LABORATORIO DE CIENCIA DE LA COMPUTACION
PROYECTO ESL (Estructuras para Sistemas Lógicos)
Informe Anual : Agosto 1996 - Agosto 1997

Juan José Cabezas - Guillermo Calderón
Sylvia da Rosa - Alvaro Tasistro
Nora Szasz - Gustavo Betarte
Eduardo Giménez

Instituto de Computación
Facultad de Ingeniería
Universidad de la República
Montevideo. Uruguay

Organización de este informe

Presentamos el informe correspondiente al periodo agosto 1995 - agosto 1996 del proyecto Bid-Conicyt 043, *Estructuras para Sistemas Lógicos (ESL).*
El informe está organizado de la siguiente manera:

- La sección 1 describe los puntos que han sido sugeridos por el *Departamento de Proyectos de Investigación del Conicyt.*
- La sección 2 es una breve *Introducción* acerca del último año de ejecución del proyecto.
- La sección 3 corresponde a las *Actividades desarrolladas* en el último año del proyecto.
- La sección 4 contiene las *Conclusiones y Perspectivas Futuras* sobre el trabajo realizado.
- Los Apéndices A, B y C contienen una descripción de los siguientes puntos:
  - Apéndice A : Detalles Técnicos
  - Apéndice B : Sesión con Glich
  - Apéndice C : Implementación

Se adjuntan los siguientes reportes y artículos:

- *Typed Windows: A Functional Programming Extension for Graphic Design.* Juan José Cabezas.
- *Implementación en Haskell de un algoritmo genérico para resolución de ecuaciones lineales por métodos iterativos.* Sylvia da Rosa y Angel Caffa.

1 Descripción general

Se describen a continuación los siguientes puntos solicitados por el Depto. de Proyectos de Investigación del Conicyt:

- Datos del Proyecto
  Proyecto Bid-Conicyt 043, *Estructuras para Sistemas Lógicos (ESL).*

- Metas previstas y logradas en el semestre: Cumplimiento del cronograma original.
  Se considera que las metas previstas se han alcanzado, con alguna modificación de caracter académico, lo cual es previsible que suceda a lo largo de la ejecución del proyecto y que ha redundado en profundización de conocimientos en los temas involucrados. Se ha continuado trabajando en el perfeccionamiento del prototipo básico. Se resume el estado del prototipo de la siguiente manera:

  - Verificador de tipos : completo
  - Interfaz en Emacs : completo
  - Parser : completo
  - Editor de pruebas :
    * motor : en desarrollo (90% completo)
    * parser : en desarrollo (80% completo)
    * interfaz : a desarrollar
  - Experimentación : prueba con ejemplos de tamaño pequeño y medio
  - Extensiones a la teoría : a desarrollar

  Se destaca que además de las tareas previstas en el cronograma original, se han llevado a cabo las siguientes:

  - Implementación de una interfaz en Emacs para GLICH
  - Experimentación con interfases gráficas.
  - Experimentación con lenguajes funcionales.
  - Enseñanza de materias relativas al proyecto:
    curso electivo de Programación Funcional Avanzada
    proyectos para Taller 5 (proyectos de graduación de la carrera de Ingeniería en Computación).

- Actividades desarrolladas en el período.
  Primer semestre (agosto 1996 - febrero 1997):
  En setiembre de 1996 tiene lugar en Buenos Aires el Primer Taller de Programación Funcional, organizado por la Universidad Nacional de La Plata, con participación de Universidades de Argentina, Brasil y Uruguay, además de algunas Universidades europeas. Esta actividad se enmarca dentro de un plan de trabajo conjunto entre los grupos

académicos de la Universidad Nacional de La Plata (UNLP), la Universidad Federal de Pernambuco (UFPE) y el Instituto de Computación de la Facultad de Ingeniería de la Universidad de la República. acordado por Gabriel Baum (UNLP, La Plata - Argentina), Rosita Wachenchauzer (UBA, Buenos Aires - Argentina), Rafael Lins (UFPE, Recife - Brasil) y Sylvia da Rosa (Universidad de la República - Uruguay).
Este grupo se reúne en diciembre de 1996 en Montevideo para evaluar el resultado del Primer Taller de Programación Funcional y discutir líneas generales de trabajo para el futuro. Se deciden, entre otros, dos puntos importantes a saber:
1) el próximo evento se llamará 2da. Conferencia Latinoamericana de Programación Funcional (CLaPF97) y tendrá lugar en La Plata en el marco del Congreso Argentino CASIC en octubre de 1997.
2) se priorizarán los trabajos sobre aplicaciones de la programación funcional a diferentes áreas.
A finales del año 96 y principios del 97, el subdirector del proyecto Guillermo Calderón, viaja a Francia para participar en el "Working Group Types - Annual Meeting".
También se realizaron en Montevideo reuniones del equipo local del proyecto con los miembros en el extranjero Gustavo Betarte y Eduardo Giménez. En ellas se discutió sobre la marcha del proyecto así como también sobre el trabajo futuro del Laboratorio de Ciencia de la Computación teniendo en cuenta el retorno de varios miembros que finalizan su doctorado entre el 97 y el 98.
Encuanto a la implementación, esta etapa se caracteriza por el estudio y adiestramiento de las herramientas provistas por la nueva versión de Haskell y su utilización en la construcción del editor. Algunas partes (por ejemplo la parte de entrada y salida), debieron ser reescritas usando facilidades recientemente incorporadas al lenguaje. Debe tenerse en cuenta que Haskell es un lenguaje de programación diseñado e implementado para, entre otras cosas, desarrollar aplicaciones de gran envergadura, a la vez que mantiene la pureza funcional, lo cual lo ha convertido en un lenguaje amplio y de manejo no trivial. Extensiones, sust. expl.
Esta etapa de estudio ha resultado en la creación de una nueva materia de posgrado para la carrera de Ingeniería en Computación, así como también al desarrollo de varios proyectos de Taller 5 en Programación Funcional.

- Capacidad generada
  Actividades de grado y posgrado:
  Doctorados concluidos:
  Substitutions, record types and subtyping in type theory, with applications to the theory of programming. - Alvaro Tasistro, Göteborgs Universitet och Chalmers Tekniska Högskola, Gotemburgo, Suecia, mayo 1997.
  A theory of Specifications, Programs and Proofs. - Nora Szasz, Göteborgs Universitet och Chalmers Tekniska Högskola, Gotemburgo, Suecia, junio 1997.
  A Calculus of Infinite Constructions and its application to the verification of communicating systems. - Eduardo Giménez, Ecole Normale Supérieure de Lyon, Francia,

diciembre 1996.
Tesis de maestría concluidas:
Typed Windows: An Implementation of a Programming Language for Graphic Design. - Pablo Queirolo , supervisor: Juan José Cabezas
Nota de aprobación de la tesis: Excelente.
Proyectos de Taller 5 concluidos exitosamente:
Programación Funcional Aplicada al Cálculo Numérico - Angel Caffa, supervisora : Sylvia da Rosa.
Un lenguaje de puertas - Patricia Pena y Willi Bauml, supervisor: Juan J. Cabezas.
Un lenguaje de Voxels - Oscar Nogueira y Alvaro Ruiz, supervisor: Juan J. Cabezas.
Porlan: Un lenguaje de puertas para TyWin. - Alexandra Fernández y Ernesto Domínguez, supervisor: Juan J. Cabezas.
Un chequeador de tipos para Bamba. - Leonardo Amor y Fernando Nahum, supervisor: Juan J. Cabezas.

- Recursos humanos : Se describen en el siguiente orden: equipo del proyecto y personal contratado y según los siguientes puntos: Integrantes del equipo de trabajo, Actividades desarrolladas por cada uno, Carga horaria dedicada al proyecto e Integrantes que reciben compensaciones (complemento Bid).
Alvaro Tasistro - trabajó con el grupo local del proyecto en las extensiones a la Teoría de Tipos de Martin Löf. Contratado por 8 hs. semanales en setiembre de 1996. Se reintegra al grupo local del proyecto en julio de 1997, una vez finalizado su doctorado en Suecia.
Nora Szasz - trabajó con el grupo local del proyecto en las extensiones a la Teoría de Tipos de Martin Löf. Contratada por 8 hs. semanales en setiembre de 1996. Se reintegra al grupo local del proyecto en julio de 1997, una vez finalizado su doctorado en Suecia.
Juan José Cabezas - director del proyecto. Ha desarrollado un lenguaje funcional, llamado Bamba, algunas de cuyas características se describen en la sección 3 de este informe. Carga horaria: 36 meses - 20 hs. semanales.

  Guillermo Calderón - desarrolla tareas de supervisión, coordinación, diseño e implementación. Actúa como subdirector del proyecto. Carga horaria: 36 meses - 20 hs. semanales. Recibe complemento Bid, correspondiente a una dedicación compensada del 50 %.

  Sylvia da Rosa - desarrolla tareas de implementación y organización. Carga horaria: 36 meses - 20 hs. semanales. Recibe complemento Bid, correspondiente a una dedicación compensada del 50 %.

  Personal contratado:
  No se contrató personal en este período.
  Personal que reside temporalmente en el exterior:
  Gustavo Betarte y Eduardo Giménez - realizaron una visita a Montevideo para mantener reuniones de trabajo y discusión con el grupo local del proyecto, en diciembre de

1996.
Eduardo Giménez es co-editor junto a Christine Paulin de los proceedings de la última conferencia anual "Types for proof and programas" (TYPES'96). Estos proceedings van a aparecer en las Lecture Notes in Computer Science de Springer-Verlag.

- Recursos materiales propios y adquiridos por el Proyecto:
  Encuanto a materiales, se compraron materiales de trabajo y material bibliográfico.

- Dificultades presentadas
  Ninguna

- Fundamentación de cualquier desviación de objetivos
  No se produjo desviación de los objetivos en este período.

