ManTa
Manejador de Tipos Abstractos de Datos

MISC-98-2-11

Vladimir Támara Patiño

Asesor:
Ing. Rodrigo Cardoso

Grupo de Construcción de Software

UNIVERSIDAD DE LOS ANDES

FACULTAD DE INGENIERÍA

DEPARTAMENTO DE SISTEMAS Y COMPUTACIÓN

Santafé de Bogotá D.C. Junio de 1997

TABLA DE CONTENIDO

El desarrollo de ManTa se ha podido continuar debido al aporte de varias personas, a quienes les debo un agradecimiento::

Al profesor Rodrigo Cardoso (Universidad de los Andes) que ha sido el guía del proyecto desde sus comienzos

Al profesor Klaus Madlener (Universidad de Kaiserslautern) quien ha colaborado en el desarrollo de las bases teóricas

Al profesor Cesar Muñoz (Universidad de Standford) que ha leído y corregido las bases teóricas

Al profesor Jean Marie Favre de quien he tomado ideas para el desarrollo del esquema de traducción de ManTa a otros lenguajes, y de Z a ManTa

1 INTRODUCCIÓN

ManTa es un sistema para desarrollo de software, basado en la teoría de Tipos Abstractos de Datos (TAD).

Algunas de sus características son:

- Manejo de Tipos Abstractos de Datos
  - Permite especificar TADs usando un lenguaje simple con un estilo funcional/ecuacional denominado ManTa
  - Puede generar semi-automaticamente nuevos TADs a partir de otros existentes por diversos métodos (Copia, Instanciación, Producto Cartesiano, Restricción, Completación, Suma directa y Variación)
  - Incluye como predefinidos los tipos: NAT, BOOL, Char y String. También incluye algunos TADs útiles y populares: List, Stack y Queue.
  - Permite representar un TAD en otro (e.g Representar el TAD Stack con el TAD List)
- Reescritura
  - Incluye un motor de reescritura, que permite evaluar expresiones.
- Prueba de teoremas
  - Permite probar igualdades con respecto a las especificaciones de los TAD empleando reescritura e inducción
  - Permite probar si una representación mantiene la semántica, es decir si cumple los axiomas del TAD representado
- Generación de Código
  - Puede generar implementaciones de un TAD en lenguajes populares (ANSI C, OCaml) para diversos compiladores (gcc, bc, djgpp, cygnus).
- Cuenta con interfaces en inglés y español, aunque puede configurarse fácilmente en otros idiomas.
- Cuenta con diversas interfaces de acuerdo a las posibilidades del usuario: Plana, EMACS, Java
- Está escrito en lenguajes estándares e independientes de plataforma (ANSI C, OCaml y Java). Se ha compilado y probado con éxito en diversas arquitecturas con varios compiladores (Ultrix, Unix, Solaris, Linux, Dos, Windows 3.x, Windows 95/98/NT)

Este sistema se ha desarrollado desde 1985 en la Universidad de los Andes, sus desarrolladores se presentan a continuación junto con la contribución realizada:

Año	Autor	Contribución	Referencia
1985-1998	Rodrigo Cardoso	Líder del proyecto. Desarrollos teóricos y del lenguaje de especificación	[Car93]
1985	Marcela Rodríguez	Herramienta para edición de TADs (DOS - Pascal)	
1987	Roberto Barón	Ambiente de desarrollo de TADs y probador de teoremas (DOS - Pascal)	
1998	Ricardo Espinel	Generador de código LISP (DOS - Pascal)	
1989	Silvia Takahashi	Ambiente ManTa 1 (DOS - Pascal)	
1990	Carolina Ruiz	Ambiente ManTa 2 (UNIX - C)	[Ruiz90]
1992	Cesar Muñoz	Probador de teoremas y generador de prototipos en C (UNIX - C)	[Muñoz92]
1995	Hernán Aponte	Nueva interfaz e interacción con GNU EMACS (UNIX - C, Emacs Lisp). ManTa 3.0	[Aponte95]
1996	Alvaro González	Comienzo de una interfaz gráfica Tcl/Tk (Xwindow -Tcl/Tk, C)	
1998	Klaus Madlener	Desarrollo de bases teóricas	
1997-1998	Vladimir Támara	Independencia de plataforma Interfaz gráfica en Java Difusión del sistema Desarrollo de bases teóricas Plan de desarrollo de Software Desarrollo de las versiones 3.1 a 3.3 (ANSI C, Ocaml, Java)	[Támara97] y este escrito

Aún cuando se han producido varias versiones de componentes de ManTa, sólo desde 1997 se cuenta con un plan de desarrollo de software y se sigue un proceso de desarrollo [Tamara97]. El actual plan de desarrollo y algunos lineamientos para realizarlo, se presentan en el capítulo 2.

De manera análoga, sólo hasta 1998 se lograron bases teóricas del lenguaje de especificación, del motor de reescritura, del demostrador de teoremas y del generador código. Estos logros se han alcanzaron después de un año de trabajo en grupo de Klaus Madlener, Rodrigo Cardoso y el autor de este escrito. Tales bases se presentan en el capítulo 3 y son uno de los logros más importantes de este trabajo.

El reciente desarrollo de las bases teóricas de ManTa ha desacelerado la producción de componentes y versiones del sistema. La primera versión que sigue los lineamientos esbozados en las bases teóricas, que ofrece todas las características prometidas desde 1992 y que es estable y utilizable es la versión 3.1. Esta versión se ha obtenido haciendo reingeniería a las fuentes de la versión 3.0, e infortunadamente aún tiene problemas. Un resumen del desarrollo de esta versión y de las técnicas de reingeneiría aplicadas, se presenta en el capítulo 4.

El capítulo 5 está dedicado a la versión 3.3, esta versión es el desarrollo más complejo logrado, hasta el momento, con apoyo del sistema ManTa. Se trata del comienzo de un *bootstrap*, es decir que se ha especificado empleando el lenguaje ManTa, se ha generado código empleando la versión 3.1 y se ha integrado con interfaces sencillas. En el momento se ha desarrollado buena parte del núcleo en el lenguaje de programación Ocaml. Este ejemplo muestra el poder del lenguaje y del sistema ManTa, y así mismo, que es una alternativa viable para desarrollo de software.

Para facilitar la lectura de este documento, cada capítulo cuenta con una breve introducción, donde se resume el contenido y se explica la organización.

2 INGENIERÍA DE SOFTWARE

Antes de 1997 no había planes de desarrollo especificados, el primero junto con un proceso de desarrollo se expone en [Tamara97]. Sin embargo, el esquema allí propuesto puede mejorarse, teniendo en cuenta que:

- No involucra el uso de ManTa como herramienta para desarrollo de Software
- Las etapas de diseño e implementación toman mucho tiempo
- La implementación está sujeta a muchos errores que son difíciles de seguir, encontrar y corregir
- Ha sido muy difícil retomar el código legado, depurarlo y tratar de darle arquitectura

En este capítulo se presenta un nuevo plan de desarrollo, junto con los requerimientos que lo han motivado y algunas metodologías de ingeniería de software que se han desarrollado y utilizado con ManTa.

2.1. REQUERIMIENTOS

Los requerimientos generales de ManTa, se recolectaron por medio de varias actividades (Ver [Tamara97]) :

- Entrevistas a usuarios de ManTa
- Lecturas relacionadas
- Evaluación de otros sistemas
- Uso y evaluación de las versiones existentes de ManTa

Se han organizado por prioridades como se presentan a continuación (después de una revisión de lo propuesto en [Tamara97]) :

Prioridad	Requerimiento	Descripción
1	Difusión	Es necesario dar a conocer el trabajo que tantas personas han desarrollado en los últimos 13 años. Esto además podría acelerar el desarrollo si personas externas a la Universidad de los Andes colaboran.
1	Calidad de software	Es necesario difundir un producto libre de errores, eficiente y de alta calidad.
1	Buenas bases teóricas	El desarrollo de ManTa debe hacerse después de desarrollar bases teóricas apropiadas (y no antes como ha ocurrido tradicionalmente)

1	Bootstrap	Una fase de bootstrap permite asegurar la corrección del sistema ManTa generado y hace posible portarlo a otros lenguajes (los soportados por ManTa).
2	Independencia de Plataforma	A partir de la versión 3.1 se logró, debe mantenerse en la siguientes versiones para lograr una difusión amplia
2	Documentación	El software distribuido debe contar con buena documentación, para que pueda ser compilado, instalado y utilizado.
2	Internacionalización	Las versiones previas de ManTa han contando con facilidades para manejar varios idiomas; esto no debe perderse en las versiones que se desarrollen (al menos debe soportarse inglés y español). Para facilitar la difusión el idioma de desarrollo será inglés
2	Generación de código en otros lenguajes	C está perdiendo su popularidad frente a los nuevos lenguajes como C++, Java y Ocaml. ManTa debe adaptarse a las nuevas tendencias, y su diseño debe propiciar la adopción de futuros lenguajes.
2	Núcleo independiente de interfaz	Para facilitar desarrollo de otras interfaces (e.g. Plana, Emacs, Gráfica). Se logró desde ManTa 3.0 y aún más con ManTa 3.2. Esta característica debe mantenerse
3	Representaciones y código generado	Es necesario implementar bien el mecanismo de representaciones. De este mecanismo depende la calidad del código generado por ManTa. Las representaciones finales en cada lenguaje deberían ser fáciles de configurar. Debería haber un buen banco de tipos básicos para construir nuevos tipos y representarlos con los básicos. Así mismo el código generado debe ser fácil de modificar por el usuario.
3	Extensiones a las bases teóricas	Tipos genéricos (los parámetros deben poder ser funciones), sobrecarga de nombres, mejores conjeturas (cláusulas de Horn) y métodos de prueba
3	Interactividad y mejoras en el demostrador	El banco de pruebas puede integrarse con ManTa, y la definición de variables debe poderse hacer interactivamente. Debería haber un banco de lemas

4	Metalenguaje para cambiar especificación	Con un metalenguaje podrían definirse operaciones sobre las especificaciones como cambiar recursión por recursión de cola. Esto podría dar lugar a código generado más eficiente (por lo menos en estilo imperativo)
4	Corrección del código del usuario	Debería contar con un mecanismo que facilite la verificación del código modificado por el usuario, para asegurar que es correcto. Así mismo la documentación del código generado debería incluir PRE y POST, deducidos de la implementación y generación.
4	Examinador de Tipos y sus relaciones	Debe contar con un navegador, que permita examinar los tipos existentes y sus relaciones y que facilite la modificación de los mismos

2.2 PLAN DE DESARROLLO

Con base en los requerimientos y después de una evaluación al plan de desarrollo propuesto en
[Tamara97] se ha llegado al siguiente plan:

Versión	Características
3.1	Primera versión realmente completa y considerablemente estable, apta para el desarrollo de otras versiones del sistema Independiente de plataforma. Protocolo para portar el sistema a nuevas plataformas Generación de código en ANSI C
3.2	Reestructuración de la arquitectura. Independiente de plataforma y protocolo para portarlo Interfaz independiente del núcleo GUI y protocolo para crear nuevas interfaces Generación de código en ANSI C
3.3	Bootstrap en OCaml Independiente de plataforma Interfaz independiente del núcleo Generación de código en ANSI C Generación de código en OCaml
3.4	Bootstrap en Java Independiente de plataforma Interfaz independiente del núcleo Generación de código en ANSI C Generación de código en OCaml Generación de código en Java
4.0	Nuevas bases teóricas: soporte de parámetros (incluyendo funciones con axiomas como parametros), sobrecarga de nombres y teoremas

	como cláusulas de Horn.

Las diferencias con respecto al anterior plan de desarrollo, están a partir de la versión 3.3, que propone implementar de nuevo las fuentes. Este trabajo, a primera vista titánico, se alivia con el uso de ManTa (ver sección 2.3).

2.2 PRÁCTICAS DE INGENIERÍA DE SOFTWARE

Junto con el desarrollo de ManTa se han adoptado algunas metodologías para desarrollo de software. Así mismo se han desarrollado algunas nuevas:

- Proceso de desarrollo de software
    - Se ha propuesto un nuevo esquema que involucra a ManTa
- Especificación:
    - Traducción de Z a ManTa
    - ManTa como lenguaje de especificación
- Implementación
    - Generación de código empleando ManTa
    - Técnicas para programar en C
- Documentación
    - Nuevo estilo de documentación en C

2.2.1 UN NUEVO PROCESO DE DESARROLLO

El desarrollo de la versión 3.3 se ha realizado sobre la versión 3.1. El proceso de desarrollo propuesto no es tradicional, dado que se emplea ManTa:

1. Requerimientos
2. Análisis de requerimientos
3. Diseño modular
4. Especificación en ManTa de cada módulo
5. Pruebas a la especificación
6. Generación de código
7. Implementación de la interfaz y módulos de control
8. Integración y pruebas al sistema completo
9. Evaluación

Este proceso se ha comenzado a probar y a aplicar para el desarrollo de la versión 3.3, como puede consultarse en el capítulo 4.

2.2.2. TRADUCCIÓN DE Z A MANTA

Las versiones más recientes de ManTa hacen bastante fácil la especificación "*on-line*" (en el computador), pues sugieren lados izquierdos de axiomas y proporcionan plantillas de forma que el usuario sólo debe escribir lo necesario. Sin embargo, especificar programas en papel empleando la sintaxis de ManTa no resulta fácil, pues tiene muchos detalles: Signatura con iniciales, constructoras y selectoras; Axiomas que requieren cubrimiento total, igualdad para todo tipo, **`if`** que requiere tanto **`else`** como **`fi`**, etc.

Con ideas tomadas de la librería ZML de Jean Marie Favre [Favre98], se han propuesto algunas guías para escribir especificaciones en un subconjunto de Z, de forma que la traducción a ManTa resulta directa.

- No se emplean esquemas
- Además de tener tipos predefinidos se pueden emplear tipos libres y productos (ver por ejemplo [Z95]), los tipos libres introducen nuevos conjuntos, junto con algunos elementos e inyecciones de otros tipos en el nuevo.

```
Bool::= true | false
Expr::= errorExpr | varExpr Ý Ident Þ |  genExpr Ý Ident ²
seq ExprÞ |
                 selExpr Ý ident ² seq ExprÞ | eqExpr Ý Expr ²
ExprÞ |
                 ifExpr Ý Expr ² Expr ² ExprÞ.
```

- Se especifican las funciones junto con sus restricciones. Debe hacerse lo posible para que las restricciones estén cuantificadas universalmente y el símbolo de la función aparezca en la parte más exterior de las expresiones.

```
ADTHierarchy::=basic|extension|generic|genericExtension
adtHierarchy:AbsType ªADTHierarchy
Õi:Ident,sp:seq     Parameter,sg:seq     Generator,ss:seq1
Selector ×
(sp=<         >         Ù         sgµ<         >         Û
adtHierarchy(absType(i,sp,sg,ss))=basic
       sp=<       >       Ù       sg=<       >       Û
adtHierarchy(absType(i,sp,sg,ss))=extension
       spµ<       >       Ù       sgµ<       >       Û
adtHierarchy(absType(i,sp,sg,ss))=generic
       spµ<       >)      Ù       sg=<       >       Û
adtHierarchy(absType(i,sp,sg,ss))=genericExtension)
```

Siguiendo estas reglas, la traducción a ManTa es bastante sencilla:

- Cada tipo libre es un nuevo TAD en ManTa, sus generadoras son los elementos e inyecciones del tipo en Z.

```
ADT Bool[]
INITIALS
```

```
    true     :                      -> Bool
    false    :                      -> Bool

  ADT Expr[]
  INITIALS
    errorExpr:                      ->Expr
    varExpr  : String               -> Expr
    genExpr  : String * Lexpr       -> Expr
    genExpr  : String * Lexpr       -> Expr
  CONSTRUCTORS
    eqExpr   : Expr*Expr            ->Expr
    ifExpr   : Expr*Expr*Expr       -> Expr
```

- Cada función es una selectora y basta modificar las restricciones para que conforme los axiomas de la nueva selectora:

```
ADT ADTHierarchy []
INITIALS
  basic                 :            -> ADTHierarchy
  extension       :            -> ADTHierarchy
  generic               :            -> ADTHierarchy
  genericExtension      :            -> ADTHierarchy
CONSTRUCTORS
SELECTORS
  adtHierarchy          : AbsType   -> ADTHierarchy
AXIOMS
 adtHierarchy(absType(i,nil_LParameter,cons_LGenerators(h,t),ss))=
    basic
 adtHierarchy(absType(i,nil_LParameter,nil_LGenerators,ss))=
    extension
 adtHierarchy(absType(i,cons_LParameter,cons_LGenerators(h,t),ss))=
   generic
 adtHierarchy(absType(i,cons_LParameter,nil_LGenerators,ss))=
  genericExtension
```

A manera de ejemplo puede consultarse en el capítulo 3, la Sintaxis Abstracta de ManTa, que se ha especificado en Z y después en ManTa.

2.2.3 TÉCNICAS PARA DESARROLLO EN C

Como se mencionó antes, la versión 3.1 se ha construido a partir de las fuentes legadas (versión 3.0) que corrían en Unix y resultaban muy inestables. En el desarrollo de la versión 3.1 se han experimentado diversas técnicas. Las técnicas más recientes y que parecen para nuestro caso ser las mejores se resumen a continuación (pueden consultarse más detalles en la descripción del desarrollo de ManTa 3.1)

- Independencia de plataforma, agrupando funciones no estándar en un sólo módulo
- Corrección y prevención de fallas empleando metódicamente `assert`
- Internacionalización empleando archivos de mensajes configurables
- Interfaz independiente del núcleo, evitando que las funciones del núcleo impriman,

sólo retornan mensajes a las funciones llamadoras, tomados de los archivos de mensajes

- Manejo de error estilo UNIX (códigos de error retornados por cada función), pero permitiendo que el núcleo genere mensajes de error en el idioma apropiado (como se mencionó antes sin mostrarlos).
- Control de versiones empleando RCS
- Programas de pruebas tanto para el núcleo como para la interfaz
- Herramientas para generar semiautomáticamente documentación y makefiles de las fuentes y distribuciones
- Métricas para evaluar el desarrollo empleando el ManTa Benchmark
- Adopción de algunos estándares de programación GNU
    - Uso de ANSI C
    - Comentarios y código fuente en inglés
    - Chequeo de códigos de error al hacer llamadas al sistema y al localizar memoria
    - Indentación GNU
- Adopción de algunos estándares de programación tomados de Java Conventions
    - Nomenclatura. Variables y funciones comienzan con minúsculas. Módulos que representan TADs comienzan con mayúsculas. Los nombres no se separan con _ sino por medio de la capitalización (e.g. esteEsUnEjemplo). Las constantes deben ser en mayúsculas (pueden separarse con _)
- Nuevos estándares de documentación en C (ver sección 2.2.2)

Infortunadamente no todas las fuentes de la versión 3.1 siguen los estándares aquí propuestos dado que el esfuerzo requerido para lograr la conversión es inmenso. A manera de ejemplo está disponible el código fuente del módulo de reescritura, que es completamente nuevo (Ver apéndice A).

