- La lógica, proporciona una formulación **simbólica** e **independiente del dominio** de las leyes del pensamiento humano. Este doble carácter hace posible mecanizar sus técnicas y métodos.

- PROBLEMA:
  - Tiene un carácter **estático**, se desarrolló para estudiar objetos matemáticos bien definidos y consistentes, se requieren formas más dinámicas (y menos perfectas) de lógica.
  - Los métodos de la lógica resultan más **caros** en términos computacionales, es necesario reducir sus costes.
- SOLUCIÓN: **Lógica computacional**, modela el conocimiento impreciso, incompleto, dinámico, distribuido. Soporta el razonamiento aproximado, temporal, no monótono,...

Lógicas para Aplicaciones software:
Restricciones: - *Lógica ecuacional*
Extensiones de la lógica: - *Lógica difusa - Lógica Lineal - Lógica Modales*
- Lógica ecuacional:

- Subconjunto de la lógica de primer orden, se eliminan todas las conectivas lógicas. El símbolo de predicado es la (=).
- s=t, dos términos son semánticamente iguales, aunque son diferentes sintácticamente. *Ej: ascii('0') = 48*
- Un conjunto E de ecuaciones con el mismo símbolo "raíz" f en las partes izquierdas de las ecuaciones se describe como "la definición" de f. *even(0)=true - even(X)=even(X-2)*

Lógica para la programación:
Restricciones - Estilo de Lenguaje:
- *Lógica ecuacional - Funcional (Haskell)*
- *Lógica clasual - Relacional (Prolog)*
Extensiones:
- *Lógica many-sorted +tipos*
- *Lógica order-sorted +herencia*
- *Lógica (modal) temporal +concurrencia*
- *Lógica (modal) dinámica +objetos*

- Unificación de lógicas: Lógica de Reescritura (RWL), lógica del cambio que permite especificar la dinámica de un sistema.

- Integración "sin costuras" de distintas características: funciones, y tipos, indeterminismo, concurrencia, reflexión y genericidad.
- Marco unificado en el que pueden definir distintas lógicas: ecuacional, clasual, lineal.
- Existe una lógica temporal, asociada a la RWL, estrictamente más potente que CTL/LTL: la temporal logic of rewriting (LTR).

- Maude:, implementa eficientemente la lógica RWL.

- Soporta de forma natural la especificación formal / modelado / programación en un estilo funcional.
- Distingue entre la parte concurrente y funcional.
- Reescritura módulo listas, conjuntos, multiconjuntos,... mediante atributos ecuacionales.
- Genericidad y tipos ordenados de datos.
- Infraestructura para análisis y verficación formal (alcanzabilidad, model-checking, theorem proving).
- Reflexión como soporte al meta-modelado, ejecución simbólica y construcción rápida de herramientas de soporte.

- Lenguaje Maude

- Sintaxis: Basada en ecuaciones y reglas de reescritura (estilo Haskell, ML, Scheme o Lisp)
- Semántica: Basada en la Lógica de Reescritura (RWL), que modela funciones, concurrencia y objetos.

- Fundamentos de Maude, consta de tres tipos de módulos:

- Módulos funcionales *fmod <conjunto de ecuaciones >endfm*
- Módulos de sistema *mod <conjunto de ecuaciones/reglas de escritura >endm*
- Módulos O2 *omod ... endom*

- E (Ecuaciones):

- Definen funciones confluentes y terminantes.
- Se definen dentro de módulos funcionales.
- E puede incluir un conjunto Ax de Axiomas algebraicos.
- Representan la parte estática del sistema.
- Se aplican de forma determinista.

- R (Reglas de reescritura):

- Definen funciones que pueden ser no confluentes y/o no terminantes.
- Se definen dentro de módulos de sistema.
- Especifican la dinámica del sistema, es decir, acciones que puedan producir transiciones del mismo.