- Publicaciones efectuadas

  - *Typed Windows: A Functional Programming Extension for Graphic Design.* Juan José Cabezas. junio 1997 (submitido para publicación). Se adjunta copia).

  - *Implementación en Haskell de un algoritmo genérico para resolución de ecuaciones lineales por métodos iterativos* Sylvia da Rosa y Angel Caffa.
    2nd. Conferencia de Programación Funcional, La Plata , octubre 1997, se adjunta copia.

  - *A Tutorial on Recursive Types in Coq.* Eduardo Giménez.
    Notas distribuidas en el curso sobre Sistema Coq llevado a cabo en la Ecole Normale Supérieure de Lyon en marzo 1996.

  - *The Coq Proof Assistant Reference Manual – Version V6.1* Eduardo Giménez et al. Reporte técnico del INRIA, 1997.

  - *Structural Inducion in Coq* Eduardo Giménez. diciembre 1997 (submitido para publicación).
  - *Extension of Martin-Löf's type theory with record types and subtyping* Gustavo Betarte y Alvaro Tasistro.
    To appear in "25 Years of Constructive Type Theory", Oxford University Press, 1997.
  - *Dependent Record Types, Subtyping and Proof reutilization.* Gustavo Betarte.
    In the online proceedings of the Working Group TYPES workshop "Subtyping, inheritance and modular development of proofs", held at Durham, England, 1997.

- *Type Theory and Functional Programming: A Work Proposal.* Alvaro Tasistro et al.
  1er Taller de Programación Funcional, Buenos Aires 1996.
- *Formalizing the semantics of While -I.* Patricia Peratto, diciembre 1997.
  Reporte técnico del Pedeciba.

- *Informe Anual del Proyecto ESL (agosto 1996 - agosto 1997).* Sylvia da Rosa et al.
  Reporte técnico del PEDECIBA, diciembre 1997.

2 Introducción

En el documento de presentación del Proyecto ESL se plantean como objetivos del mismo, la implementación de un prototipo de editor de Pruebas, experimentación en la representación de sistemas lógicos y construcción de pruebas, e investigación en las teorías que sirven de soporte a los sistemas de edición de pruebas. Se plantea también la necesidad de consolidar y desarrollar un grupo de trabajo en temas de programación funcional, con buen manejo del lenguaje Haskell, en el cual está implementado el prototipo.
Durante tres años el equipo del proyecto ha desarrollado las tareas necesarias para alcanzar los objetivos propuestos, con las variaciones que obviamente es pertinente hacer para adecuar la actividad académica al desarrollo de la investigación científica. El proyecto se ha desarrollado en el marco del Laboratorio de Ciencia de la Computación (LCC), nucleando actividades provenientes fundamentalmente de las áreas generales siguientes:
1) Teoría de Tipos, 2) Programación Funcional, 3) Computación Gráfica. Estas áreas son especialmente dinámicas y en particular en estos últimos años se han producido avances notables tanto a nivel de la teoría como en aplicaciones prácticas. El proyecto, en tanto proyecto de investigación científica, ha proporcionado un marco ideal para el estudio y la experimentación permanente.
El equipo que ha llevado adelante el proyecto ESL está formado en parte por miembros que han estado todo el tiempo de ejecución del mismo en el país y en parte por miembros que parte del tiempo han residido en el exterior, realizando estudios de doctorado. Todos los miembros residentes en el exterior, han realizado visitas periódicas al país para trabajar en el proyecto, cuya duración ha oscilado entre uno y seis meses. En este último año, tres de ellos culminaron sus estudios de doctorado y dos han retornado al país. Otro estudiante de doctorado, termina en febrero 1998 y se espera su retorno para el primer semestre del año próximo.
Los miembros que han residido todo el tiempo en el país han realizado pasantías en el exterior.
En este informe se describen las actividades realizadas en el último año de ejecución del proyecto, resultados, conclusiones y perspectivas de trabajo futuro.

3 Actividades desarrolladas

Se describen las principales actividades del último año de ejecución del proyecto ESL, divididas en: actividades de implementación, actividades de la subárea de Computación Gráfica y actividades generales.

3.1 Actividades de implementación.

La etapa anterior dio como resultado un prototipo del editor de pruebas, sobre el cual se trabaja en esta etapa, en dos direcciones. Por un lado, en la introducción de mejoras a la implementación lo que implica la reescritura de algunas partes para utilizar las herramientas provistas por la nueva versión de Haskell, por ejemplo para la entrada y salida y para el manejo de mensajes de error. (La nueva implementación se adjunta en el Apéndice A).
Durante los meses de enero y febrero de este año se se implementó en el lenguaje Emacs-Lisp, una interface para el verificador de tipos GLIH. Esta interface, permite editar archivos de definiciones en el editor Emacs contando con un número de facilidades que permite escribir el código más facilmente, asi como una comunicación más simple con el verificador de tipos GLICH. Este enfoque surgió en base a una propuesta de diseñar un editor de pruebas a la manera de un ambiente de programación donde el usuario pueda escribir su código con mayor libertad que en los asistentes usuales.

Entre las facilidades provistas podemos mencionar:

Indentación automática Cada vez que se produce un cambio de línea, el editor automaticamente posiciona el cursor en la columna adecuada de acuerdo al contexto sintáctico del código que se está editando.

Apareo de paréntesis El editor produce un pestañeo (highlight) cada vez que se cierra un paréntesis de cualquier tipo.

Fontificación sintáctica. Las palabras y símbolos del código se escriben en *fonts* y *colores* diferentes de acuerdo con la sintaxis del lenguaje.

Terminación automática El editor completa en forma automática algunos símbolos que el usuario comenzó a escribir (esta facilidad no está aún implementada).

Manual y help en línea Consulta en línea de un manual en hipertexto, con facilidades de búsqueda por palabra clave. (esta facilidad no está aún implementada).

Edición de texto. Aquí incluimos las facilidades habituales de cualquier editor de texto con interface gráfica (cortado y pegado).

Con respecto a la comunicación con el verificador GLICH, disponemos de los siguientes comandos:

- Iniciar el verificador de tipos. Esta acción produce la ejecución del verificador de tipos en un *subshell* cuya salida aparece en una ventana inferior del editor.

- Salvar el *buffer* y mandar el contenido al verificador de tipos. Las definiciones del buffer son agregadas a las ya existentes en el ambiente.
- Mandar el contenido de un archivo cualquiera al verificador de tipos.
- Borrar todas las definiciones del ambiente.
- Cargar la definición que rodea al cursor.

Todas estas operaciones se pueden invocar, abriendo un menú y seleccionando con el *mouse* o con una combinación adecuada de teclas (por ej: crtrl-c, ctrl-u).

El diseño de esta interfaz involucró una cantidad de horas importantes en el estudio del editor Emacs y en particular su lenguaje de programación emacs-lisp.

3.2 Actividades desarrolladas en Computación Gráfica.

A continuación se describen las actividades de una subárea del proyecto, que desarrolla un sistema de Computación Gráfica.

- Experimentación con interfases gráficas
  En esta subárea del proyecto se ha concluido la implementación y experimentación de un prototipo de TyWin (Typed Windows), un sistema para diseño gráfico constructivo propuesto por Juan José Cabezas. Se adjunta el artículo "Typed Windows: A Functional Programming Extension for Graphic Design", escrito por Juan José Cabezas.

  La principal base del sistema consiste en extender el concepto clásico de ventana utilizado en computación gráfica. Una ventana clásica se define en función de un producto cartesiano de subrangos de enteros o reales y acepta tuplas de enteros o reales para ser visualizadas en el monitor. En el sistema TyWin cada ventana tiene asociado un tipo (cualquier tipo de un lenguaje de programación). El sistema define reglas de representación asociadas a estos tipos considerados. De esta forma, cada ventana del sistema de tipo T acepta valores de su tipo, los cuales serán representados utilizando la regla correspondiente al tipo T.

  El sistema toma además algunas ideas del Üniversalismo Constructivo"del pintor uruguayo Joaquín Torres García y de la "Teoría de Tipos"del matemático sueco Per Martin-Löf.

  Cuatro módulos del sistema fueros implementados: un intérprete del lenguaje para diseño gráfico en 2D , un sistema para definir "íconos"que se utilizarán en el lenguaje, el lenguaje funcional Bamba y un chequeador de tipos.

  El resultado es un sistema para diseño gráfico en 2D, con varias características particulares. Algunas de estas características son introducidas en un conjunto de ejemplos que incluyen pinturas de Joaquín Torres García.

  Los dos primeros módulos han sido realizados por Pablo Queirolo durante su tesis de maestría que culminó exitosamente.

El lenguanje Bamba ha sido desarrollado por Juan J. Cabezas en tanto el chequeador de tipos fue implementado por Leonardo Amor y Fernando Nahum como parte de su proyecto de grado para el título de Ingeniero en Computación.

En la actualidad el trabajo de TyWin se concentra en la integración de los cuatro módulos y la experimentación con el sistema completo.

- Experimentación con lenguajes funcionales.

  En su segundo año de desarrollo, el lenguaje de programación Bamba, construido en el marco del proyecto ESL, se ha transformado en un banco de experimentación sumamente productivo.

  Bamba ha crecido o modificado en varios aspectos: nuevos tipos, administador de memoria, inferencia de tipos. Su velocidad se ha incrementado sensiblemente con respecto a los primeros prototipos. El siguiente cuadro comparativo con versiones anteriores de Bamba o con otros lenguajes funcionales da una idea de su rapida evolución. Se evalúa (length [1..1000000]).
  La versión actual de Bamba es 0.58 (Octubre 1996).

```
                                          tiempo
   Lugar  Lenguaje Versi\'on   WS          min  seg
   ------------------------------------------------
    1      lmli    0.999.5   Sparc4       00   08
    2      hbi     0.999.5   Sparc4       00   24
    3      hugs    1.0       Sparc4       02   35
 -> 4      bamba   0.58      Sparc4       03   25
    5(*)   gofer   2.39a     Sparc4       05   30
    6      bamba   0.56      Sparc4       06   19
    7      bamba   0.5       Sparc4       08   35
    8      bamba   0.4       Sparc4       12   15
   ------------------------------------------------
   (*) gracias a Alberto Pardo y su Sparc 4 en Alemania.
```

  Estos muy buenos resultados han permitido comenzar a utilizar el prototipo como parte del proyecto de ventanas tipadas con resultados hasta el momento exitosos. Bamba está siendo desarrollado e implementado por Juan José Cabezas.

