0(A) = M_\sigma[A] = M_\sigma[3]+1 = 4$

c) $|\sigma|=0$

$\therefore M_\sigma[w] = true$ sii $(\sigma=<\sigma_0>,\ \sigma_0(A)=4$ y $\gamma(p)(a,b) \equiv a=b+1)$.

En este ejemplo se observa claramente como la fórmula *predicate* controla el significado del predicado p ($\gamma(p)$).

Una restricción a las fórmulas

predicate $p(v_1, v_2, \ldots, v_n)$:w y *function* $f(v_1, v_2, \ldots, v_n)$:e,

es que solamente permiten definir predicados y funciones de *primer orden*, sus parámetros sólo pueden ser referencias a expresiones de Tempura. Esta restricción puede relajarse para permitir la definición de predicados y funciones de *orden superior* que acepten como parámetros referencias a predicados o funciones.

Para lograr una extensión de este tipo, es necesario modificar, en la sintaxis de Tempura, las cláusulas para predicados y funciones en de la siguiente manera:

Si p es un símbolo de predicado, f un símbolo de función, y $\forall i\ (1 \le i \le n \supset x_i \in EXPT \cup \{p_i^j\} \cup \{f_i^j\})$, entonces:

$f(x_1, x_2, \ldots, x_n) \in EXPT$. y

$p(x_1, x_2, \ldots, x_n) \in FORMT$.

Sin embargo, esta modificación complica la semántica. La clase de los valores básicos, Bv, ya no podrá servir como base de las interpretaciones. Las interpretaciones de Tempura que tienen a Bv como dominio sólo cuentan con predicados y funciones aplicables a elementos de Bv. Es decir, estas interpretaciones sólo tienen predicados y funciones de primer orden. Por lo tanto, no es posible interpretar los predicados y funciones de orden superior con interpretaciones cuyo dominio sea Bv.

El significado formal de los predicados y funciones de orden superior se escapa del alcance del método que hasta ahora hemos empleado para definición de la semántica de Tempura. Sin embargo cabe comentar que una forma de obtener un dominio D donde se puedan interpretar funciones de orden superior es resolver una ecuación del siguiente tipo:

$$D = Bv + D \longrightarrow D + D^2 \longrightarrow D + \ldots + D^i \longrightarrow D + \ldots$$

Una ecuación de este tipo pertenece a la teoria de dominios que nos lleva directamente al área de la semántica denotacional[1].

Aunque nuestra implementación de Tempura actualmente permite usar predicados y funciones como parámetros, no profundizaremos más en este tema y pospondremos la definición de la semántica formal de los predicados y funciones de orden superior para un trabajo futuro. El apéndice A contiene semántica de Tempura al estilo de la semántica denotacional. Esta semántica incluye predicados y funciones de orden superior.

3.2.3 Entrada y salida de datos

Hasta ahora, la semántica de Tempura le asocia a las fórmulas
- *request* $(i_1, i_2, \ldots, i_n)$, y
- *display* $(e_1, e_2, \ldots, e_n)$

el valor de verdad true. Salta a la vista que esta semántica no

[1] [Allison 86].

refleja las acciones de solicitud y despliegue de datos implícitas en estas fórmulas.

Una forma para remediar esto es extender la semántica de tal forma que además de asociarle a cada fórmula w un valor de verdad, también se le asocien un par de sucesiones de elementos del dominio que llamaremos entrada y salida w. A continuación se formalizan estos conceptos y se aclaran con algunos ejemplos.

En lo que sigue, si $D \neq \Phi$, D^ω es el conjunto de *sucesiones* de elementos de D. Si $\alpha \in D^\omega$, $Dom(\alpha) = \{0,1,\ldots,n\}$, y $\alpha(i) = d_i$, usaremos $'d_0 d_1 \ldots d_n'$ o $'d_0, d_1, \ldots, d_n'$[2] en lugar de α. Si $\alpha, \beta \in D^\omega$, $\alpha\beta$ representa la *concatenación* de α y β, si α no es finita $\alpha\beta = \alpha$. Usamos $\Pi_{i \in I} \alpha_i$ para indicar la concatenación de un conjunto de elementos indexado por I. λ representa a la *sucesión nula* que tiene la propiedad de que $\forall\, \alpha \in D^\omega$: $\alpha\lambda = \lambda\alpha = \alpha$.

Decimos que $S = \langle D, \varepsilon, \delta, r \rangle$, es un *marco de entrada-salida* sobre D sii se cumple que:

- D es una clase no vacía,
- ε, la *función de entrada* de S le asigna a cada predicado p_i^j una función $\varepsilon(p_i^j) : D^j \longrightarrow D^\omega$.
- δ, la *función de salida* de S le asigna a cada predicado p_i^j una función $\delta(p_i^j) : D^j \longrightarrow D^\omega$.
- r, la *entrada-salida* de S es un elemento de $D^\omega \times D^\omega$.

Decimos que $J = \langle D, \gamma, \sigma, S \rangle$, es una *interpretación con entrada-salida* sii se cumple que:

- $\langle D, \gamma, \sigma \rangle$ es una interpretación de Tempura,
- $S = \langle D, \varepsilon, \delta, r \rangle$ es un marco de entrada-salida sobre D.

Si $J = \langle D, \gamma, \sigma, S \rangle$ es una interpretación con entrada-salida donde $S = \langle D, \varepsilon, \delta, r \rangle$, la *entrada asociada* a J es la función

E_J: FORMT $\longrightarrow D^\omega$

---

[2] usamos "d_i, d_{i+1}" cuando "$d_i d_{i+1}$" es ambigua.

que cumple las siguientes reglas:
(Escribimos $E_\sigma(x)$ en lugar de $E_J(x)$ considerando que D,τ,S están fijos. p es un predicado, las e_i son expresiones, las l_i son localidades y las w_i son fórmulas.)

- $E_\sigma[w_1 \wedge w_2] = E_\sigma[w_1]E_\sigma[w_2]$ (concatenación)
- $E_\sigma[p(e_1,\ldots,e_k)] = c(p)(M_\sigma[e_1],\ldots,M_\sigma[e_k])$.
- $E_\sigma[\forall\ v:\ w] = \Pi_{\tau\in X}\ E_\tau[w]$ (concatenación)
  donde $X= \{\tau \in \Sigma_D^+ \ /\ \tau \approx_v \sigma.\}$
- $E_\sigma[\exists\ v:\ w] = 'E_\tau[w]'$ si $\exists\ \tau$ $(\tau \in \Sigma_D^+ \wedge \tau \approx_v \sigma \wedge M_\tau[w] = true)$
  $= \lambda$ si no.
- $E_\sigma[l = e] = \lambda$
- $E_\sigma[c] = \lambda$ para $c \in \{empty, more, true, false, quit\}$
- $E_\sigma[\Box\ w] = \Pi_{\tau\in X}\ E_\tau[w]$ donde $X=\{\tau/\ \exists i\ \tau= \sigma^i\ \}$
- $E_\sigma[\bigcirc\ w] = E_{\sigma^1}[w]$
- $E_\sigma[w_1;\ w_2]$ =
  $= E_{i_\sigma}[w_1]E_{\sigma^i}[w_2]$ si $\exists\ i \leq |\sigma|$ $(M_{i_\sigma}[w_1]= true \wedge M_{\sigma^i}[w_2]= true\ )$
  $= \lambda$ si no.
- $E_\sigma[$*if* e *then* w_1 *else* $w_2]$ $= E_\sigma[w_1]$ si $M_\sigma[e] = true$, $= E_\sigma[w_2]$ si $M_\sigma[e] \neq true$
- $E_\sigma[request(l_1, l_2, \ldots, l_n)] = 'M_\sigma[l_1]M_\sigma[l_2]\ldots M_\sigma[l_n]'$
- $E_\sigma[display(e_1, e_2, \ldots, e_n)] = \lambda$
- $E_\sigma[predicate\ p(x_1,x_2,\ldots,x_n):\ w] = \lambda$
- $E_\sigma[function\ f(x_1,x_2,\ldots,x_n):\ e] = \lambda$
- $E_\sigma[execute\ e_1] = \lambda$[3]
- $E_\sigma[execute\ e_1\ e_2] = \lambda$
- $E_\sigma[process\ w] = E_\sigma[w]$

---

[3]La entrada de la fórmula contenida en el archivo $M_\sigma[e_1]$ no interfiere con la entrada de (*execute* e_1).

Si $J=<D,\gamma,\sigma,S>$ es una interpretación con entrada-salida donde $S=<D,\varepsilon,\delta,r>$, la *salida asociada* a J es la función

S_J: FORMT $\longrightarrow D^\omega$

que cumple las siguientes reglas:

(Escribimos $S_\sigma(x)$ en lugar de $S_J(x)$ considerando que D,γ,S están fijos. p es un predicado, las e_i son expresiones, las l_i son localidades y las w_i son fórmulas.)

- $S_\sigma[w_1 \wedge w_2] = S_\sigma[w_1]S_\sigma[w_2]$ (concatenación)
- $S_\sigma[p(e_1,\ldots,e_k)] = \delta(p)(M_\sigma[e_1],\ldots,M_\sigma[e_k])$.
- $S_\sigma[\forall\ v:\ w] = \Pi_{\tau\in X}\ S_\tau[w]$ (concatenación)

  donde $X=\{\tau \in \Sigma_D^+ \ /\ \tau \approx_v \sigma.\}$
- $S_\sigma[\exists\ v:\ w] = 'S_\tau[w]'$ si $\exists\ \tau$ $(\tau \in \Sigma_D^+ \wedge \tau \approx_v \sigma \wedge M_\tau[w] = true)$

  $= \lambda$ si no.
- $S_\sigma[l = e] = \lambda$
- $S_\sigma[c] = \lambda$ para $c \in \{empty, more, true, false, quit\}$
- $S_\sigma[\Box\ w] = \Pi_{\tau\in X}\ S_\tau[w]$ donde $X=\{\tau/\ \exists i\ \tau = \sigma^i\ \}$
- $S_\sigma[\bigcirc\ w] = S_{\sigma^1}[w]$
- $S_\sigma[w_1;\ w_2] =$

  $= S_{1_\sigma}[w_1]S_{\sigma^1}[w_2]$ si $\exists\ i \leq |\sigma|$ $(M_{1_\sigma}[w_1] = true \wedge M_{\sigma^1}[w_2] = true\ )$

  $= \lambda$ si no.

- $S_\sigma[$*if* e *then* w_1 *else* $w_2]$ $= S_\sigma[w_1]$ si $M_\sigma[e] = true$, $= S_\sigma[w_2]$ si $M_\sigma[e] \neq true$
- $S_\sigma[request(l_1,l_2,\ldots,l_n)] = \lambda$
- $S_\sigma[display(e_1,e_2,\ldots,e_n)] = 'M_\sigma[e_1]M_\sigma[e_2]\ldots M_\sigma[e_n]'$
- $S_\sigma[predicate\ p(x_1,x_2,\ldots,x_n):\ w] = \lambda$
- $S_\sigma[function\ f(x_1,x_2,\ldots,x_n):\ e] = \lambda$

- $S_\sigma[execute\ e_1] = \lambda$[4]
- $S_\sigma[execute\ e_1\ e_2] = \lambda$
- $S_\sigma[process\ w] = S_\sigma[w]$

Si J=<D,γ,σ,S> es una interpretación con entrada-salida, $w \in$ FORMT, $n \geq 0$ y $l=(x_1,x_2,\ldots,x_n) \in VAR^n$, la *entrada asociada* a w respecto a l, $Ent_{w,l}$, es la función:

$Ent_{w,l} : D^n \longrightarrow D^\omega$

$Ent_{w,l}(d_1,d_2,\ldots,d_n) = E_\sigma[w[(d_1,d_2,\ldots,d_n)/(x_1,x_2,\ldots,x_n)]]$

Si J=<D,γ,σ,S> es una interpretación con entrada-salida, $w \in$ FORMT, $n \geq 0$ y $l=(x_1,x_2,\ldots,x_n) \in VAR^n$, la *salida asociada* a w respecto a l, $Sal_{w,l}$, es la función:

$Sal_{w,l} : D^n \longrightarrow D^\omega$

$Sal_{w,l}(d_1,d_2,\ldots,d_n) = S_\sigma[w[(d_1,d_2,\ldots,d_n)/(x_1,x_2,\ldots,x_n)]]$

Las definiciones de *predicada asociada* y *función asociada*, para interpretaciones (sin entrada-salida) se aplican en forma análoga para interpretaciones con entrada-salida.

Si J=<D,γ,σ,S> es una interpretación con entrada-salida, la *semántica asociada* a J, M_J, es la función que cumple:

- M_J: EXPT $\cup$ FORMT $\longrightarrow$ D $\cup$ {true,false}
- M_J se comporta igual que la semántica asociada a la interpretación <D,γ,σ> (sin entrada-salida) excepto en el siguiente caso:
- $M_\sigma[predicate\ p(x_1,x_2,\ldots,x_n)\colon w] =$ true sii se cumplen:
  - $\gamma(p) = w_{(x_1,x_2,\ldots,x_n)}$
  - $\varepsilon(p) = Ent_{w,(x_1,x_2,\ldots,x_n)}$
  - $\delta(p) = Sal_{w,(x_1,x_2,\ldots,x_n)}$

---

[4]La salida de la fórmula contenida en el archivo $M_\sigma[e_1]$ no interfiere con la salida de $(execute\ e_1)$.

Si $J=<D,\gamma,\sigma,S>$ es una interpretación con entrada-salida donde $S=<D,\varepsilon,\delta,(E,S)>$ y $w \in FORMT$, decimos que J *realiza* a w o que J es un *modelo* para w, $J \models w$, sii se cumple que:

- $M_\sigma[w]$= true
- $E_\sigma[w]$= E
- $S_\sigma[w]$= S.

