BENEMÉRITA UNIVERSIDAD AUTÓNOMA DE PUEBLA

FACULTAD DE CIENCIAS DE LA COMPUTACIÓN

INTÉRPRETE DE ML Y DEMOSTRACIÓN AUTOMÁTICA DE TEOREMAS EN LÓGICA DE ALTO ORDEN

TESIS

QUE PARA OBTENER EL TÍTULO DE

LICENCIADO EN COMPUTACIÓN

PRESENTA

MANCILLA SÁNCHEZ JOSÉ ISAURO

ASESOR

LIC. LAVALLE MARTÍNEZ JOSÉ DE JESÚS

Puebla, Pue. Marzo 1999

Contenido

Introducción

La especificación formal es una representación abstracta de la realidad y se recomienda en sistemas críticos.

Los métodos formales ofrecen ventajas considerables sobre los métodos tradicionales cuando los sistemas que se desarrollan requieren la más alta integridad; esto es, en sistemas de los que dependen vidas humanas, grandes cantidades de dinero, etc.

El uso de métodos formales incrementa la confianza en la integridad de un sistema y en que se comportará verdaderamente como se espera; aunque hay que recalcar que los métodos formales no garantizan que el sistema esté libre de errores, ya que intervienen muchos factores, entre ellos el humano [BoJ+90].

Uno de los propósitos de la especificación formal es la expresividad, y esta meta se alcanza mejor si se escoge un formalismo tan cercano como sea posible a la representación convencional matemática. Esto desalienta el uso de lógicas no-clásicas y sugiere una notación basada en el cálculo de predicados de primer orden.

Pero no sólo es importante para un lenguaje de especificación el proporcionar una lógica rica y poderosa, también se considera importante el que esa lógica tenga notación atractiva y familiar, y más aún, que tenga tipos.

Para especificación, un sistema con tipos es generalmente preferible a un sistema sin tipos. Hay dos razones para esto:

1. En computación es natural y conveniente distinguir los diferentes tipos de las entidades que son manipuladas: booleanos de enteros, listas de funciones y así por ejemplo. Para reconocer estas diferencias en un lenguaje de especificación se requieren especificaciones claras y que puedan detectar muchos errores simples usando verificación estricta de tipos.

2. Un probador de teoremas eficiente requiere que ciertos tipos de entidades (por ejemplo booleanos y enteros) sean reconocidos y tratados especialmente; claramente, esto es más natural cuando el lenguaje de especificación soporta las distinciones necesarias de tipo.

En computación como en matemáticas, en muchos problemas de especificación es conveniente el utilizar un sistema formal más poderoso que el cálculo de predicados de primer orden. Una de las elecciones más delicadas en el diseño de un lenguaje de especificación aparece cuando el sistema formal se extiende más allá del cálculo de predicados de primer orden. Usualmente, una mejor elección de diseño debe ser negociada entre la conveniencia de expresividad del lenguaje de especificación o la automatización y efectividad del soporte mecánico que puedan ser proporcionados.

En esta elección, los extremos son ilustrados por la notación Z y la lógica computacional de Boyer y Moore. Z es una notación de especificación muy rica basada en teoría de conjuntos y ha sido usada para diseñar muchos sistemas, pero sólo tiene disponible un soporte mecánico elemental. El sistema Boyer-Moore por otro lado, utiliza una lógica muy limitada como su lenguaje de especificación, pero tiene un poderoso probador de teoremas que ha sido usado para comprender varias especificaciones muy grandes y completas. Ninguna de estas opciones es la ideal, por lo que se necesita una manera intermedia que evite la necesidad de escoger entre expresividad y/o soporte mecánico [RuJ91].

La Lógica de Alto Orden es una buena opción para seleccionarla como lenguaje de especificación ya que:

1. Maneja tipos y la verificación estricta de tipos es natural.
2. Su lógica es muy expresiva y poderosa, ya que puede cuantificar variables, predicados y funciones [HolT91].
3. Se puede tomar como base para crear un probador de teoremas confiable (como se verá a lo largo de este trabajo).
4. Las funciones que maneja son totales, lo cual es una ventaja, ya que las funciones parciales hacen más complicada la verificación estricta de tipos y el desarrollo de probadores de teoremas [RuJ91].

A lo largo de este trabajo se utilizarán las siglas HOL (Higher Order Logic) para denotar a la Lógica de Alto Orden [HolT91].

En esta tesis se desarrolla un Probador de Teoremas para HOL, el cual soporta pruebas hacia adelante y hacia atrás. El estilo de pruebas hacia adelante aplica reglas de inferencia a teoremas existentes para obtener nuevos teoremas, y eventualmente, obtener el teorema deseado. Las pruebas hacia atrás u orientadas a metas, empiezan con la meta a ser probada, y después se aplican tácticas para la meta y submetas hasta que la meta es descompuesta en teoremas más simples ya existentes [HolD91].

Para poder implantar un probador de teoremas para HOL, se necesita un metalenguaje que sirva como medio ambiente en el cual se puedan construir herramientas para asistir en la construcción de pruebas.

Se utilizará el lenguaje funcional ML, el cual fue desarrollado justamente como un metalenguaje para la implantación de probadores de teoremas, y ha sido usado para este propósito en sistemas probadores de teoremas importantes como Edimburgh LCF, Nuprl e Isabelle [HolD91].

ML contiene muchas características útiles para el desarrollo de probadores de teoremas.

1. ML está basado en el cálculo lamba, y los términos de la lógica de alto orden son justamente expresiones lambda.

2. Es de alto orden, lo que permite que operaciones complejas sean compuestas fácilmente.

3. Contiene la capacidad de empatar patrones, lo cual permite una manipulación flexible de objetos.

Para desarrollar el probador de teoremas, primero implantamos un intérprete de ML y después lo extendemos para que pueda manejar los símbolos y construcciones utilizados en HOL.

Una vez implantado el ML extendido (lo llamamos ML-HOL), podemos programar en él los axiomas; las teorías básicas[1]; funciones para cargar, extender y salvar teorías; reglas de inferencia; reglas de inferencia derivadas; tácticas, estrategias[2] y todas las funciones necesarias para demostrar teoremas.

En el intérprete se implantó un sistema de inferencia de tipos y un comprobador de tipos. Un sistema de inferencia de tipos determina el tipo de una construcción del lenguaje a partir del modo en que se usa. El término "inferencia de tipos" se aplica a menudo al problema de determinar el tipo de una función a partir de su cuerpo. El comprobador de tipos sirve para verificar que las funciones sean llamadas con los parámetros del tipo correcto [AhA+90]. Esto es muy importante ya que se puede asegurar que todo valor de los tipos que representan teoremas sean verdaderamente teoremas y de esta forma, no puede haber inconsistencias al construir teorías.

Este trabajo está estructurado de la siguiente manera: las características de ML que se implantaron se muestran en el capítulo 1.

Los fundamentos teóricos de HOL se explican en el capítulo 2. En ese capítulo se estudian los conceptos del cálculo lambda; la lógica de alto orden; que es una teoría de HOL y como extender las teorías.

La implantación del intérprete se explica en el capítulo 3. Se muestran las principales estructuras de datos que se utilizaron; los métodos que se emplearon para implantar el sistema de inferencia de tipos, el comprobador de tipos y el intérprete. También se explica el algoritmo del recolector de basura que se utiliza.

En el capítulo 4 se muestra la representación concreta de la Lógica de Alto Orden. Se analiza la representación de términos, teoremas y teorías. Se explica el proceso de pruebas hacia adelante, las reglas de inferencia derivadas, el proceso de pruebas orientadas a metas, las tácticas y estrategias. Finalmente en el capítulo 5 se estudian dos aplicaciones.

Cuando el sistema ML-HOL se inicia, un número de funciones son predefinidas. Este trabajo contiene un apéndice en el que se describen las funciones predefinidas más importantes, entre las que se incluyen funciones de propósito general, funciones que comprenden la representación de términos lógicos, de

[1] Las teorías son la base de HOL.

[2] En inglés son Tactic y Tactical. Son la base para el proceso de demostración hacia atrás.

teoremas y de teorías. También se muestran las funciones que implantan las reglas de inferencia primitivas y derivadas; las tácticas y estrategias.

Capítulo 1

El metalenguaje ML

ML es un lenguaje de programación de propósito general. Fue implantado por Milner, Morris y Wadsworth en la Universidad de Edinburgh al inicio de la década de los 70's [HolD91].

ML es un lenguaje interactivo y de alto nivel con el que se pueden hacer declaraciones y evaluar expresiones.

Hay muchas variantes de ML, pero decidimos trabajar con el Estandár ML (SML) debido a que su sintaxis es más clara. En este capítulo introduciremos las características de SML que se implantaron.

En los ejemplos, las declaraciones y expresiones se escriben en **negritas**; y si es necesario, inmediatamente después se escribe en *cursivas* lo que regresa SML.

1.1 Valores simples y operadores

La forma más simple de una declaración asocia un nombre con un valor básico, por ejemplo un entero:

val n = 2;
val n = 2 : int

lo que nos dice que el tipo de n es int.

Una forma particular de declaración se escribe como una expresión:

2+2;
val it = 4 : int

Una expresión como *2+2* es la declaración implícita de la variable *it* propia de SML. *it* guarda el valor de la última expresión evaluada.

Los operadores aritméticos binarios que maneja ML son:

*+, −, *, /, MOD*

y corresponden a las operaciones matemáticas de suma, resta, multiplicación, división y módulo, respectivamente. Existe un operador unario "menos", como en la expresión *–3*, pero no hay un operador unario "más". La división devuelve el valor entero que obtuvo y desecha cualquier fracción restante. Por ejemplo, *1/2* tiene el valor *0* y *7/3* tiene valor *2*. El resultado de *a MOD b* es el residuo de la división de *a* entre *b*. Por ejemplo, *5 MOD 2* tiene valor *1* y *6 MOD 3* tiene el valor *0*.

ML utiliza un operador de igualdad = para determinar si dos expresiones son iguales. ML también utiliza operadores de relación:

$<$ *: menor que*
$>$ *: mayor que*
$<=$ *: menor o igual que*
$>=$ *: mayor o igual que*

ML maneja dos operadores lógicos, *&* y *OR*, que corresponden a las operaciones lógicas de conjunción y disyunción, respectivamente. Todos los operadores de igualdad, de relación y lógicos son binarios. Cada uno de ellos toma dos expresiones como operandos y ejecuta la operación correspondiente.

1.2 Valores de función

Para definir una función f con parámetro formal x y cuerpo e se ejecuta la declaración

fun f(x) = e;

Una función en SML se aplica mencionándola junto con sus argumentos. De esta forma, $f(x)$ y $f\ x$ son equivalentes. La aplicación de funciones se asocia a la izquierda, por lo que $d\ x\ E$ corresponde a $(d\ x)\ E$; es decir, el resultado de aplicar d a x es una función que se aplica a E [SeR89].

La respuesta a la declaración de función

fun sucesor (n) = n+1;
val sucesor = fn : int –> int

puede leerse como "el valor de *sucesor* es una función de enteros a enteros", aunque no se haya expresado explícitamente que n tenga tipo int. Esto es porque en la expresión *n+1*, la variable n debe tener tipo entero ya que *1* es un entero y el operador interconstruido + se aplica a pares de enteros.

En SML, se utilizan variables de tipo. Una variable de tipo es una expresión que puede substituirse por cualquier tipo. Para identificar a una variable de tipo se utiliza un apóstrofe, por ejemplo a'.

Un constructor de tipo toma uno o más tipos y forma un nuevo tipo. Por ejemplo, *'*'* es un constructor de tipo, tiene dos argumentos y forma el tipo producto cartesiano. denotado en SML por (*a' * b'*) [HolD91].

En SML, se le llama patrón a los parámetros de una función [HolD91]. Un patrón p se define de la siguiente manera:

p	:	c	constante
	|	*var*	variable
	|	*p1,p2*	construcción de pares
	|	*p1::p2*	construcción de listas
	|	*[ ]*	lista vacía
	|	*[p1,p2,...,pn]*	lista de n elementos
	|	*(p)*	equivalente a p (los paréntesis sólo sirven para agrupar)

Los tipos pueden proporcionarse directamente mediante la notación

<patrón> : <tipo>

Los tipos explícitos [SeR89] son necesarios algunas veces para resolver la sobrecarga, como sucede en

fun suma (x, y) = x + y
Error : operador sobrecargado, no puede resolverse.

Cualquier sugerencia acerca del tipo de x o y es suficiente para que ML-HOL infiera el tipo de esta función:

fun suma (x:int, y) = x + y;
*val suma = fn : int * int −> int*

En donde se expresa que x es un entero por lo que y debe de ser entero y por lo tanto el resultado también debe de ser entero.

Como ya se mencionó, SML maneja expresiones lambda; aunque por razones tipográficas no se utiliza el símbolo λ. En ML-HOL, la expresión $\lambda x.\,\lambda y.\,x + y$ se escribe

fn x => fn y => x+y;

En SML, las funciones al igual que cualquier otro tipo de valor pueden ser tomadas como argumentos y ser devueltas como resultados de funciones. Así, la siguiente declaración tiene sentido:

fun doble f y = f(f(y)):
val doble = fn : (a' −> a') −> a' −> a'

La respuesta significa que *doble* se refiere a una función que toma como argumentos a una función unaria de cualquier tipo *a'* y un valor de tipo *a'*, y el resultado que regresa también es de tipo *a'*. Este ejemplo también introduce la idea de *funciones polimorfas*. Una función *polimorfa* puede aplicarse a argumentos de más de un tipo [ReS90].

1.3 Listas y funciones polimorfas

Las funciones polimorfas surgen de manera natural al manipular listas. Las listas se escriben entre corchetes '*[*' y '*]*', las comas separan los elementos de una lista. La lista vacía se escribe como *[ ]*.

Todos los elementos de una lista deben ser del mismo tipo. Si este es *a'*, entonces el tipo de la lista es *a' list*. Una lista de enteros tiene tipo *int list*, una lista de cadenas de carácteres (*strings*) tiene tipo *string list*, etc. [SeR89]. *list* es un constructor de tipo y tiene sólo un argumento.

Las listas se pueden formar usando el constructor :: (también conocido como "cons") que es una función de elementos y listas a listas. Así *1::(2::(3::[ ]))* es igual a *[1,2,3]*.

Otro operador que se puede utilizar con listas es @, el cual sirve para concatenar listas [ReS90]. Por ejemplo, *[1,2] @ [3,4]* es igual a *[1,2,3,4]*.

Un ejemplo de función polimorfa es

fun longitud [ ] = **0**
  | **longitud (a::y) = 1 + longitud (y);**
val longitud = fn : (a' list −> int)

La función *longitud* está definida por casos (mediante el símbolo |) que se verifican en secuencia. Si el argumento es *[ ]* (lista vacía) entonces la longitud es 0. De lo contrario, si el argumento tiene la forma *a::y* con cabeza *a* y cola *y* entonces su longitud es uno más que la longitud de *y*.

La función *longitud* se puede utilizar con cualquier instancia del tipo *a' list*.

1.4 Tipos de datos

Otro concepto en SML que es importante es la declaración de tipos de datos (*datatype*).

Se puede ver a una declaración de tipos de datos como la descripción de una manera de hacer un nuevo tipo mediante la mezcla de viejos tipos. La declaración puede ser recursiva. Por ejemplo, si un término puede ser una cadena o un término aplicado a un término podemos escribir

datatype termino = var of string | app of termino * termino;

y esto introduce dos nuevos valores (*constructores*), llamados *var* y *app*, los cuales toman una cadena y un par de términos, respectivamente, y regresan un término [ReS90]. Así, si tenemos la cadena *"abc"*, se puede convertir en un término usando el constructor *var*. Teniendo el término *var "abc"* podemos hacer otro término usando *app*, lo que nos da *app(var "abc",var "abc")*.

1.5 Otras formas de expresiones

Una de las expresiones más comunes es de la forma *if-then-else*. Se puede utilizar en definiciones como:

fun fact n = if n=0 then 1 else n*(fact(n-1));
val fact = fn : int −> int

Cualquier nombre de función debe ser definido antes de utilizarse; esto significa que las definiciones de funciones mutuamente recursivas no son permitidas. Así que

fun f x = if x=0 then 1 else g(x-1);
fun g x = if x=0 then 1 else f(x-1);

produce un mensaje de error desde el intérprete ya que *g* está indefinida en la definición de *f*.

Otra forma de expresión es proporcionada por la forma *let-in-end*. Ésta permite que las definiciones se hagan locálmente, lo que realza la modularidad de los programas. Por ejemplo, se puede escribir

fun f x y z = if x=2 * (y + z) then 3 * (y + z) else 0;

pero el mismo efecto se produciría si se escribe

```
fun f x y z = let val u = y + z in
                  if x=2 * u then 3 * u else 0
              end;
```

La ventaja de la segunda definición es que $y + z$ se calcula una sola vez, mientras que en la primera definición tiene que calcularse dos veces. Claramente, si hay muchos cálculos complicados repetidos, el uso de este mecanismo puede ahorrar una gran cantidad de cálculos, con lo que se incrementa la eficiencia [ReS90].

1.6 Manejo de excepciones en SML

Las excepciones son un mecanismo para manejar casos especiales o fallas que ocurren durante la ejecución de un programa. Por ejemplo, si se trata de evaluar *1 mod 0* entonces se obtendrá el mensaje

Failure: mod

lo que significa que la evaluación de la función *mod* para este par de argumentos ha fallado.

El mecanismo de excepciones en SML permite que las fallas se hagan explícitas. La forma más simple de una declaración de excepción consiste en la palabra clave *exception* seguida del nombre de una excepción [SeR89]:

exception grande;

También se pueden declarar varias excepciones al mismo tiempo utilizando la palabra clave *and* [ReS90]:

```
exception REFL
and       DISCH
and       MP;
```

Una excepción es activada (*raised*) escribiendo

raise <nombre de excepción>

Por ejemplo, se puede declarar la función *prueba*, la cual recibe un número como parámetro; si el número es mayor que 10000, activa la excepción *grande*, de lo contrario divide el número entre 10.

fun prueba n = if n > 10000 then raise grande else n/5;

lo que da resultados tales como:

prueba 50;
val it = 5 : int

prueba 15000;
failure: grande

Capítulo 2

Fundamentos teóricos de HOL

La Lógica de Alto Orden es una generalización del cálculo de predicados con tres extensiones principales [HolT91]:

1. Los cuantificadores se pueden aplicar a variables de predicado y función, a diferencia del Cálculo de Predicados de Primer Orden en el que los cuantificadores se aplican exclusivamente a variables.

   Por ejemplo, para formalizar una afirmación

   Existe al menos una función tal que $\varphi(a) = b$

   se define el predicado P.

   $P(x, y) \rightleftharpoons x = y$

   donde $\rightleftharpoons$ indica que cuando se encuentre una expresión de la forma *P(x,y)* se debe sustituir por $x = y$. La fórmula que representa la estructura de la frase es:

   $\exists \varphi \; P(\varphi(a), b)$

   En otro ejemplo, para analizar la afirmación

   Existen algunas relaciones P que son simétricas

   la fórmula que representa la estructura de la frase es:

   $\exists P \; \forall x \, \forall y \; (P(x, y) \rightarrow P(y, x))$

   Esto está fuera del alcance del cálculo de predicados de primer orden, ya que en el primer ejemplo se está cuantificando a una variable de función y en el segundo a una variable de predicado; por lo que para expresar las fórmulas se tiene que hacer uso de una lógica de más alto orden [CuJ86].

2. La lógica es con tipos.

   De hecho, la Lógica de Alto Orden tiene su origen en la Teoría Simple de Tipos, formulada por Alonzo Church.

Como se mencionó en la Introducción, es una ventaja que un lenguaje de especificación tenga tipos.

3. No se hace distinción entre términos y fórmulas.

   En lógicas tales como cálculo de predicados de primer orden, lógica modal o lógica temporal, se tienen términos, pero en adición a éstos se tienen conectivos lógicos y fórmulas, las cuales son distintas de los términos. Sin embargo, en la Lógica de Alto Orden, no es necesario esta distinción. Las fórmulas son sólo términos de tipo 'bool' y los conectivos lógicos son términos de tipo 'función' [GuE89].

2.1 Cálculo lambda

La Lógica de Alto Orden está basada en la teoría simple de tipos, que es una variación del cálculo lambda puro introducido por Church en la década de los 30's. El cálculo lambda puro sólo tiene tres constructores: variables, aplicación de funciones y creación de funciones. Su riqueza proviene de la libertad para crear y aplicar funciones, especialmente funciones de más alto nivel construidas por medio de funciones.

El nombre de cálculo lambda proviene de la letra griega lambda, λ. Se utiliza la notación $\lambda x.\, M$ para representar una función con parámetro x y cuerpo M. Así, $\lambda x.\, x * x$ es una función que hace corresponder *5* con *5*5*. Las funciones se escriben junto a sus argumentos, así que $f\ a$ representa la aplicación de la función f al argumento a, como en seno θ o $\log n$. En $(\lambda x.\, x * x)5$ la función $(\lambda x.\, x * x)$ se aplica a *5*. Las fórmulas como $(\lambda x. x * x)5$ se denominan términos.

Los términos del cálculo lambda puro se expresan mediante la siguiente gramática:

$$M ::= x \mid c \mid (M_1\ M_2) \mid (\lambda x.\, M)$$

Un término puede ser una variable x, una aplicación $(M_1\ M_2)$ de la función M_1 a M_2, o una abstracción $(\lambda x.\, M)$. Una constante c puede representar valores como enteros y operaciones sobre estructuras de datos como listas.

Los cálculos lambda son, por lo tanto, una familia de lenguajes para hacer cálculos con funciones. Los miembros de la familia se obtienen escogiendo un conjunto de constantes. De manera informal, el cálculo lambda se refiere a cualquier miembro de esta familia.

El cálculo lambda puro no tiene tipos. Las funciones pueden aplicarse sin restricciones [SeR89].

2.2 Lógica de alto orden

La Lógica de Alto Orden se puede organizar en una jerarquía de tipos, términos, secuentes, reglas de inferencia y teorías.

Los tipos y términos de HOL son expresiones que denotan conjuntos y elementos de conjuntos en un universo $\mathcal{U}$. Se asume que este universo tiene las siguientes propiedades:

Inhab Cada elemento de $\mathcal{U}$ es un conjunto no vacío.

Sub Si $X \in \mathcal{U}$ y $\emptyset \neq Y \subseteq X$, entonces $Y \in \mathcal{U}$.

Prod Si $X \in \mathcal{U}$ y $Y \in \mathcal{U}$, entonces $X \times Y \in \mathcal{U}$. El conjunto $X \times Y$ es el producto cartesiano, que consiste de pares ordenados (x, y) con $x \in X$ y $y \in Y$, con el significado usual de pares ordenados, $(x, y) = \{\{x\}, \{x, y\}\}$.