3.3 Actividades generales.

En diciembre de 1996 se realiza el Encuentro Anual del LCC, donde se evalúa la etapa pasada y se planifica la futura. Surge entonces la necesidad de redefinir algunos objetivos específicos, y por lo tanto algunos de los puntos descritos en las fases 4 y 5 del plan de trabajo original. Esto se debe en parte al gran impulso que ha tenido la programación funcional a raíz de la difusión del lenguaje Haskell. En nuestra región ese hecho se ve reflejado en el crecimiento

de los grupos de trabajo y por tanto en la necesidad de atender a múltiples actividades relacionadas con el proyecto. Es así que a raíz del éxito que significó la realización del Primer Taller de Programación Funcional en Buenos Aires en setiembre de 1996, se intensifican las tareas tendientes a profundizar este tipo de actividad conjunta con la Universidad Nacional de La Plata y con La Universidad Federal de Pernambuco.
En diciembre de 1996 se realiza una reunión del Comité organizador del Taller, en el que participan miembros del proyecto, en la que se prepara el próximo evento, que pasa a llamarse Conferencia Latinoamericana de Programación Funcional (CLaPF). Se decide que la próxima será en octubre 1997 en La Plata, dentro del marco del Congreso Argentino de Ciencia de la Computación. Para la misma se decide invitar al Profesor John Hughes, de la Universidad de Gotemburgo y uno de los miembros del Comité creador del lenguaje Haskell. Se decide también solicitar al Profesor John Hughes que dicte un curso en Montevideo una vez finalizada la Conferencia, sobre temas avanzados de Programación Funcional, para estudiantes y profesores de la región.
Al mismo tiempo, en el ámbito local, se decide intensificar las actividades tendientes a integrar estudiantes a proyectos del LCC. De esta manera, se plantean para el año 1997 varios proyectos de grado sobre temas afines, asi como también se planifica y elabora un segundo curso de Programación Funcional, que incluya las herramientas avanzadas del lenguaje Haskell.
En setiembre de 1997 se realiza en Montevideo un seminario sobre programación funcional y concurrencia, a cargo del profesor Ricardo Peña de la Universidad Complutense de Madrid (España). R. Peña trabaja en colaboración con un grupo de investigadores de la Universidad de Manheim (Alemania) en el diseño y la implementación de un lenguaje funcional concurrente, llamado Eden.
A principio de octubre 1997 se realiza en La Plata (Argentina), la 2da. Conferencia de Programación Funcional, con abundante concurrencia de docentes, investigadores y estudiantes de la región, y con participación de los investigadores extranjeros Ricardo Peña de la Universidad Complutense de Madrid (España) y John Hughes de la Universidad de Gotemburgo (Suecia), así como varios participantes de universidades europeas.
Una vez finalizada la Conferencia en La Plata, se realiza en Montevideo un curso sobre Evaluación Parcial a cargo del Profesor John Hughes, con participación de estudiantes de Argentina, Brasil y Montevideo.
Se considera que las actividades realizadas en conjunto por los distintos grupos de investigación de la región, han resultado un éxito hasta el momento y se espera su intensificación en el futuro. La próxima Conferencia Latinoamericana de Programación Funcional, queda fijada para marzo de 1999 en Recife (Brasil) y la siguiente en Montevideo (Uruguay) en fecha a determinar. Se buscará la continuación y profundización de actividades de coperación entre los grupos académicos de la región, así como también la integración de grupos de universidades de otros países de América Latina.

4 Conclusiones

Se presentan algunas conclusiones que resultan de la evaluación de la ejecución del proyecto a lo largo de estos últimos tres años, así como también desde el punto de vista de la perspectiva del trabajo futuro del LCC.
La evaluación se realiza teniendo en cuenta los siguientes puntos:
a) el cumplimiento del plan original,
b) el aporte académico-científico,
c) el aporte al mejoramiento de la docencia,
d) el aporte a la consolidación de las relaciones con otras universidades de la región

- Cumplimiento del plan original Llamamos plan original del proyecto al conjunto de acciones que se realizaron dentro del plazo de ejecución del proyecto con financiación del mismo, tal como estaba previsto en el documento de presentación del proyecto. Se describen a continuación:

  - Adquisición de materiales: el proyecto permitió al LCC comprar el equipamiento mínimo adecuado para el trabajo científico: workstations Sparc4 e impresora laser. También se adquirió material bibliográfico, muebles, material necesario para la instalación eléctrica, toner y papel para la impresora, disquetes.
  - Pasantías en el exterior y visitas de profesores extranjeros: se realizaron con financiación a cargo del proyecto, varias pasantías de miembros del equipo local del proyecto al exterior, así como visitas de los miembros del proyecto que hacían sus estudios de doctorado en el extranjero, ambas para trabajar conjuntamente en el prototipo, ya sea en la elaboracón del diseño como en la implementación. También se financió la visita de profesores extranjeros.
  - Participación en eventos científicos: los miembros del proyecto participaron en varios eventos científicos, (congresos, conferencias, workshops), tanto en países europeos como en países latinoamericanos, (con financiación a cargo del proyecto), lo que permitió establecer contactos con grupos académicos de otros países, así como también dar a conocer al LCC.
  - obtención de los objetivos específicos: los objetivos específicos que se señalan en el documento de presentación del proyecto son:
    1. Implementación de un prototipo de editor de Pruebas para la Teoría de Tipos
    2. Experimentación en la representación de sistemas lógicos y construcción de pruebas
    3. Investigación en las teorías que sirven de soporte a los sistemas de edición de Pruebas

    Estos objetivos se concretaron con algunas desviaciones menores, surgidas del propio desarrollo del prototipo.

- aporte académico científico: se considera que el aporte académico científico que significó la ejecución del proyecto, tanto para el InCo como para el LCC, ha sido muy importante. La metodología de trabajo adoptada permitió mantener un estrecho contacto entre el equipo local del proyecto y los miembros residentes en el exterior, lo que está posibilitando hoy día la reinserción de estos últimos, una vez que finalizan sus estudios de doctorado, en forma totalmente armónica y natural. Se realizaron varias tesis de maestría así como proyectos de grado (Taller V) sobre temas relacionados con el proyecto. Se efectuaron varias publicaciones y participaciones en eventos científicos importantes.

- aporte al mejoramiento de la enseñanza: las actividades de investigación tienen una incidencia inmediata y directa sobre la calidad de la enseñanza impartida. Se han elaborado programas para cursos nuevos, de grado y de posgrado, sobre temas de avanzada en el área de investigación del LCC. También se llevan adelante talleres V (proyectos de grado) y tesis de maestrías en temas afines al proyecto y se prevee que en el futuro se continúe con esta tarea.

- relaciones con grupos académicos de la región: se ha avanzado notablemente en la realización de actividades en conjunto con las Universidades de La Plata (Argentina) y Recife (Brasil), siendo la más importante el establecimiento de la Conferencia Latinoamericana de Programación Funcional. En el futuro se prevee la concreción de otros eventos tendientes a profundizar la colaboración, así como a fomentar el acercamiento con otros países de América Latina.

4.1 El futuro

Creemos que la ejecución del proyecto permitió crear una base mínima indispensable para continuar con el trabajo académico y científico.
Desde el punto de vista del trabajo realizado y sus perspectivas, han quedado planteados para ser resueltos varios problemas interesantes, uno de ellos y el más importante, concluir la construcción del prototipo del editor de pruebas y usarlo en forma exhaustiva. Esto incluye proveerlo de una interfase gráfica, amigable y cómoda para el usuario, utilizando las herramientas accesibles actualmente.

5 Apéndice A: Algunos detalles técnicos

Sobre el Editor de Pruebas Glich:

Plataforma : SUNOS 4 -sparc			
		lineas de codigo	lenguaje
verificador de tipos :	motor	875	Haskell (hbc 1.2, 1.3)
	parser	419	" "
	interfaz	742	" "
modo emacs		326	elisp (xemacs 19)
editor		1292	Haskell (hbc 1.4)
prototipo		873	Lambda Prolog

Sobre el lenguaje Haskell:

- Creado en 1987
- Lenguaje puramente funcional no estricto
- Ultimas versiones:

  ```
  1995 - version 1.3
  1996 - version 1.4
  1998 - Haskell Standard
  ```

- Distribución libre
- Basado en ideas de amplio consenso

6 Apéndice B: Ejemplos de sesiones con el editor

Se describen a modo de ejemplo, tres sesiones con el editor, en las cuales se pueden apreciar los mensajes de error y la posibilidad de escribir comentarios en los archivos.

```
Welcome to the Glich Interactive System and enjoy it !

 enter command: decla Prim Nat : Set.

 enter command: decla cero : Nat.
Ilegal command, try again

 enter command: decla Prim cero : Nat.

 enter command: decla Prim succ : (x:Nat) Nat.

 enter command: decla Impl add : (x,y:Nat) Nat
                                 {
                                   (cero,x) -> x;
                                   (succ x, y) -> succ (add x y)
                                 }.

 enter command: decla Expl one = succ cero : Nat.

 enter command: decla Expl ident = [A,x]x : (A : Set; a:A) A.

 enter command: decla Expl one = succ cero : Nat.

Already defined constant: one

 enter command: quit.


 Bye, bye !
```