- Paso de reescritura (Maude step), dado un término (o estado) s, un paso de reescritura de t a t' se consigue aplicando una regla de R módulo las ecuaciones de E.
El estado t se simplifica usando las ecuaciones de E hasta alcanzar su forma irreducible (tE) con respecto E.
Una traza de ejecución es una secuencia de Maude steps. Se representa como: t —— >t'
- Reescritura módulos axiomas, dado E=Eo U Ax, las ecuaciones de Ax no se usan para reescribir sino para hacer un pattern-matching especializado. Cuando el conjunto de axiomas algebraicos no es vacio, cada paso de reescritura se hace sustituyendo el pattern-matching convencional por pattern-matching módulo axiomas.
- Árbol de ejecución, un árbol de computación es un conjunto de secuencias de Maude steps que se organizan en forma de árbol.
- Optimización, Maude mantiene en memoria versiones normalizadas de las reglas y ecuaciones del programa, donde sus lh's están normalizadas respecto de Ax.
En cada Maude step, la fase inicial se descompone en dos normalizaciones, la primera respecto a los axiomas de Ax y la segunda respecto de las ecuaciones de Eo.
- Atributos ecuacionales, cuando aparecen operadores con axiomas asociativos, conmutativos, etc, la reescritura no es capaz de computar de forma efectiva con ellos (espacio de búsqueda infinito). Para ello, dichas propiedades se tratan aparte, como **atributos**.

pares no ordenados — >comm $XY=YX$
listas — >assoc $X(YZ) = (XY)Z$ — id:null $Xnull = nullX = X$
multiconjuntos — >assoc, comm, id:null
conjuntos — >assoc, comm, id:null, idem $XX = X$
[assoc idem] PROHIBIDO EN MAUDE

- Otros atributos ecuacionales
- [ctor] — >símbolo constructor
- [gather (e E)] — >assoc por la derecha
- [item] — >abrevia s(s(..(0))) como sn(0)
- [ditto] — >mismos axiomas que el anterior

- Ecuaciones, se usan para normalizar los estados. **Reglas,** se usan para hacer evolucionar el sistema produciendo cambios de estado.

- Estrategias de evaluación, a diferencia de otros lenguajes funcionales como Haskell, que son perezosos, Maude es un lenguaje impaciente (sólo para las ecuaciones).
Para garantizar la convegencia y terminación de los cómputos, es necesario que las ecuaciones satisfagan buenas condiciones.
- Terminación, un programa es terminante si no existen cadenas infinitas de pasos de reducción.
Cuando un programa es terminante, la forma normal de cualquier término siempre existe.
La terminación es una propiedad indecidible de los programas.
- Confluencia, un programa es confluente si, siempre que un término se puede reescribir a dos términos distintos, éstos, a su vez, convergen a un mismo término.
- Ortogonalidad — >Confluencia, cuando un programa es confluente, la forma normal de cualquier término, si existe, es única.

- La confluencia es indecidible.
- Hay condiciones suficientes. Ej: la ortogonalidad donde Ortogonalidad = no solapamiento de lh's (ninguna instancia en común) + linealidad por la izquierda (no variables repetidas)

- Reescritura condicional, las reglas/ecuaciones condicionales tienen la forma general:

- crl l = >r if C
- ceq l = >r if C

Donde C es una conjunción de expresiones del tipo: *equation, matching equation, sort expression, rule.*
- Lógica Fuzzy (borrosa, difusa) - Extensiones de la lógica, soporta el razonamiento aproximado (útil en robótica, sistemas expertos,...).

- Multivaluada, en vez de binaria, que se deriva -o es una aplicación- de la teoría de "conjuntos Fuzzy", (valores entre 0 y 1)
- Ejemplos: Lukasiewicz, Goedel, Pavelka.

- Lógica Lineal (Girald, 1987) - Extensiones de la lógica, permite actualizar la cantidad de recursos tras cada cambio de estado (útil en control de recursos)

- La implicación en lógica lineal se escribe —o y se llama "lolli"
- En dicha implicación, se consume el recurso A para producir B, las condiciones se modifican tras su uso, es decir, los recursos se consumen al usarlos.
- Nuevas conectivas lógicas: ! of course (replicación), ? why not (borrado)
- Separación en dos clases de las conectivas estándar: "multiplicativa" o simultánea, y otra "aditiva".
- La conjunción & es una conjunción ya que se puede probar simultáneamente y no es una disyunción.