Presentamos a continuación algunos ejemplos que sirven para aclarar los conceptos anteriores.

Sea $J=<D,\gamma,\sigma,<D,\varepsilon,\delta,(E,S)>$ > una interpretación con entrada salida sujeta a las hipótesis de los siguientes ejemplos:

-a) Supongamos que:

- w= request(X) $\wedge$ display(X)
- $\sigma_0(a) = 1$ si a=X
  $= \perp$ si no
- $\sigma=<\sigma_0>$.
- Si además se tiene que
  E='2' y S='2'..... (a1)

entonces

M_σ[request(X)]=true, M_σ[display(X)]=true,
E_σ[request(X)]=M_σ[X]='1', E_σ[display(X)]= λ,
S_σ[request(X)]= λ, S_σ[display(X)]='M_σ[X]'='1',
$\therefore$ M_σ[w]=true, E_σ[w]='1' $\neq$ E, S_σ[w]='1' $\neq$ S,

$\therefore$J no realiza a w .

- Si en lugar de (a1) se tiene
  E='1' y S='1'..... (a2)

entonces

M_σ[w]=true, E_σ[w]='1' = E, S_σ[w]='1' = S,
$\therefore$ $J \models w$

-b) Supongamos que

- w = (function f(X): X+3) $\wedge$
  (predicate p(X,Y): O Y=f(X) $\wedge$ O X=X+1) $\wedge$
  len(1) $\wedge$ Y=0 $\wedge$ request(X) $\wedge$ p(X,Y) $\wedge$
  $\Box$ display(X,Y) .
- $\sigma_0(a) = 0$ si a=X o a=Y
  $= \perp$ si no

- $\sigma_1(a)$ = 1 si a=X
  = 3 si a=Y
  = $\perp$ si no
- $\sigma=<\sigma_0\sigma_1>$.
- $\gamma(f)$: D $\longrightarrow$ D,
  $\gamma(f)(a)= a+3= M_\sigma[a+3] = M_\sigma[(X+3)[a/X]]$
- $\gamma(p)$: D^2 $\longrightarrow$ {true,false},
  $\gamma(p)(a,b)$ = true sii a=0,
  ahora bien, respecto a $\gamma(p)$ tenemos que:
  $M_\sigma[(\bigcirc Y=f(X) \wedge \bigcirc X=X+1)\ [(a,b)/(X,Y)]] =$ true sii
  $M_\sigma[(\bigcirc Y=a+3 \wedge \bigcirc X=a+1)]$ = true sii
  ($\sigma_1(Y)=a+3$ y $\sigma_1(X)=a+1$) sii
  ($a=\sigma_1(Y)-3$ y $a=\sigma_1(X)-1$) sii a=0
  $\therefore \gamma(p)(a,b)= M_\sigma[(\bigcirc Y=f(X) \wedge \bigcirc X=X+1)\ [(a,b)/(X,Y)]]$
- $\varepsilon(p)$: D^2 $\longrightarrow$ D^ω, $\varepsilon(a,b)= \lambda$.
- $\delta(p)$: D^2 $\longrightarrow$ D^ω, $\delta(a,b)= \lambda$.

entonces:

calculando $M_\sigma[w]$:

$M_\sigma[$(function f(X): X+3)$]$=true,

$M_\sigma[$(predicate p(X,Y): $\bigcirc Y=f(X) \wedge \bigcirc X=X+1$)$]$=true

$M_\sigma[$len(1)$]$=true ya que $|\sigma|=1$,

$M_\sigma[Y=0]$=true porque $\sigma_0(Y)=0$,

$M_\sigma[$request(X)$]$=true,

$M_\sigma[p(X,Y)] = \gamma(p)(M_\sigma[X],M_\sigma[Y])= \gamma(p)(\sigma(X),\sigma(Y))=$
$= \gamma(0,0)=$ true

$M_\sigma[\Box$ display(X,Y)$]$= true porque
$\forall\sigma\ M_\sigma[$display(X,Y)$]$= true,

$\therefore M_\sigma[w]$=true .

Calculando $E_\sigma[w]$:

$E_\sigma[$(function f(X): X+3)$]= \lambda$,

$E_\sigma[$(predicate p(X,Y): $\bigcirc Y=f(X) \wedge \bigcirc X=X+1$)$]= \lambda$,

$E_\sigma[$len(1)$]= \lambda$,

$E_\sigma[Y=0]= \lambda$,

$E_\sigma[$request(X)$]=$ '$M_\sigma[X]$'= '0',

$E_\sigma[p(X,Y)]= \varepsilon(p)(M_\sigma[X],M_\sigma[Y])= \lambda$,

$E_\sigma[\Box$ display(X,Y)$]= \lambda$ porque $\forall\sigma\ E_\sigma[$display(X,Y)$]= \lambda$,

$\therefore E_\sigma[w]=$ '0' .

Calculando $S_\sigma[w]$:

$S_\sigma[(\text{function } f(X): X+3)] = \lambda$,

$S_\sigma[(\text{predicate } p(X,Y): \bigcirc Y=f(X) \wedge \bigcirc X=X+1)] = \lambda$,

$S_\sigma[\text{len}(1)] = \lambda$,

$S_\sigma[Y=0] = \lambda$,

$S_\sigma[\text{request}(X)] = \lambda$,

$S_\sigma[p(X,Y)] = \delta(p)(M_\sigma[X], M_\sigma[Y]) = \lambda$,

$S_\sigma[\Box\ \text{display}(X,Y)] = \text{'0,0,1,3'}$[5] porque

$S_{\sigma^0}[\text{display}(X,Y)] = \text{'}M_{\sigma^0}[X]M_{\sigma^0}[Y]\text{'} = \text{'0,0'}$,

$S_{\sigma^1}[\text{display}(X,Y)] = \text{'}M_{\sigma^1}[X]M_{\sigma^1}[Y]\text{'} = \text{'1,3'}$

$\therefore S_\sigma[w] = \text{'0,0,1,3'}$.

- Si además se tiene que
  E='2' y S='2' .....(b1)
  entonces
  $M_\sigma[w]$=true, $E_\sigma[w]$='0' $\neq$ E, $S_\sigma[w]$='0,0,1,3' $\neq$ S,
  $\therefore$J no realiza a w .
- Si en lugar de (b1) se tiene
  E='0' y S='0,0,1,3' .....(b2)

entonces
$M_\sigma[w]$=true, $E_\sigma[w]$='0' = E, $S_\sigma[w]$='0,0,1,3' = S,
$\therefore$ J $\models$ w

---

[5] usamos "," sólo para distinguir los elementos de la cadena.

3.3 Práctica

El propósito de esta sección es exponer de manera informal qué es y cómo se comporta un programa en Tempura. Para esto, se presentan varios programas que ejemplifican el uso de las instrucciones de Tempura en su forma más simple.

Las condiciones de implementación del intérprete nos han llevado a realizar una transliteración de algunos símbolos de Tempura que no pueden introducirse con un teclado de PC.

Tempura	Teclado
□ w	*always* w.
○ w	*snext* w.
(○ l) = e	*next*v l = e[1]
$w_1 \wedge w_2$	w_1 & w_2.
l ≈ e	l *tequ* e.
l ⟵ e	l *tass* e.
∃ v: w	*exist* v: w.
∀ u < e: w	*all* u < e: w.
e ≤ d	e <= d
e ≥ d	e >= d
e ≠ d	e <> d

Aunque el enfoque adoptado aquí es de tipo práctico, es importante tener siempre en mente que el significado de las instrucciones de Tempura está respaldado por la semántica de las fórmulas de la TTI.

3.3.1 Visión general

Para comprender la forma de operación de Tempura es conveniente tener siempre en cuenta dos hechos: primero, aunque Tempura es un lenguaje de programación imperativo, tiene características propias que no son comunes a los lenguajes imperativos; segundo, aunque su sintaxis se hereda de un formalismo lógico, Tempura no

[1] diferenciamos ○ w y ○ l (w ∈ FORMT, l ∈ LOC).

es un lenguaje de programación lógica en el sentido de que no es un probador automático de teoremas.

Lo primero que se necesita saber para ejecutar el intérprete es lo siguiente:

- El intérprete entra en operación al ejecutar el programa Tempura en el ambiente del sistema operativo,
- Tempura indica que se encuentra en espera de instrucciones mediante el indicador (prompt) "ASK" y que reconoce el final de las instrucciones mediante el símbolo "?".
- La instrucción *quit* sirve para indicar a Tempura el fin de una sesión, es decir, que ya no habrá más instrucciones.

La ejecución del intérprete con un fin de sesión inmediato se ilustra como sigue (">" es el prompt de MS-Dos) :

```
> tempura
ASK. quit ?
> ...
```

La forma en que Tempura ejecuta un programa F, que puede ser visto como una fórmula de la TTI, es buscando un intervalo σ que lo satisfaga, $M_\sigma[F]$=true. Teniendo esto en cuenta veamos como se ejecuta el programa más simple en Tempura, *empty*.

Si en respuesta al prompt de Tempura tecleamos "*empty* ?" el intérprete buscará un intervalo σ tal que $M_\sigma[empty]$= true. De acuerdo a la semántica de la TTI $M_\sigma[empty]$=true sii $|\sigma|=0$, por lo tanto, cualquier intervalo con un sólo estado realiza al programa "*empty* ?". Lo único que hace el intérprete al recibir la instrucción "*empty*" es indicar que se requirió un intervalo de longitud cero. A continuación mostramos una sesión de Tempura en la que se ejecuta este programa:

```
> tempura
ASK. empty ?
State 0:
ASK. quit ?
> ...
```

La instrucción más simple que permite asignarle un valor a una variable es la asignación simple (=). El programa "X = 1?" le asigna el valor 1 a la variable X en el estado inicial, sin embargo, al ejecutarlo se generará un error:

```
ASK.X=1 ?
State 0:
[RunError]3302: Termination status not especified
ASK.
```

Esto sucede porque porque los programas en Tempura deben especificar en qué momento (estado) termina su ejecución. Una manera de corregir este error es mediante la instrucción y (&) que permite ejecutar dos instrucciones en el mismo estado. La siguiente ejecución no produce el error anterior:

```
ASK. X = 1 & empty?
State 0:
ASK.
```

Si además deseamos desplegar el valor de X, se puede agregar la instrucción "display(X)":

```
ASK. X = 1 & empty & display(X)?
State 0:
X= 1
ASK.
```

Es necesario notar que el orden de los operandos en la instrucción "r & s" no afecta el resultado, así el programa "display(X) & empty & X = 1" produce el mismo efecto que el ejemplo anterior.

Los operadores *después* (*snext*) y *siempre* (*always*) permiten ir más allá del primer estado. La siguiente ejecución es una muestra simple de su uso

```
ASK. X = 1 & (snext (X = 2)) & len(1) &
     always display(X)?
State 0:
X= 1
State 1:
X= 2
ASK.
```

Es importante hacer algunas observaciones en este momento. Se cuenta con dos tipos de variables: locales, cuyo nombre empieza con mayúscula y globales, cuyo nombre empieza con minúscula. Con variables globales la asignación simple (=) sólo puede usarse una vez porque el valor asignado se conserva en todo el intervalo. Con variables locales la asignación simple sólo modifica el valor de la variable en el estado en que se ejecuta y no se conserva en los estados siguientes. Esto marca una diferencia con la asignación en los lenguajes imperativos donde el valor asignado se mantiene mientras no se asigne otro valor. Tempura cuenta con otros tipos de asignación además de la asignación simple:

La *igualdad temporal* (tequ) permite asignar un valor a una variable durante todo un intervalo de ejecución:

```
ASK. X tequ 3 & len(1) & always display(X)?
State 0:
X=3
State 1:
X=3
ASK.
```

La operación *gets* permite que una variable tome siempre el valor actual de una expresión en el siguiente estado:

```
ASK. X=0 & X gets X+1 & len(2) & always display(X)?
State 0:
X=0
State 1:
X=1
State 2:
X=2
ASK.
```

La asignación temporal *tass* permite que al final de un intervalo una variable tome el valor actual de una expresión.

```
ASK. X=0 & X tass 1 & len(1) & always display(X)?
State 0:
X=0
State 1:
X=1
ASK.
```

Tempura permite la definición de predicados y funciones mediante rutinas similares a las de los lenguajes imperativos. Las rutinas que definen un predicado agrupan una serie de instrucciones cuya ejecución depende de los parámetros de la llamada a la rutina. Las rutinas que definen funciones agrupan una serie de operaciones que determinan un valor de acuerdo a los parámetros entregados. El mecanismo que usa Tempura para evaluar los parámetros de las llamadas a las rutinas es el de referencia. Se permite la definición recursiva de rutinas (inclusive rutinas mutuamente recursivas) y un parámetro puede ser la referencia a una rutina:

```
ASK. (predicate p(X,Z): X=f(Z)+1) &
     (function f(X): if X=0 then 1 else X*f(X-1)) &
     len(0) & p(A,3) & display(A)?
State 0:
A=7
ASK.
```

Tempura puede evaluar predicados de segundo orden como el del siguiente ejemplo. La semántica de este tipo de predicados no se incluye dentro de la semántica *lógica* pero la semántica denotacional[2] de Tempura si la contempla.

```
ASK. (predicate p(X,y,Z): X=y(Z)) &
     (function f(X): if X=0 then 1 else X*f(X-1)) &
     len(0) & p(A,f,3) & display(A)?
State 0:
A=6
ASK.
```

La instrucción de composición secuencial *corta* (;) de Tempura permite indicar que la ejecución de una fórmula se efectuará en dos subintervalos de tiempo en los que se traslapa el último estado de uno con el primer estado del otro:

---

[2]ver el apéndice A.

```
ASK. ( len(1) & X=0 & (nextv X=1) & Y tequ 'algo' &
     always display(X,Y) ) ;
     ( len(1) & Y=X & (nextv Y=2) & X tequ 'otro' &
     always display(X,Y) ) ?
State 0:
X=0 Y= algo
State 1:
X=1 Y= algo X= otro Y= 1
State 2:
X= otro Y= 2
ASK.
```

Las instrucciones de iteración de Tempura permiten concatenar las ejecuciones sucesivas de una fórmula mediate el operador *corta*. Su comportamiento es similar al de las instrucciones de iteración de los lenguajes imperativos, pero es importante recordar el funcionamiento de las variables de Tempura y que al concatenar dos instrucciones mediante *corta* los valores de las variables en el último estado de la ejecución de la primera instrucción serán los valores de las variables en el primer estado de la ejecución de la segunda instrucción:

```
ASK. (X=0) & (trough 2 do (skip & X tass X+1)) &
     always display(X) ?
State 0:
X=0
State 1:
X=1
State 2:
X=2
ASK.

ASK. (X=0) & (for K<3 do (skip & X tass X+K) &
     always display(X) ?
State 0:
X=0
```

```
State 1:
X=0
State 2:
X=1
State 3:
X=3
ASK.

ASK. (M=6)&(N=15)&
     (while (M≠0) do (skip & (M tass N mod M) & (N tass M))&
     always display(M,N) ?
State 0:
M=6 N=15
State 1:
M=3 N=6
State 2:
M=0 N=3
ASK.
```

Tempura cuenta además con las instrucciones de iteración

- *for* e *in* l *do* w
- *repeat* w *until* e
- *loop* w_1 *exit when* e *otherwise* w_2

que funcionan en forma similar a las instrucciones de iteración que hemos mostrado.

