TD : Analyse Sémantique et génération de code trois adresses

2023-2024

Le but de ce TD est à la fois de faire de l'analyse sémantique (ici uniquement de la vérification de type, de portée et d'initialisation des variables), de la simplification d'expressions ou instructions constantes, et la génération de code trois adresses à partir d'un arbre de syntaxe abstraite. On travaillera également le rapport d'erreurs.

Votre travail sera à tester avec le seul exécutable, `analyser.out` généré qui vous permet d'activer et de désactiver les différentes passes. Vous avez à votre disposition un exécutable produit avec la solution, `analyser.solution` pour contrôler votre résultat – notamment en terme de rapport d'erreur et de simplifications. Il a été compilé au CREMI, il est fort probable qu'il ne fonctionne que là.

Le dossier qui vous est fourni reprend la structure que l'on avait au TD2 sur l'interpréteur. Il contient :

- Le dossier `programs` contient des exemples d'expressions, instructions et programmes, corrects ou incorrects, vous permettant de tester vos fonctions. Tous les codes incorrects devraient lever des erreurs ou avertissements dans le vérificateur de type.
- Le dossier `compiler` contient le code complet de notre interpréteur, contenant mes implémentations des TDs précédents, des modules utilitaires, ainsi que les trois fichiers que vous avez à compléter dans le cadre de ce TD:
  - `course_language/simplifier.ml` qui concerne le premier exercice.
  - `course_language/type_analyser.ml` qui concerne le second.
  - `three_address/course_language_to_three_address.ml` qui concerne le dernier.

Exercice 1: Simplification d'AST

Dans cet exercice on va commencer par le plus simple : la simplification de termes constants. L'exercice consiste à compléter le fichier `course_language/simplifier.ml`.

On peut voir cette passe de notre outil comme une compilation de notre langage dans lui-même, puisqu'il consiste à obtenir un AST à partir de l'AST d'origine tout en conservant la sémantique. C'est sans doute un peu abusif d'utiliser ce nom pour une passe de compilation, mais il me semble important d'insister sur le fait que conceptuellement, c'est la même chose.

L'idée de la simplification est de remplacer un AST complexe qui aurait toujours le même effet (valeur pour une expression) quelque soit le contexte par un arbre plus petit ayant le même effet (mais avec moins de calculs à l'exécution). Par exemple, si on écrit dans le code `x := 4*5-2;`, si on compile le code tel quel, toutes les exécutions passant par ce point devront calculer 4*5, puis y ajouter 2. Alors que le code `x := 18;` aurait exactement le même effet, mais sans avoir à recalculer 18.

Vous avez à compléter les fonctions `simplify_expr` et `simplify_instruction`. Vous pouvez constater qu'elles ne disposent d'aucun autre paramètre que le nœud de l'AST à simplifier. Ces deux fonction suivent un algorithme similaire :

- Commencer par simplifier tous les sous-AST du nœud courant (en effectuant des appels récursifs à ces deux fonction).
- Déterminer si l'arbre obtenu est un arbre que l'on peut simplifier.
- Si oui, renvoyer la simplification, qui portera l'annotation portée par le nœud d'origine.
- Si non, reconstruire le nœud à l'identique en remplaçant ses sous-arbres par leur simplification (calculée au premier point).

Les cas de simplification à considérer sont les suivants. Pour une expression :

- Si on a une `Binop(op,e1,e2,ann)`, et que `e1` et `e2` sont des constantes, le remplacer par la constante correspondant au résultat du calcul : `Cst_x(e1 op e2,ann)` (où x est selon les cas i, f ou b).
- Si on a une `Unop(op,e,ann)`, et que `e` est une constante, le remplacer par la constante correspondant au résultat du calcul : `Cst_x(op e,ann)`.
- Les autres expressions ne peuvent pas être simplifiées.

Pour une instruction :

- Si on a un `IfThenElse(t,i_then,i_else,ann)`. Si `t` est la constante `Cst_b(true)`, on le simplifie en `i_then`. Si `t` est la constante `Cst_b(false)`, on la simplifie en `i_else`, sinon, on ne la simplifie pas.
- Si on a un `While(t,body,ann)`, et que `t` est la constante `Cst_b(false)`, on le simplifie en `Block([],ann)`, sinon on ne la simplifie pas.
- Les autres cas ne sont pas simplifiable (mais n'oubliez pas de simplifier les sous-arbres).