Pow Si $X \in \mathcal{U}$, entonces el conjunto potencia $P(X) = \{Y : Y \subseteq X\}$ también es un elemento de $\mathcal{U}$.

Infty $\mathcal{U}$ contiene un conjunto distinguido I, el cual es un conjunto infinito.

Choice Hay un elemento distinguido $ch \in \prod_{X \in \mathcal{U}} X$. Los elementos del producto $\prod_{X \in \mathcal{U}} X$ son funciones.

Además, hay algunas consecuencias de estas propiedades.

Fun Si $X \in \mathcal{U}$ y $Y \in \mathcal{U}$, entonces $X \to Y \in \mathcal{U}$.

En teoría de conjuntos, las funciones son identificadas con sus grafos, los cuales son ciertos conjuntos de pares ordenados. Así, el conjunto $X \to Y$ de todas las funciones del conjunto X al conjunto Y es un subconjunto de $P(X \times Y)$. Como $X \in \mathcal{U}$ y $Y \in \mathcal{U}$, aplicando **Prod** se obtiene $X \times Y \in \mathcal{U}$, y por **Pow** obtenemos $P(X \times Y) \in \mathcal{U}$. Como $X \to Y \subseteq P(X \times Y)$, aplicando **Sub** se obtiene que $X \to Y \in \mathcal{U}$.

Unit $\mathcal{U}$ contiene un conjunto distinguido 1, el cual sólo contiene un elemento, $1 = \{0\}$.

Bool $\mathcal{U}$ contiene un conjunto distinguido 2, el cual tiene sólo dos elementos, $2 = \{0, 1\}$.

Aplicando la propiedad **Sub** a cualquier conjunto en $\mathcal{U}$, tenemos que existen conjuntos de sólo un elemento y conjuntos de sólo dos elementos. En particular nos interesan los conjuntos $\{0\}$ y $\{0, 1\}$.

2.2.1 Tipos

Los tipos de HOL son expresiones que denotan conjuntos en el universo $\mathcal{U}$. Los tipos se construyen a partir de variables de tipo y constructores de tipo.

Las variables de tipo se utilizan para representar conjuntos arbitrarios en el universo [HolD91].

En adición a las variables de tipo, para cada número natural n, existe un número finito de constructores de tipo de aridad n. Si $\sigma_1, \ldots, \sigma_n$ son tipos y con_n es un constructor de tipo de aridad n entonces $(\sigma_1, \ldots, \sigma_n)con_n$ es un tipo y denota el conjunto resultante de aplicar la operación denotada por con_n a los conjuntos denotados por $\sigma_1, \ldots, \sigma_n$. A los constructores de tipo también se les denomina operadores de tipo.

A los constructores de tipo de aridad 0 se les llama *tipos atómicos* y denotan conjuntos fijos en el universo. Existen dos tipos atómicos primitivos, bool e ind, que denotan respectivamente el conjunto {0,1} y a un conjunto infinito [GuE89].

Si σ_1 y σ_2 son tipos, entonces $\sigma_1 \to \sigma_2$ es un tipo función con dominio σ_1 y rango σ_2 y denota el conjunto de todas las funciones totales que van del conjunto denotado por su dominio al conjunto denotado por su rango. Sintácticamente $\to$ es simplemente un operador de tipo de aridad 2 escrito en notación infija [HolD91].

Estructuras de tipo

Se asume que existe un conjunto infinito *Nom_Tipos* de nombres de los constructores de tipo. También se asume que existe un conjunto infinito *Var_Tipos* de variables de tipos. Los conjuntos *Nom_Tipos* y *Var_Tipos* son disjuntos.

Una *estructura de tipos* es un conjunto Ω de constantes de tipo. Una constante de tipo es un par (ν, n) donde $\nu \in Nom\_Tipos$ es el nombre del constructor y n es su aridad. Se asume que dos distintas constantes de tipo no pueden tener el mismo nombre, es decir, si $(\nu, n_1) \in \Omega$ y $(\nu, n_2) \in \Omega$, entonces $n_1 = n_2$.

El conjunto de tipos $Tipos_\Omega$ sobre una estructura Ω se define como el conjunto más pequeño tal que:

- $Var\_Tipos \subseteq Tipos_\Omega$. Las variables de tipo están en el conjunto de tipos.
- Si $(\nu, 0) \in \Omega$ entonces $(\ )\nu \in Tipos_\Omega$. Los tipos atómicos están en el conjunto de tipos.
- Si $(\nu, n) \in \Omega$ y $\sigma_i \in Tipos_\Omega$ para $1 \leq i \leq n$, entonces $(\sigma_1, \ldots, \sigma_n)\nu \in Tipos_\Omega$. El conjunto $Tipos_\Omega$ es cerrado bajo los operadores de tipo.
- Si $\sigma_1 \in Tipos_\Omega$ y $\sigma_2 \in Tipos_\Omega$ entonces $\sigma_1 \to \sigma_2 \in Tipos_\Omega$. En particular $Tipos_\Omega$ es cerrado bajo el operador de tipo $\to$.

La notación $tyvars(\sigma)$ se usa para denotar el conjunto de variables de tipo que ocurren en σ

Semántica de tipos

Un *modelo* M de una estructura de tipo Ω se especifica dando para cada constante de tipo (ν, n) una función

$$M(\nu) \ : \ \mathcal{U}^n \to \mathcal{U}$$

Así, dados los conjuntos $X_1, \ldots, X_n$ en el universo $\mathcal{U}$, $M(\nu)(X_1, \ldots, X_n)$ es también un conjunto en el universo. En el caso $n = 0$, hay que especificar un elemento $M(\nu) \in \mathcal{U}$ para el tipo atómico ν.

Los tipos que no contienen variables de tipo se llaman monomórficos, mientras que los que si contienen variables de tipo se llaman polimórficos.

Un *contexto de tipo*, αs, es una lista finita (puede ser una lista vacía) de distintas variables de tipo $\alpha_1, \ldots, \alpha_n$. Un *tipo en contexto* es un par $(\alpha s, \sigma)$, donde αs es un contexto de tipo, σ es un tipo (sobre alguna estructura de tipo dada) y todas las variables de tipo que ocurren en σ también aparecen en la lista αs. La lista αs también puede contener variables de tipo que no ocurran en σ. Se utiliza $\alpha s.\sigma$ para denotar el par $(\alpha s, \sigma)$.

Se dice que αs es un contexto mínimo para σ si $\alpha s.\sigma$ es un tipo en contexto y αs sólo contiene variables de tipo que aparecen en σ. Para cada σ pueden existir varios contextos mínimos αs, los cuales sólo difieren en el orden en el cual las variables de tipo de σ son listadas en αs. Para seleccionar alguno de estos contextos, asumimos que *Var_Tipos* tiene algún orden fijo y definimos el *contexto canónico* del tipo σ como una lista ordenada de las variables de tipo que contiene σ.

Sea M un modelo de una estructura de tipos Ω. Para cada tipo en contexto $\alpha s.\sigma$ sobre Ω, se define una función

$[\![\alpha s.\sigma]\!]_M \ : \ \mathcal{U}^n \to \mathcal{U}$

donde n es la longitud del contexto de tipo. La función se define por indución sobre la estructura de σ como sigue:

- Si σ es una variable de tipo, σ debe ser σ_i para algún único i (donde $1 \leq i \leq n$) y entonces $[\![\alpha s.\sigma]\!]_M$ es la i-ésima función proyección, la cual envía $(X_1, \ldots, X_n) \in \mathcal{U}^n$ a $X_i \in \mathcal{U}$.

- Si σ es un tipo función $\sigma_1 \to \sigma_2$, entonces $[\![\alpha s.\sigma]\!]_M$ envía $Xs \in \mathcal{U}^n$ al conjunto de todas las funciones desde $[\![\alpha s.\sigma_1]\!]_M(Xs)$ a $[\![\alpha s.\sigma_2]\!]_M(Xs)$. Esto hace uso de la propiedad **Fun** de $\mathcal{U}$ que nos dice que si $X \in \mathcal{U}$ y $Y \in \mathcal{U}$ entonces $X \to Y \in \mathcal{U}$.

- Si σ es un tipo compuesto $(\sigma_1, \ldots, \sigma_m)\nu$, entonces $[\![\alpha s.\sigma]\!]_M$ envía Xs a $M(\nu)(S_1, \ldots, S_m)$ donde cada S_j es $[\![\alpha s.\sigma_j]\!]_M(Xs)$.

El significado de un tipo σ en un modelo M es la función

$[\![\sigma]\!]_M \ : \ \mathcal{U}^n \to \mathcal{U}$

dada por $[\![\alpha s.\sigma]\!]_M$, donde αs es el contexto canónico de σ. Si σ es monomórfico, entonces $n = 0$ y $[\![\sigma]\!]_M$ puede ser identificado con el elemento $[\![\sigma]\!]_M(\ )$ de $\mathcal{U}$. Si se sabe por el contexto cual es el modelo M, entonces se puede omitir.

Así, dado un modelo en $\mathcal{U}$ de una estructura de tipo Ω, la semántica interpreta los tipos monomórficos sobre Ω como conjuntos en $\mathcal{U}$ y a los tipos polimórficos que involucran n variables de tipo los interpreta como funciones $\mathcal{U}^n \to \mathcal{U}$ en el universo. Los tipos función son interpretados por conjuntos de funciones totales.

Ejemplo
Suponer que Ω contiene un tipo constante $(b, 0)$ y que el modelo M le asigna a b el conjunto 2. Entonces:

1. $[\![b \to b \to b]\!] = 2 \to 2 \to 2 \in \mathcal{U}$.
2. $[\![(\alpha \to b) \to \alpha]\!] : \mathcal{U} \to \mathcal{U}$ es la función que envía $X \in \mathcal{U}$ a $(X \to 2) \to X \in \mathcal{U}$.
3. $[\![(\alpha, \beta).(\alpha \to b) \to \alpha]\!] : \mathcal{U}^2 \to \mathcal{U}$ es la función que envía $(X, Y) \in \mathcal{U}^2$ a $(X \to 2) \to X \in \mathcal{U}$. $\diamond$[1]

Si el tipo τ es el resultado de reemplazar simultáneamente todas las ocurrencias de algún conjunto de variables de tipo $\sigma_1, \ldots, \sigma_n$ por algún conjunto de tipos $\nu_1, \ldots, \nu_n$ en el tipo σ, entonces se dice que τ es una instancia de σ. Todo tipo es una instancia de sí mismo y la única instancia de un tipo monomórfico es él mismo [HolD91].

2.2.2 Términos

Los términos de HOL son expresiones que denotan elementos de los conjuntos denotados por tipos [HolD91]. Todo término en HOL tiene asociado un tipo.

Los términos se construyen a partir de constantes, variables, aplicación de funciones y abstracciones lambda. Los términos se expresan mediante la siguiente gramática:

$$t_\sigma ::= x_\sigma \mid c_\sigma \mid (t_{\sigma' \to \sigma}\, t'_{\sigma'})_\sigma \mid (\lambda x_{\sigma_1} . t_{\sigma_2})_{\sigma_1 \to \sigma_2}$$

Los subíndices indican el tipo de los términos. El símbolo '\' es la manera de representar a λ en HOL.

Si se compara esta gramática con la gramática para un término del cálculo lambda puro, se nota que son muy semejantes, de hecho la diferencia radica en que en HOL a cada término se le asocia un tipo.

Así como la definición de tipos es relativa a una estructura de tipos Ω, la definición formal de términos es relativa a una colección dada de constantes de tipo sobre Ω.

Se asume que existe un conjunto infinito *Nombres* de nombres. Una *constante* sobre Ω es un par (c, σ) donde $c \in$ *Nombres* y $\sigma \in Tipos_\Omega$. Una *signatura* sobre Ω es un conjunto Σ_Ω de tales constantes.

El conjunto $Terms_{\Sigma_\Omega}$ de términos sobre Σ_Ω se define como el conjunto más pequeño tal que:

1. **Constantes**: Si $(c, \sigma) \in \Sigma_\Omega$ y $\sigma' \in Tipos_\Omega$ es una instancia de σ, entonces $(c, \sigma') \in Terms_{\Sigma_\Omega}$.
2. **Variables**: Si $x \in$ *Nombres* y $\sigma \in Tipos_\Omega$, entonces var $x_\sigma \in Terms_{\Sigma_\Omega}$. Términos formados de esta manera son llamados variables. La palabra var es solamente para distinguir variables de constantes con el mismo nombre. Una variable var x_σ se escribirá x_σ si se sabe por el contexto que x es una variable y no una constante.

[1] El símbolo $\diamond$ significa fin de ejemplo.

3. **Aplicaciones de función**: Si $t_{\sigma' \to \sigma} \in Terms_{\Sigma\Omega}$y $t'_{\sigma'} \in Terms_{\Sigma\Omega}$, entonces $(t_{\sigma'\to\sigma} t'_{\sigma'})_\sigma \in Terms_{\Sigma\Omega}$(los términos formados de esta manera algunas veces son llamados combinaciones).

4. **Abstracciones** λ: Si var $x_{\sigma_1} \in Terms_{\Sigma\Omega}$ y $t_{\sigma_2} \in Terms_{\Sigma\Omega}$, entonces $(\lambda x_{\sigma_1}.\, t_{\sigma_2})_{\sigma_1 \to \sigma_2} \in Terms_{\Sigma\Omega}$.

Todo término en HOL es una constante, una variable, una aplicación de dos términos, o una abstracción de una variable a partir de un término.

Aunque todo término en HOL tiene asociado un tipo, la información del tipo puede ser omitida si se puede inferir a partir del contexto.

La aplicación de funciones se asocia por la izquierda, así que $t\ t_1\ t_2 \ldots t_n$ abrevia a $(\ldots((t\ t_1)t_2)\ldots)$. El término $\lambda x_1.\, (\lambda x_2. \cdots (\lambda x_n.\, t)\ldots)$ se abrevia con $\lambda x_1\ x_2 \cdots x_n.\, t$.

Un término es llamado polimórfico si contiene una variable de tipo, de otra manera es llamado monomórfico. Un término t_σ puede ser polimórfico aunque σ sea monomórfico; por ejemplo, $(f_{\alpha\to b} x_\alpha)_b$, donde b es un tipo atómico.

Una ocurrencia de una variable x_σ es llamada acotada si ocurre dentro del alcance del símbolo λ (λx_σ), de otra manera la ocurrencia es llamada libre. Un término en el que todas las variables son acotadas se llama *término cerrado*.

La idea del término lambda $\lambda x_\sigma.\, t$, es que es una función la cual cuando se da un valor para x, regresa el resultado de substituir el valor de todas las ocurrencias libres de x en t.

Semántica de términos

Un *contexto* es un par $\alpha s, xs$, donde αs es un contexto tipo y xs es una lista de variables distintas, cuyos tipos sólo contienen variables de tipo de la lista αs.

Un *término en contexto* $\alpha s, xs.t$ está compuesto de un contexto y de un término t que satisfacen las siguientes condiciones:

1. αs contiene cualquier variable de tipo que ocurra en xs y en t.

2. xs contiene cualquier variable que ocurra libremente en t.

3. xs no debe contener variables que estén acotadas en t.

El contexto $\alpha s, xs$ puede contener variables de tipo que no aparezcan en t. La combinación de las condiciones 2 y 3 implica que una variable no puede tener ocurrencias libres y acotadas en t.

Se dice que $\alpha s, xs$ es un contexto mínimo para t si $\alpha s, xs.t$ es un término en contexto, αs sólo contiene variables de tipo que aparecen en los tipos de xs y xs sólo contiene variables que aparecen en t. Para cada t pueden existir varios contextos mínimos $\alpha s, xs.t$, los cuales sólo difieren en el orden en el cual las variables son listadas. Al igual que con las variables de tipo, asumimos que hay un orden fijo en las variables y definimos el *contexto canónico* del término t como un contexto mínimo con las variables listadas en orden.

Sea Σ_Ω una signatura sobre una estructura de tipo Ω. Un modelo M de Σ_Ω se especifica por un modelo de la estructura de tipo y para cada constante $(c, \sigma) \in \Sigma_\Omega$ un elemento

$$M(c,\sigma) \in \prod_{Xs\in\mathcal{U}^n} [\![\sigma]\!]_M(Xs)$$

del producto cartesiano indicado, donde n es el número de variables de tipo que ocurren en σ. En otras palabras $M(c,\sigma)$ es un función que asigna a cada $Xs \in \mathcal{U}^n$ un elemento de $[\![\sigma]\!]_M(Xs)$. En el caso de que $n = 0$, $[\![\sigma]\!]_M$ se identifica con un conjunto en $\mathcal{U}$ y entonces $M(c,\sigma)$ se puede identificar con un elemento de ese conjunto.

La semántica de HOL interpreta los términos con base en sus variables y variables de tipo. Los términos cerrados que no involucran variables de tipo se interpretan como elementos de conjuntos en $\mathcal{U}$. Si un término cerrado involucra n variables de tipo, estonces se interpreta como un elemento de un producto $\prod_{Xs\in\mathcal{U}^n} Y(Xs)$ donde la función $Y : \mathcal{U}^n \to \mathcal{U}$ se deriva desde el tipo del término.

Por otro lado, si un término involucra m variables libres sin variables de tipo, entonces se interpreta como una función $Y_1 \times \cdots \times Y_m \to Y$ donde los conjuntos $Y_1, \cdots, Y_m$ en $\mathcal{U}$ son las interpretaciones de los tipos de las variables libres en el término y el conjunto $Y \in \mathcal{U}$ es la interpretación del tipo del término.

Finalmente, el caso más general es el de un término que involucra n variables de tipo y m variables libres, el cual se interpreta como un elemento del producto

$$\prod_{Xs\in\mathcal{U}^n} Y_1(Xs) \times \cdots \times Y_m(Xs) \to Y(Xs)$$

donde las funciones $Y_1, \ldots, Y_m : \mathcal{U}^n \to \mathcal{U}$ son determinadas por los tipos de las variables libres y la función $Y : \mathcal{U}^n \to \mathcal{U}$ es determinada por el tipo del término.

Más precisamente, dado un término en contexto $\alpha s, xs.t$ sobre Σ_Ω, suponer

- t tiene tipo τ
- $xs = x_1, \ldots, x_m$ y cada x_i tiene tipo σ_j
- $\alpha s = \alpha_1, \ldots, \alpha_n$

Entonces, dado que $\alpha s, xs.t$ es un término en contexto, $\alpha s.\tau$ y $\alpha s.\sigma_j$ son tipos en contexto, y por lo tanto, dan origen a las funciones $[\![\alpha s.\tau]\!]_M$ y $[\![\alpha s.\sigma_j]\!]_M$ desde $\mathcal{U}^n$ a $\mathcal{U}$. El significado de $\alpha s, xs.t$ en el modelo M es dado por un elemento

$$[\![\alpha s, xs.t]\!]_M \in \prod_{Xs\in\mathcal{U}^n} \left(\prod_{j=1}^m [\![\alpha s.\sigma_j]\!]_M(Xs)\right) \to [\![\alpha s.\tau]\!]_M(Xs).$$

En otras palabras, dados

$$Xs = (X_1, \ldots, X_n) \in \mathcal{U}^n$$
$$ys = (y_1, \ldots, y_m) \in [\![\alpha s.\sigma_1]\!]_M(Xs) \times \cdots \times [\![\alpha s.\sigma_m]\!]_M(Xs)$$

se toma un elemento $[\![\alpha s, xs.t]\!]_M(Xs)(ys)$ de $[\![\alpha s.\tau]\!]_M(Xs)$. La definición de $[\![\alpha s, xs.t]\!]_M$ se hace por inducción sobre la estructura del término t como sigue:

- Si t es una variable, t debe ser x_j para algún único j (donde $1 \leq j \leq m$), así que $\tau = \sigma_j$ y entonces $[\![\alpha s, xs.t]\!]_M(Xs)(ys)$ se define como y_j.

- Suponer que t es una constante $c_{\sigma'}$ donde $(c, \sigma) \in \Sigma_\Omega$ y σ' es una instancia de σ, es decir que $\sigma' = \sigma[\beta_1, \ldots, \beta_p/\tau_1, \ldots, \tau_p]$ donde $\tau_1, \ldots, \tau_p$ son tipos y $\beta_1, \ldots, \beta_p$ son las variables de tipo que ocurren en σ. Se define $[\![\alpha s, xs.t]\!]_M(Xs)(ys)$ como $M(c,\sigma)([\![\alpha s.\tau_1]\!]_M(Xs), \ldots, [\![\alpha s.\tau_p]\!]_M(Xs))$, el cual es un elemento de $[\![\alpha s.\tau]\!]_M(Xs)$.

- Suponer que t es un término aplicación de función $(t_1\ t_2)$ donde t_1 es de tipo $\tau' \to \tau$ y t_2 es de tipo τ'. Entonces $f = [\![\alpha s, xs.t_1]\!]_M(Xs)(ys)$, siendo un elemento de $[\![\alpha s.\tau' \to \tau]\!]_M(Xs)$, es una función desde el conjunto $[\![\alpha s.\tau']\!]_M(Xs)$ al conjunto $[\![\alpha s.\tau]\!]_M(Xs)$ la cual se puede aplicar al elemento $y = [\![\alpha s, xs.t_2]\!]_M(Xs)(ys)$. La definición de $[\![\alpha s, xs.t]\!]_M(Xs)(ys)$ es $f(y)$.

- Suponer que t es un término abstracción $\lambda x.\, t_2$, donde x es de tipo τ_1 y t_2 de tipo τ_2. Así $\tau = \tau_1 \to \tau_2$ y $[\![\alpha s.\tau]\!]_M(Xs)$ es el conjunto de funciones $[\![\alpha s.\tau_1]\!]_M(Xs) \to [\![\alpha s.\tau_2]\!]_M(Xs)$. La definición de $[\![\alpha s, xs.t]\!]_M(Xs)(ys)$ es el elemento de este conjunto, el cual es la función que envía $y \in [\![\alpha s.\tau_1]\!]_M(Xs)$ a $[\![\alpha s, (xs, x).t_2]\!]_M(Xs)(ys, y)$.

El significado de un término t_r en un modelo M es la función

$$[\![t_r]\!]_M \in \prod_{Xs \in \mathcal{U}^n} \left(\prod_{j=1}^{m} [\![\alpha s.\sigma_j]\!]_M(Xs)\right) \to [\![\alpha s.\tau]\!]_M(Xs)$$

dada por $[\![\alpha s, xs.t_r]\!]_M$, donde $\alpha s, xs$ es el contexto canónico de t_r. Así, n es el número de variables de tipo en t_r, αs es una lista de esas variables de tipo, m es el número de variables que ocurren libremente en t_r y $\sigma_1, \ldots, \sigma_m$ son los tipos de estas variables.