Ha sido realmente penosa la labor de desarrollar la versión 3.1, pero se ha hecho así para poder
completar una primera versión del sistema ManTa efectivamente completa y funcional. Además sobre esta versión se han construido las versiones 3.2 y 3.3. Esperamos que este las técnicas mencionadas puedan servir para continuar el desarrollo de este sistema y de otros.

2.2.4 Documentación de fuentes en C

El estilo de documentación adoptado para el desarrollo de ManTa hasta la versión 3.1b2 era tomado de [Toro91]. Este enfoque tiene, sin embargo, algunas desventajas:

- Requiere demasiado tiempo del programador
- Hace innecesariamente largas las fuentes del programa, dado que tiene demasiados adornos

- No tiene herramientas que lo soporten
- Es muy fácil olvidar su actualización cuando se cambian las fuentes

Por otra parte, el estilo de documentación de Java ha dejado una grata experiencia::

- Comentarios con sintaxis
- **`Javadoc`**, herramienta para generación de documentación HTML a partir de fuentes en Java con comentarios apropiados

Infortunadamente, **`Javadoc`** no puede usarse con fuentes en C, sólo con fuentes Java. Hay una herramienta similar denominada **`DOC++,`** enfocada especialmente a C++. Dado que la versión legada está en C y que aún se programa bastante en C (especialmente en cursos introductorios en el país), se ha preferido proponer un estilo propio, que puede apreciarse en el programa presentado en el Anexo A.

3 DESCRIPCIÓN FORMAL DE MANTA

Este capítulo resume uno de los logros más importantes de este trabajo, las bases teóricas de ManTa, que se han obtenido después de varios intentos fallidos:

- En [Car93] se inicia el trabajo con base en ideas de Manna (Ver [Manna74]). Uno de los atractivos de este trabajo es que da una semántica a las funciones definidas (sintacticamente) empleando un teorema de punto fijo. Sin embargo en [Car93] no hay un tratamiento satisfactorio del símbolo **`ERROR`**.
- El primer intento del autor de este escrito fué continuar por la línea de Manna, partiendo de la equivalencia a nivel de categorías entre un álgebra con funciones parciales y un álgebra extendida con un símbolo **`ERROR`** y funciones totales. Siguiendo este enfoque se concluyó que toda función excepto **`if`** debía extenderse naturalmente ($F(a_1,..$**`ERROR`**$,..a_n)$ debía ser **`ERROR`**) para asegurar que simpre hubise un punto fijo. Este requerimiento generó serias dificultades para el motor de reescritura pues empleando una estrategia de reducción de punto fijo (e.g. *leftmost*) se perdía la confluencia.
- Se continuó el trabajo tratando de mantener extensiones normales e if no estricto, pero ideando un algoritmo de reescritura que mantuviera la confluencia y fuese de punto fijo. Tal algoritmo se concibió aunque dependía de un predicado indecidible (determinar si una función es estricta o no en un parámetro). Se desarrollo teoría para lograr un procedimiento de semidecisión que funcionaba para muchos casos, y así se solucionó en buena parte el problema de la reescritura. Sin embargo el mayor problema de todo el enfoque de Manna, se encuentra cuando se intentan demostrar teoremas, pues los modelos subyacentes a una teoría con funciones parciales son completamente diferentes a los modelos de la teoría asociado con el símbolo **`ERROR`**. Para notarlo basta pensar en los naturales y en los naturales extendidos con un símbolo **`ERROR`**, en el primero vale x*0=0, mientras que en el segundo no necesariamente (pues si se emplea una extensión natural **`ERROR`***0 debe ser **`ERROR`**).
- Se optó entonces por un enfoque que da prioridad a la consistencia de los modelos subyacentes y le quita importancia a la existencia de un punto fijo para las funciones. En este nuevo enfoque la semántica de las funciones es enteramente operaciona y el símbolo **`ERROR`** no se interpreta en los módelos, sólo es azucar sintáctica y una facilidad (o dificultad) para el usuario. Este trabajo se basa una tesis de doctorado, desarrollada en la Universidad de Kaiserslautern [Kuhler98].

En este capítulo se presenta la semántica de un TAD en ManTa, junto con una descripción del lenguaje inspirada en [Meyer90]. Se trata de una descripción usual para lenguajes de programación, que ha sido adaptado para ManTa.

Esta descripción se da en "etapas", cada vez más abstractas y se compone de:

Sintaxis concreta (3.1)	Sintaxis de los lenguajes que el usuario puede emplear para interactuar con el sistema
Arbol de sintaxis abstracta (3.2)	Una representación abstracta de la sintaxis concreta
Semántica estática (3.3)	Restricciones impuesta a los arboles de sintaxis abstractas (y por tanto a la sintaxis concreta)
Semántica dinámica	Comportamiento esperado o significado del lenguaje

A su vez la semántica dinámica se describe de varias formas:

Semántica operacional (3.4)	Especifica la semántica por medio de un interprete abstracto. En nuestro caso el motor de reescritura
Semántica denotacional (3.5)	Asocia a toda construcción del lenguaje un conjuntos de funciones matemáticas, que definen su significado
Semántica translacional (3.6)	Generalmente esta semántica presenta una traducción a un lenguaje más sencillo o mejor entendido. En nuestro caso presentamos un esequema general de traducción de ManTa a otros lenguajes de programación y a manera de ejemplo un traductor a OCaml.

Aún cuando las versiones 3.1 y 3.3 de ManTa soportan tipos paramétricos (o TADs genéricos), estos no se describirán por completo. Este tema y otras posibles extensiones al lenguaje se presentan informalmente en la sección 3.7.

3.1 SINTAXIS CONCRETA

Basicamente un usario puede especificar TADs, proponer términos para reescribir (encontrar formas normales), o proponer conjeturas para probar. Cada una de estas operaciones tiene una sintaxis específica.

3.1.1 CONVENCIONES LÉXICAS

- Es *case-sensitive*
- Los espacios, CR/LF y tabuladores son ignorados
- Se pueden escribir comentarios entre /* y */ al igual que en C. No se permiten comentarios anidados
-
- Son palabras reservadas : **ADT**, **INITIALS**, **CONSTRUCTORS**, **SELECTORS**, **AXIOMS**, **ERROR**, **EQUAL**, **if**, **then**, **else**, **fi**, **(i)**, **(c)**.
- Los identificadores deben comenzar con una letra (minúscula o mayúsucla), seguida de otras letras, dígitos o '_'. La máxima longitud de un identificador es 32 caracteres.

3.1.2 SINTAXIS DEL LENGUAJE PARA DESCRIBIR TADS

Intuitivamente un TAD define datos y funciones que operan con tales datos, para describir los datos se emplean <u>funciones constructoras</u>. mientras que las funciones que operan sobre los datos se describen con <u>funciones selectoras</u>. La gramática para describir TADs consta de dos partes: (1) una signatura donde se declara el nombre del TAD, sus parámetros y el dominio y codominio de cada función (constructoras y selectoras) y (2) una sección con axiomas que describen el comportamiento de cada selectora.

ident = ['A'-'Z' 'a'-'z'] ['A'-'Z' 'a'-'z' '0'-'9' '_']*	Un identificador debe comenzar con una letra y debe componerse sólo de letras y números, deben tener a lo sumo 32 caracteres (esto evita problemas al traducir a otros lenguajes)
expr = **ERROR** | **EQUAL(** *expr* , *expr* **)** | **if** *expr* **then** *expr* **else** *expr* **fi** | *ident* **(** *arguments* **)** | *ident*	Las expresiones se construyen con **ERROR** (para expresar casos no definidos), **EQUAL(.,.)** (para decidir igualdad entre dos datos de un mismo TAD), **if . then . else . fi** (condicional típico), funciones instanciadas y constantes. Las constantes son funciones sin argumentos (su dominio es vacío) que deben escribirse sin paréntesis.

arguments = *expr* , *arguments* | *expr*	Si una función recibe varios argumentos, estos deben separarse con comas ','.
axiom= *expr* = *expr* *axioms* = *axiom axioms*	Cada selectora de un TAD debe tener un conjunto de axiomas que la definan. Un axioma es una regla de reescritura (no una igualdad), de forma que el lado izquierdo reescribe en el derecho.
notEmptyDomain= *ident* (i) * *notEmptyDomain* | *ident* * *notEmptyDomain* | *ident* (i) | *ident*	El dominio de una función (bien sea constructora o selectora) debe ser un producto de TADs. Cada TAD del producto puede ser o no inductivo. Los parámetros inductivos son candidatos para hacer inducción en las demostraciones.
domain= *notEmptyDomain* | ε	Una función puede tener dominio vacío, y en tal caso, es una constante.
constructor= *ident* : *domain* -> *ident* *constructors*= *generator generators* | ε	Una constructora es una función cuyo codominio debe ser el TAD definido (pues permite construir los elementos del tipo). El usuario no da axiomas para las constructoras, sólo especifica su dominio y codominio.
selector = *ident* (c) : *domain* -> *ident* *ident* : *domain* -> *ident* *selectors* = *selector selectors* | *selector*	Una selectora es una función que opera sobre elementos de un tipo. Su comportamiento debe describirse con axiomas. La (c) indica si es o no caracterizadora, una selectora caracterizadora permite dar un criterio de igualdad entre dos elementos de un mismo tipo. Note que el usuario debe escribir por lo menos una selectora
notEmptyParameters = *ident* , *notEmptyParameters* | *ident* *parameters*= *notEmptyParameters* | ε	Un parámetro de un TAD es un *placeholder* para un tipo, así puede implementarse genericidad.[1] Si un TAD tiene varios parámetros, estos deben separarse por medio de comas.

[1] La gramática y las versiones hasta ahora construidas, brindan un mínimo de genericidad, sin embargo este tema se evitará en este documento, dado que se están implementando mejoras a este mecanismo que aún no se han formalizado.

adt= **ADT** *ident* [*parameters*] **INITIALS** *constructors* **CONSTRUCTORS** *constructors* **SELECTORS** *selectors* **AXIOMS** *axioms*	Un TAD tiene un nombre, parámetros, constructoras llamadas iniciales (no incluyen el TAD definido en su dominio) y constructoras que incluyen el TAD definido en el dominio. Cuenta además con funciones llamadas selectoras y cada selectora declarada debe tener axiomas que definan su comportamiento sobre las constructoras.

La gramática presentada es LALR, hay disponibles un analizador léxico y un *parser* en el sitio de WEB **`http://wwwisc.uniandes.edu.co/~manta`**.

3.1.3 SINTAXIS DE EXPRESIONES POR NORMALIZAR

El motor de reescritura de ManTa recibe una expresión como las especificada (*expr*) y la reescribe hasta obtener la forma normal o determinar que no está definida.

3.1.4 SINTAXIS DEL LENGUAJE DE CONJETURAS

El actual demostrador de teoremas permite probar sólo teoremas ecuacionales. La teoría desarrollada permite probar claúsulas de Horn donde cada literal y la conclusión son bien igualdades o el símbolo especial **DEF** (para denotar el predicado cierto para términos definidos). En este documento nos limitaremos a teoremas ecuacionales cuyas conjeturas deben ser de la forma:

conjExpr=
expr **=** *expr*

Como podrá notar es la misma gramática de axiomas sólo que en este caso el símbolo = es un predicado de igualdad (igualdad fuerte). Vale la pena diferenciar esta igualdad de **EQUAL** que representa un igualdad débil que no necesariamente es la sintáctica.

3.1.5 TIPOS ABSTRACTOS PREDEFINIDOS

El único TAD primitivo es **BOOL**, que define los valores booleanos **TRUE** y **FALSE** junto con las funciones `AND(.,.)`, `OR(.,.)`, `NOT(.)`, `IMP(.,.)` y `EQ_BOOL(.,.)`. Este debe ser el tipo del primer argumento de la expresión: **if** E_1 **then** E_2 **else** E_3 **fi**.

Hay además algunos tipos predefinidos que por defecto no pueden ser modificados: **NAT** (números naturales), **Char** y **String**. Además de estos tipos básicos hay predefinidos algunos TADs clásicos: **List**, **Queue**, y **Stack**.

A manera de ejemplo incluimos la definición del TAD `BOOL` y del TAD genérico `List[X]`

```
TAD BOOL [  ]
INICIALES
  TRUE        : -> BOOL
  FALSE       : -> BOOL
CONSTRUCTORAS
SELECTORAS
  AND      (c): BOOL * BOOL(i) -> BOOL
  OR       (c): BOOL * BOOL(i) -> BOOL
  NOT      (c): BOOL(i) -> BOOL
  IMP      (c): BOOL(i) * BOOL(i) -> BOOL
  EQ_BOOL  (c): BOOL(i) * BOOL(i) -> BOOL
AXIOMAS
  AND(B1,TRUE) = B1
  AND(B1,FALSE) = FALSE
  OR(B1,TRUE) = TRUE
  OR(B1,FALSE) = B1
  NOT(TRUE) = FALSE
  NOT(FALSE) = TRUE
  IMP(TRUE,TRUE) = TRUE
  IMP(TRUE,FALSE) = FALSE
  IMP(FALSE,TRUE) = TRUE
  IMP(FALSE,FALSE) = TRUE
```

```
ADT List [ X ]
INITIALS
  nil           : -> List
CONSTRUCTORS
  cons          : X * List-> List
SELECTORS
  head          : List(i) -> X
  tail          : List(i) -> List
  length        : List(i) -> NAT
  concat        : List(i) * List-> List
  isEmptyList     : List(i) -> BOOL
EQ_List   (c): List(i) * List(i) -> BOOL
AXIOMS
  head(nil) = ERROR
  head(cons(X1,L2)) = X1
  tail(nil) = ERROR
  tail(cons(X1,L2)) = L2
  length(nil) = ZERO
  length(cons(X1,L2)) = SUC(length(L2))
  concat(nil,L1) = L1
  concat(cons(X1,L2),L3) =
     cons(X1,concat(L2,L3))
  isEmptyList(nil) = TRUE
  isEmptyList(cons(X1,L2)) = FALSE
  EQ_List(nil,nil) = TRUE
  EQ_List(nil,cons(X1,L2)) = FALSE
  EQ_List(cons(X1,L2),nil) = FALSE
  EQ_List(cons(X1,L2),cons(X3,L4)) =
     AND(EQUAL(X1,X3),EQ_List(L2,L4))
```

3.1.6 Convenciones sugeridas

Para facilitar la escritura y lectura de TADs y conjeturas, y para facilitar la traducción a otros lenguajes se sugieren algunas convenciones

- Los identificadores deben tener relación con lo que representan.
- No debe usarse *underline* _ para separar palabras en un identificador, sino minúsculas y mayúsculas (e.g en vez de `este_es_un_identificador` escribir `esteEsUnIdentificador`).
- Los nombres de TADs deben comenzar con mayúsculas (e.g. `AbsType`)
- Los nombres de funciones deben comenzar con minúsculas (e.g. `absType`)
- Los parámetros de tipos deben ser en mayúsculas
- Los nombres de variables deberían ser cortos y dicientes, deberían comenzar con minúscula

3.2 Arbol de Sintaxis Abstracta

Los arboles de sintaxis abstracta permiten aislar la información más importante de la sintaxis concreta en estructuras de datos. Estos arboles pueden ser descritos en ManTa (y así se ha hecho para el desarrollo de la versión 3.3 empleando la 3.1), sin embargo usaremos el lenguaje de especificación Z como metalenguaje. Emplearemos sólo algunas características del lenguaje Z y un nuevo tipo `Bool` (se hizo así con el fin de facilitar la traducción al lenguaje de ManTa)

3.2.1 ESPECIFICACIÓN DE LOS AST

```
[Bool]
     true,false:Bool
       (true=false)
       x:Bool x=true Ù x=false
```

Identificadores	`[Ident]`
Un parámetro de un TAD	`Parameter::=parameter ÝIdentÞ`
Domino de una función. Puede ser inductivo	`Domain::=domainÝIdent ² BoolÞ`
Un generador consta de un nombre, un dominio y un codominio	`Generator::=generatorÝ Ident ² seq DomainÞ`
Las expresiones posibles son: **ERROR**, variables, constantes, funciones instanciadas, **EQUAL(.,.)** o **if** ***E1*** **then** ***E2*** **else** ***E3*** **fi**	`Expr::=errorExpr, varExprÝ Ident Þ` `│ genExprÝIdent²seq ExprÞ` `│ selExprÝIdent²seq ExprÞ` `│ eqExprÝExpr ² ExprÞ` `│ ifExprÝ Expr ² Expr ² ExprÞ`
Un axioma consta de una expresión izquierda y una expresión derecha	`Axiom::=axiomÝ Expr ² ExprÞ`
Una función selectora tiene un nombre, un dominio y un codominio puede además ser caracterizadora y debe tener al menos un axioma	`Selector::=selectorÝ Ident ² seq Domain ² Ident` `² Bool ² seq`$_1$ `ExprÞ`
Un tipo abstracto de datos tiene un nombre, un estado (modificable o no), una lista de parameteros, un conjunto de generadoras y un conjunto no vacío de selectoras	`AbsType::=` `absTypeÝ Ident ² Bool` `² seq Parameter ² seq Generator` `² seq`$_1$ `SelectorÞ`

3.2.2 JERARQUÍA DE TADS EN MANTA

Los TADs en ManTa pueden clasificarse dependiendo si tiene generadoras y/o parámetros:

	Generadoras	**Parámetros**
Básico	Si	No
Extensión	No	No
Genérico	Si	Si
Extensión genérica	No	Si

3.3 SEMÁNTICA ESTÁTICA

La semántica estática identifica (1) violaciones a ciertas reglas de nomenclatura (2) tipo correcto en las expresiones y (3) restricciones en axiomas para asegurar propiedades del sistema de reescritura (Ver sección 5). Para verificar la semántica estática definimos el predicado **significado-estático**.

Vale la pena resaltar que la semántica de cualquier componente del AST, depende de los TADs que se hayan definido, al conjunto de los TADs definidos le llamaremos el **ambiente**.

3.3.1 EXPRESIONES

Para que una expresión tenga significado en un ambiente de TADs debe darse:

- Que sea una variable y su nombre no sea identificador de una función (generadora o selectora) de TAD alguno en el ambiente. Además en una expresión con varias variables, las apariciones de una misma variable deben ser en parámetros del mismo tipo.
- Si es una generadora $g(E_1,...,E_n)$, debe estar definida en algún TAD del ambiente con signatura $g{:}T_1{}^{2}...{}^{2}T_{n-1}{}^{a}T_n$. Cada argumento E_i debe ser una expresión con significado-estático y debe tener tipo T_i.
- Si es una selectora $s(E_1,...,E_n)$, debe estar definida en algún TAD del ambiente con signatura $s{:}T_1{}^{2}...{}^{2}T_{n-1}{}^{a}T_n$. Cada argumento E_i debe ser una expresión con significado-estático y debe tener tipo T_i.
- Si es una expresión de igualdad **EQUAL**(E_1,E_2) , E_1 y E_2 deben ser expresiones con el mismo tipo y con significado-estático.
- Si es un condicional **if** E_1 **then** E_2 **else** E_3 **fi**, E_1 debe ser de tipo **BOOL**, E_2 y E_3 deben tener el mismo tipo. Además, cada una de las tres expresiones E_1, E_2 y E_3 debe tener significado-estático.