Vous noterez qu'on cherche à conserver les annotation de l'arbre d'origine. Cela aura une importance à l'exercice suivant pour que les messages d'erreur correspondent bien au texte du code d'origine.

Exercice 2: Vérification de types

Dans cet exercice, on va vérifier les types utilisé dans le programme, en typant chaque expression, que les variables qui apparaissent sont bien déclarées, et que les variables utilisées ont bien été initialisées au préalable. Le but de cette passe est de rejeter des programmes potentiellement incorrect et d'en informer le programmeur, on s'efforcera donc de rapporter les erreurs rencontrées. Il s'effectue en remplissant le ficher `course_language/type_analyser.ml`.

Cette tâche peut être vue comme un interpréteur, si au lieu de considérer que les programmes définissent un effet sur l'état de mémoire d'une machine, on considère qu'ils calculent des types. Tout comme à l'exercice précédent, on peut considérer qu'il s'agit là d'un léger abus de langage, mais cela permet de comprendre que l'algorithme mis en jeu est assez similaire. En effet, tout comme dans le cas de l'interpréteur, on aura la présence d'environnements qui associent les variables à des valeurs (ici des types et le fait qu'elles

soient initialisées ou non), et les expressions auront une valeur (leur type, qu'on stockera dans l'annotation, ce qui est une légère différence).

On va ici faire une supposition qui nous simplifiera la tâche : on va supposer que dans une fonction donnée, toutes les déclarations de variables sont distinctes les unes des autres : si x est déclarée, elle ne sera pas redéclarée plus tard. Évidemment, le sens de cette supposition est qu'on pourrait tout à fait forcer cette propriété par une passe préalable (similaire à la simplification). Dans le but de limiter la taille des TDs, on ne vous demande pas de la faire, et on ne vous fournira que des programmes où c'est bien le cas.

Vous avez à compléter les fonctions `type_expr`, `type_instruction` et `type_func_decl` qui réalisent les tâches décritent plus loin sur les nœuds de l'AST correspondant (ainsi que `type_arg`, mais qui est plus une fonction technique). Elles disposent des mêmes arguments (à part que `type_func_decl` ne contient pas ceux parlant des variables locales d'une fonction). Ces arguments sont:

- `type_env`: un environnement (cf module `Util.Environment`) qui associe chaque variable locale d'une fonction à son type (`Ast.type_value`), si elle est définie.
- `init_env`: un environnement qui associe chaque variable locale d'une fonction à un booléen déterminant si elle est initialisée ou non.
- `func_env`: un environnement qui associe chaque nom de fonction à son type de fonction (`type_basic * (type_argument * type_value)list`, c'est-à-dire son type de retour et la liste des types de ses paramètres).
- `report`: un rapport d'erreur qui servira à stocker les erreurs et avertissements rencontrés. Le module définissant cette structure est `Util.Error_report`.
- Le nœud de l'arbre à analyser, dont le type dépend de la fonction (expression, instruction ou déclaration de fonction).

Toutes ces fonctions auront un fonctionnement similaire : elles commenceront par s'appeler récursivement pour chaque sous-nœud (quand il y en a), puis détermineront pour le nœud courant son type (si c'est pertinent), en utilisant si besoin (pour les variables) les environnements, en mettant à jour les environnements si c'est pertinent, et en reportant une erreur si une erreur est détectée.

Décrivons maintenant les tâches à réaliser. Premièrement, le typage des variables et des expressions. Cela ne concerne que les expressions, ainsi que les instructions de déclarations de variables et de tableaux. On va annoter les expressions et déclaration de fonction avec leur type. Plus précisément, pour chaque cas:

- Chaque déclaration de variable à de tableau affecte le type correspondant dans l'environnement `type_environnement` (similaire à une affectation dans la sémantique du langage).
- Chaque variable reçoit pour type le type de la variable stockée dans l'environnement `type_environnement`. Si aucun type n'est stocké, on reporte l'erreur, et on affecte le type `TNull`.
- Chaque `ArrayVal(x,p,ann)` a pour type le type stocké dans le tableau `x` si c'est bien un tableau. Sinon, une erreur est reportée, et on affecte le type `TNull`.
- Les constantes sont typées par leur type.

