Functions in C5.

Juan José Cabezas
PEDECIBA Informática
Instituto de Computación, Universidad de la República,
Casilla de Correo 16120, Montevideo, Uruguay
Email: jcabezas@fing.edu.uy

Abstract

In this paper we present the function type in C5.

C5 is a superset of the C programming language. The main difference between C and C5 is that the type system of C5 supports the definition of types of dependent pairs, i.e., the type of the second member of the pair depends on the value of the first member (which is a type).

Another C5 extension is the type initialization expression which is a list of dependent pairs that can be attached to type expressions in a type declaration.

These extensions provide C5 with dynamic type inspection at run time and attribute type definition. The result is a powerful framework for generic programming.

The version 0.98 (September, 2006) of the C5 compiler includes the function type extending the power and expressiveness of the language.

The paper introduces C5 function types, the functions `C5_fapply` and `C5_compil`, and a collection of examples.

Keywords: *dynamic type; dependent pair type ; generic programming ; function type*

1 Introduction

Polymorphic functions are a well known tool for developing generic programs.

For example, the function *pop* of the Stack ADT

$$pop: \forall\, T.\ Stack\ of\ T \ \rightarrow\ Stack\ of\ T$$

has a single algorithm that will perform the same task for any stack regardless of the type of its elements. In this case, we say that the *pop* algorithm is similar for different instantiations of T.

A more complex and powerful way to express generic programs are the functions with dependent type arguments (i.e., the type of an argument may depend on the value of another) that perform different tasks depending on the argument type. These functions may inspect the type of the arguments at run time to select the specific task to be performed.

The C `printf` and `scanf` functions are two widely used examples of this kind of generic programs that are defined for a finite number of argument types. As we will see later, the type of these useful functions cannot be determined at compile time by a standard C compiler.

Even more powerful generic programs are achieved when we extend the finite number of argument types to the entire type system. This class of generic functions can perform different tasks depending on the argument type extending its expression power to include generic programs like parser generators (a top paradigm in generic programming).

C5 is a superset of the C programming language. The extensions introduced in C5 are the notion of Dependent Pair Type (DPT) and that of a Type Initialization Expression (TIE).

These extensions provide C5 with dynamic type inspection at run time and attribute type definition.

C5 is a minimal C extension that express a wide class of generic programs where the functions `C5_fapply` and `C5_compil` presented in this paper are representative examples.

1.1 The type of `printf`

The C creators [11] warn about the consequences of the absence of type checking in the `printf` arguments:

> " ... printf, the most common C function with a variable number of arguments, uses information from the first argument to determine how many other arguments are present and what their types are. It fails badly if the caller does not supply enough arguments or if the types are not what the first argument says."

Let us see through the simple example in Figure 1 how `printf` works. The first argument of `printf`, called the *format string*, determines the type of the other two: the expressions `4s` and `6.2f` indicate that the type of the second

```
main(){
      double n=42.56;
      char st[10]="coef";
      printf("%4s %6.2f",st,n);
      }
```

Figure 1: A simple C `printf` example.

argument is an array of characters while the third argument is a floating point notation number.

In the case of `printf` and `scanf`, the types declared in the format string are restricted to atomic, array of character and character pointer types. There is also some numeric information together with the type declaration (`4` and `6.2` in our example) that defines the printing format of the second and third arguments. These numeric expressions (attributes) will be called Type Initialization Expressions (TIEs) in C5.

A standard C compiler cannot type check statically the second and third arguments of the example presented in figure 1 because their types depend on the value of the first one (the format string).

In functions like `printf` and `scanf`, expressiveness is achieved at a high cost: type errors are not detected and, as a consequence unsafe code is produced.

However, some C compilers (e.g. the `-Wformat` option in `gcc` [7]) can check the consistence of the format string with the type of the arguments of `printf` and `scanf`. In this case, the format argument is a constant string (readable at compile time) and the C syntax is extended with the format string syntax.

This is not an acceptable solution of the problem because the syntax of the format string is specific for the functions `printf` and `scanf`.

A better solution can be found in Cyclone [15], a safe dialect of C. In this case, the type of the arguments of `printf` and `scanf` is a *tagged union* containing all of the possible types of arguments for `printf` or `scanf`. These tagged unions are constructed by the compiler (*automatic tag injection*) and the functions `printf` and `scanf` include the code to check at run time the type of the arguments against the format string.

