

# Formalización de un lenguaje imperativo con procedimientos, arreglos y apuntadores en Isabelle/HOL

Gabriela Limonta

Universidad Simón Bolívar

24/02/2016

# Motivación

## El lenguaje de programación C

# Motivación

El lenguaje de programación C

Lenguaje C:

# Motivación

## El lenguaje de programación C

### Lenguaje C:

- Cercanía a la máquina y bajo *overhead* permiten eficiencia.

# Motivación

## El lenguaje de programación C

### Lenguaje C:

- Cercanía a la máquina y bajo *overhead* permiten eficiencia.
- Utilizado para sistemas operativos, aplicaciones de sistemas embebidos, compiladores, librerías e interpretadores.

# Motivación

## El lenguaje de programación C

### Lenguaje C:

- Cercanía a la máquina y bajo *overhead* permiten eficiencia.
- Utilizado para sistemas operativos, aplicaciones de sistemas embebidos, compiladores, librerías e interpretadores.

### Desventaja?

# Motivación

## El lenguaje de programación C

### Lenguaje C:

- Cercanía a la máquina y bajo *overhead* permiten eficiencia.
- Utilizado para sistemas operativos, aplicaciones de sistemas embebidos, compiladores, librerías e interpretadores.

### Desventaja?

- Parte de la semántica se define en lenguaje natural, lo cual la hace vulnerable a ambigüedades.

# Motivación

## Objetivos del trabajo



# Motivación

## Objetivos del trabajo

- Formalizar la semántica operacional de pasos cortos de un lenguaje imperativo que represente un subconjunto especializado y determinístico de la semántica de C.

# Motivación

## Objetivos del trabajo

- Formalizar la semántica operacional de pasos cortos de un lenguaje imperativo que represente un subconjunto especializado y determinístico de la semántica de C.
- Escribir un interpretador dentro del ambiente de Isabelle/HOL y demostrar su correctitud.

# Motivación

## Objetivos del trabajo

- Formalizar la semántica operacional de pasos cortos de un lenguaje imperativo que represente un subconjunto especializado y determinístico de la semántica de C.
- Escribir un interpretador dentro del ambiente de Isabelle/HOL y demostrar su correctitud.
- Generar código C a partir de programas escritos en la semántica formal.

# Motivación

## Objetivos del trabajo

- Formalizar la semántica operacional de pasos cortos de un lenguaje imperativo que represente un subconjunto especializado y determinístico de la semántica de C.
- Escribir un interpretador dentro del ambiente de Isabelle/HOL y demostrar su correctitud.
- Generar código C a partir de programas escritos en la semántica formal.
- Crear un ambiente de pruebas y una batería de pruebas que incrementen la confianza en el proceso de generación de código.

# Chloe

Un lenguaje imperativo, subconjunto de C

Características actuales:

# Chloe

Un lenguaje imperativo, subconjunto de C

## Características actuales:

- Variables

# Chloe

Un lenguaje imperativo, subconjunto de C

## Características actuales:

- Variables
- Arreglos

# Chloe

Un lenguaje imperativo, subconjunto de C

## Características actuales:

- Variables
- Arreglos
- Aritmética de apuntadores



# Chloe

Un lenguaje imperativo, subconjunto de C

## Características actuales:

- Variables
- Arreglos
- Aritmética de apuntadores
- Ciclos

# Chloe

Un lenguaje imperativo, subconjunto de C

## Características actuales:

- Variables
- Arreglos
- Aritmética de apuntadores
- Ciclos
- Condicionales

# Chloe

Un lenguaje imperativo, subconjunto de C

## Características actuales:

- Variables
- Arreglos
- Aritmética de apuntadores
- Ciclos
- Condicionales
- Funciones

# Chloe

Un lenguaje imperativo, subconjunto de C

## Características actuales:

- Variables
- Arreglos
- Aritmética de apuntadores
- Ciclos
- Condicionales
- Funciones
- Memoria dinámica

# Chloe

Un lenguaje imperativo, subconjunto de C

## Características actuales:

- Variables
- Arreglos
- Aritmética de apuntadores
- Ciclos
- Condicionales
- Funciones
- Memoria dinámica

## Extensiones a futuro:

# Chloe

Un lenguaje imperativo, subconjunto de C

## Características actuales:

- Variables
- Arreglos
- Aritmética de apuntadores
- Ciclos
- Condicionales
- Funciones
- Memoria dinámica

## Extensiones a futuro:

- Sistema de tipos estático correcto y completo

# Chloe

Un lenguaje imperativo, subconjunto de C

## Características actuales:

- Variables
- Arreglos
- Aritmética de apuntadores
- Ciclos
- Condicionales
- Funciones
- Memoria dinámica

## Extensiones a futuro:

- Sistema de tipos estático correcto y completo
- Concurrencia

# Chloe

Un lenguaje imperativo, subconjunto de C

## Características actuales:

- Variables
- Arreglos
- Aritmética de apuntadores
- Ciclos
- Condicionales
- Funciones
- Memoria dinámica

## Extensiones a futuro:

- Sistema de tipos estático correcto y completo
- Concurrencia
- Operaciones I/O



# Chloe

Un lenguaje imperativo, subconjunto de C

## Características actuales:

- Variables
- Arreglos
- Aritmética de apuntadores
- Ciclos
- Condicionales
- Funciones
- Memoria dinámica

## Extensiones a futuro:

- Sistema de tipos estático correcto y completo
- Concurrencia
- Operaciones I/O
- Goto

# Chloe

Un lenguaje imperativo, subconjunto de C

## Características actuales:

- Variables
- Arreglos
- Aritmética de apuntadores
- Ciclos
- Condicionales
- Funciones
- Memoria dinámica

## Extensiones a futuro:

- Sistema de tipos estático correcto y completo
- Concurrencia
- Operaciones I/O
- Goto
- Etiquetas

# Chloe

Un lenguaje imperativo, subconjunto de C

## Características actuales:

- Variables
- Arreglos
- Aritmética de apuntadores
- Ciclos
- Condicionales
- Funciones
- Memoria dinámica

## Extensiones a futuro:

- Sistema de tipos estático correcto y completo
- Concurrencia
- Operaciones I/O
- Goto
- Etiquetas
- Instrucciones break y continue

# Chloe

Un lenguaje imperativo, subconjunto de C

## Características actuales:

- Variables
- Arreglos
- Aritmética de apuntadores
- Ciclos
- Condicionales
- Funciones
- Memoria dinámica

## Extensiones a futuro:

- Sistema de tipos estático correcto y completo
- Concurrencia
- Operaciones I/O
- Goto
- Etiquetas
- Instrucciones break y continue

# Semántica

Existen tres enfoques principales:

Existen tres enfoques principales:

- Operacional

Existen tres enfoques principales:

- Operacional
  - ▶ Pasos largos
  - ▶ Pasos cortos



Existen tres enfoques principales:

- Operacional
  - ▶ Pasos largos
  - ▶ Pasos cortos
- Denotacional

Existen tres enfoques principales:

- Operacional
  - ▶ Pasos largos
  - ▶ Pasos cortos
- Denotacional
- Axiomática

Existen tres enfoques principales:

- Operacional
  - ▶ Pasos largos
  - ▶ Pasos cortos
- Denotacional
- Axiomática



- Isabelle/HOL es un demostrador interactivo de teoremas escrito en ML.



- Isabelle/HOL es un demostrador interactivo de teoremas escrito en ML.
- Desarrollado por Larry Paulson y Tobias Nipkow.



- Isabelle/HOL es un demostrador interactivo de teoremas escrito en ML.
- Desarrollado por Larry Paulson y Tobias Nipkow.
- Utiliza el lenguaje HOL para realizar las pruebas.



- Isabelle/HOL es un demostrador interactivo de teoremas escrito en ML.
- Desarrollado por Larry Paulson y Tobias Nipkow.
- Utiliza el lenguaje HOL para realizar las pruebas.
- Permite hacer definiciones y demostrar propiedades acerca de las mismas.



