APS - Evaluateur de mini-Pascal

Jordi Bertran de Balanda

Résumé

Utilisation

Des fichiers de test syntaxiquement corrects et bien typés sont disponibles dans le souds-dossier `refs`.

- Parser :
  - `$ make parser`
- Génération d'AST et du fichier de typage
  - `$ make typing`
  - `$ ./type <fichier>`
    - Génère le code Prolog correspondant dans `tmptypes.pl`
- Typage d'un programme
  - `$ swipl typechecker.pl`
  - `?- [tmptypes].`
  - `?- typeProg().`
    - Prolog renvoie faux si le programme est mal typé.
- Génération d'AST et évaluation :
  - `$ make eval`
  - `$ ./eval <fichier>`

Arbre de syntaxe abstrait

Pour simplifier le parser, éviter d'avoir une union de types énorme dans celui-ci et réduire le nombre de tokens à typer différemment dans l'en-tête, tous les noeuds de l'AST sont de type AST, une union entre les différents types concrets de noeuds – expression, déclaration, etc... Le désavantage de la couche d'indirection supplémentaire est contrebalancé par la facilité de modification du contenu de la structure à l'écriture.

Typage

Génération du code Prolog

Le code Prolog est généré par écriture dans un fichier avec la seule fonction rendue visible par `prolog_gen.h`, qui propose de fournir un file handle dans lequel écrire le code Prolog correspondant au `Programme *` passé en argument.

On aurait éventuellement pu récupérer à partir du parser le nombre exact de caractères nécessaires à l'écriture du programme de typage Prolog pour se passer de fichier et ainsi gérer directement le typage en C (avec un appel à `pipe` pour ouvrir une instance de SWIPL dans laquelle on lirait le résultat du typage par exemple), mais ceci nécessite de coder en dur un nombre important de correspondances entre les noeuds de l'AST et le code Prolog généré.

En théorie, pour être cohérent avec la logique du typage de *tous* les noeuds de l'AST en `AST`, on devrait utiliser un `switch` sur le type interne du noeud dans une fonction générale de gestion de l'AST pour aiguiller l'opération à faire. En pratique, ceci rend le code difficile à lire, et on préfère donc aiguiller manuellement sur fonctions de gestion des types internes, ce qui est gérable vu le faible nombre de types internes. Cette remarque est également valable pour le code de l'évaluateur.

Typage du programme

Le typeur lui-même ne contient que 2 règles différentes:

- type(instruction, environnement, type)
- typeNamed(instruction, environnement, nouvel environnement, type)

L'environnement est ici représenté sous la forme d'une liste de triplets tr(nomVar, const, type). Le suivi de trois valeurs dans l'environnement au lieu de deux nous permet de détecter au typage les tentatives d'accès en écriture à des variables constantes.

Evaluation

L'évaluateur suppose le programme correctement typé après la passe de parsing et de typage prolog. On vérifie quand même que les types concordent lors de l'évaluation, notamment avec l'opérateur `default` des `switch` qui permet de repérer quand une valeur d'énumération ne concorde pas avec les valeurs attendues. Ceci nous prémunit aussi contre les malformations de l'AST, sans coût supplémentaire. Les erreurs de confusion entre variable et constante ainsi que les tentatives d'accès en lecture à une variable non définie sont également signalées par l'évaluateur, qui les traite comme des erreurs fatales.

Valeurs

Par soucis de séparation des étapes, toutes les énumérations qu'on peut retrouver à la fois dans la génération de l'AST (ast.x) et dans l'évaluation du programme – par exemple l'énumération des types primitifs – sont séparées, avec une énumération côté AST (T_X) et une côté évaluateur (EVAL_X).

Par soucis de clarté de relecture, même si il est possible de gérer toutes nos valeurs primitives (entiers et booléens) avec des `int` C, on préfère boxer nos types. Ceci améliore aussi la facilité d'ajout de nouveaux types primitifs, puisqu'il suffit alors de les rajouter dans la structure de boxing et dans les `switch` traitant les types primitifs dans l'évaluateur.