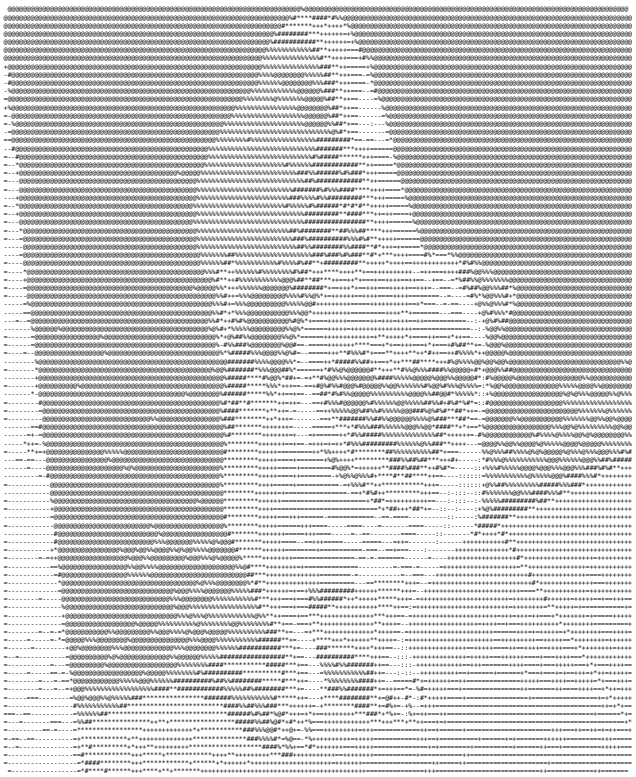
Universidad Nacional Autónoma de México
Facultad de Ciencias
Lenguajes de Programación
Benito Hernandez Ivan
Morales Flores Pablo
Perez Servin Darshan Israel
Ramirez Luna Gibran

PROYECTO 2

Sistemas de tipos y verificaciòn automàtica



Índice

1. Introducción

En 1996, el cohete Ariane 5 se autodestruyó 37 segundos después de iniciar su secuencia de vuelo debido a una excepción de software. Este fallo crítico ocurrió al intentar convertir un número de punto flotante de 64 bits a un entero con signo de 16 bits, lo que provocó un desbordamiento de datos (overflow) [1]. Este incidente ilustra cómo errores pueden escapar a las pruebas de software tradicionales y subraya la necesidad de utilizar sistemas de tipos como herramientas de verificación formal para prevenirlos.

Estos errores se pueden prevenir mediante la implementaciòn de sistemas de tipos junto con sus extensiones que permiten expresar propiedades sobre los tipos, de tal forma que se pueda plasmar de manera directa ciertas invariantes que se necesiten cumplir para el correcto funcionamiento de un programa, asì facilitando la tarea de detecciòn de errores.

En los siguientes pàrrafos se hace un anàlisis sobre como los sistemas de tipos pueden servir como primera barrera de verificaciòn en programas crìticos, se presenta un lenguaje de programaciòn reducido, que incorpora un sistema de tipos con refinamientos de forma nativa, posteriormente se ilustra un ejemplo de un programa que estàticamente impide divisiones entre cero.

2. Marco teórico

2.1 Introducción a los sistemas de tipos

Un sistema de tipos puede ser pensado como una tripleta $(T, \Gamma, \vdash)$, donde:

1. T es un conjunto de tipos,
2. Γ es un contexto de tipificado, una lista de variables asociadas a su tipo,
3. $\vdash$ es una relación de derivación de tipos. [2]

La relación de derivación se modela a traves de juicios de tipos de la forma $\Gamma \vdash t : T$ que se lee "Bajo el contexto de tipos Γ se deduce que el tipo de la expresión t es T ." [2]

En el mundo de los sistemas de tipos existen dos modelos principales:

- Tipificado Implícito: Las reglas de tipo definen la relación entre los términos sin tipo y los tipos del lenguaje que los clasifican.
- Tipificado Explícito: Los términos cargan con la información de su tipo. [3]

2.2 Tipos Dependientes

Los tipos dependientes son tipos que dependen de otros valores, por ejemplo las matrices de números enteros de longitud $n \times m$ o los vectores de longitud k. [4]

Para representarlos contamos con los siguientes constructores:

- **Π-types:** Generaliza el tipo de las funciones permitiendo que el tipo de dato que retornan cambie según el argumento.