- Pour typer une opération unaire ou binaire, on type les sous-expressions, et si les deux types sont égaux et que l'opération est compatible avec ce type, on affecte le même type. Sinon, on reporte une erreur et on affecte le type `TNull`.
- `Size_tab(x,ann)` est de type `TInt`, mais si `x` n'est pas déclarée comme un tableau, une erreur est reportée.
- Pour typer un appel de fonction, si la fonction est déclarée dans `func_env`, le type est simplement le type de retour, sinon une erreur est reportée. On doit également typer chaque argument, en vérifiant que le type de l'argument correspond à celui stocké dans `func_env`, et que seules des variables sont passées par référence (et reporter des erreurs lorsque ce n'est pas le cas).
- Pour les affectation, il faut vérifier que le type de l'expression correspond à celui de la variable (ou tableau) affecté, et reporter une erreur si c'est le cas.
- Pour les blocs, les if-then-else et les while, il convient de tenir compte de la portée : les déclarations à l'intérieur de ces constructions disparaissent après. Aussi, avant de typer leur contenu, il convient de copier l'environnement de type (avec la fonction correspondante dans `Util.Environment`), de manière à ce que les déclaration n'aient pas d'effet visible à l'extérieur.
- Les return avec valeur vérifient que le type de l'expression renvoyée est le même que celui de la variable `#result`.
- Pour les instructions, il faut simplement typer les sous-expressions qu'elles contiennent.
- Les déclarations de fonctions typent leur corps à partir d'un environnement de types dans lequel leurs paramètres sont associés à leur types, et associent leur type de retour à la variable `#result`.

Ensuite, l'analyse d'initialisation concerne les affectations et les variables :

- Si on affecte une variable `x`, on lui associe `true` dans `init_env`.
- Si on utilise une variable (dans une expression), on reporte un avertissement si elle n'est pas associée à `true` dans `init_env`.
- Si on a un if-then-else, une variable n'est initialisée sûrement que si elle est affectée avant, ou dans les deux branches (il faut donc copier les environnements d'initialisation avant de typer les deux branches, et déterminer le résultats pour toutes les variables à la fin).
- Si on a un while, on ne peut être certain qu'une variable est initialisée (puisqu'on pourrait ne pas entrer dans la boucle).
- Les déclarations de fonctions évaluent leur corps sur un environnement d'initialisation où tous leurs paramètres sont associés à `true`.

Enfin, il faut typer et utiliser les déclarations de fonctions. Pour cela, on affecte les noms à leur type de fonction dans la fonction `type_decl_func`, et on vérifie leur type dans les expressions et instructions correspondantes.

Pour les erreurs, on va utiliser la structure `Error_report`. Elle contient une liste d'erreur et une liste de warning. Le but est de ne pas arrêter l'analyse à la première

erreur, mais de les stocker dans une structure qui permettra de les afficher à la fin de l'analyse. Les erreurs et les warning sont simplement un couple formé d'un message (à adapter en fonction du cas) et de la position à laquelle l'erreur est survenue (celle fournie par l'annotation du terme analysé). Elle contient les fonctions nécessaires pour ensuite afficher l'erreur dans son contexte.

On souhaite également ce que les erreurs redondantes ne soient reportées qu'une seule fois, et d'éviter des erreurs en cascade à cause d'une seule erreur. Notamment, les erreurs liées aux variables non-déclarée ou non-initialisée ne devraient être reportées qu'une fois par variable. Pour cela, on adopte une convention assez simple : une expression pour laquelle une erreur de type survient se voit associée le type `TNull` (qui n'est pas un vrai type), et toutes les erreurs impliquant ce type sont ignorées (elles sont provoquées par les erreurs précédentes). Pour l'initialisation, si on utilise une variable non-initialisée, on modifie l'environnement pour la mettre à initialisée de manière à ce que les prochaines utilisation ne reportent pas la même erreur.

Exercice 3: Traduction en code à trois adresses

Fichier `three_address/course_language_to_three_address.ml`.

Dans cet exercice, on va effectuer la traduction de notre langage (annoté par les types à la passe précédente) vers un code beaucoup plus bas niveau, du code trois adresses. Ce code est une version assez abstraite d'assembleur, définie dans `three_address/ast.mli`.

Vous pourrez contrôler votre résultat avec l'exécutable fourni (avec la bonne option), et vous pourrez voir son exécution (l'interpréteur est fourni).

