Jinja

Abstract Syntax

Names

Three types of names:

- Class names: *cname*
- Variable and field names: *vname*
- Method names: *mname*

Variable conventions

- C, D: class name
- e: expressions
- F: field name
- M: method name
- P: program
- T: type
- v: value
- V: variable name

Values

A value (type *val*) is one of the following:

- a boolean *Bool b*, where $b :: bool$
- an integer *Intg i*, where $i :: int$
- a reference *Addr a*, where $a :: addr$
  (*addr*: some arbitrary type of addresses)
- the null reference *Null*
- the dummy value *Unit*

Jinja is a hybrid object oriented language:
it has primitive values and classes

The formal model

$$\textbf{datatype}\ \mathit{val} = \mathit{Unit} \mid \mathit{Null} \mid \mathit{Bool}\ \mathit{bool} \mid \mathit{Intg}\ \mathit{int} \mid \mathit{Addr}\ \mathit{addr}$$

Jinja Types

A Jinja type (type *ty*) is one of the following:

- *Boolean*
- *Integer*
- *Class C*
- ...

More later.

Expressions and statements

In Jinja, everything is an *expression*.

Statements are expressions evaluating to *Unit*.

Expressions (1)

Type: *expr*

- `new` C
- `Cast` C e
- `Val` v
- e_1 «*bop*» e_2
- `Var` V
- V := e
- $e.F\{D\}$
- $e.F\{D\}$:= e'
- $e.M(es)$

Interlude: annotation $\{\mathbf{D}\}$

$e.F\{D\}$ and $e.F\{D\} := e'$:

- D is class where F is declared
- D is needed during runtime:
  field access is static, not dynamic!
- D is added by the type checker (see later)

Expressions (2)

- $\{V\!:\!T;\ e\}$
- e_1`;` e_2
- `if` (e) e_1 `else` e_2
- `while` (e) e'
- Exceptions: later

Exercise: which Java expressions/statements are missing?

Abstract and concrete syntax

Formal definition:

datatype $expr = new\ cname \mid Cast\ cname\ expr \mid \ldots$

This is *abstract* syntax.

For readability: additional *concrete* syntax, e.g.

$V := e$ is just syntactic sugar for $LAss\ V\ e$

Jinja is a mix of concrete and abstract syntax

Example

Translate `i = i + 1` into Jinja syntax.

Abbreviations

$$\begin{array}{lcl@{\qquad\qquad}lcl}
\mathit{true} & \equiv & \mathtt{Val}(\mathit{Bool\ True}) & \mathit{false} & \equiv & \mathtt{Val}(\mathit{Bool\ False}) \\
\mathit{addr\ a} & \equiv & \mathtt{Val}(\mathit{Addr\ a}) & \mathit{null} & \equiv & \mathtt{Val}\ \mathit{Null} \\
\mathit{unit} & \equiv & \mathtt{Val}\ \mathit{Unit} & & &
\end{array}$$

Why no return statement?

Because everything has a value.

How to eliminate `return`:

- `return e` $\rightsquigarrow$?
- `while (a) {if (b) return c else s};`
  `return d`
  $\rightsquigarrow$?

Eliminating return in general

- Introduce new variables `res`(ult) and `cont`(inue).
- Eliminate `return`:
  `return e` $\leadsto$ `res := e; cont := false`
- Restructure control flow:
  `e1; e2` $\leadsto$ `e1; if (cont) e2`
  `while (b) e` $\leadsto$ `while (b & cont) e`
- Transform procedure body:
  `e` $\leadsto$ `cont := true; e; res`

The need for Var

Without `Var`, $V := e$ is ambiguous: is e

- a program variable or
- a meta-language variable ranging over expressions?

Binary operators

For simplicity: only $=$ and $+$
Semantics: *binop* $::$ *bop* $\times$ *val* $\times$ *val* $\Rightarrow$ *val option*

$$\begin{array}{lcl} \textit{binop}\ (=,\ v_1,\ v_2) & = & \lfloor \textit{Bool}\ (v_1 = v_2) \rfloor \\ \textit{binop}\ (+,\ \textit{Intg}\ i_1,\ \textit{Intg}\ i_2) & = & \lfloor \textit{Intg}\ (i_1 + i_2) \rfloor \\ \textit{binop}\ (\textit{bop},\ v_1,\ v_2) & = & \textit{None} \end{array}$$

Can only add integers!

