Ημερομηνία Παράδοσης Άσκησης: 20/02/2023, 23:59μμ
Αρχές Γλωσσών Προγραμματισμού

Υλοποίηση μίας Lazy Συναρτησιακής Γλώσσας Πρώτης Τάξης

Σκοπός της άσκησης αυτής είναι η υλοποίηση ενός διερμηνέα για μία απλή, lazy συναρτησιακή γλώσσα προγραμματισμού πρώτης τάξης, που στη συνέχεια θα αναφέρεται ως MiniHaskell. Η γλώσσα αυτή πρέπει να ακολουθεί τις εξής προδιαγραφές:

1. Ένα πρόγραμμα της MiniHaskell είναι ένα σύνολο από ορισμούς (ακριβώς όπως στην Haskell). Ο πρώτος από αυτούς τους ορισμούς θα πρέπει να ορίζει την μεταβλητή result, η οποία θα "επιστρέφει" το τελικό αποτέλεσμα του προγράμματος. Όλοι οι υπόλοιποι ορισμοί ορίζουν συναρτήσεις που δέχονται μηδέν (π.χ. $f() = 5$) ή περισσότερες (π.χ. $f(x, y) = x + y * 2$) *τυπικές παραμέτρους*.

2. Κάθε σύμβολο συνάρτησης ορίζεται μόνο μία φορά σε ολόκληρο το πρόγραμμα.

3. Όλες οι τυπικές παράμετροι μίας συνάρτησης είναι διαφορετικές μεταξύ τους και διαφορετικές από όλες τις υπόλοιπες τυπικές παραμέτρους των υπόλοιπων συναρτήσεων.

4. Η MiniHaskell υποστηρίζει μόνο ακέραιες και Boolean εκφράσεις. Επιπλέον, επιτρέπει τα ακόλουθα: if-then-else, πρόσθεση, αφαίρεση, πολλαπλασιασμό, ακέραια διαίρεση, τελεστές σύγκρισης (`==`, `/=`, `>`, `>=`, `<`, `<=`), Boolean τελεστές (`&&`, `||`, `not`) και μοναδιαίους τελεστές (`-`, `+`).

5. Κάθε ορισμός τερματίζεται με ερωτηματικό.

Ένα παράδειγμα προγράμματος της MiniHaskell είναι το εξής:

```
result  = fib(10) + fact(8);
fib(n)  = if (n<=1) then 1 else fib(n-1) + fib(n-2);
fact(m) = if (m<=1) then 1 else m*fact(m-1);
```

Η υλοποίηση ενός διερμηνέα για την γλώσσα μπορεί να επιτευχθεί ακολουθώντας τα εξής βήματα:

1. Στην αρχική φάση, θα γίνεται έλεγχος του συντακτικού του προγράμματος και θα αναφέρονται πιθανά συντακτικά λάθη. Αυτή η διαδικασία είναι γνωστή ως *parsing*. Για διευκόλυνση, σας δίνεται έτοιμος ένας parser για την MiniHaskell, οπότε δεν απαιτείται κάποια υλοποίηση για αυτό το βήμα.

2. (75%) Αν το πρόγραμμα περάσει τον συντακτικό έλεγχο, τότε στην δεύτερη φάση θα πρέπει να μετατραπεί σε *νοηματικό κώδικα*, ο οποίος περιγράφεται στην επόμενη ενότητα. Για παράδειγμα, το προηγούμενο πρόγραμμα θα μετατραπεί σε νοηματικό κώδικα ως εξής:

   ```
   result = call_0 fib + call_0 fact;
   fib    = if (n<=1) then 1 else call_1 fib + call_2 fib;
   n      = actuals(10,n-1,n-2);
   fact   = if (m<=1) then 1 else m * call_1 fact;
   m      = actuals(8,m-1);
   ```

3. (25%) Στην τελική φάση, ο νοηματικός κώδικας εκτελείται και παράγεται το ζητούμενο αποτέλεσμα, σύμφωνα με την περιγραφή που δίνεται στην ενότητα "Μηχανισμός Εκτέλεσης".