- Lógica Modal - Extensiones de la lógica, nuevos cuantificadores (modales):

- Temporales: UNIVERSAL always, EXISTENCIAL sometimes para formalizar el tiempo, creencias,...
- Dinámicas: Lógicas de la acción: lógica modal para razonar acerca de las acciones y procesos.
- Epistémicas: Lógicas del Conocimiento y de la Creencia/Ignorancia.

- Lógica del Tiempo, se desarrollaron por los filósofos para investigar cómo el tiempo se usa en el lenguaje natural de las personas.
Superan la aproximación de primer orden, donde lo habitual es introducir un simple argumento extra para el tiempo.
- Aproximaciones rivales, las dos aproximaciones, clásica y temporal, son rivales.
En terminología de BDs:

- *tensers*, son partidarios de la aproximación modal temporal.
- *detenser*, partidarios de la aproximación de primer orden.

- La Trilogía del Software, consta de los siguientes componentes de software:

- **Propiedades**: Especificaciones, requerimientos, tipos, modelos (grafos, fórmulas).
- **Juegos de datos**: Seq: trazas, Out: Escenarios, In: Ejemplos
- **Programas**: Código, documentación, Proofs.

- Procesos Formales de la Trilogía:

Propiedades a Programas: Derivación formal
Programas a Propiedades: Model checking. Análisis y verficación automática del Software.
Propiedades a Propiedades: Transformación de modelos. Programación automática.
Programas a Programas: Transformación de programas. Especialización, Fragmentación, Refactorización.
Datos a Datos: Minería de datos.
Programas/Propiedades a Datos: Generación automática de casos de prueba.
Datos a Programas/Propiedades: Aprendizaje automático de programas/ propiedades a partir de Ejemplos.

- Hiperprocesos en la Trilogía

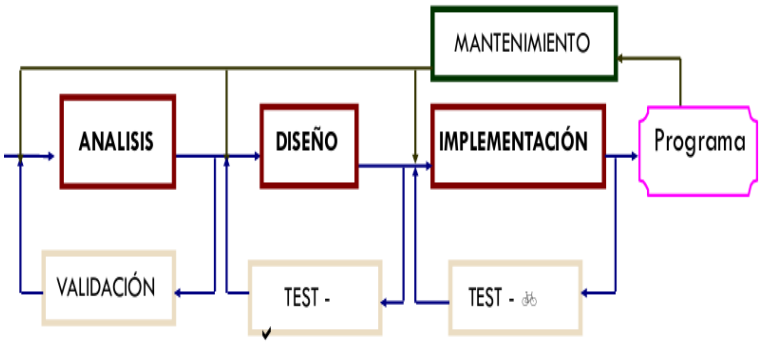
- Depuración/Diagnóstico Racional/ Declarativo
- Model Checking
- Certificación de Programas

- De Propiedades a Programas

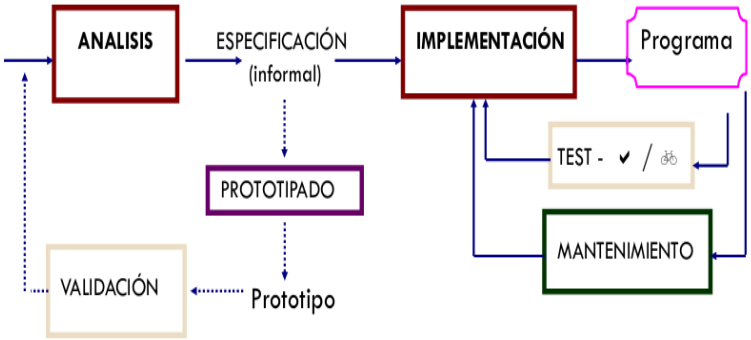
- →Derivación Formal de programas
- →Programación automática (a gran escala)

- De Propiedades a Programas (y viceversa)

- →Derivación Formal de programas
- ←Verificación Formal
- ←Type interface
- ←Síntesis de Especificaciones a partir de programas



