I Sistemas de Tipos

0. Introducción

Un sistema de tipos dota a los lenguajes de la capacidad de restringir los objetos que pueden ser asignados a las variables. Esto permite una cierta potencia a la hora de detectar errores y mejora la comprensión del código. Por el contrario, se pierde flexibilidad, aunque este hecho se palia al permitir un polimorfismo restringido, esto es, que se puedan asignar ciertos tipos más -++o menos parecidos en cuanto a estructura pero diferentes en definitiva, a una misma variable.

Los sistemas de tipos se dividen en dos grandes grupos, a saber:

-Los tipos **Church**, también llamados **prescriptivos** o tipos **à la ML**. Más adelante, en la parte teórica, se explicará con detalle en qué consisten. Están relacionados con el concepto de tipificación fuerte, en el sentido de que una expresión tiene un único tipo asociado. Pero podemos indicar que lenguajes como λProlog, Mercury o ML parten de estos sistemas de tipos.

-Los tipos de **Curry**, también conocidos como **descriptivos**, están asociados al concepto de tipificación débil, en el sentido de que una expresión puede tener varios tipos, aunque deben adaptarse a la estructura de la expresión. Más adelante se verá en detalle qué queremos decir con esto. Haskell es uno de los lenguajes más conocidos que se basan en estos sistemas de tipos.

En el presente trabajo se pretende dar una idea general sobre la teoría de los sistemas de tipos para luego entrar de lleno en Mercury, que es un intento de dotar a los lenguajes tipo Prolog de un sistema de tipos polimórfico.

Hay poca información sobre la teoría de los sistemas de tipos en medios como Internet. Sin embargo, si aparece una cierta cantidad de ella sobre el lambda-cálculo, base de los sistemas de tipos que describiremos a continuación. No obstante, el objetivo de nuestro trabajo es centrarnos en los sistemas de tipos, con lo que recomendamos al lector consultar la bibliografía para estudiar el lambda-cálculo de un modo más extenso.

1. Teoría de los Sistemas de Tipos

1.1 ¿En qué consiste un sistema de tipos?

Siempre informalmente, podríamos dar una definición aproximada y más bien práctica de sistema de tipos.

En el contexto de la programación declarativa, un sistema de tipos consiste en la teoría base que permite asociar a cada término de un lenguaje declarativo un tipo.

Un tipo, a su vez, se podría definir como un conjunto de términos de un lenguaje que poseen una serie de características en común que les permite interactuar entre sí o

ser objeto de transformaciones sólo aplicables a ellos, por el hecho de poseer esas características.

1.2 Los sistemas de tipos de Church

Se trata de cualquier sistema de tipos que sea **explícito** (en las abstracciones del lenguaje, el tipo de la variable ligada se explicita).

P. ej.:

$\lambda x : A.\ x$ (se interpreta como la función que toma como parámetro un objeto denominado "x", que es de tipo "A" y devuelve ese mismo objeto. El tipo de la propia abstracción sería $A \rightarrow A$)

$\lambda x : (B \rightarrow C).\ x$ (se interpreta como la función que toma como parámetro un objeto "x" de tipo $B \rightarrow C$, esto es, una función de $B \rightarrow C$, y devuelve esa misma función "x". El tipo de la abstracción sería $(B \rightarrow C) \rightarrow (B \rightarrow C)$)

En estos dos ejemplos podemos ver que si admitiéramos que el tipo $B \rightarrow C$ se pudiera renombrar al tipo A de la abstracción de arriba, la de abajo se ajustaría a la de arriba. Esto, sin embargo, no se puede dar al contrario, es decir, no toda función que se ajuste a la abstracción de arriba puede hacerlo a la de abajo.

1.3 Los sistemas de tipos de Curry

1.3.1 Definición

Son el caso opuesto al anterior, esto es, son los sistemas que son **implícitos**, esto es, en las abstracciones no se explicita el tipo de la variable ligada. Las abstracciones poseen la forma de las expresiones del lambda-cálculo.

P. ej.

$\lambda x.x$ Se interpreta como una función que toma como parámetro un objeto "x" y devuelve ese mismo objeto. No se especifica, precisamente, el tipo de "x", con lo que podría ser de tipo A, $B \rightarrow C$, $A \rightarrow B \rightarrow C$, ... o cualquier otro.

Como se ve en el propio ejemplo, no existe **unicidad** de tipos para el caso de variables ligadas de una lambda-expresión. Sin embargo, sí la hay cuando hablamos de una variable libre.

Las fórmulas de este sistema son, en realidad, pares formados por un **sujeto** y un predicado, se escriben de la forma **a : A**. A cada una de estas fórmulas se les denomina **asignación de tipos**.

El **sujeto** (**a** en el ejemplo) es una lambda-expresión y el **predicado** es el tipo al que pertenece el sujeto.

Por otro lado, sea Γ un conjunto de fórmulas sin sujetos repetidos (a : A, a : B ∈ Γ, entonces A≡B). A este conjunto se le denomina **contexto** o **base**. Se pueden definir una serie de axiomas y reglas de inferencia de tal modo que si de un contexto Γ es posible inferir la fórmula a : A (es decir, que la lambda-expresión "a" es de tipo "A"), notaremos a este hecho con $\Gamma \vdash a : A$. Los axiomas y reglas de inferencia son los siguientes para los sistemas de tipos de Curry:

(ax) $\dfrac{}{\Gamma \vdash x : A}$ $\quad x: A \in \Gamma$

($\rightarrow$ e) $\dfrac{\Gamma \vdash f : A \rightarrow B \quad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B}$

($\rightarrow$ i) $\dfrac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x .b : A \rightarrow B}$

El axioma nos indica algo ciertamente sencillo de entender. Si una variable tiene asignado un tipo en un contexto, de éste podemos inferir que su tipo es el anteriormente citado.

La primera regla nos proporciona el tipo de la aplicación de una función sobre un elemento del dominio de dicha función. Y la segunda nos genera el tipo de una lambda-expresión a partir del tipo de su variable y del objeto devuelto.

1.3.2 Propiedades de los sistemas de tipos de Curry

La siguiente regla nos indica que la evaluación de una expresión mediante β-reducciones no altera su tipo:

(Sβ) $\Gamma \vdash a : A \ \wedge\ a \rightarrow_\beta a' \Rightarrow \Gamma \vdash a' : A$ (subject β-reduction)

También existe una propiedad que nos asegura que todo término tipificable es **fuertemente normalizante**, es decir, que cualquier secuencia de reducciones (β-reducciones) es **finita**. Y sabemos, aplicando a este hecho el teorema de estandarización, que la forma normal es computable y que la relación de igualdad en β ($=_\beta$) es decidible (siempre hablando de términos tipificables, por supuesto).

Utilizando lo anterior, y sabiendo además que cualquier subtérmino de un término tipificable es tipificable, podemos concluir que **todo programa siempre termina y cualquier subprograma suyo también**.

La aplicación del siguiente lema nos introduce el no cumplimiento de la propiedad de unicidad de tipos:

(Lema de sustitución) $\Gamma \vdash a : A \Rightarrow <B := C>\ \Gamma \vdash a : <B := C>\ A$

Sin embargo, esta propiedad (unicidad de tipos) sí es cierta para las variables del lenguaje:

(Unicidad Variables) $\Gamma \vdash x : A \ \wedge \ \Gamma \vdash x: B \Rightarrow A \equiv B$

El siguiente lema es el recíproco de las reglas del sistema axiomático que se propone como base para los sistemas de tipos de Curry:

(Lema de generación)
(a) Si $\Gamma \vdash x : A$, entonces $x : A \in \Gamma$
(b) Si $\Gamma \vdash f\ a : B$, entonces existe un tipo A tal que $\Gamma \vdash f : A \to B \ \wedge \ \Gamma \vdash a : A$
(c) Si $\Gamma \vdash \lambda x.b : C$, entonces existen dos tipos A y B tales que $C \equiv A \to B$, y además se cumple que $\Gamma, x : A \vdash b : B$

Por último, diremos que si un término es tipificable entonces admite un **par principal**, compuesto por un **contexto principal** y un **tipo principal** tales que si existiese otro contexto a partir del cual se pudiera inferir que el término es de otro tipo, entonces se demuestra que existe una sustitución que aplicada al contexto y tipo principales, nos conduce a que dicho contexto contiene al principal (con la sustitución) y el tipo es equivalente al principal (con la sustitución). Así, dos tipos principales de un término coinciden salvo en los identificadores de sus variables de tipo.

Además, este par principal es computable, si es que el término es tipificable. Si no lo fuera, el propio algoritmo nos indicaría este hecho.

1.4 Problemas de los sistemas de tipos

Los tres principales problemas que debe resolver un sistema de tipos de cualquier lenguaje son los siguientes:

a) comprobación de tipos (type checking), denotado con $\vdash^{?} a : A$
Consiste en comprobar que el tipo de "a" es efectivamente "A". Este problema es decidible utilizando la existencia de un par principal para el término "a". Una vez calculado dicho par, no queda más que buscar una sustitución que haga que el contexto principal quede incluido en el contexto del programa y el tipo sea equivalente al dado.

b) tipificación (typability), denotado con $\vdash a : ?$
Consiste en averiguar el tipo de un elemento "a". Simplemente hay que llamar al algoritmo que calcula el par principal y extraer su tipo principal. Por tanto, también es decidible.

c) habitabilidad (inhabitation), denotado con $\vdash ? : A$
Consiste en comprobar si existe al menos un elemento tipificable de nuestro lenguaje que sea del tipo "A". Este problema es decidible si aplicamos la correspondencia Howard-Curry-de Bruijn, puesto que se demuestra que es equivalente la derivación de la lógica proposicional. Más tarde lo volveremos a mencionar al hablar de dicha correspondencia. No obstante, en cuanto se extiende el sistema de tipos de Curry para introducir, p .ej. polimorfismo, este problema pasa a ser indecidible.

1.4 La correspondencia Howard-Curry-de Bruijn

Si eliminamos los sujetos de una deducción e interpretamos el constructor $\rightarrow$ como la implicación de la lógica proposicional, obtenemos la forma de una deducción en la lógica proposicional intuicionista.

Como hemos comentado anteriormente, de esta forma es posible hacer corresponder el problema de la habitabilidad con el de la derivación lógica en el cálculo de proposiciones.

El trabajo más interesante en este aspecto es el de W. Howard, en 1969 con el título *The formulae-as-types notion of construction*.

1.5 Lambda cálculo polimórfico

Dentro de los sistemas de tipos de Curry, aparte del ya comentado, existen otros sistemas de segundo orden, tal y como el **lambda-cálculo polimórfico** o también conocido como **λ2**. La idea que se perseguía cuando se creó fue la modelar el **polimorfismo explícito**.

Como ejemplo, podemos notar en el sistema de tipos de Curry de primer orden

$$\lambda x \,.\, x : A \rightarrow A$$

esta lambda-expresión tiene el tipo A->A sea cual sea A, es decir, para un tipo A cualquiera. Pues bien, en el sistema propuesto, λ2**,** deberíamos escribir:

$$\lambda x \,.\, x : \forall A.\, A \rightarrow A$$

que se leería como que posee el tipo A->A *uniformemente* en A.

En el sistema axiomático lo que se hace es incorporar dos nuevas reglas que permiten la introducción y eliminación del cuantificador:

(ax) $\dfrac{}{\Gamma \vdash x : A}$ $\quad x: A \in \Gamma$

($\rightarrow$ e) $\dfrac{\Gamma \vdash f : A \rightarrow B \quad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B}$

($\rightarrow$ i) $\dfrac{\Gamma , x : A \vdash b : B}{\Gamma \vdash \lambda x .b : A \rightarrow B}$

($\forall$e) $\dfrac{\Gamma \vdash M: \forall A.B}{\Gamma \vdash M : \langle A: = C \rangle B}$

$$(\forall i) \qquad \frac{\Gamma \vdash M: B}{\Gamma \vdash M : \forall A . B} \qquad A \notin FV(\Gamma)$$

donde $FV(\Gamma)$ representa la unión de las variables libres de los tipos de cada asignación de tipo de Γ. Por ejemplo, si $\Gamma = \{ x_1 : tipo_1, ..., x_n: tipo_n \}$, entonces tenemos que $FV(\Gamma) = FV(tipo_1) \cup ... \cup FV(tipo_n)$, sabiendo además que el conjunto de variables de tipo libres se define de forma análoga al caso de las abstracciones.