Dans ce code, on conserve des constantes typées (int, float et bool). Cela faciliterait une traduction en assembleur par la suite (et l'interprétation). Les objets de base sont des adresses (qui représentent des cases mémoires, soit des registres du processeurs, soit celles sur la pile du programme) dans lesquelles on peut écrire, des `r_value` qui sont les valeurs pouvant être manipulées et qui sont des constantes ou des adresses. Il existe également des tableaux qui sont stockés ailleurs que dans la mémoire directement accessible (c'est-à-dire qu'ils sont stockés dans le tas). On ne peut que lire et écrire dedans et non utiliser directement les valeurs qu'ils contiennent dans les opérations. Les instructions sont les suivantes :

- Des instructions correspondant à des opérations unaires ou binaires sur des `r_value` qui stockent le résultat dans une adresse.
- Des instructions de copies entre adresses et entre adresses et cases de tableaux.
- Une instruction de réservation de mémoire pour un tableau (correspondant à un malloc).
- Des instructions de saut, inconditionnels ou dépendant de la valeur d'une `r_value`.
- Des instructions concernant les fonctions, une préparant une paramètre, une effectuant l'appel et une retournant à la position de l'appel précédent. Elles correspondent à la manipulation de fonctions qui seraient faite en assembleur.
- Une instruction ne faisant rien.
- Des instructions d'affichages (qui correspondent à des appels de fonctions bas niveau particulières qui seraient traduites vers des appels dans la traduction vers l'assembleur).

Chaque instruction est porteuse d'un label (potentiellement le label vide), qui sont à la fois la cible des sauts et des appels de fonctions.

Les labels et les adresses sont ici des types abstraits qui doivent être générés avec les fonctions fournies.

Vous devez traduire un AST annoté en une séquence d'instructions 3 adresses étiquetées. Vous pouvez commencer, comme d'habitude, par les expressions (le programme permet de ne tester que cela) – bien que vous ne pourrez pas avoir d'affichage de cette exécution.

Pour cette traduction, les fonctions prennent comme arguments, en plus de l'AST à traduire, deux structure pour vous aider à gérer les adresses et les labels. L'argument `counter` vous permet de générer au besoin des adresses et des labels qui n'existent pas déjà dans le code généré. L'argument `mapper` vous permet de gérer les adresses des arguments des fonctions de manière à ne pas avoir à connaître leur nom exact : lorsque vous utilisez un nom du programme, il faut récupérer son adresse grâce à `mapper`. Lors d'une déclaration de fonction, il faut indiquer à `mapper` les noms des arguments dans l'ordre où ils apparaissent.

1. Commencez par traduire les expressions (sans fonctions). Cela consiste essentiellement à découper les expressions en calculs simples en stockant les valeurs intermédiaires dans des adresses fraîches.

2. Passez ensuite aux instructions. Les affectations et le bloc ne contiennent pas de difficulté. Les déclarations de tableau non plus (les tailles de tableaux sont affectées par l'interpréteur). Les principales difficultés sont sur le `IfThenElse` et le `While` : il faut traduire les branches et le test, puis, en plaçant des labels au bon endroit (au début et à la fin des codes des branches et du test) mettre les sauts conditionnels pertinents. Pour le `Print_expr`, il faut utiliser l'annotation de type de l'expression affichée pour déterminer quelle instruction utiliser.

3. Enfin, vous pourrez vous attaquer à la gestion des fonctions qui est un peu plus complexe. Les déclarations de fonctions consistent essentiellement en un label portant le nom de la fonction précédent la traduction du corps de la fonction, mais elles doivent avant de traduire le corps de la fonction construire un `mapper` avec les paramètres. Il faut également penser à terminer la fonction par un `Return` (au cas où la fonction ne termine pas elle-même par un `Return`). Pour les appels de procédures et de fonctions, il faut d'abord évaluer les arguments et les empiler grâce à l'instruction `Param`. Attention, l'évaluation et l'empilement doivent se faire par la droite, ce qui permet d'identifier correctement les arguments de manière relative au sommet de la pile de paramètres. Ainsi, le premier argument d'un appel est le premier à être empiler. Il faut également faire attention à traiter différemment les paramètres par valeur et par argument : si vous utilisez `Param add`, l'adresse `add` sera modifiée par l'appel, donc si vous faites un passage par valeur, il faut copier au préalable cette valeur. Une fois les paramètres empilés, il reste simplement à réaliser l'appel avec l'instruction `Call`.

Une fois tout cela fait, vous devriez être en mesure de produire un code trois adresses avec le même comportement que l'AST dont il est issu.

La suite des évènements consisterait à optimiser ce code et à le traduire en assembleur, notamment en déterminant par rapport au processeur les registres et zones mémoires exactes, mais c'est une tâche qu'on n'effectuera pas dans ce cours, par manque de temps.