  Ejemplo: Una función para obtener el primer elemento de un vector.

  first : Π n:Nat.Vector$(n + 1)$ ⟶ data

  En este ejemplo, el parámetro n es un valor y la longitud del vector depende directamente de ese valor; es decir, el tipo de la entrada está indexado por n. Esto provoca que la función first solo opere con vectores no vacíos.
- **Σ-types:** Generaliza el producto cartesiano, de modo que los elementos de Σ A y B son pares (a, b) donde el primer elmento es $a : A$ y el segundo depende del primero $b : B(a)$. [3]

  Ejemplo:

  FlexVec : Set → Set

  FlexVec A = Σ$[n \in N]$ Vec A n

  En este ejemplo, los elementos de **FlexVec A** son pares (n, v), donde $n : \mathbb{N}$ -n es un numero natural- y v : Vec A n, es decir, v es un vector cuya longitud depende del primer componente del par. [5]

El uso de tipos dependientes provee al lenguaje con una capa de seguridad extra al permitir, por ejemplo especificar longitudes en los tipos de entrada y/o salida.

Ejemplo usando Agda:

```
module dt where
open import Data.Nat using (ℕ; zero; suc)

data Vec (A : Set) : ℕ → Set where
  []  : Vec A zero
  _::_ : ∀ {n} (x : A) (xs : Vec A n) → Vec A (suc n)

-- head solo funciona para vectores no vacíos
head : ∀ {A m} → Vec A (suc m) → A
head (x :: xs) = x
```

[6]

En el ejemplo anterior, **Vec A zero** es el tipo asociado a los vectores vacíos. Por otro lado, la expresión **∀ {n} (x : A) (xs : Vec A n) → Vec A (suc n)** se interpreta como: "Para todo número natural **n**, el operador **cons** toma un elemento de tipo **A** y un vector de **A** de longitud **n**, y devuelve un vector de longitud **n + 1**". En cuanto a la función **head**, su firma exige que el vector que recibe tenga longitud **suc m**, es decir, al menos 1.

2.3 Tipos de Refinamiento

Los tipos de refinamiento son una herramienta para enriquecer un lenguaje a través del uso de predicados que funcionan como restricciones sobre los datos que maneja el programa. [7]

Liquid Haskell es una herramienta que tiene el propósito de fortalecer los tipos de Haskell usando una sintaxis similar a la de las listas por comprensión, permitiendo imponer precondiciones y postcondiciones a las funciones. [8]

Por ejemplo:

En Haskell podemos efectuar una división de la siguiente manera:

```
unsafeDiv :: Int -> Int -> Int
unsafeDiv n m = n `div` m
```

No obstante, si el segundo argumento que recibe la función es 0, obtenemos el siguiente error:

```
unsafeDiv 4 0
*** Exception: divide by zero
```

Con Liquid Haskell, mediante el uso de tipos de refinamiento, podemos realizar una verificación más exhaustiva de los argumentos que recibe la función de la siguiente manera:

```
{-@ safeDiv :: Int -> {v:Int | v /= 0} -> Int @-}
safeDiv :: Int -> Int -> Int
safeDiv n d = n `div` d
```

En este ejemplo utilizamos la abreviación NonZero para identificar cuándo un número entero es distinto de cero, y luego refinamos la firma de la función *divide* para impedir que el segundo argumento con el que se invoque sea cero.

2.4 Verificación Formal

Se refiere a un conjunto de métodos fundamentados en la matemática y la lógica, que tienen el propósito de identificar errores en un sistema o bien probar que dicho sistema cumple con cierta especificación. [9]

Entre los métodos de verificación más utilizados se encuentran la verificación basada en pruebas, en modelos y en tipos. Para los fines de este proyecto, nos centramos unicamente en la verificación basada en tipos.