- Isabelle/HOL es un demostrador interactivo de teoremas escrito en ML.
- Desarrollado por Larry Paulson y Tobias Nipkow.
- Utiliza el lenguaje HOL para realizar las pruebas.
- Permite hacer definiciones y demostrar propiedades acerca de las mismas.
- Se usa la máquina para asistir en las demostraciones





- Isabelle/HOL es un demostrador interactivo de teoremas escrito en ML.
- Desarrollado por Larry Paulson y Tobias Nipkow.
- Utiliza el lenguaje HOL para realizar las pruebas.
- Permite hacer definiciones y demostrar propiedades acerca de las mismas.
- Se usa la máquina para asistir en las demostraciones

# Ejemplo de una prueba en Isabelle/HOL

```
datatype nat = 0 | Suc nat
```

```
fun add :: ‘‘nat => nat => nat’’ where
```

```
Base: ‘‘add 0 n = n’’ |
```

```
Rec: ‘‘add (Suc m) n = Suc (add m n)’’
```

# Ejemplo de una prueba en Isabelle/HOL

```
lemma add_right:  
  "add m 0 = m"  
proof(induction)
```

## Output de Isabelle

goal (2 subgoals):

1.  $\text{add } 0 \ 0 = 0$

2.  $\bigwedge x. \text{add } x \ 0 = x \implies$   
 $\text{add } (\text{Suc } x) \ 0 = \text{Suc } x$

# Ejemplo de una prueba en Isabelle/HOL

```
lemma add_right:  
  ‘‘add m 0 = m’’  
proof(induction)  
  show ‘‘add 0 0 = 0’’
```

## Output de Isabelle

```
goal (1 subgoal):  
1. add 0 0 = 0
```

# Ejemplo de una prueba en Isabelle/HOL

```
lemma add_right:  
  ‘‘add m 0 = m’’  
proof(induction)  
  show ‘‘add 0 0 = 0’’  
using Base
```

## Output de Isabelle

```
using this:  
add 0 ?n = ?n  
  
goal (1 subgoal):  
1. add 0 0 = 0
```

# Ejemplo de una prueba en Isabelle/HOL

```
lemma add_right:  
  ''add m 0 = m''  
proof(induction)  
  show ''add 0 0 = 0''  
using Base  
by simp
```

## Output de Isabelle

```
goal (1 subgoal):  
1.  $\bigwedge x. \text{add } x \ 0 = x \implies$   
    $\text{add } (\text{Suc } x) \ 0 = \text{Suc } x$ 
```

# Ejemplo de una prueba en Isabelle/HOL

```
lemma add_right:  
  ‘‘add m 0 = m’’  
proof(induction)  
  show ‘‘add 0 0 = 0’’  
  using Base  
  by simp
```

```
fix x  
assume IH: ‘‘add m 0 = m’’
```

## Output de Isabelle

```
this:  
add x 0 = x  
goal (1 subgoal):  
1.  $\bigwedge x. \text{add } x \ 0 = x \implies$   
   add (Suc x) 0 = Suc x
```

# Ejemplo de una prueba en Isabelle/HOL

```
lemma add_right:
  ‘‘add m 0 = m’’
proof(induction)
show ‘‘add 0 0 = 0’’
using Base
by simp

fix x
assume IH: ‘‘add m 0 = m’’
show ‘‘add (Suc m) 0 = Suc
m’’
```

## Output de Isabelle

```
goal (1 subgoal):
1. add (Suc x) 0 = Suc x
```



# Ejemplo de una prueba en Isabelle/HOL

```
lemma add_right:
  ‘‘add m 0 = m’’
proof(induction)
  show ‘‘add 0 0 = 0’’
  using Base
  by simp

  fix x
  assume IH: ‘‘add m 0 = m’’
  show ‘‘add (Suc m) 0 = Suc
  m’’
  using Rec and IH
```

## Output de Isabelle

```
using this:
add (Suc ?m) ?n = Suc (add
?m ?n)
add x 0 = x

goal (1 subgoal):
1. add (Suc x) 0 = Suc x
```

# Ejemplo de una prueba en Isabelle/HOL

```
lemma add_right:
  ‘‘add m 0 = m’’
proof(induction)
show ‘‘add 0 0 = 0’’
using Base
by simp

fix x
assume IH: ‘‘add m 0 = m’’
show ‘‘add (Suc m) 0 = Suc
m’’
using Rec and IH
by simp
qed
```

## Output de Isabelle

```
goal:
No subgoals!
```

# Ejemplo de una prueba en Isabelle/HOL

```
lemma add_right:  
  ‘‘add m 0 = m’’
```

```
apply induction  
apply auto  
done
```

## Output de Isabelle

```
goal:  
No subgoals!
```

# Antecedentes

Michael Norrish: Semántica de C formalizada en HOL

- Semántica operacional de pasos largos para el subconjunto Cholera de C en HOL.
- Considera todo posible orden de evaluación para efectos de borde.
- Deriva una lógica de Hoare para programas en C.
- Sistema para analizar cuerpos de ciclos y generar postcondiciones correctas.

# Antecedentes

## Proyecto CompCert

- Compilador moderadamente optimizador
- Traduce código del subconjunto Clight de C a código ensamblador para PowerPC.
- Modelo de memoria interesante para lenguajes imperativos.
- Formalización y demostración de propiedades en operaciones de memoria usando Coq.

# Antecedentes

## Proyecto AutoCorres

- Enfoque *bottom-up*
- Reconoce lexicográficamente código C y genera una representación monádica de más alto nivel.
- Esta representación generada facilita el razonamiento sobre programas.
- Se utiliza en varios proyectos de verificación de C:
  - ▶ Verificación de una librería de grafos a gran escala
  - ▶ Verificación de un sistema de archivos
  - ▶ Verificación de un sistema operativo de tiempo real

# ¿Qué es un programa?

Un programa es una secuencia de una o mas instrucciones que realizan una tarea en una computadora:

## Example

```
x = 21;  
y = 21;  
if (x == y) {  
    z = x + y;  
}
```

# Sintaxis

## Expresiones en Chloe

```
type_synonym vname = string

datatype exp = Const int
  | Null
  | V      vname
  | Plus  exp exp
  | Subst exp exp
  | Minus exp
  | Div   exp exp
  | Mod   exp exp
  | Mult  exp exp
  | Less  exp exp
  | Not   exp
  | And   exp exp
  | Or    exp exp
  | Eq    exp exp
  | New   exp
  | Deref exp
  | Ref   lexp
  | Index exp exp
and
datatype lexp = Deref exp
  | Index1 exp exp
```



# Sintaxis

## Expresiones en Chloe

¿Por qué diferenciamos entre *exp* y *lexp*?

### Example

```
foo = *bar;  
*baz = 1;
```

# Tipos

## Enteros y apuntadores

Los enteros son palabras de precisión 64.

```
type_synonym int_width = 64
type_synonym int_val = int_width word
```

Y tienen los siguientes límites:

```
INT_MIN == - (2^(int_width - 1))
INT_MAX == ((2^(int_width - 1)) - 1)
```

Un apuntador es un par que representa un identificador de un bloque y un desplazamiento.

# Memoria dinámica

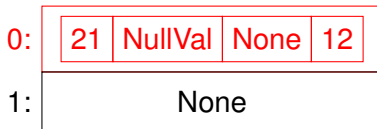
## Diseño del *heap*

0:	21	NullVal	None	12
1:	None			

- Bloque asignado
- Bloque Libre
- Celdas inicializadas
- Celda no inicializada

# Memoria dinámica

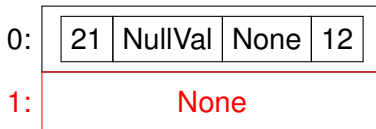
## Diseño del *heap*



- Bloque asignado
- Bloque Libre
- Celdas inicializadas
- Celda no inicializada

# Memoria dinámica

## Diseño del *heap*



- Bloque asignado
- **Bloque Libre**
- Celdas inicializadas
- Celda no inicializada

# Memoria dinámica

## Diseño del *heap*

0:	21	NullVal	None	12
1:	None			

- Bloque asignado
- Bloque Libre
- Celdas inicializadas
- Celda no inicializada

# Memoria dinámica

## Diseño del *heap*

0:	21	NullVal	None	12
1:	None			

- Bloque asignado
- Bloque Libre
- Celdas inicializadas
- Celda no inicializada

# Problema de la memoria ilimitada

## Soluciones posibles

La asignación de memoria no puede fallar porque se supone una cantidad ilimitada de memoria.