II El Lenguaje Mercury

1. Introducción al lenguaje Mercury

1.1 Motivaciones para su desarrollo

Los lenguajes de programación lógica son teóricamente superiores a los lenguajes imperativos ya que permiten a los programadores centrarse en especificar *qué* hay que realizar (centrarse en la ingeniería del conocimiento), mientras que pueden olvidarse de los detalles de implementación y los aspectos de bajo nivel, es decir de *cómo* realizar el mecanismo de resolución del problema. Pero, a pesar de las cualidades de las que dotan a las aplicaciones desarrolladas (legibilidad, facilidad de mantenimiento, ...), la utilización de los lenguajes de programación lógica no se ha extendido en el ámbito industrial y comercial.

La mayoría de las aplicaciones actuales son tan complejas que es necesario de varios grupos de programadores para realizar todo el trabajo. Los lenguajes deberían, por ello, estar dotados de mecanismos para que dichos grupos pudieran cooperar. El método más efectivo para hacerlo es proporcionar un sistema de módulos que separe la interfaz de la implementación, de forma que los pequeños cambios en una parte de un programa no afecten a las otras partes.

Además, los programadores desean tener toda la ayuda posible por parte del compilador a la hora de localizar errores en las aplicaciones, ya que la fiabilidad es una característica crítica de una aplicación. Esto requiere un lenguaje de programación con información redundante, como la que producen las declaraciones de tipo y modo. El programador aporta mucha más información con las declaraciones, por lo que el compilador es capaz de realizar chequeos más exhaustivos, que le permiten detectar mayor cantidad de errores y ubicarlos en el programa de forma más exacta.

Otro requerimiento, a veces prioritario en aplicaciones industriales, es que la implementación produzca código razonablemente eficiente en espacio y tiempo; y que sea capaz de funcionar en multitud de plataformas, de forma que la aplicación no se vea limitada a máquinas concretas.

Los lenguajes de programación lógica actuales no cumplen estos requerimientos. Sólo unos pocos dialectos Prolog soportan módulos, y su sistema de módulos interactua mal con el resto del lenguaje. Además, la ayuda de la mayoría de compiladores existentes en el proceso de depuración es escasa, lo que implica invertir mucho tiempo para dotar de la suficiente fiabilidad a los programas; y el código resultante no está suficientemente optimizado, a causa de la falta de información que poseen.

Los creadores de Mercury intentaron añadir mecanismos para solventar esos problemas, sacrificando parte de la generalidad de la que hacen gala los lenguajes de programación lógica, en los que el tipo de todas las variables se conoce en tiempo de ejecución.

1.2 Características de Mercury

Mercury, es un lenguaje puramente declarativo enfocado al diseño de aplicaciones complejas, y aporta un conjunto de características gracias a las que compite con los lenguajes imperativos para las aplicaciones de ámbito industrial y comercial.

A continuación se detallarán las características principales de Mercury.

1.2.1 Todo predicado es puro

Mercury es un lenguaje puramente declarativo, lo que significa que los predicados y funciones en Mercury no tienen efectos laterales de tipo no-lógico. Es decir, conseguimos separar el conocimiento de un problema de la forma utilizada para resolverlo en todos los casos, y, con ello, aumentamos la fiabilidad del programa.

Los predicados no-lógicos de Prolog (corte, E/S, assert/retract) tendrán su contrapartida puramente declarativa en Mercury. En el caso del corte, por ejemplo, éste se mantiene por razones históricas, ya que en principio el compilador de Mercury estuvo escrito en Prolog-NU o Prolog-SCSI (actualmente el compilador de Mercury está escrito en Mercury).

Las operaciones de E/S se han encapsulado en predicados que tomarán el estado actual del mundo y producirán un nuevo estado, por lo que los cambios producidos por dichos predicados quedan visibles a través de dichos parámetros, que serán del tipo especial '`io_state`' y que se instanciarán con variables globales que representan al sistema.

De todas formas, Mercury proporciona mecanismos para definir predicados no puramente declarativos (impuros), que son necesarios en algunas ocasiones, tales como la generación de algunas de las primitivas del sistema; o la reutilización de código escrito en C (que no tiene que estar libre de efectos laterales). En cualquier caso, debe indicarse la impureza al compilador con las anotaciones adecuadas tanto en la declaración/definición del predicado como en cada utilización del mismo.

1.2.2 Sistema fuerte de tipos

Mercury es un lenguaje fuertemente tipado, por lo que todos los elementos de un programa tendrán un tipo en tiempo de compilación.

Permite polimorfismo paramétrico, así como clases de tipos con una estructura muy parecida a la de Haskell y proporciona un sistema de chequeo de tipos de predicados y funciones, y un sistema de inferencia de tipos para las variables de la aplicación, para lo cual el compilador necesita que se especifiquen los tipos de los parámetros de cada predicado y función. Para los argumentos de predicados y funciones locales a un módulo, el compilador intentará inferir su tipo si no están especificados, y, en caso, de no poder hacerlo rechazará el programa.

En las secciones 2-6 del bloque se trata con mayor detalle el sistema de tipos de Mercury.

1.2.3 Sistema de análisis de modos y de determinismo

Mercury presenta un sistema de modos fuerte, de forma que en tiempo de compilación quedan prefijados los modos de cada predicado o función empleados en el programa de entre todos los modos indicados por el usuario.

El programador debe declarar el estado de instanciación de los argumentos de los predicados en el momento de su llamada y el estado que tendrán cuando el predicado se satisfaga.

El compilador cuenta con un mecanismo de chequeo de modos, que, añadido al chequeo de tipos, captura multitud de errores que puedan haberse escapado al programador, y realiza mejoras en el código más eficientes, además de contribuir a la documentación de la aplicación.

Junto a los modos, Mercury posee un sistema de análisis del determinismo. Para cada modo de un predicado, el usuario declarará el tipo de determinismo que presenta, con lo que el compilador chequeará si es correcto o no.

Para predicados o funciones locales a un módulo, es decir, sólo utilizados en dicho módulo, el sistema de análisis de determinismo, permite inferir su determinismo, por lo que es posible obviar la declaración de determinismo.

En las secciones 7 y 8 del bloque se habla más detenidamente del sistema de modos y del determinismo.

1.2.4 Sistema de módulos

Los programas Mercury se componen de uno o más módulos. Éstos permiten definir una parte de interfaz, que contiene la declaración de los tipos, funciones y predicados que se exportarán; y una parte de implementación, con la definición de las entidades exportadas y locales.

Es posible utilizar tipos abstractos que sólo sean manipulados por los predicados o funciones definidas en el mismo módulo, con sólo indicar, en la sección de interfaz, la signatura de los predicados y el nombre de los tipos.

Se puede encontrar más información sobre la utilización de módulos en Mercury en la sección 9.

1.2.5 Lenguaje multiplataforma

Los desarrolladores de Mercury, se impusieron como requisito realizar un lenguaje que se pudiese compilar en la mayoría de las máquinas actualmente existentes en el mercado. Intentaron no sacrificar portabilidad a cambio de eficiencia, lo que consiguieron generando compiladores que no produjeran código máquina sino código en algún lenguaje extendido ampliamente.

El lenguaje escogido inicialmente como producto final de la compilación fue C, pues goza de una gran difusión en el ámbito informático, con lo que la presencia, o en

su defecto existencia, de un compilador de C está casi asegurada en la máquina en la que instalemos Mercury.

Con la aparición de nuevos lenguajes que facilitan la creación de aplicaciones que utilicen Internet, se está intentando desarrollar back-ends (engloban las fases del proceso de compilación independientes del lenguaje fuente) para el compilador que proporcionen un fichero escrito en dichos lenguajes. Así, podemos encontrar una versión beta de un compilador que genera código para la plataforma .NET de Microsoft, o una versión, en proceso de desarrollo, que producirá código para la máquina virtual Java.

Además, hay que mencionar que Mercury puede conseguir la portabilidad deseada interaccionando con CORBA, debido a que es esta aplicación (de tipo Middleware) la encargada de efectuar los cambios de protocolos y formatos necesarios para que los programas se ejecuten en sistemas heterogéneos. Para ello, se está desarrollando la interfaz MCORBA que permite definir componentes CORBA en Mercury, así como utilizar componentes creados en programas Mercury.

1.2.6 Nuevo modelo de ejecución

El modelo de ejecución de Mercury se ha diseñado partiendo de cero, sin basarse en la máquina abstracta de Warren (WAM), debido a que los autores de Mercury han intentado obtener un código más eficiente, utilizando la información proporcionada por el usuario en las declaraciones. Aunque, para ser sinceros, el resultado obtenido es muy parecido a una máquina abstracta de Warren, optimizada en ciertos aspectos.

El compilador, entre otras optimizaciones, diferenciará entre la ejecución determinista, semideterminista y no determinista. En el primer caso, generará funciones al más puro estilo imperativo, puesto que el resultado será único. En los dos casos siguientes, es necesario una pila auxiliar como complemento(la pila del no determinismo). El caso semideterminista utilizará menos espacio en esta pila, que el caso no determinista, para el que la ejecución está menos restringida y puede proporcionar cualquier número de resultados.

Las principales diferencias entre el modelo de ejecución de Mercury y WAM suelen estar provocadas por el afán de los desarrolladores de Mercury de obtener un código lo más eficiente posible sin dejar de cumplir los requisitos impuestos. Presentamos algunas a continuación:

- El modelo de ejecución de Mercury no tiene primitivas de unificación general. Los tipos y modos permiten al compilador generar código especializado para las unificaciones; códigos que no tienen equivalente al pasar al modo lectura/escritura de la máquina abstracta de Warren.

- Mercury tiene algoritmos separados para las partes deterministas y no deterministas, y combina los puntos de selección (choice points) y los entornos en una sola estructura de datos, los marcos de la pila no determinista. El modelo de ejecución clasifica los predicados en tres clases: predicados con una cláusula que contiene sólo llamadas a predicados deterministas; predicados con una cláusula que contiene alguna llamada a

predicados no determinista; y predicados con varias cláusulas. La máquina WAM no puede distinguir entre los dos primeros grupos, por lo que las optimizaciones realizadas son menores.

- Mercury evita mantener el rastro de las referencias. En cada momento, se sabe exactamente el estado de instanciación de las variables, por lo que las actualizaciones destructivas de las variables sólo se producirán en aquellos casos deterministas en los que no volveremos a usar esos valores.

- El modelo de ejecución de Mercury no necesita derreferenciación. El sistema de modos se asegura que los productores ocurrirán antes que los consumidores, por lo que los valores están siempre disponibles cuando son necesitados. Esto evita, en muchísimos casos, la indirección.

- WAM permite la optimización de funciones recursivas con llamadas de cola de una forma más sencilla. Esto se debe a que el algoritmo de análisis de modos reordena los componentes de la cláusula, por lo que la llamada recursiva de cola suele dejar de ser la última acción de la cláusula. Mercury subsana el problema, con una pequeño cambio del modelo de ejecución.

1.2.7 Proporciona cualidades funcionales

En Mercury existe la posibilidad de definir predicados y funciones de orden superior, es decir, funciones y predicados cuyos argumentos (y resultado para el caso de las funciones) puedan ser otros predicados o funciones. También se pueden especificar expresiones lambda o, lo que es lo mismo, funciones o predicados para los que no se da un nombre, así como ligar variables al valor representado por funciones o predicados con argumentos parcialmente instanciados (currificación).

Teniendo en cuenta que la forma en que el modelo de ejecución tratará las funciones dependerá del determinismo indicado para estas (serán funciones totales si se indica determinismo, y parciales si se declara semideterminismo), entonces si se declaran todos los argumentos de la función como de entrada, y sólo, el resultado como salida, se podría simular, más o menos, la forma de trabajar con un lenguaje funcional en el que todos las entidades del programa se pueden ver como funciones. Sin embargo, a nivel interno, el compilador de Mercury transforma las funciones de n argumentos, en predicados con n+1 argumentos (los n argumentos y otro argumento que representa al resultado).