3.3.2 DOMINIOS

Un producto cartesiano de TADs conforma el dominio para una generadora/selectora de un tipo a, tiene significado-estático si cada TAD del productodo está en el ambiente o es a. Además cada TAD debe ser básico (a manera de excepción y para brindar un mínimo de genericidad a puede ser genérico, tener un sólo parámetro y aparecer como parte del dominio.)

3.3.3 GENERADORAS

Una generadora tiene significado-estático cuando su nombre no es el nombre de otra función (bien sea del mismo TAD o de otro) y cuando su dominio tiene significado-estático. Además el nombre de una generadora no debe ser una palabra reservada de ManTa ni un nombre que comience con `EQ_` (estos nombres estan reservados para funciones de igualdad).

Un conjunto de generadoras tiene significado si cada generadora tiene significado.

3.3.4 AXIOMAS

Un conjunto de axiomas para una selectora tiene significado si cumple cada una de las siguientes condiciones:

1. Cada axioma debe tener significado-estático

   Un axioma para una selectora $S{:}D_1 \times D_2 ... D_{n-1} \rightarrow D_n$ tiene significado-estático si:
   - Es de la forma $S(E_1,...,E_{n-1})=T$, con E_i expresión de tipo D_i
   - T cumple una de las siguientes:
     - Es expresión de tipo D_n.con significado-estático
     - Es **`ERROR`**.
     - Es de la forma **`if`** E_1 **`then`** E_2 **`else ERROR fi`**, con E_1 de tipo **`BOOL`** y con significado estático, y E_2 de tipo D_n y con significado-estático
     - Es de la forma **`if`** E_1 **`then ERROR else`** E_3 **`fi`**, con E_1 de tipo **`BOOL`** y con significado estático, y E_3 de tipo D_n y con significado-estático
   - Cada E_i es una variable o una generadora del tipo D_i instanciada en generadoras o variables.
   - No hay variables repetidas al lado izquierdo.
   - Todas las variables al lado derecho del axioma aparecen al lado izquierdo

   Las tres últimas condiciones permiten asegurar algunas condiciones importantes para la reescritura (i.e. Linealidad izquierda y no traslapamiento)

2. El conjunto de axiomas de la selectora cubre por completo el dominio de la selectora

   En [Huet82] se expone una propiedad (denominada completitud de un conjunto de axiomas sobre generadoras), que permite verificar que una función sea total:

   Definición. Dada una selectora S de aridad k definida por p axiomas, cada uno de la forma $S(A_i^1, A_i^2, ..., A_i^k) = T_i$, donde A_i^j es una variable o una. Sea R el conjunto de las k-tuplas que forman los lados izquierdos de los axiomas para S, es decir $R = \{< A_i^1, A_i^2, ..., A_i^k > \mid 1 \le i \le p\}$

   Se dice que **el conjunto de axiomas de *S* es completo para las generadoras** si se da una sola de las siguientes 3 condiciones:
   1. k=0 y R={<>}
   2. El conjunto de (k-1)-tuplas $\{< A_i^2, A_i^3, ..., _i^k > \mid A_i^1 \text{ es variable}\}$ es completo para las generadoras
   3. (a) Para toda generadora G del tipo del primer parámetro, hay al menos un axioma, que comienza con el símbolo de función G.

(b) la unión de los dos conjuntos de (n+k-1)-tuplas $\{< x_1,...,x_n, A_i^2,...,A_i^k > \mid A_i^1 \text{ variable}\}$ y $\{< P_1,...,P_n, A_i^2,...,A_i^k > \mid A_i^1 = G(P_1,...,P_n) \text{ con } G \text{ generadora}\}$ es completo para las generadoras, donde las x_i son variables nuevas, distintas a las de los axiomas.

3. El conjunto de axiomas es local - fuertemente confluentes

   Un conjunto de axiomas es local-fuertemente confluente si dados dos axiomas cuyos lados izquierdos unifiquen con unificador más general σ, se tiene que sus lados derechos son iguales después de aplica la sustitución σ.

4. Si la selector es una igualdad para un tipo T (i.e EQ_T:T*T->BOOL), los axiomas deben definir una congruencia:

- Reflexividad: Si E es término de tipo T compuesto de generadoras, entonces **EQ_T**(E,E) debe reescribir a **TRUE**.
- Simetría: Si E_1 y E_2 son términos de tipo T compuestos por generadoras entonces **EQ_*T***(E_1, E_2) y **EQ_**T(E_1, E_2) deben reescribir al mismo término
- Transitividad: Si E_1, E_2 y E_3 son términos de tipo T compuestos por genradoras y **EQ_**T(E_1, E_2) reescribe a **TRUE**, **EQ_**T(E_2, E_3) reescribe a **TRUE** entonces
- **EQ_**T(E_1, E_3) debe reescribir a **TRUE**
- Preservar funciones: Si F es una selectora con signatura F:$T_1 \times T_2 ... \times T_n \rightarrow T$, y E_1: T_1, E_2: T_2..., E_n: T_n, D_1: T_1, D_2: T_2,... D_n: T_n son términos compuestos por generadoras y **EQ_** T_1(E_1,D_1) reescribe a **TRUE**, ...**EQ_** T_n(E_n, D_n) reescribe a **TRUE** entonces **EQ_**T(F(E1,E2,...En),F(D1,D2,...Dn)) debe reescribir a **TRUE**.

3.3.5 Selectoras

Una selectora tiene significado-estático si su nombre no es el mismo de otra función, si su dominio tiene significado-estático y sus axiomas tienen significado-estático. El nombre de la selectora no debe ser una palabra reservada del lenguaje y si su nombre es de la forma **EQ_*T***, debe ocurrir que ***T*** sea el nombre del TAD al que pertenece la selectora definido.

Un conjunto de selectoras de un TAD tiene significado si cada selectora tiene significado.

3.3.6 Parámetros

Una lista de parámetros de un tipo tiene significado en un ambiente de TADs si no hay parámetros repetidos y ninguno tiene el nombre del tipo definido o de otro.

3.3.7 Tipos Abstractos de Datos

Un TAD tiene significado-estático si su nombre no es de otro TAD del ambiente ni es nombre de algún parámetro de otro TAD. Así mismo los parámetros, las generadoras y las selectoras debe ser válidos. Si es un TAD básico o genérico debe tener por lo menos una función inicial (para asegurar que haya datos del TAD).

e.g. El siguiente TAD aunque tiene una constructora, no define elementos.

```
ADT NoElem[]
INITIALS
CONSTRUCTORS
  no: NAT * NoElem->NoElem
```

3.4 SEMÁNTICA OPERACIONAL

Un TAD de ManTa no es más que una definición, sólo adquiere carácter práctico cuando se evalúa una selectora, así la semántica operacional se especifica indicando como evalúa ManTa una expresión.

Al igual que en otros lenguajes ecuacionales y funcionales, en ManTa la evaluación de expresiones se hace por reescritura. Para un conjunto de TADs el sistema de reglas de reescritura lo constituyen los axiomas de las selectoras, la semántica estática permite asegurar algunas propiedades de este sistema: Linealidad izquierda y confluencia local.

Comenzaremos por definir la notación que usaremos y siguiendo las recomendaciones de [O'Donell85] describiremos completamente[2] la reescritura con:

- Una representación para las expresiones
- Un algoritmo para correspondencia de patrones[3]
- Un algoritmo para seleccionar la siguiente expresión a reducir (redex[4])
- Un algoritmo para realizar la reducción escogida

Definición: Dado un conjunto de TADs, sea ***Cons*** el conjunto de todas las generadoras (iniciales y constructoras), ***Sel*** el conjunto de todas las selectoras y ***Var*** un conjunto de variables.

El conjunto ***GroundGen*** de **generadoras básicas** consta de:

i. Toda inicial de aridad 0

ii. Si $E_1,E_2,..E_n$ son generadoras básicas y $C\in$***Gen*** es generadora de aridad n entonces $C(E_1,E_2,..E_n)$ es generadora básica siempre y cuando tenga significado estático.

El conjunto ***Ground*** de **términos básicos**[5] consta de:

i. Toda generadora básica

---

[2] No trataremos en este documento los detalles de estos algoritmos, que son sin embargo, muy importantes para lograr sistemas utilizables. (Ver [O'Donell85])

[3] En inglés *Pattern Matching*

[4] *redex* es sigla de *reduction expression*, se refiere a un subtérmino que puede reescribirse

[5] En inglés *Ground Terms*

ii. Si $E_1,E_2,..E_n$ son términos básicos y $F\in$***Cons***$\cup$***Sel*** es función (inicial, constructora o selectora) de aridad n entonces $F(E_1,E_2,..E_n)$ es término básico siempre y cuando tenga significado estático

El conjunto de **términos** ***Term*** consta de:

i. Toda variable $v\in$ ***Var***

ii. Todo término básico

iii. Si $E_1,E_2,..E_n$ son términos y $F\in$***Cons***$\cup$***Sel*** es función (inicial, constructora o selectora) de aridad n entonces $F(E_1,E_2,..E_n)$ es término siempre y cuando su tipo sea correcto.

Definición : Una **sustitución** es una función σ: ***Var*** →***Term*** tal que $\sigma(x)\neq x$ sólo para un número finito de variables. Se prefiere la notación $x\,\sigma$ a la tradicional $\sigma(x)$.

Una sustitución σ se puede extender naturalmente a una función con dominio y codominio subconjuntos de ***Term*** así:

- El símbolo **`ERROR`**
- Si c es constante $c\,\sigma = c$
- Si F es función n-aria y $t_1, t_2..., t_n$ son términos $F(t_1, t_2..., t_n)\,\sigma = F(t_1\,\sigma, t_2\,\sigma,..., t_n\,\sigma)$

Una sustitución puede escribirse como $\sigma= \{x_1\Rightarrow t_1, x_2 \Rightarrow t_2 ,..., x_n \Rightarrow t_{n,}\}$ donde x_i $(1\leq i\leq n)$ es variable $x_i \neq t_i$ y si para alguna variable v se da $v\,\sigma \neq v$ existe $1\leq j\leq n$ tal que $v=x_j$.

Se denota por ***Sust*** el conjunto de las sustituciones, i.e. ***Sust***={σ | σ es sustitución }

Definición: Sean $P,O\in$ ***Term***, se dice que la sustitución σ es *match* entre el patrón P y el objetivo O o que P corresponde a O si y solo si $P\sigma = Q$.

3.4.1 REPRESENTACIÓN DE EXPRESIONES Y ALGORITMO PARA CORRESPONDENCIA DE PATRONES

Para facilitar la descripción, cada expresión se representa como un árbol n-ario derivado del tipo `Expr` presentado en la sección 3 (no tendremos en cuenta DAG[6], paralelismo o cuestiones de eficiencia de la implementación en este escrito).

Algoritmo: Dado un patrón P y un objetivo O:

- Si P es variable hay correspondecia y el match es $\{P \Rightarrow O\}$
- Si P es constante y O es la misma constante hay correspondencia y el match es { }
- Si P es función n-aria $F(P_1,...P_n)$ y O es la misma función $F(O_1,..O_n)$ hay

[6] Sigla de *Directed Acyclic Graph*, estructuras de datos apropiadas para expresiones, donde puede compartirse memoria y hacer más eficiente la correspondencia de patrones y la reescritura.

correspondencia si para todo i ($1 \leq i \leq n$) P_i corresponde a O_i con match M_i . En tal caso el match es $M_1 \cup M_2 \cup ... \cup M_n$.

- En cualquier otro caso no hay correspondencia

Un detalle importante de este algoritmo es que en el caso de una función de aridad n, no revisa si una variable está en dos *matchs* de subexpresiones distintas (un algoritmo típico debería chequear esto). Tal caso no se da en ManTa, pues el algoritmo de reescritura, sólo hace *pattern matching* con lados izquierdos de axiomas y la semántica estática exige que cada áxioma sea lineal izquierdo, así que no hay variables repetidas en patrón alguno.

Otro detalle interesante es el simbolo **`ERROR`** al que no se prohibe corresponder con una variable. Esto puede permitirle al demostrador probar propiedades más generales.

3.4.2 REGLA DE REESCRITURA

Inicialmente ManTa empleaba una regla funcional (se empleaba una lista con los axiomas y se reescribía con el primero que se pudiera aplicar). Ahora se emplea una regla más tradicional y cuyas propiedades son más conocidas: **Regla más izquierda** (o más externa, o llamado por nombre)[7]. Está regla para reescritura con una sola regla tiene propiedades muy deseables [Manna74] y para un sistema de reglas es "mejor" que el método usual de llamada por valor [O'Donell77].

Dada una expresión por reescirbir la regla más izquierda consiste en recorrer el árbol de la expresión por niveles y cada nivel de izquierda a derecha, en cada nodo del árbol se hace un paso de reescritura: si es posible reescribir se comienza nuevamente desde la raíz, de lo contrario, se continúa el recorrido. El proceso termina cuando se llegue al último nodo del arbol sin realizar cambios.

3.4.3 ALGORITMO DE REESCRITURA

De acuerdo a la regla de reescirtura más izquierda, en cada nodo se hace un **paso de reescritura**, esto consiste en examinar la expresión que comienza en el nodo:

Si es una variable o el símbolo **`ERROR`** ya está en forma normal

Si es un **`if-then-else-fi`**

Si su primer argumento es **`TRUE`** reescribir en el segundo

Si su primer argumento es **`FALSE`** reescribir en el tercero

Si no es ninguno de los anteriores (en el recorrido deben saltarse 2do y 3er argumento del **`if-then-else-fi`**, dado que **`if`** no es estricto)

Si es **`EQUAL`**

Si puede determinarse el tipo de sus argumentos y el tipo coincide, digamos es ***`T`*** , reescribir a la igualdad del tipo (**`EQ_`*`T`***). (A manera de ejemplo la expresión

[7] En inglés *Leftmost* o *Leftmost-Outermost*

EQUAL(X,Y) no puede reescribirse dado que no es posible determinar el tipo de los argumentos)

Si es una generadora continuar el recorrido

Si es una selectora *S* (incluyendo **EQ_*T***)

Examinar los axiomas de la selectora *S*, si hay correspondencia (el patrón es el lado izquierdo del axioma el objetivo es *S*), reescribir con el lado derecho del axioma haciendo las sustituciones del *match*

La semántica estática de axiomas y los algoritmos aquí presentados tiene varias propiedades importantes que se exponen a continuación.

Definición: Llamamos sistema de reglas de reescritura de un conjunto de TADs a las reglas definidas por los axiomas de cada selectora de cada TAD junto con las reglas:

if TRUE then E_1 **else** E_2 **fi** $\rightarrow$ E_1

if FALSE then E_1 **else** E_2 **fi** $\rightarrow$ E_2

EQUAL(E_1**,** E_2**)** $\rightarrow$ **EQ_*T*(**E_1**,** E_2**)** si tanto E_1 como E_2 tienen tipo T

Propiedad 1: El sistema de reglas de reescritura de un conjunto de TADs en ManTa es:

1. líneal izquierdo
2. sin traslapamientos
3. local-fuertemente confluente

Propiedad 2: El sistema de reglas de un conjunto de TADs en ManTa preserva el significado-estático.

Dado que sólo reescribe con términos de tipo apropiado y emplea nombres definidos por el usuario (que tienen significado-estático)

Teorema 1. Todo sistema terminante de reglas de reescritura de un conjunto de TADs en ManTa es confluente.

Esto ocurre con todo sistema lineal izquierdo, localmente confluente es confluente [Toyama88]. En ManTa las reglas son localmente confluentes pues son local-fuertemente confluentes.

Note que el algoritmo de reescritura presentado no tiene tratamiento especial para **ERROR**, ni para términos no definidos. Así que la forma normal de muchas expresiones que incluyan el símbolo ERROR, es el mismo término (e.g. **SUC(ERROR)**, **ADD(ERROR,ERROR)**, **NOT(ERROR)**, etc).

Otra propiedad muy importante en sistemas de reescritura es la terminación, sin embargo aún no hemos trabajado en esta dirección y por ahora es responsabilidad del usuario.

3.5 SEMÁNTICA DENOTACIONAL

Damos semántica a un conjunto de TADs definiéndolo como una especificación con constructores, como se describe en un manuscrito no publicado en el momento de este escrito: "A Tactic-Based Inductive Theorem Prover for Data Types with Partial Operations" [Kuhler98].

Primero se presenta la notación, algunas definiciones y resultados de tal trabajo y después definimos un conjunto de TADs en ManTa y mostramos algunas propiedades.

3.5.1. NOTACIÓN Y DEFINICIONES BÁSICAS

Definición 1:

- Una especificación con constructores *spec*=(*sig*,*C*,*R*) es una tupla, donde *sig* es una signatura, *C* es un conjunto de constructores y *R* un conjunto de ecuaciones condicionales.
- Una **signatura** es una tupla, *sig* =(*S*,*F*,α), donde *S* es un conjunto de nombres para *sorts*, *F* es un conjunto de símbolos de funciones y α es una función de aridad. i.e. Una función cuyo dominio es *F* y cuyo codominio es *S*+ (*S*+ es el conjunto de cadenas no vacías con S como alfabeto)
- Las **constructoras** son símbolos de función y para cada sort debe haber al menos uno.
- Una **ecuación condicional** es una claúsula, $l=r \leftarrow \lambda_1,\ldots,\lambda_n$, donde cada λ_i es uno de los siguientes (o su negación):
  - $t_1=t_2$
  - ***def***(*t*)

Definición 2 : Dada una especificación con constructores *spec*=(*sig*,*C*,*R*)

- Una sig-algebra *A* (un álgebra donde la signatura se interpreta) se denomina un **modelo** de *spec* si la interpretación de cada ecuación condicional de *R* es valida en *A*. La clase de todos los modelos de *spec* se denota *Mod(spec)*
- Decimos que un modelo *A* de *spec* es un **modelo de datos** de *spec*, si para todo cualquier par de términos básicos compuestos de constructoras t_1,t_2 cuyas interpretaciones en *A* sean iguales, se tiene que $t_1=t_2$ es valido en todo modelo de *spec*. *DMod(spec)* denota la clase de todos los modelos de datos de *spec*.

En el manuscrito [Kuhler98] hay dos clases de variables, variables de constructoras (V^C) y variables generales (V^G). *Term*(*sig*,V^G) denota el conjunto de terminos construidos con los símbolos de la signatura *sig* y con variables generales.