Con la instrucción *more* se puede indicar que el intervalo de ejecución tiene un siguiente estado, que la ejecución no ha terminado:

```
ASK. (I=0) & (I gets I+1) &
     (always display(I) & if I<2 then more else empty)?
State 0:
I=0
State 1:
I=1
State 2:
I=2
ASK.
```

La instrucción *skip* permite expresar que el intervalo de ejecución es de longitud 1, que sólo hay dos estados:

```
ASK. (always display(I)) & (I=0) & (nextv I = 1) & skip ?
State 0:
I=0
State 1:
I=1
ASK.
```

Con la instrucción *halt e* de Tempura es posible detener la ejecución de un programa cuando se cumple la condición *e*:

```
ASK. (always display(I)) & (I=0) & (I gets I+1) &
     (halt I=1) ?
State 0:
I=0
State 1:
I=1
ASK.
```

La instrucción *fin w* sirve para indicar que la fórmula *w* debe ejecutarse al final del intervalo:

```
ASK. len(1) & (I=1) & (I gets I*3) &
     (fin display(I)) ?
State 0:
State 1:
I=3
ASK.
```

Con la instrucción *stable* se pude expresar que una variable conservará el valor que tiene hasta el final del intervalo:

```
ASK. (len(2) & (I=0) & (snext (I = 5 & stable I) &
     (always display(I)) ?
State 0:
I=0
```

```
State 1:
I=5
State 2:
I=5
ASK.
```

La instrucción de cuantificación existencial de Tempura permite manejar una variable con diferentes contextos (alcances) dentro de una misma fórmula. El siguiente ejemplo hace uso de la variable I con dos contextos, un dentro de alcance de *exist* I y otro fuera de dicho alcance:

```
ASK. (always display(I)) &
     (I=0) & (I gets I+1) & (halt I=2) &
     (exist I:( (I=1) & (I gets 3*I) & always display(I) )) ?
State 0:
I=1  I=0
State 1:
I=3  I=1
State 2:
I=9  I=2
ASK.
```

Con la instrucción de cuantificación universal *all* u<e: w, es posible ejecutar todas las fórmulas que resultan de sustituir en w cada uno de los posibles valores de u (w[c/u], c=0,1,...,e). Esta instrucción resulta útil cuando se necesita manejar todos los elementos de una lista en forma homogénea:

```
ASK. (all x<3: display(x)) & empty ?
State 0:
x=0  x=1 x=2
ASK.

ASK. (m=3) & (n=2) & (always  display(L)) &
     len(n) & (always list(L,m)) &
     (all k<m: ( (L[k]=1) & (L[k] gets (k+1)*L[k]) ) ) ?
State 0:
L= {1 1 1}
```

```
State 1:
L= {1 2 3}
State 2:
L= {1 4 9}
ASK.
```

Finalmente, presentamos un ejemplo de ejecución de la instrucción *execute* que permite ejecutar un programa contenido en un archivo de texto. Se puede usar un editor de archivos de texto que no introduzca caracteres de control para construir el archivo de nombre "Ejem" que contenga las siguientes tres líneas:

```
len(1) &
(always X=1) &
(always display(X)) ?
```

y ejecutar "Ejem" de la siguiente manera:

```
ASK. Exec 'Ejem' & empty ?
State 0:
State [Ejem]0:
X= 1
State [Ejem]1:
X= 1
ASK.
```

o bien, redireccionando la salida hacia el archivo SalTxt:

```
ASK. Exec 'Ejem','SalTxt' & empty ?
State 0:
ASK.
```

3.3.2 Ejemplos de programas

La cualidad principal de Tempura es su cercanía sintáctica y semántica a la LTI. Esto permite que un programa pueda ser visto como una fórmula que plantea las características lógico-temporales del problema que el programa resuelve. La dualidad programa-fórmula de los programas de Tempura ayuda a entender la naturaleza de los problemas y se traduce en una herramienta útil en el estudio y solución de los mismos.

Tempura resulta útil en la solución de problemas tanto de

software como de hardware. B.C. Moszkowski[3] muestra las capacidades de Tempura presentando programas que resuelven varios problemas. A continuación presentamos los listados de estos programas en la versiones que ejecuta nuestro intérprete. Se incluye con los listados una descripción breve de los problemas que resuelven y los nombres de los archivos que contienen a los listados.

1) Sumar las hojas de un árbol binario de enteros.

Dado un árbol binario de enteros, Tree, entregar la suma de sus hojas.

Solución serial (listado Q711):

```
(predicate serial_sum_tree(Tree) :
  if TypeOf(Tree)=TReal then empty
  else(
    sum_subtree(Tree,0);
    sum_subtree(Tree,1);
    (
      skip&(Tree tass Tree[0]+Tree[1] )
    )) )&
(predicate sum_subtree(Tree,i) :
  serial_sum_tree(Tree[i])
  &stable_struct Tree
  &stable Tree[1-i]  )&

Tree = {{{3,3},{{4,4},3}},1}
     &serial_sum_tree(Tree)&always display(Tree)?
```

Solución paralela (listado Q712):

```
(predicate par_sum_tree(Tree) :
  if TypeOf(Tree)=TReal then empty
  else(
       (exist Done:(
```

[3] [Moszkowski 86].

```
            (Done tequ
   ((TypeOf(Tree[0])=TReal)and(TypeOf(Tree[1])=TReal)) )
            & halt Done
            & stable_struct Tree
            & sum_tree_process(Done,Tree[0])
            & sum_tree_process(Done,Tree[1]) ));
          (skip & (Tree tass Tree[0]+Tree[1]) )
        ) )&
   (predicate sum_tree_process(Done,Tree) :
       process(
           par_sum_tree(Tree);
           (halt Done & stable Tree)
           ) )&

   Tree = {{{3,3},{{4,4},3}},1}
        &par_sum_tree(Tree) &always display(Tree)?
```

2) Ordenar una lista de enteros.

Dada una lista de enteros, L, entregarla ordenada.

Solución serial (listado Q731):

```
  (predicate partition_list(L,key,left_len) :
    exist I,J:(
          (I=0)&(J= #L) &(left_len=-1)
          &( part_loop(L,key,I,J);
             (empty&(left_len=I)) ))  )&
  (predicate part_loop(L,key,I,J) :
    while I<J do (
      skip&part_step(L,key,I,J))  )&
  (predicate swap_list(L,i,j) :
    all k< #L:
       (if k=i then (L[k] tass L[j])
        else if k=j then  (L[k]  tass  L[i])
             else  (L[k]  tass L[k])) )&
  (predicate part_step(L,key,I,J) :
    if L[I]< key then (
      stable L
```

```
      &(I tass I+1)&(J tass J) )
    else (
      swap_list(L,I,J-1)
      &(I tass I)&(J tass J-1) )  )&


  (predicate serial_quicksort(L) :
    if (#L) <= 1 then empty
    else exist pivot:(
          quick_partition(L,pivot);
          serial_sort_parts(L,pivot) )  )&
  (predicate serial_sort_parts(L,k) : (
    (serial_quicksort(L[0:k])
     & stable L[k: #L] );
    (serial_quicksort(L[k+1: #L])
     & stable L[0:k+1] ) )  )&
  (predicate quick_partition(L,pivot) : (
    (partition_list(L[0: (#L)-1],L[(#L)-1],pivot)
    & stable L[(#L)-1] );
    (skip& swap_list(L,pivot,(#L)-1) ) ) )&

  fixed_list(L,3)& fixed_list(T,3)& (always display(L,T))
    &(L={2,0,1})& serial_quicksort(L)
    & all i<#L : (T[i] tequ (if i=L[i] then 1 else 0))?


Solución paralela (listado Q732):
  (predicate partition_list(L,key,left_len) :
    exist I,J:(
          (I=0)&(J=#L) &(left_len=-1)
          &( part_loop(L,key,I,J);
             (empty&(left_len=I)) ))  )&
  (predicate part_loop(L,key,I,J) :
    while I<J do (
      skip & part_step(L,key,I,J))  )&
  (predicate swap_list(L,i,j) :
    all k<#L:
       (if k=i then (L[k] tass L[j])
```

```
        else if k=J then  (L[k]  tass  L[1])
             else  (L[k]  tass L[k])) )&
    (predicate part_step(L,key,I,J) :
      if L[I]< key then
        stable L
        &(I tass I+1)&(J tass J)
      else (
        swap_list(L,I,J-1)
        &(I tass I)&(J tass J-1) )  )&
    (predicate quick_partition(L,pivot) : (
      (partition_list(L[0:(#L)-1],L[(#L)-1],pivot)
      & stable L[(#L)-1] );
      (skip& swap_list(L,pivot,(#L)-1) ) ) )&

    (predicate par_quicksort(L) :
      if (#L) <= 1 then empty
      else exist pivot:(
             quick_partition(L,pivot);
             par_sort_parts(L,pivot) )  )&
    (predicate par_sort_parts(L,pivot) : (
      (exist Done,Ready1,Ready2: (
        (Done tequ (Ready1 and Ready2))
        & halt Done
        & sort_process(Done,Ready1,L[0:pivot])
        & sort_process(Done,Ready2,L[pivot+1:#L]) ))
      &stable L[pivot] )  )&
    (predicate sort_process(Done,Ready,L) :
      process( ((par_quicksort(L)&(Ready tequ empty) );
               (halt Done)&(stable L)&(stable Ready) ) )  )&

    fixed_list(L,3)& fixed_list(T,3)& (always display(L,T))
      &(L={2,0,1})& par_quicksort(L)
      & all i<#L: (T[i] tequ (if i=L[i] then 1 else 0))?
```

3) Modelar un circuito que multiplica enteros.

Simular el comportamiento de algunos componentes tales como multiplexores, decrementadores, tests, flipflops, etc. y su

interconexión para producir un circuito que multiplica enteros.

Solución (listado Q741):

```
(predicate mux(Switch,In1,In2,Out) :
  always(if Switch then Out=In1 else Out=In2)  )&
(predicate reg(In,Out) : (Out gets In)  )&
(predicate flipflop(In,Out) : (Out gets In)  )&
(predicate dec(In,Out) :
  (Out tequ (if In=0 then 0 else In-1))  )&
(predicate adder(In1,In2,Out) : (Out tequ(In1+In2))  )&
(predicate zero_test(In,Out) :  (Out tequ (In=0))  )&
(predicate or_gate(In1,In2,Out) : (Out tequ (In1 or In2))  )&
(predicate zero(Out) : (Out tequ 0)  )&

(predicate mult_imp(in1,in2,Done,Out) :
  exist B1,B2,B3,B4,L1,L2,L3,L4,L5,L6,L7,L8,L9,L10: (
        (Done=true)&(Out=0)&(L2=0)
        &mux(Done,L9,L8,L7)
        &reg(L7,Out)
        &adder(L9,Out,L8)
        &dec(in1,L6)
        &mux(Done,L6,L4,L5)
        &mux(Done,in1,L3,L1)
        &reg(L1,L2)
        &dec(L2,L3)
        &dec(L3,L4)
        &zero(L10)
        &mux(B4,L10,in2,L9)
        &zero_test(in1,B4)
        &zero_test(L5,B1)
        &zero_test(in2,B2)
        &or_gate(B1,B2,B3)
        &flipflop(B3,Done)
        )  )&

mult_imp(4,9,Done,Out)
  &(snext halt Done)& always display(Done,Out)?
```

4) Modelar un generador de pulsos.

Simular el comportamiento de ondas digitales con determinadas características. Por ejemplo, cuatro ondas (W,X,Y,Z) tales que W inicia en cero y cambia de valor cada 4 unidades de tiempo; X inicia en 0 y después se comporta como W pero retrasada una unidad de tiempo; Y se comporta respecto a X como X respecto a W; Z es el producto de W,X y Y.

```
Solución (listado Q751):
  (W=false)&(X=false)&(Y=false)&
  (trough 5 do (
    (len(3)&(stable W));
    (skip& (W tass NOT W)) ) )
  &(X gets W)&(Y gets X)
  &(Z tequ (W and X and Y))
  &always display(Z,Y,X,W)?
```

5) Modelar un flipflop (SR-latch).