2. Tipos

Vamos a explicar algunos aspectos de cómo se programa con el lenguaje de programación declarativo con tipos Mercury, pero más concretamente en lo que se refiere a los tipos, aspectos como qué hacer para declarar un tipo, qué hacer para declarar tipos de funciones y predicados, modos, declaración de tipos clase, etc.

2.1 Construcción de tipos

Algunos tipos que ya vienen definidos por Mercury son:

- *Char, int, float, string.*
- Tipos predicados: *pred,pred(T),pred(T1,T2),...*
- Tipos función: *(func)=T,func(T1)=T,func(T1,T2)=T,...*
- Tipos tupla: *{},{T}, {T1, T2},...*
- Tipo Universal: *univ*.
-

3. Tipos definidos por el usuario

Se pueden declarar nuevos tipos utilizando ':- type'. Hay varias formas de hacerlo:

3.1 Discriminated Unions

Un tipo derivado se define: ':- type type'--->body'

type' define el nuevo tipo que se va a crear, y *body* define el esquema que siguen todos los términos que pertenecen a ese tipo. *body* es una secuencia de constructores separados por ';', y estos constructores también pueden tener una serie de argumentos, que llamaremos campos, argumentos que deberán estar etiquetados. Estas etiquetas hace que el compilador genere funciones que se pueden usar para seleccionar y actualizar los campos de un término (campos que serán del tipo de los argumentos de los constructores). Una etiqueta sigue el siguiente patrón: *::Type*. No se puede poner la misma etiqueta a dos campos en un mismo módulo.

Ejemplos:

```
:- type fruit
        --->    apple
        ;       orange
        ;       banana
        ;       pear.

:- type strange
        --->    foo(int)
        ;       bar(string).

:- type employee
        --->    employee(
                        name        :: string,
                        age         :: int,
                        department  :: string
                ).

:- type tree
        --->    empty
        ;       leaf(int)
        ;       branch(tree, tree).

:- type list(T)
        --->    []
        ;       [T | list(T)].

:- type pair(T1, T2)
        --->    T1 - T2.
```

Puesto que ‘;’ se utiliza para separar a los constructores, es difícil definir un constructor que sea ‘;’. Para permitirlo, se hace como se muestra en el ejemplo:

```
:- type tricky
        --->    { int ; int }
        ;       { { int } }.
```

Esto define dos constructores, ;/2 y {}/1 , cuyos tipos de argumentos son int.

Cada unión discriminada es un tipo distinto. Mercury considera que dos tipos de uniones discriminadas que tienen el mismo cuerpo son distintos. Tener dos definiciones diferentes de un tipo con el mismo nombre e igual aridad en el mismo módulo es un error.

Los constructores se pueden sobrecargar: pueden haber varios con el mismo nombre y aridad, pero con tipos diferentes. Sin embargo, no puede haber más de un constructor con el mismo nombre, aridad y tipo resultado en el mismo modulo.

Ejemplo:

Bien: `fruta1--->platano(int)  fruta2--->platano(int)`

Mal: `fruta--->platano(int);platano(float)`, aunque se supone que no sería el mismo constructor pues el tipo de los argumentos no es el mismo.

3.2 Equivalencia de tipos

Se hace utilizando ==. Por ejemplo

```
:- type money == int.
:- type assoc_list(KeyType, ValueType)
        == list(pair(KeyType, ValueType)).
```

Así es como se renombran los tipos. La definición no debe ser cíclica, es decir, no debe aparecer el mismo nombre de tipo a la izquierda y derecha de ==.

3.3 Tipos Abstractos

Son tipos cuya implementación está oculta.

```
:- type t1.
:- type t2(T1, T2).
```

Se define t1/0 y t2/0 como tipos abstractos. Tales declaraciones son sólo útiles en la sección de interfaz de un módulo. Esto significa que los nombres de tipos se exportarán, pero los constructores (functors) para estos tipos no se exportarán. Todos los módulos que los importen deberán dar una definición de todos los tipos abstractos que importan.

4. Declaración de tipos de funciones y predicados

Los tipos de los argumentos de cada predicado deben estar declarados mediante ‘:- pred’. El tipo de argumento y el tipo resultado de cada función debe estar declarado explícitamente con una declaración ‘:- func’. Por ejemplo:

```
:- pred is_all_uppercase(string).
:- func strlen(string) = int.
```

Sólo puede haber un predicado con un mismo nombre y aridad en cada módulo, y sólo una función con un mismo nombre y aridad en cada módulo. Es un error declarar la misma función o predicado dos veces.

El compilador infiere los tipos de los términos, y en particular los tipos de las variables y los constructores, funciones y predicados sobrecargados. Una asignación de tipos es una asignación de tipos a cada variable y función, constructor o predicado. Una asignación de tipos es válida si verifica:

- Cada constructor de una cláusula debe haber sido declarado al menos en una declaración de tipos visible. El tipo asignado a cada término constructor debe coincidir con una de las declaraciones de tipos para ese constructor, y los tipos asignados para cada argumento debe coincidir con los tipos de los argumentos de la declaración de tipos.

- El tipo asignado a cada término de llamada de función debe coincidir con el tipo del resultado de una de las declaraciones ‘:- func’ para esa función, y los tipos asignados a los argumentos de esa función debe coincidir con los tipos especificados en la declaración de tipos.

- El tipo asignado a cada argumento de predicado debe coincidir con el tipo especificado en una de las declaraciones ‘:- pred’ para ese predicado. El tipo asignado a cada argumento de la cabecera de la cláusula de predicado debe coincidir exactamente con el tipo de argumento especificado en la correspondiente declaración ‘:- pred’.

- El tipo asignado a cada argumento de la cabecera en una cláusula de función debe coincidir exactamente con el tipo de argumento especificado en la correspondiente declaración ‘:- func’, y el tipo asignado al término resultado en una cláusula de función debe coincidir exactamente con el tipo resultado especificado en la declaración ‘:- func’.

- El tipo asignado a cada término con un tipo cualificación debe coincidir con el tipo especificado por el tipo expresión de cualificación.

- Una asignación de tipo A se dice ser más general que otra B si hay una combinación de los tipos de los parámetros de A que la hace idéntica a B. Si hay más de una asignación de tipos válida, el compilador debe elegir la más general. Si hay dos asignaciones de tipos válidas que no son idénticas y ninguna es más general que la otra, entonces hay una ambigüedad de tipos, y el compilador generará un error. Una cláusula es de tipo correcto si la asignación de tipos más general es única. Cada cláusula en un programa Mercury debe ser de tipo correcto.

5. Funciones de acceso a los campos

Los campos de los constructores de los tipos uniones discriminadas deben estar etiquetados. Estas etiquetas hacen que el compilador genere funciones que pueden consultar y actualizar los campos de un término de manera independiente de la definición del tipo.

5.1 Selección de campo

field(Term)

Cada campo etiquetado ‘field’ en un constructor hace que se genere una función de selección de campo ‘field/1’, la cual toma un término del mismo tipo que el constructor y devuelve el valor del campo etiquetado.

Si la declaración se hace en la sección de intefaz del módulo, la correspondiente función de selección también se exporta desde el módulo.

Por defecto, esta función no tiene declaraciones de modo – los modos se infieren en cada llamada a la función. Sin embargo, el tipo y los modos de esta función se

pueden declarar explícitamente, en cuyo caso se tendrá únicamente los modos declarados.

5.2 Actualizar campo

'field:='(Term,ValueTerm)

Cada campo etiquetado con 'field' en un constructor genera una función de actualización ''field:='/2'. El primer argumento de esta función es un término del mismo tipo que el constructor. El segundo argumento es un término del mismo tipo que el campo etiquetado. El valor de retorno es una copia del primer argumento en el que se ha sustituido el campo etiquetado por el segundo argumento. ''field:='/2' falla si el constructor del primer argumento no es el constructor que contiene el campo etiquetado.

Si la declaración del campo se hace en la sección de interfaz del módulo, la correspondiente función de actualización también se exporta desde el módulo.

Por defecto, esta función no tiene modos declarados – el modo se infiere en cada llamada a la función. Sin embargo, el tipo y los modos de esta función se pueden declarar explícitamente, en cuyo caso tendrá sólo los modos declarados.

Algunos campos no se pueden actualizar usando funciones de actualización de campos. Para el constructor 'unsettable/2', ningún campo se puede actualizar porque el término resultante no estaría bien tipado.

```
:- type unsettable
        ---> some [T] unsettable(
                unsettable1 :: T,
                unsettable2 :: T
        ).
```

5.3 Ejemplo de acceso a campos

```
:- type type1
        ---> type1(
                field1 :: type2,
                field2 :: string
        ).

:- type type2
        ---> type2(
                field3 :: int,
                field4 :: int
        ).
```

El compilador genera algunas funciones de acceso a field1. Las funciones generadas para acceder a los otros campos son similares

```
:- func field1(type1) = type2.
field1(type1(Field1, _)) = Field1.

:- func 'field1 :='(type1, type2) = type1.
'field1 :='(type1(_, Field2), Field1) = type1(Field1, Field2).
```

El programador también puede escribir código como:

```
:- func increment_field3(type1) = type1.

increment_field3(Term0) =
        Term0 ^ field1 ^ field3 := Term0 ^ field1 ^ field3 + 1.
```

código que el compilador expande a:

```
incremental_field3(Term0) = Term :-
        OldField3 = field3(field1(Term0)),

        OldField1 = field1(Term0),
        NewField1 = 'field3 :='(OldField1, OldField3 + 1),
        Term = 'field1 :='(Term0, NewField1).
```

utilizando las funciones anteriormente descritas.

6. Tipo clases

Mercury soporta polimorfismo en la forma de tipos clases. El tipo clase permite al programador escribir predicados y funciones que operan sobre variables de cualquier tipo para las cuales se define un conjunto de operaciones.

6.1 Declaraciones Typeclass

Un *typeclass* es un nombre para un conjunto de tipos (o conjunto de secuencia de tipos) para los cuales ciertos predicados y/o funciones, llamadas los métodos de ese tipo clase, se definen. Una declaración 'typeclass' define un nuevo tipo de clase, y especifica el conjunto de predicados y/o funciones que se deben definir en un tipo o secuencia de tipos para ser considerados como una instancia de ese tipo clase.

La declaración *typeclass* da el nombre del tipoclase que se está definiendo, los nombres de las variables de tipos que son parámetros y las operaciones (es decir, los métodos) que forman parte del tipo clase.

Por ejemplo:

```
:- typeclass point(T) where [
        % coords(Point, X, Y):
        %       X and Y are the cartesian coordinates of Point
        pred coords(T, float, float),
        mode coords(in, out, out) is det,

        % translate(Point, X_Offset, Y_Offset) = NewPoint:
        %       NewPoint is Point translated X_Offset units in the
X direction
        %       and Y_Offset units in the Y direction
        func translate(T, float, float) = T
].
```

Esto declara el tipo *class point*, que representa puntos en el espacio bidimensional.

Las declaraciones pred, func y mode son las únicas declaraciones legales dentro de una declaración typeclass. El número de parámetros del tipo clase no está limitado. Por ejemplo, se permite poner:

```
:- typeclass a(T1, T2) where [...].
```

Los parámetros deben ser variables distintas. Cada declaración typeclass debe tener al menos un parámetro.

Se puede no declarar métodos dentro de una declaración de tipo clase, por ejemplo:

```
:- typeclass foo(T) where [].
```

No debe haber más de una declaración de tipo clase con el mismo nombre y aridad en el mismo módulo.

6.2 Declaraciones de Instancias

Una vez declarada la interfaz del tipo clase, podemos usar una declaración instancia para definir cómo un tipo particular satisface el interfaz declarado en la declaración typeclass.