Νοηματικός Κώδικας

Η τεχνική υλοποίησης που παρουσιάζεται παραπάνω είναι αρκετά συχνή στον κόσμο των Γλωσσών Προγραμματισμού. Συγκεκριμένα, αντί να εκτελείται το πηγαίο πρόγραμμα απευθείας, πρώτα μεταφράζεται σε πρόγραμμα μίας άλλης γλώσσας (intermediate representation) και στην συνέχεια εκτελείται αυτό το πρόγραμμα. Ένα βασικό πλεονέκτημα αυτής της προσέγγισης είναι ότι η ενδιάμεση γλώσσα είναι συχνά πιο απλή, με αποτέλεσμα να διευκολύνεται η εκτέλεση του προγράμματος, ή/και πιθανοί μετασχηματισμοί βελτιστοποίησης πάνω σε αυτό.

Η ενδιάμεση γλώσσα που έχει επιλεγεί για την υλοποίηση της MiniHaskell λέγεται *νοηματικός κώδικας* (intensional code). Η γλώσσα αυτή μοιάζει με την MiniHaskell, αλλά έχει την ιδιότητα ότι υποστηρίζει μόνο συναρτήσεις μηδενικού βαθμού, δηλαδή μεταβλητές. Αυτό το πετυχαίνει εισάγοντας δύο τελεστές: τον τελεστή *actuals*, ο οποίος χρησιμοποιείται για την κωδικοποίηση των τιμών που λαμβάνουν οι τυπικές παράμετροι στις κλήσεις συναρτήσεων του πηγαίου προγράμματος, και τον τελεστή *call*, ο οποίος χρησιμοποιείται για την αντικατάσταση των κλήσεων συναρτήσεων στο πηγαίο πρόγραμμα με "κλήσεις" των αντίστοιχων μεταβλητών.

Στο πλαίσιο αυτής της άσκησης, καλείστε να υλοποιήσετε τον ακόλουθο αλγόριθμο μετατροπής ενός πηγαίου προγράμματος MiniHaskell σε νοηματικό κώδικα:

1. Έστω f μία συνάρτηση που ορίζεται στο πηγαίο πρόγραμμα. Απαριθμούμε όλες τις εμφανίσεις κλήσεων της f, από αριστερά προς τα δεξιά και από πάνω προς τα κάτω, ξεκινώντας από το 0 (συμπεριλαμβανομένων και των κλήσεων στον ορισμό της f).

2. Αντικαθιστούμε την i-οστή κλήση της f με $call_i\ f$ και αφαιρούμε τις τυπικές παραμέτρους στον ορισμό της, ώστε να μετατραπεί σε μεταβλητή.

3. Εισάγουμε έναν νέο ορισμό για κάθε τυπική παράμετρο της f, στο δεξί μέλος του οποίου χρησιμοποιείται ο τελεστής *actuals*, εφαρμοσμένος σε μία λίστα που περιέχει όλες τις τιμές που λαμβάνει η εν λόγω παράμετρος, σύμφωνα με την σειρά αρίθμησης των κλήσεων της f και αφού πρώτα μετασχηματιστούν οι υπο-κλήσεις στις αντίστοιχες *call* εκφράσεις.

Για να γίνει αντιληπτή η λειτουργία του αλγορίθμου, ας τον εφαρμόσουμε στο παρακάτω πρόγραμμα:

```
result = f(f(4)) + f(5);
f(x)   = g(x+1);
g(y)   = y;
```

1. Σύμφωνα με το βήμα (1) του αλγορίθμου, απαριθμούμε όλες τις εμφανίσεις κλήσεων των συναρτήσεων f και g, ως εξής:

```
          0 1       2
result = f(f(4)) + f(5);

          0
f(x)   = g(x+1);
g(y)   = y;
```

   Παρατηρήστε ότι οι κλήσεις κάθε συνάρτησης αριθμούνται ξεχωριστά από τις κλήσεις των υπόλοιπων συναρτήσεων.

