Translation Schemes for TassKaf

Stephan Diehl, Universität des Saarlandes

May 15, 1997

Contents

In the following translation scheme we assume that the function ξ translates types into equivalent types in JASMIN notation.

0.1 Expressions

Table 1: Translation of Expressions

Function		Restriction
$code(e_1 == e_2)\ \rho$	$= code\ e_1\ \rho; code\ e_2\ \rho;$ **isub**; **ifeq** l_1; **iconst_0**; **goto** $l_2; l_1:$ **iconst_1**; $l_2:$	$Type(e_1) = Type(e_2) = N$
$code(e_1 != e_2)\ \rho$	$= code\ e_1\ \rho; code\ e_2\ \rho;$ **isub**;	$Type(e_1) = Type(e_2) = N$
$code(e_1 < e_2)\ \rho$	$= code\ e_1\ \rho; code\ e_2\ \rho;$ **isub**; **ifle** l_1; **iconst_0**; **goto** $l_2; l_1:$ **iconst_1**; $l_2:$	$Type(e_1) = Type(e_2) = N$
$\vdots$		
$code(e_1 + e_2)\ \rho$	$= code\ e_1\ \rho; code\ e_2\ \rho;$ **iadd**	$Type(e_1) = Type(e_2) = N$
$code(e_1 - e_2)\ \rho$	$= code\ e_1\ \rho; code\ e_2\ \rho;$ **isub**	$Type(e_1) = Type(e_2) = N$
$code(e_1 * e_2)\ \rho$	$= code\ e_1\ \rho; code\ e_2\ \rho;$ **imul**	$Type(e_1) = Type(e_2) = N$
$code(e_1 / e_2)\ \rho$	$= code\ e_1\ \rho; code\ e_2\ \rho;$ **idiv**	$Type(e_1) = Type(e_2) = N$
$code(-e)\ \rho$	$= code\ e\ \rho;$ **ineg**	$Type(e) = N$
$code\ x\ \rho$	$=$ **iload** $\rho(x)$	x local variable of type N
$code\ c\ \rho$	$=$ **ldc** $\gamma_K^{-1}(c)$	c constant of type N
$code(x = e)\ \rho$	$= code\ e\ \rho;$ **istore** $\rho(x)$	$Type(e) = T$, x local variable

In TassKaf we have not only local integer variables, but class and object variables and local variables of array or object type:
code x $\rho=$ **getfield** $\xi(x)\ \xi(t)$ where x is object variable of type t.
code x $\rho=$ **getstatic** $\xi(x)\ \xi(t)$ where x is class variable of type t.
code x $\rho=$ **aload** ρ(x) where x is a local variable of array or object type.

0.2 Statements

$code($**if** e **then** st_1 **else** st_2 **fi**$)\ \rho =$
  $code\ e\ \rho;$ **ifeq** $l_1;\ code\ st_1\ \rho;$ **goto** $l_2;\ l_1:\ code\ st_2\ \rho;\ l_2:$

$code($**if** e **then** st **fi**$)\ \rho = code\ e\ \rho;$ **ifeq** $l;\ code\ st\ \rho;\ l:$

$code$ (**while** e **do** st **od**) $\rho = l_1:\ code\ e\ \rho;$ **ifeq** $l_2;\ code\ st\ \rho;$ **goto** $l_1;\ l_2:$

$code$ (**do** st **while** e) $\rho = l:\ code\ st\ \rho;\ code\ e\ \rho;$ **ifne** l

$code\ (st_1;st_2)\ \rho = code\ st_1\ \rho;\ code\ st_2\ \rho$

$$code \begin{pmatrix} \textbf{case}\ e\ \textbf{of} & \\ \quad i_1 : & st_1\ ; \\ \quad \vdots & \\ i_k : st_k\ ; & \\ \textbf{default} : & st \\ \textbf{end} & \end{pmatrix} \rho \ = \ \begin{array}{rl} & \textbf{lookupswitch}\ l_d\ k\ i_1\ l_1 \ldots\ i_k\ l_k\ ; \\ l_1 : & code\ st_1 \rho'; \\ & \vdots \\ l_k : & code\ \ st_k\ \rho'; \\ l : & code\ \ st\ \rho'; \\ b : & \end{array}$$

where $\rho' = \rho[break \mapsto b]$.

$code$ **break** ρ = **goto** ρ(break)

The **for** loop can be translated into a **while** loop:

$$\begin{array}{c} \textbf{for}(init;condition;step)\ body \\ \Downarrow \\ init;\textbf{while}\ condition\ \textbf{do}\ body;step\ \textbf{od} \end{array}$$

But we can also combine the individual translation schemes into one:

$code$ (**for**($init$;$condition$;$step$) $body$) ρ =
$code$ $init$; l_1: $code$ $condition$ ρ; **ifeq** l_2; $code$ $body$ ρ; $code$ $step$ ρ; **goto** l_1; l_2:

0.3 Arrays

To access the value of an multidimensional array of integers we generate the following code:

$code$ $c[i_1]\ldots[i_k]$ ρ =
aload ρc;
$code$ i_1 ρ;
aaload;
⋮
$code$ i_{k-1} ρ;
aaload;
$code$ i_k ρ;
iaload;

If k is smaller than the dimension of the array stored in c, then we have to use **aaload** instead of **iaload**. We also use **aaload** if c contains an array of objects.