Sesión para cargar un archivo:

```
 Welcome to the Glich Interactive System and enjoy it !

 enter command: load Prelude.
------------------------------------------------------------------------
--
```

```
--  BASIC DEFINITIONS
--
-------------------------------------------------------------------------


--
--   Natural Numbers
--

Prim Nat  : Set;
Prim cero : Nat;
Prim succ : (x:Nat) Nat;
Impl natrec : (C:(x:Nat)Set;
               b:C cero;
               f:(u:Nat;p:C u) C (succ u);
               n:Nat
              )
              C  n
{
 (C,b,f,cero)   -> b;
 (C,b,f,succ x) -> f x (natrec C b f x)
};
Impl add :(x,y:Nat)Nat      -- addition of natural numbers
        {
                (cero,x)    -> x;
                (succ x, y) ->  succ (add x y)
        };

--
--  Sigma.  Existential Quantification
--
Prim Sigma : (A:Set)(B:(x:A)Set)Set;
Prim pair  : (A:Set)(B:(x:A)Set)(a:A)(b:B a) Sigma A B;


--
-- Identity
--
Prim Id : (A: Set; x,y:A)Set;
Prim refl : (A:Set; a:A) Id A a a;
Impl idpeel:( A:Set;
             C: (x,y:A; u:Id A x y) Set;
             r: (x:A) (C x x (refl A x));
```

```
              x, y: A;
              u: Id A x y
            ) C x y u
        {
                (A,C,r,x,x,refl A x) -> r x
        };

--
--  Pi.  Universal Quantification
--
Prim Pi : (A:Set; B:(x:A)Set) Set;
Prim lam : (A:Set; B:(x:A)Set; b:(x:A)B x) Pi A B;
Impl funsplit : (A:Set; B:(x:A)Set;
                 C: (u:Pi A B) Set;
                 f: (b:(x:A)B x) C (lam A B b);
                 u: Pi A B
                ) C u
{
 (A,B,C,f,lam A B b) -> f b
};
--
-- Implication
--
Expl Arrow = [A,B]Pi A ([x]B) : (A,B:Set)Set;

Expl apply = [A,B,u,a] funsplit A ([x]B) ([p]B) ([g]g a) u
                                  :(A,B:Set; u: Arrow A B; a:A) B;

--
--  Disjunction.
--

Prim Or : (A, B:Set) Set;

Prim inl : (A, B:Set)(x:A) Or A B;
Prim inr : (A,B:Set)(y:B) Or A B;

Impl orElim: (A:Set; B:Set)
                (C : (u: Or A B) Set;
                 l : (x:A) C (inl A B x);
                 r : (y:B) C (inr A B y);
                 u : Or A B
                )
```

```
                C u
        {
                (A, B, C, l, r, inl A B x) -> l x ;
                (A, B, C, l, r, inr A B y) -> r y
        };


------------------------------------------------------------------------
--
--       Some Examples of proofs
--
------------------------------------------------------------------------

-- Symetry of the Equality

Expl symId = [A,a,b,w] idpeel A ([x,y,p]Id A y x) ([y] refl A y) a b w
           : (A:Set; x,y: A; u: Id A x y) Id A y x;

-- Substitution of equal objects

Expl substId = [X,Y,a,b,u] idpeel X ([a,b,u] Arrow (Y a) (Y b)) ([x] lam (Y x)
                                  ([i]Y x) ([e]e)) a b u
             : (X:Set; Y:(x:X)Set; a:X ; b:X; u:Id X a b) Arrow (Y a) (Y b);

-- Another substitution (without "Arrow")


Expl  substId1 = [X,Y,a,b,u,p] apply (Y a) (Y b) (idpeel X
                                      ([a,b,u] Arrow (Y a) (Y b))
                                      ([x] lam (Y x) ([i]Y x)
                                      ([e]e)) a b u) p
              : (X:Set; Y:(x:X)Set; a:X; b:X; u:Id X a b; p:Y a) Y b;

-- For all natural number n :  (n + 0 = n)

Expl addCero = natrec ([u]Id Nat u (add u cero))  (refl Nat cero)
                      ([x][u] substId1 Nat  ([y]Id Nat (succ x) (succ y))
                       x (add x cero) u (refl Nat (succ x)))
             :   (x:Nat)Id Nat x (add x cero);


--  For all x,y in Nat:  (succ x) + y = x + (succ y)
```

```
Expl lemma1 = [x,y] natrec ([x] Id Nat (add (succ x) y) (add x (succ y)))
 (refl Nat (succ y))([n,w] substId1 Nat  ([z] Id Nat (succ (add (succ n) y))
(succ z))   (add (succ n) y) (add n (succ y)) w  (refl Nat (succ (add (succ n)
 y)))) x

           : (x,y: Nat) Id Nat (add (succ x) y) (add x (succ y));


-- x+y = y+x   implies  (succ x)+y = y + (succ x)

Expl lemma2 =[y,x,u] substId1 Nat ([t] Id Nat (add (succ x) y) (t))
               (add (succ y) x)
               (add y (succ x))
               (lemma1 y x)

(substId1 Nat ([m] Id Nat (succ(add x y)) (succ m)) (add x y)  (add y x) u (refl Nat (succ (ad

           : (y,x: Nat; u : Id Nat (add x y) (add y x)) Id Nat (add (succ x) y)
                                                          (add y  (succ x));


-- conmutativity of addition using lemmas
Expl addCom = [x,y]natrec ([x] Id Nat (add x y) (add y x))  (addCero y) ([x,u] substId1 Nat ([
        : (x,y:Nat) Id Nat (add x y) (add y x);


-- conmutativity of addition without lemma
-- (this term was obtained computing (by program) the normal form of "addcom")
Expl one = succ (cero) : Nat

 enter command: showenv.
The contants are:
["one", "addCom", "lemma2", "lemma1", "addCero", "substId1", "substId", "symId", "orElim", "in

The variables are:
[]


 enter command: defvar x Nat.

 enter command: defvar y Nat.

 enter command: showenv.
```

```
The contants are:
["one", "addCom", "lemma2", "lemma1", "addCero", "substId1", "substId", "symId", "orElim", "in

The variables are:
["y", "x"]

 enter command: defvar x Nat.

Already defined variable: x

 enter command: decla Prim Nat : Set.

Already defined constant: Nat

 enter command: quit.
```

Sesión donde se intenta cargar un archivo con errores:

```
 Welcome to the Glich Interactive System and enjoy it !

 enter command: load PreErr.
Syntax error at line 22

 enter command: quit.


 Bye, bye !
```

7 Apéndice C: Implementación

Se incluye a continuación la implementación de los principales módulos del prototipo, a saber:

- Módulos que contienen funciones sobre los comandos del editor:
  - Com: definición de los comandos
  - Command : lectura de un comando
  - Pars-Com : analizador sintáctico de los comandos
- Módulos que contienen funciones sobre Contexto y Ambiente:
  - Context : funciones sobre el contexto
  - Env : funciones sobre el ambiente
- Módulos que contienen funciones sobre las definiciones:
  - Defs: definición de las definiciones
  - Pars: analizador sintáctico de las definiciones
  - Isvalid : validación de una definición
  - ValidDefs : funciones de validación de las definiciones
- Módulos que contienen funciones sobre los términos:
  - Term : definición de los términos
  - Conv : conversión de términos a forma normal
  - Names : funciones sobre los nombre de las variables
- Módulos que contienen funciones sobre los tipos:
  - Typ : definición de los tipos
  - Tyinf : función que computa el tipo de un término en beta-normal-form (no una abstarcción)
  - Tycheck : verificador de tipos
- Módulos que contienen funciones generales:
  - Lex : analizador lexicográfico
  - Loop : función iterativa y de tratamiento de errores
  - Pars-Comb : combinadores de parsers
  - Save-File : función para guardar un archivo de definiciones
  - Main : programa principal

```
module Com (Com(Quit,ShowEnv,Decla,Load,
       SaveFile,Show,DefVar,Check1,Check2,Clear,ErrCom,Help)) where

import Maybe
import Names
import Term
import Typ
import Defs

data Com = Quit | ShowEnv | Decla Def | Load String | SaveFile String
         | Show VName | DefVar (VName,Typ) | Check1 | Check2 (Term,Typ)
 | Clear | ErrCom | Help deriving (Show, Read)

module Command(getCommand) where

import Context
import Env
import Com
import Pars_Com
import Pars_Comb
import Pars
import List
import Char(isSpace)

-- function getCommand :

getCommand :: String -> Envr -> IO (Com,String)

getCommand (_:"") _ = return (Quit,"")
getCommand l env =
            let ctes = ext env
                vars = dom (snd env)
             in do
                 case (com' (l,1) ctes vars) of
                        (OK (c,_) rest _,_) ->
                                                do
                                                 return (c,rest)
                        (FailAt _ rest _,_) -> let l1 = after '.' rest
                                                 in return (ErrCom,l1)

after :: Char -> String -> String
after c = tail.(dropWhile (/=c))
```

```
module Context (Context(Context),getTypeC, emptyCnt, addC, dom) where

import Names
import Term
import Typ
import MaybeMsg
import List

data Context = Context [(VName,Typ)] deriving (Show,Read)

emptyCnt = Context []

instance Names Context where
free(Context us) = map fst us
allVar (Context us) = nub (map fst us ++ concat (map  (allVar . snd) us))

addC :: Context -> (VName,Typ) -> Context
addC (Context ps) p = Context (p:ps)

getTypeC :: VName -> Context -> MaybeMsg Typ
getTypeC  x (Context gamma) = case gamma of
[]           -> (Wrong ("undefined variable: " ++ x))
((y,t):yts)  -> if x == y   then (return t)
                            else getTypeC x (Context yts)

dom :: Context -> [VName]
dom (Context g) = map fst g

module Conv(conv,whnf,unpack,MatchResult(OkMatch,FailLcl,FailGbl),match,match_s) where

import Names
import Term
import Typ
import Defs
import Assoc
import MaybeMsg

type Subst = [Assoc VName Term]  -- simultaneous substitution

data MatchResult =  OkMatch Subst   -- result of a pattern matching
  | FailLcl
  | FailGbl
```

```
deriving (Show,Read)
--
-- object conversion
--
conv :: Defs -> Term -> Term -> MaybeMsg ()
conv sigma (Abs x a) (Abs y b) = conv sigma a' b'
where
a' = subs x (Var z) a
b' = subs y (Var z) b
z  = fresh_name [a,b]

conv sigma (Abs x a) b = conv sigma a' (App b (Var z))
where
a' = subs x (Var z) a
z  = fresh_name [a,b]

conv sigma a (Abs y b) = conv sigma (App a (Var z)) b'
where
b' = subs y (Var z) b
z  = fresh_name [a,b]

conv sigma a b = let a' = whnf sigma a
     b' = whnf sigma b
         in if isAbs a' || isAbs b' then
conv sigma a' b'
    else
hconv a' b'

where
hconv (Var x) (Var y) = if x==y then
                           return ()
else
   Wrong ("Unconvertible variables "
          ++ x ++ " and " ++ y)
hconv (Con a) (Con b) = if a==b then
                           return ()
else
   Wrong ("Uncenvertible constants "
          ++ a ++ " and " ++ b)
hconv (App a1 b1) (App a2 b2) = hconv  a1 a2 >>
        conv sigma b1 b2
hconv _ _              = Wrong "Este error no deberia ocurrir"
```

```
isAbs (Abs _ _) = True
isAbs _       = False

--
-- Weak head normal form
--
whnf sigma = pack . whnf' . unpack
     where
whnf' a@(Abs _ _,[])  = a   -- abstraction.

whnf' (Abs x a,(b:bs)) = let (a',as) = unpack (subs x b a)
in whnf' (a', as++bs)  -- beta-reduction

whnf' a@(Var _,_)     = a   --variable in head

whnf' a@(Con c, bs) =  case getInfo c sigma of
Primitive _ ->  a            -- constructor in head
Explicit e _  -> let (e',es) = unpack e
           in  whnf' (e',es++bs)   -- unfolding
Implicit eqs _  ->                 -- pattern matching
case redPat sigma bs eqs of
             Just e  -> whnf' (unpack e)
     Nothing -> a




--
-- Pattern Matching
--
redPat :: Defs -> [Term] -> Eqts -> Maybe Term

-- return the result of trying patern matching reduction
--
--  redPat normalizes the list of equation as follows
--
--    Let   es =   [e1,...,em]         list of expressions
--          equ = ([p1,..,pm,...pr],e)  equation
--
-- then we transform equ into
--
--([p1,...,pm], [p(m+1),...,pr]e)
--
```

```
-- This transformation is valid only if p(m+1),...,pr are all variables.
-- We only check this condition over the first equation since we assume
-- the equations are exhaustive and exclusive
--
redPat sigma bs es@((as,e):eqs) = if all isVar (drop (length bs) as) then
             lookMatch sigma bs (map shift es)
  else
     Nothing
where
shift (as,e)  = let (a1s,a2s) = splitAt (length bs) as
    va2s      = [x| Var x <- a2s]
        in  (a1s, foldr Abs e va2s)
isVar (Var _) = True
isVar _       = False

lookMatch :: Defs -> [Term] -> Eqts -> Maybe Term
--
lookMatch sigma es eqs =  case this of
(FailGbl,_)    -> Nothing
(OkMatch rho, e)    -> Just (appSubst rho e)
where
this = head (dropWhile isFailLocal (map matchEq eqs))

isFailLocal (FailLcl,_) = True
isFailLocal _           = False

matchEq (pats,e) = (match_s sigma pats es,e)

match_s sigma pats es = foldl comp (OkMatch []) (map (match sigma) (zip pats es))
where
comp (OkMatch s1) (OkMatch s2) = OkMatch (s1 ++ s2)
comp FailLcl _       = FailLcl
comp FailGbl _       = FailGbl
comp _      FailLcl = FailLcl
comp _      FailGbl = FailGbl

match sigma pair = case pair of
(Var x , e) -> OkMatch [x:=e]
( pat  , e) -> let (he,es)   = unpack (whnf sigma e)
   (hp,pats) = unpack pat
       in  if isConst he then
      if (eqCon he hp) then
 match_s sigma  pats es
```

```
      else
 FailLcl
   else
      FailGbl
   where
isConst (Con c) = case getInfo c sigma of
Primitive _ -> True
_           -> False
isConst _       = False
eqCon (Con a) (Con b) = a == b
eqCon _ _             = False


appSubst :: Subst -> Term -> Term
appSubst rho e = foldl App (foldr Abs e vars) terms
where
vars  = [x| (x:= _) <- rho]
terms = [a| (_:= a) <- rho]


--normal form

nf ::  Defs -> Term -> Term


nf defs a = case whnf defs a of
Abs x e  -> Abs x (nf defs e)
App a b  -> App (nf defs a) (nf defs b)
a        -> a

module Defs (Eqt(),Eqts(),Info(Primitive,Explicit,Implicit),
                 isPrimitive,Def(),Defs(),isDef, getInfo,getTypeD) where
import Names
import Term
import Typ
import MaybeMsg

type Eqt = ([Term],Term)

type Eqts = [Eqt]


data Info =  Primitive Typ
```

```
           | Explicit  Term Typ
   | Implicit  Eqts  Typ
deriving (Show,Read)

type Def  = (VName,Info)

{-- definitions are encoded as a pair (name, information)
--
--   Explicit definition:
--
--       c = e : A  --> ( c, Explicit e A)
--
--
--   Implicit definition:
--              type      : c: A
--              equations : c(p11,...,p1n) = e1
--       ...
--            c(pm1,...,pmn) = em
--
--               --> (c, Implicit eqs A)
--
--where  eqs = [ ([pi1,...,pin],ei) | i <- [1..m]]
-}


type Defs = [Def]

isDef :: VName -> Defs -> Bool
isDef c =  any ((c==).fst)

getInfo               :: VName -> Defs -> Info
getInfo c ((c1,i):ds) | c1 == c     = i
      | otherwise   = getInfo c ds


getTypeD c [] = Wrong ("Undefined constant: " ++ (show c))
getTypeD c ((u,i):ds) | c == u     = case i of
Primitive t  -> return t
Explicit _ t -> return t
Implicit _ t -> return t
      | otherwise  = getTypeD c ds

isPrimitive defs c = isDef c defs &&
```

```
                    case (getInfo c defs) of
                          (Primitive _) -> True
                          _             -> False

module Env where

import Term
import Typ
import List
import Defs
import Context
import ValidDefs
import Istype
import MaybeMsg
import Pars_Comb
import Pars
import IO

type Envr = (Defs, Context)
emptyEnvr :: Envr
emptyEnvr = ([],emptyCnt)

type EnvIO a = Envr -> IO a

showEnvr :: EnvIO ()
showEnvr env = let ctes = ext env
                   vars = dom (snd env)
                   s =  "The contants are: \n" ++ showList ctes "" ++ "\n\n" ++
                        "The variables are: \n" ++ showList vars "" ++ "\n\n"
                in putStr s

decla :: Def -> EnvIO Envr
decla def = \env -> let defs = fst env
                     in case (validCte def defs) of
                          Wrong s -> fail (userError ("\n" ++ s ++ "\n"))
                          Ok _    -> case (validDef def defs) of
                                          Wrong s -> fail (userError (s ++ "\n"))
                                          Ok _    -> do
                                                       e1 <- update (insertDef def) env
                                                       return e1

loadfile :: String -> EnvIO Envr
loadfile fn = \env -> let dffs = fst env
```

```
                          ctes = ext env
                          vars = dom (snd env)
                          argu = fn ++ ".E"
                        in
                         do
                          contents <- readFile argu
                          case contents of
                            "" -> do
                                  putStr ("\n file " ++ argu ++ "not found or empty \n")