2.4 Software crítico y sistemas de tipos

Los sistemas críticos son aquellos donde un fallo puede resultar en lesiones a personas, daño ambiental o pérdidas económicas extensas[10].

El software crítico es aquel cuyo fallo podría causar la muerte, daños graves o un impacto social negativo generalizado [11].

En este sentido el software crítico es el componente lógico que controla o forma parte del sistema crítico

Sommerville define la Confiabilidad como el grado de confianza que tiene un usuario en que el sistema funcionará como espera y que no fallará en condiciones normales de uso[10].

3. ¿Cómo puede un sistema de tipos servir como primer nivel de verificación formal de propiedades en programas críticos?

Dado que las consecuencias de un fallo en este tipo de software son catastróficas, es necesario garantizar que un software crítico cumpla con ciertas propiedades antes de ejecutarse, con la finalidad de que el sistema sea más confiable. Para esto Sommerville propone el análisis estático. A diferencia de las pruebas dinámicas, estas técnicas no requieren la ejecución del programa, sino que operan directamente sobre la representación del código fuente o el modelo de diseño, permitiendo detectar errores y anomalías estructurales antes de ejecutarlo. [10].

En primer lugar, para entender la importancia de la verificación de tipos, debemos notar que, si los programas no pasaran por esta fase de validación, durante su ejecución podrían producirse comportamientos erráticos o impredecibles, comprometiendo la fiabilidad del software y vulnerando una de las propiedades esenciales de los sistemas críticos: la garantía de funcionamiento correcto bajo todas las condiciones previstas.

De esta forma, podemos entender a un sistema de tipos como una herramienta de análisis estático. Esto es posible debido a que podemos realizar la verificación de tipos en tiempo de compilación, comprobando que las expresiones cumplan con las reglas definidas por el propio sistema de tipos. Así, se identifican inconsistencias y se verifica que el programa se adhiera a una especificación formal.

Adicionalmente, dotar al lenguaje de un sistema de tipos no solo es útil para garantizar que los programas estén correctamente tipificados; mediante el uso de conceptos más avanzados, como los tipos dependientes o de refinamiento, podemos evitar errores en tiempo de ejecución, como acceder a índices fuera de rango, y verificar la correctitud de ciertos algoritmos a través del uso de invariantes o bien especificando precondiciones y/o postcondiciones.

Por ejemplo, el incidente del Ariane 5 se podría haber evitado mediante un sistema de tipos refinados. El fallo de conversión ocurrió porque los programadores asumieron erróneamente que los valores de entrada se mantendrían dentro de un rango seguro. Bajo un enfoque de verificación estática, el sistema habría determinado los límites matemáticos de las variables, detectando que en la operación de `cast`, el dominio de la entrada excedía la capacidad del tipo de salida. Esto habría generado un error en tiempo de compilación, obligando a los programadores a ser conscientes del riesgo y a manejar el desbordamiento antes de intentar la operación.

4. Desarrollo de un Ejemplo Práctico (Mini-Cepuac)

Mini-Cepuac es un lenguaje funcional declarativo que permite realizar operaciones aritméticas básicas, así como las operaciones lógicas and, or y not. Además, incluye expresiones let, funciones lambda y aplicación de funciones, se implementa un sistema de tipos con refinamientos.

La característica más destacada del lenguaje es el uso de tipos de refinamiento de manera nativa, sin embargo los refinamientos son muy simples y por lo tanto limitados, ya que solo expresan condiciones respecto al tipo number y su relaciòn con el cero. El tener un sistema de predicados mas expresivo, complicaba significativamente la tarea de implementacion, en particular el hecho de modelar los predicados respecto a los demas componentes del lenguaje, ya que el proceso es mas extenso, esto queda ejemplificado en el uso verificador automàtico de satifasibilidad en Liquid Haskell [8].