Si t_r es un término cerrado, entonces $m = 0$ y para cada $Xs \in \mathcal{U}^n$ se puede relacionar $[\![t_r]\!]_M$ con el elemento $[\![t_r]\!]_M(Xs)(\ ) \in [\![\alpha s.\tau]\!]_M(Xs)$. Así que para términos cerrados se toma

$$[\![t_r]\!]_M \in \prod_{Xs \in \mathcal{U}^n} [\![\alpha s.\tau]\!]_M(Xs)$$

donde αs es la lista de variables de tipo que ocurren en t_r y n es la longitud de esta lista. Si además, en t_r no ocurren variables de tipo, entonces $n = 0$ y $[\![t_r]\!]_M$ se puede identificar con el elemento $[\![t_r]\!]_M(\ )$ del conjunto $[\![\tau]\!]_M \in \mathcal{U}$. Si se sabe por el contexto cual es el modelo M, entonces se puede omitir.

Estructuras de tipo y signaturas estándar

Una estructura de tipo Ω es estándar si contiene los tipos atómicos *bool* (de valores de verdad) e *ind* (de individuales).

Un modelo M de Ω es estándar si *M(bool)* y *M(ind)* son respectivamente los conjuntos distinguidos 2 e I en el universo $\mathcal{U}$.

Una signatura Σ_Ω es estándar si contiene las siguientes constantes

$=_{\alpha \to \alpha \to bool}$

$\Rightarrow_{bool \to bool \to bool}$

$\epsilon_{(\alpha \to bool) \to \alpha}$

La interpretación de estas constantes es que $=_{\alpha\to\alpha\to bool}$ denota igualdad en el conjunto denotado por α, $\Rightarrow$ denota implicación y $\epsilon_{(\alpha\to bool)\to\alpha}$ denota una función de selección en el conjunto denotado por α.

2.2.3 Secuentes, teoremas y reglas de inferencia

En HOL las pruebas de deducción natural son implantadas usando una forma de cálculo de *secuentes* [GuE89]. Un secuente en HOL es un par (Γ, t) donde Γ es un conjunto finito de fórmulas (términos de tipo bool) y t es sólo una fórmula. El conjunto de fórmulas Γ que forma el primer componente de un secuente es llamado su conjunto de premisas y el término t que forma el segundo componente es llamado conclusión.

Un sistema deductivo $\mathcal{D}$ es un conjunto de pares $(\mathcal{L}, (\Gamma, t))$ donde $\mathcal{L}$ es una lista (posiblemente vacía) de secuentes y (Γ, t) es un secuente.

Un secuente (Γ, t) se sigue a partir de un conjunto de secuentes Δ por un sistema deductivo $\mathcal{D}$ si y solo si existen secuentes $(\Gamma_1, t_1), \ldots, (\Gamma_n, t_n)$ tales que:

1. $(\Gamma, t) = (\Gamma_n, t_n)$.
2. para toda i tal que $1 \leq i \leq n$
   (a) $(\Gamma_i, t_i) \in \Delta$.
   (b) $(\mathcal{L}_i, (\Gamma_i, t_i)) \in \mathcal{D}$ para alguna lista $\mathcal{L}_i$ de miembros de $\Delta \cup \{(\Gamma_1, t_1), \ldots, (\Gamma_{i-1}, t_{i-1})\}$.

La secuencia $(\Gamma_1, t_1), \ldots, (\Gamma_n, t_n)$ es llamada una prueba de (Γ, t) a partir de Δ con respecto a $\mathcal{D}$. La notación $t_1, \ldots, t_n \vdash_{\mathcal{D},\Delta} t$, significa que la secuencia $(\{t_1, \ldots, t_n\}, t)$ se sigue a partir de Δ mediante $\mathcal{D}$. Si se sabe por el contexto quienes son $\mathcal{D}$ o Δ, entonces se pueden omitir. En el caso de que no haya premisas ($n = 0$), se escribirá $\vdash t$.

Sistema deductivo de HOL

Una *regla de inferencia* en HOL es una regla que permite obtener un secuente a partir de un conjunto de secuentes. Una regla de inferencia toma la siguiente forma:

$$\frac{\Gamma_1 \vdash t_1 \quad \cdots \quad \Gamma_n \vdash t_n}{\Gamma \vdash t}$$

Los secuentes arriba de la línea se llaman *hipótesis* de la regla y el secuente bajo la línea se llama *conclusión* de la regla. Así, la regla nos indica que se puede obtener el secuente $\Gamma \vdash t$ a partir de los secuentes $\Gamma_1 \vdash t_1 \quad \cdots \quad \Gamma_n \vdash t_n$.

En la práctica, un sistema deductivo usualmente se especifica por un número de reglas de inferencia. El sistema deductivo de HOL se especifica por ocho reglas de inferencia dadas a continuación. Los identificadores entre corchetes son los nombres de las funciones ML que implantan las correspondientes reglas de inferencia [HolD91].

Introducción de Premisa [ASSUME]
Esta regla nos indica que se pueden generar secuentes de la forma $t \vdash t$ sin necesidad de hipótesis.

$$\overline{t \vdash t}$$

Reflexividad [REFL]
La regla nos indica que cualquier fórmula es igual a si misma, y que esto se puede concluir sin necesidad de premisas o hipótesis.

$$\overline{\vdash t = t}$$

Conversión Beta [BETA_CONV]
En el cálculo lambda el término $\lambda x.\, t_1$ representa una función con parámetro x y cuerpo t_1. La *conversión beta* se relaciona con el resultado de aplicar una abstracción $\lambda x.\, t_1$ a un argumento t_2. En otras palabras, la conversión beta se relaciona con las nociones de invocación a una función y paso de parámetros. El término $(\lambda x.\, t_1)\, t_2$ representa la invocación de la función $\lambda x.\, t_1$ con el argumento t_2, por lo tanto $(\lambda x.\, t_1)\, t_2$ es igual a la substitución de x con t_2 en el término t_1.

$$\overline{\vdash (\lambda x.\, t_1)\, t_2 = t_1[x/t_2]}$$

- Donde $t_1[x/t_2]$ es el resultado de substituir x con t_2 en t_1, con el renombramiento apropiado de variables para prevenir que las variables libres en t_2 se conviertan en acotadas después de la sustitución.

Substitución [SUBST]
Si tenemos una igualdad de términos $t_1 = t_2$ y en el término t hay ocurrencias de t_1, mediante esta regla se puede substituir en t cada ocurrencia de t_1 por t_2.

$$\frac{\Gamma_1 \vdash t_1 = t_1' \quad \cdots \quad \Gamma_n \vdash t_n = t_n' \qquad \Gamma \vdash t[t_1, \ldots, t_n]}{\Gamma_1 \cup \cdots \cup \Gamma_n \cup \Gamma \vdash t[t_1', \cdots t_n']}$$

- Donde $t[t_1, \ldots, t_n]$ denota un término t con algunas ocurrencias libres de subtérminos $t_1, \ldots, t_n$; y $t[t_1', \ldots, t_n']$ denota el resultado de reemplazar cada ocurrencia seleccionada de t_i por t_i' (para $1 \leq i \leq n$), con el renombramiento apropiado de variables para prevenir que las variables libres en t_i' se conviertan en acotadas después de la sustitución.

Abstracción [ABS]
Si los términos t_1 y t_2 son iguales, entonces también serán iguales las abstracciones $(\lambda x.\, t_1) = (\lambda x.\, t_2)$.

$$\frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash (\lambda x.\, t_1) = (\lambda x.\, t_2)}$$

- Asegurarse que x no esté libre en Γ, ya que si x está libre en Γ, al aplicar la regla de inferencia, x se volverá acotada.

Instanciación de Tipos [INST_TYPE]
Si un término t tiene variables de tipo, mediante esta regla, esas variables se pueden instanciar.

$$\frac{\Gamma \ \vdash \ t}{\Gamma \ \vdash \ t[\alpha_1, \ldots, \alpha_n / \sigma_1, \ldots, \sigma_n]}$$

- Donde $t[\alpha_1, \ldots, \alpha_n / \sigma_1, \ldots, \sigma_n]$ es el resultado de substituir en paralelo en el término t las variable de tipo $\alpha_1, \ldots, \alpha_n$ con los tipos $\sigma_1, \ldots, \sigma_n$, con la restricción que ninguna de las variables de tipo $\alpha_1, \ldots, \alpha_n$ ocurre en Γ.

Descargando una Premisa [DISCH]
Esta regla es similar al teorema de la deducción en lógica clásica. Si se tiene un secuente $[t_1, t_2, \ldots, t_n] \ \vdash \ t$, se puede obtener el secuente $[t_1, t_2, \ldots, t_n] - [t_i] \ \vdash \ t_i \Rightarrow t$ (para $1 \leq i \leq n$).

$$\frac{\Gamma \ \vdash \ t_2}{\Gamma - \{t_i\} \ \vdash \ t_i \Rightarrow t_2}$$

- Donde $\Gamma - \{t_i\}$ es el conjunto resultante de substraer t_i de Γ. Es importante hacer notar que no es necesario que t_i esté en Γ; en este caso $\Gamma - \{t_i\} = \Gamma$.

Modus Ponens [MP]
Esta regla de inferencia es el equivalente a la regla de demostración *modus ponens* de la lógica clásica.

$$\frac{\Gamma_1 \ \vdash \ t_1 \Rightarrow t_2 \qquad \Gamma_2 \ \vdash \ t_1}{\Gamma_1 \cup \Gamma_2 \ \vdash \ t_2}$$

En adición a estas reglas de inferencia, HOL tiene cinco axiomas principales, que se verán un poco más adelante.

2.2.4 Teorías

Los tipos, constantes y axiomas de HOL se organizan en Teorías. En HOL siempre se trabaja con teorías [HolD91].

Una teoría T de HOL es una 4-tupla

$T = \{Estruct_T, Sig_T, Axiomas_T, Teoremas_T\}$

donde:

- $Estruct_T$ es una estructura de tipos llamada la estructura de tipos de T;
- Sig_T es una signatura sobre $Estruct_T$ llamada la signatura de T;
- $Axiomas_T$ es un conjunto de secuentes sobre Sig_T llamado los axiomas de T;

- $Teoremas_T$ es un conjunto de secuentes sobre Sig_T llamado los teoremas de T, con la propiedad de que todo miembro se sigue a partir de $Axiomas_T$ mediante el sistema deductivo de HOL.

HOL se basa en teorías para la demostración de teoremas, ya que con ellas se lleva un registro de hechos (axiomas, teoremas, etc.) ya probados y disponibles como lemas sin tener que volverlos a probar.

Mediante las teorías se logra que HOL sea extensible. Para extender las teorías se comienza especificando una teoría mínima, se le asigna un modelo que satisfaga los axiomas de T y después se expande especificando nuevos tipos, términos, axiomas y teoremas, pero siguiendo ciertas reglas de tal forma de que se pueda encontrar un modelo (basando en el modelo original) que satisfaga los nuevos axiomas y teoremas. De esta forma, al extender las teorías se va enriqueciendo la lógica; ya que a diferencia de la lógica clásica, en donde se tiene un conjunto fijo de axiomas y a partir de los cuales se demuestran teoremas; en HOL se van coleccionando los axiomas y teoremas para demostrar nuevos teoremas.

En HOL, la teoría mínima es llamada MIN y está definida por

$$\begin{aligned} MIN = (&\{(bool, 0),\ (ind, 0)\},\ \{\Rightarrow_{bool \to bool \to bool}, =_{\alpha \to \alpha \to bool}, \epsilon_{(\alpha \to bool) \to \alpha}\},\\ &\{\ \},\ \{\ \}) \end{aligned}$$

Después se extiende a la teoría LOG, la cual tiene 10 axiomas en donde se definen las constantes lógicas para veracidad, cuantificación universal, cuantificación existencial, falsedad, negación, conjunción y disyunción. Se definen las propiedades de que una función sea inyectiva o suprayectiva y el último axioma define una constante usada para definición de tipos. Ahora ya con diez axiomas se tienen más herramientas para poder demostrar teoremas.

Los axiomas de la teoría LOG consisten de los siguientes secuentes:

$$\begin{array}{ll}
\vdash & T = ((\lambda\, x_{bool}.\ x) = (\lambda\, x_{bool}.\ x)) \\
\vdash & \forall = \lambda\, P_{\alpha \to bool}.\ P = (\lambda\, x.\ T) \\
\vdash & \exists = \lambda\, P_{\alpha \to bool}.\ P(\epsilon\, P) \\
\vdash & F = \forall\, b_{bool}.\ b \\
\vdash & \sim = \lambda\, b.\ b \Rightarrow F \\
\vdash & \wedge = \lambda\, b_1\, b_2.\ \forall\, b.\ (b_1 \Rightarrow (b_2 \Rightarrow b)) \Rightarrow b \\
\vdash & \vee = \lambda\, b_1\, b_2.\ \forall\, b.\ (b1 \Rightarrow b) \Rightarrow ((b_2 \Rightarrow b) \Rightarrow b) \\
\vdash & One_One = \lambda\, f_{\alpha \to \beta}. \forall\, x_1\, x_2.\ (f\ x_1 = f\ x2) \Rightarrow (x_1 = x_2) \\
\vdash & Onto = \lambda\, f_{\alpha \to \beta}.\ \forall\, y.\ \exists\, x.\ y = f\ x \\
\vdash & Type_Definition = \lambda\, P_{\alpha \to bool}\ rep_{\beta \to \alpha}.\ One_One\ rep \\
 & \wedge\ (\forall\, x.\ P\ x = (\exists\, y.\ x = rep\, y))
\end{array}$$

El término *rep* en la definición de *Type_Definition* establece si un nuevo tipo está bien definido.

La siguiente extensión considerada es la teoría INIT, la cual tiene los axiomas de la teoría LOG y se le adicionan los siguientes cinco axiomas.

$$\begin{array}{rcl}
BOOL_CASES_AX & \vdash & \forall b.\,(b = T) \vee (b = F) \\
IMP_ANTISYM_AX & \vdash & \forall b_1\, b_2.\,(b_1 \Rightarrow b_2) \Rightarrow (b_2 \Rightarrow b_1) \Rightarrow (b_1 = b_2) \\
ETA_AX & \vdash & \forall f_{\alpha\rightarrow\beta}.\,(\lambda x.\, f\ x) = f \\
SELECT_AX & \vdash & \forall P_{\alpha\rightarrow bool} x.\, P\ x \Rightarrow P(\epsilon\ P) \\
INFINITY_AX & \vdash & \exists f_{ind\rightarrow ind}.\, One_One\ f \wedge \sim\ (Onto f)
\end{array}$$

La teoría INIT es la teoría inicial de la lógica HOL. Una teoría que se extiende a partir de INIT se llama teoría estándar.

Así se puede ir aumentando axiomas; de hecho cada vez que se define una constante lógica da lugar a un nuevo axioma. Los nuevos axiomas se guardan en teorías y sirven para demostrar nuevos teoremas, los cuales también se pueden incorporar a las teorías. Con esto, la demostración de teoremas en algunos casos se hace más fácil, ya que se pueden utilizar los axiomas y teoremas de alguna teoría, los cuales no necesitan volver a demostrarse.

Capítulo 3

Construcción de ML-HOL

Una vez que hemos definido los conceptos teóricos y las construcciones de ML que se necesitan, analizaremos la construcción del sistema ML-HOL.

Durante la creación de un intérprete, las fases de análisis léxico y sintáctico, sobre todo esta última, pueden ser muy laboriosas y consumir gran parte del trabajo de escribir el intérprete; por lo que para su construcción utilizamos una herramienta que genera analizadores sintácticos. Elegimos a YACC [JoS90], debido a que es una herramienta poderosa, además de que ya hemos desarrollado otros programas con ella.

En este capítulo describimos brevemente a YACC. Después se muestra la gramática que se le da como entrada a YACC; se presentan los algoritmos para la construcción del sistema de inferencia de tipos, del comprobador de tipos y del intérprete; también se explican las estructuras de datos utilizadas más importantes. Finalmente, se da una breve explicación del algoritmo que utilizamos para hacer la recolección de basura.

3.1 YACC -yet anoter compiler-compiler

YACC es un programa que genera un analizador sintáctico (*parser*) a partir de una descripción gramatical de un lenguaje. Para utilizar YACC se prepara una especificación del proceso de entrada; esto incluye reglas que describen la estructura de entrada, código que debe ser invocado cuando estas reglas son reconocidas y una rutina en bajo nivel para la entrada básica.

La entrada que se lea puede no comportarse de acuerdo a la especificación. Estos errores de entrada son detectados lo más pronto posible. El manejo de errores proporcionado como parte de la especificación de entrada, permite el reingreso de los datos erróneos o la continuación del proceso de entrada después de haber "saltado" a todos los datos erróneos.

Aunque YACC tiene muchas características importantes, sólo se explicarán las que se usaron para el desarrollo del intérprete.

3.1.1 Análisis léxico

Cuando se dá una especificación a YACC (el archivo que contenga la especificación debe tener extensión ".y"), la salida es un programa en C[1] cuyo nombre es el mismo que tiene la especificación, pero con extensión ".c". YACC produce una función para controlar el proceso de entrada, llamada *yyparse( )*. Cuando se invoca a *yyparse( )*, ésta llama repetidamente a una función (un *analizador léxico*) que lee la entrada que se analiza, la divide en componentes sintácticos (llamados *tokens*) y se los comunica al parser.

Para generar el analizador léxico se intentó utilizar el programa LEX [LeM90] ya que éste genera analizadores léxicos a partir de la descripción de las expresiones que hay que reconocer y cuenta con la ventaja de que está pensado para trabajar junto con YACC.

El analizador generado por LEX funciona bien cuando se da la definición de una función y hay que analizarla de izquierda a derecha; pero cuando se está interpretando una expresión y se debe de saltar a diferentes partes del código, presenta problemas, ya que LEX para analizar un token analiza el token siguiente (*token lookahead*) y lo guarda en una cola. Si se salta a otra parte del código y el analizador sintáctico pide un token, LEX no regresa el token que está en el flujo de entrada sino que regresa el token que está en la cola, que seguramente no es el que se espera y por lo tanto, se produce un error de sintaxis.

Ejemplo
Sea la función

fun length [ ] = **0**
  | **length(x::l)** = **1** + **length(l);**

Al interpretar la expresión[2] *length [1,2]*, se salta a la parte de código donde está definida *length*. Cuando en el flujo de entrada está el caracter '=', LEX no sabe si ese caracter es parte del token '=>' (usado como símbolo de implicación) o si es propiamente el token '=', y por lo tanto, LEX lee también el siguiente caracter en el flujo de entrada, el cual es 0 y corresponde al token ENTERO. Lo que hace LEX es devolver el token '=' y el token ENTERO lo guarda en una cola. Como ya se ha analizado el encabezado de la función tratamos de unificar *[ ]* con *[1,2]* lo cual falla, por lo tanto, verificamos si la función tiene otro caso. Cuando LEX lee el caracter | , en lugar de regresar el token correspondiente, regresa el token que está disponible en la cola (ENTERO) y entonces se produce un error de sintaxis. ◇

Para evitar estos errores, construimos nuestro propio analizador léxico, en el cual, para analizar un token también analizamos el token siguiente, pero con la diferencia de que este último token se desecha y los caracteres que lo forman se devuelven al flujo de entrada.

[1] YACC genera sus acciones y subrutinas de salida en C. Más aún, muchas de las convenciones sintácticas de YACC siguen a las de C.

[2] En la Sección 3.4 se muestra en forma más específica cómo se interpreta una expresión.

3.1.2 Especificaciones básicas

Todo archivo de especificación consiste de tres secciones: declaraciones, reglas gramaticales y programas. Las secciones son separadas por la marca %%. Un archivo de especificación completo es semejante a:

declaraciones
%%
reglas
%%
programas

Los nombres que representan tokens deben ser declarados, a menos que sean literales de un carácter como '+' y '*'. Esto se hace escribiendo en la sección de *declaraciones*

%token nombre1 nombre2 ...

Cada nombre no definido en la sección de declaraciones se asume que representa un símbolo no terminal.

La sección de *reglas* se compone de una o más reglas gramaticales. Una regla gramatical tiene la forma:

A : CUERPO ;

A representa un nombre no terminal y *CUERPO* representa una secuencia de cero o más nombres y literales. Los dos puntos (:) y el punto y coma (;) son puntuaciones de YACC.

Los nombres usados en el cuerpo de una regla gramatical representan tokens o símbolos no terminales. Una literal consiste de una carácter encerrado entre apóstrofes. Como en el lenguaje C, el backslash '\' es un carácter de escape para literales y todos los escapes de C son reconocidos.

Si hay varias reglas gramaticales con el mismo lado izquierdo, la barra vertical | se usa para evitar reescribir el lado izquierdo.

Todo lo que esté después del delimitador %% que separa las reglas de los programas se copia en el programa de salida.

3.1.3 Acciones en YACC

Con cada regla gramatical se pueden asociar acciones para que se ejecuten cada vez que la regla se reconoce en el proceso de entrada.Una acción se especifica por una o más declaraciones encerradas entre llaves { }. Por ejemplo,

A : '(' B ')' { printf("un mensaje\n"); }

es una regla gramatical con una acción.

Aunque generalmente todas las acciones vienen hasta el final de sus reglas; si se desea tener el control antes de que una regla sea totalmente analizada, YACC también permite que una acción se escriba en medio de una regla. Por ejemplo,

A : B { printf("acción media"); } C { printf("acción final"); }

3.1.4 Precedencia

Muchas de las construcciones usadas comúnmente para expresiones aritméticas pueden ser escritas naturalmente por la noción de niveles de precedencia para operadores, junto con información acerca de la asociatividad (izquierda o derecha).

La noción básica es escribir reglas gramaticales de la forma

expr : expr OP expr

para todos los operadores binarios que se deseen. Esto crea una gramática muy ambigüa, con muchos conflictos de análisis. Como reglas de desambigüedad, se específica la precedencia de todos los operadores y la asociatividad de los operadores binarios. Esta información es suficiente para permitir a YACC resolver los conflictos de análisis y construir un parser que realice la precedencia y asociatividad deseadas.

La precedencia y asociatividad son ligadas a tokens en la sección de declaraciones. Esto se hace con una serie de líneas que empiezan con una palabra clave de YACC: *%left*, *%right*, o *%nonassoc*, seguido por una lista de tokens. Todos los tokens que estén en una misma línea se asume que tienen el mismo nivel de precedencia y la misma asociatividad; las líneas se listan en orden para incrementar la precedencia. Así

%left '+' '-'
%left '' '/'*

describe la precedencia y asociatividad de los cuatro operadores aritméticos. '+' y '-' tienen asociatividad izquierda y tienen menor precedencia que '*' y '/' quienes también tienen asociatividad izquierda. *%right* se usa para describir asociatividad derecha de los operadores y *%nonassoc* se usa para expresar que un operador no puede asociarse con él mismo.

Para cambiar el nivel de precedencia asociado a una regla gramatical se utiliza la palabra clave *%prec*, la cual debe aparecer inmediatamente después del cuerpo de la regla gramatical, antes de la acción o del punto y coma, y es seguido por un nombre de token o literal. Esto cambia la precedencia de la regla gramatical, por la precedencia que tiene el token o la literal.