Una declaración de instancia es de la forma:

```
:- instance classname(typename(typevar, ...), ...)
        where [methoddefinition, methoddefinition, ...].
```

Una declaración de instancia da un tipo a cada parámetro del tipo clase. Cada uno de estos tipos debe ser un tipo sin argumentos o un tipo polimórfico cuyos argumentos son todos variables de tipos distintas. Por ejemplo *int*, *list(T)* y *bintree(K,V)* se permiten, pero *T, list(int)* y *bintree(T,T)* no. Los tipos en una declaración de instancia no deben ser tipos abstractos. Un programa puede que no contenga más de una declaración de instancia para un tipo particular y tipo clase.

Cada definición de método que hay en *where[...]* define la implementación de uno de los métodos de clase para esta instancia. Hay dos formas de definir métodos. La primera forma es definir un método dando el nombre del predicado o función que implementa el método. En este caso, la definición de método es de una de las formas siguientes:

```
pred(methodname/arity) is predname
func(methodname/arity) is funcname
```

predname y *funcname* deben referirse a una función o predicado con aridad especificada por *arity*, cuyos tipos, modos, determinismo y pureza son al menos tan permisivo como los tipos, modos, determinismo y pureza del método de clase al que se refiere *methodname* con aridad *arity.*

Otra forma de definir métodos es haciéndolo dentro de la declaración de instancia. Una definición de método puede ser una cláusula. Estas cláusulas son justamente como las cláusulas usadas para definir predicados o funciones ordinarias, y también pueden ser hechos, reglas o DCG reglas. La única diferencia es que en la declaración de instancias, las cláusulas están separadas por comas mejor que por ‘;’, y también que las reglas y DCG reglas en la declaración de instancias deben estar encerradas en paréntesis. Como en los predicados ordinarios, puedes tener más de una cláusula para cada método.

Estas dos formas son mutuamente exclusivas: cada método debe estar definido de una forma o de otra, pero no de ambas.

Ejemplo de las múltiples formas de definir un método:

```
:- typeclass foo(T) where [
        func method1(T, T) = int,
        func method2(T) = int,
        pred method3(T::in, int::out) is det,
        pred method4(T::in, io__state::di, io__state::uo) is
det,
        func method5(bool, T) = T
].

:- instance foo(int) where [
        % method defined by naming the implementation
        func(method1/2) is (+),

        % method defined by a fact
        method2(X) = X + 1,

        % method defined by a rule
        (method3(X, Y) :- Y = X + 2),

        % method defined by a DCG rule
        (method4(X) --> io__print(X), io__nl),

        % method defined by multiple clauses
        method5(no, _) = 0,
        (method5(yes, X) = Y :- X + Y = 0)
].
```

Cada declaración instancia debe definir una implementación para cada método declarado en la declaración *typeclass*. Es un error definir más de una implementación para el mismo método dentro de una declaración ‘instance’.

En cada llamada a un método los tipos de los argumentos deben ser miembro del tipo clase a la que pertenece ese método, o coincidir con una de las declaraciones de instancias visibles en el punto de la llamada. Una llamada a un método invocará el predicado o la función especificada para ese método en la declaración de instancia que tenga tipos que coincidan con los tipos de los argumentos de la llamada.

```
:- type coordinate
        ---> coordinate(
                float,          % X coordinate
                float           % Y coordinate
        ).

:- instance point(coordinate) where [
        pred(coords/3) is coordinate_coords,
        func(translate/3) is coordinate_translate
].


:- pred coordinate_coords(coordinate, float, float).
:- mode coordinate_coords(in, out, out) is det.

coordinate_coords(coordinate(X, Y), X, Y).

:- func coordinate_translate(coordinate, float, float) = coordinate.

coordinate_translate(coordinate(X, Y), Dx, Dy) = coordinate(X + Dx, Y + Dy).
```

Si introducimos un nuevo tipo, *coloured_coordinate* que representa un punto en el plano bidimensional con un color asociado, también puede ser una instancia del tipo clase:

```
:- type rgb
        ---> rgb(
                int,
                int,
                int
        ).

:- type coloured_coordinate
        ---> coloured_coordinate(
                float,
                float,
                rgb
        ).

:- instance point(coloured_coordinate) where [
        pred(coords/3) is coloured_coordinate_coords,
        func(translate/3) is coloured_coordinate_translate
].


:- pred coloured_coordinate_coords(coloured_coordinate, float, float).
:- mode coloured_coordinate_coords(in, out, out) is det.

coloured_coordinate_coords(coloured_coordinate(X, Y, _), X, Y).

:- func coloured_coordinate_translate(coloured_coordinate, float, float)
        = coloured_coordinate.

coloured_coordinate_translate(coloured_coordinate(X, Y, Colour), Dx, Dy)
        = coloured_coordinate(X + Dx, Y + Dy, Colour).
```

Si llamamos a ‘translate/3’ con un tipo ‘coloured_coordinate’ como primer argumento, invocará ‘coloured_coordinate_translate’. Si llamamos a ‘translate/3’ con tipo ‘coordinate’ como primer argumento, esto llamará a ‘coordinate_translate’.

6.3 Declaraciones de Instancias ABSTRACTAS

Consiste en una declaración de instancia cuya implementación está oculta. Una declaración de instancia abstracta es de la misma forma que una declaración de instancia, pero sin ‘where [...]’. Funciona igual que una declaración de instancia normal, pero sin definir la implementación de los métodos. Cada declaración de instancia abstracta debe estar acompañada por la correspondiente declaración de instancia no abstracta, es decir, se deberá completar su definición en el módulo que la usa. Ejemplo:

```
:- module hashable.
:- interface.
:- import_module int, string.

:- typeclass hashable(T) where [func hash(T) = int].
:- instance hashable(int).
:- instance hashable(string).

:- implementation.

:- instance hashable(int) where [func(hash/1) is hash_int].
:- instance hashable(string) where [func(hash/1) is hash_string].

:- func hash_int(int) = int.
hash_int(X) = X.

:- func hash_string(string) = int.
hash_string(S) = H :-
        % use the standard library predicate string__hash/2
        string__hash(S, H).

:- end_module hashable.
```

6.4 Restricciones de tipo clase sobre predicados y funciones

Mercury permite una restricción de tipo clase como parte de un tipo función o predicado. Esto restringe los valores que pueden tomar las variables de tipo.

Una restricción de tipo clase es de la forma:

```
<= Typeclass(TypeVariable, ...), ...
```

Typeclass es el nombre de un tipo clase y *TypeVariable* es una variable de tipo que aparece en la signatura del tipo predicado o función:

```
:- pred distance(P1, P2, float) <= (point(P1), point(P2)).
:- mode distance(in, in, out) is det.

distance(A, B, Distance) :-
        coords(A, Xa, Ya),
        coords(B, Xb, Yb),
        XDist = Xa - Xb,
        YDist = Ya - Yb,
        Distance = sqrt(XDist*XDist + YDist*YDist).
```

En el ejemplo, el predicado distancia puede calcular la distancia entre cualquiera dos puntos, sin tener en cuenta la representación. Estas restricciones se chequean en tiempo de compilación.

6.5 Restricciones de tipo clase sobre declaraciones de tipo clase

Las restricciones de tipo clase también aparecen en las declaraciones tipoclase, significando que un tipo clase es superclase de otro.

Las variables que aparecen como argumento de los tipos clase en las restricciones también tienen que ser argumentos del tipo clase en cuestión.

Por ejemplo, lo siguiente declara el tipo clase ring, que describe tipos con un conjunto particular de operaciones numéricas definidas:

```
:- typeclass ring(T) where [
        func zero = (T::out) is det,                % '+' identity
        func one = (T::out) is det,                 % '*' identity
        func plus(T::in, T::in) = (T::out) is det, % '+'/2 (forward mode)
        func mult(T::in, T::in) = (T::out) is det, % '*'/2 (forward mode)
        func negative(T::in) = (T::out) is det     % '-'/1 (forward mode)
].
```

Ahora podemos añadir la siguiente declaración:

```
:- typeclass euclidean(T) <= ring(T) where [
        func div(T::in, T::in) = (T::out) is det,
        func mod(T::in, T::in) = (T::out) is det
].
```

Esto introduce un nuevo tipo clase, euclidean, de la cual *ring* es una superclase. La operación definida por el tipo clase *euclidean* son *div, mod*. Cualquier tipo declarado para ser una instancia de *euclidean* también debe ser declarado para ser una instancia de *ring*.

6.6 Restricciones de tipo clase sobre declaraciones de instancias

Las restricciones de tipo clase también se pueden hacer sobre declaraciones de instancias. Las variables que aparecen como argumentos al tipo clase en las restricciones deben ser todas variables de tipo que aparecen en los tipos de las declaraciones de instancia.

Por ejemplo, considere la siguiente declaración de tipo clase:

```
:- typeclass portrayable(T) where [
        pred portray(T::in, io__state::di, io__state::uo) is
det
].
```

El programador podría declarar instancias como:

```
:- instance portrayable(int) where [
        pred(portray/3) is io__write_int
].

:- instance portrayable(char) where [
        pred(portray/3) is io__write_char
].
```

Sin embargo, cuando escribimos declaraciones de intancias para tipos como list(T), queremos poder escribir los elementos de la lista usando *portray/3* para los tipos particulares de los elementos de la lista. Esto se puede hacer poniendo una restricción de tipo clase sobre la declaración de instancia, como en el siguiente ejemplo:

```
:- instance portrayable(list(T)) <= portrayable(T) where [
        pred(portray/3) is portray_list
].

:- pred portray_list(list(T), io__state, io__state) <= portrayable(T).
:- mode portray_list(in, di, uo) is det.

portray_list([]) -->
        [].
portray_list([X|Xs]) -->
        portray(X),
        io__write_char(' '),
        portray_list(Xs).
```

Para declaraciones abstractas de instancias, la restricción de tipo clase sobre la declaración de instancia abstracta debe coincidir exactamente con la restricción del tipo clase sobre la correspondiente declaración de instancia no abstracta que define esa instancia.

7. Modos en Mercury

7.1 Estados de instanciación

El estado de instanciación de una variable, o modo de una variable, indica la forma en que la estructura representada por dicha variable está instanciada.

Para describir los estados de instanciación, utilizamos la información proporcionada por el sistema de tipos. Los tipos pueden ser vistos como árboles regulares con dos tipos de nodos: los nodos OR, que representan los tipos y los nodos AND, que representan los constructores. Los hijos de un nodo OR son los constructores que pueden usarse para construir términos de ese tipo; los hijos de un nodo AND son los tipos de los argumentos de los constructores.

Un árbol de instanciación (*instantiatedness tree*) es un árbol de tipos a cuyos nodos OR se ha añadido una etiqueta que representa la instanciación de dicho nodo

(libre o ligado), con la restricción de que todos los descendientes de un nodo libre deben estar etiquetados como libres.

Un término es *aproximado* por un árbol de instanciación si para cada nodo del árbol se cumple:

- Si el nodo está libre, entonces el correspondiente nodo en el término es una variable libre que no está compartida con otra variable(variables distintas).
- Si el nodo está ligado, el correspondiente nodo en el término es un símbolo de función.

Cuando un árbol de instanciación nos dice que una variable está ligada, existen muchos símbolos de función que podrían ligar dicha variable. El árbol de instanciación no nos dice a cuál de ellos está ligada; en cambio para cada posible símbolo de función indica exactamente cuántos argumentos del símbolo de función estarán libres y cuántos ligados. El mismo principio se aplica recursivamenta a los argumentos ligados.

Mercury permite a los usuarios declarar nombres para los árboles de instanciación utilizando declaraciones como:

```
:- inst listskel == bound( [] ; [free | listskel] ).
```

Este árbol de instanciación describe listas cuyo esqueleto es conocido, pero cuyos elementos son variables distintas. Por ello, aproxima el término [A,B] pero no [H|T] (sólo una parte del esqueleto es conocida), ni [A,2] (no todos los elementos son variables), ni [A,A] (los elementos no son variables distintas).

También, es posible definir estados de instanciación parametrizados:

```
:- inst listskel(Inst) == bound( [] ; [Inst | listskel(Inst)] ).
```

Mercury impone la restricción de que no debe haber más de una definición de estado de instanciación con el mismo nombre y aridad en el mismo módulo.