0.4 Classes

Declarations of classes are translated as follows (JASMIN)

$code$ (**class** c **extends** s $body$) ρ =
.class public $\xi(c)$

.super $\xi(s)$
$code_{Fields}$ $body\ \rho$
.method public <init>()V
aload_0
invokenonvirtual $\xi(s)$/ <init>()V
$code_{InitObjectFields}$ $body\ \rho$
return
.end method
.method public <clinit>()V
$code_{InitClassFields}$ $body\ \rho$
return
.end method
$code\ body\ \rho$;
.end method

In the above translation scheme we have three auxiliary function which traverse the *body* of the class declaration and process field definitions (i.e. class or object variables) in a special way:
$code_{Fields}\ (t\ x = e)\ \rho\ =$ .field public $x\xi(t)$
$code_{Fields}\ (static\ t\ x = e)\ \rho\ =$ .field public static $x\xi(t)$
$code_{InitObjectFields}\ (t\ x = e)\ \rho\ =\ code\ (x = e)\ \rho$
$code_{InitClassFields}\ (static\ t\ x = e)\ \rho\ =\ code\ (x = e)\ \rho$

Finally an instance of an object is created with the **new** instruction and has to be initialized by calling the $< init >$ method of the class:
$code$ **new** $c()\ \rho$ = **new** c; **dup**; **invokenonvirtual** $c/ < init > ()V$

0.5 Methods

0.5.1 Local Variables

Declarations of local variables are translated as follows:
$code\ \ (t\ x{=}init)\ \rho\ = code\ init\ \rho$; **istore** ρ(x)
where $t \in \{int, bool\}$.

$code\ \ (t\ x{=}init)\ \rho\ = code\ init\ \rho$; **astore** ρ(x)
where t is a reference type.

0.5.2 Method Declarations

Declarations of methods are translated as follows (JASMIN)

$code$ (**static** $t\ m(t_1\ p_1,\ldots,t_k\ p_k);body)\ \rho\ =$
.method public static $m(\xi(t_1)...\xi(t_k))\xi(t)$
$code\ body\ \rho$;
.end method

code $(t\ m(t_1\ p_1, \ldots, t_k\ p_k); body)\ \rho\ =$
.method public $m(\xi(t_1)...\xi(t_k))\xi(t)$
code body ρ;
.end method

0.5.3 Method Return

code **return** ρ = **return**

code (**return** e) ρ =
code $e\ \rho$
ireturn
if $infer\_type(e) \in \{int, bool\}$

code (**return** e) ρ =
code $e\ \rho$
areturn
if $infer\_type(e)$ yields a reference type

0.5.4 Method Call

For static methods we only have to pass the arguments to the method:
code $c.m(e_1, \ldots, e_n)\ \rho\ =$
code $e_1\ \rho$;
$\vdots$
code $e_n\ \rho$;
invokestatic $c/m(\xi(t_1)\ldots\xi(t_k))\xi(t)$
where c is a class name. If c is not explicitly given we use the name of the current class instead.
For virtual methods we have to pass a reference of the object:
code $e.m(e_1, \ldots, e_n)\ \rho\ =$
code $e\ \rho$;
code $e_1\ \rho$;
$\vdots$
code $e_n\ \rho$;
invokevirtual $c/m(\xi(t_1)\ldots\xi(t_k))\xi(t)$
where c is the type infered for the expression e.

If there is no expression e, then we generate **aload** 0 to get a reference to the current object and let c be the name of the current class.

0.6 this

$code$ **this** ρ = **aload** 0

0.7 Assignments

In TassKaf we can assign values to local variables, object fields and class fields. For object and class fields the expression on the left of the assignment operator can be a qualified name, which contains a reference to the object or the name of a class.

$code(c.x = e)\ \rho{=}code\ e\ \rho;$ **putstatic** $\xi(c)/x\ \xi(t);$
where x is a class fields and t its type.

$code(e_1.x = e_2)\ \rho{=}code\ e_1\ \rho;code\ e_2\ \rho;$ **putfield** $\xi(c)/x\ \xi(t);$
where x is an object fields and $Type(e_1) = c$ and t the type of x.

$code(x = e)\ \rho{=}code\ e\ \rho;$ **istore** $\rho(x);$
where x is a local variable and $Type(e_1) \in \{int, bool\}$.

$code(x = e)\ \rho{=}code\ e\ \rho;$ **astore** $\rho(x);$
where x is a local variable and $Type(e_1)$ is a reference type (array or object).

In assignments to object fields of the current object, it is legal not to explicitly reference the object:

$code(x = e)\ \rho{=}$**aload** $0;code\ e\ \rho;$ **putfield** $\xi(c)/x\ \xi(t);$
where x is an object field of the current class c and t the type of x.

$code(x = e)\ \rho{=}code\ e\ \rho;$ **putstatic** $\xi(c)/x\ \xi(t);$
where x is a class field of the current class c and t the type of x.

The translation scheme for assignments to arrays is a little bit more complicated:

$code(e_1.x[d_1]\ldots[d_k] = e_2)\ \rho=$
  $code\ e_1\ \rho;$ **getstatic** $\xi(c_1)/x\ \xi(t);$
  $code\ d_1\ \rho;$ **aaload**;
  $\vdots$
  $code\ d_{k-1}\ \rho;$ **aaload**;
  $code\ d_k\ \rho;\ code\ e_2\ \rho;$ **aastore**;
where $Type(e_1) = c_1$ and $Type(e_2)$ is a reference type and t is the type of x.

For object variables we use **getfield** and for local variables **aload** instead of **getstatic**. If the type of the expression e_2 is *int* or *bool* we use **iastore** instead of **aastore**. Finally if the assignment is to an object field of the current object (i.e. there is no e_1), we use **aload** 0 instead of **getstatic**.