3.1.5 Manejo de errores

Cuando el parser generado por YACC encuentra un error descarta un número de tokens de la cadena de entrada y se hacen ajustes para que se pueda continuar analizando la entrada.

YACC permite tener algún control sobre este proceso. El token llamado *'error'* se reserva para manejo de errores. Este token puede usarse en reglas gramaticales, para sugerir lugares en donde se esperan los errores y tratar de recuperarse del error. El parser se ajusta hasta que se llegue a un estado donde el token *'error'* sea legal y ejecuta la acción correspondiente. Si no se especificó ninguna regla de error, el análisis se detiene cuando se detecta un error.

Una regla fácil de aplicar es

expr : error ';'

Así, cuando hay un error, el parser intenta saltar toda la expresión hasta que encuentre un ';'.

Si se detecta un error y la recuperación del error no es posible *yyparse( )* regresa el valor 1; pero si el analizador léxico regresa el fin de archivo y el parser lo acepta, *yyparse( )* regresa el valor de 0.

3.2 Gramática de ML

Ahora se mostrará y analizará parte del contenido del archivo fuente (*parser.y*) para YACC, con el que se obtiene el analizador sintáctico del intérprete.

```
%token AND CADENA CONS DATATYPE ELSE END ENTERO
%token EXCEPTION FN FUN GE IDENTIFICADOR IF IN INT
%token LAMBDA LE LET LIST MOD OF OR RAISE STRING
%token THEN TIPO_FUN VAL VOID

%left ';'
%left AND
%left '|'
%left ',', '¿'
%left LAMBDA
%nonassoc '<' '>' LE GE
%nonassoc '='
%nonassoc CONS '@'
%right TIPO_FUN
%left '+' '-'
%left '*' '/' MOD
%nonassoc ':'
%%

s             : inicio program
inicio        :
program       : decl          | error ';'      | program decl
decl          : datatype      | valor ';'      | funcion
              | exception     | expr ';'
datatype      : DATATYPE IDENTIFICADOR '=' nuevos_tipos ';'
nuevos_tipos  : un_nuevo   | nuevos_tipos '|' un_nuevo
un_nuevo      : IDENTIFICADOR    | IDENTIFICADOR OF nuevo
nuevo         : tipo               | '(' varios_tipos ')'
varios_tipos  : tipo '*' tipo      | varios_tipos '*' tipo
valor         : VAL pat '=' expr
funcion       : FUN funcion1 ';'
funcion1      : funcion2           | funcion1 '|' funcion2
```

```
funcion2      : IDENTIFICADOR patron '=' expr
patron        :          | sec_pat
sec_pat       : pat      | sec_pat pat
pat           : pat1     | pat1 ':' tipo
pat1          : pat2     | pat ',' pat     | pat1 CONS pat2
pat2          : IDENTIFICADOR    | ENTERO    | CADENA
              | '[' ']'          | '[' pat ']'  | '(' pat ')'
sec_expr      : expr3    | sec_expr expr3
expr          : expr1
              | LET valor IN expr END
              | IF expr1 THEN expr else
              | param LAMBDA expr
              | VOID
else          :          | ELSE expr
expr1         : expr2    | RAISE IDENTIFICADOR
expr2         : expr3
              | IDENTIFICADOR sec_expr
              | expr2 ',' expr2
              | expr2 op1 expr2          %prec '*'
              | expr2 op2 expr2          %prec '+'
              | expr2 op3 expr2          %prec '='
              | expr2 op4 expr2          %prec '<'
              | expr2 op5 expr2          %prec CONS
              | expr2 op6 expr2          %prec '&'
              | expr2 '?' expr2
              | '-' expr3                %prec '*'
op1           : '*'      | '/'    | MOD
op2           : '+'      | '-'
op3           : '='
op4           : '<'      | '>'    | LE    | GE
op5           : CONS     | '@'
op6           : '&'      | OR
expr3         : IDENTIFICADOR    | ENTERO    | CADENA
              | '[' ']'          | lista     | '(' expr ')'
lista         | '[' expr2 ']'
param         : FN pat           | param LAMBDA param
tipo          : IDENTIFICADOR    | tipo LIST
exception     : EXCEPTION varias_exc ';'
varias_exc    : IDENTIFICADOR    | varias_exc AND IDENTIFICADOR
%%
```

En la sección de declaraciones, se declaran los tokens que se utilizan y la precedencia que tienen los operadores.

En la sección de reglas se muestran todas las reglas necesarias para el intérprete; las reglas adicionales para el demostrador de teoremas se dan en la sección 4.1.

3.3 Sistema de inferencia de tipos y comprobador de tipos

Cada vez que se define una función o una variable se activa el sistema de inferencia de tipos para determinar el tipo de esa función o variable.

El comprobador de tipos se activa en la definición de funciones o variables para verificar que las funciones que se involucran en la definición tengan los argumentos de tipo correcto. También se activa cada vez que se escribe una expresión para verificar que las funciones que se involucran en la expresión sean llamadas con argumentos de tipo correcto. Si la comprobación de tipos es exitosa se puede asegurar que cuando se interprete la expresión no ocurrirá ningún error de tipos.

Se utilizan las siguientes reglas de comprobación e inferencia de tipos:

1. Si x tiene tipo X y y tiene tipo Y, entonces x,y tiene tipo X * Y.
2. Si x tiene tipo X entonces *(x)* tiene tipo X.
3. Si el dominio de una función f tiene tipo X y lo que regresa es de tipo Y, entonces f tiene tipo $X -> Y$.
4. Si una función f tiene tipo $X_1 -> X_2 -> \cdots -> X_n -> Y$ y se tiene la expresión $f\, x_1\, x_2 \ldots x_n$, o si f tiene tipo $X_1 * X_2 * \cdots * X_n -> Y$ y se tiene la expresión $f(x_1, x_2, \ldots, x_n)$, donde Y y X_i (para $1 \leq i \leq n$) son cualquier tipo entonces:
   - Si el tipo de X_i es una variable de tipo entonces no importa el tipo de x_i.
   - Si el tipo de X_i no es una variable de tipo y el tipo de x_i es una variable de tipo, entonces se infiere que el tipo de x_i es X_i.
   - Si X_i y el tipo de x_i no son iguales y no son variables de tipo entonces se marca un error.
5. Si en la definición de una función g, se tiene un parámetro f y en el cuerpo de g se tiene la expresión $f\, x_1\, x_2 \ldots x_n$, y el tipo de x_i es X_i (para $1 \leq i \leq n$) entonces:
   - Si el tipo de f es una variable de tipo, se infiere que su tipo es $X_1 -> X_2 -> \cdots -> X_n -> Y$.
   - Si el tipo de f no es una variable de tipo y no es igual a $X_1 -> X_2 -> \cdots -> X_n -> Y$, entonces se marca un error. Y es una variable de tipo.
6. Si en la definición de una función g, se tiene un parámetro f, en el cuerpo de g se tiene la expresión $f(x_1, x_2, \ldots, x_n)$, y el tipo de x_i es X_i (para $1 \leq i \leq n$) entonces:

- Si el tipo de f es una variable de tipo, se infiere que su tipo es $X_1 * X_2 * \cdots * X_n -> Y$.
- Si el tipo de f no es una variable de tipo y no es igual a $X_1 * X_2 * \cdots * X_n -> Y$, entonces se marca un error. Y es una variable de tipo.

7. Si se tiene la expresion *x::rest* y el tipo de x es X entonces:
   - Si el tipo de *rest* es una variable de tipo entonces se infiere que su tipo es X *list*.
   - Si el tipo de *rest* es X *list* entonces *x::rest* tiene tipo X *list*.
   - Si el tipo de *rest* no es una variable de tipo y no es igual a X *list* entonces se marca un error.

Para realizar la inferencia y comprobación de tipos hacemos las siguientes consideraciones.

El nombre de una variable de tipo inicia con una de las primeras cinco letras del alfabeto seguida de cero o más digitos y finaliza con un apóstrofe. Son variables de tipo: a', b', c', d', e', a1', b1' y así sucesivamente.

Los tipos se representan por medio de árboles (árboles de tipo). En sus nodos interiores se colocan los constructores de tipo y en sus nodos hojas se colocan las variables de tipo o los tipos básicos. Si el árbol F representa el tipo de una función f, entonces el subárbol izquierdo de F (árbol I) representa el tipo que debe recibir la función f y el subárbol derecho de F (árbol D) representa el tipo que regresa f. Sea la función[3]

#fun combine ([ ],[ ])=[ ]
    | combine ((x::lx),(y::ly))=(x,y)::combine(lx,ly);
*val combine : (a' list * b' list) -> (a' * b') list*

La representación del tipo de la función *combine* mediante un árbol binario se muestra en la Figura 3.1.

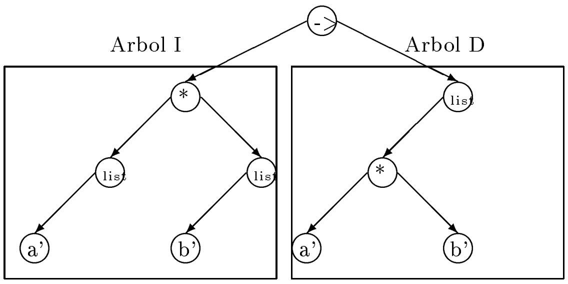

Figura 3.1: Representación del tipo a' list * b' list) -> (a' * b') list.

[3]El indicador de ML-HOL es #, así que las líneas que empiezan con # contienen lo que escribe el usuario.

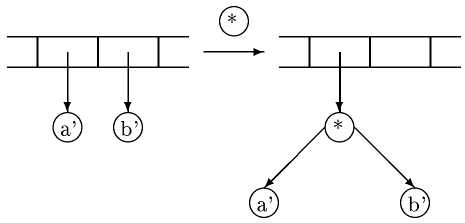

Figura 3.2: Agrupación de tipos.

La principal estructura de datos que utilizamos es una *pila* (la llamamos *pila de evaluación* o simplemente *pila*), ya que sobre ella se hace la comprobación e inferencia de tipos, y cuando se está interpretando una expresión mediante la *pila* se hace el paso de parámetros.

Cada entrada de la *pila* tiene dos campos. En el primer campo almacenamos un apuntador (llamado *sim*) y en el segundo campo almacenamos un arreglo de apuntadores (lo llamamos *pila de valores* o PV). El campo *sim* apunta al árbol de tipo de una función (*árbol de función*) y en la PV se construye un árbol de tipo (*árbol de expresión*), el cual se tratará de unificar con el árbol al que apunta *sim*. Mediante esa unificación se hace propiamente la inferencia y comprobación de tipos. Cuando se mencione a la PV actual o *sim* actual nos estaremos refiriendo a la PV y al apuntador *sim* que están en la cima de la *pila*, respectivamente.

La construcción del árbol de expresión se hace de la siguiente manera:

- Sean $stat_1$ y $stat_2$ dos expresiones, dos patrones o dos argumentos de una función.

- Agrupar dos tipos significa crear un nodo y hacer que los árboles que representan a los dos tipos sean subárboles del nodo creado.

- Si se lee $stat_1, stat_2$ entonces se introducen los tipos de $stat_1$ y $stat_2$ consecutivamente en la PV actual y se agrupan mediante el constructor de tipo *. Ver Figura 3.2.

- Si se lee $stat_1$ $stat_2$ entonces se introducen los tipos de $stat_1$ y $stat_2$ consecutivamente en la PV actual y se agrupan mediante el constructor de tipo $->$.

Los operadores binarios se tratan como funciones y los operandos son los argumentos. Los tipos de las funciones respectivas son:

**, /, +, -, DIV*	*: int * int -> int*
>, <, <=, >=	*: int * int -> bool*
=	*: a' * a' -> bool*
@	*: a' list * a' list -> a' list*
::	*: a' * a' list -> a' list*

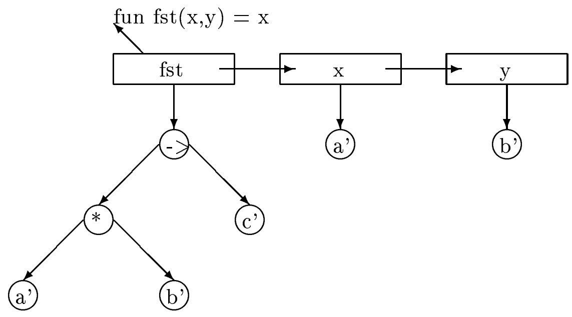

Figura 3.3: Lista creada al leer el encabezado de la función fst(x,y) = x.

Las construcciones *if-then-else* y *(expresión)* también se tratan como funciones:

if : *bool -> a' -> a' -> a'*
par : *a' -> a'*

Tomando en cuenta las consideraciones anteriores, a continuación se explica como se infiere y se comprueba el tipo de una función f.

Se analiza la entrada de izquierda a derecha. Conforme se va leyendo el nombre de la función y los parámetros, se va creando una lista ligada con las siguientes características:

- El primer nodo representa el nombre de la función. En este nodo se coloca un apuntador a la localidad en memoria en donde se está definiendo la función.
- Por cada parámetro se crea un nodo asociándole como tipo una variable de tipo.
- Se crea una variable de tipo que representa el tipo que regresará la función.
- Se construye el árbol de tipo de la función mediante las reglas de agrupación dadas anteriormente. En este árbol sólo existen variables de tipo, las cuales se instanciarán cuando se analice el cuerpo de la función y así obtener el verdadero tipo de la función.

Ejemplo
Sea la función

#fun fst (x,y) = x;

La función *fst* obtiene el primer elemento de un par ordenado. Se analiza la función *fst* de izquierda a derecha y al analizar el nombre y sus argumentos se genera la lista que se muestra en la Figura 3.3. La lista tiene tres nodos, uno que representa el nombre de la lista y dos nodos que representan a los argumentos *x,y*. Cada nodo de la lista apunta a su correspondiente árbol de tipo. ◇

Una vez creada la lista, se empila el árbol de tipo de f[4] y se analiza el cuerpo de la función.

Si se encuentra una expresión *g argumentos* y g es una función previamente definida entonces se empila el árbol de tipo de g (árbol G). Si g no es una función previamente definida entonces se verifica que su tipo sea una variable de tipo. Si no lo es, se marca un error. Si lo es, se sustituye esa variable por un tipo de la forma *a' −> b'* y se empila el árbol de tipo de g.

Se leen los argumentos de g. Cuando se han leído todos sus argumentos y sus tipos se han agrupado y formado un árbol A, se trata de unificar el árbol A con el subárbol izquierdo de G (árbol I), es decir, unificamos el tipo que recibe la función con el tipo que realmente debe recibir.

Para hacer la unificación recorremos juntos los árboles A e I, es decir, si visitamos el árbol izquierdo de A, también debemos visitar el árbol izquierdo de I. En la unificación, hacemos al mismo tiempo la inferencia y la comprobación de tipos. Si los nodos que estamos visitando tienen tipos básicos, entonces ellos deben ser iguales (comprobación), pero si alguno tiene una variable de tipo, esta variable es sustituida por el tipo que tiene el otro nodo (inferencia). Cuando se sustituye una variable de tipo por otro tipo, la variable se debe sustituir en todas sus ocurrencias.

Si la unificación falla, es indicio de que la función fue aplicada a argumentos de tipo incorrecto y se marca un error. Pero si la unificación es exitosa, se saca lo que esté en la cima de la *pila* y el tipo que regresa la función (subárbol derecho de G) se empila en la PV que ahora está en la cima de la *pila*.

Se puede decir que la comprobación e inferencia de tipos se basa en empilar tipos, construir el árbol de expresión, unificar este árbol con el árbol de función y empilar el subárbol derecho del árbol de función.

Ejemplo

Suponer que se han definido las funciones *ident* y *suc* cuyos tipos son respectivamente *a' -> a'* y *int -> int*. Al analizar la expresión *suc(ident(5))* se procede de la siguiente manera:

Como la función *suc* está definida, se empila su árbol de tipo y se analiza su argumento (*ident(5)*). Como *ident* es una función, también se empila su árbol de tipo. Cuando se lee 5, que es el argumento de *ident*, se crea un árbol de expresión, que en este caso consta de un sólo nodo. El estado de la *pila* en este momento se muestra en la parte A de la Figura 3.4. A la izquierda de la *pila* se muestran los árboles de función y a la derecha se muestra la PV actual con el árbol de expresión.

Como ya se tiene el argumento de *ident*, se unifica el árbol de expresión con el subárbol izquierdo del árbol de función de *ident*. Esta unificación da como resultado la sustitución de *a'* con *int*. Después de la unificación, se saca lo que está en la cima de la *pila* y el subárbol derecho del árbol de función de *ident* se empila en la PV actual.

[4]Empilar un árbol de tipo significa hacer que el apuntador *sim* que está en la cima de la *pila* apunte a este árbol de tipo.

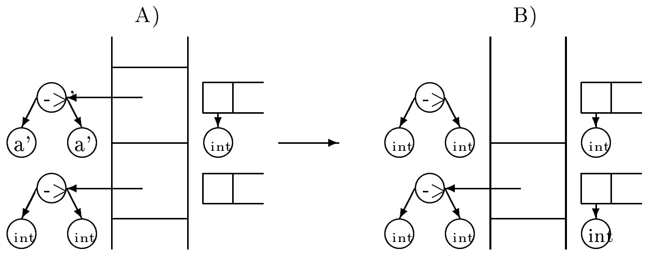

Figura 3.4: Unificación y devolución de tipos.

El nuevo estado de la *pila* se muestra en la parte B de la Figura 3.4. En la parte superior se muestran los árboles de función y de expresión después de haberlos unificados y de sacarlos de la *pila*. En la PV actual se ha empilado el tipo *int* que devuelve la función *ident*. $\diamond$

Cuando se termina de leer el cuerpo de una función, la información necesaria para inferir el tipo de la función está en la primera entrada de la *pila*, ya que *sim* apunta al árbol de tipo de la función[5] y en la PV se ha construido un árbol de expresión que representa el tipo que debe de regresar la función.

Se unifica el árbol de expresión con el subárbol derecho del árbol de función y si la unificación no falla, en el árbol de función queda representado el tipo de la función.

En esta sección se explicó como inferir el tipo de una función. Inferir el tipo de una variable, es básicamente lo mismo, ya que se trata a la variable como una función sin argumentos. Cuando se define una función por casos, se analiza cada caso como si tratara de funciones diferentes y al final se comprueba que todas tengan el mismo tipo.

3.4 Intérprete

Para realizar la interpretación de una expresión hacemos las siguientes consideraciones.

La interpretación de una expresión es muy semejante a la comprobación de tipos. Los valores se representan por medio de árboles (árboles de valor). En sus nodos interiores se colocan los constructores de tipo y en sus nodos hojas se colocan valores.

La construcción de un árbol de valor se hace de la siguiente manera.

- Sean $expr_1$ y $expr_2$ dos expresiones.

[5]Recordemos que cuando se lee el encabezado de una función se construye y se empila el árbol de tipo de la función.

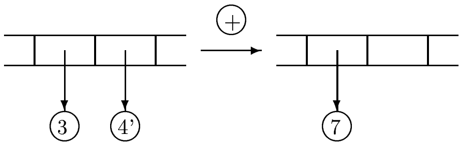

Figura 3.5: Operación binaria.

- Agrupar dos valores, significa crear un nodo y hacer que los árboles que representan a los dos valores sean subárboles del nodo creado.
- Si se lee $expr_1\, op\, expr_2$ entonces se introducen los valores[6] de $expr_1$ y $expr_2$ consecutivamente en la PV actual y se agrupan mediante un constructor, el cual depende de op. Si $op =','$ se utiliza *, si $op =' \; '$ se utiliza $->$ y si $op =' ::'$ se utiliza *list*. Si op es un operador binario, entonces más que agrupación se hace una reducción, ya que se realiza la operación que indica op, el resultado se coloca en el nodo correspondiente al primer operando y se deshecha el otro nodo. Ver Figura 3.5.

Se utiliza un arreglo llamado *ambiente*. Los elementos de este arreglo son apuntadores, los cuales apuntan a listas que contienen los valores de los argumentos de las funciones que se van invocando.

Para manejar el *ambiente* se hace uso de dos variables enteras: *p_amb* y *n_amb*. La variable *p_amb* se incrementa cada vez que se encuentra el nombre de una función e indica la posición del arreglo que apuntará a la lista de valores de los parámetros de esa función y *ambiente[n_amb]* apunta a la lista de donde se toman los valores de los argumentos de la función que se está interpretando.

Tomando en cuenta las consideraciones anteriores, a continuación se explica como se interpreta una expresión.

Primero se comprueba si las funciones que intervienen en la expresión son llamadas con los parámetros de tipo correcto; si es así, se inicia la interpretación.

Si se encuentra una expresión *f argumentos* entonces se empila el árbol de tipo de *f*. Se leen los argumentos de f. Cuando se han leído todos sus argumentos y sus valores se han agrupado en la PV actual para formar un árbol de valor (árbol de valor actual), se guarda el punto de regreso, se guarda el valor de *n_amb*, se hace la asignación *n_amb=p_amb* y se hace un salto a la parte del código en donde está definida la función *f*.

El paso de parámetros y la asignación de valores consta de los siguientes pasos:

1. Al interpretar el encabezado de una función se crea una lista ligada llamada *lista_de_parámetros*. Cada nodo de esta lista representa un parámetro de la función. Esta lista es referida por *ambiente[n_amb]*.

[6]Aunque un valor y un tipo están en el mismo nodo y al empilar un valor también estamos empilando un tipo, mencionamos *valor* durante la interpretación y *tipo* durante la inferencia de tipos.

2. Por cada parámetro de la función se crea una variable de tipo. Estas variables de tipo se agrupan conforme a las reglas que ya se han visto, formando un árbol de tipo al que le llamaremos *arbol_de_asignación*.

3. Cuando se terminan de leer los parámetros, se recorren juntos en pre-orden el árbol de asignación y el árbol de valor actual, es decir, si se visita el subárbol izquierdo del árbol de asignación, también debe visitarse el subárbol izquierdo del árbol de valor actual.

   Cuando se encuentra el i-ésimo nodo en el árbol de asignación que tiene una variable de tipo, el valor del nodo que se está visitando del árbol de valor actual se le asigna al i-ésimo nodo de la lista de parámetros.

   Si se visita el i-ésimo nodo en el árbol de asignación y tiene una constante, el nodo que se está visitando del árbol de valor actual debe tener también una constante y las dos constantes deben ser iguales.

Ejemplo
En este ejemplo se muestra el paso de parámetros y la asignación de valores. Sea la función

#fun head hd::tl = hd;
val head : a' list −> a'

Para interpretar la expresión

head [1];

Primero se hace la comprobación de tipos. Como *head* espera una lista y su argumento es *[1]*entonces no hay error de tipos y se interpreta la expresión.

Se empila el árbol de tipo de *head*. En la PV actual se construye el árbol de valor correspondiente a la lista *[1]*. La PV actual con su correspondiente árbol de valor se muestra en la Figura 3.6.A.