Sin embargo, se tienen recursos limitados en una computadora.  
Hay dos soluciones posibles:

- Modelar un comportamiento no-determinístico en la función que asigna memoria.
- Suponer una cantidad fija de memoria.



# Problema de la memoria ilimitada

## Nuestra solución

Se mantiene la suposición de memoria ilimitada.

Luego en el proceso de traducción se envuelve la función malloc de C en una definida por nosotros.

Esta función verifica si la llamada a malloc falla.

Si falla, entonces el programa se aborta.

# Semántica de las expresiones

¿Cuál es el significado de una expresión?

La semántica de una expresión es su valor y efecto en el estado del programa.

# Semántica de las expresiones

¿Cuál es el significado de una expresión?

La semántica de una expresión es su valor y efecto en el estado del programa.

## Example

$21 + 21 = 42$

# Semántica de las expresiones

¿Cuál es el significado de una expresión?

La semántica de una expresión es su valor y efecto en el estado del programa.

## Example

$21 + 21 = 42$

$foo + 42 = ?$

# Semántica de las expresiones

¿Cuál es el significado de una expresión?

La semántica de una expresión es su valor y efecto en el estado del programa.

## Example

$21 + 21 = 42$

$foo + 42 = ?$

$bar = new(12)$

# Semántica de las expresiones

¿Cuál es el significado de una expresión?

La semántica de una expresión es su valor y efecto en el estado del programa.

## Example

$21 + 21 = 42$

$foo + 42 = ?$

$bar = new(12)$

Se debe saber el valor de una variable a tiempo de ejecución.

# Semántica de las expresiones

## Valuaciones

Una valuación es una función que mapea un nombre de variable a un valor.

# Semántica de las expresiones

## Valuaciones

Una valuación es una función que mapea un nombre de variable a un valor.

**type\_synonym** valuation = vname  $\Rightarrow$  val option option



# Semántica de las expresiones

## Valuaciones

Una valuación es una función que mapea un nombre de variable a un valor.

**type\_synonym** valuation = vname  $\Rightarrow$  val option option

Dado un nombre de variable puede producir tres resultados diferentes:

# Semántica de las expresiones

## Valuaciones

Una valuación es una función que mapea un nombre de variable a un valor.

**type\_synonym** valuation = vname  $\Rightarrow$  val option option

Dado un nombre de variable puede producir tres resultados diferentes:

- None: valor sin definir.

# Semántica de las expresiones

## Valuaciones

Una valuación es una función que mapea un nombre de variable a un valor.

**type\_synonym** valuation = vname  $\Rightarrow$  val option option

Dado un nombre de variable puede producir tres resultados diferentes:

- None: valor sin definir.
- Some None: valor sin inicializar.

# Semántica de las expresiones

## Valuaciones

Una valuación es una función que mapea un nombre de variable a un valor.

**type\_synonym** valuation = vname  $\Rightarrow$  val option option

Dado un nombre de variable puede producir tres resultados diferentes:

- None: valor sin definir.
- Some None: valor sin inicializar.
- Some (Some  $v$ ): variable inicializada que tiene valor  $v$ .

# Semántica de las expresiones

## Estado visible

El estado de un programa esta conformado por:

- Pila de ejecución
- Variables globales
- Memoria

# Semántica de las expresiones

## Estado visible

El estado de un programa esta conformado por:

- Pila de ejecución
- Variables globales
- Memoria

Cuando se ejecuta una instrucción, la misma solo puede *ver* cierta parte del estado:

# Semántica de las expresiones

## Estado visible

El estado de un programa esta conformado por:

- Pila de ejecución
- Variables globales
- Memoria

Cuando se ejecuta una instrucción, la misma solo puede *ver* cierta parte del estado:

- Variables locales a la función actual

# Semántica de las expresiones

## Estado visible

El estado de un programa esta conformado por:

- Pila de ejecución
- Variables globales
- Memoria

Cuando se ejecuta una instrucción, la misma solo puede *ver* cierta parte del estado:

- Variables locales a la función actual
- Variables globales



# Semántica de las expresiones

## Estado visible

El estado de un programa esta conformado por:

- Pila de ejecución
- Variables globales
- Memoria

Cuando se ejecuta una instrucción, la misma solo puede *ver* cierta parte del estado:

- Variables locales a la función actual
- Variables globales
- Memoria

# Semántica de las expresiones

Funciones `eval` y `evalL`

Se definen dos funciones para calcular el valor de una expresión.

La evaluación de estas funciones puede fallar:

# Semántica de las expresiones

Funciones `eval` y `evalL`

Se definen dos funciones para calcular el valor de una expresión.

La evaluación de estas funciones puede fallar:

- Variables sin definir

# Semántica de las expresiones

Funciones `eval` y `eval_1`

Se definen dos funciones para calcular el valor de una expresión.

La evaluación de estas funciones puede fallar:

- Variables sin definir
- Operandos ilegales dados a la función

# Semántica de las expresiones

Funciones `eval` y `eval_1`

Se definen dos funciones para calcular el valor de una expresión.

La evaluación de estas funciones puede fallar:

- Variables sin definir
- Operandos ilegales dados a la función
- Acceso a memoria inválida

# Semántica de las expresiones

Funciones `eval` y `evalL`

Se definen dos funciones para calcular el valor de una expresión.

La evaluación de estas funciones puede fallar:

- Variables sin definir
- Operandos ilegales dados a la función
- Acceso a memoria inválida
- *Overflow* de enteros

# Semántica de las expresiones

Funciones `eval` y `evalL`

Se definen dos funciones para calcular el valor de una expresión.

La evaluación de estas funciones puede fallar:

- Variables sin definir
- Operandos ilegales dados a la función
- Acceso a memoria inválida
- *Overflow* de enteros
- División por cero

# Semántica de las expresiones

## Aspectos destacados de la evaluación de expresiones

- Se detecta *overflow* de enteros y lleva a un estado erróneo
- Evaluación de corto circuito
- División y módulo con truncamiento a cero
- Alcance estático de las variables



# Sintaxis de las instrucciones

## Sintaxis Abstracta

```
datatype com = SKIP
  | Assignl lexp exp
  | Assign  vname exp
  | Seq      com  com
  | If       exp com com
  | While    exp com
  | Free     lexp
  | Return  exp
  | Returnv
  | Callfunl lexp fname " exp list"
  | Callfun  vname fname " exp list"
  | Callfunv fname " exp list"
```

# Sintaxis de las instrucciones

## Llamadas a funciones

Las funciones no pueden ser expresiones.

# Sintaxis de las instrucciones

## Llamadas a funciones

Las funciones no pueden ser expresiones.

Al llamar a una función hay tres opciones con respecto al valor de retorno:

# Sintaxis de las instrucciones

## Llamadas a funciones

Las funciones no pueden ser expresiones.

Al llamar a una función hay tres opciones con respecto al valor de retorno:

### Asignación a celda en memoria

```
Callfun1 lexp fname "exp list"
```

# Sintaxis de las instrucciones

## Llamadas a funciones

Las funciones no pueden ser expresiones.

Al llamar a una función hay tres opciones con respecto al valor de retorno:

### Asignación a celda en memoria

```
Callfunl lexp fname "exp list"
```

### Asignación a una variable

```
Callfun vname fname "exp list"
```

# Sintaxis de las instrucciones

## Llamadas a funciones

Las funciones no pueden ser expresiones.

Al llamar a una función hay tres opciones con respecto al valor de retorno:

### Asignación a celda en memoria

```
Callfunl lexp fname "exp list"
```

### Asignación a una variable

```
Callfun vname fname "exp list"
```

### Asignación sin valor de retorno

```
Callfunv fname "exp list"
```

# Funciones

Se tienen funciones que retornan un valor y aquellas que no.

```
record fun_decl =  
  name :: fname  
  params :: vname list  
  locals :: vname list  
  body :: com
```

Una función está bien formada  $\iff$  los parámetros y variables locales tienen nombres diferentes.

Cuando una función retorna un valor, este valor es:

- Asignado a una ubicación en memoria
- Asignado a una variable
- Ignorado

# Programas

```
record program =  
  name :: fname  
  globals :: vname list  
  procs :: fun_decl list
```



# Programas

```
record program =  
  name :: fname  
  globals :: vname list  
  procs :: fun_decl list
```

Un programa se considera bien formado si:

- Las variables globales tienen nombres diferentes entre sí.