Definición 3: Dada una especificación con constructores *spec*=(*sig*,*C*,*R*), *sig* =(*S*,*F*,α) con $C\subseteq F$:

- Una **regla de constructora** es una regla de reescritura $l=r \leftarrow u_1=v_1,\ldots,u_n=v_n$ tal que
  - *l*,*r* están compuestos con constructoras y variables y u_i,v_i se componene de constructoras y variables de constructoras V^C
  - $Var(l)\subseteq Var(r)$, y $Var(u_i)\cup Var(v_i)\subseteq Var(l)$ para $i=1,2,\ldots n$
- Una **regla de definición** es una regla de reescritura $l=r \leftarrow \Delta$ que satisface
  - *l* no se componen únicamente de constructoras.
  - Δ no contiene literales de la forma $\neg$***def***(*t*)

Definición 4: Sea *spec=(sig,C,R)* un especificación con constructoras y $R=R^C\cup R^D$ donde R^C

es un conjunto de reglas de constructoras y R^D un conjunto de reglas de definición

i) Sea la secuencia $(\rightarrow_{R^C,i})_{i\in\mathrm{N}}$ de relaciones sobre $Term(sig,V^G)$ definida por

a) $\rightarrow_{R^C,0}=\emptyset$
b) $t_1\rightarrow_{R^C,i+1}t_2$ si hay una regla de reescritura $l=r\leftarrow u_1=v_1,...,u_n=v_n$, en R^C , una posición p en t y una sustitución inductiva σ tales que (1) $t_1/p=l\sigma$ (2) $t_2=t_1[r\sigma]_p$ *y* (3) $u_k\sigma\downarrow_{R^C,i}v_k\sigma$ para k=1,2,..n

Entonces $\rightarrow_{R^C} = \cup_{i\in\mathrm{N}}\rightarrow_{R^C,i}$

ii) Sea la secuencia $(\rightarrow_{R,i})_{i\in\mathrm{N}}$ de relaciones sobre $Term(sig,V^G)$ definida por

a) $\rightarrow_{R,0}=\rightarrow_{R^C}$
b) $t_1\rightarrow_{R,i+1}t_2$ if $t_1\rightarrow_{R^C}t_2$ o hay una regla de reescritura $l=r\leftarrow\Delta$ en R^D , una posición p en t_1 y una sustitución inductiva σ tales que (1) $t_1/p=l\sigma$ (2) $t_2=t_1[r\sigma]_p$ (3) para cada $u=v$ en Δ, $u\sigma\downarrow_{R,i}v\sigma$ (4) para cada $def(u)$ en Δ hay un término básico u' tal que $u\sigma\rightarrow_{R,i}^* u'$ ($\rightarrow_{R,i}^*$ es la clausula transitiva, reflexiva de $\rightarrow_{R,i}$) y (5) para cada $u\neq v$ en Δ hay términos básicos u',v' tales que $u\sigma\rightarrow_{R,i}^* u'$, $v\sigma\rightarrow_{R,i}^* v'$ y $u'\not\downarrow_{R^C}v'$

Entonces $\rightarrow_R = \cup_{i\in\mathrm{N}}\rightarrow_{R,i}$

Definición 5: Una especificación con constructoras $spec=(sig,C,R)$ es **admisible** si satisface:

a) $R=R^C\cup R^D$, donde R^C es un conjunto de reglas de constructoras R^D es un conjunto de reglas de definición
b) $\rightarrow_{\mathrm{R}}$ es confluente
c) Para cada $l=r\leftarrow\lambda_1,...,\lambda_n$ y para cada término t (no solo compuesta de constructoras), que ocurre con una negación en un literal λ_i hay otro literal λ_j=***def***(t).

Proposición 1: Sea $spec=(sig,C,R)$ una especificación admisible. Entonces $Term(sig,V^G)/\Leftrightarrow^*_R$ es un modelo de datos de *spec*.

Theorem 1: Sea $spec=(sig,C,R)$ una especificación admisible, y sea $t_1,t_2\in Term(sig,V^G)$. Entonces $t_1\sigma \leftrightarrow_{R,i}^* t_2$ si y solo si $DMod(spec)\models t_1=t_2$.

3.5.2. Interpretación para un conjunto de TADs en ManTa

Definición 6: Un conjunto de TADs sin parámetros que incluya el TAD BOOL es una especificación con constructoras.

Dado un conjunto finito de TADs básicos en ManTa (**BOOL** debe estar) $\{A_1, A_2, ... A_n\}$, sea S el conjunto de nombres de los TADs $\{name(A_1), name(A_2), ..., name(A_n)]$, F es el conjunto que contiene el nombre de todas las funciones $F = functions(A_1) \cup functions(A_2) .... \cup functions(A_n)$ y α es una función de aridad definida sobre F como $\alpha(f) = domain(f)\ codomain(f)$. Estos tres elementos conforman la signatura del conjunto de TADs $sig = (S, F, \alpha)$.

Sea C el conjunto de todas las constructoras (hay por lo menos una por tipo, puesto que todos son típos básicos), y R el conjunto de ecuaciones condicionales obtenida traduciendo cada axioma del conjunto $axioms(A_1) \cup axioms(A_1) \cup ... axioms(A_1)$.

Para traducir un axioma debemos usar las siguientes reglas:

Toda variable en una especificación ManTa es una variable de constructoras.

Toda ocurrencia de **EQUAL** debe ser sustituida por **EQ_**T donde T es el tipo de los argumentos (El símbolo **EQUAL** es sólo azucar sintactia)

Los axiomas que no incluyan **if** o **ERROR**, digamos de la forma $L=R$, se interpretan como:
$L=R \leftarrow def(Var(L)), def(R)$ Donde $Var(L)$ denota el conjunto de variables en la expresión L $\{v_1, v_2 ..., v_k\}$ y $def(Var(L))$ donota $def(v_1), def(v_2), ... def(v_k)$.

Los axiomas cuyo lado derecho sea **ERROR** no son traducidos (dado que representan términos donde la función es indefinida)

Los axiomas de la forma $L=$**if** $E1$ **then** $E2$ **else** $E3$ **fi**, pueden interpretarse como:
$L=E2 \leftarrow def(Var(L)), def(E1), E1=\mathbf{TRUE}, def(E2)$
$L=E3 \leftarrow def(Var(L)), def(E1), E1=\mathbf{FALSE}, def(E2)$

Para manejar claúsulas con **if** anidados, los dos últimos pasos pueden iterarse.
Un teorema por probar de la forma $l=r$ debe interpretarse como

$$l=r \leftarrow def(Var(l)), def(Var(r)), def(l), def(r)$$

con las mismas reglas de conversión expuestas para axiomas (las variables son variables de constructoras, **EQUAL** debe remplazarse con **EQ_**T, etc.)
La definición anterior no incluye el caso de extensiones (TADs sin constructoras), podemos aumentarla incluyendo las funciones de la extensión en el conjunto de funciones F, en la función de aridad α y traduciendo los axiomas como parte del conjunto de ecuaciones condicionales (No se debe incluir el nombre del TAD pues no tiene constructoras).

Ejemplo 1: Traduciremos el conjunto de TADs {**BOOL**, **NAT**} a una especificación con constructoras.

S={BOOL,NAT}, *F*={TRUE,FALSE,AND,OR,...,ZERO,SUC,ADD,ISZERO,DIV...} son los nombres de las funciones de **BOOL** junto con las de **NAT**.
La función de aridad α: $F \rightarrow S+$ está dada por:

α(TRUE)=BOOL
α(FALSE)=BOOL
α(AND)=BOOL BOOL BOOL
...
α(ZERO)=NAT
α(SUC)=NAT NAT
α(ADD)=NAT NAT NAT
α(ISZERO)=NAT BOOL
...

Con las definiciones anteriores la signatura ha sido construida *sig=(S,F,*α*)*.
Defina *C* como el conjunto de constructoras *C*={TRUE,FALSE,ZERO,SUC} y el conjunto R de ecuaciones condicionales puede obtenerse traduciendo los axiomas:

AND(B1,TRUE)=B1 ←*def*(B1)
AND(B1,FALSE)=FALSE ←*def*(B1),,*def*(FALSE)
...
ADD(N1,ZERO)=N1 ←*def*(N1)
ADD(N1,SUC(N2))=SUC(ADD(N1,N2)) ←
def(N1),*def*(N2),*def*(SUC(ADD(N1,N2)))
ISZERO(ZERO)=TRUE ←
ISZERO(SUC(N1))=FALSE ←*def*(N1)

...
Como ejemplo de la traducción de **if** anidados y axiomas con **ERROR** se traducir el axioma para **DIV(N1,N2)**. El axioma original es:

```
DIV(N1,N2) = if ISZERO(N2)
             then ERROR
             else if GE(N1,N2)
                  then SUC(DIV(SUBS(N1,N2),N2))
                  else ZERO
                  fi
             fi
```

Aplicando las reglas para **if** se obtiene

```
DIV(N1,N2)=ERROR ←
def(N1),def(N2),def(ISZERO(N2)),ISZERO(N2)=TRUE
DIV(N1,N2)=if GE(N1,N2) then SUC(DIV(SUBS(N1,N2),N2)) else ZERO
fi
  ← def(N1),def(N2),def(ISZERO(N2)),ISZERO(N2)=FALSE
```

La primera ecuación condicional puede quitarse (porque **ERROR** es un símbolo para especificar donde la función es indefinida). El segundo puede interpretarse, traduciendo la iguald con la regla de if y agregando las condiciones ya establecidas.

```
DIV(N1,N2)=SUC(DIV(SUBS(N1,N2)))
  ←
def(N1),def(N2),def(ISZERO(N2)),ISZERO(N2)=FALSE,def(GE(N1,N2))
,
     GE(N1,N2)=TRUE,def(SUC(DIV(SUBS(N1,N2))))
DIV(N1,N2)=ZERO
  <-
def(N1),def(N2),def(ISZERO(N2)),ISZERO(N2)=FALSE,def(GE(N1,N2))
,
     GE(N1,N2)=FALSE,def(ZERO)
```

Proposición 2: Un conjunto de TADs en ManTa es una especificación admisible con constructoras

Para verlo basta verificar cada condición de la definición 5:

a) $R=R^C\cup R^D$, donde R^C es un conjunto de reglas de constructoras yR^D es un conjunto de reglas de definición

En ManTa no hay reglas de constructoras, y todo axioma es traducible a una ecuación condicional sin literales de la forma $\neg$***def****(t)*.

b) $\rightarrow_R$ es confluente

Ya probamos que nuestro sistema de reescritura es confluente, debemos verificar que esta de acuerdo con la definición de $\rightarrow_R$
No tenemos reglas para constructoras, así que sólo debemos verificar las condiciones ii.b de la definición 4. .
(1) $t_1/p=l\,\sigma$ establece que existe un redex
(2) $t_2=t_1[r\,\sigma]_p$ Establece la reescritura de un redex empleando un axioma
(3) Para cada $u=v$ en Δ, $u\sigma\downarrow_{R,i}v\sigma$. Solo tenemos este caso en traducciones de instrucciones **`if`** (*E1*=**TRUE** o *E1*=**FALSE**), y el algoritmo de reescritura no reescribe un **`if`** a menos que su condición reescriba a **`TRUE`** o **`FALSE`** .
(4) Para cada ***def***(*u*) en Δ hay un término básico *u*' tal que $u\sigma \rightarrow_{R,i}^* u'$. La regla de reescritura innermos asegura que los componentes de una expresión sean evaluados a términos básicos antes de reescribir la expresión.
(5) Para cada $u\neq v$ en Δ hay términos *ground* *u*',*v*' tales que $u\sigma \rightarrow_{R,i}^* u'$, $v\sigma \rightarrow_{R,i}^* v'$ y $u' \not\downarrow_R^C v'$. En nuestro caso esto nunca ocurre puesto que los axiomas y teoremas nunca se traducen a ecuaciones condicionales con negaciones de igualdad.

c) Para cada $l=r\leftarrow \lambda_1,\ldots,\lambda_n$ y para cada término *t* (no sólo compuesto de constructoras), que ocurra dentro de una negación en un literal λ_i hay otro literal λ_j=***def***(*t*).

No tenemos literales con negaciones.

Proposición 3. Las extensiones a un conjunto de TADs son conservatias o constructor-

consistent
Esto es así porque no permitimos condiciones para las constructoras y los axiomas para funciones nunca dañan los tipos subyacentes. La semántica estática asegura esto, pues un símbolo de constructora no puede ser el más externo al lado izquierdo de un axioma y porque la confluencia local fuerte asegura que no haya contradicciones..

Proposición 4 Cualquier conjunto de TADs en ManTa tiene un modelo de datos.
Como todo conjunto de TADs en ManTa es una especificación admisible, por la proposición 1 tiene un modelo de datos

Proposición 5. La prueba de igualdades por reescritura tiene sentido para cualquier conjunto de TADs.

Esto es así porque el teorema 1 establece que la igualdad por reescritura representa la igualdad en los modelos de datos.

Proposición 6. La prueba de igualdades por inducción tiene sentido para cualquier conjunto de TADs en ManTa.

> Cuando se hacen pruebas por inducción deben introducirse nuevos axiomas (hipótesis de inducción). En ManTa no se permite hipotesis de inducción para constructoras, sólo para selectoras. Así que estas extensiones también son conservativas.

3.6 Semántica Translacional y Esquema de Traducción

3.6.1 Esquema de Traducción

Se ha concebido un esquema de traducción bastante general para traducir especificaciones ManTa a lenguajes de programación. El esquema consta de las siguientes etapas:

1. Definir la sintaxis del lenguaje de programación
2. Definir un AST del lenguaje de programación en ManTa
3. Escribir un Pretty Printer para el lenguaje, que recibe ASTs y retorne representaciones textuales (es decir un String)
4. Escribir algunas representaciones de TADs en el lenguaje para lograr eficiencia (esto no es necesario pero es altamente recomendable, especialmente para los tipos **`NAT`**, **`BOOL`**, **`Char`** y **`String`**)
5. Escribir en ManTa rutinas de conversión de AST ManTa a AST del lenguaje La rutina principal debe recibir un AbsType (ver capítulo 3.2) y producir un AST del lenguaje de programación objeto.
6. Generar código, integrar a rutinas que no pueden hacerse en ManTa y construir un ejecutable.

A manera de ejemplo se presenta un traductor de ManTa a Ocaml. (Este traductor se ha implementado, como puede consultarse en la sección 5.5)

3.6.2 Sintaxis de OCaml

Esta sintaxis está inspirada en [Leroy96] y [Leroy98]. Hemos omitido:

Características orientadas a objetos. El sistema de módulos, Streams, Rangos,
Chequeo de aserciones, evaluación retardada y apuntadores **`ref`**.

Una unidad de compilación es una interfaz (o signatura de módulo) y una implementación. La sintaxis de las implementaciones Ocaml es:

implCaml ::= {*implPhraseCaml* ;;}
implPhraseCaml ::= *exprCaml*
| *valueDefCaml*
| *typeDefCaml*
| *exceptionDefCaml*
| *directiveCaml*

La implementación de un módulo es una secuencia de frases, cada una terminada con doble punto y coma. Una frase es una expresión, una definición de un valor, una definición de tipo e excepción o una directiva. En tiempo de ejecución se evalúan en el orden en el que aparecen (las expresiones también son evaluadas).

La definición de tipos y excepciones introduce constructores de tipo, variantes y registros. El alcance lo conforman las frase que siguen la definición (no tienen efecto en tiempo de ejecución).

Las directivas modifican el comportamiento del compilador y las frases que le siguen. No tienen efecto en tiempo de ejecución ni en otros módulos. El nombre de cada módulo debe comenzar en mayúscula.

directiveCaml ::=
open *module_name*

Una *valueDefCaml* es un **let** [**rec**] *let-binding* {**and** *let-binding*} y permite asociar nombre a cuerpo de funciones. Su alcance comienza después de su definición y termina junto con el módulo. El nombre de toda función debe comenzar con mayúscula..

La definición de tipos debe seguir la sintaxis presentada a continuación:

typeDefCaml: **type** *typeBodyCaml* {**and** *typeDefCaml*}	Tipos mutuamente recursivos puede definirse con **and**.
typeBodyCaml: [*typeParamsCaml*] *ident* [*typeEquationCaml*] [*typeRepCaml*] {*constraintTypeCaml*}	El nombre de un tipo debe comenzar con minúsculas
typeEquationCaml: = *typeExpr*	Un tipo así definido es equivalente a una expresión de tipo. Pueden ser intercambiados. El resto de estilos de definición crea tipos incompatibles. `type id=string;;`
typeRepCaml: =*constrDeclCaml* {| *constrDeclCaml*} | ={ *labelDeclCaml* {; *labelDeclCaml*} }	El usuario puede definir variantes (suma directa de diversos tipos) o registros con etiquetas. `type num=Integer of int | Real of float;;` `type reg={nom:String;mutable sal:int};;`
typeParamsCaml: ' *ident* | (' *ident* {, ' *ident*})	Un tipo puede ser paramétrico `type 'a myList=NIL | CONS of 'a*('a myList);;`
constrDeclCaml: *ident* | *ident* **of** *typeExprCaml*	Un constructor de un tipo variante puede tener argumentos (ser una inyección). El nombre de todo constructor debe comenzar con mayúscula
labelDeclCaml: *ident* **:** *typeExprCaml* | **mutable** *ident* **:** *typeExprCaml*	Una etiqueta de un registro puede ser modificable usando la palabra reservada **mutable**.

constraintTypeCaml: **constraint** '*ident* **=***typeExprCaml*	Permite al usuario poner restricciones a los parámetros
typeExprCaml: ' *ident* | (*typeExprCaml*) | *typeExprCaml* **-***typeExprCaml* | *typeExprCaml* {**typeExprCaml*}+ | *ident* | *typeExprCaml ident* | (*typeExprCaml*{,*typeExprCaml*})*ident* | *typeExprCaml* **as** '*ident*	Una expresión de tipo representa un tipo y puede ser: Un parámetro (**'a**), una función de un tipo en otro (**int -> float**), un producto cartesiano (**int*string**), un tipo existente (**int**) un tipo paramétrico existente (**int list**) o puede ser una asociación con un identificador (la primera letra del identificador debe ser minúscula).

Al definir excepciones se agregan nuevos constructores al tipo *built-in* **exn**. La sintaxis es:

exceptionDefCaml:
exception *constrDeclCaml*

En cuanto a las expresiones, su sintaxis se presenta a continuación:

exprCaml:

value-path	Un identificador asociado a un valor (puede tener la vía del módulo) e.g. **var1**
| *constantCaml*	e.g. 5
| (*exprCaml*)	e.g. (3+6)
| begin *exprCaml* end	Para hacer bloques
| (*exprCaml*:*typeExprCaml*)	e.g. **5:int**
| *exprCaml* ,*exprCaml* {, *exprCaml*}	Productos Cartesianos e.g. **(4,3.14,"w")**
| *ident exprCaml*	Constructores con argumentos e.g **CONS(2,NIL)**
| *exprCaml* ::*exprCaml*	Constructor de lsitas e.g. **3::4::[]**
| [*exprCaml* {; *exprCaml*}]	Notación para listas e.g **[3;4]**
| [| *exprCaml* {; *exprCaml*} |]	Notación para vectores e.g. **[|1;2;3|]**
| { *ident*=*exprCaml* {;*ident* =*exprCaml*} }	Registro, el orden no importa e.g. **{nombre="J";salario=2}**
| *exprCaml exprCaml*	Aplicación funcional. e.g. **f 6**
| *prefix-symbol exprCaml*	Los símbolos prefijos e infijos son tratados como funciones, de forma que pueden ser redefinidos. Los operadores prefijos predefinidos son la negación entera y flotante - -.
| *exprCaml infix-op exprCaml*	Los operadores infijos se presentan más
| *exprCaml* . *ident*	Evalua el campo *label* del registro *expr* e.g. **r.nombre**

| *exprCaml* . *ident* <- *exprCaml*	Modifica un campo **mutable** e.g. **r.salario<-5**
| *exprCaml* .(*exprCaml*)	Evalua una posición de un vector, la primera expresión es el vector y la segunda el índice. e.g. **v.(i)**
| *exprCaml* .(*exprCaml*)<-*exprCaml*	Modifica una posición de un vector e.g. **v.(i)<-8**
| *exprCaml* .[*exprCaml*]	Evalua una posición de una cadena
| *exprCaml* .[*exprCaml*]<-*exprCaml*	Modifica un caracter de una cadena
| if *exprCaml* then *exprCaml* [else *exprCaml*]	Condicional, el primer argumento debe ser de tipo bool y los otros dos deben tener el mismo tipo. e.g `if a=b then 0 else 1`
| while *exprCaml* do*exprCaml* done	Evalua la segunda expresión mientra la primera sea **true**. Siempre evalua a (). e.g. `while true do print_string "etern" done`
| for *ident*=*exprCaml* (to | downto) *exprCaml* do *exprCaml* done	Evalua la tercer expresión, mientras el identificador ident va de la primera expresión a la segunda e.g. `for k=1 to 10 do (print_int k) done`
| *exprCaml* ; *exprCaml*	Secuencia, evalua la primera expresión, luego la segunda y retorna el valor de la segunda e.g. `print_int a;a`
| match *exprcaml* with*patternMatchCaml*	Evalua la primera expresión y la compara con los patrones, el primero que unifique determina el valor retornado.
| function *patternMatchCaml*	Define una función de un argumento
| fun *multipleMatchCaml*	Define una función de uno o más argumentos
| try *exprCaml* with*patternMatchCaml*	Evalua la primera expresión, si una excepción ocurre intenta unificarla con los patrones para tratarla.
| let [rec] *letBindingCaml* {and*letBindingCaml*} in *exprCaml*	Asocia nombres a valores que pueden ser usados en la última expresión.