2. Στην συνέχεια, σύμφωνα με το βήμα (2), πρέπει να αντικαταστήσουμε κάθε κλήση των f και g με $call_i$ f και $call_i$ g, όπου i είναι ο αντίστοιχος αριθμοδείκτης της κλήσης. Κατόπιν, αφαιρούμε όλες τις τυπικές παραμέτρους από τους αντίστοιχους ορισμούς:

```
result = call_0 f + call_2 f;
f      = call_0 g;
g      = y;
```

3. Τέλος, σύμφωνα με το βήμα (3), εισάγουμε τους εξής ορισμούς για τις παραμέτρους x και y, καταλήγοντας στην τελική μορφή του προγράμματος σε νοηματικό κώδικα:

```
result = call_0 f + call_2 f;
f      = call_0 g;
g      = y;
x      = actuals(call_1 f,4,5);
y      = actuals(x+1);
```

   Παρατηρήστε ότι η κεφαλή της λίστας $actuals$ της τυπικής παραμέτρου x (η πρώτη πραγματική τιμή της παραμέτρου, σύμφωνα με την προηγούμενη αρίθμηση) είναι η έκφραση $call_1$ f, και όχι η έκφραση $f(4)$. Ο λόγος που γίνεται αυτός ο μετασχηματισμός θα φανεί στα παραδείγματα εκτέλεσης, στο Παράρτημα Γ'.

Μηχανισμός Εκτέλεσης

Αφού μεταφραστεί το πηγαίο πρόγραμμα σε νοηματικό κώδικα, μπορεί πλέον να εκτελεστεί για να παράξει το επιθυμητό αποτέλεσμα. Για αυτό τον σκοπό, πρέπει αρχικά να ορίσουμε την σημασιολογία των τελεστών $call$ και $actuals$. Το ακόλουθο μοντέλο αποδεικνύεται ότι είναι επαρκές, για την αποτίμηση νοηματικών προγραμμάτων:

- Ο τελεστής $call_i$ επαυξάνει μία λίστα w, προσαρτώντας το i στην κεφαλή της: $(call_i\ a)(w) = a(i : w)$.
- Ο τελεστής $actuals$ παίρνει τον αριθμοδείκτη i στην κεφαλή μίας λίστας, και τον χρησιμοποιεί για να επιλέξει το i-οστό της στοιχείο: $(actuals(a_0, ..., a_{n-1}))(i : w) = (a_i)(w)$.

Συνδυάζοντας τη σημασιολογία των $call$ και $actuals$ με τους κανόνες αποτίμησης εκφράσεων που φαίνονται παρακάτω, μπορούμε πλέον να εκτελέσουμε ένα πρόγραμμα σε νοηματικό κώδικα.

- EVAL(E1 BinaryOp E2, tags) = EVAL(E1, tags) BinaryOp EVAL(E2, tags)
- EVAL(if B then E1 else E2, tags) = if EVAL(B, tags) then EVAL(E1, tags) else EVAL(E2, tags)
- EVAL(constant, tags) = constant
- EVAL((E1), tags) = EVAL(E1, tags)
- EVAL(UnaryOp E, tags) = UnaryOp EVAL(E, tags)

Στο Παράρτημα Γ' θα βρείτε ένα σύνολο παραδειγμάτων εκτέλεσης, για να γίνουν σαφείς οι παραπάνω κανόνες.

Πλαίσιο Υλοποίησης

Σε αυτή την άσκηση καλείστε να υλοποιήσετε τον αλγόριθμο μετάφρασης προγραμμάτων MiniHaskell σε νοηματικό κώδικα, καθώς επίσης κι έναν διερμηνέα προγραμμάτων νοηματικού κώδικα. Επιπλέον, θα πρέπει να αναφέρετε και να τεκμηριώσετε σύντομα τις όποιες σχεδιαστικές σας επιλογές σε ένα αρχείο README.