Similar results can be obtained with other polymorphic disciplines in statically typed programming languages such as finite disjoint unions (e,g, Algol 68) or function overloading (e.g. C++).

This kind of solution of the `printf` typing problem has the following restrictions:

- The consistency of the format string and the type of the arguments is checked at run time and
- the set of possible types of the arguments of `printf` and `scanf` is finite and included in the declaration (program) of the functions.

However, the concept of *object with dynamic types* or *dynamics* for short, introduced by Cardelli [4] [1] provides an elegant and general solution for the `printf` typing problem.

A *dynamics* is a pair of an object and its type. Cardelli also proposed the introduction in a statically typed language of a new datatype (`Dynamic`) whose values are such pairs and language constructs for creating a dynamic pair (`dynamic`) and inspecting at run time its type tag (`typecase`).

Figure 2 shows a functional program using the `typecase` statement where `dv` is a variable of type `Dynamics` constructed with `dynamic`, *Nat* (natural numbers) and `X * Y` (the set of pairs of type `X` and `Y`) are types to be matched against the type tag of `dv`, `++` is a concatenation operator, and `fst` amd `snd` return the first and second member of a pair.

```
typetostring(dv:Dynamics): Dynamics -> String
    typecase dv of
        (v: Nat) " Nat "
        (v: X * Y) typetostring(dynamic fst(v):X)
                   ++ "  *  "
                   ++ typetostring(dynamic snd(v):Y)
        else "??"
    end
```

Figure 2: The statement `typecase`

Tagged unions or finite disjoint unions can be thought of as *finite versions* of `Dynamics`: they allow values of different types to be manipulated uniformly as elements of a tagged variant type, with the restriction that the set of variants must be fixed in advance.

C5 offers a way to embed *dynamics* within the C language that follows the concepts proposed by Cardelli.

The goal of the C5 language is to experiment with generic programs based on functions with dependent arguments under the following conditions:

- the type dependency of the arguments is checked at compile time and
- the functions accept (and are defined for) arguments of any type.

2 The C5 extensions

Dynamics has been implemented in C5 as an abstract data type called Dependent Pair Type (DPT). Instead of the statement `typecase` there are a set of functions that construct DPTs, inspect the type tag and read or assign values.

Since the use of DPTs is limited to a special class of generic functions, there is a C5 statement called `DT_typedef` to declare valid type definitions for the DPT library.

The major difference of the DPT library with Cardelli's *Dynamics* is concerned with the communication between the static and the dynamic universes:

- In the case of *dynamics*, there is a pair constructor (`dynamic`) for passing a static object to the dynamic universe. The inverse operation –the `typecase` statement– is a selector that retrieves the dynamic object to the static universe if it matches with a given static type.

- In the case of the DPT library, the constructor `DT_pair` is the `dynamic` counterpart, but nothing equivalent to `typecase` can be found in C5. The only way to inspect a DPT object is by using a generic object selector (`C5_gos`) that encodes the static C selectors into the dynamic universe. In other words, it is easy to transfer a static object to the dynamic universe but the inverse is limited to atomic types. In compensation, it is possible to do some *object processing* within the dynamic universe.

This difference allows C5 to construct new dynamic objects at run-time without the *Dynamics* type checking requirements.

2.1 Dependent pairs in C5

For the sake of readability, we will simplify the C type system to `int`, `double`, `char` , `struct` , `union`, array, pointer, defined and function types.

The following is a brief introduction to the most important functions of the DPT library:

- `DPT DT_pair( C_Type t, t object)`
  The function returns a dependent pair where the type tag is the dynamic representation of the first argument `t` and the object member is a reference to the second argument `object`. The C5 compiler assures that DPTs are well formed by checking that the second argument is a variable whose type is the value of the first which is a `DT_typedef` type definition.