Simular el comportamiento de un latch SR mediante compuertas NOR.

```
Solución (listado Q761):
  (S=false)&(R=false)&(Q=false)&(Qbar=false)&
  (for l in
  {{true,false},{false,false},{false,true},{true,false},
   {false,false}}
  do
    ( len(5)&(S gets l[0])&(R gets l[1]) ) )
  &(Q gets not(R or Qbar))
  &(Qbar gets not(S or Q))
  &always display(Qbar,Q,R,S)?
```

6) Comunicación sincronizada entre procesos paralelos.

Sin mostrar los detalles de implementación, B.C. Moszkowski menciona otra aplicación de Tempura que consiste en realizar un mecanismo que modele comunicación sincronizada entre procesos paralelos. B.C. Moszkowski llama *streams* al mecanismo que implementa e ilustra su uso con un ejemplo que resuelve la

evaluación de expresiones aritméticas mediante los siguientes procesos:

a) análisis léxico,
b) análisis sintáctico,
c) conversión a notación polaca y
d) evaluación de la expresión.

Se establece comunicación mediante streams entre los pares de procesos: (a,b),(b,c) y (c,d).

3.4 Implementación

En esta sección se presentan las ideas principales de la implementación de Tempura que describe B.C. Moszkowski[1]. La idea principal de esta implementación consiste en ejecutar una fórmula w buscando un intervalo que la realice. Para esto, se usa una estrategia de evaluación cuyo núcleo es una transformación de fórmulas T. La aplicación sucesiva de T permite encontrar un intervalo que satisface a w y la ejecución de w termina al encontrar un intervalo que la realiza. Aquí se define T solamente para un subconjunto de Tempura porque la definición en el caso general contiene muchos detalles técnicos que obscurecen la idea principal[2]. Al final se ilustra el funcionamiento T con algunos ejemplos.

En lo que se refiere a implementaciones de Tempura, Moszkowski reporta la realización de un intérprete de Tempura en Lisp. Menciona también la existencia de un intérprete en C desarrollado por Roger Hale en la universidad de Cambridge y de otro en Prolog desarrollado por Masahiro Fujita en la universidad de Tokio. El intérprete que acompaña a este trabajo fue desarrollado en la versión 6.0 de Pascal de Borland para correr en PC's bajo MS-Dos.

3.4.1 Estrategia de evaluación

Hemos definido anteriormente la semántica de Tempura en términos lógicos mediante interpretaciones temporales. Esta semántica tiene el defecto de no ser constructiva. Permite saber si una fórmula es satisfacible en un intervalo dado, pero no ayuda a encontrar intervalos en los que una fórmula sea satisfacible. Es decir, esta semántica no contiene explícitamente

---

[1] [Moszkowski 86].

[2] El apéndice A contiene una definición completa de T al estilo de la semántica denotacional.

un método para construir intervalos en los que una fórmula dada sea satisfacible. Nos referimos a los métodos que permiten construir intervalos para satisfacer una fórmula como *estrategias de evaluación*.

La idea principal para evaluar (ejecutar) una fórmula temporal W consiste en analizar W dividiéndola en dos partes:

Parte *actual* que se refiere al instante actual y

Parte *futura* que se refiere a posibles instantes futuros.

La parte actual permite construir el primer estado. Hecho esto, pasamos al siguiente instante y procedemos con la parte futura igual que con W, la dividimos en parte actual y parte futura. Ahora la parte actual permite construir el segundo estado. Se procede sucesivamente de la misma manera construyendo estados en cada instante hasta encontrar una condición que indique que ya no hay más instantes futuros.

El proceso que sigue la estrategia de evaluación para evaluar (ejecutar) fórmulas temporales se resume en los siguientes pasos:

1) W ⟵ la fórmula a evaluar. N ⟵ 0.
2) *Construir* el estado S_N extrayendo de W las condiciones que deben cumplir la constante *empty* y las variables de W en el estado inicial y *Transformar* W a una fórmula de la forma ⊖ G.
3) De acuerdo a las condiciones obtenidas para la constante *empty* hacer lo siguiente:
   Si no hay condiciones sobre *empty* indicar Error.
   Si *empty* debe ser *cierta* terminar el proceso.
   Si *empty* debe ser *falsa*:
   W ⟵ G . N ⟵ N+1. Repetir el paso 2.

Si este proceso termina con N=k, el intervalo que satisface a W será $\langle S_0, S_1, \ldots, S_k \rangle$.

Es importante acotar que el valor de las variables locales se construye en cada estado mientras que el valor de las variables globales se construye solamente en el estado inicial (N=0) y se conserva en los siguientes estados.

La definición del proceso anterior queda incompleta si no se detalla lo que significan los términos *Construir* y *Transformar*. Para precisar estos términos comenzamos con una definición parcial semi-formal de una transformación que opera sobre parejas fórmula-estado. Aunque esta definición no es exacta ni completa, permite ver en forma clara las ideas generales. En el Apéndice A se encuentra la semántica denotacional de Tempura y [Moszkowski 86] presenta una definición semi-formal y completa de la implementación de Tempura.

Más adelante necesitaremos otra definición:

Si $f:A \longrightarrow B$ es una función, $b \in B$ y $x \in A$, definimos f *con valor* b *en lugar de* x, $f[b/x]$, como:

$f[b/x]:A \longrightarrow B$

$f[b/x](z) = b$ si $z=x$

$= f(z)$ si no.

En lo que sigue, usaremos T, F, Φ, $\mathbb{I}$ y $\perp$ en lugar de true, false, empty, error y valor indefinido respectivamente. Usamos los términos EXPRESIONES y FORMULAS en un sentido relajado para referirnos a un par de subconjuntos EXPT y FORMT que se ajustan a las necesidades de esta sección. Usamos el término ESTADOS para referirnos a *estados* que le asignan valores a las variables y a la constante local *empty*.

Primero se necesita una función que entregue el *valor* de una expresión en un estado dado:

E: EXPRESIONES x ESTADOS $\longrightarrow$ Bv

$E(c,s) = c$

$E(v,s) = s(v)$

$E(\Phi,s) = s(\Phi)$

$E(f(e_1,e_2,\ldots,e_n), s) = f(E(e_1,s),E(e_2,s),\ldots,E(e_n,s))$

La transformación de parejas *fórmula-estado* se define como sigue:

T: *FORMULAS* x *ESTADOS* $\longrightarrow$ *FORMULAS* x *ESTADOS* .

$T(T,\ s) = (\ominus\ T, s)$

$T(F,\ s) = (\ominus\ F,\ s[\,!/\Phi\,])$

$T(\Phi,\ s) = (\ominus\ T, s[T/\Phi])$

$T(more, s) = (\ominus\ T, s[F/\Phi])$

$T(v=a, s) = (w', s')$

donde

$w' = v=a$ si $E(a,s)=\perp$

$= \ominus\ T$ si no.

$s' = s$ si $E(a,s)=\perp$

$= s[E(a,s)/\ v]$ si $E(a,s)\neq\perp$ y $s(v)=\perp$

$= s[\,!/\ \Phi\,]$ si $E(a,s)\neq\perp$ y $s(v)\neq\perp$

$T(O\ l=a, s) = (w', s)$

donde

$w' = (\textit{if more then } \ominus\ (l = E(a,s))\ \textit{else}\ F)$ si $E(a,s)\neq\perp$

$= O\ l=a$ si $E(a,s)=\perp$

$T(f \wedge g, s) = (w, s'')$

donde

$T(f,s) = (f', s')$

$T(g,s') = (g', s'')$

$w = \ominus\ (f'' \wedge g'')$ si $f' = \ominus\ f'' \wedge g' = \ominus\ g''$

$= f' \wedge g'$ si no.

(Una optimización sería :

$w = \ominus\ (f'' \wedge g'')$ si $f' = \ominus\ f'' \wedge g' = \ominus\ g''$

$= \ominus\ T$ si $f' = T \wedge g' = T$

$= g'$ si $f' = T \wedge g' \neq T$

$= f'$ si $f' \neq T \wedge g' = T$

$= f' \wedge g'$ si no.)

$T(\ominus\ w, s) = (\ominus\ w, s)$

$T(O\ w, s) = T(more \wedge \ominus\ w,\ s)$

$T(\Box w,s) = T(w \wedge \ominus(\Box w), s)$

$T(\mathit{if}\ a\ \mathit{then}\ f\ \mathit{else}\ g,\ s) =$

$= (\mathit{if}\ a\ \mathit{then}\ f\ \mathit{else}\ g,\ s)$	si $E(a,s)=\perp$
$= T(f,s)$	si $E(a,s)=T$
$= T(g,s)$	si $E(a,s)=F$
$= (\ominus T,\ s[!/\Phi])$	si no.

La definición de T en los casos restantes no es directa. Se requiere modificar el dominio T para manejar conceptos (ambiente local, parámetros por referencia, longitud de intervalo local, etc.) que se necesitan para definir T en los casos de predicados, funciones y fórmulas que contienen cuantificadores o al operador *corta*. El apéndice A aborda el problema de la definición de esta transformación desde un punto de vista más formal.

3.4.2 Ejemplos de evaluación

En esta sección se ilustra el funcionamiento de la transformación T mediante una tabla que contiene un resumen de las transformaciones que resultan al evaluar algunas fórmulas sencillas. Seguimos usando T, F, Φ, ! y ⊥ en lugar de true, false, empty, error y valor indefinido respectivamente.

N	W	S_N
0	(empty & X=1)	{(Φ,⊥)(X,⊥)}
0	⊖ T & ⊖ T	{(Φ,T)(X,1)}
0	⊖ (T)	{(Φ,T)(X,1)}
0	len(1)& a=3& A=0& nextv A=1	{(Φ,⊥)(a,⊥)(A,⊥)}
0	⊖ len(0)& ⊖ T& ⊖ T& ⊖ (A=1)	{(Φ,F)(a,3)(A,0)}
0	⊖ (len(0)& T& T& A=1)	{(Φ,F)(a,3)(A,0)}
1	len(0)& T& T& A=1	{(Φ,⊥)(a,3)(A,⊥)}
1	⊖ T& ⊖ T& ⊖ T& ⊖ T	{(Φ,T)(a,3)(A,1)}
1	⊖ (T)	{(Φ,T)(a,3)(A,1)}
0	X=1& always(if X=3 then empty else (more & nextv X = X+1))	{(Φ,⊥)(X,⊥)}
0	⊖ T & (⊖ T & ⊖ (X=2) & ⊖ □ w)	{(Φ,F)(X,1)}
0	⊖ (T & T & X=2 & □ w)	{(Φ,F)(X,1)}
1	T & T & X=2 & □ w	{(Φ,⊥)(X,⊥)}
1	⊖ T & (⊖ T & ⊖ (X=3) & ⊖ □ w)	{(Φ,F)(X,2)}
1	⊖ (T & T & X=3 & □ w)	{(Φ,F)(X,2)}
2	T & T & X=3 & □ w	{(Φ,⊥)(X,⊥)}
2	⊖ T & (⊖ T & ⊖ T & ⊖ □ w)	{(Φ,T)(X,3)}
2	⊖ (T & T & T & □ w)	{(Φ,T)(X,3)}

3.5 Ubicación

En esta sección se dan algunos comentarios que sirven para ubicar a Tempura en relación a otros lenguajes de programación. Tal vez el mejor punto de partida para clasificar a Tempura dentro de la familia de lenguajes de programación es el que establece que este lenguaje intenta reducir la brecha que existe entre un lenguaje de programación (L) y el formalismo de la lógica temporal (LT) en la tarea de probar propiedades de un programa escrito en L[1]. Ya que Tempura y LT difieren muy poco, estas dos notaciones (L y LT) se reducen a una sola simplificando la tarea de verificación de programas. En forma simple, Tempura es un lenguaje imperativo cuya sintaxis y semántica están basadas en la lógica temporal. Esto ubica a Tempura en un lugar que queda entre los lenguajes imperativos y los lenguajes de programación lógica.

3.5.1 Características imperativas

Tempura es un lenguaje imperativo cuyo conjunto de instrucciones permite indicar cuando debe ejecutarse una instrucción . En particular, no sólo permite asignarle valores a las variables sino que además es posible indicar en qué instante de la ejecución debe hacerse esto.

Tempura cuenta también con instrucciones de iteración como *for* y *while*, entre otras, que funcionan en forma similar a las instrucciones de iteración clásicas. La semántica de estas instrucciones se define mediante el operador *corta* (;) que permite componer secuencialmente la ejecución de dos instrucciones o bien indicar que una instrucción es prefijo o sufijo temporal de otra.

La utilización de rutinas es posible en Tempura mediante la definición de predicados y funciones. Estas utilizan llamada por

---

[1]Las primeras aplicaciones de la lógica temporal fueron en la verificación de programas concurrentes [Manna-Pnueli 81].

referencia en el paso de parámetros y pueden ser recursivos o pasar en un parámetro la referencia a un predicado o una función.

En lo que se refiere al uso de distintos alcances (scope) de las variables para el procesamiento de un grupo de instrucciones, Tempura cuenta con las instrucciones de cuantificación $\forall$ u < e : w y $\exists$ v_1, v_2, .., v_n : w que permiten fijar el alcance de una variable dentro de la fórmula cuantificada (w).

Otra característica importante de Tempura como lenguaje imperativo es la de ejecución concurrente. En cierta forma, Tempura procesa cada uno de los componentes de la instrucción r $\wedge$ s en forma concurrente con un mecanismo de sincronización que consiste en evaluar r y s determinando los valores de las variables en cada estado con la suposición de que una variable puede tener un y sólo un valor en cada estado. De tal manera que dada una variable X sólo una de las instrucciones (r o s) puede determinar su valor en un estado cualquiera.