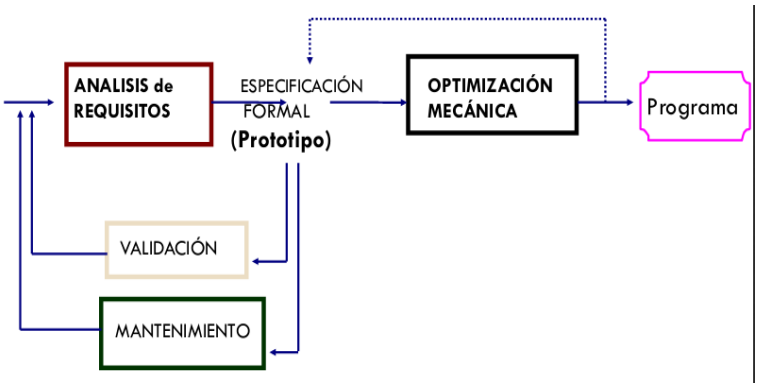
- Ciclo de Vida Clásico -



- Ciclo de Vida con Prototipado -

- De Propiedades a Propiedades

- →Transformación de Modelos
- →Programación Automática



- Programación Automática -

- De Programas a Programas, transformación de programas.

- →Compilación, Optimización...
- →Especialización, Fragmentación, Refactorización, Ofuscación,...

- Transformaciones

- Compilación de programas.
- Optimización de programas: transformaciones "fuente a fuente" para mejorar la eficiencia.
- Ofuscación: dificultar el plagio o la ingeniería inversa.
- Mantenimiento y evolución del software:

  - Fragmentación (slicing): extrae fragmentos ejecutables en base a un análisis de dependencias en el programa.

  - Refactorización: reestructura el código para mejorar atributos no funcionales como la legibilidad, complejidad, extensibilidad...

  - Derivación de programas: obtiene código eficiente a partir de especificaciones correctas.

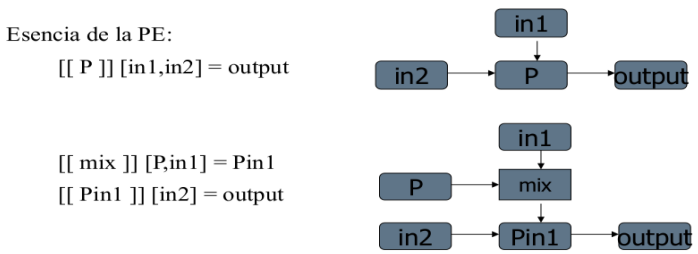
- Transformación de programas

- Especialización (partial evaluation/program specialization)
- Fragmentación (program slicing)
- Refactorización (program refactoring)

- Aplicaciones

- Especialización de programas paramétricos: Mejora hasta el 400%: Reconocimiento patrones, sistemas expertos, redes neuronales,...
- Optimización de programas (oculta en compiladores).
- Compilación y Generación de compiladores: Proyecciones de Futamura.

- Proyecciones Futamura



[[P]] [in1,in2] = [[Pin1]] [in2] = [[[[mix]] [P,in1]]] [in2]

- Futamura -

1. mix hace papel de compilador. Limitación: mix compila el lenguaje en que está escrito el propio mix.

$$\begin{aligned} \texttt{output} &= [[\ \texttt{int}\ ]]\ [\texttt{fuente,input}] \\ &= [[\ [[\ \texttt{mix}\ ]]\ [\texttt{int,fuente}]\ ]]\ [\texttt{input}] \\ &= [[\ \texttt{objeto}\ ]]\ [\texttt{input}] \end{aligned}$$

$$\texttt{objeto} = [[\ \texttt{mix}\ ]]\ [\texttt{int,fuente}]$$

2. genera compilador a través del intérprete (automáticamente). Limitación: mix debe ser autoaplicable:

$$\begin{aligned} \texttt{objeto} &= [[\ \texttt{mix}\ ]]\ [\texttt{int,fuente}] \\ &= [[\ [[\ \texttt{mix}\ ]]\ [\texttt{mix,int}]\ ]]\ [\texttt{fuente}] \\ &= [[\ \texttt{comp}\ ]]\ [\texttt{fuente}] \end{aligned}$$