El sistema de modos proporciona '`free`' y '`ground`' como nombres para árboles de instanciación con todos los nodos libres y ligados respectivamente. La forma de estos árboles está determinada por el tipo de la variable a la cual se apliquen.

7.2 Transformaciones de estados de instanciación.

Conforme la ejecución de una aplicación avanza, las variables se van instanciando más. Una transformación de modo (o de estado de instanciación) es una transformación desde un árbol de instanciación inicial a un árbol de instanciación final, con la restricción de que ningún nodo del árbol cambia de ligado a libre. Mercury permite al usuario definir estas transformaciones utilizando el operador '>>'.

Ejemplo:

```
:- mode m == inst1 >> inst2.
```

Mercury aporta dos transformaciones predefinidas, correspondientes a las nociones estándar de entrada (argumentos que deben leerse) y salida(argumentos que deben rellenarse con algún valor):

```
:- mode in == ground >> ground.
:- mode out == free >> ground.
```

Estos modos son suficientes para la mayoría de las funciones y predicados. Sin embargo, el sistema de modos de Mercury es suficientemente expresivo como para manejar patrones más complejos, incluyendo aquellas estructuras con datos parcialmente instanciados, aunque ésta característica no está implementada en el compilador actual de Mercury.

Los estados de instanciación y las transformaciones de modo, definidas por los operadores '`inst`' y '`mode`' respectivamente pueden tener parámetros. La librería estándar provee al usuario de los siguientes:

```
:- mode in(Inst) == Inst >> Inst.
:- mode out(Inst) == free >> Inst.
```

Al igual que para los estados de instanciación, no debe haber más de una definición de modo con el mismo nombre y aridad en el mismo módulo.

7.3 Modos únicos

Los estados de instanciación únicos permiten especificar cuándo hay sólo una referencia a un valor particular, y cuando no habrá más referencias a dicho valor, teniendo en cuenta que un valor no se considerará único si podría ser necesitado en el proceso de backtracking.

Si el compilador sabe que no habrá más referencias a un valor, puede, en tiempo de compilación, insertar código para liberar el espacio asociado a dicho valor o, mejor aún, reutilizar dicho espacio inmediatamente. Los modos únicos se emplean también para proporcionar predicados puramente declarativos de entrada/salida.

Aparte de los estados de instanciación mencionados anteriormente ('`free`', '`ground`' y '`bound()`'), Mercury también proporciona '`unique`' y '`unique()`', que son parecidos a '`ground`' y '`bound()`', con la diferencia de que en ellos se cumple la restricción de que sólo puede haber una referencia al correspondiente valor. También existe el estado de instanciación '`dead`', que indica que no hay más referencias al

```
% salida única
:- mode uo == free >> unique.

% entrada única
:- mode ui == unique >> unique.

% entrada destructiva
:- mode di == unique >> dead.
```

correspondiente valor, de forma que el compilador es libre para generar código que reutilice dicho valor.

Se presentan tres modos estándar predefinidos para la manipulación de valores únicos:

El modo '`uo`' se emplea para crear un valor único, '`ui`' para inspeccionar un valor único, sin perder la característica de ser único; mientras que '`di`' es usado para liberar o reutilizar la memoria ocupada por un valor que no se utilizará más.

7.4 Modo de un predicado

El *modo de un predicado*, o función, es una transformación del estado de instanciación inicial de los argumentos del predicado, o de los argumentos y el resultado de la función, a su estado de instanciación final.

Con una declaración de modo de un predicado (función) el programador está realizando el siguiente aserto:

Para todos los posibles términos argumento (y términos resultado) que son aproximados por los árboles de instanciación iniciales de la declaración de modo, y en los que todas las variables libres son distintas; si el predicado (función) acaba con éxito, entonces las ligaduras resultantes para los términos argumentos (y términos resultado) estarán aproximadas por los árboles de instanciación finales de la declaración de modo, donde se vuelve a cumplir que todas las variables libres son distintas. A este tipo de restricciones los desarrolladores de Mercury las denominan *restricciones de declaración de modo.*

Una declaración de modo de un predicado asigna una transformación de modo para cada argumento del predicado, mientras que una declaración de modo de una función asigna una transformación de modo a cada argumento de la función y una al resultado de la función.

Ejemplo:

```
:- pred append(list(T), list(T), list(T)).
:- mode append(in, in, out).
:- mode append(out, out, in).
```

Si sólo existe un modo, las declaraciones de modo y tipo pueden combinarse en una sola declaración, como muestra el siguiente ejemplo:

```
:- pred append(list(T)::in, list(T)::in, list(T)::out).
```

Las funciones, al igual que los predicados, pueden tener más de un modo. Si no se especifica ningún modo para una función, el compilador asume un modo por defecto en el que todos los argumentos tienen modo '`in`' y el resultado modo '`out`'.

En Mercury siempre es posible invocar a un procedimiento con un argumento que está más instanciado de lo que el árbol de instanciación inicial de la declaración de modo especifica. En esos casos, el compilador insertará unificaciones adicionales para asegurarse de que el argumento actualmente pasado al procedimiento tendrá el modo adecuado, apareciendo de esta manera modos que no habíamos declarados (*modos implícitos*)

Por ejemplo, si el predicado '`p/1`' tiene el modo '`p(out)`', podíamos efectuar la llamada '`p(X)`' si `X` estuviera ligado. El compilador entonces cambiaría el código a '`p(Y), X=Y`' donde `Y` es una variable nueva. Es como si el predicado '`p/1`' tuviese el modo implícito '`p(in)`'.

Diremos, por tanto, que un término *satisface* un árbol de instanciación si para cada nodo del árbol de instanciación se cumple:

- Si el nodo está libre, el correspondiente nodo en el término es o un símbolo de función o una variable libre distinta.
- Si el nodo está ligado, el correspondiente nodo del término es un símbolo de función.

El conjunto de modos para un predicado o función es el conjunto de declaraciones de modos para dicho predicado o función. Un conjunto de modos es una indicación del programador de que el predicado sólo debería ser llamado con términos argumento que satisfagan el estado de instanciación inicial de una de las declaraciones de modos del conjunto (que engloba las declaraciones explícitas e implícitas). Esta restricción se conoce como *restricción del conjunto de modos*.

7.5 Chequeo de modos

Se dice que un predicado o función p tiene un *modo correcto* (well-moded) *con respecto a una declaración de modo* dada si, sabiendo que los predicados y funciones llamados por p todos satisfacen sus restricciones de declaración de modo, existe un orden de los literales de la definición de p tal que:

- p satisface las restricciones de declaración de modo.
- P satisface las restricciones del conjunto de modos para todos los predicados y funciones a los que llama.

Un predicado o función tendrá un *modo correcto*, si tiene un modo correcto respecto a todas declaraciones de modo de su conjunto de modos. De esta manera, un programa tendrá un modo correcto si todos los predicados y funciones que contiene tienen un modo correcto.

El algoritmo de análisis de modo chequea un modo cada vez. Abstractamente interpreta la definición del predicado o función, guardando las instanciaciones de cada variable, y seleccionando un modo para cada llamada y unificación de la definición.

Para asegurarse de que las restricciones del conjunto de modos de los predicados y funciones llamados se satisfacen, el compilador puede reordenar los elementos de las conjunciones. Se comunicará un error si no existe un orden satisfactorio. Finalmente chequea que la instanciación resultante para los argumentos en ese modo es la misma que la dada en la declaración de modo.

8. Determinismo en Mercury

8.1 Clases de determinismo

Para cada modo de un predicado o función, clasificamos ese modo de acuerdo a cuántas veces puede satisfacerse (cuántos resultados puede proporcionar), y si puede fallar o no antes de producir su primera solución:

- Si todas las posibles llamadas a un modo particular de un predicado o función tienen exactamente una solución, entonces ese modo es *determinista* (`det`).

- Si todas las posibles llamadas a un modo particular de un predicado o función no tienen solución, o bien tienen sólo una solución, entonces el modo es *semideterminista* (`semidet`).

- Si todas las posibles llamadas a un modo particular de un predicado o función tienen al menos una solución, entonces el modo es *multisolución* (`multi`).

- Si algunas llamadas a un modo particular de un predicado o función no tienen solución, pero otras pueden tener una o más soluciones, entonces el modo es *no determinista* (`nondet`).

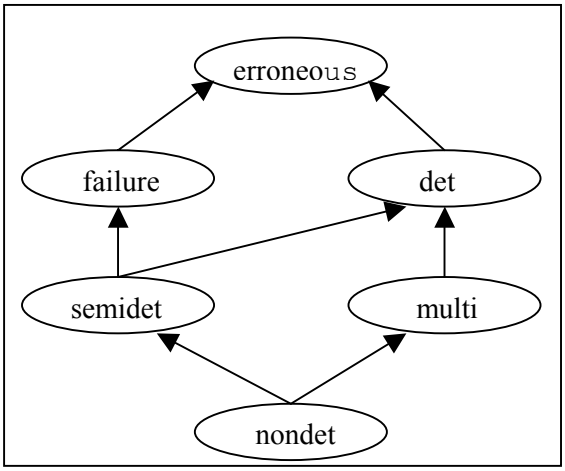
- Si todas las posibles llamadas a un modo particular de un predicado o función, fallan sin producir una solución, el modo tiene un determinismo de tipo *fallo* (`failure`).

- Si todas las posibles llamadas a un modo particular de un predicado o función producen un error en tiempo de ejecución, esto es, ni se satisfacen ni fallan; entonces el modo tiene determinismo de tipo *erróneo* (`erroneous`).

Las anotaciones ‘`det`’ o ‘`multi`’ se pueden considerar asertos que indican que para cada combinación de valores de las entradas, habrá, al menos, una combinación de valores de salida para la cual el predicado es verdad, o, en el caso de las funciones, para la cual el término función es igual al término resultado. De igual forma, una anotación ‘`det`’ o ‘`semidet`’ es una restricción que informa de que, para cada combinación de valores de las entradas, existirá, a lo sumo, una combinación de valores de salida para la cual el predicado se satisface o, en el caso de funciones, para la que el término función es igual al término resultado. Este tipo de imposiciones se conocen como *restricciones*

sobre el determinismo del modo, y en ciertas situaciones permiten realizar optimizaciones en la implementación, que de otra forma sería imposible conseguir.

A continuación se presenta la jerarquía formada por los tipos de determinismo. Cuanto más arriba se encuentre el tipo de determinismo más restrictivo será, debido a lo cual el compilador tendrá más información.



El determinismo de cada modo de un predicado o función se indica mediante una anotación que se añade a la declaración de modo; o a la declaración de tipo en el caso de que el modo y tipo se indiquen con una sola declaración, o el predicado no tenga argumentos.
Ejemplos:

```
:- pred append(list(T), list(T), list(T)).
:- mode append(in, in, out) is det.
:- mode append(out, out, in) is multi.
:- mode append(in, in, in) is semidet.

:- pred p is det.
p.
```

En Mercury, se supone que una función es una función de sus argumentos según el concepto matemático. Por ello, el valor del resultado de la función debería estar determinado sólo por los valores de sus argumentos, con lo que para cada modo de una función que especifique que todos los argumentos son de entrada (‘`ground`’), el determinismo sólo podrá ser ‘`det`’ (función total), ‘`semidet`’ (función parcial), ‘`erroneous`’ o ‘`failure`’. En caso de que no se especifique el modo (y con ello, tampoco el determinismo) de una función se supondrá que la función es determinista.

Parece interesante mencionar que Mercury proporciona versiones ‘*commited choice’* para el determinismo ‘`multi`’ y ‘`nondet`’, llamadas ‘`cc_multi`’ y ‘`cc_nondet`’. Estas se usarán si todas las llamadas a un modo del predicado o función ocurren en una situación en la que sólo una solución es necesitada, como podría ser el caso de cualquier objetivo sin variables de salida. Las razones para utilizar éstas anotaciones son las dos siguientes: para conseguir mayor eficiencia; y para realizar operaciones de entrada/salida, para las que sólo está permitido el determinismo ‘`det`’ o ‘`cc_multi`’

8.2 Chequeo e inferencia del determinismo

El determinismo de los objetivos se infiere a partir del determinismo de sus partes componentes, de acuerdo a un conjunto de reglas prefijadas.