3.5.2 Características lógicas

Cometiendo un abuso de particularización, usaremos a Prolog como representante de los lenguajes de programación lógica para compararlo con Tempura. Las comparaciones que haremos tienen como propósito fijar un punto de referencia que permita ver las ventajas y desventajas de Tempura respecto a los lenguajes de programación lógica y no intentan establecer que uno es absolutamente mejor o peor que el otro. Las siguientes son las principales diferencias entre Prolog y Tempura:

Tempura no tiene incorporado ningún mecanismo probador de teoremas como el algoritmo de resolución que emplea Prolog. Tampoco cuenta con técnicas de backtracking de hecho, los operadores $\vee$ y $\Diamond$ fueron eliminados de Tempura por necesitarse técnicas de backtracking para su evaluación. Sin embargo, cabe mencionar que parece muy posible el diseño de un lenguaje que incorpore una extensión del algoritmo de resolución a un

subconjunto de la lógica temporal[2].

Los operadores temporales son los que permiten que Tempura maneje el valor de las variables con la forma dinámica de los lenguajes imperativos pero mediante un lenguaje lógico. Prolog no cuenta con estos operadores debido a que su base es un subconjunto de la lógica clásica llamado cláusulas de Horn. Esto le impide contar con instrucciones de tipo imperativo tales como la asignación. Aunque en la práctica construcciones como *assert* y *retract* compensan la carencia de instrucciones de asignación, su uso no está aprobado por un sector de los partidarios de la programación lógica.

Otra diferencia entre Prolog y Tempura es su forma de evaluación. Prolog evalúa una consulta (query) tratando de establecer bajo qué condiciones (valores de las variables) ésta se deduce de la información contenida en las cláusulas proporcionadas (hechos y reglas). Por otro lado, Tempura evalúa una fórmula tratando de construir un intervalo donde ésta sea cierta. La construcción se realiza mediante una serie de transformaciones de la fórmula que determinan qué debe ejecutarse en el instante actual y qué debe posponerse al siguiente estado. Esta diferencia se comprende mejor si se observa que la definición de predicados en Prolog es declarativa y en Tempura es imperativa.

Finalmente, es importante recalcar que aunque Tempura no es un lenguaje de programación lógica en el sentido de que no es un probador de teoremas, puede considerarse que sí lo es en vista de que su lenguaje es un subconjunto de la lógica temporal.

3.5.3 Características funcionales

Al comparar Tempura con los lenguajes de programación funcional es útil señalar que estos también carecen de operadores

---

[2] [Abadi 87], [Abadi-Manna 85]

de tiempo. Por esta razón, en la programación funcional, el comportamiento en el tiempo de una variable tiene que ser modelado de manera indirecta mediante: recursión, una lista de valores, o una función con un parámetro de tiempo. Algunos lenguajes funcionales incorporan construcciones que permiten asignación de valores u otras operaciones imperativas como las instrucciones *setq* y *prog* de Lisp, pero esta construcciones no se apegan al paradigma de los lenguajes funcionales y su uso no es totalmente aceptado.

Tempura no tiene el poder de expresión funcional ni la flexibilidad en los tipos de datos que caracterizan a los lenguajes funcionales. Los mecanismos de Tempura para definir predicados y funciones son relativamente simples y rígidos comparados con los mecanismos de los lenguajes funcionales. Sin embargo el poder expresivo de estos mecanismos aumenta al combinarlos con los operadores de tiempo. Esto puede apreciarse cuando se necesita resolver un problema que tiene rasgos temporales.

Tal vez el atractivo principal de los lenguajes de programación lógica y los lenguajes de programación funcional es su *elegancia matemática* y su capacidad de expresión superiores a las de los lenguajes imperativos tradicionales. Desde este punto de vista, puede decirse que Tempura se ubica en una posición intermedia al permitir programación imperativa sin perder del todo la *elegancia matemática*.

CONCLUSIONES

Hemos resumido en este escrito el trabajo que consistió en el estudio de la lógica temporal y en la realización del intérprete de programas lógico-temporales, Tempura, que se describe en [Moszkowski 86]. En seguida mencionamos los resultados que produjo esta experiencia.

4.1 Resultados y observaciones

El resultado más palpable de este trabajo es el intérprete de Tempura que fue desarrollado en Pascal para correr en PC's compatibles con IBM bajo MS-DOS. La realización de este intérprete no fue solamente un ejercicio de programación, su desarrollo y la experimentación con él resultaron ser una buena herramienta de aprendizaje práctico. Poder ejecutar programas lógico-temporales y ver los resultados que producían sirvió para hacer a un lado ciertas especulaciones y permitió ver en forma concreta algunos detalles de la semántica de Tempura. Como cualquier lenguaje, Tempura se conoce mejor a través de la práctica.

Actualmente nuestro intérprete ejecuta exitosamente todos los ejemplos que se presentan en [Moszkowski 86] excepto aquellos del capítulo 9 que se refieren a algunas construcciones experimentales de Tempura tales como el operador de proyección temporal *proj*, expresiones lambda, apuntadores y el operador *prefix*. De las construcciones que se mencionan en dicho capítulo la instrucción *process* es la única que tenemos implementada actualmente debido a que Moszkowski la utiliza en el capítulo de aplicaciones. Nuestro intérprete fue desarrollado siguiendo casi fielmente la descripción contenida en [Moszkowski 86] y la ejecución de los programas que ahí se muestran han servido como una prueba práctica de que el mismo funciona correctamente.

En la mayoría de los casos, las aplicaciones de la lógica en el campo de la Inteligencia Artificial resultan en definiciones semánticas semi-formales que no progresan hacia una teoría formal[1]. Estamos convencidos de que el desarrollo de aplicaciones prácticas resulta más sólido a la larga si se hace tomando como base una teoría formal y no solamente un conjunto de ideas más o menos precisas. Guiados por esta convicción empezamos a explorar los sistemas axiomáticos de la LTI y a definir la semántica de Tempura al estilo de la semántica denotacional.

4.2 Trabajo futuro

Antes de proponer el estudio posterior de algunos puntos relacionados con este trabajo es conveniente mencionar las metas que plantea Moszkowski que en relación a Tempura y a la LTI: construir un compilador de Tempura, definir una semántica operacional de Tempura en la LTI que permita formalizar la relación entre diversas estrategias de ejecución tanto secuenciales como paralelas, generalizar la LTI permitiendo intervalos infinitos para poder manejar computaciones que no terminan y desarrollar la teoría de la demostración (proof theory) de la LTI. En otra parte de su trabajo menciona también la posibilidad de incorporar resolución y backtracking a una variante de Tempura.

Consideramos de interés todas las metas planteadas por Moszkowski y algunas otras que delineamos a continuación.

Algo que hemos notado al estudiar el intérprete de Tempura es que aunque la TTI permite la ocurrencia del operador O en fórmulas tales como X= O Y, p(O X) y *if* O X < 2 *then* Z=1, Tempura no puede evaluarlas. El uso de expresiones en Tempura está

---

[1] [Turner 84] y [Smets 88] contienen comentarios en este sentido.

restringido a localidades (LOC) para el miembro izquierdo de las igualdades y a expresiones sin el operador O (EXPT) en en resto de los casos. Es posible eliminar esta restricción y aceptar todas las expresiones de la TTI (EXPD) en Tempura. En tal caso, la evaluación se haría con la ayuda de las transformaciones:

$f(O\,\alpha,\ \beta, \ldots) \longmapsto O\,(f(\alpha,\ c_\beta, \ldots))$,
$O\,\alpha \in EXPD,\ \beta \in EXPT,\ c_\beta = M[\beta] \in D$

$O\,\alpha = O\,\beta \longmapsto O\,(\alpha = \beta)$

$V = O\,\beta \longmapsto O\,\beta = V\ ,\ V \in VAR \cup D$

Esta extensión permitiría evaluar, por ejemplo, X= O Y, X= O Y+Z y O X= O Y.

El problema de la evaluación de la igualdad es un caso particular de la evaluación de predicados con parámetros temporales. En el caso de Tempura los predicados deben definirse con una expresión del tipo *predicate* $p(v_1, v_2, \ldots, v_n)=w$ de tal manera que a final de cuentas quedan definidos en términos de la igualdad. La igualdad es el único predicado que no necesita definición en Tempura y los demás predicados están definidos con base en la igualdad. En Tempura los parámetros de una llamada a un predicado están restringidos a expresiones sin el operador O (EXPT). Una extensión a Tempura sería permitir que dichos parámetros fueran expresiones de la TTI (EXPD). En tal caso, la evaluación de una llamada de predicado sería con la ayuda de la siguiente transformación:

$p(O\,\alpha,\ \beta,\ \ldots) \longmapsto O\,(p(\alpha,\ c_\beta,\ \ldots))$
$O\,\alpha \in EXPD,\ \beta \in EXPT,\ c_\beta = M[\beta] \in D$

Con esta extensión Tempura podría ejecutar programas como el siguiente:

```
ASK. (predicate p(X,Y)= X=Y ∧ O Y= Y+2) ∧
     len(1) ∧ X=0 ∧ Y=1 ∧ p(O X, Y) ∧ □ display(X,Y) ?
State 0: X=0 Y=1
State 1: X=1 Y=3
ASK.
```

En el caso de las formas condicionales *if e the* w_1 *else* w_2 y *if e then* e_1 *else* e_2 el *test e* se restringe a EXPT. No vemos ninguna posibilidad de que este *test* se extienda a EXPD debido a que la evaluación de estas formas requeriria conocer el valor de las variables en más de un instante de la ejecución del programa.

Actualmente los únicos mecanismos de entrada/salida de Tempura son el teclado y el display. Uno de los factores que impiden a un lenguaje pasar del campo de la investigación al área de aplicación es la carencia de instrucciones para manejar archivos de datos. Tempura podria extenderse para manejar datos de archivos secuenciales o indexados. Más aún, tal vez sea posible definir un concepto de base de datos temporal en la que existan operaciones para recuperación y almacenamiento que incluyan operadores temporales. En un modelo tal los datos tendrian un atributo que indicaría en que instante son válidos.

Dentro del campo de la Inteligencia Artificial, podrian darse aplicaciones de Tempura o de la LTI en el área de la representación de conocimiento. Las aplicaciones de la lógica temporal en esta área ya han sido consideradas por varios investigadores [2].

La variedad de conceptos que exhiben N. Rescher y A. Urquhart[3] es suficiente para darse cuenta que la LTI es una lógica que captura solamente algunos de los conceptos de la lógica temporal. La LTI es un modelo particular y simplificado de algunos aspectos temporales de algunos mecanismos de razonamiento. Este modelo puede ampliarse o modificarse incluyendo varios aspectos filosóficos de la lógica temporal que normalmente se ignoran para

---

[2] [Smets 88]

[3] [Rescher-Urquhart 71]

lograr claridad y simplicidad. La interacción de este modelo ampliado con la computación podría producir resultados de mayor impacto que los que se ilustran en este trabajo. Tampoco se debe olvidar que al ubicarse dentro del panorama más amplio de las lógicas no-clásicas, las posibilidades de aplicación aumentan.

Tal vez existe gente que cree en la falacia de que *se pueden desarrollar sistemas haciendo abstracción del concepto de tiempo para evitar arrastrar detalles insignificantes o tediosos y al final agregar este concepto*, pero estamos convencidos de que incluir los aspectos temporales, al nivel de la lógica, en el planteamiento de un problema puede resultar decisivo en que la solución buscada sea la apropiada.

TEMPURA

Este apéndice contiene un resumen de Tempura que incluye la transliteración de símbolos, la sintaxis y la semántica.

A.1 Sintaxis

En esta sección presentamos la transliteración de símbolos y la sintaxis de Tempura. Esta sintaxis es la que se conoce como sintaxis abstracta y su único fin es especificar en forma simple la relación entre los elementos del lenguaje. La sintaxis abstracta puede ser ambigua y no contener información suficiente para realizar automáticamente el análisis sintáctico. Para esto se necesita lo que se conoce como sintaxis concreta del lenguaje.

A.1.1 Transliteración

La siguiente tabla muestra la forma en que se introducen los símbolos especiales de Tempura mediante el teclado de una PC.