Las constantes se especifican continuación:

constantCaml:	Los enteros varian de -2^30 a 2^30 -1.
integer-literal	Los valores flotantes depende de la implementación
| *float-literal*	Los caracteres son de 8 bits
| *char-literal*	Las cadenas son secuencias de caracteres, hasta con 2^16-1 caracteres
| *string-literal*	
| *cconstr*	Las constructores de tipos variantes también valen como constantes, los constructores predefinidos son: `true`, `false`, `[]`, `()`.

A continuación se presenta la sintaxis de algunos símbolos antes introducidos:

patternCaml:
value-name
| _
| *constant*
| *patternCaml* **as***value-name*
| (*patternCaml*)
| (*patternCaml* : *typexpr*)
| *patternCaml* | *patternCaml*
| *ident patternCaml*
| *patternCaml* {,*patternCaml*}
| { *ident* = *patternCaml* {;*ident* = *patternCaml*} }
| [*patternCaml* {;*patternCaml*}]
| *patternCaml* **::** *patternCaml*

patternMatchCaml:
patternCaml [**when** *exprCaml*]**-***exprCaml*
{| *patternCaml* [**when** *exprCaml*] **-***exprCaml*}

multipleMatchCaml:
{*patternCaml*}+ [**when** *exprCaml*]**-***exprCaml*

letBindingCaml:
patternCaml = *exprCaml*
| *value-name* {*patternCaml*}+ [**:** *typeExprCaml*] **=** *exprCaml*

Ahora se presenta los operadores y constructores con su precedencia:
.

Constructor / operador	**Signiricado**
símbolos prefijos, !	! is un operador para desreferencia (**ref** objects)
. .(.[	
Aplicación funcional	
Aplicación de constructores	
**	Potencia
*. * / /. % mod	Producto, Division, Residuo
+. + -. -	Adición, Substracción
::	Constructo de listas
@ ^	Concatenación de listas y cadenas
= <> == != < <= = > >=	Igualdad estructural, diferencia estructural, igualdad física, diferencia físical, menor que, menor o igual, igual, mayor, mayor igual

land lor lxor lsl lsr asr	logical and, logical or, xor, logical shift to left, logical shift to right, aritmetical shift to right
not	
& &&	
or ||	
,	Constructor de tuplas. Una tupla puede tener hasta 2^14 - 1 componentes
<- :=	Etiquetas y asignación de referencias
if	Secuencia
;	
let match fun function try	

Una interfaz o especificación de módulo es una secuencia de especificaciones. Una especificación es la signatura de una función o constante, una definición de tipo, de excepcion o una directiva **open**:

interCaml:
{*specificationCaml* [;;]}

specificationCaml:
val *value-name*:*typeExprCaml*
| *typeDefCaml*
| *exceptionDefCaml*
| *directiveCaml*

3.6.3 AST DE OCAML

Casi todos los símbolos no terminales de la gramática presentada son TADs:.

- ConstantCaml
- TypeExprCaml LTypeExprCaml
- LabelPatternCaml LLabelPatternCaml
- PatternCaml LPatternCaml
- CasePMCaml LCasePMCaml
- PatternMatchCaml
- MultipleMatchCaml
- LabelExprCaml LLabelExprCaml
- LetBindingCaml LLetBindingCaml
- ExprCaml LExprCaml
- ConstraintTypeCaml LConstraintTypeCaml
- LabelDeclCaml LLabelDeclCaml

- ConstrDeclCaml LConstrDeclCaml
- TypeRepCaml LTypeRepCaml
- TypeBodyCaml LTypeBodyCaml
- TypeDefCaml
- ValueDefCaml
- DirectiveCaml
- ExceptionDefCaml
- ImplPhraseCaml LImplPhraseCaml
- ImplCaml
- SpecificationCaml LSpecificationCaml
- InterCaml

La signatura de cada uno de estos TADs puede consultarse en el Anexo C.

3.6.4 PRETTY PRINTER

Todo tipo del AST presentado cuenta con una función para imprimir una cadena con la sintaxis concreta. El nombre de tales funciones es **StringOf*T*** donde ***T*** es el TAD (e.g. **StringOfImplCaml**).

3.6.5. REPRESENTACIONES

Algunos tipos de ManTa tiene una representación natural en Ocaml.
`NAT` puede representarse con el tipo de Ocaml `int`, `BOOL` con `bool`, `Char` con `char`, `String` con `string` y `List` con el tipo `list`. Estas representaciones se han escrito en los TADs NATExprCaml, BOOLExprCaml, CharExprCaml, StringExprCaml y ListExprCaml, que pueden consultarse en el Anexo C.

Además de representaciones simples para los generadoras y selectoras, los términos constantes (compuestos solo por generadoras) de los tipos `NAT`, `BOOL`, `Char` `y` `String` son convertidos a constantes en OCaml (e.g `SUC(SUC(SUC(ZERO)))` se traduce a 3, `charOfNAT(65)` se traduce a `'A'`)

Las representaciones mencionadas se usan junto con rutinas para identificar, en ASTs de ManTa, rutinas con representaciones. Tales rutinas están en el TAD RepCaml.

3.6.6 RUTINA DE CONVERSIÓN

Los generadores de un TAD son convertidos a constructores de un tipo variante. e.g. Los generadores de **NAT** son **ZERO:->NAT** y **SUC:NAT–NAT**, así que son convertidos al tipo:

```
type nat=ZERO | SUC of nat ;;
```

Si un TAD es genérico el tipo traducido es paramétrico. Por ejemplo el siguiente TAD representa un árbol binario genérico, puede ser vacío o tener un hijo izquierdo, una raíz y un hijo derecho

```
ADT BTtree[X]
  INITIALS
    empty:–BTree
  CONSTRUCTORS
    consBTree:BTree*X*Btree–BTree
```

La traducción a OCaml produce el siguiente tipo:

```
type 'x btree=Empty | ConsBTree of ('x btree)*'x*('x btree) ;;
```

Los selectores de un TAD son traducidos a funciones, por ejemplo la función para sumar dos número naturales definida en el TAD NAT es:

```
ADD(ZERO,y)=y
ADD(SUC(x),y)=SUC(ADD(x,y))
```

y se traduce a

```
let rec add=function
    (ZERO,y) – y
  | (SUC(x),y) – SUC(add(x,y))
;;
```

Así los lados izquierdo de todo axioma se traducen en casos de pattern matching, y los lados derechos a expresiones Caml

Un tipo completo debe traducirse a una interfaz y una implementación:

Una interfaz se compone de:

1. Algunas directivas **open** para abrir módulos empleados en la signatura
2. Si el tipo no es una extensión, una definición de tipo
3. Un **val** por cada selectora para declarar su tipo

Una implementación se compone de:

1. Algunas directivas **open** para abrir otros módulos usados
2. Una definición de tipo si no es extensión
3. Un **let** o `let rec` por cada selector

ManTa sigue convenciones de nomenclatura tomadas de Java y Z: los tipos comienzan con mayúscula y las funciones con minúscula (incluyendo las generadoras de tipos). Sin embargo Ocaml tiene convenciones diferentes: Los tipos deben comenzar con minúscula, los constructores de tipos variantes deben comenzar con mayúscula, los parámetros de tipos, las funciones y las variables con minúscula. Los nombres de los módulos deben comenzar con mayúscula. Los nombres de TADs, funciones y generadores deben ser alterados para seguir estas convenciones.

4 MANTA 3.1 - LA PRIMERA VERSIÓN COMPLETA

Esta versión se viene trabajando desde 1997 a partir de las fuentes legadas. Aún cuando a la vista de un usuario no hay cambios mayores, en el fondo hay muchas mejoras:

- Relacionadas con las bases teóricas
  - Cambios a la gramática del lenguaje ManTa como consecuencia de las nuevas bases teóricas
  - Nueva semántica para generadores semiautomáticos de TADs: restricciones, sumas directas, completaciones y variaciones
  - El esquema de representaciones funciona
  - Tabla de símbolos y reglas de nomenclatura
  - Mejoras en la definición de axiomas para asegurar que son lineales izquierdas, localmente confluentes y que cubren el dominio de la selectora que definen.
  - Nuevo algoritmo de reescritura
- Relacionadas con la interfaz
  - La interfaz EMACS vuelve a funcionar
  - Independiente de plataforma, se ha probado en muchas arquitecturas y con diversos compiladores con éxito
- Relacionadas con el código fuente
  - Interfaz independiente del núcleo y API para usarlo facilmente
  - Código fuente menos voluminoso
  - Mejor administración de memoria
  - Uso metódico de assert
  - Reformateo de las fuentes y mejoras a la documentación
  - Mas de 40 fallas corregidas (ver `http://wwwisc.uniandes.edu.co/ ~manta/bugs.html`)
- Relacionadas con el ambiente de desarrollo
  - Cuenta con un plan de desarrollo de software
  - Nueva jerarquía de directorios y nomenclatura de archivos, apropiadas para el desarrollo multiplataforma y control de versiones
  - Uso de RCS para control de versiones
  - Tiene programas de prueba para núcleo e interfaz
  - Cuenta con un benchmark para evaluar el desarrollo del sistema y su desempeño en diversas plataformas

4.1 REQUERIMIENTOS

- Ofrecer una versión realmente completa y funcional del sistema ManTa, con las características prometidas desde la versión 2.0
- Independiente de plataforma
- Con facilidades para configurar en otros idiomas
- Con el núcleo separado de la interfaz y mejor ordenado para facilitar la creación de nuevas interfaces
- Debe servir como plataforma de desarrollo para las siguientes versiones de ManTa

4.2 PLAN DE DESARROLLO

Para lograr los requerimientos mencionados se han desarrollado y liberado versiones Beta, cada una de las cuales cumple más requerimientos:

Desarrollo	Descripción
ManTa 3.1 Beta 1	Independencia de plataforma, protocolo para portarlo a nuevas plataformas
ManTa 3.1 Beta 2	Corrección de varias fallas (14)
Bases teóricas	Esta fue la etapa más demorada, pues se intentaron varias aproximaciones que no dieron resultados. Se lograron unas bases satisfactorias después de un año de trabajo en equipo con Klaus Madlener y Rodrigo Cardoso. Cesar Muñoz también ha colaborado en este aspecto.
ManTa 3.1 Beta 3	Primeras mejoras debidas a las bases teóricas: Nueva semántica para operaciones y mecanismo de representaciones apropiado. Se corrigen algunas fallas (7)
ManTa 3.1 Beta 4	Cumple todas las mejoras propuestas a las bases teóricas y corrige varias fallas (18)

Infortunadamente, no todas las fallas se han logrado solucionar. Hay dos bastante graves, cuya solución podría requerir un esfuerzo mayor al de construir de nuevo el sistema completo:

- Fallas de memoria en las fuentes legadas, debido a inexistencia de políticas para manejo de memoria.
- Fallas de memoria en los prototipos generados en C, aún cuando tales prototipos se generan con un recolector de basura primitivo.

4.3 IMPLEMENTACIÓN

En el capítulo 2.2.3 se han mencionado algunas técnicas que han permitido recuperar la versión legada, a continuación se describen algunas con más detalle. Para ver un ejemplo de su aplicación puede consultarse el Apéndice A o revisar las fuentes del sistema, disponibles en **`http://wwwisc.uniandes.edu.co/~manta`**

4.3.1 INTERNACIONALIZACIÓN

Se ha mantenido el mismo esquema de internacionalización de la versión legada, este consiste en tener archivos de mensajes en diversos idiomas y un archivo de configuración donde el usuario determina que archivo de mensajes planea usar. En todas las fuentes los mensajes se referencian por números, y hay un módulo que maneja e imprime los mensajes (módulo **`interfaz.c`**)

4.3.2 MANEJO DE ERRORES

Las funciones de los nuevos módulos de ManTa, tienen un manejo de error inspirado en UNIX:

- Toda función que pueda fallar retorna un entero (código de error, donde 0 representa que no hubo errores). Los valores que deba retornar la función lo hacen en parámetros que pasan por referencia.
- Las funciones que puedan producir errores reciben un parámetro adicional, se trata de un buffer de errores donde la función podrá agregar mensajes de error (empleando los mensajes identificados con números que aseguran internacionalización).
- Si una función invoca a otra que pueda producir errores, después de la invocación debe chequear el error y manejarlo si le es posible o salir retornando el código de error que recibe y el buffer de errores.

4.3.3 NÚCLEO INDEPENDIENTE DE INTERFAZ

Ninguna función del núcleo imprime o efectúa acciones relacionadas con la consola. Todos los mensajes dirigidos al usuario se manejan con el buffer de errores mencionado en la sección anterior, esto se había logrado con la versión 3.0 (ver [Aponte95]), y aún más con la versión 3.2 (ver [Tamara97]).
Por otra parte se desarrollo una especie de API (módulo **`apimanta.c`**), que permite al programador de la interfaz acceder a las funciones del núcleo de forma homogénea y segura. Este módulo divide las funciones del núcleo en diversos TADs y requiere que al comienzo del programa se haga una invocación a una rutina de inicialización (**`iniMantaAPI`**) y al final se invoque una rutina de terminación (**`finMantaAPI`**). Pueden consultarse más detalles en [Tamara97].

4.3.4 Independencia de plataforma

Se ha logrado, agrupando todas las funciones dependientes de plataforma en un sólo módulo que varia de plataforma a plataforma (las diversas plataformas se manejan empleando una estructura de directorios apropiada, ver [Jam94]), así mismo la nomenclatura a las fuentes y la estructura de directorios es apropiada para desarrollo multiplataforma.
Para portar ManTa de una plataforma a otra basta implementar algunas funciones, en la nueva arquitectura (e.g. copiar un archivo, archivos en un directorio, touch, etc.), pueden consultarse los detalles en [Tamara97]: Empleando esta técnica se ha logrado compilar y probar ManTa en las siguientes plataformas:

MAQUINA	**SISTEMA OPERATIVO**	**COMPILADOR**
PC (≥386)	DOS 32 bits Windows 3.x Windows 95/NT/98	DJGPP V 2
PC (≥386)	Windows 95/NT/98	CYGNUS C++ Beta 17-19
PC (≥386)	Windows 95/NT/98	BORLAND C++ 4.52/5.0
PC (≥386)	Windows 95/NT/98	Visual C++ 5.0
PC (≥386)	Linux, X-Window	GNU C++ 2.6.3,2.7.2,2.8.1
SUN SPARC	SOLARIS	GNU C++ 2.7.2
MIPS	ULTRIX	GNU C++ 2.6.3

4.3.4 Depuración del código legado

Para detectar y prevenir errores se ha empleado **`assert`** metódicamente (Ver [Cod85]), se ha compilado en diversas plataformas, se han verificado localizaciones de memoria y aperturas de archivos. Pueden consultarse más detalles en [Tamara97]

4.4 Pruebas

Esta etapa es nueva en ManTa, se ha introducido un programa para hacer pruebas de regresión al núcleo junto con ManTa 3.1b3 y otro para probar la interfaz con ManTa 3.1b4.

4.4.1 Pruebas al núcleo

Las pruebas al núcleo las realizan cuatro programas:

Programa	**Pruebas efectuadas**
`mantst1`	Pruebas al módulo de tipos. Carga más de 70 especificaciones que prueban diversos casos de especificaciones.

mantst2	Pruebas al módulo de tipos, específicamente a las funciones para generar nuevos tipos: copia, producto, restricciones, suma directa, completación, variación, Instanciación y representaciones.
mantst3	Pruebas al motor de reescritura y el demostrador de teoremas. Prueba diversos casos de reescritura y varias técnicas para demostrar teoremas.
mantst4	Pruebas al generador de código. Entre otras pruebas genera un programa completo, se trata de un Pretty Printer de archivos con especificaciones ManTa, este ejemplo evidenció la posibilidad de realizar un bootstrap.

Estos programas generan archivos con los resultados de las pruebas que deben compararse con archivos con información correcta.

4.4.2 PRUEBAS DE LA INTERFAZ PLANA

La interfaz plana de ManTa 3.1, requiere un editor de texto para interactuar con el usuario. Se agregó como editor de texto posible la consola, de forma que puede redireccionarse la entrada y la salida de ManTa (a partir de la versión Beta 4).
Las pruebas consisten en redireccionar la entrada de una sesión de ManTa con un archivo que contiene secuencias de teclas para probar todas las opciones del sistema, y redireccionar la salida a otro archivo que después es comparado con un archivo ya existente que tiene los resultados esperados.

4.5 DIFUSIÓN Y DOCUMENTACIÓN

Se ha creado un sitio WEB de ManTa, donde se mantiene información actualizada del sistema y de donde pueden obtenerse documentación, el sistema precompilado y las fuentes, que se distribuyen bajo la GNU General Public License.

La documentación que se tiene disponible y hace parte de la distribución de ManTa 3.1 es:

- Descripción formal de ManTa 3.1 (ver capítulo 3)
- Manual del usuario de la versión plana
- Documentos de compilación e instalación para cada plataforma soportada
- Historia de cambios
- Soluciones a problemas comunes
- Fallas conocidas

Uno de los componentes importantes de este sitio WEB es el sistema para reportar bugs que incluye, donde pueden consultarse los bugs descubiertos, el estado de cada uno y pueden reportarse nuevos bugs.

4.6 EVALUACIÓN Y MÉTRICAS

Para evaluar el desempeño de esta versión y compararlo con futuras versiones se ha desarrollado el ManTa Benchmark.