Si una anotación de determinismo es proporcionada para un predicado, el determinismo declarado se compara con el determinismo deducido por el compilador. Si el determinismo declarado es mayor o no comparable con el inferido (en el orden parcial mostrado anteriormente), se produce un error. Si el determinismo declarado es menor que el inferido, se avisa al usuario de tal situación, pero no se considera un error.

Para los predicados locales a un módulo las indicaciones de determinismo pueden omitirse, por lo que el determinismo asignado será el inferido. Sin embargo, es un error omitir el determinismo para los predicados que son exportados desde el módulo que los define, con lo que de esta forma se afianza la fiabilidad de los servicios prestados al módulo cliente.

La inferencia perfecta del determinismo es un problema indecidible, porque requiere resolver el problema de la parada. Esa es la causa de que, en ocasiones, el mecanismo de inferencia de determinismo de Mercury, deducirá un determinismo que no es precisamente el que debería ser. De todas formas, suele ser sencillo resolver estos casos. La forma de hacerlo es reemplazar el chequeo estático del compilador con algún tipo de chequeo manual en tiempo de ejecución.. Por ejemplo, si sabemos que un objetivo específico nunca debería fallar, pero el compilador infiere que el objetivo es semideterminista, entonces, podemos chequear en tiempo de ejecución si el objetivo se satisface y, en caso contrario, llamar al predicado '`error/1`' de la siguiente forma:

```
:- pred q(T, T).
:- mode q(in, out) is det.

q(A, B) :-
        ( objetivo_que_no_debería_fallar_nunca(A, B0) ->
                B = B0
        ;
                error("el objetivo que no falla ha fallado!")
        ).
```

En el ejemplo anterior `X->Y;Z` es una expresión sintáctica condicional permitida por Mercury, que significa que si X se satisface entonces deberemos satisfacer Y, y, en caso contrario, deberemos satisfacer Z.

9. Sistema De Módulos

9.1 Módulos

Cada módulo debe comenzar con una declaración que especifique el nombre del módulo (`` `:- module *ModuleName*' ``), y se divide en dos partes bien diferenciadas: la zona de interfaz y la zona de definición.

Una declaración del tipo ‘`:-interface.`‘ indica el comienzo de la *sección de interfaz* del módulo. En ella se especifican las entidades que serán exportadas por el módulo, y no puede contener la definición de las funciones o predicados que exporte, o definiciones de submódulos.

Mercury proporciona soporte para tipos de datos abstractos, permitiendo que la definición del tipo esté escondida en la parte de implementación y que sólo se exporte el nombre del tipo.

Con la declaración ‘`:- implementation.`‘ se informa del comienzo de la *sección de implementación* del módulo. Cualquier entidad declarada en esta sección es local al módulo (y a sus submódulos) y no puede ser utilizada por otros módulos. La sección de implementación puede omitirse si está vacía.

El módulo puede terminar, aunque no es necesario que lo haga, con una declaración del tipo ‘`:-end_module` *ModuleName*‘, donde el nombre de módulo indicado debe coincidir con el nombre dado en la declaración de inicio del módulo.

Si un módulo desea utilizar las entidades exportadas por otros módulos, debe importar explícitamente esos módulos para hacer sus declaraciones visibles. Para ello, han de emplearse alguna de las dos siguientes declaraciones:

```
:- import module Modules.
```

```
:- use module Modules.
```

donde Modules es una lista de nombres de módulo separados por comas.

Las declaraciones de importación pueden aparecer tanto en la sección de interfaz como en la de implementación, dependiendo del lugar en el que se utilizarán las entidades importadas.

Los nombres de los predicados, funciones, constructores, tipos, modos, clases de tipos, estados de instanciación y submódulos pueden ser cualificados explícitamente utilizando el operador ‘`:`’, antecedido del nombre del módulo(y submódulo) en el que se define la entidad cualificada; con lo que se aumenta la legibilidad y se resuelven los posibles conflictos de nombres. En futuras versiones de Mercury, probablemente, se cambie el operador ’`:`’ por el operador ‘`.`’.

Además, es posible definir y exportar un predicado `` `main/2' ``, que controlará el flujo principal, con su consiguiente flujo de entrada/salida del módulo, y que debe ser declarado, de forma equivalente a alguna de las siguientes:

```
:- pred main(io__state::di, io__state::uo) is det.
```

```
:- pred main(io__state::di, io__state::uo) is cc_multi.
```

Para finalizar este apartado, presentamos, a modo de ejemplo, la definición de un módulo que define las operaciones básicas para trabajar con una cola:

```
:- module cola.
:- interface.

% Declaración de un tipo de datos abstracto.
:- type cola(T).

% Declaración de predicados que operan sobre
% el tipo abstracto de datos.

:- pred cola_vacia(cola(T)).
:- mode cola_vacia(out) is det.
:- mode cola_vacia(in) is semidet.

:- pred meter(cola(T), T, cola(T)).
:- mode meter(in, in, out) is det.

:- pred sacar(cola(T), T, cola(T)).
:- mode sacar(in, out, out) is semidet.

:- implementation.
% Implementamos las colas como listas. Por ello,
% necesitamos el módulo 'list' para poder obtener
% las declaraciones del tipo list(T), con sus
% constructores '[]'/0 y '.'/2; y las del predicado
% list__append/3.

:- import_module list.

% Definición del tipo de datos abstracto cola.
:- type cola(T) == list(T).

% Definición de los predicados exportados.

cola_vacia([]).

meter(Cola0, Elem, Cola) :-
      list__append(Cola0, [Elem], Cola).

sacar([Elem | Cola], Elem, Cola).

:- end_module cola.
```

9.2 Submódulos

Mercury permite que un módulo contenga submódulos, siempre que no haya más de un submódulo con el mismo nombre. Los submódulos se pueden diferenciar en dos tipos: los *submódulos anidados* y los *submódulos separados*. La diferencia es que los primeros están definidos en el mismo fichero fuente que el módulo mientras que los segundos se definen en distintos ficheros fuente.

Cualquier declaración en el módulo padre, incluyendo aquellas en la sección de implementación del módulo padre, son visibles en los submódulos de dicho módulo, incluyendo los submódulos indirectos(submódulos de submódulos suyos). Similarmente, las declaraciones en las interfaces de cualquier módulo importado por el módulo padre son visibles en los submódulos del módulo importador, incluyendo los submódulos indirectos. De todas formas las declaraciones en un módulo hijo no son visibles en el padre ni en los módulos 'hermanos' (hijos del mismo padre) a menos que el hijo sea explícitamente importado.

9.2.1 Submódulos anidados

Están delimitados por parejas de declaraciones:

```
:- module NombreSubmódulo.
....
:- end_module NombreSubmódulo.
```

Siendo la declaración de terminación obligada en todos los casos, para poder diferenciar claramente dónde acaba cada submódulo.

Las partes de interfaz e implementación de un submódulo anidado pueden estar especificadas en diferentes declaraciones de submódulo. Si una secuencia de elementos de un submódulo(elementos que se encuentran entre el par de declaraciones que delimita al submódulo anidado) incluye una sección de interfaz, entonces será una declaración para ese submódulo; si incluye una sección de implementación, entonces es una definición para dicho submódulo; y si incluye ambos, es tanto una declaración como una definición.

Es un error declarar o definir un submódulo dos veces. También definirlo sin haberlo declarado. Si un submódulo es declarado pero no definido explícitamente, entonces hay una definición implícita con una sección de implementación vacía para dicho submódulo.

9.2.2 Submódulos separados

Se declaran usando declaraciones con la forma `` `:- include_module Modules' `` donde `Modules` especifica una lista de submódulos separados por coma, cada uno de los cuales debe estar definido en un fichero fuente separado.

La proyección entre el espacio de nombres de módulos y el espacio de nombres de ficheros fuentes está prefijada en la implementación de Mercury. Se buscarán los ficheros fuente que tengan como nombre alguna subcadena del nombre del submódulo completamente cualificado. Es decir, para un submódulo ‘foo:bar:baz’ se buscarán los ficheros ‘foo.bar.baz.m’, ‘bar.baz.m’ y ‘baz.m’ como posibles ficheros fuentes. Los ficheros fuentes deberán contener la declaración y definición del submódulo.

Si aparece una declaración de submódulos separados en la sección de interfaz de un módulo, entonces sólo las declaraciones de los submódulos se incluyen en el interfaz del módulo padre. Las implementaciones de los submódulos se consideran, implícitamente, como parte de la implementación del padre.

III Mercury vs. Haskell

1 Introducción

Mercury tiene muchas cosas en común con los lenguajes funcionales. Pero ¿Cómo se puede comparar Mercury con lenguajes como Haskell? La respuesta es que el sistema de tipos de Mercury y el de Haskell son similares (la comparación entre Mercury y Haskell puede no ser justa debido a que Hugs y Ghc tienen muchas extensiones de Haskell)

Así como también enumeramos la diferencias en el sistema de tipos, también describimos algunas de las diferencias en el sistema de módulos, puesto que están relacionados.

2 Otras áreas en las que Mercury y Haskell difieren

Aunque Mercury y Haskell tienen un sistema de tipos similar, son diferentes respecto a:

- Sintaxis (obviamente)
- Semántica (cálculo de predicados frente a lambda cálculo)
- Semántica operacional (perezosa /orden de evaluación)
- Manejo de excepciones
- Tratamiento de I/O
- Librerías estándar
- Soporte para la programación lógica

3 Sistemas de tipos de Mercury y Haskell 98

A partir de ahora nos centraremos en las diferencias y similitudes del sistema de tipos y del sistema de módulos de ambos lenguajes

3.1 Similitudes

- El constructor de tipos de datos “type” de Mercury es similar al de Haskell “data”. (“discriminated union (d.u) types, with support for named fields”)

  Ej:

  En mercury:

```
:- type fruta
      --->    manzana
        ;     naranja
        ;     platano
        ;     pera.
```

  En Haskell:

```
data fruta = manzana | naranja | platano | pera
```

En mercury (el tipo lista):

```
:- type list(T)
      --->    []
        ;     [T | list(T)].
```

En Haskell:

```
data [a] = []  | a : [a]   (*definido por defecto Haskell*)
```

- Funciones de orden superior (y expresiones lambda)
  - Las funciones que tienen a otra función como algún parámetro o como resultado, se llaman funciones de orden superior.

Ej:

En mercury:

```
fun :: (Int -> Int) -> Int -> Int
fun f x = f x
```

En Haskell:

```
:- func fun( func(int)=int, int) = int
fun(F, X) = F(X).
```

- Tipos sinónimos (equivalentes)

Ej:

En mercury:

```
type Dinero = Int
```

En Haskell:

```
:- type dineto == int
```

- Tipos parametrizados

Ej:

En mercury:

```
(* "T" es la variable de tipo (parámetro) *)
:-type Punto(T)
      ---> Punto(
            x :: T,
            y :: T).
```

En Haskell:

```
(* "a" es la variable de tipo (parámetro)*)
data Punto a = Par a a
data Punto a = Par { x,y :: a }
```

- Polimorfismo parametrizado
    - función polimórfica con tipos paramétricos

      Ej:

      En mercury:

```
:- func comp_x(Punto(T)) = T.
comp_x(Punto) = Punto ^ x.
```

      En Haskell:

```
comp_x  :: Punto a -> a
comp_x (Par x y) = x
```

- Tipos sinónimos polimórficos
- Funciones polimórficas
- Recursión polimórfica (Como el tipo lista antes definido)
- Clases
- Inferencia de tipos
- Un conjunto similar de tipos básicos
    - Mercury:
        - `Básicos; char, int, float, string`
        - `Predicados; pred, pred(T), pred(T1, T2), ...`
        - `Funciones; (func) = T, func(T1) = T, func(T1, T2) = T, ...`
        - `Tuplas; {}, {T}, {T1, T2}`
    - Haskell
        - `Básicos; Int, Float, Bool, ....`

3.1 Diferencias

- Haskell 98 en cuanto al léxico distingue:
    - Constructores (letra inicial en mayúscula) de funciones/variables (letra inicial en minúscula)
- En Mercury distinge:
    - Variables (letra inicial en mayúscula) de funciones/constructores (letra inicial en minúscula)

- Haskell 98 permite anidar definiciones de funciones, Mercury no.
    - En Mecury puedes usar variables e inicializarlas con expresiones lambda, pero las variables están siempre monomorficamente tipadas y funciones definidas usando expresiones lambda no pueden ser recursivas.