Tempura	Teclado
$\Box$ w	*always* w.
O w	*snext* w.
(O l) = e	*nextv* l = e
$w_1 \wedge w_2$	w_1 & w_2.
l ≈ e	l *tequ* e.
l ⟵ e	l *tass* e.
∃ v: w	*exist* v: w.
∀ u < e: w	*all* u < e: w.
e ≤ d	e <= d
e ≥ d	e >= d
e ≠ d	e <> d

A.1.2 Expresiones

```
<Expresión> ::
     <Constante>/ <Variable>/ <Función>/ ( <Expresión> ) /
     if <Expresión> then <Expresión> else <Expresión>

<Constante>     :: <Constante local>/ <Constante global>
<Variable>      :: <Variable local>/ <Variable global>
<Función>       :: <Función definida> / <Función predefinida>

<Constante local>   :: empty/ more
<Constante global>  :: <Atomo> / <Lista>
<Variable local>    :: <Mayúscula><Letras o dígitos>
<Variable global>   :: <Minúscula><Letras o dígitos>
<Función definida>  :: <Variable local> <Parámetros actuales>
<Función predefinida> ::
     <Op unario> <Expresión> /
     <Expresión> <Op binario> <Expresión> /
     <Expresión> <Indice> /
     { <Expresión> :
          <Expresión> <= <Variable local> < <Expresión> } /
     { <Expresión> : <Variable local> < <Expresión> }

<Atomo>   :: <Número>/ <Booleano>/ <Cadena>/ <Tipo>
<Lista>   :: {} / { <Expresiones> }
<Parámetros actuales>     :: () / ( <Expresiones> )
<Op unario>     :: <Opu booleano> / <Opu numérico> / <Opu lista>
<Op binario>    ::
     <Opb booleano> / <Opb numérico> / <Opb cadena> /
     <Opb Tipo>
<Expresiones>   :: <Expresión>/ <Expresión> , <Expresiones>
<Indice>  :: [ <Expresión> ] / [ <Expresión> : <Expresión> ]

<Número>  :: <Dígitos>/ <Dígitos> . <Dígitos>/ <Dígitos><Escala>
<Booleano>      :: true/false
<Cadena>        :: '' / ' <Caracteres> '
<Tipo>          :: tboolean / treal / tstring / tlist / ttype
```

```
<Opu booleano> :: NOT
<Opu numérico> :: + / -
<Opu lista>    :: # / CAR / CDR
<Opb booleano> :: AND / OR / IMP / = / > / ≥ / < / ≤ / <>
<Opb numérico> ::
    + / - / * / / / ^ / MOD / % / = / > / ≥ / < / ≤ / <>
<Opb cadena>   :: + / = / > / ≥ / < / ≤ / <>
<Opb tipo>     :: = / > / ≥ / < / ≤ / <>

<Letras o digitos> :: <Let_dig> / <Let_dig><Letras o digitos>
<Digitos>      :: <Digito> / <Digito> <Digitos>
<Escala>       :: @ <Digitos> / @ <Signo> <Digitos>
<Caracteres>   :: <Carácter> / <Carácter> <Caracteres>

<Let_dig>      :: A/B/.../Z/a/b/.../z/_/0/1/.../9
<Mayúscula>    :: A/B/.../Z
<Minúscula>    :: a/b/.../z
<Digito>       :: 0/1/2/3/4/5/6/7/8/9
<Signo>        :: + / -
<Carácter>     ::   /.../} [1]
```

[1] caracteres ACII del blanco a } excepto '

A.1.3 Instrucciones

```
<Programa>      :: <Instrucción> ?
<Instrucción>  ::
     <Instrucción básica>/ <Instrucción derivada>/
     ( <Instrucción> )

<Instrucción básica> ::
     empty / more / true / false / quit /
     <Localidad> = <Expresión>/
     <Instrucción> & <Instrucción>/
     snext <Instrucción>/
     always <Instrucción>/
     <Instrucción> ; <Instrucción>/
     if <Expresión> then <Instrucción> else <Instrucción>/
     exist <Variables> : <Instrucción>/
     predicate <Variable local> <Parámetros formales> :
          <Instrucción>/
     function <Variable local> <Parámetros formales> :
          <Expresión> /
     <Predicado>/
     process <Instrucción>/
     exec <Expresión> / exec <Expresión> , <Expresión>

<Instrucción derivada> ::
     skip /
     if <Expresión> then <Instrucción> /
     <Localidad> gets <Expresión> /
     <Localidad> tequ <Expresión> /
     <Localidad> tass <Expresión> /
     stable <Variable local> /
     halt <Expresión> /
     fin <Instrucción> /
     while <Expresión> do <Instrucción> /
     repeat <Instrucción> until <Expresión> /
     loop <Instrucción> exit when <Expresión>
          otherwise  <Instrucción> /
```

```
    for <Expresión> times do <Instrucción> /
    for <Variable local> < <Expresión> do <Instrucción> /
    for <Variable local> in <Expresión> do <Instrucción> /
    all <Variable local> < <Expresión> : <Instrucción>

<Predicado>    ::
    <Predicado definido>/ <Predicado predefinido>
<Localidad>    ::
    <Variable>/ nextv <Localidad>/ <Localidad> <Indice>
<Parámetros formales> :: () / ( <Variables> )

<Predicado definido> :: <Variable local> <Parámetros actuales>
<Predicado predefinido> ::
    request <Lista localidades>/
    display <Parámetros actuales>/
    len (<Expresión>)/
    list (<Variable> , <Expresión>)/
    fixed_list <Variable>/
    stable_struct <Variable>
<Variables>    :: <Variable>/ <Variable> , <Variables>

<Lista localidades> :: () / ( <Localidades> )
<Localidades> :: <Localidad>/ <Localidad> , <Localidades>
```

A.2 Semántica abstracta

Presentamos aquí un resumen de la semántica Tempura con un estilo semi-formal que recurre a la semántica formal de la TTI y a descripciones informales.

En lo que sigue usamos z ∈ <Xxxx> para indicar que z cumple con la sintaxis definida por <Xxxx>. De tal manera que:

k ∈ <Atomo>,
w, w_1, w_2 ∈ <Instrucción>,
e, e_1, e_2 ∈ <Expresión>, l ∈ <Localidad>,
v ∈ <Variable>, V ∈ <Variable local>, u ∈ <Variable global>,
o^1 ∈ <Op unario>, o^2 ∈ <Op binario>
$e_1, e_2, \ldots, e_n$ ∈ <Expresiones> (n > 0),
$(v_1, v_2, \ldots, v_n)$ ∈ <Parámetros formales> (n ≥ 0),
$(e_1, e_2, \ldots, e_n)$ ∈ <Parámetros actuales> (n ≥ 0),
$(l_1, l_2, \ldots, l_n)$ ∈ <Lista localidades> (n ≥ 0).

A.2.1 Expresiones y Localidades

- $M_\sigma[empty]$ = true sii $|\sigma|=0$.
- $M_\sigma[more]$ = true sii $|\sigma|\neq 0$.
- $M_\sigma[k] = \gamma(k)$
- $M_\sigma[\{e_1, e_2, \ldots, e_n\}] = \{M_\sigma[e_1], M_\sigma[e_2], \ldots, M_\sigma[e_n]\}$
- $M_\sigma[v] = \sigma_0[v]$.
- $M_\sigma[u\ (e_1, e_2, \ldots, e_n)] = \gamma(u)(M_\sigma[e_1], M_\sigma[e_2], \ldots, M_\sigma[e_n])$
- $M_\sigma[o^1\ e] = M_\sigma[o^1] M_\sigma[e]$
- $M_\sigma[e_1\ o^2\ e_2] = M_\sigma[e_1]\ M_\sigma[o^2]\ M_\sigma[e_2]$
- $M_\sigma[e_1[e_2]] = M_\sigma[e_1]\ [M_\sigma[e_2]]$
- $M_\sigma[e_1[e_2 : e_3]] = M_\sigma[e_1]\ [M_\sigma[e_2] : M_\sigma[e_3]]$
- $M_\sigma[\{e_1 : e_2 \leq u < e_3\}]$ =
  = {} si $c_3 - c_2 \leq 0$
  = Cons(c_1, $M_\sigma[\{e_1 : c_2+1 \leq u < c_3\}$ si no.
  donde $c_1 = M_\sigma[e_1[c_2/u]]$, $c_2 = M_\sigma[e_2]$, $c_3 = M_\sigma[e_3]$.
- $M_\sigma[\{e_1 : u < e_2\}] = M_\sigma[\{e_1 : 0 \leq u < e_2\}]$
- $M_\sigma[if\ e_1\ then\ e_2\ else\ e_3] = M_\sigma[e_2]$ si $M_\sigma[e_1]$ = true,
  $= M_\sigma[e_3]$ si $M_\sigma[e_1] \neq$ true.
- $M_\sigma[nextv\ l] = M_{\sigma_1}[l]$.

A.2.2 Instrucciones

Instrucciones básicas:

$M_\sigma[\textit{empty}]$ = true sii $|\sigma|=0$.
$M_\sigma[\textit{more}]$ = true sii $|\sigma| > 0$.
$M_\sigma[\textit{true}]$ = true.
$M_\sigma[\textit{false}]$ = false.
$M_\sigma[\textit{quit}]$ = true (Abandona Tempura).
$M_\sigma[1 = e]$ = true sii $M_\sigma[1] = M_\sigma[e]$
$M_\sigma[w_1 \;\&\; w_2]$ = true sii $M_\sigma[w_1]$= true y $M_\sigma[w_2]$= true .
$M_\sigma[\textit{snext}\ w]$ = true sii $M_{\sigma^1}[w]$ = true
$M_\sigma[\textit{always}\ w]$ = true sii $\forall i(M_{(^i\sigma)}[w]$ = true).
$M_\sigma[w_1 \;;\; w_2]$ = true sii $\exists i(M_{(^i\sigma)}[w_1]$ = true y $M_{\sigma^i}[w_2]$ = true).
$M_\sigma[\textit{if}\ e\ \textit{then}\ w_1\ \textit{else}\ w_2]$ = $M_\sigma[w_1]$ si $M_\sigma[e]$ = true,
= $M_\sigma[w_2]$ si $M_\sigma[e] \neq$ true.
$M_\sigma[\textit{exist}\ v : w]$ = true sii $\exists \sigma'(\sigma' \propto_v \sigma$ y $M_{\sigma'}[w]$ = true)
$M_\sigma[\textit{request}\ (l_1, l_2, \ldots, l_n)]$ = true
(se solicitan valores para $l_1, l_2, \ldots, l_n$).
$M_\sigma[\textit{display}\ (e_1, e_2, \ldots, e_n)]$ = true
(se despliegan los valores de $e_1, e_2, \ldots, e_n$).
$M_\sigma[\textit{len}\ (e)]$ = true sii $|\sigma| = M_\sigma[e]$.
$M_\sigma[\textit{list}\ (v,e)]$ = true sii $M_\sigma[v]$ es una lista de longitud $M_\sigma[e]$.
$M_\sigma[\textit{fixed\_list}\ v]$ = true sii $M_\sigma[v]$ es una *lista fija* en σ.
$M_\sigma[\textit{stable\_struct}\ v]$ = true sii $M_\sigma[v]$ es una *estructura fija* en σ.
$M_\sigma[\textit{predicate}\ u\ (v_1, v_2, \ldots, v_n) : w]$ = true sii
$\gamma(u) = w_{(v_1, v_2, \ldots, v_n)}$.
$M_\sigma[\textit{function}\ u\ (v_1, v_2, \ldots, v_n) : e]$ = true sii
$\gamma(u) = e_{(v_1, v_2, \ldots, v_n)}$.
$M_\sigma[u\ (e_1, e_2, \ldots, e_n)]$ = true sii
$\gamma[u]\ (M_\sigma[e_1], M_\sigma[e_2], \ldots, M_\sigma[e_n])$ = true
$M_\sigma[\textit{process}\ w]$ = $M_\sigma[w]$

$M_\sigma[\textit{exec}\ e]$ = true
(Redirecciona la entrada Tempura hacia el archivo $M_\sigma[e]$).

M_σ[*exec* e_1 , e_2] = true
(Redirecciona la entrada y la salida de Tempura hacia los archivos $M_\sigma[e_1]$ y $M_\sigma[e_2]$ respectivamente)

Instrucciones derivadas:

skip $\equiv_{def}$ *snext empty* .
if e *then* w $\equiv_{def}$ *if* e *then* w *else true*.
l *gets* e $\equiv_{def}$ *always* (*if* more *then* ((*nextv* l) = e)).
l *tequ* e $\equiv_{def}$ *always* (l = e).
l *tass* e $\equiv_{def}$ *exist* u:((u = e) & *fin* (l = u)) .
stable V $\equiv_{def}$ V *gets* V.
halt e $\equiv_{def}$ *if* e *then empty else more* .
fin w $\equiv_{def}$ □ (*if empty then* w).
while e *do* w $\equiv_{def}$ *if* e *then* (w;(*while* e *do* w)) *else empty*
repeat w *until* e $\equiv_{def}$ w;(*while* (NOT e) *do* w).
loop w_1 *exit when* e *otherwise* w_2 $\equiv_{def}$
$\equiv_{def}$ w_1;(*while* (NOT e) *do* (w_2;w_1)) .
for c *times do* w $\equiv_{def}$ *if* c ≤ 0 *then empty*
else (w ; *for* c-1 *times do* w) .
donde c= M_σ[e].
for e_1 ≤ u < e_2 *do* w $\equiv_{def}$
$\equiv_{def}$ *if* $c_1 \geq c_2$ *then empty*
else ((*exist* u:(u=c_1 & w));(*for* c_1+1 ≤ u < c_2 *do* w))
donde c_1= $M_\sigma[e_1]$ y c_2= $M_\sigma[e_2]$.
for u < e *do* w $\equiv_{def}$
$\equiv_{def}$ *if* $c_1 \geq c_2$ *then empty*
else ((*exist* u:(u=c_1 & w)) ;
(*for* c_1+1 ≤ u < c_2 *do* w)) .
donde c_1= 0 y c_2= M_σ[e].

for u *in* c *do* w $\equiv_{def}$
  $\equiv_{def}$ *if* c={} *then empty*
    else ((*exist* u:(u=Car(c) & w)) ;
      (*for* u *in* Cdr(c) *do* w)) .
  donde c= $M_\sigma[e]$.
all $e_1 \leq u < e_2$: w $\equiv_{def}$
  $\equiv_{def}$ *if* $c_2-c_1 \leq 0$ *then true*
    else ((*exist* u: $u=c_1$ & w) & (*all* $c_1+1 \leq u < c_2$: w))
  donde $c_1 = M_\sigma[e]$, $c_2=M_\sigma[e_2]$.
all u < e: w $\equiv_{def}$ *all* $0 \leq u < e$: w

A.3 Semántica denotacional

Esta sección contiene la semántica de Tempura al estilo de la semántica denotacional[1]. Esta semántica es un complemento de la semántica lógica, definida anteriormente, en el aspecto de implementación. En cierta forma, entre la semántica lógica y la denotacional existe una relación semejante a la que hay entre la sintaxis abstracta y la sintaxis concreta de los lenguajes de programación:

- La sintaxis abstracta permite entender la *forma* de las instrucciones de un lenguaje. La sintaxis concreta permite realizar un analizador sintáctico para el lenguaje.
- La semántica lógica de Tempura permite entender el *significado* de sus instrucciones. La semántica denotacional permite realizar un intérprete del lenguaje.