Esta herramienta desarrollado junto con la versión 3.1b3 y distribuida con la versión 3.1b4 cumple los siguientes objetivos:

- Establece algunas métricas para el kernel de ManTa:
  - Poder del probador de teoremas
  - Tiempo y espacio consumido por los módulos de reescritura y prueba de teoremas
  - Administración de memoria de todo el kernel de ManTa
- Sirve como medida común para diversas plataformas (incluyendo arquitectura, sistema operativo y compiladores)

Se ha construido a partir del Benchmark NATHM92, este benchmark se compone de casi 1900 definiciones y lemas escritos en un lenguaje funcional sin tipos.
La traducción se ha hecho semi-automáticamente, dando resultado a 546 lemas y 210 definiciones (algunas definiciones y conjeturas no pudieron traducirse porque son débilmente tipados, mientras que ManTa es fuertemente tipado).

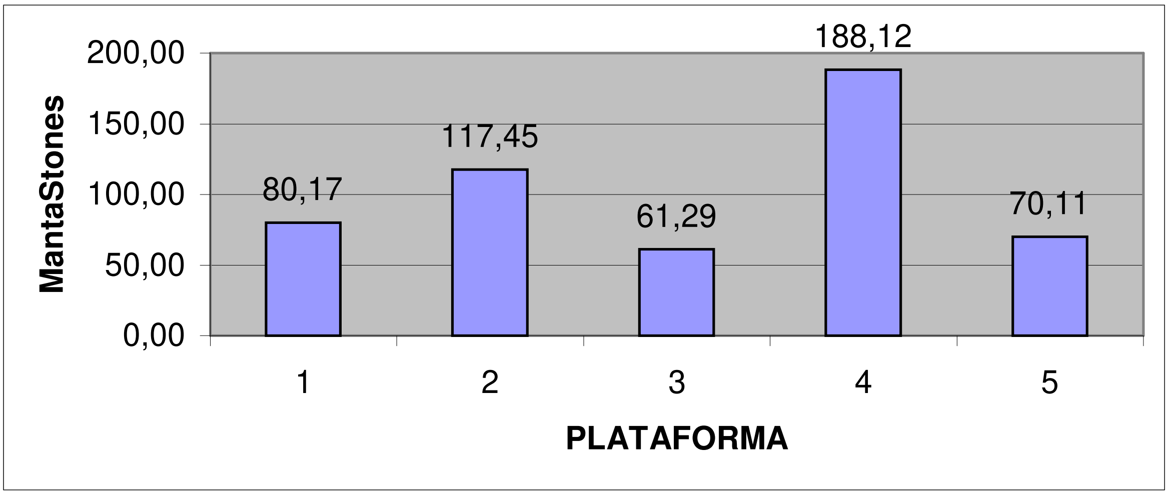
El primer resultado que se ha obtenido es una comparación entre las versiones 3.1b3 y 3.1b4. El ambiente de prueba fue un Pentium II de 300mhz, con 128MB en RAM y 100MB en memoria Virtual, Sistema Operativo Windows 95 y binarios compilados con Borland C++ 5.0.

Concepto	Medida	3.1 Beta 3	3.1 Beta 4
Tamaño del Ejecutable	Bytes	650496	554582
Conjeturas		193	193
Probadas		18	19
No Probadas		174	172
Problemas		1	2
Tiempo en pruebas	Segundos	48,24	20,99
Tiempo promedio por Conjetura	Segundos	0,27564	0,120672
Capacidad de prueba	%	9,326425	9,8446

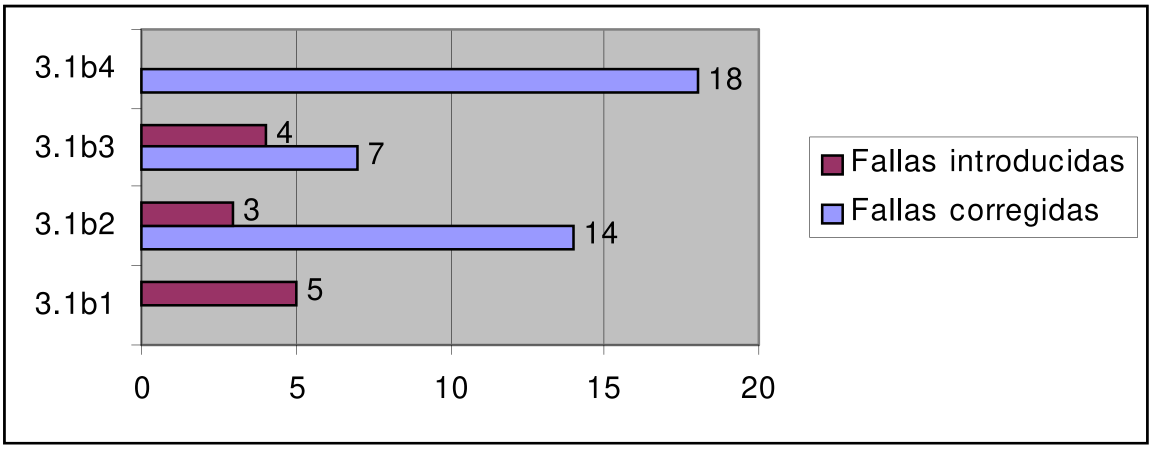
Como puede apreciarse es más pequeño el nuevo código, es más rápido en las pruebas y puede hacer una prueba más, lo que aumenta su capacidad de prueba en 0.5 puntos. (La gran diferencia está en el nuevo módulo de reescritura que tiene la versión 3.1b4).

A continuación presentamos los resultados del Benchmark empleando el kernel 3.1b4 y aplicado en diversas plataformas, (se dejan las conclusiones al lector):

1	2	3	4	5
Pent. II - 300MHz	Pent.II - 300MHz	Pent.II - 300MHz	Pent.II - 266MHz	Pent. II - 300MHz
Windows 95	Windows 95	Windows 95	Windows 98	Linux RedHat
128MB	128MB	128MB	64MB	64MB
100MB	100MB	100MB	100MB	0MB
BC++ 5.0	DJGPP V2	Cygnus C++ B19	DJGPP V2	GCC 2.7.3



Otra medida que se ha logrado establecer con el sistema para seguimiento de fallas es la cantidad de fallas corregidas en introducidas en cada versión:



5 MANTA 3.3 - HACIA EL PRIMER BOOTSTRAP

Esta versión, en construcción en el momento de este escrito, se ha desarrollado empleando la versión 3.1 para especificarla. No hay mejoras en las bases teóricas, así que la descripción formal presentada en el capítulo 3 es apropiada para esta versión.

5.1 REQUERIMIENTOS

- Debe ser especificado en ManTa y desarrollado con ayuda de la versión 3.1
- El sistema generado debe tener por lo menos las mismas características de la versión 3.1 pero menos errores (especialmente relacionados con memoria)
- Debe ser independiente de plataforma, configurable en diversos idiomas y el núcleo debe ser independiente de la interfaz

5.2 ANÁLISIS DE REQUERIMIENTOS

Se ha decidido cambiar el lenguaje de programación. El nuevo lenguaje por emplear es Ocaml algunas de cuyas bondades se presentan a continuación

- Es sencillo y tiene alguna similitud con ManTa, de forma que la traducción es directa
- Tiene un recolector de basura, de forma que no habrá problemas de memoria
- Es independiente de plataforma, hay versiones para PC, Unix y MAC, de forma que el sistema generado resultará también independiente de plataforma.
- Recientemente ha ganado en una competencia contra C++, Dylan, Haskell, etc.

Así como la mayoría de lenguajes y a diferencia de ManTa 3.1, la versión 3.3 se compondrá de varias herramientas independientes :

Nombre Clave	Herramienta	Descripción
mpp	Preprocesador de ManTa	Es un preprocesador que facilita la escritura de nuevos TADs
cgen	Generador de código en C	El mismo de la versión 3.1, pero como herramienta independiente. (Así ofrecemos al menos la misma funcionalidad de la versión 3.1). Sus fuentes quedan en C.

ocamlgen	Generador de código en Ocaml	Especificado en ManTa y generado en dos etapas con cgen y el mismo. De forma que sus fuentes estén en OCaml
manrew	Motor de Reescritura	Es el algoritmo de reescritura, especificado en ManTa y con fuentes en Ocaml
mancomp	ManTa Compiler	Es un chequeador de la semántica estática (ver sección 3.3) especificado en ManTa y generado en Ocaml
mantheo	Probador de teoremas	Probador de teoremas legado de la versión 3.1, con fuentes en C.

En la versión 3.1 no hay claridad sobre el estado. Para esta versión se definirá el estado como una tabla de símbolos, directorios de configuración y archivos de configuración y mensajes.

La tabla de símbolos se construirá dinámicamente a partir de información de un directorio: el directorio de tipos.

Una de las características de la versión 3.1 era emplear un código intermedio, ilegible para el usuario final, pero fácil de reconocer para un programa. Este código intermedio, no es necesario si se emplean herramientas para generar lexers y parsers eficientes como **`ocamllex`** y **`ocamlyacc`**. Así los tipos se mantendrán como especificaciones, tal y como las digite el usuario. Una ventaja de esto es que no se pierden los comentarios del usuario, como ocurría en la versión 3.1.

5.3 MPP - UN PREPROCESADOR PARA MANTA

La gramática de ManTa hace difícil escribir números, caracteres y cadenas (por ejemplo para representar el 4 debe escribirse `SUC(SUC(SUC(SUC(ZERO))))` como en notación de Peano), esta restricción facilita las demostraciones pero dificulta la escritura. El preprocesador alivia esta labor.

El preprocesador es un transformador de textos que trabaja sólo a nivel léxico, sin chequear la sintaxis del los archivos (así puede usarse en archivos con TADs, con términos por reescribir o conjeturas), las reglas que sigue son:

- Los comentarios no son alterados
- Los caracteres entre apóstrofes (e.g `'a'`), son traducidos a `charOfAscii(`*N*`)` donde *N* es el código ASCII del carácter en notación de Peano.
- Se permiten como caracteres algunas secuencias especiales: `'\n'` LF, `'\r'` CR , `'\t'` tabulador, `'\b'` *backspace*, `'\"'` comillas , `'\\'` *backslash*, `'\''` apostrofe, `'\d'` con `d` número en decimal representa el caracter con ASCII `d`.
- Las cadenas deben estar entre comillas (e.g. `"Esta es una cadena"`) y son traducidas a listas (con constructoras `nil_string` y `cons_string`). Las cadenas pueden incluir además los mismos caracteres especiales antes descritos.
- Los enteros positivos escritos en decimal son convertidos a la notación de Peano.

Este preprocesador se ha especificado y generador con la herramienta **ocamllex**. A continuación se presenta la especificación de este generador (faltan algunas funciones auxiliares para convertir los datos analizados):

```
rule get_token = parse

  "/*"                  { "/*"^(comment lexbuf) }
| ['a'-'z' 'A'-'Z' '_'] ['a'-'z' 'A'-'Z' '_' '0'-'9']*
                        { Lexing.lexeme lexbuf }
| "'" '\\' _ "'"        { one_esp_char (Lexing.lexeme lexbuf) }
| "'" '\\' ['0'-'9'] ['0'-'9']? ['0'-'9']? "'"
                        { one_esp_char (Lexing.lexeme lexbuf) }
| "'" _ "'"             { one_char (Lexing.lexeme lexbuf) }
| "\""                    { string_literal_str:="";string lexbuf }
| ['0'-'9']*                    { one_nat (int_of_string (Lexing.lexeme lexbuf)) }
| eof                   { raise EndOfFile }
| _                     { Lexing.lexeme lexbuf }
and comment = parse
  "*/"                  { "*/" }
| eof                   { raise StmtLexicalError }
| _                     { let i=(Lexing.lexeme lexbuf) in i^(comment lexbuf)}
and string = parse
  "\\\""                { string_literal_str:=((!string_literal_str)^"\"");string lexbuf }
| "\\n"                 { string_literal_str:=((!string_literal_str)^"\n");string lexbuf }
| "\\r"                 { string_literal_str:=((!string_literal_str)^"\r");string lexbuf }
| "\\t"                 { string_literal_str:=((!string_literal_str)^"\t");string lexbuf }
| "\\b"                 { string_literal_str:=((!string_literal_str)^"\b");string lexbuf }
| "\\'"                 { string_literal_str:=((!string_literal_str)^"'");string lexbuf }
| "\\\\"                { string_literal_str:=((!string_literal_str)^"\\");string lexbuf }
| '\\' ['0'-'9'] ['0'-'9']? ['0'-'9']?
  { let t=(Lexing.lexeme lexbuf) in
    let ss=(String.make 1 (Char.chr (int_of_string (String.sub t 1 ((String.length t)-1) ))))
    in string_literal_str:=((!string_literal_str)^ss);string lexbuf }
| "\""                  { string_literal (!string_literal_str) }
| eof                   { raise StmtLexicalError }
| _                     { string_literal_str:=((!string_literal_str)
                                       ^(Lexing.lexeme lexbuf));string lexbuf }
```

5.4 CGEN - GENERADOR DE CÓDIGO C

El API descritó en la sección 4.3.3, ha facilitado el desarrollo de este generador de código tanto, que el cuerpo principal de este generador puede escribirse en unas cuantas líneas:

```
if (iniMantaAPI("manta.ini")<0)
  {
    /* Manejo de error */
  }
if ((et=loadTextType(argv[1]))==NULL)
  {
    /* Manejo de error */
  }
if ((rc=generaCProt(et,CP_TODO))!=0)
  {
    /* Manejo de error */
  }
```

Sin embargo, el núcleo de la versión 3.1 es monolítico y debe unirse completo para poder compilar este generador. En consecuencia el tamaño del ejecutable final es 672KB y la compilación de las fuentes requiere analizar cerca de 217000 lineas (de acuerdo a BC ++ 5.0).

Como lo indica el bug 42 (ver `http://wwwisc.uniandes.edu.co/~manta/bugs.html`), el código generado por ManTa 3.1 tiene problemas de memoria, así que este programa y los que este genera, infortunadamente, tienen problemas de memoria.

5.5 OCAMLGEN - GENERADOR DE CÓDIGO OCAML.

5.5.1. DISEÑO EN MÓDULOS

La traducción de ManTa a Ocaml puede hacerse empleando módulos como los presentados en la Figura 1. Un parser de ManTa lee especificaciones ManTa y genera un AST, un traductor entonces lo convierte en un AST de OCaml y un Pretty Printer escribe los archivos Ocaml correspondientes.

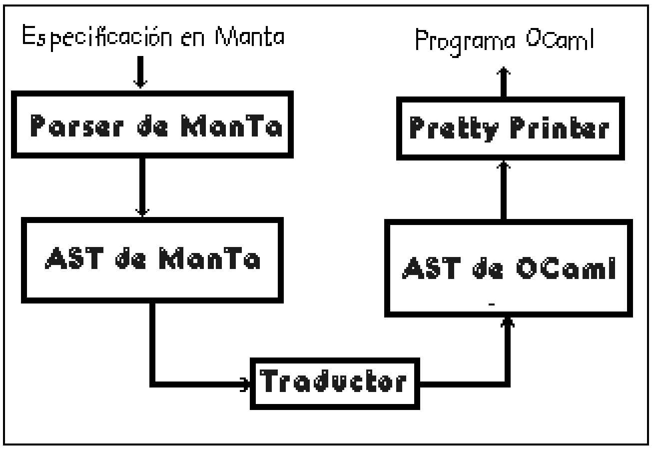


Figura 1. Módulos del traductor a OCaml

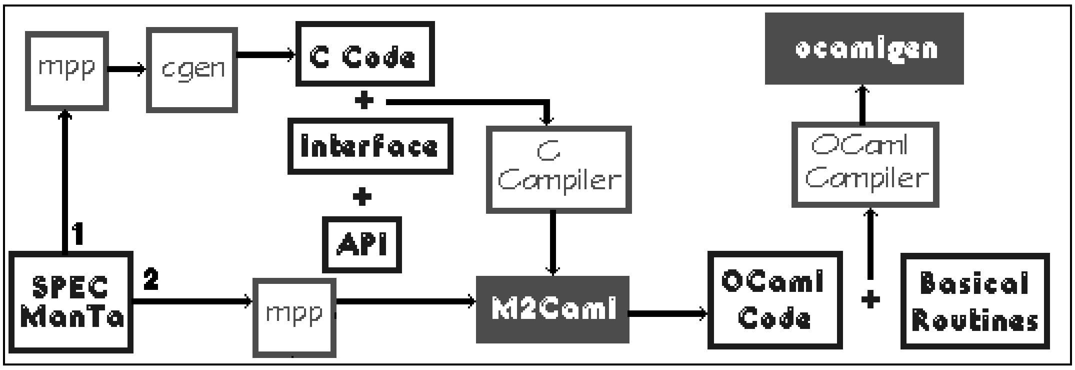
De estos módulos el de AST es requerido por todas las otras herramientas (excepto el preprocesador), así que podrá ser reutilizado

5.5.2 ESPECIFICACIÓN

El *parser*, infortunadamente, no se ha hecho en ManTa sino en el lenguaje de la herramienta **`ocamlyacc`**. La especificación del AST de ManTa puede consultarse en el capítulo 3.2. El AST de Ocaml, su pretty printer y el traductor se han especificado enteramente en ManTa (ver capítulo 3.6). A manera de ejemplo algunas de las especificaciones se han dejado en el anexo B, donde estan en su enteridad las funciones de traducción de ASTs.

5.5.3 IMPLEMENTACIÓN

El proceso seguido para construir ocamlgen ha sido una doble generación de código, como se presenta en la figura 2.



El proceso comienza con la especificación en ManTa del traductor, que se pasa por el preprocesador y el generador de código en C. El código obtenido se une con el kernel de ManTa 3.1 y una interfaz sencilla. Esto se compila para generar un ejecutable (al que le hemos dado el nombre clave **`M2Caml`**). Este ejecutable recibe especificaciones en ManTa y las traduce a programas Ocaml.

La segunda fase del proceso consiste en pasar la especificación del traductor por M2Caml de forma que se generen fuentes en Ocaml, estas fuentes se unen a rutinas básicas en Ocaml para convertir textos con especificaciones ManTa en ASTs (parte de esas rutinas son generadas automáticamente por los programas **`ocamlyacc`** y **`ocamllex`** con las especificaciones del apéndice B).

5.5.4 .PRUEBAS Y EVALUACIÓN

Se hicieron varios archivos de prueba para M2Caml que cubrían buena parte de los casos de especificaciones que pueden hacerse en ManTa: tipos básicos, tipos genéricos, extensiones y extensiones genéricas. Además al desarrollar las funciones básicas en Ocaml, se empleó la interfaz ocaml, para hacer algunos casos de pruebas para cada función.

El sistema obtenido tiene las siguientes restricciones:

- No pueden definirse por casos funciones cuyo dominio sea un tipo representado (en nuestro caso hay representaciones para BOOL, NAT, Char y String). En vez de esto debe usarse if/else/fi. Esto debe hacerse porque las constructoras en ManTa de estos tipos no tienen un análogo en Ocaml.

- No pueden traducirse tipos mutuamente recursivos. Esto ocurre porque cada tipo de ManTa es traducido a un módulo en Ocaml y en Ocaml no puede haber módulos mutuamente recursivos
- Dados dos ADT T1 y T2, si T1 tiene selectoras con dominio o codominio T2, y T2 tiene selectoras con dominio o codominio T1 las fuentes generadas no podrán compilarse. La razón es la misma de los tipos mutuamente recursivos.

