Projet JCas – Passe 3

Guillaume Fourret Benjamin Lecomte Ivan Baheux
Zoé Lagache

Novembre 2020

1 Messages d'erreur

- Débordement d'intervalle
  Cette erreur se produit lors de l'affectation d'une variable de nature Interval. En effet, nous devons vérifier si la valeur à affecter se trouve bien entre les bornes du type de la variable. Cela est vrai pour les affectations simples (avec une variable de nature Interval) et également pour les affectations composées (entre deux variables de nature Array où la nature de chaque élément est Interval).

- Débordement arithmétique
  Cette erreur peut se produire dans plusieurs cas : Lorsque que le programme attend une valeur par l'utilisateur et que celui-ci rentre une valeur incompatible. Lorsque le programme essaie de faire une division par zéro. Lorsque le programme effectue des opérations sur des entiers, pour vérifier que l'opération s'est bien déroulée et que la machine abstraite n'a pas détecté de débordement de valeurs.

- Débordement d'indice de tableau
  Cette erreur se produit lorsque le programme essaie d'accéder à un indice d'un tableau qui n'est pas compris dans les bornes de l'intervalle du tableau. Cela doit être vérifié dès qu'il y a un nœud Index, c'est-à-dire lorsqu'on accède à un élément d'un tableau.

- Débordement de pile
  Cette erreur se produit lors de la modification du pointeur de pile. En effet, il faut vérifier que le pointeur ne dépasse pas la taille maximale de la pile. On branche ainsi vers cette erreur lors de l'allocation de place en mémoire pour des variables permanentes ou temporaires.

2 Architecture

La passe 3 utilise 5 classes différentes :

- La classe principale du programme : JCasc, elle fait appel aux deux premières passes puis utilise la méthode coder() de la classe génération pour produire le fichier .ass exécutable par la machine abstraite.
- La classe Generation qui contient la majorité des méthodes de la passe 3. La méthode principale est coder() qui prend en argument l'arbre abstrait décoré défini dans la passe 2 et qui, par un parcours de cet arbre, génère des blocs d'instructions en code machine pour chaque nœud de l'arbre. La plupart des algorithmes des méthodes de cette classe sont détaillés dans la section suivante.
- La classe Erreur qui contient la liste des erreurs détaillées précédemment. Elle contient également une méthode ecrireErreurs() qui permet de rajouter les étiquettes correspondantes à toutes les erreurs qui seront utilisées par le programme à générer. Elle comporte également une méthode pour chaque erreur qui s'occupe de vérifier si l'erreur doit être lancée ou non. Par exemple, avant de réserver une place en pile, on vérifie si la pile déborde ou non, si c'est le cas, on saute vers l'étiquette associée à l'erreur, on affiche l'erreur et on arrête le programme.
- La classe Pile contient une HashMap contenant les chaînes de toutes les variables stockées en pile et l'adresse choisie pour chacune d'elles. Cette classe nous permet donc à la fois d'allouer des emplacements en pile pour toutes les variables permanentes définies au début du programme mais également de récupérer ces adresses pour manipuler les variables. On peut également allouer/libérer des variables temporaires en pile lorsque nous ne pouvons pas utiliser les registres.
- La classe Registres contient une HashMap et des méthodes pour garder à jour l'état des registres (alloué ou libre). Lorsque nous avons besoin d'un registre pour une opération, cela nous permet de récupérer un registre non utilisé ou de savoir qu'il n'en reste plus. Elle contient aussi une liste des registres privés que nous utilisons dans les méthodes de Generation : R0, R1, R2 .

3 Algorithmes utilisés

3.1 Coder_Pour

```
Coder_Pour(Pas, ListeInst)=
declare
E_Début: Etiq := Nouvelle_Etiq;
begin
```

```
Réserver(R0);
Réserver(R1);
Coder_EXP(Noeud(Fils2(Pas)),R0);
Coder_EXP(Noeud(Fils3(Pas)),R1);
Générer_Etiq(E_Début);
Coder_ListeInst(ListeInst);
Coder_EXP(Cond, R0);
Si (Noeud(Fils1(Pas)) = Increment) alors
    R0 = R0+1;
    Comparer (R0, R1);
    Générer(BGE,Etiq)
Si (Noeud(Fils1(Pas)) = Decrement) alors
    R0 = R0-1;
    Comparer (R0, R1);
    Générer(BLE,Etiq)
Libérer(R0);
Libérer(R1);
end;
```