# Programas

```
record program =  
  name :: fname  
  globals :: vname list  
  procs :: fun_decl list
```

Un programa se considera bien formado si:

- Las variables globales tienen nombres diferentes entre sí.
- Los nombres de funciones son diferentes entre sí.

# Programas

```
record program =  
  name :: fname  
  globals :: vname list  
  procs :: fun_decl list
```

Un programa se considera bien formado si:

- Las variables globales tienen nombres diferentes entre sí.
- Los nombres de funciones son diferentes entre sí.
- Los nombres de variables globales y funciones son diferentes entre sí.

# Programas

```
record program =  
  name :: fname  
  globals :: vname list  
  procs :: fun_decl list
```

Un programa se considera bien formado si:

- Las variables globales tienen nombres diferentes entre sí.
- Los nombres de funciones son diferentes entre sí.
- Los nombres de variables globales y funciones son diferentes entre sí.
- Todas las declaraciones de funciones están bien formadas.

# Programas

```
record program =  
  name :: fname  
  globals :: vname list  
  procs :: fun_decl list
```

## Un programa se considera bien formado si:

- Las variables globales tienen nombres diferentes entre sí.
- Los nombres de funciones son diferentes entre sí.
- Los nombres de variables globales y funciones son diferentes entre sí.
- Todas las declaraciones de funciones están bien formadas.
- La función main está definida.

# Programas

```
record program =  
  name :: fname  
  globals :: vname list  
  procs :: fun_decl list
```

## Un programa se considera bien formado si:

- Las variables globales tienen nombres diferentes entre sí.
- Los nombres de funciones son diferentes entre sí.
- Los nombres de variables globales y funciones son diferentes entre sí.
- Todas las declaraciones de funciones están bien formadas.
- La función main está definida.
- Ninguno de los nombres de variables o funciones en el programa es una palabra reservada.

# Pila de ejecución

Ubicaciones de retorno y marcos de pila

**datatype** `stack_frame` = `com` × `valuation` × `return_loc`

La pila de ejecución es una lista de marcos de pila.

# Pila de ejecución

## Ubicaciones de retorno y marcos de pila

**datatype** `stack_frame` = `com` × `valuation` × `return_loc`

La pila de ejecución es una lista de marcos de pila.

**datatype** `return_loc` = `Ar addr` | `Vr vname` | `Invalid`

Un valor de retorno de una función puede ser:



# Pila de ejecución

## Ubicaciones de retorno y marcos de pila

**datatype** `stack_frame` = `com` × `valuation` × `return_loc`

La pila de ejecución es una lista de marcos de pila.

`datatype return_loc = Ar addr | Vr vname | Invalid`

Un valor de retorno de una función puede ser:

- **Asignado a una ubicación en memoria**

# Pila de ejecución

## Ubicaciones de retorno y marcos de pila

**datatype** `stack_frame` = `com` × `valuation` × `return_loc`

La pila de ejecución es una lista de marcos de pila.

`datatype return_loc = Ar addr | Vr vname | Invalid`

Un valor de retorno de una función puede ser:

- Asignado a una ubicación en memoria
- **Asignado a una variable**

# Pila de ejecución

## Ubicaciones de retorno y marcos de pila

**datatype** `stack_frame` = `com` × `valuation` × `return_loc`

La pila de ejecución es una lista de marcos de pila.

**datatype** `return_loc` = `Ar addr` | `Vr vname` | **Invalid**

Un valor de retorno de una función puede ser:

- Asignado a una ubicación en memoria
- Asignado a una variable
- **Ignorado**

# Pila de ejecución

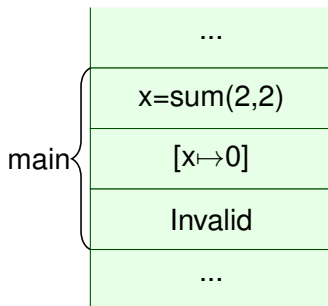
## Convención de llamada

```
int sum(int a; int b)
    return a+b;
```

```
int main()
    int x = 0;
    x = sum(2,2);
```

# Pila de ejecución

## Convención de llamada



# Pila de ejecución

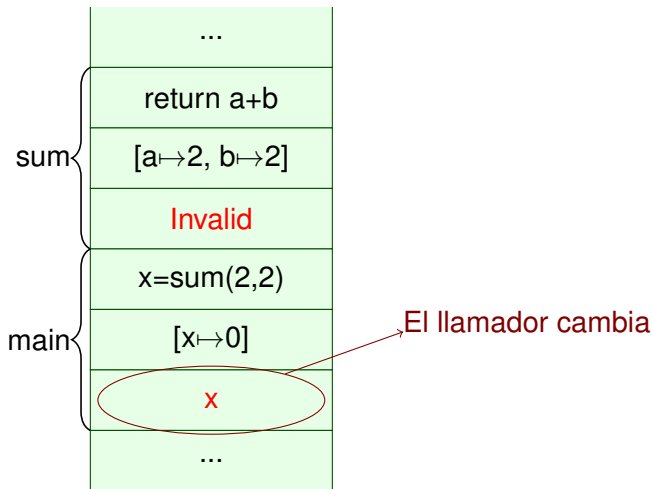
## Convención de llamada

```
int sum(int a; int b)
    return a+b;
```

```
int main()
    int x = 0;
    x = sum(2,2);
```

# Pila de ejecución

## Convención de llamada



# Tabla de procedimientos y estados

La tabla de procedimientos mapea nombres de funciones a sus declaraciones:

**type\_synonym** `proc_table` = `fname`  $\Rightarrow$  `fun_decl` option



# Tabla de procedimientos y estados

La tabla de procedimientos mapea nombres de funciones a sus declaraciones:

**type\_synonym** `proc_table` = `fname`  $\Rightarrow$  `fun_decl` option

Se construye juntando cada declaración de función con su nombre.

# Tabla de procedimientos y estados

La tabla de procedimientos mapea nombres de funciones a sus declaraciones:

**type\_synonym** `proc_table` = `fname`  $\Rightarrow$  `fun_decl` option

Se construye juntando cada declaración de función con su nombre.

Un estado se define como:

# Tabla de procedimientos y estados

La tabla de procedimientos mapea nombres de funciones a sus declaraciones:

**type\_synonym**  $\text{proc\_table} = \text{fname} \Rightarrow \text{fun\_decl option}$

Se construye juntando cada declaración de función con su nombre.

Un estado se define como:

**type\_synonym**  $\text{state} = \text{stack\_frame list} \times \text{valuation} \times \text{mem}$

# Tabla de procedimientos y estados

La tabla de procedimientos mapea nombres de funciones a sus declaraciones:

**type\_synonym**  $\text{proc\_table} = \text{fname} \Rightarrow \text{fun\_decl option}$

Se construye juntando cada declaración de función con su nombre.

Un estado se define como:

**type\_synonym**  $\text{state} = \text{stack\_frame list} \times \text{valuation} \times \text{mem}$

- Pila de ejecución
- Variables globales
- Memoria

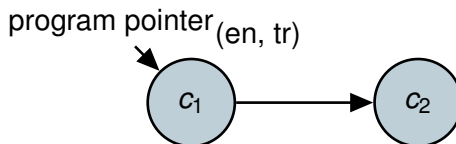
Un estado inicial es la tupla que contiene:

- Pila inicial: Pila de ejecución con el marco de pila de la función main
- Variables globales iniciales: todos los nombres de variables se mapean a indefinido y las variables globales a no-inicializado
- Memoria inicial: Memoria vacía

# Grafo de flujo de control

¿Qué es un CFG?

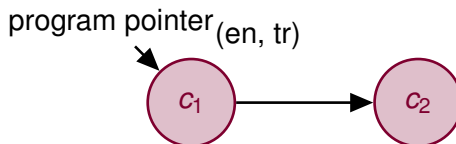
Una representación en forma de grafo que cubre los diferentes caminos que un programa puede tomar durante su ejecución.



# Grafo de flujo de control

¿Qué es un CFG?

Una representación en forma de grafo que cubre los diferentes caminos que un programa puede tomar durante su ejecución.