Como se puede verificar (sección 3.6) la especificación de la traducción de ManTa a Ocaml tiene tipos mutuamente recursivos, de forma que el código generado no puede ser compilado directamente y debe ser modificado manualmente para poder obtener la herramienta ocamlgen.

5.6 MANREW - MOTOR DE REESCRITURA

Esta herramienta debe hacerse antes del chequeador de semántica estática, porque en ese chequeo se hace reescritura. Sin embargo el motor de reescritura depende de funciones para determinar tipo.

5.6.1. DISEÑO DE MÓDULOS Y ESPECIFICACIÓN

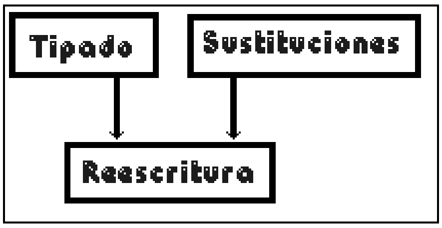


Figura 3. Módulos del motor de reescritura

Módulo de Tipado: TADs para determinar si una expresión está bien tipada y en tal caso su tipo
Módulo de Sustituciones: TADs para administrar sustituciones
Módulo de Reescritura: TADs para controlar la reescritura

De estos módulos el de tipado se podrá reutilizar al hacer el chequeador de sintaxis estática

5.6.1.1. Módulo de Tipado

Si una expresión está bien tipada y se cuenta con una tabla donde este el tipo de las variables de la expresión y una tabla de símbolos, determinar el tipo de la expresión es una labor trivial :

```
typeOfExpr(errorExpr,vt,symt)=""
typeOfExpr(varExpr(s),vt,symt)=typeOfLVarType(vt,s)
typeOfExpr(genExpr(s,a),vt,symt)=codomainFunction(s,symt)
typeOfExpr(selExpr(s,a),vt,symt)=codomainFunction(s,symt)
typeOfExpr(eqExpr(e1,e2),vt,symt)="BOOL"
typeOfExpr(ifExpr(e1,e2,e3),vt,symt)=
  if EQ_String(typeOfExpr(e2,vt,symt),"") then
    typeOfExpr(e3,vt,symt)
  else
    typeOfExpr(e2,vt,symt)
```

```
fi
```

Como puede notarse el tipo de una expresión bien tipada es una cadena con el nombre de un TAD o la cadena vacía "", si no es posible determinar el tipo de la expresión (esto ocurre por ejemplo con la expresión **`if x then y else z fi`**)

El módulo de tipado consta de los siguientes TADs:

`ADT VarType[]` `INITIALS` `  varType:String*String->VarType` `CONSTRUCTORS` `SELECTORS`	Mantiene información del tipo de una variable
`ADT LVarType[]=List[VarType[]]`	Mantiene información de los tipos de varias variables y varias funciones auxiliares para manejar estas listas
`ADT Typing[]` `INITIALS` `CONSTRUCTORS` `SELECTORS` `typeOfExpr: Expr * LVarType * LAbsType` `  -> String` `hasTypeExpr: Expr * LAbsType -> BOOL` `lVarTypeOfExpr: Expr * LAbsType -> LVarType`	Las funciones más importantes de tipado son: hasTypeExpr que determina si una expresión tiene tipo, typeOfExpr que retorna el tipo de una expresión bien tipada y lVarTypeOfExpr que retorna la tabla de los tipos de las variables de una expresión

5.6.1.2. Módulo de Sustituciones

La definición formal de una sustitución puede consultarse en el capítulo 3.4. En ManTa pueden representarse con el TAD LSubst presentado:

```
ADT Subst []
INITIALS
  subst: String * Expr -> Subst
CONSTRUCTORS
SELECTORS
applySubstExpr: Subst(i)*Expr(i) -> Expr
applySubstLExpr: Subst(i)*LExpr(i) -> LExpr
EQ_Subst: Subst(i)*Subst(i) -> Expr ADT s

ADT LSubst[]=List[Subst[]]
applyLSubstExpr: Lsubst(i) * Expr -> Expr
```

La función applyLSubstExpr aplica una sustitución a una expresión.
Además se tienen las funciones de patternMatching:

```
existMatchExpr: Expr(i) * Expr(i) -> BOOL
matchExpr       : Expr(i) * Expr(i) -> LSubst
```

La primer determina si hay *match* entre un par de expresiones y la segunda retorna el *match* entre un par de expresiones (cuando lo hay).

5.6.1.2. Módulo de Reescritura

La reescritura en ManTa se describe en la sección 3.4, se ha especificado en una extensión cuyas principales funciones se presentan a continuación:

```
ADT Rewriting[]
INITIALS
CONSTRUCTORS
SELECTORS
/* Determina si es posible hacer un paso de reescritura */
canRewStep: Expr * LVarType * LAbsType -> Expr

/* Realiza un paso de reescritura */
rewStep: Expr * LVarType * LAbsType -> Expr

/* Determina si es posible hacer un paso de reescritura en un nivel
dado del arbol
     (recorriendo tal nivel de izquierda a derecha) */
canRewNthLevelExpr: Expr * LVarType * LAbsType *NAT -> BOOL

/* Realiza un paso de reescritura en el n-esimo nivel del árbol (a
la expresión más
      a la izquierda que pueda reescribirse) */
rewNthLevelExpr: Expr * LVarType * LAbsType *NAT -> Expr

/* Reescribe una expresión empleando una tabla de símbolos */
rewExpr: Expr * LAbsType -> Expr
```

5.6.2. Implementación

Se ha generado código en Ocaml empleando ocamlgen, a partir de las especificaciones de los TADs. Este código se ha unido al código antes generado con ASTs de ManTa y a las funciones básicas en Caml desarrolladas para ocamlgen.

Tales funciones básicas administran la tabla de símbolos y pueden cargar el AST de un TAD en memoria.

El Pretty Printer para ManTa fue tomado de un ejemplo desarrollado y distribuido con la versión 3.1b4 (ver sección 4.4)

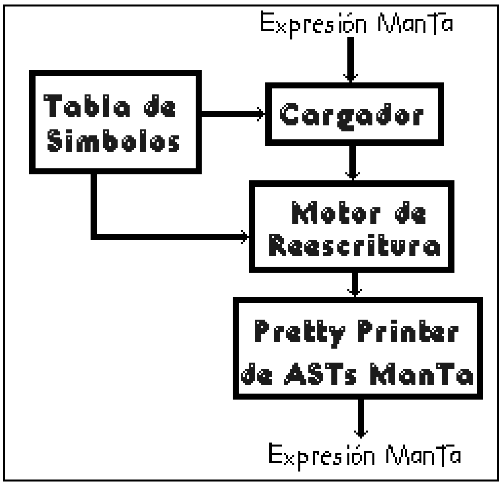


Figura 4. Implementación de manrew

6. CONCLUSIONES

- El proceso de desarrollo de ManTa ha tenido muchos errores (comenzando porque no había antes de 1997). Esperamos haber aprendido de nuestros errores y dejar testimonio para que otros no incurran en ellos. En este documento hemos presentando las soluciones que se han dado en ManTa, de forma que puedan servir para continuar su desarrollo y para hacer mejor el desarrollo de otros proyectos.

- En este documento se presentan, por primera vez, las bases teóricas de ManTa, de forma completa y consistente. Así mismo, se proponen extensiones al sistema básico que podrán emplearse para continuar el desarrollo de ManTa. De los desarrollos teóricos logrados vale la pena destacar:
    - El símbolo ERROR era el mayor problema en las bases teóricas de ManTa, creemos que finalmente le hemos dado un sentido y tratamiento satisfactorio.
    - Aunque otros enfoques para especificación de TADs sugieren el uso del álgebra inicial (por ejemplo [Goguen]), hemos optado por trabajar en otro modelo dada la dificultad para manejar funciones parciales en el álgebra inicial. También a diferencia de otros enfoques (e.g. [Kühler98]), exigimos al usuario definir explicitamente los casos en los que no está definida una función.

- Se ha logrado un bootstrap de ManTa, lo que demuestra las posibilidades del sistema y su utilidad práctica. Así mismo esto evidencia que es posible desarrollar software formalmente en nuestro medio.

- Falta desarrollar teoría e implementaciones con extensiones como: sobrecarga de nombres, genericidad (incluyendo teorías como parámetros de un TAD), más y mejores métodos de prueba y un probador interactivo. (La actual implementación sólo logra probar 6.9% de las demostraciones del Benchmarck NQTHM-92).

- Las fuentes legadas y el tiempo requerido para el desarrollo de la versión 3.1 evidencian un problema en la forma de enseñar a programar en nuestro medio. Se implementa pensando que todo funcionará. Por el contrario, el enfoque debería ser, todo debe chequearse porque cualquier componente puede fallar. Es pues, responsabilidad de la universidad y sus profesores brindar herramientas para desarrollar software de calidad. Una buena alternativa son los métodos formales y ,porque no, el sistema ManTa.

- El trabajo empezado en 1985 con la primera versión de ManTa, aún puede continuarse de varias formas:
    - Deben mejorarse las versiones 3.1 a 3.3, especialmente con relación a la interfaz
    - Debe desarrollarse la versión 3.4, esto es, un *bootstrap* en Java. El trabajo realizado para desarrollar la versión 3.3 facilitará enormement esta labor.
    - Deben desarrollarse bien las bases teóricas de las extensiones presentadas en el capítulo 3.7: tipos paramétricos con otros tipos o funciones como parámetros, mejores conjeturas al éstilo de cláusulas de Horn. Una vez desarrolladas pueden implementarse en la versión 4.0

- Por último, concluimos que la investigación en métodos formales es posible en nuestro medio y con nuestros recursos, que puede dar buenos resultados con dedicación y disciplina, y que debe ser apoyada. También creemos que ManTa puede servir en el aula como material pedagogico en diversos cursos y que vale la pena darle una oportunidad.

BIBLIOGRAFÍA

[Aponte95] APONTE, Hernán. MANTA 3.0: Un sistema para la implantación certificada y automática de tipos abstractos de datos. Tesis de Magister en Ingeniería de Sistemas y Computación. Universidad de los Andes. 1995.

[Barwise77] BARWISE, Jon. Handbook of mathematical logic. North-Holland Publishing. 1977

[Bertoni] BERTONI, A. MIGLIOLI P, Towards a Theory of Abstract Datya Types: a discussion of problems and tools. International Symposium on Programming. LNCS No. 83. Springer-Verlag.

[Cardoso93] CARDOSO, Rodrigo. Verificación y Desarrollo de Programas. Ediciones Uniandes. 1ra edición. 1993.

[Cod94] Código sin Errores. Microsoft Press. 1994

[Favre98] FAVRÉ, Jean Marie. ZML, ZLN. Notas del curso "Reingeniería de Software".

[Gries96] GRIES, David. „The Need for Education in Useful Formal Logic“. *Computer. IEEE Computer Society*. Abr/1996. pp 29.

[Goguen78] GOGUEN, J.A. THATCHER J.W. WAGNER E.G. An initial approach to the specification, correctness, and implementation of abstract data types. 1978

[Huet82] HUET, Gérard. HULLOT, Jean-Marie. Proofs by induction in equational theories with constructors. Journal of computer and system science. 1982. Pg. 239-266.

[Jameson94] JAMESON, Kevin. Multi-platform Code Managment. O’Relly & Associates. 1994

[Kühler98] KÜHLER, Ulrich. A Tactic-Based Inductive Theorem Prover for Data Types with Partial Operations. Manuscrito preliminar de Doctorado. Universidad Kaiserslautern. 1998

[Leroy98] LEROY, Xavier. The Objective Caml system, documentation and user's guide. Release 2.00. 1998

[Leroy96] LEROY, Xavier. The Caml Light system Release 0.72. Documentation and user's manual. 1996

[Manna74] MANNA, Zohar. Mathematical Theory of Computation. McGraw Hill Company. 1974

[Meyer90] MEYER, Bertrand. Introduction to the Theory of Programming Languages. Prentice Hall International Series in Computer Science. 1990

[Muñoz92] MUÑOZ, Cesar. ManTa: Un sistema para la implantación certificada y automática de Tipos Abstractos de Datos. Tesis de Magister. Departamento de Sistemas y Computación. Universidad de los Andes. 1992.

[O'Donell77] O'DONELL, Michael J. Lecture Notes in Computer Science: Computing in Systems Described by Equations. Springer-Verlag. 1977

[O'Donell85] O'DONELL, Michael J. Equational Logic as a Programming Language. The MIT Press. 1985

[Ruiz90] RUIZ, Carolina. ManTa 2. Tesis de Magister. Departamento de Sistemas y Computación. Universidad de los Andes. 1990

[Támara97] TÁMARA, Vladimir. ManTa: Manejador de Tipos Abstractos de Datos. Tesis de Pregrado. Departamento de Sistemas y Computación. Universidad de los Andes. 1997.

[Toro91] TORO, Victor Manuel. Organización y documentación de programas grandes en C (ANSI), Memo de investigación 52, Circulación privada Centro de Documentación de la Facultad de Ingeniería de la Universidad de los Andes, 1991.

[Toyama88] TOYAMA 88. Commutativity of term rewriting systems en Programming of future generation Computers. North Holland. 1988

[Whitten95] WHITTEN, Neal. Managing Software Development Projects : formula for success. 2da. Edición. Editorial John Wiley & Sons, Inc. 1995.

[Z95] Z Notation. Members of Z standard Panel. 1995

ANEXO A. EJEMPLO DE PROGRAMACIÓN EN C

La siguiente porción de código ejemplifica varias de las técnicas de programación en C, descritas en el capítulo 2.2. Note los comentarios con sintaxis, las aserciones para verificar precondiciones, el manejo de error y el manejo de mensajes.

```
/**
 * Rewriting module
 * This module rewrites an expression using an eventually outermost strategy
 * The rewriting rules are:
 *   - Axioms given by the user (left to right)
 *   - if TRUE then E1 else E2 fi -> E1
 *   - if FALSE then E1 else E2 fi -> E2
 *   - Si E1:T o E2:T, EQUAL(E1,E2) -> EQ_T(E1,E2)
 *
 * @author: Vladimir Tamara
 * @version: 2.0
 **/

#include <stdio.h>
...

/**
 * Creates an expression from a string
 * @author Vladimir Tamara
 **/
int                                         /* 0 iff everything is fine */
expressionsOfString(char *s,                /* String to parse */
            nodo_argum **e,                 /* Resulting expressions */
            char *merr)                     /* Errors buffer */
{
  FILE *fp;
  cad50 arcnam;
  texpr mens_error;

  assert(s!=NULL);
  assert(e!=NULL);

  if (*e!=NULL)
    { elimNodoArgumento(e); }

  strcpy(arcnam,"normaliz");
  fp=fopen(arcnam,"wt");
  if (fp==NULL)
    {
      sprintf(mens_error,mensajes[196],arcnam);
      strcat(mens_error,"\n");
      strcat(merr,mens_error);
      return -1;
    }
  fprintf(fp,"%s",s);
  fclose(fp);

  fp=fopen(arcnam,"rt");
  if (fp==NULL)
    {
      sprintf(mens_error,mensajes[196],arcnam);
      strcat(mens_error,"\n");
      strcat(merr,mens_error);
```

```
      return -1;
    }
  *e = analice_lista_exp (fp, mens_error);
  if (*e==NULL)
    {
      strcat(merr,mens_error);
      strcat(merr,"\n");
      return -1;
    }
  fclose(fp);

  unlink("normaliz");
  return 0;
} /* expressionsOfString */



/**
 * Changes the subexpression expr and puts the expression newexp.
 * @author Vladimir Tamara
 **/
void                          /* This function can't fail since it only frees  memory */
changeSubexpressionByExpression(
  nodo_expresion **entexpr,  /* Entire expression */
  nodo_expresion **expr,     /* Subexpression to replace */
  nodo_expresion *newexp)    /* New expression to put */
{
  assert(entexpr!=NULL);
  assert(expr!=NULL);

  if (*expr==*entexpr)
    {
      elimNodoExpresion(expr);
      *expr=*entexpr=newexp;
    }
  else
    {
      elimNodoExpresion(expr);
      *expr=newexp;
    }
} /* changeSubexpressionByExpression */
```

ANEXO B. APARTES DE OCAMLGEN

En este anexo se presentan algunas especificaciones representativas del programa de traducción de ManTa a Ocaml. Puede encontrarse más información en la página WEB de ManTa

http://wwwisc.uniandes.edu.co/~manta

B.1. AST DE MANTA

Cada elemento del AST de ManTa es un TAD, a continuación se presenta el TAD AbsType que corresponde a un TAD en ManTa.

```
ADT AbsType [  ]

INITIALS
  absType    : String * LParameter * LGenerator * LSelector -> AbsType
CONSTRUCTORS
SELECTORS
  nameAbsType    : AbsType(i)                              -> String
  parametersAbsType    : AbsType(i)                        ->   LParameter
  generatorsAbsType    : AbsType(i)                        -> LGenerator
  selectorsAbsType    : AbsType(i)                         -> LSelector
  EQ_AbsType(c): AbsType(i) * AbsType(i)                   -> BOOL
AXIOMS
  nameAbsType(absType(S1,L3,L4,L5)) = S1
  parametersAbsType(absType(S1,L3,L4,L5)) = L3
  generatorsAbsType(absType(S1,L3,L4,L5)) = L4
  selectorsAbsType(absType(S1,L3,L4,L5)) = L5
  EQ_AbsType(absType(S1,L3,L4,L5),absType(S6,L8,L9,L10)) = EQ_String(S1,S6)
```

B.2 AST Y PRETTY PRINTER DE OCAML

El ejemplo anterior y este ejemplifican lo fácil que es pasar de una sintaxis concreta a estructuras de datos que representen programas. Este ejemplo también ejemplifica el pretty printer (que depende bastante del preprocesador)

```
ADT ImplPhraseCaml [  ]

INITIALS
  exprIPC     : ExprCaml                          -> ImplPhraseCaml
  valueDefIPC : ValueDefCaml                      -> ImplPhraseCaml
  typeDefIPC  : TypeDefCaml                       -> ImplPhraseCaml
  exceptionDefIPC: ExceptionDefCaml                  -> ImplPhraseCaml
  directiveIPC: DirectiveCaml                     -> ImplPhraseCaml

CONSTRUCTORS

SELECTORS
  StringOfImplPhraseCaml(c): ImplPhraseCaml(i)  -> String
  EQ_ImplPhraseCaml(c): ImplPhraseCaml(i) * ImplPhraseCaml(i)  -> BOOL

AXIOMS

  StringOfImplPhraseCaml(exprIPC(E1))=StringOfExprCaml(E1)
  StringOfImplPhraseCaml(valueDefIPC(V1))=StringOfValueDefCaml(V1)
  StringOfImplPhraseCaml(typeDefIPC(T1))=StringOfTypeDefCaml(T1)
  StringOfImplPhraseCaml(exceptionDefIPC(E1))=StringOfExceptionDefCaml(E1)
  StringOfImplPhraseCaml(directiveIPC(D1))=StringOfDirectiveCaml(D1)
```