Ahora se guarda la posición de regreso y se salta a la parte del código donde está definida *head*. Al interpretar el encabezado de *head* se crea la lista de parámetros que contiene dos elementos, uno por cada parámetro. Esta lista se muestra en la Figura 3.6.B. También por cada parámetro se crea una variable de tipo. Como los parámetros están separados por *::* las variables de tipo se agrupan mediante el constructor *list* formando el árbol de asignación que se muestra en la Figura 3.6.C.

Recorriendo en pre-orden los arboles de asignación y de valor y siguiendo las reglas mencionadas anteriormente se asocian valores a los nodos en la lista de parámetros. En la Figura 3.6.D se muestra la lista de parámetros con sus respectivos valores. ⋄

Una vez realizado el paso de parámetros se interpreta el cuerpo de la función, el cual es una expresión; así que se procede como arriba, con la única diferencia de que no se hace ninguna comprobación de tipos.

Una PV cumple dos propósitos: agrupa los argumentos de una función y los valores que regresará esa función.

Cuando se termina de interpretar una expresión, se realiza lo siguiente:

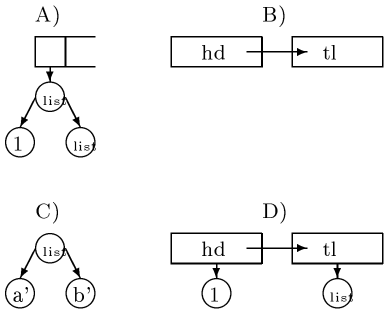

Figura 3.6: Paso de parámetros y asignación de valores.

1. Se destruye la lista de parámetros a la cual apunta *ambiente[n_amb]*. Se decrementa *p_amb*.

2. Si la *pila* tiene más de una entrada ocupada:

   (a) El valor que está en la PV actual (valor devuelto por la función) se empila en la PV que se encuentra una entrada abajo en la *pila* y se saca lo que está en la cima de la *pila*.

   (b) Se restaura el valor de *n_amb*.

   (c) Se salta al punto de regreso.

3. Si la *pila* sólo tiene la primera entrada ocupada, el valor que está en la PV actual es el valor de la expresión dada inicialmente.

Cuando en una expresión se involucra una variable, en lugar de buscar en la lista de parámetros se busca en una tabla donde estén definidas las variables, se empila su valor y se procede como si fuera un parámetro.

La asignación de valores no siempre tiene éxito. Por ejemplo, si se tiene la función:

fun suma 0 x = x;

y se da la expresión

suma 1 x;

la asignación de valores fallará, porque aunque se invocó a *suma* con argumentos de tipo adecuados, la constante con la que se invoca no es la que se espera.

Cuando se interpreta una expresión y falla alguna asignación de valores, se dice que la expresión falla.

Cuando una expresión involucra una función que está definida por casos, se trata de hacer la asignación de valores con el primer caso; si falla se trata de hacer la asignación con el segundo caso y así sucesivamente. Si en todos los casos falla la asignación de valores, entonces la expresión falla.

3.5 Estructuras de datos

Las estructuras de datos y variables que se utilizan son las siguientes:

```
typedef struct tipos{
        char name[20];
        char marca;
        struct tipos *post;
} TIPOS;

TIPOS *tabla_tip[DISPERT];
```

En el campo *name* se guarda el nombre de un tipo; *marca* nos indica si el tipo es básico (int, string), es un constructor de tipos (−> , *, *list*), es un tipo definido por el usuario o es una variable de tipo. Se utiliza una tabla de dispersión [AhA+90] para guardar los tipos, *tabla_tip* es el arreglo de apuntadores a entradas de tabla, *post* apunta al siguiente tipo que tenga la misma dispersión.

```
union val{
        int entero;
        char *cad;
};

typedef struct nodo{
        TIPOS *op;
        struct nodo *izq,*der;
        union val valor;
} NODO;
```

Mediante esta estructura se crean los árboles binarios que representan tipos y valores; *op* apunta a un tipo; *izq* y *der* apuntan a los subárboles izquierdo y derecho respectivamente y en *valor* se guarda un entero o un apuntador a cadena. Este último campo sólo importa cuando *op* apunta a un tipo básico y se está interpretando una expresión.

```
typedef struct simbolo{
        char *nombre;
        char tipo;
        char *loc;
        NODO *raiz;
        struct simbolo *sig;
} SIMBOLO;

SIMBOLO *tabla_sim[DISPERS];
```

En esta estructura se guarda información acerca de un identificador *id*. El campo *nombre* apunta al nombre de *id*; *tipo* nos indica que representa *id* (función, variable, parámetro, etc.). Si *id* representa el nombre de una función o de una variable, *loc* apunta a la localidad en memoria donde está su definición y *raiz* apunta al árbol binario que representa su tipo o valor. Se utiliza una tabla de dispersión para guardar los *símbolos*; *tabla_sim* es el arreglo de apuntadores a entradas de tabla; *sig* apunta al siguiente *simbolo* que tenga la misma dispersión, la cual se calcula en base al *nombre*.

*SIMBOLO *ambiente[TAMAMB];*

Este es un arreglo de apuntadores, los cuales apuntan a listas que contienen los valores de los argumentos de las funciones que se invocan durante la interpretación de una expresión.

```
struct anidar{
        SIMBOLO *sim;
        NODO *PV[TAM_PV];
        int tope;
};
```

struct anidar pila[TAMPILA];

El arreglo *pila* es la estructura de datos que implanta la pila sobre la que realizamos la comprobación e inferencia de tipos y la interpretación de una expresión.

Cada entrada de *pila* tiene tres campos: *sim*, *PV* y *tope*. Los dos primeros campos se han explicado y utilizado a lo largo de este capítulo. El campo adicional *tope* se utiliza como índice para indicar el tope de la *PV*.

3.6 Recolector de basura

Un recolector de basura nos sirve para poder recuperar espacios de memoria que ocupaban objetos que ahora son inaccedibles. Como en el intérprete continuamente se crean y se destruyen árboles y listas, es indispensable contar con un recolector, para que cuando se destruyan objetos, se puedan recuperar esos espacios de memoria. Para hacer el recolector de basura se utilizó una versión modificada del Algoritmo de Baker [LiH+83].

Baker propone dividir la memoria en dos regiones, *fromspace* y *tospace* (Ver Figura 3.7.A). Los objetos son creados en localidades sucesivas de memoria en tospace. El proceso de recolección de basura encuentra los objetos accesibles y los evacúa moviéndolos de fromspace a tospace.

Cuando un objeto es evacuado de fromspace a tospace un *apuntador hacia adelante* se deja en la celda de memoria de fromspace apuntando a su nueva localidad en tospace (Ver Figura 3.7.B).

Cuando un nodo en fromspace que contiene un apuntador hacia adelante es referido, se sigue la liga hacia tospace y se regresa el objeto. Además se altera la referencia original para apuntar hacia el objeto que ahora está en tospace.

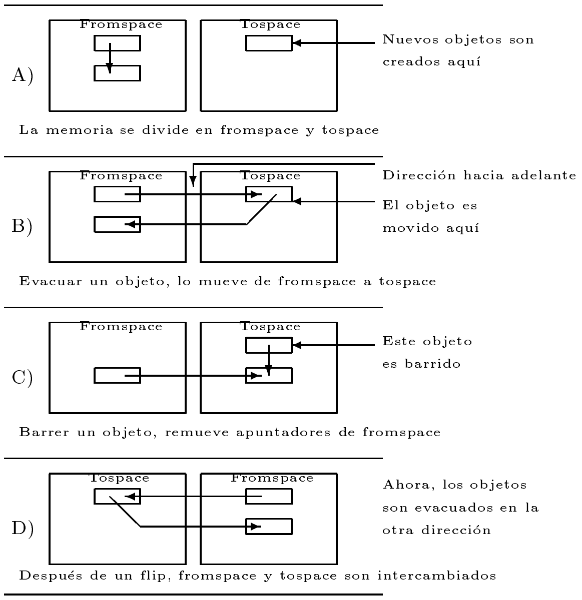

Figura 3.7: Recolector de basura de Baker.

Cuando se acceden componentes de un objeto, se verifica que estén en tospace. El objeto localizado en fromspace se evacúa a tospace y se actualiza la referencia.

Cuando se evacúa un objeto a tospace, uno de sus componentes puede apuntar a fromspace. Lo que se desea es remover todos los apuntadores de fromspace para que esa memoria pueda ser reusada; así que, siempre que se encuentre un apuntador de tospace a fromspace, se remueve el apuntador evacuando el objeto de fromspace a tospace y actualizando el apuntador a la nueva posición en tospace. A este proceso se le llama *barrer* (Ver Figura 3.7.C).

Tospace se divide en dos áreas, el área de creación donde se colocan los objetos que son creados, y el área de evacuación, la cual contiene los objetos evacuados de fromspace.

El proceso *barrer* recorre linealmente el área de evacuación de fromspace; si un componente de un objeto apunta a fromspace, el objeto en fromspace es evacuado a tospace (apendizado al área de evacuación). Cuando ya no hay objetos accedibles en fromspace, esa memoria puede ser reusada. Una operación llamada *flip* ocurre cuando tospace se convierte en fromspace y viceversa (Ver Figura 3.7.D).

Se desea que el recolector actúe sobre objetos de tipo NODO y SIMBOLO, pero para implantar el recolector de basura, es conveniente que todos los elementos que se vayan a recolectar, sean del mismo tipo; por lo tanto las estructuras *nodo* y *simbolo* deben de fusionarse. La estructura que resulta es:

```
union val{
        int entero;
        char *cad;
};

typedef struct nodo{
         TIPOS *op;
         struct nodo *fwp;
         struct nodo *izq,*der;
         union val valor;
         char tipo;
} NODO;
```

Ahora sólo tenemos una estructura, pero al momento de programar, se trata como si en realidad tuvieramos dos estructuras. Esto se hace con ayuda de las siguientes instrucciones para el preprocesador.

```
typedef NODO SIMBOLO
#define raiz izq
#define sig der
#define nombre valor.cad
```

Así se simula que existe el tipo SIMBOLO y los campos *raiz* y *sig*.

El campo adicional *fwp* sirve como apuntador hacia adelante en el proceso de recolección de basura.

Capítulo 4

La lógica de alto orden en ML-HOL

Una vez que se ha implantado el intérprete de ML, éste se extiende para que pueda manejar los símbolos y construcciones de HOL y se monta el demostrador de teoremas sobre el intérprete; por lo tanto, la manera de inferir, comprobar tipos y evaluar expresiones que involucren construcciones lógicas, será la misma que la descrita en el capítulo anterior. Para la inferencia de tipos se agregan algunas reglas que se muestran más adelante.

En este capítulo se describe la representación concreta de HOL. Se inicia mostrando las extensiones realizadas a la gramática del intérprete y después, se describe la representación de tipos y términos lógicos (Sección 4.2 y 4.3), la representación de teorías (Sección 4.4) y los métodos para extender teorías.

También se describe el proceso de demostración hacia adelante (Sección 4.5), incluyendo como se obtienen las reglas de inferencia derivadas y el proceso de demostración orientado a metas (Sección 4.6).

En el sistema ML-HOL se proporcionan los tipos *type* y *term* para representar los tipos y términos de HOL, como se definieron en las Secciones 2.2.1 y 2.2.2, respectivamente. También se proporcionan funciones para crear y manipular valores de estos tipos. Los teoremas se representan por el tipo *thm* [HolD91].

Cuando se inicia el sistema ML-HOL, una serie de teoremas útiles son preprobados y un conjunto de herramientas son predefinidas. Así, el sistema básico ofrece un medio ambiente inicial rico; los usuarios pueden enriquecerlo más implantando sus propias herramientas y extendiendo las teorías.

4.1 Extensiones a la gramática de ML

En está sección se muestran y analizan las reglas gramaticales y tokens que se adicionan a la gramática de la Sección 3.2 para obtener el analizador sintáctico del demostrador de teoremas. Se muestran las reglas de asociatividad para aclarar la precedencia de los operadores.

```
%token CONST_LOG NEG SIM_TEOR

%left ';'
%left AND
%left '|'
%left ',' '¿'
%nonassoc SIM_TEOR
%left '.'
%left '&' OR CONST_LOG
%right NEG
%right '!' '?' '\\'
%left LAMBDA
%nonassoc '<' '>' LE GE
%nonassoc '='
%nonassoc CONS '@'
%left TIPO_FUN
%left '+' '-'
%left '*' '/' MOD '#'
%nonassoc ':'
%%

termino        : '\" subterm '\"
pat1           : pat2 SIM_TEOR pat2
               | pat2 CONST_LOG pat2
               | cuant IDENTIFICADOR '.' pat2
cuant          : '\\'   | '!'    | '?'
expr2          : type
               | expr2 op7 expr2          %prec SIM_TEOR
               | op7 expr2                %prec CONST_LOG
op7            : SIM_TEOR
expr3          : termino
sec_term       : subterm2     | sec_term subterm2
subterm        : subterm2
               | IDENTIFICADOR sec_term
               | '(' termlambda ')' subterm     %prec '\\'
               | subterm CONST_LOG subterm
               | '~' subterm
               | termlambda
termlambda     | paramterm '.' subterm
subterm2       : IDENTIFICADOR        | ENTERO
               | '(' subterm ')'
type           : ':' types
types          : IDENTIFICADOR vartype
               | types '#' types
               | types TIPO_FUN types
vartype        :      | '\"
```

paramterm : *cuant IDENTIFICADOR*
| *paramterm '.' paramterm*
%%

El token CONST_LOG representa a cualquier constante lógica infija y el token SIM_TEOR representa al símbolo $|-$, el cual se usa como una aproximación ASCII de $\vdash$ cuando se imprimen teoremas.

Según esta gramática se pueden introducir incosistencias, ya que la regla

teorema : hipotesis SIM_TEOR termino

indica que el usuario puede construir teoremas arbitrariamente, escribiendo expresiones de la forma $['t_1','t_2',\ldots,'t_n']\ \ |-'t'$

Para evitar inconsistencias, el sistema ML-HOL trabaja en dos modos: *modo inicio* y *modo prueba*. El modo inicio se utiliza cuando se inicia el sistema y son cargadas todas las funciones predefinidas (funciones de uso general; teorías básicas; reglas de inferencia primitivas y derivadas; tácticas y estrategías). En este modo se pueden utilizar construcciones de la forma anterior, es decir, se pueden construir teoremas en forma directa, lo cual hace más fácil la implantación de las reglas de inferencia (primitivas y derivadas), pero al mismo tiempo se corre el riesgo de introducir teoremas inconsistentes, por lo que fuimos muy cuidadosos al programar las reglas de inferencia.

Una vez cargadas todas las funciones predefinidas inmediatamente se hace un cambio a modo prueba. En este modo no se pueden construir directamente los teoremas. La única manera de construir teoremas es aplicar reglas de inferencia a axiomas o teoremas existentes y por lo tanto, se preserva la consistencia de la lógica.

En la Sección 3.3 se dieron las reglas de inferencia y comprobación de tipos para las construciones de SML. A esas reglas se le deben de adicionar otras reglas para las construcciones de HOL:

1. t_1 y t_2 tienen tipo *term* si y solo si t_1 CONST_LOG t_2 tiene tipo *term*.
2. Si $t, t_1, t_2, \ldots, t_n$ tienen tipo *term*, entonces $[t_1, t_2, \ldots, t_n]\ \vdash\ t$ tiene tipo *thm*

4.2 Tipos

Los tipos que se pueden construir dependen de las constantes de tipo que hayan sido declaradas en la teoría actual [HolD91]. Hay dos funciones constructoras para valores de tipo *type*:

mk_vartype : string $->$ type
*mk_type : (string * type list) $->$ type*

La función *mk_vartype* construye una variable de tipo con un nombre dado.

La función *mk_type* construye un tipo compuesto a partir de una cadena representando el nombre del operador de tipo y una lista de tipos representando los argumentos del operador.

La evaluación de *mk_type("nombre",* $[\sigma_1, \ldots, \sigma_n]$) falla si:

1. *nombre* no es un identificador.
2. *nombre* no es un operador de tipo de la teoría actual.
3. *nombre* es un operador de tipo en la teoría actual, pero su aridad no es n.

Además de las funciones constructoras, se pueden formar tipos utilizando apóstrofos y dos puntos ':'. Por ejemplo, el tipo representando $num \rightarrow num \rightarrow num$ se puede escribir como ': num $->$ num $->$ num'.

4.3 Términos

Las cuatro clases de términos de HOL se describen en la Sección 2.2.2. Las funciones ML para manipular a éstos se describen en esta sección.

Los términos que se pueden construir dependen de las constantes que hayan sido declaradas en la teoría actual [HolD91]. Hay cuatro funciones constructoras para valores de tipo *term*:

*mk_var : (string * type) $->$ term*

mk_var(x, σ) evalúa a una variable con nombre x y tipo σ.

*mk_const : (string * type) $->$ term*

mk_const(c, σ) evalúa a un término representando la constante con nombre c y tipo σ.

*mk_comb : (term * term) $->$ term*

mk_comb(t_1, t_2) evalúa a un término representando la combinación $t_1\, t_2$. Falla si el tipo de t_1 no tiene la forma $\sigma_2 - > \sigma_1$, o el tipo de t_1 tiene la forma $\sigma_2 - > \sigma_1$, pero el tipo de t_2 no es igual a σ_2

*mk_abs : (term * term) $->$ term*

mk_abs(x, t) evalúa a un término representando la abstracción $\lambda x.\, t$; falla si x no es una variable.

Las variables y las constantes tienen un nombre y un tipo; la diferencia es que las constantes no se pueden cuantificar y su tipo se establece cuando se declaran. El comprobador de tipos usa los tipos de las constantes para inferir los tipos de variables en un término. Si no hay suficiente información de tipos se produce un error.

#val t1 = '$\sim$ x';
val t1 = '$\sim$ x' : term

#val t2 = 'x';
tipo indeterminado

En el primer caso, el comprobador de tipos sabe que el tipo de $\sim$ es bool $->$ bool y por lo tanto, deduce que la variable x debe tener tipo bool. En el segundo caso, no puede deducir el tipo de x.

Al igual que con los tipos, además de las funciones constructoras, los términos pueden formarse utilizando apóstrofos [HolT91]; la expresión $'x/\backslash\ y ==> z'$ representa al término lógico $x \wedge y \Rightarrow z$. Los términos se pueden manipular por varias funciones ML predefinidas (ver Sección A.2). Por ejemplo, la función *dest_imp* con tipo *term $->$ (term * term)*, divide una implicación en un par de terminos consistentes de su antecedente y consecuente, y la función *dest_conj* de tipo *term $->$ (term * term)* divide una conjunción en sus dos componentes.

```
#val t3 = 'x /\ y ==> z';
val t3 = 'x /\ y ==> z' : term

#val t4 = dest_imp t3;
val t4 = ('x /\ y', 'z') : (term * term)

#val t5 = dest_conj(fst t4);
val th5 = ('x', 'y') : (term * term)
```

En la siguiente tabla se resume la notación que se usa para los términos de HOL.

Tipo de término	Notación estándar	Notación HOL	Descripción
Verdad	$\top$	T	verdadero
Falsedad	$\bot$	F	falso
Negación	$\neg t$	$\sim t$	not t
Disyunción	$t_1 \vee t_2$	$t_1 \backslash / t_2$	t_1 o t_2
Conjunción	$t_1 \wedge t_2$	$t_1 / \backslash t_2$	t_1 y t_2
Implicación	$t_1 \Rightarrow t_2$	$t_1 ==> t_2$	t_1 implica t_2
Cuantificación $\forall$	$\forall x.t$	$!x.t$	para todo $x : t$
Cuantificación $\exists$	$\exists x.t$	$?x.t$	para algún $x : t$
Término e	$e\,x.t$	$@x.t$	un x tal que : t
Condicional	$(t \rightarrow t_1, t_2)$	$(t => t_1 \mid t_2)$	if t then t_1 else t_2

Tabla 4.1: Términos de HOL.

4.4 Teorías

El resultado de una sesión con el sistema ML-HOL es un objeto llamado *Teoría*. Una teoría contiene conjuntos de tipos, constantes, definiciones y axiomas; contiene además una lista de teoremas que han sido previamente probados [HolT91]. Las teorías son potencialmente extensibles y para esto se proporcionan herramientas.

El propósito del sistema ML-HOL es proporcionar herramientas para construir teorías bien formadas. Todos los teoremas de tales teorías son consecuencias lógicas de las definiciones y axiomas de la teoría. El sistema se encarga de asegurar que sólo las teorías bien formadas puedan ser construidas permitiendo la creación de teoremas solamente mediante una *prueba formal*.

Una teoría se representa como una colección de archivos llamados *archivos teoría*. Cada archivo tiene un nombre de la forma *nombre.th*, donde *nombre* es una cadena dada por el usuario.

Una pieza de trabajo con el sistema ML-HOL puede consistir de un número de sesiones. En la primera de éstas se crea una nueva teoría, digamos T. El resultado concreto de la sesión será un archivo teoría T.th cuyo contenido se crea durante la sesión. En sesiones subsecuentes dicha teoría se puede extender probando nuevos teoremas, los cuales serán guardados en el archivo T.th. El significado lógico de estas sesiones es que se crea una nueva extensión a T, la cual reemplaza la vieja versión.

4.4.1 Funciones ML para la creación de teorías

Las funciones ML para la creación de teorías son [HolD91]:

1. *new_theory : string −> void*

   Al ejecutar *new_theory "nombre"*, se crea una nueva teoría llamada *nombre*; falla si ya existe un archivo *nombre.th* en el directorio de trabajo.

2. *new_type : (int * string) −> void*

   Al ejecutar *new_type(n,"op")* hace de *op* un nuevo operador de tipo de aridad n en la teoría actual. Si $n = 0$ entonces *op* es un operador de tipo sin argumentos, es decir, un tipo constante. Falla si existe un operador de tipo llamado *op* en la teoría actual.

3. *new_constant : (string * type) −> void*

   Evaluando *new_constant("c",* σ) hace de c una nueva constante de tipo σ en la teoría actual. Ésto falla si ya existe una constante c en la teoría actual.

4. *new_infix : (string * type) −> void*

   Ejecutando *new_infix("ix",*σ) declara *ix* como una nueva constante de tipo σ y estado infijo. Falla si ya existe una constante c en la teoría actual y si σ no es de la forma $\sigma_1 -> \sigma_2 -> \sigma_3$.

5. *new_axiom : (string * term) −> thm*

   Ejecutando *new_axiom("nombre",t)* declara que el secuente ({ },t) será un axioma de la teoría actual; el axioma será llamado *nombre*. Falla si ya existe un axioma, definición o especificación llamada *nombre* en la teoría actual.

6. *save_thm : (string * thm) −> thm*

   Una vez que un teorema ha sido probado, se puede salvar. Evaluando *save_thm("nombre",th)* salvará el teorema *th* con nombre *nombre* en la teoría actual.

