Exercise Compiler Construction (10)

Hans de Nivelle

Due: 15.01.2014

We are going to implement the type checking algorithm in the slides **typechecking.pdf**. It is easy to implement a stack machine, and we have essentially done that in the previous exercises by implementing a LISP interpreter. As soon as you want a more than just a stack machine, there is a very steep curve: To get only a little bit more costs very much additional effort. We are going to try anyway. Download the file `intermediate2013.tar.gz` from the course homepage. It contains various classes:

- **ctype**: A class representing C-types, as described in **compilation.pdf**, starting at page 15.
- **ctree**: A simple tree class that represents trees obtained by parsing. A tree is either
  - Concrete data, represented by a binary string `std::vector<char>`.
  - An identifier, represented by an **identifier**.
  - A functional tree, represented by an **identifier** (the function) and its subtrees.

  In addition, a **ctree** has a field **ct** of type **ctype**, which can be used for storing the type of the tree.
- **structstore**: Stores type definitions of structs. (what fields they have, and what types the fields have.)
- **functionstore**: Stores definitions of concrete functions (and possibly of global variables).
- **varstore**: Stores types of local variables.

The goal is to implement the type checking algorithm on slides 15, etc. in **typechecking.pdf**. The type checking algorithms should check a **ctree**, using a **varstore**, a **structstore** and a **functionstore**. It should return a tree, in which all subtrees have the field **ct** filled in, and with the implicit conversions filled in.

1. Add some more functions to the functionstore in **intermediate.cpp** for example

```
double dotproduct( const vector& , const vector& );
vector crossproduct( const vector& , const vector & );
double length( const vector& );

int first( const list& l );
int sum( const list& l );
int size( const list& l );
```

Don't add copy constructors, assignment operators, etc. Also don't add primitive operators on basic datatypes. I think that these need to be hard coded into the type checking algorithm.

2. Write a simple class **conversion** that essentially contains a **ctree**, and an **unsigned int**, that represents a conversion of some term, and the cost that was required to obtain it.

3. Write a class **possibleconversions** that contains all possible conversions of a given term, and the cost that was required to obtain them. I wouldn't bother to add any indexing, just a simple list or vector of conversions is enough.

   A member of **possibleconversions** is constructed with a single typed **ctree**.

   The class must have a method **close( )** that constructs all possible conversions (with their associated cost). Function **close**() should be able to deal with possible circular conversions.

   Method **close( )** must take the following into account:

   - Implicit conversion between data types, as in **bool** → **char** → **int** → **double**.
   - Copy constructors, box operators, read/write operators.
   - Pointer operators `&` and `*`. These operators are polymorphic!

4. Write the function

```
ctree typecheck( ctree t, varstore& local, structstore& st,
                          functionstore& fs );
```

   Typechecking must take into account field functions, and it must replace implicit operators on primitive types by their concrete instances.

Grading
Grading will be based on the following aspects:

- completeness of task: Is the algorithm completely implemented? Are all types and conversions of page 54 (**compilation.pdf**) present?

- general quality of the design: Is there a good class design?
- coding quality? Is the code readable?