$$\texttt{comp} = [[\ \texttt{mix}\ ]]\ [\texttt{mix,int}]$$

3. produce un "generador de compiladores"

$$\begin{aligned} \texttt{comp} &= [[\ \texttt{mix}\ ]]\ [\texttt{mix,int}] \\ &= [[\ [[\ \texttt{mix}\ ]]\ [\texttt{mix,mix}]\ ]]\ [\texttt{int}] \\ &= [[\ \texttt{gen\_comp}\ ]]\ [\texttt{int}] \end{aligned}$$

$$\texttt{gen\_comp} = [[\ \texttt{mix}\ ]]\ [\texttt{mix,mix}]$$

- Fragmentación (program slicing), obtención de un fragmento del programa (program slice), que contiene las instrucciones que puedan afectar al valor de una variable v en un punto o instrucción p del código.

- Suele usarse como apoyo a: Depuración y Mantenimiento y análisis del código.
- Para identificar el fragmento de interés, se usan técnicas de análisis de flujo de datos.

- Criterio de slicing, se puede generalizar como <p,v >donde:
- p es el punto de observación
- v es un conjunto de variables que se desea observar el punto "p".

- Propiedades de un slice correcto:

- Obtenido a partir del programa original aplicando ÚNICAMENTE borrado.
- El fragmento debe comportarse como el programa original en lo que respecta al criterio fijado.
- Ejemplos: Jslice / Indus - Fragmentación de programas y JRefactory - Refactorización.

- Refactorización de programas, mejorar el diseño del código existente, cambia sistematicamente la estructura del programa, sin alterar su funcionalidad.

- Contexto de aplicación:
  →mantenimiento del software: independiza los cambios funcionales de los estructurales.
  →procesos de desarrollo ágiles: se facilita la mejora continua del diseño y la adaptabilidad a cambios frente a las metodologías basadas en un diseño monolítico desde el principio.
- Pasos en un proceso de refactorización:
  →transformaciones de programas "pequeñas" e incrementales que preservan la semántica.

Metodologías Ágiles	Metodologías Tradicionales
Basadas en heurísticas provenientes de prácticas de producción de código	Basadas en normas provenientes de estándares seguidos por el entorno de desarrollo
Especialmente preparados para cambios durante el proyecto	Cierta resistencia a los cambios
Impuestas internamente (por el equipo)	Impuestas externamente
Proceso menos controlado, con pocos principios	Proceso mucho más controlado, con numerosas políticas/normas
No existe contrato tradicional o al menos es bastante flexible	Existe un contrato prefijado
El cliente es parte del equipo de desarrollo	El cliente interactúa con el equipo de desarrollo mediante reuniones
Grupos pequeños (<10 integrantes) y trabajando en el mismo sitio	Grupos grandes y posiblemente distribuidos
Pocos artefactos	Más artefactos
Pocos roles	Más roles
Menos énfasis en la arquitectura del software	La arquitectura del software es esencial y se expresa mediante modelos

- Diferencias entre metodologías ágiles y no ágiles -

- SoftWare

- Asunciones tradicionales: el código queda congelado en su forma inicial, cualquier cambio es caro y puede introducir errores.
  →requiere un análisis y diseño exhaustivo inicial.
- La refactorización cambia estas asunciones: el código se mantiene maleable durante el proceso, cambios estructurales baratos y seguros.
  →soporta análisis incremental y diseño continuo adaptativo.

- De datos a programas (y viceversa), generación Juegos de Datos, Testing Estructural (white-box)

- ←→Inferencia Inductiva (Síntesis de Programas a partir de Ejemplos).
  1.Definir caminos de prueba.
  2.Generar bancos o juegos de datos que hagan seguir cada camino (acumulando constraints y aplicando CONSTRAINT SOLVING)

- De datos a propiedades (y viceversa)

- ←Generación de Escenarios, Testing Funcional (black-box).
- →Inferencia Inductiva (Aprendizaje de Especificaciones)

- De datos a datos →Minería de datos.