Documentation

Overview

This research aims to apply the Algebraic data types for the language C which supports the pattern matching, type checker scheme and a tricky tag scheme. Support for algebraic data types in C via macros (mostly) with minimal runtime overhead, compact data representation (tagged pointers) and type+other errors checked by built in checking done by C compiler (ie, we avoid implementing a parser for extended C + our own type checker etc, so the design is somewhat constrained). The executable ADT file named "adtpp", by which can parse the .adt file and generate the corresponding header file. For instance, type "adtpp foo.adt" will create the "foo.h" file and it can be included in your C codes.

Environment Requirement(Basic):

1. Gcc version 4.0.0
2. Bison version 2.4.2
4. Flex version 2.5.35
5. Make 3.81

Installation instruction:

1. unzip the archive file, go into "src" directory
2. type "make"
3. type "make install"

Commands:

1. make (all): compile all source codes to generate the executable file "adtpp"
2. make install: copy the "adtpp" file into local "/bin/"
3. make dist: zip all source codes to a .zip file
4. make test: kind of the bootstrapping checking for correctness. It would be used to compare the generated file with original header file.
5. make clean: clean up all generated files

File Reference:

1. adtpp: the ADT tool which is generated after the command"make"
2. adt_master.h: the source header file which helps to created ADT tool
3. parser.l & parser.y: the Yacc & Lex tool which is employed to parse the .adt file as input
4. adt_master.adt: the original .adt file which creates the source header file
5. adtpp.c: the main C version source to generate ADT tool

6. adt_t.adt: a brief example of ADT file

ADT syntax:

A keyword “data” should lead the start of definition, and the type name should be given. The content of this type should be involved in the couple of open and close braces, which is similar with the function definition in C. Each type constructor in the definition is formed by its constructor name and a number of (or none of) types, which are involved in the pair of open and close parentheses. These types can be a primitive type, a user defined data type or a Algebraic Data Type. For instance, an int list in ADT below with two constructors, namely, “nil” and “next_list”.

```
data int_list{
    nil();
    next_list(int, int_list);
}
```

Examples:

This Algebraic Data Type tool supports all primitive types in C and all user defined types, such as “struct foo” or “union foo”. However, the pointers and arrays cannot be the arguments of constructors. For instance, Node(int*, char[32]) would not be accepted and runned successfully. Some type definition examples are given as follows.

1. Typical tree data structure

```
data tree{
     leaf();
     node(int, tree, tree);
}
```

2. Expression representation

```
data Expr{
    True();
    False();
    Var(string);
    Not(Expr);
    And(Expr, Expr);
    Or(Expr, Expr);
    Lt(Expr, Expr);
    Le(Expr, Expr);
    Eq(Expr, Expr);
    Ne(Expr, Expr);
    Ge(Expr, Expr);
    Gt(Expr, Expr);
}
```

Macros and inline functions:

A macro in C is a fragment of code whose content will be replaced during compile time. The .h file which is created by ADT tool provides a number of macros. Take the previous tree data type as example, It gives macros like "if_Node()" and "switch_Tree()", which implements pattern matching and tag scheme. One the other hand, the inline functions are quite similar with macros, which offers the corresponding construction functions. A simple instance is given below.

```
#define if_next_list(v, v0, v1) \
{int_list _ADT_v=(v);\
if ((uintptr_t)(_ADT_v) >= 1) {\
int v0=((struct _ADT_int_list_next_list*)((uintptr_t)_ADT_v-0))->f0; \
struct _ADT_int_list* v1=\
    ((struct _ADT_int_list_next_list*)((uintptr_t)_ADT_v-0))->f1;
```

The codes above are generated based on the previous data type of “int_list”, which has two arguments of “int” and “int_list”. These are the typical macro codes that a keyword “#define” leads the start of the definition and the following is its macro name, namely, “if_next_list”. This macro is designed to check whether the input from variable “v” is the “next_list” type. Once it is true, its first field (type of “int”) and the second field will be assign to variables “v0” and “v1”, which are the substitutes of users-define names. For example, the macro codes above can be used to check the length of a list(see codes below).

```
int length_list(int_list l){
    if_next_list(l, val, next)
        return 1 + length_list(next);
    else()
        return 0;
    end_if()
}
```