A continuación se presentan los dominios necesarios para definir las funciones semánticas de las expresiones simples y de las instrucciones.

---

[1] [Allison 86]

A.3.1 Dominios

Dominios sintácticos.

Dominio	Definición	Descripción
ExpT	sintaxis expresiones	expresiones de Tempura
Atom	sintaxis átomos	átomos
List	sintaxis listas	listas
ConG	sintaxis const. globales	constantes globales
VarL	sintaxis var. locales	variables locales
VarG	sintaxis var. globales	variables globales
Var	sintaxis variables	variables
OpU	sintaxis op. unario	operadores unarios
OpB	sintaxis op. binario	operadores binarios
PAct	sintaxis parám. actuales	parámetros actuales
FormT	sintaxis fórmulas	fórmulas de Tempura
Loc	sintaxis localidades	localidades
PFor	sintaxis parám. formales	parámetros formales
LLoc	sintaxis lista localidades	listas de localidades

Dominios semánticos.

Dominio	Definición	Descripción
N		números naturales
Q		números racionales
B		valores de verdad
C		caracteres imprimibles
S	$= C^*$	cadenas de caracteres
T	Q,B,S,T,L : T^2	tipos
A	= Q+B+S+T	átomos
Nil	nil : Nil	nil
L	= Nil + (A+L) x L	listas
Bv	= A + L	valores básicos
DoneVar	@done : DoneVar	trabajo terminado
StateVar	@state : StateVar	número de estado
WNext	$\ominus$: WNext	weak next
FreeMem	@free : FreeMem	memoria libre
MemInx	= N	índices de memoria
IList	=MemInx x MemInx	listas internas
IPred	=PFor x Env x Cmd	predicados internos
IFunc	=PFor x Env x ExpT	funciones internas
PCall	=Env x Cmd	llamada a predicado
FCall	=Env x ExpT	llamada a función
IChop	=Cmd x Cmd x MemInx	chops internos
IProc	=Cmd x MemInx	procesos internos
ICmd	=Ipred+IFunc+PCall+ FCall+IChop+IProc	comandos internos
WCmd	=WNext x Cmd	comandos débiles

---

[2] T es un dominio finito cuyos elementos son Q,B,S,T,L .

Dominio	Definición	Descripción
Input	= ExpT*	entrada de datos
Ouput	= Bv*	salida de datos
Sv	=A + Nil + IList + IPred + IFunc + FreeMem	valores almacenables
Memory	=MemInx ⟶ Sv	memoria
EObj	=Var + ConG + DoneVar+StateVar	objetos de ambiente
Env	=EObj ⟶ MemInx	ambientes
Cmd	=FormT + ICmd + WCmd	comandos
State	=CmdxMemoryxInputxOutput	estados

Observación:

Consideramos que los dominios se construyen a partir de latices planas y con el símbolo ⊥ representamos al *elemento indefinido* de cada dominio.

En lo que sigue usaremos los símbolos T y F para abreviar los valores y las fórmulas *true* y *false* respectivamente, es decir, T,F : B y T,F : FormT. También usaremos ! para abreviar *nil* cuando representa *error*, es decir, !:Nil.

A.3.2 Semántica de expresiones

El significado de una expresión simple, en un ambiente dado, está determinado por el *valor* del índice de memoria que le corresponde en dicho ambiente. Esto se precisa con las siguientes definiciones .En este caso, las definiciones auxiliares aparecen después de la definición principal:

(Usamos las siguientes variables:
e, e_1, e_2, e_3: ExpT, ε: Env, μ: Memory, v: Var, u: VarG, k: ConG,
o_1: OpU, o_2: OpB, ξ: PFor, α: PAct)

Significado de Expresiones

E: ExpT x Env x Memory ⟶ Bv

$\mathcal{E}\langle e,\varepsilon,\mu\rangle$ = μ j si μ j : A + Nil
= list (μ j) si μ j : IList
= $\perp$ si no.

donde
j = $\mathcal{I}\langle e,\varepsilon,\mu\rangle$
list : IList ⟶ Bv
list⟨h,t⟩ = cons ⟨μ h, list(μ t)⟩ si μ h: A + Nil
= cons ⟨list(μ h), list(μ t)⟩ si μ h: IList
= $\perp$ si no

Indice de memoria para expresiones

$\mathcal{I}$: ExpT x Env x Memory ⟶ MemInx
$\mathcal{I}\langle$empty,$\varepsilon,\mu\rangle$ = ε ⊡done
$\mathcal{I}\langle$more,$\varepsilon,\mu\rangle$ = ε (NOT ($\mathcal{E}\langle$empty,$\varepsilon,\mu\rangle$))
$\mathcal{I}\langle k,\varepsilon,\mu\rangle$ = ε k
$\mathcal{I}\langle v,\varepsilon,\mu\rangle$ = ε v
$\mathcal{I}\langle u\ \alpha,\varepsilon,\mu\rangle$ = $\mathcal{I}\langle e,\delta',\mu\rangle$
donde
δ' = δ[(1 ⟨α,ε,μ⟩) / ξ]
⟨ξ,δ,e⟩ = μ (ε u)
$\mathcal{I}\langle o_1\ e,\varepsilon,\mu\rangle$ = ε ((O_1 o_1) ($\mathcal{E}\langle e,\varepsilon,\mu\rangle$))
$\mathcal{I}\langle o_2\ e_1\ e_2,\ \varepsilon,\mu\rangle$ = ε ((O_2 o_2) < $\mathcal{E}\langle e_1,\varepsilon,\mu\rangle$, $\mathcal{E}\langle e_2,\varepsilon,\mu\rangle$ >)
$\mathcal{I}\langle e_1[e_2],\varepsilon,\mu\rangle$ = elem < $\mathcal{E}\langle e_2,\varepsilon,\mu\rangle$, $\mathcal{I}\langle e_1,\varepsilon,\mu\rangle$, μ>
donde
elem : N x MemInx x Memory ⟶ MemInx

elem $\langle n, j, \mu \rangle$ = if μ j = $\langle h, t \rangle \wedge n \geq 0$ then
(if n=0 then h else elem $\langle n-1, t, \mu \rangle$)
else $\perp$

I $\langle e_1[e_2:e_3], \varepsilon, \mu \rangle$ =
= if $(c_2 \geq 0) \wedge (c_3 \geq 0)$ then
if $(c_3 - c_2 \leq 0)$ then ε nil
else subl $\langle I\langle e_1, \varepsilon, \mu \rangle, c_2, \mu \rangle$
else $\perp$.

donde

c_2 = E $\langle e_2, \varepsilon, \mu \rangle$

c_3 = E $\langle e_3, \varepsilon, \mu \rangle$

subl : MemInx x N x Memory $\longrightarrow$ MemInx

subl $\langle j, n, \mu \rangle$ = if μ j = $\langle h, t \rangle$ then
(if n=0 then j else subl $\langle t, n-1, \mu \rangle$)
else $\perp$

I $\langle \{e_1 : e_2 \leq u < e_3\}, \varepsilon, \mu \rangle$ =
= if $(c_1 \neq \perp \wedge c_2 \neq \perp \wedge c_3 \neq \perp)$ then
if $c_3 - c_2 \leq 0$ then ε nil
else ε (Cons $\langle c_1,$ E $\langle \{e_1 : c_2+1 \leq u < c_3\}, \varepsilon, \mu \rangle \rangle$)
else $\perp$.

donde

c_1 = E $\langle e_1[c_2/u], \varepsilon, \mu \rangle$

c_2 = E $\langle e_2, \varepsilon, \mu \rangle$

c_3 = E $\langle e_3, \varepsilon, \mu \rangle$

Cons: Bv x L $\longrightarrow$ L

Cons $\langle x, y \rangle$ = $\langle x, y \rangle$

I $\langle$ *if* e_1 *then* e_2 *else* $e_3, \varepsilon, \mu \rangle$ =
= if E $\langle e_1, \varepsilon, \mu \rangle$=true then I $\langle e_2, \varepsilon, \mu \rangle$
else if E $\langle e_1, \varepsilon, \mu \rangle$=false then I $\langle e_3, \varepsilon, \mu \rangle$ else $\perp$

Operadores unarios y binarios

O1: OpU $\longrightarrow$ Bv $\longrightarrow$ Bv

O1 o_1 = o_1

O2: OpB $\longrightarrow$ Bv x Bv $\longrightarrow$ Bv

O2 o_2 = o_2

A.3.3 Semántica de instrucciones

El significado de una instrucción, dada una entrada, está determinado por la pareja (valor, salida) que produce el intérprete al aplicarlo a dicha instrucción en las condiciones iniciales. Esto se precisa con las siguientes definiciones. En este caso, las definiciones auxiliares aparecen después de la definición principal:

(Usamos las siguientes variables:
w, c: Cmd, μ: Memory, ε: Env, u: VarG, a: Atom, ξ: PFor, α: PAct, $e, e_1, e_2, \ldots, e_n$: ExpT, $i, i_1 i_2 \ldots i_m$: Input, $o, o_1 o_2 \ldots o_m$: Output, $v, v_1, v_2, \ldots, v_n$: Var, $l, l_1, l_2, \ldots, l_n$: Loc)

Significado de instrucciones

M: FormT $\longrightarrow$ Input $\longrightarrow$ Sv x Output

M w i = C $\langle w, \mu, i, o \rangle$ ε

donde o: Output, ε: Env y μ: Memory se definen como sigue:

La salida o es la cadena vacía:

o = <>

El ambiente ε es una numeración de EObj :

ε x $\neq$ ε z si x $\neq$ z

La memoria μ cumple lo siguiente:

μ (ε x) = x si x : Atom

μ (ε x) = nil si x = {}

μ (ε x) = < ε e_1, ε {$e_2, e_3, \ldots, e_n$} > si x = {$e_1, e_2, \ldots, e_n$}

μ (ε x) = 0 si x = @state

μ (ε x) = $\perp$ si x: DoneVar + Var(w)

μ (ε x) = @free si no.

Intérprete

G: State ⟶ Env ⟶ Sv x Output

```
G <c,μ,i,o> ε = if c= ⊙ c' then (
                    if μ(ε @done)=⊥ then <!, o> else
                    if μ(ε @done)=! then <!, o> else
                    if μ(ε @done)=T then <T, o> else
                    if μ(ε @done)=F then G(F <c',μ',i,o> ε)
                else G( F <c,μ,i,o> ε) .
```

donde

```
μ' z = μ z                  si ∃x((z = ∈ x) ∧ (x:VarG + ConG))
     = (μ (ε @state))+1     si z = ε @state
     = ⊥                    si no.
```

Transformación de estados

F: State ⟶ Env ⟶ State

```
F<empty,μ,i,o>ε = <⊙ T,μ',i,o>
     donde μ'= if μ (ε @done) ≠ ⊥ then μ [! /ε @done]
               else μ [T /ε @done]

F<more,μ,i,o>ε = <⊙ T,μ',i,o>
     donde μ'= if μ (ε @done) ≠ ⊥ then μ [! /ε @done]
               else μ [F /ε @done]

F<true,μ,i,o>ε      = <⊙ T,μ,i,o>

F<false,μ,i,o>ε     = <⊙ F,μ [! /ε @done],i,o>

F<quit,μ,i,o>ε      = <⊙ T,μ [quit /ε @done],i,o>

F<v=e,μ,i,o>ε = <w',μ',i,o>
     donde
     w' = if μ (I <e,ε,μ>) =⊥ then w
          else ⊙ T
     μ'=  if μ (I <e,ε,μ>) =⊥ then μ else
          if μ (ε v) = ⊥ then μ [ μ (I <e,ε,μ>) / ε v ] else
          μ [! /ε @done]
```

```
F<O 1= e,μ,i,o>ε = <w',μ,i,o>
     donde
     w' = if μ (I <e,ε,μ>) =⊥ then O 1= e
          else (if more then ⊖ (1 = μ (I <e,ε,μ>) ) else false)

F<r∧s,μ,i,o>ε = <w,μ'',i'',o''>
     donde
     F<r,μ,i,o>ε = <r',μ',i',o'>
     F<s,μ',i',o' ε> = <s',μ'',i'',o''>
     w =  if r'= ⊖ r'' ∧ s'= ⊖ s'' then
               (if r''=T ∧ s'' = T then ⊖ T else
               if r'' = T then s' else
               if s'' = T then r' else
               ⊖ (r'' ∧ s'') )
          else r' ∧ s'

F<snext w,μ,i,o>ε   = F<if more then ⊖ w,μ,i,o>ε

F<⊖ w,μ,i,o>ε       = <⊖ w,μ,i,o>

F<always w,μ,i,o>ε  = F<w & ⊖ always w,μ,i,o>

F<w1;w2,μ,i,o>ε     = F<<w1,w2,j>,μ',i,o>
     donde
     j es el menor indice nuevo de μ, j= Min {x/ μ x = @free}
     μ'= μ[⊥ /j]

F<<w1,w2,j>,μ,i,o>ε = F<w,μ',ī',o'>ε
     donde
     ε'= ε[j / @done]
     F<w1,μ,i,o>ε' = <w1',μ',i',o'>
     w=   if w1'= ⊖ w1'' then
               if μ(ε' @done)= T then w2 else (snext <w1'',w2,j>)
          else <w1',w2,j>

F<if e then w1 else w2,μ,i,o>ε = F<w,μ,i,o>ε
     donde
```

w= if $\mu(I\langle e,\varepsilon,\mu\rangle)=\perp$ then (*if* e *then* w_1 *else* w_2)
else if $\mu(I\langle e,\varepsilon,\mu\rangle)=T$ then w_1 else w_2

F<*exist* $v_1, v_2, \ldots, v_n$:w, μ, i, o>ε = <<ε', w>, μ', i, o>
donde
ε' x = ε x si x ≠ $v_1, v_2, \ldots, v_n$
ε' v_i = J_i si no
las J_i son los menores indices nuevos de μ, μ J_i = @free
μ' = μ[$\perp$/ J_i]

F<predicate u ξ :w, μ, i, o>ε = <$\ominus$ T, μ', i, o>
donde μ'= μ [<ξ, ε, w>/ε u]

F<function u ξ :e, μ, i, o>ε = <$\ominus$ T, μ', i, o>
donde μ'= μ [<ξ, ε, e>/ε u]

F<u α, μ, i, o>ε = <<δ', w>, μ, i, o>
donde
μ (ε u) = <ξ, δ, w>
δ'= δ[(I <α, ε, μ>)/ ξ]

F<<δ, w>, μ, i, o>ε = <w', μ', i', o'>
donde F<w, μ, i, o>δ = <γ, μ', i', o'>
w'= if γ= $\ominus$ γ' then $\ominus$ <δ, γ'> else <δ, γ>

F<*request* ($l_1, l_2, \ldots, l_n$), μ, i $i_1 i_2 \ldots i_m$, o>ε = F<w, μ', i', o>ε
donde

w	= F	si n > m
	= (l_1=i_1 ∧ l_2=i_2 ∧ ... ∧ l_n=i_n)	si no.
μ'	= μ[! / c @done]	si n > m
	= μ	si no.
i'	= i	si n > m
	= $i_{n+1} i_{n+2} \ldots i_m$.	si no.