                                  return env
                            _  -> do
                                  putStr contents
                                  case (defs (contents, 1) ctes vars) of
                                   (OK ((d:ds),_) _ _,_) ->do
                                                           e1 <- insDefs (d:ds) env
                                                           return e1
                                   (FailAt _ _ pos,_)    -> fail (userError ("\nerror at line:
{-
-- load File using handles (to be done in the future)

loadfile :: String -> EnvIO Envr
loadfile fn = \env -> let dffs = fst env
                          ctes = ext env
                          vars = dom (snd env)
                          argu = fn ++ ".E"
                        in
                         do
                          fhandle <- openFile argu ReadMode
                          contents <- hGetContents fhandle
                          hClose fhandle
                          putStr contents
                          case (defs (contents, 1) ctes vars) of
                             (OK ((d:ds),_) _ _,_) ->do
                                                      e1 <- insDefs (d:ds) env
                                                      return e1
                             (FailAt _ _ pos,_)    -> fail (userError ("\nerror at line: " ++ s
-}
defvar :: (VName,Typ) -> EnvIO Envr
defvar (var,typ) = \env -> let defs = fst env
                               ctex = snd env
                           in if (not (isOk (istype defs ctex typ)))
                              then fail (userError ("\n" ++ show typ ++ "ilegal typ \n"))
                              else
```

```
                                if (isDef var defs)
                                then fail (userError ("\n" ++ var ++ " cte. already defined \n"
                                else
                                if (var `elem` (dom ctex))
                                then fail (userError ("\n" ++ var ++ " var. already defined \n"
                                else return (defs,addC ctex (var,typ))

update :: (Envr -> Envr) -> EnvIO Envr
update f = \e -> return (f e)

insertDef :: Def -> Envr -> Envr
insertDef d (ds,ct) = (d:ds,ct)

insDefs :: Defs -> Envr -> IO Envr
insDefs [] env = return env
insDefs (d:ds) env = decla d env >>= \y -> insDefs ds y

ext :: Envr -> [VName]
ext (ds,_) = map fst ds

module Istype(istype) where

import Names
import Term
import Typ
import Context
import Defs
import MaybeMsg
import Tycheck

istype :: Defs -> Context -> Typ -> MaybeMsg ()

istype _ _ Set = return ()
istype sigma gamma (El a) = tycheck sigma gamma a Set
istype sigma gamma (Fun x alpha beta) = istype sigma gamma alpha >>
istype sigma gamma' beta'
where
z      = fresh_name (gamma,beta)
gamma' = addC gamma (z,alpha)
beta'  = subs x (Var z) beta

module Isvalid(isvalid) where
```

```
import Term
import Typ
import Context
import Defs
import Maybe
import Istype

isvalid :: Defs -> Context -> Bool

isvalid sigma gamma = isvalid'  (reverse gamma)
where
isvalid' []             = True
isvalid' ((x,alpha):bs) = isvalid' bs
&&
  not (x `elem` (map fst bs))
&&
  istype sigma bs alpha
module Lex where

import Char

type Rest = String
type Token = String
type Inp = String

lexer :: Int -> Inp -> ((Token,Rest),Int)

lexer i [] = (([],[]),i)
lexer i xxs@('\n':xs) = lexer (i+1) xs
lexer i     ('-':'-':xs) = lexer i (dropWhile (/='\n') xs)
lexer i xxs@(x:xs) = if (space x) then lexer i xs
                     else if isAlphanum x
      then (token (span isAlphanum xxs),i)
      else if x=='\"' then
         -- proceso string
         let
   (string,resto) = span (/='\"') xs
         in ((string,tail resto),i)
                     else (([x],xs),i)

space x = (x == ' ' || x == '\t' || x == '\r' || x == '\f' || x == '\v')

token (t,xs) = (t,xs)
```

```
module Loop(loop) where

import Context
import Defs
import Com
import Command
import Env
import IO

loop l e = do
            putStr "\n enter command: "
            (com,rest) <- getCommand l e
            case com of
             ErrCom           -> do
                                  putStr "ilegal command, try again \n"
                                  loop rest e
             Quit             -> return ()
             Help             -> do
                                  putStr (helpmsg ())
                                  loop rest e
             ShowEnv          -> do
                                  showEnvr e
                                  loop rest e
             Decla def        -> do
                                  e1 <- catch (decla def e) (errfun e)
                                  loop rest e1
             DefVar (var,typ) -> do
                                  e1 <- catch (defvar (var,typ) e) (errfun e)
                                  loop rest e1
             Load fn          -> do
                                  e1 <- catch (loadfile fn e) (errfun2 fn e)
                                  loop rest e1
             SaveFile fn      -> do
                                  script l
                                  loop rest e
             Show id          -> do
             Check1           ->
             Check2 (te,ty)   ->

errfun e = (\err -> case (isUserError err) of
                    (Just err) -> putStr err >>= \() -> return e
                    (Nothing)  -> return e)
```

```
errfun2 fn e = (\_ -> putStr ("Cannot open " ++ fn ++ "\n") >>= \() -> return e)

helpmsg :: () -> String
helpmsg () = "\n The following commands are available: \n\n" ++
             "quit:  quit the system \n" ++
             "showenv: show the current environment \n" ++
             "decla definition: add a constant definition to the environment \n" ++
                       "where definition may have the following forms: \n" ++
                                "Prim name : type, \n" ++
                                "Impl name : type {eqts}, \n" ++
                                "Expl name = term : type \n" ++
                                        "where name is the name of a constant, \n" ++
                                              "term is an expression, \n" ++
                                              "type is a type, \n" ++
                                              "eqts are equations \n" ++
             "load name: load a file, where name specifies what should be loaded \n" ++
             "save name: save the file, where name specifies the name of the file \n" ++
             "show name: give information about the term named name \n" ++
             "defvar var type: add a variable definition to the current environment \n" ++
                "where var specifies the name of the variable and type its type \n" ++
             "check: \n" ++
             "check term type: \n" ++
             "clear: \n"

module Main(main) where

import Env
import Loop
import List
import IO

main :: IO ()

main = do
        putStr "\n Welcome to the Glich Interactive System and enjoy it ! \n"
        l <- getContents
        do
         body <- loop l emptyEnvr
         case body of
          () -> do
                 putStr "\n Bye, bye ! \n"
```

```
module Names  where

import List

class Names a where
free      :: a -> [String]
-- return a list of the variables which occur free in a
allVar    :: a -> [String]

fresh_name :: a -> String
-- return a new name which does not occur in a
fresh_name   = head . (names \\) . allVar
                where
   names = simples ++ [x ++ show i| i <- [0..],
 x <- simples]
   simples = ["x","y","z","u","w"]



--instance Names for lists

instance (Names a) => Names [a] where

free [] = []
free (x:xs) = union (free x) (free xs)

allVar (x:xs) = union (allVar x) (allVar xs)
allVar [] = []




-- instance Names for pairs

instance (Names a,Names b) => Names (a,b) where

free (a,b) = free a  `union` free b
allVar (a,b) = allVar a `union` allVar b


--instance Names for 3-uples

instance (Names a, Names b, Names c) => Names (a,b,c) where
```

```
free (a,b,c) = union (union (free a) (free b))  (free c)
allVar (a,b,c) = union (union (allVar a)  (allVar b))  (allVar c)


--instance Names for 4-uples
instance (Names a,Names b,Names c,Names d) => Names (a,b,c,d) where
free (a,b,c,d) =  (union (free a)  (free b)) `union` (free c `union` free d)
allVar (a,b,c,d) = (allVar a `union` allVar b) `union` (allVar c `union` allVar d)


module Pars (defs) where

import Names
import Term
import Typ
import Defs
import Lex
import Pars_Comb
import Char

term' = succeed id `ap` term
                   `chk` literal "."
typ'  = succeed id `ap` typ
                   `chk` literal "."

defs = lstNoNil def'

def' = initP def


def = ((literal "Prim"
      `alt`
      literal "Expl"
      `alt`
      literal "Impl") `using` (\v -> case v of
                                      "Prim" -> primitive
                                      "Expl" -> explicit
                                      "Impl" -> implicit))
      `into` pars

primitive = (variable `chk` literal ":") `seqq` (succeed Primitive `ap` typ)

explicit = (variable `chk` literal "=") `seqq` (succeed Explicit `ap` term2
```

```
                                                          `chk` literal ":"
                                                          `ap` typ)
implicit = (globalvar `chk` literal ":") `seqq` (succeed swcu `ap` typeqts)

typeqts = (typ `chk` literal "{") `seqq` (eqts `chk` literal "}")

swcu (v,w) = Implicit w v

eqts = lstNoNil eqt'

eqt' = initP eqt

eqt = (succeed id `chk` literal "("
                  `ap` (listNoNil term4)
                  `chk` literal ")"
                  `chk` literal "-"
                  `chk` literal ">")
     `seqq`
     term2

typ = succeed Set `chk` literal "Set"
      `alt`
      succeed El `chk` literal "El"
                 `chk` literal "("
                 `ap` term
                 `chk` literal ")"
      `alt`
      succeed El `chk` literal "("
                 `ap` term
                 `chk` literal ")"
      `alt`
      succeed El `ap` term
      `alt`
      funtyp seqtyp typ

distribut ([],t) = []
distribut ((x:xs),t) = (x,t) : distribut (xs,t)

myuncurry :: (String -> Typ -> Typ -> Typ) -> (String,Typ) -> Typ -> Typ