Sistema de Tipos en Mini-Cepuac

El lenguaje tiene un estilo de tipificado explicito que podemos ver reflejado en la gramática de las expresiones let y lambdas.

```
ASA:  ...
   |'(' "lambda" ':' Arrow '(' var ')' ASA ')' { Lambda $4 $6 $8 }
   |'('"let"'('var':'Type ASA')'ASA')'{Let ($4,(wrapIfPrimitive $6))$7 $9 }
```

Como podemos notar, luego de especificar el argumento formal, la gramática exige especificar el tipo que debe tener el argumento real cuando la función sea aplicada.

Los tipos que se tienen son:

```
Type : '(' Type ')' { $2 }
    | PrimitiveType { $1 }
    | Arrow { $1 }
    | RefinementType { $1 }
```

```
PrimitiveType: "number" { Number }
    |          "boolean" { Boolean }

Arrow: Type "->" Type { Arrow ( wrapIfPrimitive $1) (wrapIfPrimitive $3) }

RefinementType: '{' var ':' Type '|' Predicate '}' { case $4 of
                                                       Arrow _ _ ->
parseError [] _ -> Refinement $4 $6 }
```

Los tipos primitivos (PrimitiveType) eventualmente son tratados como tipos refinados con su respectivo predicado, esto para tener uniformidad sobre los tipos y que la resolucion de los predicados de los tipos refinados sea mas directa.

Respecto a los predicados solo existen tres tipos y son respecto a la relacion del tipo con el cero, estos son:

- MaybeZero -> Aquellos mayores o iguales a cero, menores o iguales a cero o cero.
- NonZero -> Aquellos mayores a cero, menores a cero.
- Zero -> El numero cero.

```
Predicate: var "==" double { if $3 == 0 then Zero else parseError [] }
       | var "!=" double { if $3 == 0 then NonZero else parseError [] }
       | var '>' double { if $3 == 0 then NonZero else parseError [] }
       | var ">=" double { if $3 == 0 then MaybeZero else parseError [] }
```

Tras el análisis sintáctico, el siguiente paso es la verificación de tipos. Para ello se define la función **tc** (*Type Checker*), cuya finalidad es determinar de manera estática si una expresión está correctamente tipificada. Por ejemplo, las operaciones aritméticas son de tipo **number** únicamente si sus dos subexpresiones también lo son.

Ejemplos y uso

Un programa que mediante uso de tipos refinados impide la divisiòn entre cero. Define una funciòn *safeDiv* cuyo paràmetro es un nùmero correspondiente al numerador de la divisiòn, y regresa una funciòn cuyo cuerpo tiene la divisiòn correspondiente con el numerador ya ligado anteriormente y espera el denominador, que se exije mediante un tipo refinado que su paràmetro no sea cero

Divisiòn entre 0

```
(let (safeDiv : (number -> ({ y : number | y != 0 } -> number))
      (lambda : number -> ({ y : number | y != 0 } -> number) (x)
        (lambda : { y : number | y != 0 } -> number (y)
          (/ x y)
        )
      )
    )
  ((safeDiv 10) 0)
)
```

```
Error: Type or predicate mismatch: expected type Number with predicate
NonZero, got type Number with predicate Zero
```

Expresión no 0

```
(let (safeDiv : (number -> ({ y : number | y != 0 } -> number))
      (lambda : number -> ({ y : number | y != 0 } -> number) (x)
        (lambda : { y : number | y != 0 } -> number (y)
          (/ x y)
        )
      )
    )
  ((safeDiv 10) (+ 4 3))
)
```

```
-4.0
```

Expresiòn no 0, pero sin poder construir la prueba

```
(let (safeDiv : (number -> ({ y : number | y != 0 } -> number))
      (lambda : number -> ({ y : number | y != 0 } -> number) (x)
        (lambda : { y : number | y != 0 } -> number (y)
          (/ x y)
        )
      )
    )
  ((safeDiv 10) (- 0 3))
)
```

```
Error: Type or predicate mismatch: expected type Number with predicate
NonZero, got type Number with predicate MaybeZero
```

5. Conclusiones

A partir del desarrollo del proyecto y nuestra implementación de nuestro lenguaje Mini-Cepuac estudiando la importancia de los sistemas de tipos y sus extensiones como una herramienta fundamental para el desarrollo de sofware crítico, podemos presentar las siguientes conclusiones:

1. La verificación de tipos actua de muy buena forma para un primer nivel de verificación formal y una buena barrera de seguridad en el código escrito. Debido a la falta de verificación de tipos en el cohete Ariane 5, este lanzamiento provocó fallos catastróficos por desbordamiento de datos, por lo que la implementación de reglas estrictas puede permitir detectar inconsistencias estructurales y prevenir comportamientos erráticos sin necesidad de ejecutar el programa.