    Ej:

    ```
    X = lambda([L::in, H::out] is det, sum(L,H))
    ```

    Nota: Una vez instanciada una veriable a una expresión lambda se puede usar una de las siguientes formas para llamar a la funcion:

    apply(Func, Arg1, Arg2, ..., ArgN) ;donde Func es el nombre de la variable
    Func(Arg1, Arg2, ..., ArgN)

- Mercury requiene la declaración de tipos para todas las funciones exportadas desde un modulo, Haskell 98 no, al menos en teoría (en la práctica, actualmente la implementación de Haskell efectivamente requiere la declaración de tipos).

- Haskell 98 permite definir implementaciones por defecto para métodos de clases, mientras que Mercury no.

    - Ej:

    ```
    class Eq a where
          (==)    :: a -> a -> Bool
          (/=)     :: a -> a -> Bool
          x/=y    = not (x==y)
    ```

- Haskell permite instancias de clases que deja algunos métodos indefinidos, en cuyo caso la llamada a tal método producirá un error en tiempo de ejecución, mientras que Mercury trata los métodos indefinidos como un error en tiempo de compilación y requiere que los usuarios definan explícitamente el método para que de error en tiempo de ejecución

- La librería estándar de Haskell hace un uso extenso de clases, la de Mercury no.

- Mercury soporta funciones y tipos de datos cuantificados existencialmente, Haskell 98 no (Hug, ghc y otras implementaciones de Haskell soportan tipos de datos cuantificados existencialmente, pero no funciones cuantificadas...)

    - Si una función está cuantificada universalmente, significa que el llamante determinará los valores de los tipos de las variables, y que la función debe estar definida para cualquier tipo
    - Si una función está definida existencialmente, significa que la función debe determinar el tipo de sus argumentos y es el llamante quien debe definir como trabajar para todos los tipos que son instanciados por la funcion

Ej:

```
% Here the type variable `T2' is existentially quantified,
% but the type variables `T1' and `T3' are universally
% quantified.

:- all [T3] some [T2] pred foo(T1, T2, T3).

:- pred bad_foo(T).
bad_foo(42). % type error

:- some [T] pred e_foo(T).
e_foo(42).
% ok

:- pred bad_call_e_foo.
bad_call_e_foo :- e_foo(42).
% type error

:- some [T] pred e_bar1(T).
e_bar1(42).
e_bar1(42).
e_bar1(43).
% ok (T = int)

 :- some [T] pred bad_e_bar2(T).
bad_e_bar2(42).
bad_e_bar2("blah").
% type error (cannot unify types `int' and `string')
```

- Mercury soporta clases multiparametrizadas, sin embargo no soporta dependencias funcionales entre las veriables de tipo y hay algunas restricciones en la forma de declarar instancias, por lo que esto podría no resultar tan útil como podría parecer.

  Ej:

  En Mercury:

  ```
  :- typeclass a(T1, T2,....) where [...].
  ```

  En Haskell:

  ```
  ;solo permite el parámetro 'a'

  class C a where
        f:: a ->Int
        g:: a ->Bool
  ```

- Mercury soporta sobrecarga (un constructor, función o método de clase con el mismo nombre, y el compilador determinará en cada ocurrencia del nombre a cual se refiere en base a los tipos y aridad), Haskell no la soporta (Por supuesto si se hace un uso significativo de la sobrecarga, esto puede dar lugar a ambigüedades que el verificador de tipos no pueda resolver; si tal ambigüedad ocurre, es necesario usar una declaración explicita del tipo para resolverla, mas bien que confiar en la inferencia de tipos).

Nota: El sistema de clases de haskell proporciona una forma para manejar la sobrecarga

Ej: (sobrecarga del método ==)

```
class Eq a where
      (==) :: a-> a -> Bool
.............................

instance Eq Int where
      0 (==) 0 = True
.............................

instance Eq Float where
      0.0   (==) 0.0  = True
.............................
```

- En Haskell 98, las funciones siempre toman un único argumento, múltiples argumentos son manejados usando "curring" (currificación o parcialización) o tuplas. En Mercury, las funciones pueden tomar múltiples argumentos, y tales funciones tienen un tipo distinto de las funciones que toman tuplas o las funciones que devuelven funciones.

- Haskell 98 tiene dos términos para definir "discriminated union types", "data" y "newtype" (se diferencian en la evaluación perezosa y en la representación), Mercury solo usa uno "type"

- Mercury soporta anidar módulos, Haskell no. Por tanto en Mercury puedes tener tipos abstractos cuya definición sea solo visible en un submódulo, mejor que en todos los módulos que contienen al submódulo.

  Ej:

```
:- module  Nombre
     ....
:- end_module
```

  Otra forma: (módulos separados)

```
:- include_module Module1, Module2, ..., ModuleN.
```

- El sistema de módulos de Mercury permite declaraciones de instancias para ser públicas (exportadas) o privadas (locales), mientras que en Haskell las declaraciones de instancia son siempre publicas (siempre se importan o exportan al importar o exportar un módulo)

- Mercury tiene predicados así como funciones (los predicados son distintos de las funciones que devuelven *bool*). Esto incluye el soporte de predicados cuantificados existencialmente, predicados como métodos de clase, predicados expresiones lambda, y predicados de alto orden.

- El tratamiento de la igualdad, comparaciones, clases derivadas automáticamente,.. difieren considerablemente entre los dos lenguajes. Haskell 98 tiene clases propias del sistema Eq, Ord, Enum, Bounded, Read, y Show, y permite instancias que sean automaticamente derivadas. Por ejemplo, derivando de Eq en la declaración del tipo. En Mercury, la igualdad es un constructor del lenguaje, no un método de una clase parecido a == en Haskell. Mercury permite definir la igualdad al usuario, pero porque esto altera la semántica de la notación de la igualdad, tiene diferentes efectos en Mercury, que la redefinición de == en Haskell no produce.

  Ej:

```
data Pp = K | J deriving (Eq)
```

IV Ejemplos Probados

EJEMPL1.M

```
%-------------------------------------------------------%
% Ejemplo de la funcion field que actualiza y consulta el campo del
termino %
%-------------------------------------------------------%
:- module ejempl1.
:- interface.
:- import_module int,string, io.
:- pred main(io__state::di, io__state::uo) is det.
:- implementation.
:- import_module int,string,io.

:- type type2--->type2(field3::int,field4::int).
:- type type1--->type1(field1::type2,field2::string).

:- func field2(type1)=string.
field2(type1(_,Field2))=Field2.

:- func 'field2:='(type1,string)=type1.
'field2:='(type1(Field1,_),Field2)=type1(Field1,Field2).

main-->
      {Term2=type2(3,4), % Construyo un termino del tipo type2
        Term1=type1(Term2,"Campo2"),  %Construyo  un  termino  del  tipo
type1
        Campo2=field2(Term1),
        Term1n='field2:='(Term1,"Nuevo Campo 2"),
        Campo2n=field2(Term1n)
   },
      io__write_string("Valores  del  campo  field2  del  termino  de  tipo
type1: "),
      io__write_string("\nValor antiguo: "),
      io__write_string(Campo2),
      io__write_string("\nValor nuevo: "),
      io__write_string(Campo2n).

:- end_module ejempl1.
```

EJEMPL2.M

```
:- module ejempl2.
:-interface.
:- import_module io,float.
:- pred main(io__state::di, io__state::uo) is det.
:- implementation.

:- typeclass point(T) where [
     pred coords(T,float,float),
     mode coords(in,out,out) is det,
     func translate(T,float,float)=T
   ].

:- type coordinate--->coordinate(float,float).

% CREAMOS INSTANCIAS DE PUNTOS %

:- instance point(coordinate) where [
                pred(coords/3) is coordinate_coords,
                func(translate/3) is coordinate_translate].

:- pred coordinate_coords(coordinate,float,float).
:- mode coordinate_coords(in,out,out) is det.
coordinate_coords(coordinate(X,Y),X,Y).
:- func coordinate_translate(coordinate,float,float)=coordinate.
coordinate_translate(coordinate(X,Y),Dx,Dy)=coordinate(X+Dx,Y+Dy).

% INSTANCIAS DE PUNTOS COLOREADOS %

:- type rgb--->rgb(int,int,int).

:- type coloured_coordinate---> coloured_coordinate(float,float,rgb).

:- instance point(coloured_coordinate) where [
pred(coords/3) is coloured_coordinate_coords,
func(translate/3) is coloured_coordinate_translate
].

:- pred coloured_coordinate_coords(coloured_coordinate,float,float).

:- mode coloured_coordinate_coords(in,out,out) is det.
coloured_coordinate_coords(coloured_coordinate(X,Y,_),X,Y).

:-                                                              func
coloured_coordinate_translate(coloured_coordinate,float,float)=
    coloured_coordinate.
coloured_coordinate_translate(coloured_coordinate(X,Y,Colour),Dx,Dy)=
coloured_coordinate(X+Dx,Y+Dy,Colour).

main-->
{P=coloured_coordinate(3.0,3.0,rgb(1,2,1)),
  P1=translate(P),
  P2=coordinate(3.0,3.0),
  P3=translate(P2)
},
io__write_string("Hola").

:- end_module ejempl2.
```

HASHABLE.M

```
:- module hashable.
:- interface.
:- import_module int, string, io.
:- typeclass hashable(T) where [func hash(T) = int].
:- instance hashable(int).
:- instance hashable(string).
:- pred main(io__state::di, io__state::uo) is det.
:- implementation.

:- instance hashable(int) where [func(hash/1) is hash_int].
:- instance hashable(string) where [func(hash/1) is hash_string].

:- func hash_int(int) = int.
hash_int(X) = X.

:- func hash_string(string) = int.
hash_string(S) = H :-
% use the standard library predicate string__hash/2
string__hash(S, H).


main-->
      {P=2},
     io__write_int(P).
:- end_module hashable.
```

V Bibliografía

Blas C. Ruiz Jiménez, Francisco Gutiérrez López, Pablo Guerrero García, José E. Gallardo Ruiz, *Razonando con Haskell. Una Introducción a la Programación Funcional*. Impreso por Imagraf S.A. en el año 2000 I.S.B.N: 84-607-12818-4

Todos los documentos, incluso aquellos para los que no se especifica una dirección de Internet concreta, tienen su correspondiente copia en la red, que puede localizarse fácilmente partiendo de <http:\\cs.mu.oz.au\research\mercury>, y siguiendo el enlace que te proporcionan en esa página hacia los escritos (papers) referentes a Mercury.

Zoltan Somogyi,Fergus Henderson, Thomas Conway, *The execution algorithm of Mercury, an efficient purely declarative logic programming language*. Artículo de investigación en Journal of Logic Programming, 26(1-3):17-64, Octubre-Diciembre 1996.

Fergus Henderson, Zoltan Somogyi, Thomas Conway, *Determinism analysis in the Mercury compiler.* Artículo técnico 95/25. Departamento de Ciencias de la Computación, Universidad de Melbourne, Melbourne, Australia, Agosto 1995.

Zoltan Somogyi, Fergus Henderson, Thomas Conway, *Mercury, an efficiente purely declarative logic programming language.* Artículo técnico. Departamento de Ciencias de la Computación, Universidad de Melbourne, Melbourne, Australia

Zoltan Somogyi, *A system of precise modes for logic programming.* Actas de la Cuarta Conferencia Internacional sobre Programación Lógica (Proceedings of the Fourth International Conference on Logic Programming). Páginas 769-787, Melbourne, Australia, Junio 1987

Zoltan Somogyi, Fergus Henderson, Thomas Conway, David Jeffrey, *The Mercury language reference manual.*

Chris Speirs, Zoltan Somogyi, Harald Sondergaard,*Termination Analysis for Mercury,* Melbourne University, Melbourne, Australia.
<http://www.cs.mu.oz.au/publications/tr_db/mu_97_09.ps.gz>.