myuncurry f (x,y) z = f x y z

funtyp p1 p2 = p1 `into`` \v -> p2 `using`` \w -> foldr (myuncurry Fun) w (concat (map distrib
```

```
term = term0
       `alt`
       term2
       `alt`
       term4

term0 = succeed (foldl1 App) `ap` some term1

term2 = succeed (foldl1 App) `ap` some term3

term4 = succeed (foldl1 App) `ap` some term5

abstraction p1 p2 = p1 `into`` \v -> p2 `using`` \w -> foldr Abs w v

term1 = abstraction seqquence term0
        `alt`
        atom

term3 = abstraction seqquence term2
        `alt`
        atom1

term5 = abstraction seqquence term4
        `alt`
        atom2

atom = succeed Var `ap` boundvar
       `alt`
       succeed Con `ap` konstant
       `alt`
       succeed Var `ap` variable
       `alt`
       succeed id `chk` literal "("
                  `ap` term0
                  `chk` literal ")"

atom1 = succeed Var `ap` boundvar
        `alt`
        succeed Con `ap` konstant
        `alt`
        succeed id `chk` literal "("
                   `ap` term2
                   `chk` literal ")"
```

```
atom2 = succeed Var `ap` boundvar
        `alt`
        succeed Con `ap` konstant
        `alt`
        succeed Var `ap` localvar
        `alt`
        succeed id `chk` literal "("
                  `ap` term4
                  `chk` literal ")"


variable = satisfy (all isAlphanum)
cadena = succeed id
                  `ap` satisfy  (\_->True)



konstant = iskonstant variable

boundvar = isboundvar variable

localvar = tick1 variable

globalvar = tick variable

seqquence = succeed id `chk` literal "["
                      `ap` (listNoNil localvar)
                      `chk` literal "]"

seqtyp = succeed id `chk` literal "("
                    `ap` (lstNoNil binding)
                    `chk` literal ")"

binding = (listNoNil localvar `chk` literal ":") `seqq` typ

par = term2 `seqq` typ

module Pars_Com(com') where

import Maybe
import Term
```

```
import Typ
import Defs
import Com
import Lex
import Pars_Comb
import Pars
import Names

com' = succeed id `ap` com
                  `chk` literal "."

com = ((literal "quit"
      `alt`
      literal "showenv"
      `alt`
      literal "decla"
      `alt`
      literal "load"
      `alt`
      literal "savefile"
      `alt`
      literal "show"
      `alt`
      literal "defvar"
      `alt`
      literal "check"
      `alt`
      literal "help") `using` (\v -> case v of
                                        "quit"     -> succeed Quit
                                        "showenv"  -> succeed ShowEnv
                                        "decla"    -> succeed Decla `ap` def
                                        "load"     -> succeed Load `ap` variable
                                        "savefile" -> succeed SaveFile `ap` variable
                                        "show"     -> succeed Show `ap` variable
                                        "defvar"   -> succeed DefVar `ap` (globalvar `seqq` t
                                        "check"    -> (succeed Check2 `ap` par)
                                                      (succeed Check1)
                                        "help"     -> succeed Help
                                        _          -> succeed ErrCom
                                  ))
      `into` pars

module Pars_Comb where
```

```
import Lex

infixr 6 `alt`
infixl 8 `ap`, `chk`, `into` , `chk2`

type Env = [String]

data Return c = FailAt Token Rest Int | OK c Rest Int deriving (Eq)
type MParser b = (Inp, Int) -> Env -> Env -> (Return (b,Env), Env)

succeed :: b -> MParser  b
succeed a (xs, i) e e' = (OK (a,e') xs i,e)

into :: MParser  b -> (b -> MParser  c) -> MParser  c
into p f (inp, i) e e' = case p (inp, i) e e' of
                           (OK (v,e1) ys i,e2) -> f v (ys,i) e2 e1
                           (FailAt v ys i ,e2) -> (FailAt v ys i ,e2)

into' p f (inp, i) e e'= case p (inp, i) e e' of
                      (OK (s,e1) xs i, e2)-> case f s (xs,i) e2 e1 of
                                  (OK (v,_) ys i,e3) -> (OK (v,e') ys i,e3)
                                  (FailAt _ _ i ,e3)-> f s (xs,i)  e2 e'
                      (FailAt s xs i ,e2) -> (FailAt s xs i ,e2)

satisfy :: (Token -> Bool) -> (Inp,Int) -> Env -> Env -> (Return (String,Env), Env)
satisfy  p ([], i) e e' = (FailAt [][] i ,e)
satisfy p (xs, i) e e' = let ((t,ys),ii) = lexer i xs
                         in if (p t) then (OK (t,e') ys ii,e)
                                     else (FailAt t ys i, e)

satisfy1 pred ([], i) e e' = (FailAt [][] i , e)
satisfy1 pred (xs, i) e e' = let ((t,ys),ii) = lexer i xs
                             in if (pred t e) then (OK (t,e') ys ii,e)
                                              else (FailAt t ys i ,e)

satisfy2 pred ([], i) e e' = (FailAt [][] i ,e)
satisfy2 pred (xs, i) e e' = let ((t,ys),ii) = lexer i xs
                             in if (pred t e') then (OK (t,e') ys ii,e)
                                               else (FailAt t ys i ,e)

eof ([], i) e e' = (OK ((),e') [] i,e)
eof (x:xs, i) e e' = let ((t,ys),_) = lexer i (x:xs)
```

```
                    in (FailAt t ys i, e)

ap :: MParser (b->c) -> MParser b -> MParser c
p1 `ap` p2 =  p1 `into` \f -> p2 `into` \x -> succeed (f x)

ap' :: MParser (b->c) -> MParser b -> MParser c
p1 `ap'` p2 = p1 `into` \f -> p2 `into'` \x -> succeed (f x)

alt :: MParser b -> MParser b -> MParser b
alt p1  p2 (inp, i) e e' = case p1 (inp, i) e e' of
                              (FailAt t ys  i ,e'') -> p2 (inp, i) e'' e'
                              (OK v ys i,e'') -> (OK v ys i,e'')

chk :: MParser b -> MParser c -> MParser b
p1 `chk` p2 = p1 `into` \f -> p2 `into` \_ -> succeed f

chk2 :: MParser b -> MParser c -> MParser c
p1 `chk2` p2 = p1 `into` \_ -> p2 `into` \f -> succeed f

p `using` f = p `into` \v -> succeed (f v)

p `using'` f = p `into'` \v -> succeed (f v)

seqq :: MParser b -> MParser c -> MParser (b,c)
p1 `seqq` p2 = p1 `into` \v -> p2 `using` \w -> (v,w)

many :: MParser b -> MParser [b]
many p = succeed (:) `ap` p `ap` many p `alt` succeed []

some p = succeed (:) `ap` p `ap` many p

listOf :: MParser b -> MParser c -> MParser [b]
listOf p s = let q = many (s `chk2` p) in succeed (:) `ap` p `ap` q `alt` succeed []

noNilList :: MParser b -> MParser c -> MParser [b]
noNilList p s = let q = many (s `chk2` p) in succeed (:) `ap` p `ap` q

commaList p = listOf p (literal ",")

listNoNil p = noNilList p (literal ",")

lstNoNil p = noNilList p (literal ";")
```

```
literal p = satisfy (p==)

option p = succeed (\x->[x]) `ap` p
           `alt` succeed []

p `no_yes` (no,yes) = succeed f `ap` p
                      where f [] = no
                            f [x] = yes x

iskonstant p (inp,i) e e' = case p (inp,i) e e' of
                              (OK (s,e1) inp' i, e'') -> if s `elem` e''
                                                         then (OK (s,e1) inp' i, e'')
                                                         else (FailAt s inp' i, e'')
                              (FailAt s inp' i, e'')  -> (FailAt s inp' i, e'')

isboundvar p (inp,i) e e' = case p (inp,i) e e' of
                              (OK (s,e1) inp' i, e'') -> if s `elem` e1
                                                         then (OK (s,e1) inp' i, e'')
                                                         else (FailAt s inp' i, e'')
                              (FailAt s inp' i, e'')  -> (FailAt s inp' i, e'')

tick p (inp, i) e e' = case p (inp, i) e e' of
                         (OK (s,e1) inp' i,e'') -> if s `elem` e''
                                                   then (FailAt s inp' i ,e'')
                                                   else (OK (s,e1) inp' i, s:e'')
                         (FailAt s inp' i ,e'') -> (FailAt s inp' i ,e'')

tick1 p (inp, i) e e' = case p (inp, i) e e' of
                              (OK (s,e1) inp' i, e'') -> if s `elem` e1
                                                            then (OK (s,e1) inp' i,e'')
                                                            else (OK (s,s:e1) inp' i ,e'')

                              (FailAt s inp' i ,e'') -> (FailAt s inp' i ,e'')

tickP :: MParser ()
tickP (ys, i) e e' = (OK ((),[]) ys i, e)

initP :: MParser b -> MParser b
initP p = tickP `into` \_ -> p `into` \f ->  succeed f

semicolon p = succeed id `ap` p
                         `chk` literal ";"
```

```
pars parser (input,i) e e' = parser (input, i) e e'

module Save_File(save_defs) where

muestro_eqt_save :: Eqt -> String -> String
muestro_eqt_save ([],t) = showString "() -> " . shows t
muestro_eqt_save ((arg:args),t) =
               showChar '(' . (shows arg) . (muestro_eqt_save' args)
               where muestro_eqt_save' [] = showString ") -> " . shows t
                     muestro_eqt_save' (arg:args) = showChar ',' . (shows arg) .

muestro_eqts_save :: Eqts -> String -> String
muestro_eqts_save [] = id
muestro_eqts_save (eqt:eqts) =
               showChar '{' . (muestro_eqt_save eqt) . (muestro_eqts_save' eqts)
                     muestro_eqts_save' (eqt:eqts) = showChar ';' .
                                                     (muestro_eqt_save eqt) .
                                                     (muestro_eqts_save' eqts)

muestro_def_save :: Def -> String
muestro_def_save (cte,info) =
           case info of
                Primitive typ -> "Prim " ++ cte ++ " : " ++ show typ
                Explicit term typ -> "Expl " ++ cte ++ " = " ++ show term ++
                                     " : " ++ show typ
                Implicit eqts typ -> "Impl " ++ cte ++ " : " ++  show typ ++
                                     " " ++ muestro_eqts_save eqts ""

muestro_defs_save :: Defs -> String
muestro_defs_save [def] = muestro_def_save def
muestro_defs_save (def:def2:defs) = muestro_def_save def  ++ ";\n" ++
                                    muestro_defs_save (def2:defs)

save_defs = muestro_defs_save

module Term(Term(Con,Var,Abs,App),CName,VName, Subs(rename,subs),
            unpack,pack) where

import List
import Names
import Char

type VName = String
```

```
type CName = String

data Term = Con CName      -- constants
          | Var VName      -- variables
          | Abs VName Term -- abstraction
          | App Term Term  -- application



-- instancia de la clase Text