Για να χρησιμοποιήσετε τον parser που σας δίνεται, πρέπει να έχετε εγκατεστημένο τον μεταγλωττιστή Glasgow Haskell Compiler (ghc) και τις βιβλιοθήκες parsec, pretty-simple. Αυτά μπορείτε να τα εγκαταστήσετε ως εξής:

1. Εγκατάσταση του installer GHCup, ο οποίος αναλαμβάνει την εγκατάσταση του μεταγλωττιστή και λοιπών χρήσιμων πακέτων:

   ```
   curl --proto '=https' --tlsv1.2 -sSf https://get-ghcup.haskell.org | sh
   ```

   Για επιπλέον πληροφορίες, μπορείτε να επισκεφτείτε την επίσημη σελίδα του GHCup στον ακόλουθο σύνδεσμο: `https://www.haskell.org/ghcup`. Σημειώνεται ότι η παραπάνω εντολή μπορεί να εκτελεστεί στα περιβάλλοντα Linux, MacOS και WSL2.

2. Εγκατάσταση των βιβλιοθηκών parsec και pretty-simple:

   ```
   cabal install --lib parsec
   cabal install --lib pretty-simple
   ```

   Πιθανώς, θα χρειαστεί να κάνετε expose στον ghc τα πακέτα parsec, pretty-simple και mtl, ώστε να μπορέσει να τα βρει κατά την διαδικασία της μεταγλώττισης / σύνδεσης. Αυτό μπορεί να γίνει με τις ακόλουθες εντολές:

   ```
   ghc-pkg expose parsec
   ghc-pkg expose pretty
   ghc-pkg expose mtl
   ```

Επιπλέον, προκειμένου να διευκολυνθεί η διόρθωση της εργασίας, προτείνεται να χρησιμοποιήσετε τους τύπους που έχουν οριστεί στο αρχείο Types.hs. Οι τύποι αυτοί μπορούν να αναπαραστήσουν πλήρως τα προγράμματα σε MiniHaskell και σε νοηματικό κώδικα.

Τέλος, συμπληρωματικά ως προς την εκφώνηση, μπορείτε να μελετήσετε και τα σχετικά άρθρα που υπάρχουν διαθέσιμα στην σελίδα του μαθήματος. Για οποιαδήποτε απορία, παρακαλούμε απευθυνθείτε μέσω email στον διδάσκοντα του μαθήματος, ή σε κάποιον από τους συνεργάτες, στις παρακάτω διευθύνσεις:

prondo [at] di.uoa.gr - Παναγιώτης Ροντογιάννης
sdi1900006 [at] di.uoa.gr - Νικόλαος Αλεξανδρής
sdi1800105 [at] di.uoa.gr - Χαράλαμπος Μαραζιάρης
sdi1800179 [at] di.uoa.gr - Γεώργιος Σίττας
sdi1800201 [at] di.uoa.gr - Κωνσταντίνος Τσικούρης

Παράρτημα Α: Συντακτική Αναγνώριση

Στο αρχείο Parser.hs βρίσκονται ορισμένες οι συναρτήσεις που χρησιμοποιούνται για την συντακτική ανάλυση των προγραμμάτων σε MiniHaskell, οι οποίες μπορούν να χρησιμοποιηθούν όπως φαίνεται στην συνέχεια:

```
$ cat ./test.hs
result = f(5);
f(x) = x + 1;
```

```
ghci> :set -package mtl
package flags have changed, resetting and loading new packages...
ghci> :l Parser.hs
[1 of 2] Compiling Types             ( Types.hs, interpreted )
[2 of 2] Compiling Parser            ( Parser.hs, interpreted )
Ok, two modules loaded.
ghci>
ghci> -- Same result as: parseFile "./test.hs"
ghci> programParser "result = f(5); f(x) = x + 1"
Right (FCall "f" [FNum 5],[("f",["x"],FBinaryOp Plus (FVar "x") (FNum 1))])
ghci>
ghci> -- Same result as: parsePrettyFile "./test.hs"
ghci> programPrettyParser "result = f(5); f(x) = x+1;" -- User-friendly formatting
Right
    ( FCall "f"
        [ FNum 5 ]
    ,
        [
            ( "f"
            , [ "x" ]
            , FBinaryOp Plus
                ( FVar "x" )
                ( FNum 1 )
            )
        ]
    )
```

Το αποτέλεσμα των παραπάνω συναρτήσεων είναι, επί της ουσίας, μία αναπαράσταση του αρχικού προγράμματος, που λέγεται Αφηρημένο Συντακτικό Δέντροε (Abstract Syntax Tree - AST). Η αναπαράσταση αυτή είναι αρκετά εκφραστική, ώστε να μπορεί να διατηρήσει πλήρως την συντακτική δομή του αρχικού προγράμματος. Μπορείτε να βρείτε περισσότερες πληροφορίες για την συγκεκριμένη αναπαράσταση στον ακόλουθο σύνδεσμο: `https://en.wikipedia.org/wiki/Abstract_syntax_tree`.