7. *close_theory : void −> void*

   La función anterior se utiliza para finalizar una sesión y salvar en un archivo teoría los cambios que se hayan realizado.

8. *extended_theory : string −> void*

   Ejecutando *extended_theory "nombre"* carga la teoría *nombre*. Falla si no existe un archivo *nombre.th*

4.5 Proceso de demostración hacia adelante

Una *prueba hacia adelante* en el sistema ML-HOL se construye aplicando reglas de inferencia a axiomas y teoremas existentes para obtener nuevos teoremas y eventualmente obtener el teorema deseado. Dado que las pruebas pueden consistir de cientos o miles de pasos, se necesitan proporcionar herramientas para hacer la construcción de pruebas más fácil para el usuario [HolT91].

Utilizaremos tres reglas de inferencia primitivas (*ASSUME, DISCH* y *MP*) para ilustrar pruebas hacia adelante.

La regla de inferencia *ASSUME* genera teoremas de la forma $t \mid\!- t$.

#val Th1 = ASSUME 't1 ==> t2';
val Th1 = ['t1==>t2'] |− 't1 ==> t2' : thm

La función *MP* toma como argumentos un par de teoremas de la forma $\Gamma_1 \mid\!- t_1 ==> t_2$ y $\Gamma_2 \mid\!- t_1$, y regresa $\Gamma_1 \cup \Gamma_2 \mid\!- t_2$.

#val Th2 = ASSUME 't1:bool';
val Th2 = ['t1'] |− 't1' : thm

#val Th3 = MP Th1 Th2;
val Th3 = ['t1 ==> t2','t1'] |− 't2' : thm

La regla de inferencia *DISCH* infiere a partir de un teorema $\Gamma, t_1 \mid\!- t_2$ el nuevo teorema $\Gamma \mid\!- t_1 ==> t_2$. *DISCH* toma como argumento el término a ser descargado (es decir t_1) y el teorema desde el que la premisa será descargada y regresa un teorema con el resultado del descargo. Descargar dos veces *Th3* establece el teorema |− t1 ==> (t1 ==> t2) ==>t2

#val Th4 = DISCH 't1 ==> t2' Th3;
val TH4 = ['t1'] |− '(t1 ==> t2) ==> t2' : thm

#val Th5 = DISCH 't1:bool' Th4;
val Th5 = |− 't1 ==> (t1 ==> t2) ==> t2' : thm

La secuencia Th1, Th2, Th3, Th4, Th5 constituye una prueba en ML-HOL del teorema $\mid\!- t1 ==> (t1 ==> t2) ==> t2$.

En notación lógica estándar la prueba anterior se puede escribir:

1. $t_1 \Rightarrow t_2 \ \vdash \ t_1 \Rightarrow t_2$ [Introducción de premisa]
2. $t_1 \ \vdash \ t_1$ [Introducción de premisa]
3. $t_1 \Rightarrow t_2, t_1 \ \vdash \ t_2$ [Modus Ponens aplicado a las líneas 1 y 2]
4. $t_1 \ \vdash \ (t_1 \Rightarrow t_2) \Rightarrow t_2$ [Descargando la 1ra. premisa de línea 3]
5. $\vdash \ t_1 \Rightarrow (t_1 \Rightarrow t_2) \Rightarrow t_2$ [Descargando la única premisa de línea 4]

4.5.1 Reglas de inferencia derivadas

Una prueba a partir del conjunto de hipótesis $th_1, \ldots, th_n$ es una secuencia, donde cada uno de sus elementos es un axioma, una de las hipotesis th_i o se sigue de uno de sus elementos por una regla de inferencia [HolT91].

Por ejemplo, una prueba de $\Gamma, t' \ \vdash \ t$ a partir de la hipótesis $\Gamma \ \vdash \ t$ es:

1. $t' \ \vdash \ t'$ [Introducción de premisa]
2. $\Gamma \ \vdash \ t$ [Hipótesis]
3. $\Gamma \ \vdash \ t' \Rightarrow t$ [Descarga t' desde la línea 2]
4. $\Gamma, t' \ \vdash \ t$ [Modus Ponens aplicado a las líneas 3 y 1]

Ahora, cada vez que tengamos un teorema $\Gamma \ \vdash \ t$ y se requiera un teorema de la forma $\Gamma, t' \ \vdash \ t$, su prueba se debe construir nuevamente. Para capturar el patrón general de inferencia, se puede escribir una función ML para implantar una regla de inferencia a partir de las reglas de inferencia primitivas. Abstractamente, una *regla de inferencia derivada* es una regla que se puede justificar tomando como base las reglas de inferencia primitivas (y/o los axiomas) [HolD91]. En el presente caso la regla requiere adicionar una premisa. Se especifica en HOL como

$$\frac{\Gamma \ \vdash \ t}{\Gamma, t' \ \vdash \ t}$$

La regla se puede implantar en ML como una función (*ADD_ASSUM*) que llama la secuencia apropiada de reglas de inferencia primitivas. La función es:

```
fun ADD_ASSUM t th = let val th1 = ASSUME t in
                         let val th2 = DISCH t th in
                             MP th2 th1
                         end
                     end;
```

Esto proporciona una función que mapea un teorema a un teorema; esto es, ejecuta pruebas en HOL. Podemos decir que una *regla de inferencia derivada* es un procedimiento en ML que genera una prueba a partir de las hipótesis dadas, cada vez que se invoca. Las hipótesis son los argumentos de la regla. A las pruebas generadas mediantes reglas de inferencia, el usuario no puede darles una orientación

El cuerpo de *ADD_ASSUM* se codificó para reflejar la prueba hecha arriba, pero *ADD_ASSUM* se puede escribir más concisamente como:

fun ADD_ASUM t th = MP (DISCH t th) (ASSUME t);

Otro ejemplo de una regla de inferencia derivada es *UNDISCH*; la cual mueve el antecedente de una implicación a las premisas.

$$\frac{\Gamma \ \vdash \ t_1 \Rightarrow t_2}{\Gamma, t_1 \ \vdash \ t_2}$$

Una función ML que implanta esto es:

fun UNDISCH th = MP th (ASSUME(fst(dest_imp(concl th))));[1]

Cada vez que se ejecuta UNDISCH $\Gamma \ \vdash \ t_1 \Rightarrow t_2$, se desarrolla la siguiente prueba:

1. $t_1 \ \vdash \ t_1$	[Introducción de premisa]
2. $\Gamma \ \vdash \ t_1 \Rightarrow t_2$	[Hipótesis]
3. $\Gamma, t_1 \ \vdash \ t_2$	[Modus Ponens aplicado a las líneas 2 y 1]

Los ejemplos anteriores son muy simples, pero una regla de inferencia derivada puede ejecutar pruebas de longitud arbitraria. También puede hacer uso de reglas definidas previamente. De esta manera, los patrones de inferencia se pueden desarrollar más fácil y rápidamente.

Un número de reglas de inferencia derivadas son predefinidas cuando se inicia el sistema ML-HOL. En la Sección A.4 se definen las funciones ML que implantan las reglas que reflejan los patrones de inferencia más usuales del cálculo de predicados.

4.6 Proceso de demostración orientado a metas

El estilo de pruebas hacia adelante descrito en la sección previa es insuficiente para muchas aplicaciones. Un avance importante en la metodología de generación de pruebas fue hecha por Robin Milner en los 70's cuando inventó la noción de táctica [HolT91].

Una táctica es una función ML que cuando se aplica a una meta, la reduce a una lista de submetas, junto con una función de *justificación* o *validación* que mapea una lista de teoremas a un teorema. La idea es que la función justifique la descomposición de la meta. La justificación es una regla de inferencia, la cual, cuando se aplica a la lista de teoremas relacionada con las submetas, regresa el teorema que está relacionado con la meta deseada. A la función justificación se le llama prueba [HolD91].

La idea de pruebas dirigidas a metas es simple: se aplican repetidamente tácticas hasta que todas las submetas sean triviales (es decir, sean declaraciones de axiomas o de teoremas previamente probados). La composición de las justificaciones resultantes es una regla de inferencia, la cual probará el teorema deseado [GuE89].

---

[1] Ver sección A.2.1 para entender las funciones implicadas.

Una meta [HolD91] es un par cuyo primer componente es una lista de términos (premisas) y su segundo componente es un término. Sus componentes corresponden respectivamente a la lista de hipótesis y a la conclusión de un teorema.

La relación de teoremas y metas es: un teorema representa a una meta si la conclusión del teorema es igual a la parte del término de la meta y si cada hipótesis del teorema es igual a alguna premisa de la meta.

Si T es una táctica y g es una meta, entonces aplicar T a g (es decir, evaluar la expresión $T\ g$) dará como resultado una submeta, es decir, un par cuya primera componente es una lista de metas y cuya segunda componente es una función justificación.

Algunas abreviaciones de tipos son utilizadas

proof	= thm list −> thm
goal	= term list * term
tactic	= goal −> (goal list * proof)
submeta	= goal list * proof

Suponer que $T\ g = ([g_1, \ldots, g_n], p)$, y que las submetas $g_1, \ldots, g_n$ han sido resueltas. Esto significa que algunos teoremas $th_1, \ldots, th_n$ han sido probados de tal forma que cada th_i $(1 \leq i \leq n)$ representa a la meta g_i. Se intenta que la justificación p sea una función que cuando se aplica a la lista $[th_1, \ldots, th_n]$ tenga éxito y regrese un teorema th, que represente a la meta original g; pero algunas veces podría no tener éxito. Si p tiene éxito para toda lista de teoremas, entonces se dice que la táctica T es *válida.* Esto no garantiza que las submetas sean solubles. Si una táctica es válida y además siempre produce submetas solubles a partir de una meta soluble, se dice que la táctica es *fuertemente válida.*

Las tácticas pueden ser útiles sin ser fuertemente válidas, o aún sin ser válidas; de hecho algunos de los métodos básicos de demostración de teoremas expresados como tácticas, son inválidos o no fuertemente válidos. Una táctica inválida no puede resultar en la prueba de teoremas falsos; los teoremas en ML-HOL son siempre el resultado de ejecutar una prueba sobre bases lógicas, ya sea que la prueba sea hecha hacia adelante u orientada a metas. Lo peor que puede pasar es que se obtenga como resultado submetas que no se puedan resolver o un teorema no deseado (uno que no represente la meta original).

Una táctica *resuelve* una meta si reduce la meta a la lista vacía de submetas. Así T resuelve g si $T\ g = ([\ ], p)$. Si éste es el caso y T es válida, entonces $p[\ ]$ evaluará al teorema que representa a g. Así, si T resuelve g entonces la expresión $snd(T, g)[\ ]$ evalúa al teorema que representa a g.

Las tácticas se especifican usando la siguiente notación:

$$\frac{\mathit{meta}}{\mathit{meta}_1 \quad \mathit{meta}_2 \quad \cdots \quad \mathit{meta}_n}$$

Por ejemplo, la táctica para descomponer conjunciones en dos submetas es llamada $CONJ\_TAC$ y es descrita por

$$\frac{t_1 / \backslash t_2}{t_1 \qquad t_2}$$

Esto indica que *CONJ_TAC* reduce una meta de la forma $(\Gamma, t_1 / \backslash t_2)$ a las submetas (Γ, t_1) y (Γ, t_2). El hecho de que las premisas de la meta original sean propagadas sin cambio a las dos submetas se indica por la ausencia de premisas en la notación. La notación no da indicación de la función de justificación.

Otro ejemplo es *INDUCT_TAC*, la táctica para hacer inducción matemática sobre los números naturales:

$$\frac{!n.\, t[n]}{t[0] \qquad \{t[n]\}\ t[SUC\ n]}$$

INDUCT_TAC reduce una meta de la forma $(\Gamma, !n.\, t[n])$ a las submetas $(\Gamma, t[0])$ y $(\Gamma \cup \{t[n]\}, t[SUC\ n])$. La premisa extra "$t[n]$" se indica en la notación de tácticas mediante llaves. La primera submeta es el caso base y la segunda submeta es el caso de paso de inducción.

Una táctica falla (en el sentido ML) si se aplica a metas inapropiadas. Por ejemplo, *CONJ_TAC* fallará si se aplica a una meta cuya conclusión no es una conjunción. Algunas tácticas nunca fallan; por ejemplo *ALL_TAC*

$$\frac{t}{t}$$

es la táctica identidad; reduce una meta (Γ, t) a la única submeta (Γ, t), es decir, no tiene efecto. *ALL_TAC* es útil para escribir tácticas compuestas.

Un ejemplo simple ilustra el uso de tácticas. Considerar la meta trivial para resolver $T\ / \backslash\ T$, donde T es la constante que representa el valor *true*.

```
#val goal1 = ([ ],'T /\ T');
val goal1 = ([ ],'T /\ T') : a' list * term
```

```
#val goal_list,just_fn = CONJ_TAC goal1;
goal_list = [([ ],'T'),([ ],'T')] : goal list
just_fn = - : proof
```

CONJ_TAC ha producido una lista de metas consistente de dos submetas identicas *([ ],'T')*. Ahora, hay un teorema pre-probado en ML-HOL llamado *TRUTH*, que representa a esta meta.

```
#TRUTH;
|- 'T'
```

Aplicando la función justificación *just_fn* a la lista de teoremas que representan a las metas en *goal_list*, resulta un teorema que representa a la meta original.

```
#just_fn [TRUTH,TRUTH];
|- 'T /\ T'
```

Aunque este ejemplo es muy trivial, ilustra la idea esencial de las tácticas.

El sistema ML-HOL proporciona una colección de tácticas que incluyen CONJ_TAC, INDUCT_TAC y ALL_TAC. Las tácticas predefinidas son adecuadas para muchas aplicaciones y además se pueden definir nuevas tácticas; hay dos medios de hacerlo.

Dado que una táctica es una función ML, el usuario puede definir a una nueva táctica directamente en ML. Definiciones de esta clase usan funciones ML para construir las submetas y especifican la justificación, la cual espera una lista de teoremas que representen a las submetas y regresa el teorema que representa a la meta. La prueba del teorema se codifica en la definición de la función justificación; esto es, el medio para derivar el teorema deseado a partir de los teoremas dados. Esto involucra referencias a axiomas, reglas de inferencia primitivas y derivadas, y es usualmente la parte más difícil de la definición.

Un ejemplo de una táctica escrita en ML es proporcionado por *CONJ_TAC*, cuya definición es como sigue:

```
fun CONJ_TAC (asl:term list,w) =
    let val l,r = dest_conj w in
        ([(asl,l),(asl,r)], fn [th1,th2] => CONJ th1 th2)
    end;
```

La función *dest_conj* divide una conjunción en dos partes. Si, $(asl,'t_1/\backslash t_2')$ es una meta, entonces *CONJ_TAC* la divide en una lista de dos submetas $(asl,'t_1')$ y $(asl,'t_2')$. La función justificación *fn [th1,th2] => CONJ th1 th2* toma una lista $[th_1, th_2]$ de teoremas y aplica la regla *CONJ* a th_1 y th_2.

El segundo método para definir nuevas tácticas es componer tácticas existentes mediante el uso de funciones ML llamadas *estrategias*. Por ejemplo, dos tácticas existentes se pueden aplicar en secuencia mediante el uso de *THEN*: si T_1 y T_2 son tácticas, entonces la expresión *THEN* T_1 T_2 evalúa a una táctica que primero aplica T_1 a una meta y después aplica T_2 a cada submeta producida por T_1.

En general, el segundo método es más fácil y más confiable. Es más fácil porque no involucra escribir procedimientos ML (usualmente muy complicados); y más confiable porque las tácticas compuestas son válidas cuando las tácticas componentes son válidas.

Algunas tácticas y estrategias importantes son predefinidas cuando se inicia el sistema ML-HOL. En la Sección A.5 se muestra una selección de las tácticas más comúnmente usadas. En la Sección A.6 se listan las estrategias que se proporcionan en el sistema ML-HOL.

Capítulo 5

Aplicaciones

En este capítulo mostramos dos aplicaciones. En la primera construimos los postulados de Peano para los números naturales y definiciones para la suma y multiplicación. Este ejemplo muestra como se establece y extiende una teoría. En la segunda aplicación se especifica un verificador de paridad. Esta aplicación ejemplifica el proceso de demostración orientado a metas.

5.1 Postulados de Peano

Los postulados de Peano son:

P1 Existe un número 0.

P2 Existe una función *Suc* llamada función sucesor, tal que si n es un número, entonces *Suc* n es un número.

P3 0 no es el sucesor de algún número.

P4 Si dos números tienen el mismo sucesor, entonces ellos son iguales.

P5 Si una propiedad se cumple para 0 y siempre que se cumple para algún número, también se cumple para el sucesor del número, entonces la propiedad se cumple para todos los números. Este es el postulado de *Inducción Matemática*

Lo primero que hacemos es crear una nueva teoría llamada *Peano*:

#new_theory "Peano";
( ) : void

Para formalizar los postulados de Peano en ML-HOL se introduce un nuevo tipo llamado *nat*:

#new_type (0,"nat");
( ) : void

Con esto, el tipo *nat* es declarado como un operador de tipo de aridad 0, es decir, es un tipo constante.

Los axiomas **P1** y **P2** pueden ser formalizados declarando dos nuevas constantes de tipo *nat* para representar 0 y *Suc*.

```
#new_constant ("0", ': nat');
la constante ya existe
```

El símbolo 0 (número cero) ya es una constante de tipo *num* y por lo tanto no se puede volver a definir como una constante en nuestra teoría. El símbolo O (letra O), puede ser usado en su lugar.

```
#new_constant ("O", ': nat');
( ) : void
```

```
#new_constant ("Suc", ': nat -> nat');
( ) : void
```

El comprobador de tipos de ML-HOL se asegura que **P1** y **P2** se cumplan. **P3** se declara como un axioma:

```
#new_axiom("P3", '!n. ~(O == Suc n)');
|-'!n. ~(O == Suc n)'
```

Esto crea un axioma en la teoría actual (es decir, en Peano) llamado **P3**. Un axioma **P4** se puede declarar similarmente:

```
#new_axiom("P4", '!m. !n. (Suc m == Suc n) ==> (m == n)');
|-'!m. !n. (Suc m == Suc n) ==> (m == n)';
```

El último axioma de Peano es el de Inducción Matemática:

```
#new_axiom("P5",'!P. P O /\ (!n . P n ==> P(Suc n) ==> (!n. P n)');
|-'!P. P O /\ (!n . P n ==> P( Suc n) ==> (!n. P n)'
```

Hay que hacer notar que **P5** no se puede declarar como un axioma en lógica de primer orden, ya que en ella los predicados (es decir P) no pueden ser cuantificados.

Para finalizar la sesión y escribir las declaraciones anteriores en el archivo *Peano.th* se ejecuta *close_theory*.

```
#close_theory;
( ) : void
```

Es usual complementar los postulados de Peano con algunos axiomas que definan la aritmética de los números naturales. Incluiremos en la teoría Peano dos axiomas que definan la adición y la multiplicación. Para hacer esto se debe extender la teoría Peano.

```
#extended_theory "Peano";
Teoria Peano cargada
( ) : void
```

Para agregar los nuevos axiomas, se declaran las constantes *add* y *mult* que representan la adición y la multiplicación. Las constantes se declaran con estado infijo usando *new_infix* en lugar de *new_constant*.

```
#new_infix ("add", ': nat -> nat -> nat');
( ) : void

#new_infix ("mult", ': nat -> nat -> nat');
( ) : void
```

Ahora se pueden crear dos axiomas para definir la adición y la multiplicación:

```
#new_axiom
#  ("add_def",
#    '(!n. 0 add n = n) /\(!m n. (Suc m) add n = Suc(m add n))');
|-(!n. 0 add n = n) /\(!m n. (Suc m) add n = Suc(m add n))

#new_axiom
#  ("mult_def",
#    '(!n. 1 mult n = n) /\(!m n. (Suc m) mult n = (m mult n) add n)');
|-(!n. 1 mult n = n) /\(!m n. (Suc m) mult n = (m mult n) add n)
```

Al ejecutar *close_theory* se finaliza la sesión y se salvan las declaraciones anteriores en el archivo *Peano.th*, con lo cual se extiende la teoría Peano.

5.2 Un verificador de paridad

Un verificador de paridad es un dispositivo que calcula la paridad de una secuencia de bits. Más especificamente, un verificador de paridad secuencial es un dispositivo con una entrada *in* y una salida *out* con la especificación de que la n-ésima salida en *out* es T si y sólo si hay un número par de entradas T en *in*.

Para especificar el dispositivo, se define una función recursiva $PARITY$ tal que para $n > 0$, $PARITY\ n\ f$ es verdadero si el número de *T's* en la secuencia $f(1), \ldots, f(n)$ es par.

```
#new_axiom("PARITY_DEF",
#      '(!f. PARITY 0 f = T) /\
#       (!n !f. PARITY(SUC n) f =
#               (f(SUC n) => ~(PARITY n f) | PARITY n f))');
```

La especificación del dispositivo verificador de paridad es dada como:

```
!t. out t = PARITY t in
```

Esta especificación será satisfecha si las señales[1] *in* y *out* satisfacen las siguientes condiciones:

[1]Las señales son modeladas como funciones de números a booleanos, donde los números representan tiempos

out(0) = T (En el tiempo 0 la salida es T)

!t. out(t+1) = (in(t+1) => ~(out t) | out t) (Si en el tiempo *t+1* la entrada es T entonces la salida cambia de estado, de verdadero a falso o de falso a verdadero)

Esto se verifica formalmente en HOL probando el siguiente teorema:

```
|-!in out.
    (out 0 = T) /\(!t. out(SUC t) = (in(SUC t) => ~(out t) | out t))
    ==> (!t. out t = PARITY t in)
```

La prueba de este teorema se hace mediante Inducción Matemática. Durante la demostración se utilizan las tácticas GEN_TAC, DISCH_TAC, INDUCT_TAC y ASN_REWRITE_TAC. GEN_TAC especializa un teorema cuantificado universalmente a un valor arbitrario, DISCH_TAC mueve el antecedente de una meta implicativa adentro de las premisas, dejando el consecuente como el componente término. Si la meta es $t_1 /\backslash \cdots /\backslash t_n$ ==> t entonces *DISCH_TAC* mueve cada t_i como una premisa separada. INDUCT_TAC sirve para hacer inducción matemática sobre los números naturales. ASN_REWRITE_TAC simplifica una meta reescribiéndola, utilizando para esto una lista de teoremas; en esta lista se incluyen las premisas de la meta. En la Sección A.5 se explican con más detalle estas tácticas.