TRANSPARENCIAS

Ejemplos de lambda-expresiones en los sistemas de tipos de Church

$\lambda x : A.\ x$ Se interpreta como la función que toma como parámetro un objeto denominado "x", que es de tipo "A" y devuelve ese mismo objeto. El tipo de la propia abstracción sería $A \rightarrow A$

$\lambda x : (B \rightarrow C).\ x$ Se interpreta como la función que toma como parámetro un objeto "x" de tipo $B \rightarrow C$, esto es, una función de $B \rightarrow C$, y devuelve esa misma función "x". El tipo de la abstracción sería $(B \rightarrow C) \rightarrow (B \rightarrow C)$

Ejemplos de lambda-expresiones en los sistemas de tipos de Curry

$\lambda x.x$ Se interpreta como una función que toma como parámetro un objeto "x" y devuelve ese mismo objeto. No se especifica, precisamente, el tipo de "x", con lo que podría ser de tipo A, $B \rightarrow C$, $A \rightarrow B \rightarrow C$, ... o cualquier otro.

$\lambda x\ y.\ y\ x$ Se interpreta como una función que toma como parámetro dos objetos "x" e "y" y devuelve un objeto resultante de aplicar "x" a "y". El tipo de "y" debe ser necesariamente $A \rightarrow B$ siempre que "x" sea de tipo A. El tipo de la abstracción sería $(A \rightarrow (A \rightarrow B)) \rightarrow B$

Axiomas y reglas de los sistemas de tipos de Curry

(ax) $\dfrac{}{\Gamma \vdash x : A} \qquad x: A \in \Gamma$

($\rightarrow$ e) $\dfrac{\Gamma \vdash f : A \rightarrow B \quad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B}$

($\rightarrow$ i) $\dfrac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x .b : A \rightarrow B}$

Propiedades de los sistemas de tipos de Curry

(Sβ) $\Gamma \vdash a : A \ \wedge\ a \rightarrow\!>_{\beta} a' \Rightarrow \Gamma \vdash a' : A$ (subject β-reduction)

(Lema de sustitución) $\Gamma \vdash a : A \Rightarrow <B := C> \Gamma \vdash a : <B := C> A$

(Lema de generación)
(d) Si $\Gamma \vdash x : A$, entonces $x : A \in \Gamma$
(e) Si $\Gamma \vdash f\ a : B$, entonces existe un tipo A tal que $\Gamma \vdash f : A \rightarrow B \ \wedge\ \Gamma \vdash a : A$
(f) Si $\Gamma \vdash \lambda x.b : C$, entonces existen dos tipos A y B tales que $C \equiv A \rightarrow B$, y además se cumple que $\Gamma, x : A \vdash b : B$

(Unicidad Variables) $\Gamma \vdash x : A \ \wedge\ \Gamma \vdash x: B \Rightarrow A \equiv B$

Problemas de los sistemas de tipos de Curry

- **comprobación de tipos (type checking),** denotado con $\vdash^{?} a : A$
- **tipificación (typability)**, denotado con $\vdash a : ?$
- **habitabilidad (inhabitation)**, denotado con $\vdash ? : A$

Meta Lenguaje Estándar (SML)

-Fuertemente tipado. Cualquier expresión legal tiene un tipo determinado automáticamente por el compilador, lo cual elimina la existencia de errores de tipo en tiempo de ejecución.
-Sistema de tipos polimórfico. Cada frase tiene un único tipo más general que determina el conjunto de contextos en los cuales la frase puede ser usada correctamente.

Lambda-Prolog

Declaraciones "amables" (*kind declaration*). Constructores de tipos

```
kind o          type.
kind int        type.

kind lista      type -> type.
kind par        type -> type -> type.
```

Con estas declaraciones, (lista int) es el tipo "lista de enteros", (par int string) es el tipo correspondiente a un par formado por un entero y una cadena y (lista (lista int)) es el tipo "lista de listas de enteros".

Declaraciones de tipos

Son de la siguiente forma:

type <nombre_tipo> <expresión de tipo> .

donde <expresión de tipo> es

<expresión de tipo> ::= <variable de tipo> |
(<expresión de tipo> **->** <expresión de tipo> **) |**
(<constructor de tipo de aridad n> <exp. tipo 1> ... <exp tipo n>**)**

Ejemplos:

```
type hola               int -> real -> string.
type mi_tipo            o -> int -> o.
type tipoa_raro         lista A -> (A -> B) -> lista B -> o.
type funcion_y_lista    (A -> B) -> lista (A -> B).
```

MODOS

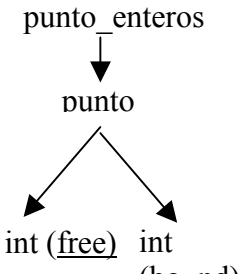
Modo de una variable

Forma en que se instancia la variable(según la estructura determinada por el tipo)

Muy relacionado con el tipo de la variable.

Representación mediante un *árbol de instanciación*.
Ejemplo:

```
:- type punto_enteros ---> punto(int,int).
:- inst instanciacion == bound(punto(free,bound))
```



Términos aproximados:
~~punto(X,Y)~~
~~punto(1,X)~~
punto(X,1)
~~X~~
~~punto(1,1)~~

Modos por defecto: free, ground.

Modo de un predicado

Transformación del estado de instanciación inicial de los argumentos al estado de instanciación final.

Modos por defecto:

```
:- mode in == ground >> ground.
:- mode out == free >> ground.
```

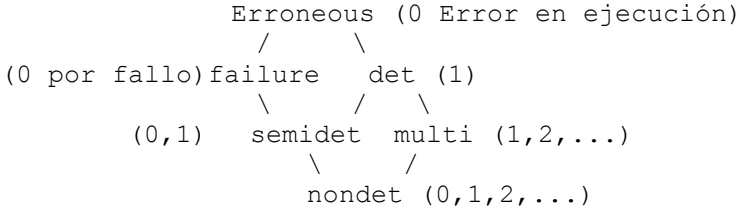
Ejemplo:

```
:- pred append(list(T), list(T), list(T)).
:- mode append(in, in, out) is det.
:- mode append(out, out, in) is multi.
:- mode append(in, in, in) is semidet.
```

DETERMINISMO

Clasifica el modo de un predicado atendiendo a la cantidad de resultados que se obtienen ante la combinación de los parámetros de entrada.

Tipos de determinismo:



```
:- type fruit
        --->    apple
        ;       orange
        ;       banana
        ;       pear.

:- type strange
        --->    foo(int)
        ;       bar(string).

:- type employee
        --->    employee(
                        name        :: string,
                        age         :: int,
                        department  :: string
                ).

:- type tree
        --->    empty
        ;       leaf(int)
        ;       branch(tree, tree).

:- type list(T)
        --->    []
        ;       [T | list(T)].

:- type pair(T1, T2)
        --->    T1 - T2.
```

1

```
:- type type1
        ---> type1(
                field1 :: type2,
                field2 :: string
        ).

:- type type2
        ---> type2(
                field3 :: int,
                field4 :: int
        ).
```

2

```
:- func field1(type1) = type2.
field1(type1(Field1, _)) = Field1.

:- func 'field1 :='(type1, type2) = type1.
'field1 :='(type1(_, Field2), Field1) = type1(Field1,
Field2).
```

3

```
:- func increment_field3(type1) = type1.

increment_field3(Term0) =
  Term0 ^ field1 ^ field3 := Term0 ^ field1 ^ field3
+ 1.
```

4

```
incremental_field3(Term0) = Term :-
  OldField3 = field3(field1(Term0)),

  OldField1 = field1(Term0),
  NewField1 = 'field3 :='(OldField1, OldField3+ 1),
  Term = 'field1 :='(Term0, NewField1).
```

5

```
:- typeclass point(T) where [
        % coords(Point, X, Y):
        %       X and Y are the cartesian coordinates of Point
        pred coords(T, float, float),
        mode coords(in, out, out) is det,

        % translate(Point, X_Offset, Y_Offset) = NewPoint:
        %       NewPoint is Point translated X_Offset units in
the X direction
        %       and Y_Offset units in the Y direction
        func translate(T, float, float) = T
].
```

6

```
:- typeclass foo(T) where [
      func method1(T, T) = int,
      func method2(T) = int].

:- instance foo(int) where [
      % method defined by naming the implementation
      func(method1/2) is (+),

      % method defined by a fact
      method2(X) = X + 1.
```

7

```
:- type coordinate
        ---> coordinate(
                float,          % X coordinate                    8
                float           % Y coordinate
        ).

:- instance point(coordinate) where [
        pred(coords/3) is coordinate_coords,
        func(translate/3) is coordinate_translate
].


:- pred coordinate_coords(coordinate, float, float).
:- mode coordinate_coords(in, out, out) is det.

coordinate_coords(coordinate(X, Y), X, Y).

:- func coordinate_translate(coordinate, float, float) = coordinate.

coordinate_translate(coordinate(X, Y), Dx, Dy) = coordinate(X + Dx, Y
+ Dy).
```

```
:- type rgb
        ---> rgb(
                int,
                int,
                int
        ).

:- type coloured_coordinate
        ---> coloured_coordinate(
                float,
                float,
                rgb
        ).

:- instance point(coloured_coordinate) where [
        pred(coords/3) is coloured_coordinate_coords,
        func(translate/3) is coloured_coordinate_translate
].


:- pred coloured_coordinate_coords(coloured_coordinate,
float, float).
:- mode coloured_coordinate_coords(in, out, out) is det.

coloured_coordinate_coords(coloured_coordinate(X, Y, _), X,
Y).

:- func coloured_coordinate_translate(coloured_coordinate,
float, float)
        = coloured_coordinate.

coloured_coordinate_translate(coloured_coordinate(X, Y,
Colour), Dx, Dy)
        = coloured_coordinate(X + Dx, Y + Dy, Colour).
```

```
:- module hashable.
:- interface.
:- import_module int, string.

:- typeclass hashable(T) where [func hash(T) = int].
:- instance hashable(int).
:- instance hashable(string).

:- implementation.

:- instance hashable(int) where [func(hash/1) is hash_int].
:- instance hashable(string) where [func(hash/1) is
hash_string].

:- func hash_int(int) = int.
hash_int(X) = X.

:- func hash_string(string) = int.
hash_string(S) = H :-
        % use the standard library predicate string__hash/2
        string__hash(S, H).

:- end_module hashable.
```

```
:- typeclass ring(T) where [
    func zero = (T::out) is det,                % '+' identity
    func one = (T::out) is det,                 % '*' identity
    func plus(T::in, T::in) = (T::out) is det, % '+'/2 (forward mode)
    func mult(T::in, T::in) = (T::out) is det, % '*'/2 (forward mode)
    func negative(T::in) = (T::out) is det      % '-'/1 (forward mode)
].
```

11

```
:- typeclass euclidean(T) <= ring(T) where [
        func div(T::in, T::in) = (T::out) is det,
        func mod(T::in, T::in) = (T::out) is det
].
```

12

```
:- typeclass portrayable(T) where [
        pred portray(T::in, io__state::di,io__state::uo)
is det
].
```

13

```
:- instance portrayable(int) where [
        pred(portray/3) is io__write_int
].

:- instance portrayable(char) where [
        pred(portray/3) is io__write_char
].
```

```
:- instance portrayable(list(T)) <= portrayable(T) where [
       pred(portray/3) is portray_list
].
```

14

```
:- pred portray_list(list(T), io__state, io__state) <=
portrayable(T).
:- mode portray_list(in, di, uo) is det.

portray_list([]) -->
        [].
portray_list([X|Xs]) -->
      portray(X),
      io__write_char(' '),
      portray_list(Xs).
```

15