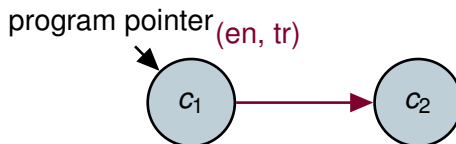


- Los nodos del CFG son instrucciones

# Grafo de flujo de control

¿Qué es un CFG?

Una representación en forma de grafo que cubre los diferentes caminos que un programa puede tomar durante su ejecución.



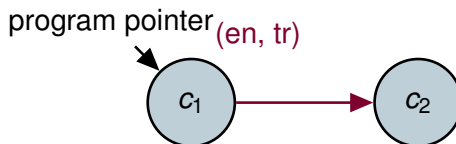
- Los nodos del CFG son instrucciones
- Las aristas del CFG están anotadas con dos funciones que dependen del estado del programa



# Grafo de flujo de control

¿Qué es un CFG?

Una representación en forma de grafo que cubre los diferentes caminos que un programa puede tomar durante su ejecución.

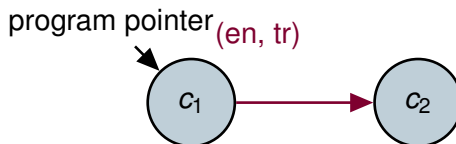


- Los nodos del CFG son instrucciones
- Las aristas del CFG están anotadas con dos funciones que dependen del estado del programa
  - ▶ La primera indica si una arista puede ser seguida o no

# Grafo de flujo de control

¿Qué es un CFG?

Una representación en forma de grafo que cubre los diferentes caminos que un programa puede tomar durante su ejecución.

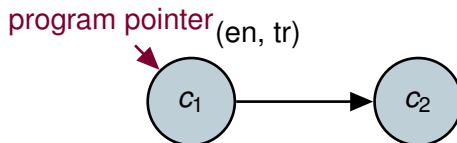


- Los nodos del CFG son instrucciones
- Las aristas del CFG están anotadas con dos funciones que dependen del estado del programa
  - ▶ La primera indica si una arista puede ser seguida o no
  - ▶ La segunda indica como se transforma el estado al seguir una arista

# Grafo de flujo de control

¿Qué es un CFG?

Una representación en forma de grafo que cubre los diferentes caminos que un programa puede tomar durante su ejecución.

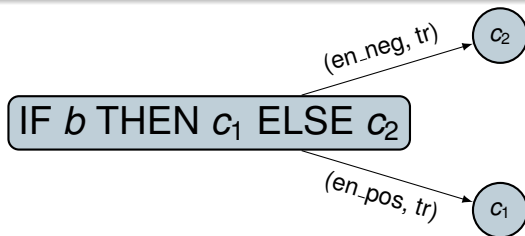


- Los nodos del CFG son instrucciones
- Las aristas del CFG están anotadas con dos funciones que dependen del estado del programa
  - ▶ La primera indica si una arista puede ser seguida o no
  - ▶ La segunda indica como se transforma el estado al seguir una arista
- Se tiene una ubicación actual, la cual es un *program pointer* al nodo actual

# Grafo de flujo de control

Funciones *enabled*

**type\_synonym** enabled = state  $\Rightarrow$  bool option

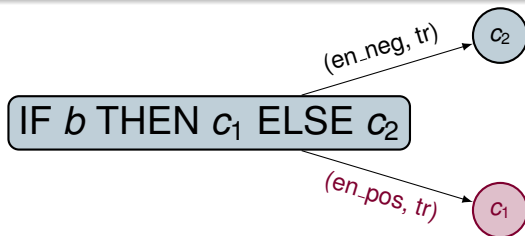


Esta función indica si un estado puede continuar su ejecución.  
Es una función parcial, por lo que puede fallar.

# Grafo de flujo de control

Funciones *enabled*

**type\_synonym** enabled = state  $\Rightarrow$  bool option

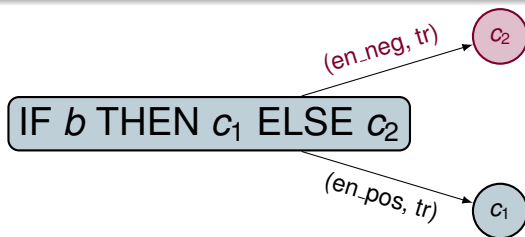


Cuando la evaluación de la condición  $b$  produce un valor true, se sigue la arista que lleva a  $c_1$ .

# Grafo de flujo de control

Funciones *enabled*

**type\_synonym** enabled = state  $\Rightarrow$  bool option

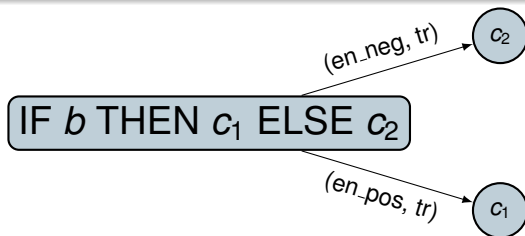


Cuando la evaluación de la condición  $b$  produce un valor false, se sigue la arista que lleva a  $c_2$ .

# Grafo de flujo de control

Funciones *enabled*

**type\_synonym** enabled = state  $\Rightarrow$  bool option

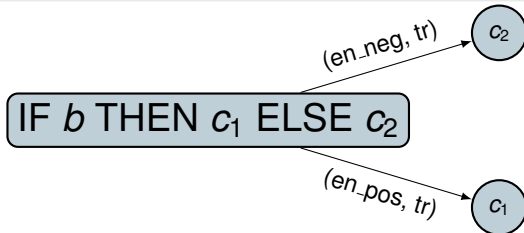


Dependiendo del resultado de la función *enabled* se decide que arista tomar.

# Grafo de flujo de control

Funciones *enabled*

**type\_synonym** enabled = state  $\Rightarrow$  bool option



Siempre habrá una arista que pueda ser seguida.



# Grafo de flujo de control

Funciones *transformer*

**type\_synonym** transformer = state  $\Rightarrow$  state option

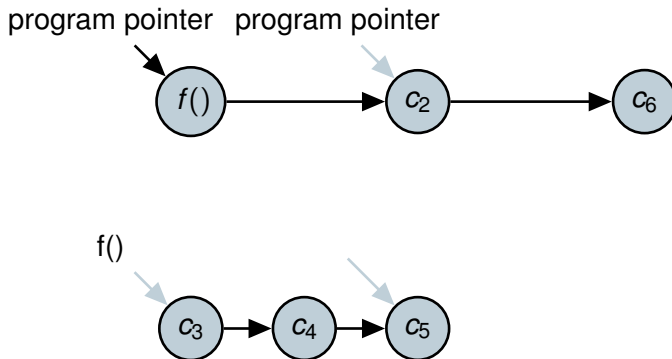
Esta función transforma un estado en uno nuevo dependiendo de la arista que se siga.

Es una función parcial, falla cuando se encuentra un error en la transformación.

Esta función actualiza el estado con el estado resultante cada vez que se sigue una arista.

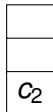
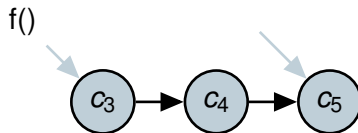
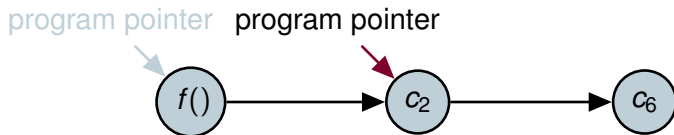
# Grafo de control de flujo

... con una pila.



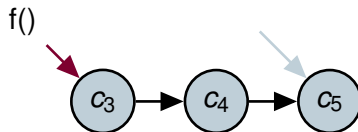
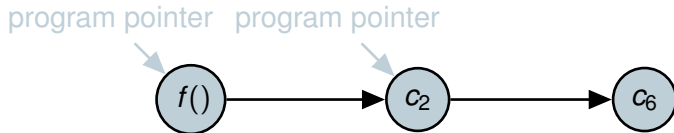
# Grafo de control de flujo

... con una pila.