F<*display* ($e_1, e_2, \ldots, e_n$), μ, i, $o_1 o_2 \ldots o_m$>ε = F<w, μ, i, o'>ε
donde

$$
\begin{array}{lll}
w & = \ominus T & \text{si } o_{m+i} \neq \perp,\ 1 \le i \le n \\
 & = \mathit{display}\ (e_1, e_2, \ldots, e_n) & \text{si no.} \\
o' & = o_1 o_2 \cdots o_m o_{m+1} o_{m+2} \cdots o_{m+n} & \text{si } o_{m+i} \neq \perp,\ 1 \le i \le n \\
 & = o & \text{si no.} \\
o_{m+i} & = \mu\ (I\ \langle e_i, c, \mu \rangle),\ 1 \le i \le n &
\end{array}
$$

$F\langle \mathit{len}\ (e), \mu, i, o \rangle c = F\langle w, \mu, i, o \rangle c$

donde

$a = \mu\ (I\ \langle e, c, \mu \rangle)$

$$
\begin{array}{lll}
w & = \mathit{len}\ (e) & \text{si } a = \perp \\
 & = \mathit{if}\ a{=}0\ \mathit{then}\ \mathit{empty}\ \mathit{else}\ \mathit{len}(a{-}1) & \text{si no.}
\end{array}
$$

$F\langle \mathit{list}\ (v, e), \mu, i, o \rangle c = \langle w, \mu', i, o \rangle$

donde

$$
\begin{array}{lll}
c & = \mu(I\ \langle e, c, \mu \rangle) & \\
w & = \mathit{list}\ (v, e) & \text{si } c = \perp \\
 & = \ominus T & \text{si no} \\
\mu' & = \mu & \text{si } c = \perp \\
 & = \mu[\langle h, t \rangle\ /\ c\ v] & \text{si no}
\end{array}
$$

$\langle h, t \rangle$ es la lista $\{\perp, \perp, \ldots, \perp\}$ de longitud c.

$F\langle \mathit{fixed\_list}\ (v, e), \mu, i, o \rangle c = \langle w, \mu', i, o \rangle$

donde

$$
\begin{array}{lll}
c & = \mu(I\ \langle e, c, \mu \rangle) & \\
w & = \mathit{fixed\_list}\ (v, e) & \text{si } c = \perp \\
 & = \ominus T & \text{si no} \\
\mu' & = \mu & \text{si } c = \perp \\
 & = \mu[\langle h, t \rangle\ /\ c\ v] & \text{si no}
\end{array}
$$

$\langle h, t \rangle$ es la lista $\{\perp, \perp, \ldots, \perp\}$ de longitud c.

A la memoria de v_c se le asigna el atributo *fija* para que se conserve de estado en estado[3].

$F\langle \mathit{stable\_struct}\ v, \mu, i, o \rangle c = \langle w, \mu, i, o \rangle$

---

[3]Agregando un parámetro del tipo $f: \mathbf{MemInx} \longrightarrow \mathbf{B}$ se puede representar la memoria *fija*.

donde

j = I <v,e,μ>
w = *stable_struct* v si j=⊥
= ⊖ T si no

A la memoria de v, e v, se le asigna el atributo *fija*.

F<*process* w,μ,i,o>e = F<<w,j>,μ',i,o'>e
j es el menor índice nuevo de μ, j= Min {x/ μ x = @free}
μ'= μ[⊥ /j]

F<<w,j>,μ,i,o>e = F<w',μ',i',o'>e
donde
e'= e[j / @done]
F<w,μ,i,o>e' = <w'',μ',i',o'>
w'= if w''= ⊜ w''' then
(*if empty* = μ(e' j) *then* ⊜ <w''',j> *else false*)
else <w'',j>

F<exec e,μ,i,o>e = <w,μ,i,o'>
donde
m= μ(I <e,e,μ>)
w = exec e si m=⊥
= ⊖ T si no.
o' = o si m=⊥
= e'' si no.
<x,o''>= G <e',μ,i',o>e,
e' es la fórmula contenida el archivo m.
i' es la entrada de datos contenida en el archivo m.

F<exec e_1 e_2, μ,i,o>e = <w,μ,i,o>
donde
m= μ(I <e_1,e,μ>), n= μ(I <e_2,e,μ>)
w = exec e_1 e_2 si m=⊥ ∨ n=⊥
= ⊖ T si no.
Se interpreta la fórmula contenida en el archivo m, la entrada se toma del archivo m y la salida se deposita en el archivo n.

Instrucciones derivadas:

F<*skip*, μ, i, o>ε = F<*snext empty*, μ, i, o>ε

F<*if* e *then* w, μ, i, o>ε = F<*if* e *then* w *else true*, μ, i, o>ε

F<l *gets* e, μ, i, o>ε = F<w, μ, i, o>ε
donde w = *always* (*if more then* ((*nextv* l)=e))

F<l *tequ* e, μ, i, o>ε = F<*always*(l=e), μ, i, o>ε

F<l *tass* e, μ, i, o>ε = F<*exist* u:((u=e) & *fin*(l=u)), μ, i, o>ε

F<*stable* V, μ, i, o>ε = F<V *gets* V, μ, i, o>ε

F<*halt* e, μ, i, o>ε = F<*if* e *then empty else more*, μ, i, o>ε

F<*fin* w, μ, i, o>ε = F<□ (*if empty then* w), μ, i, o>ε

F<*while* e *do* w, μ, i, o>ε = F<w', μ, i, o>ε
donde w'= *if* e *then* (w; *while* e *do* w) *else empty* .

F<*repeat* w *until* e, μ, i, o>ε = F<w;*while* e *do* w, μ, i, o>ε

F<*loop* w_1 *exit when* e *otherwise* w_2, μ, i, o>ε = F<w, μ, i, o>ε
donde w= w_1;(*while* (NOT e) *do* (w_2;w_1)) .

F<*for* e *times do* w, μ, i, o>ε = F<w', μ, i, o>ε
donde

c	= μ(I <e, ε, μ>)	
w'	= *for* e *times do* w	si c=$\perp$
	= *if* c≤0 *then empty*	
	else (w; *for* c-1 *times do* w)	si no

$F\langle for\ u \langle e\ do\ w, \mu, i, o\rangle c = F\langle w', \mu, i, o\rangle c$

donde

$c_1 = 0$

$c_2 = \mu(I\ \langle e, c, \mu\rangle)$

$w' = for\ u \langle e\ do\ w$ si $c_2 = \perp$

$= if\ c_1 \geq c_2\ then\ empty$

$else\ (\ (\exists u: (u = c_1 \wedge w));$

$(for\ c_1 + 1 \leq u < c_2\ do\ w)\ )$ si no.

$F\langle for\ u\ in\ e\ do\ w, \mu, i, o\rangle c = F\langle w', \mu, i, o\rangle c$

donde

$c = \mu(I\ \langle e, c, \mu\rangle)$

$w' = for\ u\ in\ e\ do\ w$ si $c = \perp$

$= if\ c = \{\}\ then\ empty$

$else\ (\ (\exists u: (u = Car(c) \wedge w));$

$(for\ u\ in\ Cdr(c)\ do\ w)\ )$ si no

$F\langle all\ u \langle e\ :\ w, \mu, i, o\rangle c = F\langle w', \mu, i, o\rangle c$

donde

$c = \mu(I\ \langle e, c, \mu\rangle)$

$w' = all\ u \langle e\ :\ w$ si $c = \perp$

$= if\ c \leq 0\ then\ true$

$else\ (\ (exist\ u:\ u = c\ \&\ w)\ \&$

$(all\ u \langle c - 1\ :\ w)\ )$ si no.

BIBLIOGRAFIA

[Abadi 87]
Abadi M. *Temporal-logic theorem proving*, PhD thesis, rep. STAN-CS-87-1151, Standford University, 1987.

[Abadi-Manna 85]
Abadi M., Manna Z., *Nonclausal Temporal deduction*. En [Parikh 85]. p.1-15.

[Allison 86]
Allison L. *A practical introduction to denotational semantics*. Cambridge University Press 1986.

[Benthem 83]
Benthem J. *The logic of time*. Reidel 1983.

[Boolos-Jeffrey 74]
Boolos G., Jeffrey R. *Computability and Logic*. Cambridge University Press, 1974.

[Boyer-Moore 81]
Boyer R., Moore J. *The Correctness Problem in Computer Science*. Academic Press, Nueva York 1981.

[Bridge 77]
Bridge J. *Beginning Model Theory*. Oxford Univeristy Press, Oxford 1977.

[Clarke-Kozen 83]
Clarke E., Kozen D. *Logic of Programs*. Lecture Notes in Computer Science 164, Springer-Verlag 1983.

[Diller 88]
Diller A. *Compiling Functional Languages*. John Wiley & Sons Ltd, Gran Bretaña 1988.

[Ebbinghaus-Flum-Thomas 84]
Ebbinghaus H., Flum J., Thomas W. *Matehmatical Logic*. Springer-Verlag, Nueva York 1984.

[Enderton 72]
Enderton H. *A Mathematical Introduction to Logic*. Academic Press, Nueva York 1972.

[Halpern-Manna-Moszkowski 83]
Halpern J., Manna Z., Moszkowski B. *A hardware semantics based on temporal intervals.* Lecture Notes in Computer Science No. 154, Springer-Verlag, Berlin 1983, p. 278-291.

[Hoare-Shepherdson 85]
Hoare C., Shepherdson J. *Mathematical Logic and Programming Languages.* Prentice-Hall 1985.

[Hughes-Cresswell 73]
Hughes G., Cresswell M. *Introducción a la Lógica Modal.* Tecnos, Madrid 1973.

[Kraus-Lehmann 88]
Kraus S., Lehmann D. *Knowledge, Belief and Time.* Theoretical Computer Science 58 p.155-174.North-Holland 1988.

[Kröger 84]
Kröger F., *A generalized nextime operator in temporal logic.* J. Comput. System Scie. 29 (1984).

[Manna-Pnueli 81]
Manna Z., Pnueli A. *Verification of concurrent programs: The temporal framework.* En [Boyer-Moore 81] p. 215-273.

[Moszkowski 83]
Moszkowski B. *Reasoning about digital circuits.* PhD Thesis, rep. STAN-CS-83-970, Standford University 1983.

[Moszkowski 85]
Moszkowski B. *A temporal logic for multilevel reasoning about hardware.* Computer 18,2 (Febrero 1985), p. 10-19.

[Moszkowski 86]
Moszkowski B. *Executing Temporal Logic Programs.* Cambridge University Press 1986.

[Moszkowski-Manna 83]
Moszkowski B.,Manna Z. *Reasoning in interval temporal logic.* En [Clarke-Kozen 83] p. 371-381.

[Odifredi 90]
Odifredi P. *Logic and Computer Science,* Academic Press 1990.

[Parikh 85]
Parikh R. *Logics of programs.* Lecture Notes in Computer Science No. 193. Springer-Verlag 1985.

[Rescher-Urquhart 71]
Rescher N., Urquhart A. *Temporal Logic.* Springer-Verlag, Nueva York 1971.

[Schmidt 86]
Schmidt D. *Denotational Semantics.* Allyn and Bacon, USA 1986.

[Smets 88]
Smets P. (Ed.) *Non-Standard logics for automated reasoning,* Academic Press, Londres 1988.

[Szalas 86]
Szalas A. *Concerning the semantic consequence relation in first-order temporal logic.* Theoretical Computer Science 47, p.329-334, North-Holland 1986.

[Szalas-Holenderski 88]
Szalas A., Holenderski L. *Incompletness of first-order temporal logic with until.* Theoretical Computer Science 57, p.317-325, North-Holland 1988.

[Thayse 89]
Thayse A. *From Modal Logic to Deductive Databases.* John Wiley & Sons Ltd, Gran Bretaña 1989.

[Turner 84]
Turner R. *Logics for Artificiall Intelligence.* E. Horwood, 1984.

[Wolper 83]
Wolper P. *Temporal Logic can be more expresive.* Information and Computation, Vol. 56, 1983, No. 1,2.