3.2 Coder_Tanque

```
Coder_Tantque(Cond, ListeInst)=
declare
E_Cond : Etiq := Nouvelle_Etiq;
E_Début: Etiq := Nouvelle_Etiq;
begin
Réserver(R0);
Générer(BRA,E_Cond);
Générer_Etiq(E_Début);
Coder_ListeInst(ListeInst);
Générer_Etiq(E_Cond);
Coder_EXP(Cond, R0);
Si(R0 = 1) alors
    Brancher(E_Début);
Libérer(R0);
end;
```

3.3 Coder_Si

```
Coder_Si(Cond, Alors, Sinon)=
declare
E_Sinon : Etiq := Nouvelle_Etiq;
E_Fin : Etiq := Nouvelle_Etiq;
begin
Réserver(R0);
Coder_EXP(Cond, R0);
```

```
Si(Sinon = Vide) alors
    Si(R0 != 1) alors
        Brancher(E_Fin);
    Coder_Inst(Alors);
Sinon
    Si(R0 != 1) alors
        Brancher(E_Sinon);
    Coder_Inst(Alors);
    Générer(BRA,E_Fin);
    Générer_Etiq(E_Sinon);
    Coder_Inst(Sinon);
Générer_Etiq(E_Fin);
Libérer(R0);
end;
```

3.4 Coder_Exp

```
Coder_Exp(Arbre, Registre)=

begin
Selon Arité(Arbre) alors
"0" :
    Si (Valeur(Arbre) existe en pile) alors
        Charger(Pile(Valeur(Arbre)),Registre);
    Sinon
        selon Noeud(Arbre) alors
        "Ident" :
            Si (Arbre.chaine = "true") alors
                Charger(Int(1),Registre);
            Si (Arbre.chaine = "false") alors
                Charger(Int(0),Registre);
            Si (Arbre.chaine = "max_int") alors
                Charger(Int(javaInt.Max_Value),Registre);
        "Entier" :
            Charger(Int(Arbre.entier),Registre);
        "Réel" :
            Charger(Int(Arbre.reel),Registre);
"1" :
    Coder_EXP(Fils1(Arbre), Registre);
    Generer_Unaire(Arbre, Registre);
"2" :
    Si Arbre.Noeud = Noeud.Index alors
        Coder_Place(Arbre, Registre);
        Charger(Indexé(0,GB,Registre),Registre)
    Sinon
        Si il y a un registre libre alors
```

```
            Coder_EXP(Fils1(Arbre), Registre);
            Reserver(Registre2);
            Coder_EXP(Fils2(Arbre), Registre2);
            Generer_Binaire(Arbre, direct(Registre2),Registre);
            Liberer(Registre2);
        Sinon
            Coder_EXP(Fils2(Arbre), Registre);
            AdressePile = AllouerPile();
            Stocker(reg,Ind(AdressePile,GB));
            Coder_EXP(Arbre.fils1, Registre);
            Generer_Binaire(Arbre, Ind(AdressePile),Registre);
            LibererPile();

end;
```

3.5 Generer_Unaire

```
Generer_Unaire(Arbre, Registre)=
begin
Selon Noeud(Arbre) alors
    "MoinsUnaire" :
        GENERER(OPP, Registre, Registre);
        Verifier_Debordement_Arithmetique();
    "PlusUnaire" :
        Ne rien faire;
    "Non" :
        Charger(CMP, 0, Registre);
        Charger(SEQ, Registre);
    "Conversion" :
        Si Non(Nature(Noeud(Arbre)) = tableau) alors
            Conversion(Registre, Registre);
end;
```