B.4. REPRESENTACIONES

El mecanismo de representaciones permite generar código eficiente, pues para algunos TADs emplea características propias del lenguaje destino. En este ejemplo se presenta la representación del tipo **BOOL** de ManTa en el tipo **bool** de OCaml.

```
ADT BOOLExprCaml [  ]
/*  BOOLExprCaml : BOOL -> ExprCaml */
INITIALS
CONSTRUCTORS
SELECTORS
  BOOLExprCaml_TRUE    :                                  -> ExprCaml
  BOOLExprCaml_FALSE    :                                  -> ExprCaml
  BOOLExprCaml_AND    : ExprCaml * ExprCaml              -> ExprCaml
  BOOLExprCaml_OR    : ExprCaml * ExprCaml              -> ExprCaml
  BOOLExprCaml_NOT    : ExprCaml                         -> ExprCaml
  BOOLExprCaml_IMP    : ExprCaml * ExprCaml              -> ExprCaml
  BOOLExprCaml_EQ_BOOL    : ExprCaml * ExprCaml              -> ExprCaml

  isBOOL_TRUE:String->BOOL
  isBOOL_FALSE:String->BOOL
  isBOOL_AND:String->BOOL
```

```
  isBOOL_OR:String->BOOL
  isBOOL_NOT:String->BOOL
  isBOOL_IMP:String->BOOL
  isBOOL_EQ_BOOL:String->BOOL

AXIOMS
  BOOLExprCaml_TRUE=constantEC(consCC("true"))
  BOOLExprCaml_FALSE=constantEC(consCC("false"))
  BOOLExprCaml_AND(E1,E2)=infixEC(E1,"&&",E2)
  BOOLExprCaml_OR(E1,E2)=infixEC(E1,"||",E2)
  BOOLExprCaml_NOT(E1)=funApp(valPathEC("not "),E1)
  BOOLExprCaml_IMP(E1,E2)=BOOLExprCaml_OR(BOOLExprCaml_NOT(E1),E2)
  BOOLExprCaml_EQ_BOOL(E1,E2)=infixEC(E1,"=",E2)

  isBOOL_TRUE(s)=EQ_String(s,"TRUE")
  isBOOL_FALSE(s)=EQ_String(s,"FALSE")
  isBOOL_AND(s)=EQ_String(s,"AND")
  isBOOL_OR(s)=EQ_String(s,"OR")
  isBOOL_NOT(s)=EQ_String(s,"NOT")
  isBOOL_IMP(s)=EQ_String(s,"IMP")
  isBOOL_EQ_BOOL(s)=EQ_String(s,"EQ_BOOL")
```

B.5. RUTINAS DE TRADUCCIÓN

Las funciones de tracción están agrupadas en una sola extensión que se presenta a continuación:

```
ADT M2Caml [ ]

INITIALS
CONSTRUCTORS
SELECTORS

/* Generates a Caml Implementation from
   a ManTa ADT */
AbsType_ImplCaml:
  AbsType(i)*           /* ADT to translate */
  LString               /* List of modules to open */
  -> ImplCaml

/* Generates a Caml Interface from a ManTa ADT */
AbsType_InterCaml:
  AbsType(i)*           /* ADT to translate */
  -> InterCaml


/* Generates open directives with the Strings received */
LString_LOpen:
  LString (i)
  -> LImplPhraseCaml


/* This function generate the type directive */
AbsType_TypeDefCaml:
  String*               /* Name of ADT */
  LString*              /* ADT's parameters */
  LGenerator            /* ADT's generators */
  ->TypeDefCaml
```

```
/* This function translates the selectors to functions */
LSelector_LvalueDef:LSelector(i)->LImplPhraseCaml


/* AUXILIAR FUNCTIONS FOR TYPE DEFINITION */

/* Translates a generator into a constructor of a variant type */
Generator_ConstrDeclCaml:
  Generator*
  String*     /* Name of translated ADT */
  LString     /* List of parameters */
  ->ConstrDeclCaml
LGenerator_LConstrDeclCaml:
  LGenerator*
  String*     /* Name of translated ADT */
  LString     /* List of parameters */
  ->LConstrDeclCaml

/* Translates a ManTa Domain to a list of type expression in Ocaml
   (it will be used as a cartesian product) */
LDomain_LTypeExprCaml:
  LDomain(i)*
  String*     /* Name of translated ADT */
  LString     /* Parameters of ADT */
  ->LTypeExprCaml

/* Converts a list of Parameters to a list of strings to
   work easier */
LParameter_LString :
  LParameter(i)
  ->LString

/* Convers a list of strings to a list of parameters in Caml */
LString_LTypeExprCaml:
  LString(i)
  ->LTypeExprCaml

/* AUXILIAR FUNCTIONS FOR FUNCTIONS DEFINITION */

/* Converts an axiom into a pattern matching case */
Axiom_CasePMCaml:
  Axiom(i)
  -> CasePMCaml
LAxiom_LCasePMCaml:
  LAxiom(i)
  -> LCasePMCaml

/* Converts an axiom's left expression into a Pattern in Ocaml */
leftExpr_PatternCaml:
  Expr(i)
  ->PatternCaml
LleftExpr_LPatternCaml:
  LExpr(i)
  ->LPatternCaml

/* Converts an axiom's right expression into an Ocaml Expression */
Expr_ExprCaml:
  Expr(i)
  ->ExprCaml
LExpr_LExprCaml:
  LExpr(i)
  ->LExprCaml
```

```
/* Translate a selector to a let Binding */
Selector_LetBindingCaml:
  Selector(i)
  -> LetBindingCaml

/* Decides if a selector name appears in a list of expressions */
appearsSelLExpr:LExpr(i)*String->BOOL
/* Decides if a selector name appears in an expressions */
appearsSelExpr:Expr(i)*String->BOOL
/* Decides if a selector name appears in the right side of a list of axioms */
appearsSelLAxiom:LAxiom(i)*String->BOOL
/* Decides if a selector is recursive */
isRecursiveSel:Selector(i)->BOOL

failwith_StringAux:->String
Manta_StringAux:->String

/* AUXILIAR FUNCTIONS FOR INTERFACE */

/* Converts a list of strings to a list of open directives
   for an interface */
LString_LOpenSpec:LString(i)->LSpecificationCaml

/* Adds a string to a list of string without repetition */
addNoRepLString:LString(i)*String->LString

/* Converts a Domain to a list of the types without repetition
   it adds strings to the second parameter */
LDomain_LStringNoRep:LDomain(i)*LString->LString

/*  Returns a list of the types required by the domain and
    codomain of the received list of generators */
LopenGenSpec: LGenerator(i)*LString->LString

/*  Returns a list of the types required by the domain and
    codomain of the received list of selectors */
LopenSelSpec: LSelector(i)*LString->LString

/* Returns a sequence of val expression for an interface
   for the list of selectors that receive */
LvalSpec:
  LSelector(i)*          /* List of selectors to translate */
  String*                /* Name of ADT */
  LString                /* Parameters */
  ->LSpecificationCaml

AXIOMS


/* OPEN DIRECTIVES */

LString_LOpen(nil_LString)=nil_LImplPhraseCaml
LString_LOpen(cons_LString(s,t))=
  cons_LImplPhraseCaml(directiveIPC(openDC(firstUpper(s))),LString_LOpen(t))

/* TYPE DEFINITION */

LParameter_LString(nil_LParameter)=nil_LString
LParameter_LString(cons_LParameter(parameter(s),t))=cons_LString(s,LParameter_LStrin
g(t))

LString_LTypeExprCaml(nil_LString)=nil_LTypeExprCaml
```

```
LString_LTypeExprCaml(cons_LString(s,t))=
  cons_LTypeExprCaml(identTEC(firstLower(s)),LString_LTypeExprCaml(t))

LDomain_LTypeExprCaml(nil_LDomain,nomTad,param)=nil_LTypeExprCaml
LDomain_LTypeExprCaml(cons_LDomain(domain(nomDom,ind),t),nomTad,param)=
  if EQ_String(nomDom,nomTad) then
    cons_LTypeExprCaml(
      parametricTEC(LString_LTypeExprCaml(param),firstLower(repADTCaml(nomDom))),
      LDomain_LTypeExprCaml(t,nomTad,param))
  else
    if isMemberList_LString(param,nomDom) then
      cons_LTypeExprCaml(
        identTEC(firstLower(repADTCaml(nomDom))),
        LDomain_LTypeExprCaml(t,nomTad,param))
    else
      cons_LTypeExprCaml(
        parametricTEC(nil_LTypeExprCaml,firstLower(repADTCaml(nomDom))),
        LDomain_LTypeExprCaml(t,nomTad,param))
    fi
  fi

LGenerator_LConstrDeclCaml(nil_LGenerator,nom,param)=nil_LConstrDeclCaml
LGenerator_LConstrDeclCaml(cons_LGenerator(g,t),nom,param)=
  cons_LConstrDeclCaml(
    Generator_ConstrDeclCaml(g,nom,param),
    LGenerator_LConstrDeclCaml(t,nom,param))

Generator_ConstrDeclCaml(generator(nomGen,ldom),nomTad,param)=
  if isEmptyList_LDomain(ldom) then
    constantCDC(firstUpper(nomGen))
  else

functionCDC(firstUpper(nomGen),productTEC(LDomain_LTypeExprCaml(ldom,nomTad,param))
)
  fi

AbsType_TypeDefCaml(S1,L2,L3)=
typeCaml(
  cons_LTypeBodyCaml(
    tbCaml(allFirstLower(L2),firstLower(S1),nil_LTypeExprCaml,
      cons_LTypeRepCaml(
        variantTRC(LGenerator_LConstrDeclCaml(L3,S1,L2)),
        nil_LTypeRepCaml),
      nil_LConstraintTypeCaml),
    nil_LTypeBodyCaml))

/* FUNCTIONS DEFINITION */

leftExpr_PatternCaml(errorExpr)=ERROR /* There can't be error at left side */
leftExpr_PatternCaml(varExpr(S1))=valueNamePC(firstLower(S1))
leftExpr_PatternCaml(genExpr(S1,L2))=constructorPC(firstUpper(S1),productPC(LleftExp
r_LPatternCaml(L2)))
leftExpr_PatternCaml(selExpr(S1,L2))=ERROR /* There can't be error at left side */
leftExpr_PatternCaml(eqExpr(E1,E2))=ERROR /* There can't be error at left side */
leftExpr_PatternCaml(ifExpr(E1,E2,E3))=ERROR /* There can't be error at left side */

LleftExpr_LPatternCaml(nil_LExpr)=nil_LPatternCaml
LleftExpr_LPatternCaml(cons_LExpr(e,t))=cons_LPatternCaml(leftExpr_PatternCaml(e),Ll
eftExpr_LPatternCaml(t))

failwith_StringAux="failwith"
Manta_StringAux="Manta"
```

```
/* Here we have to add representations in generators and selectors */
Expr_ExprCaml(errorExpr)=funApp(valPathEC(failwith_StringAux),constantEC(stringCC("E
RROR")))
Expr_ExprCaml(varExpr(S1))=valPathEC(firstLower(S1))
Expr_ExprCaml(genExpr(S1,L2))=
    repGenCaml(S1,L2)
Expr_ExprCaml(selExpr(S1,L2))=
    repSelCaml(S1,LExpr_LExprCaml(L2))
Expr_ExprCaml(eqExpr(E1,E2))=infixEC(Expr_ExprCaml(E1),"=",Expr_ExprCaml(E2))
Expr_ExprCaml(ifExpr(E1,E2,E3))=ifEC(Expr_ExprCaml(E1),Expr_ExprCaml(E2),cons_LExprC
aml(Expr_ExprCaml(E3),nil_LExprCaml))

LExpr_LExprCaml(nil_LExpr)=nil_LExprCaml
LExpr_LExprCaml(cons_LExpr(e,t))=cons_LExprCaml(Expr_ExprCaml(e),LExpr_LExprCaml(t))

Axiom_CasePMCaml(axiom(errorExpr,der))=ERROR /* Axioms only for selectors */
Axiom_CasePMCaml(axiom(varExpr(S1),der))=ERROR /* Axioms only for selectors */
Axiom_CasePMCaml(axiom(genExpr(S1,L2),der))= ERROR /* Axioms only for selectors */
Axiom_CasePMCaml(axiom(selExpr(S1,L2),der))=cPMC(productPC(LleftExpr_LPatternCaml(L2
)),nil_LExprCaml,Expr_ExprCaml(der))
Axiom_CasePMCaml(axiom(eqExpr(E1,E2),der))=ERROR /* Axioms only for selectors */
Axiom_CasePMCaml(axiom(ifExpr(E1,E2,E3),der))=ERROR /* Axioms only for selectors */

LAxiom_LCasePMCaml(nil_LAxiom)=nil_LCasePMCaml
LAxiom_LCasePMCaml(cons_LAxiom(a,t))=cons_LCasePMCaml(Axiom_CasePMCaml(a),LAxiom_LCa
sePMCaml(t))

Selector_LetBindingCaml(selector(nom,dom,cod,car,lax))=
  if (isEmptyList_LDomain(dom)) then /* It's a constant */
    valLBC(firstLower(nom),nil_LPatternCaml,nil_LTypeExprCaml,
      Expr_ExprCaml(rightAxiom(head_LAxiom(lax))))
  else
    valLBC(firstLower(nom),nil_LPatternCaml,nil_LTypeExprCaml,
      functionEC(pmCaml(LAxiom_LCasePMCaml(lax))))
  fi

appearsSelLExpr(nil_LExpr,sel)=FALSE
appearsSelLExpr(cons_LExpr(h,t),sel)=if   appearsSelExpr(h,sel)   then   TRUE   else
appearsSelLExpr(t,sel) fi

appearsSelExpr(errorExpr,sel)=FALSE
appearsSelExpr(varExpr(S1),sel)=FALSE
appearsSelExpr(genExpr(S1,L2),sel)=appearsSelLExpr(L2,sel)
appearsSelExpr(selExpr(S1,L2),sel)=if    EQ_String(S1,sel)    then    TRUE    else
appearsSelLExpr(L2,sel) fi
appearsSelExpr(eqExpr(E1,E2),sel)=OR(appearsSelExpr(E1,sel),appearsSelExpr(E2,sel))
appearsSelExpr(ifExpr(E1,E2,E3),sel)=OR(OR(appearsSelExpr(E1,sel),appearsSelExpr(E2,
sel)),appearsSelExpr(E3,sel))

appearsSelLAxiom(nil_LAxiom,nom)=FALSE
appearsSelLAxiom(cons_LAxiom(axiom(izq,der),t),nom)=
  if appearsSelExpr(der,nom) then TRUE else appearsSelLAxiom(t,nom) fi

isRecursiveSel(selector(nom,dom,cod,car,lax))=
  appearsSelLAxiom(lax,nom)

LSelector_LvalueDef(nil_LSelector)=nil_LImplPhraseCaml
LSelector_LvalueDef(cons_LSelector(sel,t))=
  if isRecursiveSel(sel) then
    cons_LImplPhraseCaml(

valueDefIPC(letRecVDC(cons_LLetBindingCaml(Selector_LetBindingCaml(sel),nil_LLetBin
dingCaml)))
```

```
      ,LSelector_LvalueDef(t))
  else
    cons_LImplPhraseCaml(

valueDefIPC(letVDC(cons_LLetBindingCaml(Selector_LetBindingCaml(sel),nil_LLetBindin
gCaml)))
      ,LSelector_LvalueDef(t))
  fi

/* OCAML IMPLEMENTATION */

AbsType_ImplCaml(absType(S1,L2,L3,L4),lopen)=
  if isEmptyList_LGenerator(L3) then /* It's an extension */
    iCaml(concat_LImplPhraseCaml(

LString_LOpen(cons_LString(Manta_StringAux,cons_LString(S1,repOpenCaml(lopen,S1))))
,
      LSelector_LvalueDef(L4)))
  else
    iCaml(concat_LImplPhraseCaml(

LString_LOpen(cons_LString(Manta_StringAux,cons_LString(S1,repOpenCaml(lopen,S1))))
,
      cons_LImplPhraseCaml(
        typeDefIPC(AbsType_TypeDefCaml(S1,LParameter_LString(L2),L3)),
        LSelector_LvalueDef(L4))))
  fi



/* INTERFACE */


LString_LOpenSpec(nil_LString)=nil_LSpecificationCaml
LString_LOpenSpec(cons_LString(s,t))=
  cons_LSpecificationCaml(directiveSpec(openDC(firstUpper(s))),LString_LOpenSpec(t))

addNoRepLString(nil_LString,s)=cons_LString(s,nil_LString)
addNoRepLString(cons_LString(h,t),s)=
  if EQ_String(h,s) then
    cons_LString(h,t)
  else
    cons_LString(h,addNoRepLString(t,s))
  fi

LDomain_LStringNoRep(nil_LDomain,r)=r
LDomain_LStringNoRep(cons_LDomain(domain(S1,B2),t),r)=
  LDomain_LStringNoRep(t,addNoRepLString(r,S1))

LopenGenSpec(nil_LGenerator,r)=r
LopenGenSpec(cons_LGenerator(generator(S1,L2),t),r)=
  LopenGenSpec(t,LDomain_LStringNoRep(L2,r))

LopenSelSpec(nil_LSelector,r)=r
LopenSelSpec(cons_LSelector(selector(S1,L2,S3,B4,L5),t),r)=
  LopenSelSpec(t,addNoRepLString(LDomain_LStringNoRep(L2,r),S3))

LvalSpec(nil_LSelector,nomTad,param)=nil_LSpecificationCaml
LvalSpec(cons_LSelector(selector(nomSel,ldom,codom,B4,L5),t),nomTad,param)=
  if isEmptyList_LDomain(ldom) then /* It's a constant */
    cons_LSpecificationCaml(
      valSpec(firstLower(nomSel),
```

```
head_LTypeExprCaml(LDomain_LTypeExprCaml(cons_LDomain(domain(codom,FALSE),nil_LDoma
in),nomTad,param)))
      ,LvalSpec(t,nomTad,param))
  else
    cons_LSpecificationCaml(
      valSpec(firstLower(nomSel),
        functionTEC(
          productTEC(LDomain_LTypeExprCaml(ldom,nomTad,param))

,head_LTypeExprCaml(LDomain_LTypeExprCaml(cons_LDomain(domain(codom,FALSE),nil_LDom
ain),nomTad,param))))
      ,LvalSpec(t,nomTad,param))
  fi


AbsType_InterCaml(absType(S1,L2,L3,L4))=
  if isEmptyList_LGenerator(L3) then /* It's an extension */
    seqSpec(concat_LSpecificationCaml(
      LString_LOpenSpec(repOpenCaml(LopenSelSpec(L4,nil_LString),S1)),
      LvalSpec(L4,S1,LParameter_LString(L2))))
  else
    seqSpec(concat_LSpecificationCaml(

LString_LOpenSpec(repOpenCaml(difList_LString(LopenGenSpec(L3,LopenSelSpec(L4,nil_L
String)),LParameter_LString(L2)),S1)),
      concat_LSpecificationCaml(
        cons_LSpecificationCaml(
          typeSpec(AbsType_TypeDefCaml(S1,LParameter_LString(L2),L3))
         ,nil_LSpecificationCaml)
        ,LvalSpec(L4,S1,LParameter_LString(L2)))))
  fi
```