Como primer paso creamos una nueva teoría llamada *Parity*:

```
#new_theory "Parity";
( ) : void
```

La prueba se inicia declarando la meta que representa al teorema que deseamos demostrar.

```
#val meta =
# ( [ ],
#    '!in out.
#      (out 0 = T) /\(!t. out(SUC t) = (in(SUC t) => ~(out t) | out t))
#      ==> (!t. out t = PARITY t in)');
```

Los cuantificadores universales se eliminan aplicando dos veces la táctica GEN_TAC; después mediante la táctica DISCH_TAC se mueven los elementos de la conjunción del antecedente de la implicación dentro de las premisas de la meta.

```
#val meta1_list,just1_fn = GEN_TAC 'in' meta;
meta1_list =
  [([ ],
    '!out.
      (out 0 = T) /\(!t. out(SUC t) = (in(SUC t) => ~(out t) | out t))
      ==> (!t. out t = PARITY t in)')] : goal list
just1_fn = - : proof
```

```
#val meta2_list,just2_fn = GEN_TAC 'out' head(meta1_list);
meta2_list =
    [([ ],
      '(out 0 = T) /\(!t. out(SUC t) = (in(SUC t) => ~(out t) | out t))
      ==> (!t. out t = PARITY t in)')] : goal list
just2_fn = - : proof

#val meta3_list,just3_fn = DISCH_TAC head(meta2_list);
meta3_list =
    [(['out 0 = T','!t. out(SUC t) = (in(SUC t) => ~(out t) | out t)'],
      '!t. out t = PARITY t in')] : goal list
just3_fn = - : proof
```

Esta parte de la demostración se dividió en tres pasos para hacer más detallado el proceso de demostración, pero los pasos anteriores se pueden resumir en uno sólo ejecutando:

```
#val meta1_list,just1_fn = THEN (REPEAT GEN_TAC) DISCH_TAC;
```

donde REPEAT es una estrategia que aplica repetidamente una táctica (en este caso GEN_TAC) hasta que falla.

El siguiente paso es hacer inducción sobre t utilizando INDUCT_TAC, la cual realiza inducción sobre una variable cuantificada universalmente.

```
#meta4_list,just4_fn = INDUCT_TAC head(meta3_list);
meta4_list =
     [ ( [ 'out 0 = T','!t. out(SUC t) = (in(SUC t) => ~(out t) | out t)'],
         'out 0 = PARITY 0 in' ),
       ( [ 'out 0 = T','!t. out(SUC t) = (in(SUC t) => ~(out t) | out t)',
           '!t. out t = PARITY t in'],
         'out(SUC t) = PARITY(SUC t) in') ] : goal list
just4_fn = - : proof
```

Al aplicar INDUCT_TAC se obtienen dos submetas, una submeta base y la submeta del paso de inducción. Estas metas son resueltas mediante la táctica ASN_REWRITE_DEF para reescribirlas utilizando las premisas y la definición de PARITY.

```
meta5_list,just5_fn=ASN_REWRITE_TAC[PARITY_DEF] head(meta4_list);
meta5_list = [ ] : goal list
just5_fn = - : proof

meta6_list,just6_fn=ASN_REWRITE_TAC[PARITY_DEF]
                        tail(head(meta4_list));
meta6_list = [ ] : goal list
just6_fn = - : proof
```

Como se han producido listas vacías de submetas quiere decir que la meta se ha probado. Ahora se aplican las funciones de justificación producidas por las tácticas a las listas de teoremas que representan las submetas (comenzando con la lista vacía).

```
# val th1 = just5_fn[ ];
val th1 =
   [ 'out 0 = T','!t. out(SUC t) = (in(SUC t) => ~(out t) | out t)' ]
   |-'out 0 = PARITY 0 in'

#val th2 = just6_fn[ ];
val th2 =
   [ 'out 0 = T','!t. out(SUC t) = (in(SUC t) => ~(out t) | out t)',
     '!t. out t = PARITY t in' ]
   |-'out(SUC t) = PARITY(SUC t) in' : thm

#val th3 = just4_fn[th1,th2];
val th3 =
   [ 'out 0 = T','!t. out(SUC t) = (in(SUC t) => ~(out t) | out t)']
   |-'!t. out t = PARITY t in' : thm

#val th4 = just3_fn[th3];
val th4 =
   |-'(out 0 = T) /\(!t. out(SUC t) = (in(SUC t) => ~(out t) | out t))
      ==> (!t. out t = PARITY t in)' : thm

#val th5 = just1_fn[just2_fn[th4] ];
val th5 =
   |-'!in out.
      (out 0 = T) /\(!t. out(SUC t) = (in(SUC t) => ~(out t) | out t))
      ==> (!t. out t = PARITY t in)' : thm
```

El teorema *th5* es el teorema que se deseaba obtener. Ahora se utiliza la función *save_thm* para salvar el teorema *th5* en la teoría *Parity*.

```
save_thm("VER_PARITY",th5);
```

Para finalizar la sesión, se ejecuta *close_theory*.

Conclusiones

La implantación del sistema ML-HOL fue muy complicada. Contra lo que pudiera pensarse, la parte más difícil fue el intérprete de SML, ya que SML tiene un poderoso sistema de inferencia y comprobación de tipos y se ofrecen muchas facilidades al usuario para definir las funciones y evaluar expresiones; y como el demostrador automático de teoremas está basado totalmente en el intérprete, éste tuvo que ser implantado tomando en cuenta que iba a ser extendido y después poder montar en él la herramienta de demostración.

Cuando elegimos a ML como el metalenguaje que nos serviría como medio ambiente para construir las herramientas necesarias para asistir en la construcción de pruebas, debido a que hay muchas variantes de ML, el primer problema que tuvimos fue el de seleccionar la variante adecuada de ML.

En un primer acercamiento, se intentó trabajar con CAML, pero nos enfrentamos con varios problemas, ya que su sintaxis es muy ambigüa. Por ejemplo, el simbolo '.' sirve para construir listas de patrones, listas de expresiones, expresiones lambda y para empatar patrones. Esta ambigüedad de la sintaxis hacía muy difícil la construcción del analizador sintáctico y eso nos hizo cambiar el rumbo hacia el Estándar ML (SML), cuya sintaxis es más clara.

El primer paso de la implantación fue establecer las estructuras de datos sobre las que se implantarían el sistema inferidor de tipos, el comprobador de tipos y el intérprete. Se analizaron y probaron varias estructuras hasta que construimos lo que llamamos *pila de evaluación*, a la cual nos referimos ampliamente en el desarrollo de este trabajo. Creemos que la *pila de evaluación* es una estructura de datos muy sólida, ya que sobre ella se realiza la inferencia de tipos, la comprobación de tipos, el paso de parámetros, la interpretación de expresiones y el regreso de valores. En otras palabras, la *pila de evaluación* es el corazón del intérprete y sobre ella se realizan la mayor parte de las operaciones. Esto tiene la ventaja de que si se quiere extender el intérprete, básicamente hay que trabajar sobre la *pila de evaluación*.

Un problema importante, y que de hecho fue de donde surgió la idea de representar los valores y tipos mediante árboles, fue el paso de parámetros. En algunos intérpretes el paso de parámetros se realiza de la siguiente manera: se leen los argumentos con que se llama a una función, y se van guardando en una cola; cuando se interpreta el encabezado de la función, cada vez que se encuentra una variable se saca un valor de la cola y se le asigna a esa variable.

En ML, más que asignarle un valor a cada variable, se necesita unificar los valores de entrada con los patrones, y al representar los valores y patrones mediante árboles, la unificación se hace de manera natural.

En este trabajo no se implantaron todas las características de SML, ya que el objetivo principal era crear una herramienta que nos sirviera como base para la demostración automática de teoremas; y por lo tanto sólo se implantaron las construcciones que consideramos necesarias. Por ejemplo, las expresiones para ciclos no se implantaron porque la demostración de teoremas la podemos realizar recursivamente. Como un trabajo a futuro se podrían implantar las expresiones para ciclos, como el *while* y el *loop*, u otro tipo de expresiones.

El sistema ML-HOL se puede utilizar con fines didácticos, para mostrar las propiedades de un lenguaje de especificación.

En el sistema se pueden definir reglas de inferencia derivadas, tácticas y estrategias, pero no se cuenta con una herramienta mediante la cual el usuario pueda dirigir una prueba. Esto también sería un trabajo a futuro.

Apéndice A

Funciones ML

Cuando el sistema ML-HOL se inicia, cierto número de funciones ya están definidas. En este apéndice se describen las funciones (y su implantación) predefinidas más importantes. Se describen funciones de propósito general (Sección A.1) o que involucran términos lógicos y teoremas (Sección A.2). También se muestran las funciones que implantan las reglas de inferencia primitivas (Sección A.3); las reglas de inferencia derivadas (Sección A.4); las tácticas y estrategias (Secciones A.5 y A.6).

A.1 Funciones de propósito general y procesamiento de listas

Esta sección describe una selección de funciones ML de uso común aplicables a pares, listas y otros valores. Cada función se documenta con:

1. Su nombre y tipo.
2. Una breve descripción.
3. Una declaración ML definiendo la función.

A.1.1 Funciones de propósito general y combinadores

head : a' list $->$ a'
tail : a' list $->$ a' list

head l y *tail l* obtienen respectivamente la cabeza y el resto de la lista *l*.
head [1,2,3] = 1 tail [1,2,3] = [2,3]

```
fun head(hd::tl) = hd;
fun tail(hd::tl) = tl;
```

fst : a' * b' $->$ a'
snd : a' * b' $->$ b'

fst obtiene el primer elemento de un par y *snd* obtiene el segundo elemento de un par.

fun fst (x,y) = x;
fun snd (x,y) = y;

not : bool $->$ bool

not x evalúa a *false* si x es *true*; de otro modo evalúa a *true*.

fun not x = if x then false else true;

I : a' -> a'
K : a' -> b' -> a'
KI : a' $->$ b' $->$ b'

I, *K* y *KI* son los combinadores estándar.

fun I x = x;
fun K x y = x;
fun KI x y = y;

funpow : int $->$ (a' $->$ a') $->$ a' $->$ a'

funpow n f x aplica f a x, n veces: funpow n $f = f^n$

fun funpow n f x = if n=0 then x else funpow (n-1) f (f x);

A.1.2 Funciones de procesamiento de listas

length : a' list $->$ int

La función *length* calcula la longitud de una lista.
length $[x_1, \ldots x_n]$ = n;

fun length [ ] = 0 | length x::l = 1+(length l);

last : a' list $->$ a'
butlast : a' list $->$ a' list

La función *last* regresa el último elemento de una lista y *butlast* regresa todos excepto el último elemento de una lista.
last $[x_1, \ldots, x_n] = x_n$
butlast $[x_1, \ldots, x_n] = [x_1, \ldots, x_{n-1}]$

fun last l = if null (tail l) then (head l) else last (tail l);
fun butlast l = if null (tail l) then [ ] else (head l)::(butlast(tail l));

append : a' list $->$ a' list $->$ a' list

La función *append* concatena listas.
append $[x_1, \ldots, x_m]$ $[y_1, \ldots, y_n] = [x_1, \ldots, x_m, y_1, \ldots, y_n]$

fun append l1 l2 = if null l1 then l2 else (head l1)::append (tail l1) l2;

e1 : int $->$ a' list $->$ a'

La función *e1* extrae un elemento específico de una lista. Falla si el argumento entero es menor que 1 o más grande que la longitud de la lista.
e1 i $[x_1, \ldots, x_n] = x_i$

fun e1 i l = if i=1 then head l else e1 (i-1) (tail l);

replicate : a' $->$ int $->$ a' list

La función *replicate* construye una lista, consistente de un valor replicado un número específico de veces.
replicate x n evalúa a $[x, \ldots, x]$, una lista de longitud n.

fun replicate x n = if n=0 then [ ] else x::(replicate x (n-1));

A.1.3 Funciones de búsqueda en listas

find : (a' $->$ bool) $->$ a' list $->$ a'

find p l regresa el primer elemento de l que satisface el predicado p.

fun find p (x::l) = if p x then x else find p l;

mem : a' $->$ a' list $->$ bool

mem x l regresa *true* si algún elemento de l es igual a x, de lo contrario regresa *false*.

fun mem x l = if null l then false else
if x=(head l) then true else mem x (tail l);

assoc : a' $->$ (a' * b') list $->$ (a' * b')
rev_assoc : a' $->$ (b' * a') list $->$ (b' * a')

assoc x l busca en una lista l de pares, un par cuyo primer componente sea igual a x, regresando el primer par encontrado; similarmente, *rev_asooc y l* busca un par cuyo segundo componente sea igual a y.

fun assoc x l = find (fn (x1,y) => x=x1) l;
fun rev_assoc y l = find (fn (x,y1) => y=y1) l;

A.1.4 Funciones de transformaciones de listas

rev : a' list −> a' list

rev $[x_1, x_2, \ldots, x_n] = [x_n, \ldots, x_2, x_1]$

fun rev l = if null l then [ ] else append (rev (tail l)) ((head l)::[ ]);

elimina : a' list −> a' −> a' list

elimina l x, elimina de la lista l el elemento x.

```
fun elimina l x = if null l then [ ]
                  else if x=(head l) then (tail l)
                       else (head l)::(elimina (tail l) x);
```

inserta : a' list −> a' −> a' list

inserta l x, agrega el elemento x al final de la lista l.

fun inserta l x = l @ (x::[ ]);

filter : (a' −> bool) −> a' list −> a' list

filter p l aplica p a todo elemento de l, regresando una lista de aquellos que satisfacieron p.

```
fun filter p [ ] = [ ]
  | filter p (x::l) = if p x then (x::filter p l) else filter p l;
```

flat : a' list list −> a' list;

flat $[[l_{11}, \ldots, l_{1m_1}], [l_{21}, \ldots, l_{2m_2}], \ldots, [l_{n1}, \ldots, l_{nm_n}]] =$
$[l_{11}, \ldots, l_{1m_1}, l_{21}, \ldots, l_{2m_2}, \ldots, l_{n1}, \ldots, l_{nm_n}]$

fun flat [ ] = [ ] | flat (x::l) = x @ (flat l);

combine : (a' list * b' list) −> (a' * b') list
split : (a' * b') list −> (a' list * b' list)

combine $([x_1, \ldots, x_n], [y_1, \ldots, y_n]) = [(x_1, y_1), \ldots, (x_n, y_n)]$
split $[(x_1, y_1), \ldots, (x_n, y_n)] = ([x_1, \ldots, x_n], [y_1, \ldots, y_n])$

```
fun combine ([ ],[ ]) = [ ]
  | combine ((x::lx),(y::ly)) = (x,y)::combine(lx,ly);
fun split [ ] = ([ ],[ ])
  | split ((x,y)::l) = let val (lx,ly) = split l in (x::lx,y::ly) end;
```

partition : (a' −> bool) −> a' list −> (a' list * a' list)
chop_list : int −> a' list −> (a' list * a' list)

partition p l regresa un par de listas: la primera lista contiene los elementos de l que satisfacen p, la segunda lista contiene los demás elementos de l.

chop_list i $[x_1, \ldots, x_n] = ([x_1, \ldots, x_i], [x_{i+1}, \ldots, x_n])$

```
fun partition p l =
    itlist (fn a => fn (yes,no) => if p a then (a::yes,no)
           else (yes,(a::no))) l ([ ],[ ]);
fun chop_list i l = if i=0
    then ([ ],l)
    else let val (l1,l2) = chop_list (i-1) (tail l) in
           (head l)::l1,l2
    end;
```

A.1.5 Funciones de mapeo de listas

map : (a' −> b') −> a' list −> b' list

map f l regresa la lista obtenida de aplicar f a los elementos de la lista l.

```
fun map f l = if null l then [ ] else f (head l)::map f (tail l);
```

maplist : (a' −> b') list −> a' list −> b' list

maplist $[f_1, f_2, \ldots, f_n]\ [x_1, x_2, \ldots, x_n] = [f_1\, x_1, f_2\, x_2, \ldots, f_n\, x_n]$

```
fun maplist (f::Lf) Lx = if null Lx then [ ]
                         else f (head Lx)::(maplist Lf (tail Lx));
```

mapshape : int list −> (a' list −> b') list −> a' list −> b' list

mapshape $[m_1, m_2, \ldots, m_n]\ [f_1, f_2, \ldots, f_n]$
$[x_{1_1}, \ldots, x_{1_{m_1}}, x_{2_1}, \ldots, x_{2_{m_2}}, \ldots, x_{n_1}, \ldots, x_{n_{m_n}}] =$
$[f_1[x_{1_1}, \ldots, x_{1_{m_1}}], f_2[x_{2_1}, \ldots, x_{2_{m_2}}], \ldots, f_n[x_{n_1}, \ldots, x_{n_{m_n}}]]$

```
fun mapshape Li (f::Lf) Lx = if null Li then [ ]
                             else let val (l1,l2)=chop_list (head Li) Lx in
                                    (f l1)::(mapshape (tail Li) Lf l2)
                                  end;
```

A.1.6 Funciones para listas representando conjuntos

intersect : a' list −> a' list −> a' list
substract : a' list −> a' list −> a' list
union : a' list −> a' list −> a' list

intersect l_1 l_2 $=$ l_1 $\cap$ l_2
substract l_1 l_2 $=$ l_1 $-$ l_2
union l_1 l_2 $=$ l_1 $\cup$ l_2

```
fun intersect l1 l2 = filter (fn x => mem x l2) l1;
fun subtract l1 l2 = filter (fn x => not(mem x l2)) l1;
fun union l1 l2 = l1 @ (substract l2 l1);
```

distintct : a' list −> bool
setify : a' list −> a' list
set_equal : a' list −> a' list −> bool

distinct l regresa *true* si todos los elementos de *l* son distintos; de lo contrario regresa *false*. *setify l* remueve elementos repetidos de *l*, dejando la última ocurrencia de cada duplicado en la lista. *set_equal* l_1 l_2 regresa *true* si todo elemento de l_1 aparece en l_2 y todo elemento de l_2 aparece en l_1; de lo contrario regresa *false*.

```
fun distinct l = if null l then true else
                    if mem (head l) (tail l) then false else
                    distinct (tail l);
fun setify l = itlist (fn a => fn s => if mem a s then s else a::s) l [ ];
fun set_equal l1 l2 = null (subtract l1 l2) & null (subtract l2 l1);
```

A.2 Funciones sobre términos y teoremas

Esta sección describe una selección de funciones ML de uso común aplicables a términos y teoremas. También se describen algunas funciones auxiliares en la definición de reglas de inferencia primitivas y derivadas. Se muestra el nombre y tipo de la función, una breve descripción y su definición.

A.2.1 Funciones destructoras

dest_imp : term −> (term * term)
dest_conj : term −> (term * term)
dest_eq : term −> (term * term)
dest_thm : thm −> (term list * term)

dest_conj divide una conjunción en sus dos componentes. *dest_imp* divide una implicación en antecedente y consecuente. *dest_eq* divide una igualdad en sus dos componentes. *dest_thm* divide un teorema en hipótesis y conclusión.

```
fun dest_conj t1/\t2 = (t1,t2);
fun dest_imp t1==>t2 = (t1,t2);
fun dest_eq t1==t2 = (t1,t2);
fun dest_thm G|-t = (G,t);
```

hyp : thm $->$ term list
concl : thm $->$ term

hyp y *concl* obtienen la hipótesis y la conclusión de un teorema respectivamente.

fun hyp G|−t = G;
fun concl G|−t = t;

A.2.2 Funciones auxiliares

sustituye : term $->$ thm list $->$ term

sustituye $t\,[G_1 \;|{-}t_1 = t'_1, G_2 \;|{-}t_2 = t'_2, \ldots, G_n \;|{-}t_n = t'_n] = t'_i$ si t es igual a t_i, para algún $1 \leq i \leq n$, de lo contrario se regresa t.

```
fun sustituye t l = if null l then t
                    else let val t1,t2 = dest_eq(concl(head l)) in
                             if t=t1 then t2 else sustituye t (tail l)
                         end;
```

subst : thm list $->$ term $->$ term

Si se tiene una lista de teoremas $[G_1 \;|{-}t_1 = t'_1, G_2 \;|{-}t_2 = t'_2, \ldots, G_n \;|{-}t_n = t'_n]$ y en el término t existen ocurrencias libres de los términos $t_1, t_2, \ldots, t_n$, entonces

subst $t\,[G_1 \;|{-}t_1 = t'_1, G_2 \;|{-}t_2 = t'_2, \ldots, G_n \;|{-}t_n = t'_n] =$
$t\,[t'_1, t'_2, \ldots, t'_n]$

donde $t\,[t'_1, t'_2, \ldots, t'_n]$ denota el resultado de reemplazar simultáneamente cada ocurrencia de t_i por t'_i (para $1 \leq i \leq n$). Esta función se utiliza para implantar la regla de inferencia SUBST.

```
fun subst L (t1==>t2) = '(subst L t1)==>(subst L t2)'
  | subst L (t1 /\t2) = '(subst L t1) /\(subst L t2)'
  | subst L (t1 \/ t2) = '(subst L t1) \/ (subst L t2)'
  | subst L (t1==t2) = '(subst L t1)==(subst L t2)';
  | fun subst L t = 'sustituye t L'
```

union_hyp : thm list $->$ term list

union_hyp toma una lista de teoremas y regresa una lista consistente en la unión de las hipótesis de los teoremas. Esta función también se utiliza para implantar la regla de inferencia SUBST.

```
fun union_hyp L = if null L then [ ] else
                      union (hyp(head L)) (union_hyp (tail L));
```

subst_term : term $->$ term $->$ term $->$ term

subst_term t1 x t2, reemplaza cada ocurrencia de x por *t1* en el término *t2*.

```
fun subst_term t1 x (t2==>t3) =
      '(subst_term t1 x t2)==>(subst_term t1 x t3)'
   | subst_term t1 x (t2 /\t3) =
      '(subst_term t1 x t2) /\(subst_term t1 x t3)'
   | subst_term t1 x (t2 \/ t3) =
      '(subst_term t1 x t2) \/ (subst_term t1 x t3)'
   | subst_term t1 x (t2==t3) =
      '(subst_term t1 x t2)==(subst_term t1 x t3)'
   | subst_term t1 x t2 = if x=t2 then 't1' else 't2';
```

A.3 Reglas de inferencia primitivas

En esta sección se describen las funciones ML que implantan las reglas de inferencia primitivas en el sistema ML-HOL. Se muestra el nombre y tipo de la función ML que implanta una regla de inferencia, su descripción y su definición.

Introducción de premisas
ASSUME : term $->$ term

$$\overline{t \;\vdash\; t}$$

ASSUME t evalúa a $t\,|-t$. Falla si t no es de tipo *bool*.

```
fun ASSUME x:bool = ['x']|-'x';
```

Reflexividad
REFL : term $->$ term

$$\overline{\vdash\; t = t}$$

REFL t evalúa a $|-t = t$. Una llamada a *REFL* nunca falla.

```
fun REFL x = |-'x=x';
```

Beta-conversión
BETA_CONV : term $->$ thm

$$\overline{\vdash\; (\lambda x.\, t_1)\, t_2 = t_1[x/t_2]}$$

donde $t_1[x/t_2]$ denota el resultado de sustituir x con t_2 en t_1, con el renombramiento apropiado de variables para prevenir que las variables libres en t_2 se conviertan en acotadas después de la sustitución.