-------------------------------------
-------------------------------------
--Funciones auxiliares para readsPrec
-------------------------------------

lexP :: ReadS String
lexP ('.':s) = [(".",s)]
------


instance Show  Term where
   showsPrec _ (Con c) = showString c
   showsPrec _ (Var x) = showString x
   showsPrec n (Abs x e) = showParen (n>5) (
                               showChar  '[' .
                               showString x .
                               showChar  ']' .
                               showsPrec 5 e
                           )

   showsPrec n (App e1 e2) = showParen (n>7) (
                                  showsPrec 7 e1 .
                                  showChar ' ' .
                                  showsPrec 8 e2
                             )

instance Read Term where
   readsPrec d r = readParen (d>10) readCon r
                   ++
   readParen (d>10) readVar r
   ++
   readParen (d>5) readAbs r
   ++
```

```
   readParen (d>7) readApp r

readCon xs = [(Con x,rs) | (x@(c:_),rs) <- getCons xs]


readVar xs = [(Var x,rs) | (x,rs) <- getVar xs]

readAbs xs = [(Abs x e,ws) | ("[",rs) <- lex xs,
                             (x@(c:_),us) <- getVar rs,
     ("]",vs) <- lex us,
                             (e,ws) <- readsPrec 5 vs]
readApp xs = [(foldl App m ms,us) | (m,rs) <- readsPrec 8 xs,
                                    (ms,us) <- readApp' rs]
             where readApp' xs = [(m:ms,us) | (m,rs) <- readsPrec 8 xs,
                                              (ms,us) <- readApp' rs]
                                 ++
                                 [([m],rs) | (m,rs) <- readsPrec 8 xs
                                                       ++
                                                       readAbs xs]

getCons xs =[ (x,rs) | (x@(c:cs),rs) <- lex xs ,
                                 isAlphanum c,
not (toUpper c `elem` "XYZUVW") ,
not (x `elem` ["Set","El"])]
getVar xs = [ (x,rs) | (x@(c:cs),rs) <- lex xs,
toUpper c `elem` "XYZUVW"]

-- Clase de sustitucion

class  Subs a where

    rename   :: VName -> VName-> a  -> a

    subs     :: VName -> Term -> a -> a


instance (Subs a) => Subs [a] where
subs x e   = map  (subs x e)


-- instancia de la clase Names
------------------------------
```

```
instance Names Term where
free a = free' a []
where
free' (Con _) = id
free' (Var x) = nub . (x:)
free' (Abs x e) = (\\[x]). free' e
free' (App a b) = nub . free' a . free' b

allVar (Con _) = []
allVar (Var x) = [x]
allVar (Abs x e) = nub (allVar e ++ [x])
allVar (App a b) = nub (allVar a ++ allVar b)

-- instancia de la clase Subs
----------------------------

instance Subs (Term) where

    rename x z (Con y)  = Con y
    rename x z (Var y) | x == y = Var z
                       | otherwise = Var y
    rename x z (App e1 e2)  = App exp1 exp2
                              where exp1 = rename x z e1
                                    exp2 = rename x z e2
    rename x z (Abs y e1) | x == y = Abs y e1
                          | otherwise = Abs y exp1
                                        where exp1 = rename x z e1

    subs  x e  (Con y)  = Con y
    subs  x e  a@(Var y) | x == y = e
                         | otherwise = a
    subs x e  (App e1 e2) = App exp1 exp2
                                where exp1 = subs x e  e1
                                      exp2 = subs x e  e2
    subs x e  a@(Abs y e1) | x == y    = a
                           | otherwise = Abs z exp3

                                where
                                      z    = fresh_name (e1,e, Var x)
                                      exp2 = rename y  z e1
                                      exp3 = subs x e  exp2
--
unpack :: Term -> (Term,[Term])
```

```
--
unpack a  = (h,as)
where
      (h:as) = unpack' a []
      unpack' (App a b) = unpack' a . (b:)
      unpack' u         = (u:)

--
pack  :: (Term,[Term]) -> Term
--
pack  (h,as) = foldl App h as

module Tycheck (tycheck) where

import Names
import Term
import Typ
import Defs
import Context
import Tyconv
import MaybeMsg

tycheck :: Defs -> Context -> Term -> Typ -> MaybeMsg ()
--
-- {-  check that a term has a given type under
--      a context and a collection of definitions-}
--

tycheck sigma gamma (Abs x a) (Fun y alpha beta) =     -- abstraction
                tycheck sigma gamma_za  axz betayz
where
gamma_za = addC gamma (z, alpha)
                axz      = subs x (Var z) a
betayz   = subs y (Var z) beta
z        = fresh_name (gamma,a,beta)

tycheck _ _ term@(Abs _ _) _ = Wrong ("\nFuncional type is required for\n\t"
                                      ++
     (show term))

tycheck sigma gamma a alpha =
tyinf sigma gamma a >>= \beta ->
tyconv sigma alpha beta
```

```
tyinf :: Defs -> Context -> Term -> MaybeMsg Typ
--
-- compute the type of a term in beta-normal form (not abstraction)
--
tyinf sigma gamma (Var x)  = getTypeC x gamma
tyinf sigma gamma (Con x)  = getTypeD x sigma

tyinf sigma gamma (App a b) =
tyinf sigma gamma a  >>= \typ_a ->
case typ_a of
(Fun w alpha beta) ->
     tycheck sigma gamma b alpha >>= \()->
     return (subs  w b beta)
no_fun ->
     Wrong ("\nA functional type is required for\n\t"
           ++
   (show a))

module Tyinf(tyinf) where

import Names
import Term
import Typ
import Defs
import Context
import Tycheck
import Maybe

tyinf :: Defs -> Context -> Term -> Maybe Typ
--
-- compute the type of a term in beta-normal form (not abstraction)
--
tyinf sigma gamma (Var x)  = getTypeC x gamma   -- variable
tyinf sigma gamma (Con c) = getTypeD c sigma  -- constant
tyinf sigma gamma (App a b) = case (tyinf sigma gamma a) of    -- application
Just (Fun w alpha beta) -> if tycheck sigma gamma b alpha then
Just (subs  w b beta)
   else
Nothing
_                       -> Nothing

module Typ (Typ(Set,El,Fun)) where
```

```
import Term
import Names
import List
import Char

data Typ = Set
         | El Term
         | Fun VName Typ Typ


instance Show Typ where
showsPrec _ Set = showString "Set"
showsPrec _ (El a) = shows a
showsPrec p (Fun x a b) = showChar '('.
  showString x.
  showString " : " .
  shows a .
  showChar ')'.
  showsPrec p b

instance Read Typ where
readsPrec d r = readParen (d>10) readSet r
++
readParen (d>10) readEl r
++
readParen (d>10) readFun r

readSet xs = [(Set,rs)| ("Set",rs) <- lex xs]
readEl  xs = [(El a,rs)| ("El",us) <- lex xs,
 ( a , rs) <- reads us]
     ++
     [(El a,rs) | (a,rs) <- reads xs]
readFun xs = [ ((Fun x a b),rs) | ("(",us) <- lex xs,
     (x, ws) <- lex us, isAlphanum (head x),
     (":",zs) <- lex ws,
     ( a, ts) <- reads zs,
     (")", vs) <- lex ts,
     (b, rs)   <- reads vs]

-- instancia de la clase Names
-----------------------------
```

```
instance Names Typ  where

free Set = []
free (El a) = free a
free (Fun x t1 t2) = (nub (free t1 ++ free t2)) \\ [x]

allVar Set = []
allVar (El a) = allVar a
allVar (Fun x t1 t2) =nub ( x : (allVar t1 ++ allVar t2))

-- instancia de la clase Subs
----------------------------

instance Subs (Typ) where


    rename x z Set  = Set
    rename x z (El e) = El (rename x z e)
    rename x z a@(Fun y t1 t2)  | x == y = a
                                | otherwise = Fun y (rename x z t1)
                                                    (rename x z t2)



    subs x e  Set  = Set
    subs x e  (El e1) = El (subs x e e1)

    subs x e a@(Fun y t1 t2) | x == y    = a
                             | otherwise = Fun z t1' t2'

where z   = fresh_name (t1,t2,(e,Var x))

            t1' = subs x e t1
                              t2' = subs x e (rename y  z t2)


module ValidDefs(validCte,validDef,validDefs, validExtension) where

import Names
import Term
import Typ
import Context
import MaybeMsg
import Defs
```

```
import Tycheck
import Istype
import Tyconv

--data ResultValidDef = OkDef | ErrorDef String

validCte :: Def -> Defs -> MaybeMsg ()
validCte def defs = let name = fst def
                    in
                     if (isDef name defs)
                     then Wrong ("Already defined constant: " ++ name)
                     else return ()

validDef :: Def -> Defs -> MaybeMsg ()
validDef (name,info)  defs = case info of
Primitive typ      -> istype defs emptyCnt typ
Explicit value typ -> istype defs emptyCnt typ >>
      tycheck defs emptyCnt value typ
Implicit eqs typ   -> istype defs emptyCnt typ
       >>
      validEqs defs eqs name typ

validDefs :: Defs -> MaybeMsg ()
validDefs ds = validExtension  [] ds

validExtension  defs  [] =  return ()
validExtension  defs (d:ds) = validDef d defs  >>
                              validExtension (d:defs) ds

validEqs :: Defs -> Eqts -> CName -> Typ -> MaybeMsg ()
validEqs defs eqs name typ =
case eqs of
[]        -> return ()
(eq:eqs') -> validEq defs name typ eq >>
             validEqs defs eqs' name typ

validEq :: Defs -> CName -> Typ -> Eqt -> MaybeMsg ()
validEq  defs   name typ (pats,result)=
           if all (isPat defs) pats
           then tycheckEq defs name typ pats result
   else Wrong ("Invalid pattern in definition of: " ++ name)
```

```
isPat  _  (Var x)   = True
isPat  defs (Con c) = isPrimitive defs c
isPat _ (Abs _ _)     = False
isPat  defs (App a b)    = isPat' a && isPat defs b
              where
                   isPat' (Var x) = False
                   isPat'  a      = isPat defs a

tycheckEq defs name typ pats result =
        checkPats defs emptyCnt pats typ >>= \(context,typ')->
        tycheck defs' context result typ'
where
  defs' = (name, Primitive typ):defs  -- esto esta feo
      -- encapsular!



--
tycheckPat :: Defs -> Context -> Term -> Typ -> MaybeMsg  Context
--
tycheckPat defs context (Var x) typ =
case (getTypeC x context) of
   Wrong _   -> return (addC context (x,typ))
                  Ok typ'   -> tyconv defs typ typ' >>
            return  context
tycheckPat defs context term typ =
let (Con c, pats) = unpack term
                in  getTypeD c defs
               >>= \typc ->
    checkPats defs context pats typc
                               >>= \(context',typ')->
            tyconv defs typ' typ >>
                    return context'

checkPats defs context [] typ = return (context,typ)
checkPats defs context (p:ps) (Fun x alpha beta) =
tycheckPat defs context p alpha  >>= \ context' ->
checkPats defs context' ps (subs x p beta)
```