3.6 Generer_Binaire

```
Generer_Binaire(Arbre, Operande, Registre)=
declare
operation : Operation;
begin
Selon Noeud(Arbre) alors
    "DivReel" :
        operation = DIV;
        Si(Nature(Fils2(Arbre)) = Interval) alors
            Conversion(Registre, Registre);
        Si(Nature(Fils1(Arbre)) = Interval) alors
            Si(Nature(Operande) = OpDirect) alors
```

```
            Conversion(Operande, Operande);
        Sinon
            Stocker(Operande, Addr_Memoire);
            Conversion(Operande, Registre);
            Stocker(Registre, Operande);
            Charger(Addr_Memoire, Registre);
"Moins" :
    operation = SUB;
"Plus" :
    operation = ADD;
"Quotient" :
    operation = DIV;
"Reste":
    operation = MOD;
"Mult":
    operation = MULT;

Si operation existe
    Générer(operation, Operande, Direct(Registre));
    Verifier_Debordement_Arithmetique();
Sinon
    Selon Arbre.noeud alors
        "Et" : #A ET B  (A+B)==2
            Générér(ADD, Operande, Direct(Registre));
            Verifier_Debordement_Arithmetique();
            Operande = Int(2);
            operation = SEQ;
        "Ou" : #A OU B  (A+B)>=1
            Générér(ADD, Operande, Direct(Registre));
            Verifier_Debordement_Arithmetique();
            Operande = Int(1);
            operation = SGE;
        "Inf" :
            operation = Inf;
        "InfEgal" :
            operation = InfEgal;
        "Sup" :
            operation = Sup;
        "SupEgal" :
            operation = SupEgal;
        "NonEgal" :
            operation = NonEgal;
        "Egal" :
            operation = Egal;
    Générer(CMP, Operande, Direct(Registre));
    Générer(operation, Direct(Registre));
```

```
end;
```

3.7 Coder_Affect

```
Coder_Affect(Place, Exp)=
declare
type : Type;
Debut_Affect : Etiq := Nouvelle_Etiq;
Fin_Affect : Etiq := Nouvelle_Etiq;
begin
Reserver(R0);
Reserver(R1);
type = Coder_Place(Place, R0);
Si(type = Tableau) alors
    Charger(Adresse(Exp), R1);
    Charger(tailleType(type)-1, R2);
    R2 = R2 + R1;
    Stocker(R2, Addr_Mémoire);
    Générer_Etiq(Debut_Affect);
    Si(R1 > Valeur(Addr_Mémoire)
        Brancher(Fin_Affect);
    Charger(Mémoire(R1), R2);
    Si (dernierType(type) = Interval) alors
        Verifier_Debordement_Intervall(dernierType(type), R2);
    Si (Noeud(Exp) = Conversion)
        Conversion(R2,R2);
    Stocker(R2, Memoire(R0));
    R0 = R0 + 1;
    R1 = R1 + 1;
    Brancher(Debut_Affect);
    Générer_Etiq(Fin_Affect);
Sinon
    Coder_EXP(Exp, R1);
    Verifier_Debordement_Intervalle(type, R1);
    Stocker(R1, R0);
Liberer(R0);
Liberer(R1);
end;
```

3.8 Coder_Place

```
Coder_Place(Arbre, Registre)=
declare
taille : Entier;
reg    : Registre;
```

```
begin
Si(Noeud(Arbre) = Ident) alors
    Charger(Adresse(Arbre), Registre);
    Retourne(Type(Arbre))
Si(Noeud(Arbre) = Index) alors
    Coder_Place(Fils1(Arbre), Registre);
    taille = tailleTableau(Fils1(Arbre));
    Si(Registre_Libre()) alors
        Reserver(reg);
        Coder_EXP(Fils2(Arbre), reg);
        Verifier_Debordement_Intervalle(Fils1(Arbre), reg);
        reg = reg-Type(Fils1(Arbre)).getIndice.getBorneInf();
        reg = reg * taille;
        Registre = Registre + reg;
        Liberer(reg);
    Sinon
        Stocker(Registre, Addr_Mémoire);
        Coder_EXP(Fils2(Arbre), Registre);
        Verifier_Debordement_Intervalle(Fils1(Arbre), Registre);
        Registre = Registre-Type(Fils1(Arbre)).getIndice.getBorneInf();
        Registre = Registre * taille;
        Registre = Registre + Valeur(Addr_Mémoire);
    Retourne(Type(Arbre));
end;
```

4 Jeu de tests

Tous les tests écrits pour la passe 3 se trouvent dans ProjetCompil/test/gencode/cas et avec le script gencode.sh génèrent des fichiers .ass exécutables par la machine abstraite dans ProjetCompil/test/gencode/ass. On peut y trouver à la fois des tests sur des fonctionnalités précises du langage ou des programmes plus complets implémentant des algorithmes connus. Nous avons rajouté les scripts de JaCoCo pour tester la couverture du code, nos tests nous permettent de couvrir environ 98% de notre code, en comptant que nous avons ajouté des instructions défensives qui ne devraient jamais être exécutées car elles concernent des programmes qui ne devraient pas passer les deux premières passes.