BETA_CONV $(\backslash x.\, t_1)\, t_2$ evalúa al teorema $|-(\backslash x.\, t_1)\, t_2 = t_1[x/t_2]$. Falla si el argumento de *BETA_CONV* no es de la forma $(\backslash x.\, t_1)\, t_2$.

```
fun BETA_CONV t2 (\x. t1) = |-'(\x. t1) t2 == subst_term t2 x t1';
```

Substitución
SUBST : thm list $->$ thm $->$ thm

$$\frac{\Gamma 1 \ \vdash \ t_1 = t_1' \quad \cdots \quad \Gamma_n \ \vdash \ t_n = t_n' \qquad \Gamma \ \vdash \ t\,[t_1, \ldots, t_n]}{\Gamma_1 \cup \cdots \cup \Gamma_n \cup \Gamma \ \vdash \ t\,[t_1', \ldots t_n']}$$

donde $t\,[t_1, \ldots, t_n]$ denota un término t con algunas ocurrencias libres de los términos $t_1, \ldots, t_n$, y $t\,[t_1', \ldots, t_n']$ denota el resultado de reemplazar simultáneamente cada ocurrencia de t_i por t_i' (para $1 \leq i \leq n$), con el renombramiento apropiado de variables para prevenir que las variables libres en t_2 se conviertan en acotadas después de la sustitución.

SUBST $[\,|-t_1 = t_1', \ldots, |-t_n = t_n']\ \ \Gamma\,|-t\,[t_1, \ldots, t_n]$ regresa $\Gamma\,|-t\,[t_1', \ldots, t_n']$

fun SUBST L t = union_hyp L $|-$ *'subst L t';*

Abstracción
ABS : term $->$ thm $->$ thm

$$\frac{\Gamma \ \vdash \ t_1 = t_2}{\Gamma \ \vdash \ (\lambda x.\, t_1) = (\lambda x.\, t_2)}$$

ABS $x\ \Gamma\,|-t_1 = t_2$ regresa el teorema $\Gamma\,|-(\backslash x.\, t_1) = (\backslash x.\, t_2)$.

fun ABS x G$|-$*t1 = G*$|-$*'\x. t1==\x t2';*

Instanciación de Tipos
INST_TYPE: (type # type) list $->$ thm $->$ thm

$$\frac{\Gamma \ \vdash \ t}{\Gamma \ \vdash \ t[\alpha_1, \ldots, \alpha_n/\sigma_1, \ldots, \sigma_n]}$$

donde $t[\alpha_1, \ldots, \alpha_n/\sigma_1, \ldots, \sigma_n]$ es el resultado de substituir en paralelo en el término t, las variable de tipo $\alpha_1, \ldots, \alpha_n$ con los tipos $\sigma_1, \ldots, \sigma_n$, con la restricción que ninguna de las variables de tipo $\alpha_1, \ldots, \alpha_n$ ocurre en Γ.

INST_TYPE $[(\alpha_1, \sigma_1), \ldots, (\alpha_n, \sigma_n)]$ *th* regresa el resultado de instanciar cada ocurrencia de α_i en el teorema *th* con el tipo σ_i (para $1 \leq i \leq n$).

Descargando una premisa
DISCH : term $->$ thm $->$ thm

$$\frac{\Gamma \ \vdash \ t_2}{\Gamma - \{t_1\} \ \vdash \ t_1 \Rightarrow t_2}$$

DISCH $t_1\ \Gamma\,|-t_2$ evalúa al teorema $\Gamma - \{t_1\}\,|-t_1 ==> t_2$. *DISCH* falla si el término dado como su primer argumento no es de tipo *bool*.

fun DISCH t1 T$|-$*t2 = elimina T t1*$|-$*'t1==>t2';*

Modus Ponens
MP : thm −> thm −> thm

$$\frac{\Gamma 1 \ \vdash \ t_1 \Rightarrow t_2 \qquad \Gamma_2 \ \vdash \ t_1}{\Gamma_1 \cup \Gamma_2 \ \vdash \ t_2}$$

MP toma dos teoremas y regresa el resultado de aplicar Modus Ponens; falla si los argumentos no son de la forma correcta.

```
fun MP T1|-t1 T2|-t3 = if fst(dest_imp(t1))=t3 then
                           union T1 T2|-'snd(dest_imp(t1))';
```

A.4 Reglas de inferencia derivadas

Una *regla de inferencia derivada* es un procedimiento ML que genera una prueba a partir de hipótesis dadas cada vez que se invoca. Las hipótesis son los argumentos de la regla [HolT91].

Esta sección describe las funciones ML [HolD91] que implantan algunas reglas de inferencia derivadas. Cada función se documenta con:

1. Su nombre y tipo.
2. La regla de inferencia que implanta.
3. Un declaración ML definiendo la función.

Adicionando una premisa
ADD_ASSUM : term −> thm −> thm

$$\frac{\Gamma \ \vdash \ t}{\Gamma, t' \ \vdash \ t}$$

```
fun ADD_ASUM t1 G|-t = inserta G t1|-'t';
```

Cargando una premisa
UNDISCH : thm −> thm

$$\frac{\Gamma \ \vdash \ t_1 \Rightarrow t_2}{\Gamma, t_1 \ \vdash \ t_2}$$

```
fun UNDISCH G|-(t1==>t2) = inserta G t1|-'t2';
```

Simetría de igualdad
SYM : thm −> thm

$$\frac{\Gamma \ \vdash \ t_1 = t_2}{\Gamma \ \vdash \ t_2 = t_1}$$

```
fun SYM G|-(t1==t2) = G|-'t2==t1';
```

Transitividad de igualdad
TRANS : thm $->$ thm $->$ thm

$$\frac{\Gamma_1 \ \vdash \ t_1 = t_2 \qquad \Gamma_2 \ \vdash \ t_2 = t_3}{\Gamma_1 \cup \Gamma_2 \ \vdash \ t_1 = t_2}$$

fun TRANS G1|−t1 G2|−t2 = if snd(dest_eq t1)=fst(dest_eq t2)
then union G1 G2|−'fst(dest_eq t1)==snd(dest_eq t2)';

Aplicación de un término a un teorema
AP_TERM : term $->$ thm $->$ thm

$$\frac{\Gamma \ \vdash \ t_1 = t_2}{\Gamma \ \vdash \ t\ t_1 = t\ t_2}$$

fun AP_TERM t G|−(t1==t2) = G|−'t t1==t t2';

Aplicación de un teorema a un término
AP_THM : term $->$ thm $->$ thm

$$\frac{\Gamma \ \vdash \ t_1 = t_2}{\Gamma \ \vdash \ t_1\ t = t_2\ t}$$

fun AP_THM t G|−(t1==t2) = G|−'t1 t==t2 t';

Modus Ponens para la igualdad
EQ_MP : thm $->$ thm $->$ thm

$$\frac{\Gamma \ \vdash \ t_1 = t_2 \qquad \Gamma_2 \ \vdash \ t_1}{\Gamma_1 \cup \Gamma_2 \ \vdash \ t_2}$$

fun EQ_MP G1|−t1 G2|−t2 = if fst(dest_eq t1)=t2
then union G1 G2|−'snd(dest_eq t1)';

Implicación a partir de una igualdad
EQ_IMP_RULE : thm $->$ (thm * thm)

$$\frac{\Gamma \ \vdash \ t_1 = t_2}{\Gamma \ \vdash \ t_1 \Rightarrow t_2 \qquad \Gamma \ \vdash \ t_2 \Rightarrow t_1}$$

fun EQ_IMP_RULE G|−(t1==t2) = (G|−'t1==>t2',G|−'t2==>t1');

Introducción de T
TRUTH : void $->$ thm

$\vdash \ T$

fun TRUTH = |−'T';

Igualdad con eliminación de T
EQT_ELIM : thm −> thm

$$\frac{\Gamma \ \vdash \ t = T}{\Gamma \ \vdash \ t}$$

fun EQT_ELIM G|−(t==T) = G|−'t';

Especialización (Eliminación de ∀)
SPEC : term −> thm −> thm

$$\frac{\Gamma \ \vdash \ \forall x.\, t}{\Gamma \ \vdash \ t[t'/x]}$$

fun SPEC t1 G|−(!x. t) = G|−'subst_term t1 x t';

Igualdad con Introducción de T
EQT_INTRO thm −> thm

$$\frac{\Gamma \ \vdash \ t}{\Gamma \ \vdash \ t = T}$$

fun EQT_INTRO G|−t = G|−'t=T';

Generalización (Introducción de ∀)
GEN : term −> thm −> thm

$$\frac{\Gamma \ \vdash \ t}{\Gamma \ \vdash \ \forall x.\, t}$$

fun GEN x G|−t = G|−'!x. t';

Introducción de ∃
EXISTS : (term * term) −> thm −> thm

$$\frac{\Gamma \ \vdash \ t_1[t_2]}{\Gamma \ \vdash \ \exists x.\, t_1[x]}$$

fun EXISTS (x,t2) G|−t1 = G|−'?x. subst_term x t2 t1';

Introducción de ∧
CONJ : thm −> thm −> thm

$$\frac{\Gamma_1 \ \vdash \ t_1 \qquad \Gamma_2 \ \vdash \ t_2}{\Gamma_1 \cup \Gamma_2 \ \vdash \ t_1 \wedge t_2}$$

fun CONJ G1|−t1 G2|−t2 = union G1 G2|−'t1 /\t2';

Eliminación de $\wedge$
CONJUNCT : thm $->$ (thm * thm)

$$\frac{\Gamma \vdash t_1 \wedge t_2}{\Gamma \vdash t_1 \qquad \Gamma \vdash t_2}$$

fun CONJUNT G|−(t1 /\t2) = (G|−'t1',G|−'t2');

Introducción derecha de $\vee$
DISJ1 : term $->$ thm $->$ thm

$$\frac{\Gamma \vdash t_1}{\Gamma \vdash t_1 \vee t_2}$$

fun DISJ1 t2 G|−t1 = G|−'t1 \/ t2';

Introducción izquierda de $\vee$
DISJ2: term $->$ thm $->$ thm

$$\frac{\Gamma \vdash t_2}{\Gamma \vdash t_1 \vee t_2}$$

fun DISJ2 t1 G|−t2 = G|−'t1 \/ t2';

Eliminacion de $\vee$
DISJ_CASES : thm $->$ thm $->$ thm $->$ thm

$$\frac{\Gamma \vdash t_1 \vee t_2 \qquad \Gamma_1, t_1 \vdash t \qquad \Gamma_2, t_2 \vdash t}{\Gamma \cup \Gamma_1 \cup \Gamma_2 \vdash t}$$

fun DISJ_CASES G1|−(t1 \/ t2) G3|−t3 G4|−t4 =
if (mem t1 G3) & (mem t2 G4) & t3=t4
then union G1 (union(elimina G3 t1) (elimina G4 t2))|−'t3';

Regla clásica de contradicción

$$\frac{\Gamma, \sim t \vdash F}{\Gamma \vdash t}$$

fun CCONTR t G|−F = if (mem t G) then G|−'∼ t';

A.5 Tácticas

Esta sección contiene una selección de las tácticas [HolD91, HolT91] más comunes usadas en el sistema ML-HOL. Cada táctica se documenta con:

1. Su nombre y tipo.
2. Una breve descripción.
3. Los usos de esa táctica.
4. La definición de la función que implanta la táctica.

Aceptación de un teorema
ACCEPT_TAC : thm $->$ tactic

ACCEPT_TAC th es una táctica que resuelve cualquier meta que se representa por *th*.

Uso: Incorporar pruebas hacia adelante o teoremas ya probados dentro de pruebas dirigidas a metas.

Adicionando una premisa
ASSUME_TAC: thm $->$ tactic

ASSUME_TAC $|-u$ adiciona u como una premisa.

$$\frac{t}{\overline{\{u\}t}}$$

Uso: Enriquecer las premisas de una meta con definiciones o teoremas previamente probados.

```
fun ASSUME_TAC G|-t (asl,w) =
    ( [(inserta asl t,'w')], (fn [th1,th2] =>
     MP (DISCH (concl th2) th1) th2));
```

Especialización
GEN_TAC : tactic

Especializa un teorema cuantificado universalmente a un valor arbitrario.

$$\frac{!x.\,t[x]}{\overline{t[x']}}$$

donde x' es una variante de x no libre en la meta o premisas.

Uso: Resolver metas cuantificadas universalmente. Frecuentemente *GEN_TAC* es el primer paso de una prueba orientada a metas. *STRIP_TAC* (ver posteriormente) aplica *GEN_TAC* para metas cuantificadas universalmente.

```
fun GEN_TAC t (asl:term list,w) =
    ([(asl,SPECT t w)], (fn [th1,th2] => GEN (concl th1) th2));
```

Conjunción
CONJ_TAC : tactic

Divide una meta $t_1/\backslash t_2$ en dos submetas t_1 y t_2.

$$\frac{t_1/\backslash t_2}{t_1 \qquad t_2}$$

Uso: Resuelve metas conjuntivas. *CONJ_TAC* se invoca por *STRIP_TAC* (ver posteriormente).

```
fun CONJ_TAC (asl:term list,w) =
    let val l,r = dest_conj w in
        ( [(asl,l),(asl,r)], (fn [th1,th2] => CONJ th1 th2)) end;
```

Descargando una premisa
DISCH_TAC : tactic

Mueve el antecedente de una meta implicativa adentro de las premisas, dejando el consecuente como el componente término.
Si la meta es $t_1/\backslash \cdots /\backslash t_n$ ==> t entonces *DISCH_TAC* mueve cada t_i como una premisa separada.

$$\frac{u ==> v}{\{u\}v}$$

Uso: Resolver metas de la forma u ==> v, asumiendo u y después resolver v bajo esa premisa. *STRIP_TAC* (ver más abajo) invoca a *DISCH_TAC* en metas implicativas.

```
fun DISCH_TAC (asl,w) =
    let val l,r = dest_imp w in
        ( [(inserta asl l,r)], (fn [th1,th2] => DISCH (concl th1) th2)) end;
```

Descomposición de combinaciones simples
STRIP_TAC : tactic

Descompone una meta. *STRIP_TAC* remueve los conectivos de la meta usando *CONJ_TAC, DISCH_TAC, GEN_TAC* y otras tácticas.

Uso: Útil para dividir metas en piezas más manejables. Lo mejor que se puede hacer primero en muchos casos es *REPEAT STRIP_TAC*, donde *REPEAT* es la estrategia que repetidamente aplica una táctica hasta que falla (ver Sección A.6).

Sustitución
SUBST_TAC : thm list −> thm

Evaluando *SUBST_TAC* $[\,|-u_1 = v_1, \ldots, |-u_n = v_n]$ cambia cada subtérmino $t\,[u_1, \ldots, u_n]$ de la meta por $t\,[v_1, \ldots, u_n]$

```
fun SUBST_TAC L (asl:term list,w) =
    ( [(asl,subst L w)], (fn (G|-t)::L1 => G|-'subst L1 t'));
```

Análisis por casos de una disyunción
DISJ_CASES_TAC : thm $->$ tactic

DISJ_CASES_TAC $|-u\backslash/v$ dividide una meta en dos casos: uno con u como premisa y el otro con v como premisa.

$$\frac{t}{\{u\}t \qquad \{v\}t}$$

Uso: Análisis por casos.

fun DISJ_CASES_TAC G|−(t1 \/ t2) (asl:term list,w) =
([(inserta asl t1,'w'),(inserta asl t2,'w')],
(fn [th1,th2,th3] => DISJ_CASES th1 th2 th3));

Análisis por casos de un término booleano
ASN_CASES_TAC : term $->$ tactic

ASN_CASES_TAC u, hace el análisis por casos de u, donde u es un término booleano,

$$\frac{t}{\{u\}t \qquad \{\sim u\}t}$$

Uso: Análisis por casos.

fun ASN_CASES_TAC t (asl,w) =
DISJ_CASES_TAC (SPEC t MIDDLE) (asl,w);

donde *MIDDLE* es el teorema $|-!t.\, t\backslash/\sim t$.

Identidad
ALL_TAC : tactic

Es la táctica identidad; reduce una meta (Γ, t) a la única submeta (Γ, t), es decir, no tiene efecto. *ALL_TAC* es útil para escribir tácticas compuestas.

$$\frac{t}{t}$$

fun ALL_TAC (asl:term list,w:term) = ([(asl,w)], (fn [th1] => IR th1));

donde *IR* esta definida por *fun IR Teo:thm = Teo;*

Dividiendo equivalencias lógicas
EQ_TAC : tactic

EQ_TAC divide una meta ecuacional en dos implicaciones (el caso "si" y el caso "solo si").

$$\frac{u = v}{u ==> v \qquad v ==> u}$$

Uso: Probar equivalencias lógicas, es decir, metas de la forma "$u = v$", donde u y v son términos booleanos.

```
fun EQ_TAC (asl,w) =
    let val l,r = dest_eq w in
        ( [(asl,'l==>r'),(asl,'r==>l')],(fn [th1,th2] => IMP_EQ th1 th2))
    end;
```

Inducción Matemática
INDUCT_TAC : tactic

INDUCT_TAC reduce una meta de la forma $(\Gamma, !n.\, t[n])$ a las submetas $(\Gamma, t[0])$ y $(\Gamma \cup \{t[n]\}, t[SUC\ n])$. La premisa extra "$t[n]$" se indica en la notación de tácticas mediante llaves. La primera submeta es el caso base y la segunda submeta es el caso de paso de inducción.

$$\frac{!n.\, t[n]}{\overline{t[0] \qquad \{t[n]\}\ t[SUC\ n]}}$$

Uso: Hacer inducción matemática sobre los números naturales.

Resolver metas existenciales
EXISTS_TAC : term $->$ tactic

EXISTS_TAC u reduce una meta existencial $!x.\, t[x]$ a la submeta $t[u]$

$$\frac{!x.\, t[x]}{\overline{t[u]}}$$

Uso: Probar metas existenciales.

A.6 Estrategias

Una *estrategia* [HolD91, HolT91] es una función ML que regresa una táctica (o tácticas) como resultado. Las estrategias pueden tomar varios parámetros; esto se refleja en los distintos tipos que tienen las estrategias construidas. Las estrategias se usan para construir tácticas compuestas.

Esta sección contiene una selección de las estrategias más comunes usadas en el sistema ML-HOL. Se muestra el nombre y tipo de la función ML que implanta la estrategia, su descripción y su definición.

Alternancia
ORELSE : tactic $->$ tactic $->$ tactic

Si T_1 y T_2 son tácticas, entonces la expresión ORELSE T_1 T_2 evalúa a una táctica, la cual aplica T_1 a menos que falle; si falla, aplica T_2.

Primer éxito
FIRST : tactic list $->$ tactic

La táctica *FIRST* aplica la primera táctica que tiene éxito en una lista de tácticas.

Detección de cambios
CHANGED_TAC : tactic $->$ tactic

CHANGED_TAC T g falla si las submetas producidas por T son justamente *[g]*; de otra manera es equivalente a *T g*.

```
fun CHANGED_TAC T1:tactic g =
    let val (g1,p) = T1 g in
        if set_equal g1 [g] then ([ ],p) else (g1,p)
    end;
```

Secuencias
THEN : tactic $->$ tactic $->$ tactic

Si T_1 y T_2 son tácticas, entonces la expresión *THEN* T_1 T_2 evalúa a una táctica, la cual primero aplica T_1 y entonces aplica T_2 a cada submeta producida por T_1.

```
fun THEN T1:tactic T2:tactic g =
    let val (g1,p) = T1 g in
        let val (g11,p1) = split(map T2 g1) in
            (flat g11, (fn th1 => p(mapshape(map length g11) p1 th1)))
        end
    end;
```

Secuencia selectiva
THENL : tactic $->$ tactic list $->$ tactic

Si $T_1, \ldots, T_n$ son tácticas y la táctica T produce n submetas, entonces la expresión *THENL T* $[T_1, \ldots, T_n]$ es una táctica la cual aplica primero T y entonces aplica T_1 a la i-ésima submeta producida por T. La estrategia *THENL* es útil si se desea aplicar diferentes tácticas a diferentes submetas.

```
fun THENL T1:tactic TL:tactic list g =
    let val (g1,p) = T1 g in
        let val (g11,p1) = split(maplist TL g1) in
            (flat g11, (fn th1 => p(mapshape(map length g11) p1 th1)))
        end
    end;
```

Aplicación sucesiva
EVERY : tactic list $->$ tactic

La estrategia *EVERY* aplica una lista de tácticas una después de otra.
$EVERY\ [T_1, T_2, \ldots, T_n] = THEN\ T_1(THEN\ T_2 \ldots (THEN T_{n-1} T_n) \ldots)$

Repetición
REPEAT : tactic $->$ tactic

SI T es una táctica, entonces $REPEAT\ T$ es una táctica que repetidamente aplica T hasta que falla.

Bibliografía

[AhA+90] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. *Compiladores: Principios, Técnicas y Herramientas.* Addison-Wesley Iberoamericana, Delaware, E.U.A., 1990.

[GuE89] Elsa L. Gunter. *Doing Algebra in Simple Type Theory.* Reporte técnico MS-CIS-89-38, Department of Computer and Information Science, Universidad de Pennsylvania, Junio 1989, distribuido con el Sistema HOL.

[LiH+83] Henry Lieberman, Carl Hewitt. *A Real-Time Garbage Collector Based on the Lifetimes of Objects.* Communications of the ACM, Volumen 26, Número 6, páginas 419-429, Junio 1983.

[ScH96] Herbert Schildt. *C Manual de Referencia.* McGraw-Hill, Tercera edición, 1996.

[RuJ91] John Rushby. *Design Choices in Specification Languajes and Verification Systems.* Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications, IEEE Computer Society, páginas 195-204, Agosto 1991.

[BoJ+90] Jonathan P. Bowen, Michael G. Hinchey. *Ten Commandments of Formal Methods.* 1990.

http://www.comlab.ox.ac.uk/archive/formal-methods.html

[CuJ86] José Cuena. *Lógica Informática.* Alianza Editorial Mexicana, S.A., México 1986

[LeM90] M. E. Lesk. *lex -a Lexical Analyzer Generator.* SUN Microsystems, páginas 203-226, Marzo 1990.

[SeR89] Ravi Sethi. *Lenguajes de Programación.* Addison-Wesley, 1989.

[JoS90] Steve Johnson. *yacc -Yet Another Compiler-Compiler.* SUN Microsystems, páginas 227-264, Marzo 1990.

[ReS90] S. Reeves, M. Clark. *Logic for Computer Science.* Addison-Wesley, 1990.

[HolD91] *The HOL System: Description.* Universidad de Cambrigde, Computer Laboratory, Julio 1991.

http://lal.cs.byu.edu/lal/hol-documentation.html.

[HolT91] *The HOL System: Tutorial.* Universidad de Cambrigde, Computer Laboratory, Julio 1991.

http://lal.cs.byu.edu/lal/hol-documentation.html.