# Grafo de control de flujo

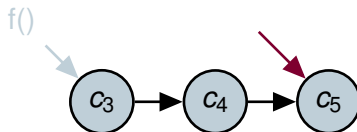
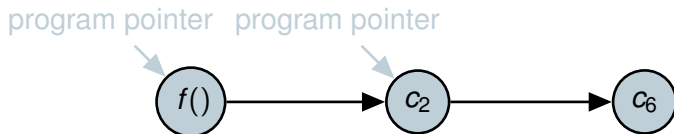
... con una pila.



f()
c2

# Grafo de control de flujo

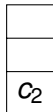
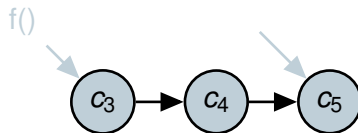
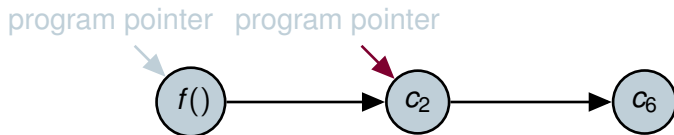
... con una pila.



c <sub>5</sub>
c <sub>2</sub>

# Grafo de control de flujo

... con una pila.



# Grafo de flujo de control

## Reglas

**type\_synonym** cfg\_label = enabled  $\times$  transformer

**inductive** cfg :: com  $\Rightarrow$  cfg\_label  $\Rightarrow$  com  $\Rightarrow$  bool

## Asignaciones

Assign: **cfg** (x ::= a) (en\_always, tr\_assign x a) SKIP

Assignl: **cfg** (x ::= a) (en\_always, tr\_assignl x a) SKIP

# Grafo de flujo de control

## Reglas

**type\_synonym** cfg\_label = enabled  $\times$  transformer

**inductive** cfg :: com  $\Rightarrow$  cfg\_label  $\Rightarrow$  com  $\Rightarrow$  bool

## Secuenciación

Seq1: **cfg** (SKIP;;  $c_2$ ) (en\_always, tr\_id)  $c_2$

Seq2:  $\llbracket \mathbf{cfg} \ c_1 \ a \ c'_1 \rrbracket \Longrightarrow \mathbf{cfg} \ (c_1;; c_2) \ a \ (c'_1;; c_2)$



# Grafo de flujo de control

## Reglas

**type\_synonym** cfg\_label = enabled  $\times$  transformer

**inductive** cfg :: com  $\Rightarrow$  cfg\_label  $\Rightarrow$  com  $\Rightarrow$  bool

## Condicionales

IfTrue: **cfg** (IF  $b$  THEN  $c_1$  ELSE  $c_2$ ) (en\_pos  $b$ , tr\_eval  $b$ )  $c_1$

IfFalse: **cfg** (IF  $b$  THEN  $c_1$  ELSE  $c_2$ ) (en\_neg  $b$ , tr\_eval  $b$ )  $c_2$

# Grafo de flujo de control

## Reglas

**type\_synonym** cfg\_label = enabled  $\times$  transformer

**inductive** cfg :: com  $\Rightarrow$  cfg\_label  $\Rightarrow$  com  $\Rightarrow$  bool

## Ciclos

While: **cfg** (WHILE  $b$  DO  $c$ ) (en\_always, tr\_id)  
(IF  $b$  THEN  $c$ ;; WHILE  $b$  DO  $c$  ELSE SKIP)

# Grafo de flujo de control

## Reglas

**type\_synonym** cfg\_label = enabled  $\times$  transformer

**inductive** cfg :: com  $\Rightarrow$  cfg\_label  $\Rightarrow$  com  $\Rightarrow$  bool

### Liberación de memoria

Free: **cfg** (FREE  $x$ ) (en\_always, tr\_free  $x$ ) SKIP

# Grafo de flujo de control

## Reglas

**type\_synonym** cfg\_label = enabled  $\times$  transformer

**inductive** cfg :: com  $\Rightarrow$  cfg\_label  $\Rightarrow$  com  $\Rightarrow$  bool

## Llamadas a funciones

Callfunl: **cfg** (Callfunl  $e$   $f$  params)

(en\_always, tr\_callfunl proc\_table  $e$   $f$  params) SKIP

Callfun: **cfg** (Callfun  $x$   $f$  params)

(en\_always, tr\_callfun proc\_table  $x$   $f$  params) SKIP

Callfunv: **cfg** (Callfunv  $f$  params)

(en\_always, tr\_callfunv proc\_table  $f$  params) SKIP

# Grafo de flujo de control

## Reglas

**type\_synonym** cfg\_label = enabled  $\times$  transformer

**inductive** cfg :: com  $\Rightarrow$  cfg\_label  $\Rightarrow$  com  $\Rightarrow$  bool

### Retorno de funciones

Return: **cfg** (Return *a*) (en\_always, tr\_return *a*) SKIP

Returnv: **cfg** Returnv (en\_always, tr\_return\_void) SKIP

# Semántica de pasos cortos

¿Por qué una semántica de pasos cortos?

# Semántica de pasos cortos

¿Por qué una semántica de pasos cortos?

Se quiere poder hablar de estados intermedios y diferenciar entre programas que no terminen y programas erróneos.

# Semántica de pasos cortos

¿Por qué una semántica de pasos cortos?

Se quiere poder hablar de estados intermedios y diferenciar entre programas que no terminen y programas erróneos.

La semántica de pasos cortos se define como una relación sobre estados:

**inductive** small\_step :: state  $\Rightarrow$  state option  $\Rightarrow$  bool

$s \rightarrow s_2$

$\equiv$

“Se toma un paso corto desde  $s$  hasta  $s_2$ ”



# Semántica de pasos cortos

¿Cuándo se puede dar un paso?

Un paso desde  $s$  hasta  $s_2$  se puede dar cuando:

- Paso regular:
  - ▶ Pila no vacía
  - ▶ Hay una arista en el CFG entre  $c_1$  y  $c_2$
  - ▶ La instrucción en el tope de la pila es  $c_1$
  - ▶ Aplicar la función `enabled` produce `True`
  - ▶ Aplicar la función `transformer` sobre  $s$  produce  $s_2$
- Paso de retorno desde una función sin valor de retorno:
  - ▶ Pila no vacía
  - ▶ La instrucción en el tope de la pila es `SKIP`
  - ▶ Aplicar la función `transformer tr_return_void` sobre  $s$  produce  $s_2$

# Semántica de pasos cortos

¿Cuándo falla el dar un paso?

Cuando hay una ejecución errónea, se da un paso desde  $s$  a  $\text{None}$ :

- Paso regular:
  - ▶ Pila no vacía
  - ▶ Hay una arista en el CFG entre  $c_1$  y  $c_2$
  - ▶ La instrucción en el tope de la pila es  $c_1$
  - ▶ Aplicar la función `enabled` o `transformer` produce  $\text{None}$
- Paso de retorno desde una función sin valor de retorno:
  - ▶ Pila no vacía
  - ▶ La instrucción en el tope de la pila es `SKIP`
  - ▶ Aplicar la función `transformer tr_return_void` sobre  $s$  produce  $\text{None}$

# Semántica de pasos cortos

¿Cómo dar varios pasos cortos?

Se lleva la definición de `small_step` a trabajar sobre `state option`:

**inductive** `small_step'` :: `state option`  $\Rightarrow$  `state option`  $\Rightarrow$  `bool`

$s \rightarrow s_2 \implies \text{Some } s \rightarrow s_2$

# Semántica de pasos cortos

¿Cómo dar varios pasos cortos?

Se lleva la definición de `small_step` a trabajar sobre `state option`:

**inductive** `small_step'` :: `state option`  $\Rightarrow$  `state option`  $\Rightarrow$  `bool`

$s \rightarrow s_2 \implies \text{Some } s \rightarrow s_2$

Para tomar varios pasos se utiliza la clausura reflexivo-transitiva sobre `small_step'`:

**inductive** `small_step'` :: `state option`  $\Rightarrow$  `state option`  $\Rightarrow$  `bool`

$s \rightarrow^* s_2$

$\equiv$

“Se toma uno o mas pasos cortos desde  $s$  hasta  $s_2$ ”

# Interpretador

## Ejecución de un programa

Un paso se ejecuta al aplicar la función *transformer* correspondiente al estado y actualizar la instrucción en el tope de la pila.