Design principle:

In ill-typed situations, semantics does nothing
Defensive semantics

Jinja types

- They exist
- *ty* is the type of all Jinja types
- Semantics does not need them
- See later

Abstract Syntax: Programs

Concrete syntax

Many possibilities:

- Subset of Java
- Imaginary syntax, e.g.

```
class B extends A
 {field F : TF; field G : TG;
  method M1 : (p1:T1,...,pn:Tn) -> T = {...}
  ...
 }
class C extends B {...}
```

- ...

We don’t care!

Abstract syntax

- A program is a list of class declarations:
  J-prog $=$ *J-cdecl list*
- A class declaration consists of the name and the class:
  J-cdecl $=$ *cname* $\times$ *J-class*
- A class is consists of a superclass name, a list of field declarations and a list of method declarations:
  J-class $=$ *cname* $\times$ *fdecl list* $\times$ *J-mdecl list*
- A field declaration is a field name and a type:
  fdecl $=$ *vname* $\times$ *ty*
- A method declaration is a method name, the argument types, the result type, and the method body:
  J-mdecl $=$ *mname* $\times$ *ty list* $\times$ *ty* $\times$ *J-mb*
- A method body is a parameter list and an expression:
  J-mb $=$ *vname list* $\times$ *expr*

Example

```
class B extends A
 {field F : TF; field G : TG;
  method M1 : (p1:T1,p2:T2) -> T =
   {i:Ti; i := i}
 }
```

$\leadsto$?

Note

Parameter names and types have been separated.

- Advantage: can re-use same declaration structure for machine programs (where method parameters do not have names) by redefining *J-mb*
- Disadvantage: need to make sure that there the same number of parameter names as types.

Exercise

Which Java features are missing?

Accessing declaration information

- *class P C* is the class associated with C in P.
- *is-class P C* means class C is defined in P.
- $P \vdash C \preceq^* D$ means C is a **subclass** of D. The relation is transitive and reflexive.
- $P \vdash C$ **sees** $M\colon Ts \rightarrow T = mb$ **in** D means that in P from class C a method M with argument types Ts (a type list), result type T, and body mb is visible in class D (taking overriding into account).
- $P \vdash C$ **sees** $F{:}T$ **in** D means that in P from class C a field F of type T is visible in class D.

Example

```
class B extends A {field F:TB
                   method M:TBs->T1 = mB}
class C extends B {field F:TC
                   method M:TCs->T2 = mC}
```

Example

$P \vdash C$ **sees** $F{:}TC$ **in** C	YES
$P \vdash C$ **sees** $F{:}TB$ **in** B	NO
$P \vdash C$ **sees** $M{:}\ TCs{\rightarrow}T2 = mC$ **in** C	YES
$P \vdash C$ **sees** $M{:}\ TBs{\rightarrow}T1 = mB$ **in** B	NO

No overloading of method names as in Java!

Formal definitions

- *class P C* $\equiv$ *map-of P C*
- *is-class P C* $\equiv$ *class P C* $\neq$ *None*
- $P \vdash C \preceq^* D$ is the reflexive transitive closure of the direct subclass relation $P \vdash C \prec^1 D$ defined by $[\![\textit{class}\ P\ C = \lfloor(D, \textit{rest})\rfloor;\ C \neq \textit{Object}]\!] \Longrightarrow P \vdash C \prec^1 D$

Collecting fields

Auxiliary relation $P \vdash C$ **has-fields** $FDTs$
with $FDTs :: ((vname \times cname) \times ty)\ list$:

- $((F, D), T)$ in $FDTs$ represents a field declaration $F{:}T$ in a superclass D of C
- $FDTs$ starts with C and climbs up the class hierarchy

$[\![class\ P\ C = \lfloor (D, fs, ms) \rfloor;\ C \neq Object;\ P \vdash D$ **has-fields** $FDTs]\!]$
$\Longrightarrow P \vdash C$ **has-fields** $map\ (\lambda(F, T).\ ((F, C), T))\ fs\ @\ FDTs$

$P \vdash Object$ **has-fields** $[]$

Example (with `A = Object`):

$P \vdash C$ **has-fields** $[((F, C), TC), ((F, B), TB)]$

sees fields

$P \vdash C$ **sees** $F{:}T$ **in** $D \equiv$

$\exists\, FDTs.$

$P \vdash C$ **has-fields** $FDTs \wedge$

$map\text{-}of\ (map\ (\lambda((F, D), T).\ (F, D, T))\ FDTs)\ F = \lfloor(D, T)\rfloor$

Exercise

Define **sees** for methods.