Παράρτημα Β: Μετάφραση σε Νοηματικό Κώδικα

1. MiniHaskell:

```
result = f(f(f(1)));
f(x) = x + 2;
```

   Intensional:

```
result = call_0 fib;
f      = x + 2;
x      = actuals(call_1 f, call_2 f, 1);
```

2. MiniHaskell:

```
result = f() + 5;
f()    = 3;
```

   Intensional:

```
result = call_0 f + 5;
f      = 3
```

3. MiniHaskell:

```
result = f(3, 5);
f(x,y) = x - (2 * y);
```

   Intensional:

```
result = call_0 f;
f      = x - (2 * y);
x      = actuals(3);
y      = actuals(5);
```

4. MiniHaskell:

```
result = f(f(1) + f(2) + g(3));
g(n) = f(n);
f(x) = x + 2;
```

   Intensional:

```
result = call_0 f;
g      = call_3 f;
f      = x + 2;
n      = actuals(3);
x      = actuals(call_1 f + call_2 f + call_0 g, 1, 2, n);
```

Παράρτημα Γ: Εκτέλεση Νοηματικού Κώδικα

1. Intensional:

```
result = call_0 f + call_2 f;
f      = call_0 g;
g      = y;
x      = actuals(call_1 f,4,5);
y      = actuals(x+1);
```

Evaluation:

```
  EVAL(result, [])
= EVAL(call_0 f + call_2 f, [])
= EVAL(call_0 f, []) + EVAL(call_2 f, [])
= EVAL(f, [0]) + EVAL(f, [2])
= EVAL(call_0 g, [0]) + EVAL(call_0 g, [2])
= EVAL(g, [0,0]) + EVAL(g, [0,2])
= EVAL(y, [0,0]) + EVAL(y, [0,2])
= EVAL(actuals(x+1), [0,0]) + EVAL(actuals(x+1), [0,2])
= EVAL(x+1, [0]) + EVAL(x+1, [2])
= EVAL(x, [0]) + EVAL(1, [0]) + EVAL(x, [2]) + EVAL(1, [2])
= EVAL(actuals(call_1 f,4,5), [0]) + 1 + EVAL(actuals(call_1 f,4,5), [2]) + 1
= EVAL(call_1 f, []) + EVAL(5, []) + 2
= EVAL(f, [1]) + 5 + 2
= EVAL(call_0 g, [1]) + 7
= EVAL(g, [0,1]) + 7
= EVAL(y, [0,1]) + 7
= EVAL(actuals(x+1), [0,1]) + 7
= EVAL(x+1, [1]) + 7
= EVAL(x, [1]) + EVAL(1, [1]) + 7
= EVAL(actuals(call_1 f,4,5), [1]) + 1 + 7
= EVAL(4, []) + 8
= 4 + 8
= 12
```

2. Intensional:

```
result = call_0 f;
f      = x + 1;
x      = actuals(call_1 f, call_2 f, 4);
```

Evaluation:

```
  EVAL(result, []) = EVAL(call_0 f, []) = EVAL(f, [0])
= EVAL(x + 1, [0]) = EVAL(x, [0]) + EVAL(1, [0])
= EVAL(actuals(call_1 f,call_2 f,4), [0]) + 1
= EVAL(call_1 f, []) + 1 = EVAL(f, [1]) + 1 = EVAL(x+1, [1]) + 1
= EVAL(x, [1]) + EVAL(1, [1]) + 1 = EVAL(actuals(call_1 f,call_2 f,4), [1]) + 2
= EVAL(call_2 f, []) + 2 = ... = 7
```