# Interpretador

## Ejecución de un programa

Un paso se ejecuta al aplicar la función *transformer* correspondiente al estado y actualizar la instrucción en el tope de la pila.

La semántica es determinística, lo que permite escribir un interpretador para la misma.

# Interpretador

## Ejecución de un programa

Un paso se ejecuta al aplicar la función *transformer* correspondiente al estado y actualizar la instrucción en el tope de la pila.

La semántica es determinística, lo que permite escribir un interpretador para la misma.

### Ejecutar un programa:

Mientras no se alcance un estado final, se ejecuta un paso.

# Interpretador

## Ejecución de un programa

Un paso se ejecuta al aplicar la función *transformer* correspondiente al estado y actualizar la instrucción en el tope de la pila.

La semántica es determinística, lo que permite escribir un interpretador para la misma.

### Ejecutar un programa:

Mientras no se alcance un estado final, se ejecuta un paso.

Solo se ejecutan programas que estén **bien formados**.

El interpretador retorna un estado final o None, que indica una ejecución errónea.



# Correctitud

Se demuestran varias propiedades con respecto a la semántica y el interpretador con asistencia de Isabelle/HOL. Tomemos como ejemplo la prueba del siguiente teorema:

```
lemma cfg_has_enabled_action:
  assumes " c \notequal SKIP"
  shows "\exists c' en tr. cfg c (en,tr) c'
    \and (en s = None \or en s = Some True)"
  using assms
proof (induction c)
  case (Seq c1 c2)
  show ?case
  proof (cases c1 = SKIP")
    case True
    thus ?thesis by (auto intro: cfg.intros)
  next
    case False
    from Seq.IH(1)[OF this]
    show ?thesis by (auto intro: cfg.intros)
  qed
next
```

```
case (If b c1 c2)
show ?case
proof (cases ".en_pos b s")
  case None[simp]
  thus ?thesis
  by (fastforce intro: cfg.intros)
next
case (Some a)[simp]
show ?thesis
proof (cases a)
  case True
  thus ?thesis
  by (fastforce intro: cfg.intros)
next
case False[simp]
have ".en_pos b s = Some False" by simp
hence ".en_neg b s = Some True"
  unfolding en_pos_def en_neg_def
  by (auto split: Option.bind_splits)
thus ?thesis
  by (fastforce intro: cfg.intros)
qed
qed
qed (auto intro: cfg.intros)
```

# Correctitud

Se demuestran dos principales teoremas.

## Teorema: La semántica es determinística

$$\begin{aligned} s \rightarrow s' \wedge s \rightarrow s'' &\implies s' = s'' \\ s \rightarrow' s' \wedge s \rightarrow' s'' &\implies s' = s'' \end{aligned}$$

Para demostrar este teorema fue necesario demostrar 9 lemas previos.

## Teorema: El interpretador es consistente con respecto a la semántica

$$\begin{aligned} \text{terminates } CS &\implies \\ \text{yields } CS \ CS' &\iff (CS' = \text{interp proc\_table } CS) \end{aligned}$$

Para demostrar este teorema fue necesario demostrar 6 lemas previos.

# Pretty Printing

## Generación de código para factorial

Se toma el siguiente programa en la semántica:

```
definition factorial_decl :: fun_decl
where "factorial_decl ==
  ( fun_decl.name = fact,
    fun_decl.params = [n],
    fun_decl.locals = [r, i],
    fun_decl.body =
      r ::= (Const 1);;
      i ::= (Const 1);;
      (WHILE (Less (V i) (Plus (V n) (Const 1))) DO
        (r ::= (Mult (V r) (V i));;
         i ::= (Plus (V i) (Const 1)))
      );;
      RETURN (V r)
  )"
```

```
definition main_decl :: fun_decl
where "main_decl ==
  ( fun_decl.name = main,
    fun_decl.params = [],
    fun_decl.locals = [],
    fun_decl.body =
      n ::= Const 5;;
      r ::= fact ([V n])
  )"
```

```
definition p :: program
where "p ==
  ( program.name = fact,
    program.globals = [n, r],
    program.procs = [factorial_decl, main_decl]
  )"
```

y se exporta código para el mismo.

# Pretty Printing

## Generación de código para factorial

Se obtiene el siguiente código:

```
#include <stdlib.h>
#include <stdio.h>
#include <limits.h>
#include <stdint.h>
#include "../test_harness.h"
#include "../malloc_lib.h"
#ifdef INTPTR_MIN
    #error ("Macro INTPTR_MIN undefined")
#endif
#ifdef INTPTR_MAX
    #error ("Macro INTPTR_MAX undefined")
#endif
#if ( INTPTR_MIN + 1 != -9223372036854775807 )
    #error (" Assertion INTPTR_MIN + 1 == -9223372036854775807 failed")
#endif
#if ( INTPTR_MAX != 9223372036854775807 )
    #error (" Assertion INTPTR_MAX == 9223372036854775807 failed")
#endif
```

```
intptr_t n;
intptr_t r;
```

```
intptr_t fact(intptr_t n)
{
    intptr_t r;
    intptr_t i;
    r = (1);
    i = (1);
    while ((i) < ((n) + (1)))
        r = ((r) * (i));
        i = ((i) + (1));

    return(r);
}
```

```
intptr_t main()
{
    n = (5);
    (r) = (fact(n));
}
```

# Pretty Printing

## Generación de código para factorial

Se obtiene el siguiente código:

```
#include <stdlib.h>
#include <stdio.h>
#include <limits.h>
#include <stdint.h>
#include "../test_harness.h"
#include "../malloc_lib.h"
#ifdef INTPTR_MIN
    #error ("Macro INTPTR_MIN undefined")
#endif
#ifdef INTPTR_MAX
    #error ("Macro INTPTR_MAX undefined")
#endif
#if ( INTPTR_MIN + 1 != -9223372036854775807 )
    #error (" Assertion INTPTR_MIN + 1 == -9223372036854775807 failed")
#endif
#if ( INTPTR_MAX != 9223372036854775807 )
    #error (" Assertion INTPTR_MAX == 9223372036854775807 failed")
#endif
```

```
intptr_t n;
intptr_t r;
```

```
intptr_t fact(intptr_t n)
{
    intptr_t r;
    intptr_t i;
    r = (1);
    i = (1);
    while ((i) < ((n) + (1)))
        r = ((r) * (i));
        i = ((i) + (1));

    return(r);
}
```

```
intptr_t main()
{
    n = (5);
    (r) = (fact(n));
}
```

# Pretty Printing

## Generación de código para factorial

Se obtiene el siguiente código:

```
#include <stdlib.h>
#include <stdio.h>
#include <limits.h>
#include <stdint.h>
#include "../test_harness.h"
#include "../malloc_lib.h"
#ifdef INTPTR_MIN
    #error ("Macro INTPTR_MIN undefined")
#endif
#ifdef INTPTR_MAX
    #error ("Macro INTPTR_MAX undefined")
#endif
#if ( INTPTR_MIN + 1 != -9223372036854775807 )
    #error (" Assertion INTPTR_MIN + 1 == -9223372036854775807 failed")
#endif
#if ( INTPTR_MAX != 9223372036854775807 )
    #error (" Assertion INTPTR_MAX == 9223372036854775807 failed")
#endif
```

```
intptr_t n;
intptr_t r;
```

```
intptr_t fact(intptr_t n)
{
    intptr_t r;
    intptr_t i;
    r = (1);
    i = (1);
    while ((i) < ((n) + (1)))
        r = ((r) * (i));
        i = ((i) + (1));

    return(r);
}
```

```
intptr_t main()
{
    n = (5);
    (r) = (fact(n));
}
```

# Pretty Printing

## Generación de código para factorial

Se obtiene el siguiente código:

```
#include <stdlib.h>
#include <stdio.h>
#include <limits.h>
#include <stdint.h>
#include "../test_harness.h"
#include "../malloc_lib.h"
#ifdef INTPTR_MIN
    #error ("Macro INTPTR_MIN undefined")
#endif
#ifdef INTPTR_MAX
    #error ("Macro INTPTR_MAX undefined")
#endif
#if ( INTPTR_MIN + 1 != -9223372036854775807 )
    #error (" Assertion INTPTR_MIN + 1 == -9223372036854775807 failed")
#endif
#if ( INTPTR_MAX != 9223372036854775807 )
    #error (" Assertion INTPTR_MAX == 9223372036854775807 failed")
#endif
```

```
intptr_t n;
intptr_t r;
```

```
intptr_t fact(intptr_t n)
{
    intptr_t r;
    intptr_t i;
    r = (1);
    i = (1);
    while ((i) < ((n) + (1)))
        r = ((r) * (i));
        i = ((i) + (1));

    return(r);
}

intptr_t main()
{
    n = (5);
    (r) = (fact(n));
}
```

