Paradigmas de Programación (EIF-400) CHVM

CARLOS LORÍA-SÁENZ

OCTUBRE 2015

EI/UNA

Objetivos

- Explicar la “arquitectura” de la CHVM
- Tener una especificación unificada para el proyecto correspondiente

Arquitectura: memorias

- La CHVM tiene cuatro memorias
- Data Stack (DS): pila donde se almacenan los datos para operaciones
- Call Stack (CS): pila donde se almacena la información de las funciones que han sido activadas (llamadas). La actual en el top
- Code Area (CA): área donde se almacena el código de cada función
- Data Memory (DM): área donde se almacenan los datos locales de cada función. Un área por cada activación

Arquitectura: registros

- SP: stack pointer apunta al inicio del área en la pila de la clausura activa
- PC: program counter apunta a la siguiente instrucción en la CA a ejecutar

Arquitectura: tabla de funciones

- CH tiene una tabla de funciones FT
- Cada función estará registrada con su número FT

Código compilado

- Ver muestra.cho

Convención

- EL compilador lee un fuente .ch y genera un .cho
- Son de texto ambos
- Esa es la entrada a la CHVM

Componentes

- La CHVM tiene dos componentes: el Loader y y el Evaluator
  - Loader lee el .cho y crea los objetos respectivos
  - Asigna segmentos en CA para cada función
  - Crea la FT y asigna una entrada a cada función
- Evaluator crea una instancia de la CHVM y un proceso (ver adelante) y le pide al proceso empezar en el main

Operación: Carga y registro de funciones

- La CHVM carga un código compilado de CH
- Lo parsea lo vuelve a representar como secuencia de funciones: programa. Siendo una función una secuencia de instrucciones que siempre termina en un ret (retorno)
- A cada función f se le asigna un segmento en la CA. Se registra en FT
- La función main es la que se ejecuta como punto de entrada

Inicio y ciclo de ejecución

- Para iniciar se crea una CHVM y un proceso (ver adelante)
- El proceso arranca con main:
  - Hace push de un registro de activación para `main` en `CS` (ver adelante)
  - Pone el `PC` a apuntar a `main.start()`
- Se inicia el ciclo de ejecución
  - nextInstruction = `CA[PC]`
  - Incrementar el `PC`
  - Ejecutar nextInstruction usando el proceso
- Cada instrucción tiene su propia lógica de ejecución

Proceso

- Un CHProcess es un objeto que encapsula el acceso a la máquina CHVM (Proxy)
- Es decir las instrucciones tienen acceso a DS, CS, M, CA, FT, etc vía un proceso
- Tiene además una salida estándar (por defecto es System.out)
- Usando CA[PC] ejecuta una instrucción a la vez
- Para ejecutar el proceso invierte el control y le pide a la instrucción que se ejecute (exec) usando el proceso

Datos: CHObject<T>

- Los datos (en M y en DS) son objetos de tipo CHObject<T>
- Se requieren CHBoolean y CHInteger como subtipos
- TRUE y FALSE son los objetos de tipo CHBoolean
- El método T value() da acceso al valor como Objeto Java

CHFunction

- Representa una función (lambda)
- Tiene un índice único asignado por el compilador
- Tiene un indicador de cuántas instrucciones (size) son y otro de cuánta memoria local requiere (local.size()). También sabe el índice de su función contenedora
- Permite acceder a sus segmentos en CA (método start) y en M (local)
- El segmento en CA lo asigna el cargador antes de ejecutar
- El de M se asigna “fresco” en cada activación

CHActivation

- Es un objeto que almacena una CHFunction y representa como está/fue activada
- Permite saber el SP y el PC que había al activarla
- EL CS lo que almacena son objetos de este tipo
- Simbólicamente escribimos <f, PC, SP> para denotar la creación de CHActivation para una función f, con PC, SP actuales
- A la activación hay que asignarle memoria en M
- Al retornar la activación libera sus recursos y restaura el SP y el PC

Instrucciones

- Son objetos de tipo CHInstruction<T> que tienen método void exec(CHProcess p)
- exec realiza la operación como sumar, almacenar, imprimir, etc usando las pilas y la memoria, etc
- En lo que sigue se describen las instrucciones. Donde se diga DS, M, CS, CA, SP, PC etc se refiere a métodos de un CHProcess
- Lo que se especifica en forma simbólica es lo que el método exec de cada instrucción debe realizar en cada caso

Lógica de ejecución (exec): loads

- LC k
  - DS.push(kObj) donde kObj es un objeto cuyo valor es k
- LD k
  - DS.push(CS.top().local[k])
- LD f k
  - DS.push(CS.la(f).local[k]) donde la(f) es la última activación de f en CS

Lógica de Ejecución: stores

- ST k
  - CS.top().local[k] = DS.pop()
- ST f k
  - CS.la(f).local[k] = DS.pop() donde la(f) es la última activación de f

Lógica de Ejecución: Aritmética

- Sea op la operación asociada a ADD, DIV, MINUS, MULT en un ChInteger
- right = DS.pop()
- left = DS.pop()
- DS.push(left.op(right))
- Se debe chequear que son CHInteger

Lógica de Ejecución: relacionales

- Sea op la operación asociada a EQ, NEQ, LEQ de un ChBoolean
  - right = DS.pop()
  - left = DS .pop()
  - DS.push(left.op(right))
- NEG con operador op
  - DS.push(op(DS.pop()))
- Los objetos deben ser Chbooleanos

Lógica de ejecución: branches

- BR k
  - PC = PC + k
- BRT k
  - Si DS.top() == TRUE PC = PC + k
- BRF k
  - Si DS.top() == FALSE PC = PC + k
- TRUE y FALSE siendo los CHBoolean de la CHVM

Lógica de Ejecución: call

- CALL
  - p = DS.pop() (parámetro de la llamada)
  - f =DS.pop() (índice de la función)
  - ar = <FT[f], PC, SP> nuevo registro de activación
  - M.allocateMemory(ar)
  - DS.push(p)
  - CS.push(ar)
  - PC = ar.start()
- Se debe chequear que f sea un índice válido de una entrada en FT

Lógica de Ejecución: return

- RET
  - ar = CS.pop() el activation record actual
  - PC = ar.PC()
  - SP = ar.SP()
  - retValue = DS.pop()
  - DS.slice(SP) borra desde el top hasta el SP anterior
  - DS.push(retValue);
  - M.deallocateMemory(ar)

Lógica de ejecución

- PRINT
  - print(DS.pop())
  - Donde print es una función del proceso que escribe a la salida estándar actual
- EXIT
  - termina la ejecución de la CHVM