2. La implementación del verificador de tipos demostró que es posible determinar si una expresión es segura. Esto valida la teoría de que los sistemas de tipos pueden funcionar como herramientas de análisis estático, permitiendo al software cumpla con especificaciones formales, aumentando la fiabilidad del mismo.

3. Se encontró que el estilo de tipar explícitamente (especialmente en la gramática de `let` y `lambda`), facilita mucho la tarea de verificación al obligar a especificar los tipos de los argumentos.

4. Se notó el desafio en implementar expresividad de los predicados. Aunque la intención original era implementar un sistema más amplio, se demostró que esto complicaba significativamente la implementación y la relacion con los demás componentes del lenguaje. Por lo que a la hora de desarrollo se optó por limitar los refinamientos a condiciones sobre el tipo `number` y su relacion con el cero, ejemplificándose con el hecho de agregar predicados (`NonZero` , `MaybeZero` y `Zero`) para poder manejar los diferentes casos y poder detectar una posible división entre cero antes de la ejecución del programa. A la vez, que se noto la efectividad de los tipos de refinamiento a la hora de impedir errores comunes, reiterando en el ejemplo de la división entre cero al exigir que el denominador cumpla con el predicado `NonZero`, Asimismo, el sistema es capaz de detectar cuando no se puede construir una prueba suficiente (por ejemplo al intentar usar un valor `MaybeZero` donde se requiere un `NonZero`)

5. Un sistema de tipos permite garantizar que las expresiones dentro de un programa son semánticamente coherentes y respetan los dominios de valores para los cuales fueron definidas. Esta verificación temprana elimina de manera automática un conjunto amplio de errores básicos —por ejemplo, intentar sumar valores booleanos, acceder a operaciones no definidas para un tipo, o invocar funciones con argumentos incompatibles— y, en consecuencia, establece una base sólida sobre la cual es posible construir verificaciones más rigurosas.

   De esta forma, tipificar correctamente el programa se convierte en un prerrequisito crucial para cualquier proceso de verificación robusto, ya que, reduce el espacio de estados a considerar y mejora la precisión de los métodos de verificación posteriores.

Bibliografía

[1] Prof. J. L. LIONS, "ARIANE 5 Flight 501 Failure." [Online]. Available: https://www-users.cse.umn.edu/~arnold/disasters/ariane5rep.html

[2] Soto M., "Nota de Clase 31: Verificación de Tipos." 2025.

[3] B. C. Pierce, *Type Systems for Programming Languages*. MIT Press, 2002.

[4] E. Brady, *Type-Driven Development with Idris*. Manning Publications, 2017.

[5] University of Nottingham, "Dependent types." [Online]. Available: https://people.cs.nott.ac.uk/psztxa/mgs.2021/dependent.pdf

[6] Agda Team, "Coverage Checking (Agda 2.6.4.1)." [Online]. Available: https://agda.readthedocs.io/en/v2.6.4.1/language/coverage-checking.html

[7] "Refinement Types: A Tutorial." [Online]. Available: https://www.nowpublishers.com/article/Details/PGL-032

[8] R. Jhala, E. Seidel, and N. Vazou, "Programming with Refinement Types An Introduction to LiquidHaskell." [Online]. Available: https://ucsd-progsys.github.io/liquidhaskell-tutorial/

[9] MathWorks, "Formal verification." [Online]. Available: https://www.mathworks.com/discovery/formal-verification.html

[10] Ian Sommerville, *Software Engineering*, 9th ed. Pearson, 2011.

[11] IEEE Computer Society, "IEEE Standard for Software Safety Plans," IEEE, 2019. [Online]. Available: https://standards.ieee.org/ieee/1228/7744/