# Pretty Printing

## Generación de código para factorial

Se obtiene el siguiente código:

```
#include <stdlib.h>
#include <stdio.h>
#include <limits.h>
#include <stdint.h>
#include "../test_harness.h"
#include "../malloc_lib.h"
#ifndef INTPTR_MIN
    #error ("Macro INTPTR_MIN undefined")
#endif
#ifndef INTPTR_MAX
    #error ("Macro INTPTR_MAX undefined")
#endif
#if ( INTPTR_MIN + 1 != -9223372036854775807 )
    #error (" Assertion INTPTR_MIN + 1 == -9223372036854775807 failed")
#endif
#if ( INTPTR_MAX != 9223372036854775807 )
    #error (" Assertion INTPTR_MAX == 9223372036854775807 failed")
#endif
```

```
intptr_t n;
intptr_t r;
```

```
intptr_t fact(intptr_t n)
    intptr_t r;
    intptr_t i;
    r = (1);
    i = (1);
    while ((i) < ((n) + (1)))
        r = ((r) * (i));
        i = ((i) + (1));

    return(r);
```

```
intptr_t main()
    n = (5);
    (r) = (fact(n));
```



# Pretty Printing

## Generación de código para factorial

Se obtiene el siguiente código:

```
#include <stdlib.h>
#include <stdio.h>
#include <limits.h>
#include <stdint.h>
#include "../test_harness.h"
#include "../malloc_lib.h"
#ifndef INTPTR_MIN
    #error ("Macro INTPTR_MIN undefined")
#endif
#ifndef INTPTR_MAX
    #error ("Macro INTPTR_MAX undefined")
#endif
#if ( INTPTR_MIN + 1 != -9223372036854775807 )
    #error (" Assertion INTPTR_MIN + 1 == -9223372036854775807 failed")
#endif
#if ( INTPTR_MAX != 9223372036854775807 )
    #error (" Assertion INTPTR_MAX == 9223372036854775807 failed")
#endif
```

```
intptr_t n;
intptr_t r;
```

```
intptr_t fact(intptr_t n)
{
    intptr_t r;
    intptr_t i;
    r = (1);
    i = (1);
    while ((i) < ((n) + (1)))
        r = ((r) * (i));
        i = ((i) + (1));

    return(r);
}
```

```
intptr_t main()
{
    n = (5);
    (r) = (fact(n));
}
```

# Pretty Printing

## Casting

Se traducen los programas usando el tipo `intptr_t` de C.

Este tipo permite que tanto un apuntador como un entero se guarde en el.

Todas las variables se definen con el tipo “`intptr_t`”.

Se debe poder imprimir un *cast* desde y hacia apuntadores para indicar a C como interpretar los valores

# Pretty Printing

## Casting

Se traducen los programas usando el tipo `intptr_t` de C.

Este tipo permite que tanto un apuntador como un entero se guarde en el.

Todas las variables se definen con el tipo “`intptr_t`”.

Se debe poder imprimir un *cast* desde y hacia apuntadores para indicar a C como interpretar los valores

### Convertir a apuntador

```
(intptr_t *) <expression>  
*((intptr_t *) foo);
```

### Convertir desde apuntador

```
(intptr_t) <expression>  
(intptr_t) __MALLOC(sizeof(intptr_t) * 5);
```

# Pretty Printing

¿Cómo se traduce malloc bajo la suposición de memoria ilimitada?

Se traduce malloc envolviendola en otra función.

## Generación de código para malloc

```
New (Const 9)  $\equiv$  __MALLOC(sizeof(intptr_t) * 9)
```

Esta función abortará la ejecución del programa si encuentra un error de memoria.

# Exportando Código C

## Exportar a un archivo

Se provee con un mecanismo que permite exportar el programa generado en código C a un archivo en un directorio determinado.

# Pruebas

## ¿Por qué hacemos pruebas?

El propósito de las pruebas es incrementar la confianza en el proceso de traducción.

El código generado a partir de la semántica se comporta como sigue:

- Abortará su ejecución si encuentra un error de memoria (no se hace promesa alguna acerca de estos programas)
- Producirá un resultado equivalente al del mismo programa interpretado en Isabelle

# Pruebas

## Tipos de verificaciones

Se definen tres tipos de verificaciones en la batería de pruebas:

- Programas de ejemplo en Chloe
- Pruebas intencionalmente incorrectas
- Programas con pruebas generadas automáticamente que verifican la igualdad de estados finales

# Pruebas

## Igualdad de estados finales

Se provee la opción de generar código para probar los programas.  
Se quiere verificar que el estado final producido por el programa en C es el mismo que al interpretar el programa.

Igualdad entre el contenido de las variables globales y la memoria alcanzable.

- Valores enteros
- Apuntadores a Null
- Apuntadores diferentes de Null



# Pruebas

## Siguiendo apuntadores

Cuando el contenido de una variable global es un apuntador a memoria, se verifica el contenido completo del bloque de memoria ya que es alcanzable.

# Pruebas

## Siguiendo apuntadores

Cuando el contenido de una variable global es un apuntador a memoria, se verifica el contenido completo del bloque de memoria ya que es alcanzable.

Cuando se encuentra un apuntador a memoria se sigue el mismo hasta que:

- Se encuentra un entero o un apuntador a null
- Se encuentra un apuntador que ya fue seguido

# Pruebas

## Siguiendo los apuntadores en orden DFS

Para hacer esto se debe:

- Seguir los apuntadores en el orden de búsqueda en profundidad
- Mantener un conjunto de bloques visitados

Para evitar seguir apuntadores indefinidamente cuando existen referencias cíclicas:

Si se encuentra un apuntador a un bloque de memoria que ya fue visitado, se deja de seguir el apuntador y se comparan los valores de los apuntadores.

# Pruebas

## Automatización del proceso de pruebas

Para automatizar el proceso de pruebas se definieron dos ambientes de pruebas (uno en Isabelle y otro en C) que ayudaran en el proceso de generación y corrida de pruebas.

# Pruebas

## Automatización del proceso de pruebas

Para automatizar el proceso de pruebas se definieron dos ambientes de pruebas (uno en Isabelle y otro en C) que ayudaran en el proceso de generación y corrida de pruebas.

Además de la generación de pruebas se necesita una manera de ejecutar todas las pruebas en la batería.

# Pruebas

## Automatización del proceso de pruebas

Para automatizar el proceso de pruebas se definieron dos ambientes de pruebas (uno en Isabelle y otro en C) que ayudaran en el proceso de generación y corrida de pruebas.

Además de la generación de pruebas se necesita una manera de ejecutar todas las pruebas en la batería.

- Se define un archivo en Isabelle que importa cada prueba en la batería.
- Se define un Makefile y un shell script que automatizará el proceso de compilación y corrida del código generado.

# Conclusiones

## Resultados de este trabajo

- Semántica formalizada para Chloe
- Interpretador correcto para Chloe dentro del ambiente Isabelle/HOL
- Traducción desde Chloe a código C
- Desarrollo de una batería de pruebas y un ambiente de pruebas
  - ▶ No se puede dar una métrica exacta para el proceso de pruebas

# Trabajos futuros

¿Qué direcciones tomar a partir de acá?

- Mejorar la batería de pruebas. Importar pruebas de fuentes externas y traducirlas a Chloe
- Formalización de una semántica axiomática con una lógica de separación para razonar acerca de programas en Chloe
- Agregar un sistema de tipos estático para demostrar que los programas son *type safe*
- Enlazar Chloe al Isabelle Refinement Framework de modo que programas del *framework* se puedan refinar a Chloe
- Aumentar el conjunto de características soportadas por Chloe
  - ▶ Agregar más expresiones o instrucciones
  - ▶ Agregar operaciones de I/O
  - ▶ Agregar concurrencia



¡Gracias!