- `DPT C5_gos(DPT dp, int i)`
  The function is a universal selector for DPT pairs. If the type tag of `dp` is a struct or a union, then `C5_gos` yields a DPT pair with the type and value of the `i`*th* field. If `dp` is an array, then `C5_gos` returns a DPT pair with the type of the array elements and the `i`*th* element of the array. If `dp` is a pointer or `DT_typedef` DPT, then `C5_gos(dp,1)` yields a DPT pair constructed from the type of the referenced object and the object itself respectively. If `i` is out of range, an dynamic pair with error information is returned.

  `C5_gos` is not defined for atomic or function types.

- `int C5_gtype(DPT)`
  The DPT library is defined for the following type classes:

  1. INT
     The `int` type in C.

2. CHAR
   The `char` type in C.
3. DOUBLE
   The `double` type in C.
4. STRUCT
   The set of `struct` types.
5. UNION
   The set of `union` types.
6. ARRAY
   The set of $type[c_expr]$ types.
7. POINTER
   The set of $*type$ types.
8. TYPEDEF
   The set of type definitions with `DT_typedef`.
9. FUNCTION
   The set of function types.

Note that the C types `register`, `unsigned`, `short long`, `float`, `void`, and `enum` are not included.

The function `C5_gtype` yields the type class of the dynamic pair argument.

- `int C5_isDUnion) DPT )`

  C Unions are not interesting for the DPT Library because there is no way to know the current field of an union at run time. For example, `C5_gos` cannot be defined for C unions.

  Instead of this kind of union , DPT functions recognizes discriminated unions as a special case of the struct type.

  A struct with two fields where the first is an union and the second an integer is matched as a C5 Discriminated Union. In this case, the integer field is supposed to keep the information about the current field of the union.

  The function `C5_isDUnion` returns 1 when the type of a dynamic pair is a Discriminated Union. Otherwise returns 0.

- `int C5_isFunction(DPT)`
  In the C language, the declaration of function types cannot be directly expressed. Instead, we declare the type of function pointer.

  The function `C5_isFunction` returns 1 if the type of the argument is a function pointer. Otherwise returns 0.

- `int C5_gsize(DPT)`
  If the type tag of the argument is a struct, union or function the function

returns respectively the field quantity, the size or the arguments number. If the tagged type is an atomic type `C5_size` returns 0, and in case of pointers or defined types the function returns 1.

- `char * C5_gname(DPT)`
  The function yields a string equal to the current type or field name of the type tag of the dynamic pair.

- `int C5_gpin(DPT)`
  The function returns the C5 pin number of the type member of the pair. Each C5 type has a unique pin number.

- `int C5_gint(DPT, int)`
  `double C5_gdouble(DPT, double)`
  `char C5_gchar(DPT, char)`
  `char *C5_gstr(DPT, char *)`
  These functions return the value of the pair if the type tag is respectively `int`, `double`, `char` and char pointer or array of char, In case of type mismatch the second argument is returned.

- `int C5_int_ass(DPT dp, int v)`
  `int C5_double_ass(DPT dp, double v)`
  `int C5_char_ass(DPT dp, char v)`
  `int C5_str_ass(DPT dp, char *v)`
  If the type tag of `dp` matches, these functions assign the value of the second argument to the second member of the first argument pair and the returned value is 1.

  In case of type mismatch no assigning is performed and the functions return 0.

The equivalence of the DPT library with *Dynamics* is showed in the following program which is a C5 version of the example presented in Figure 2:

```
void typetostring(DPT dv){
        switch(C5_gtype(dv)){
           case INT:    printf(" Int ");
                        break;"
           case STRUCT: if(C5_gsize(dv)==2){
                           typetostring(C5_gos(dv,1));
                           printf(" * ");
                           typetostring(C5_gos(dv,2));
                           }
                        else printf(" ?? ");
                        break;
           default:     printf(" ?? ");
           }
        }
```

We will use DPTs to express the C5 version of `printf` with the form:

$$void\ C5\_printf(DPT)$$

where the format string of the C `printf` function is expressed by the dynamic type of the pair argument. Notice that in this version the type dependency of the argument is checked at compile time while the possible types of the argument are not fixed.

The program below is a first C5 approach to the C `printf` example presented in figure 1:

```
DT_typedef char String[5];
DT_typedef float Fnr;
main(){
      String st="coef";
      Fnr n=42.56;
      c5_printf(DT_pair(String,st));
      c5_printf(DT_pair(Fnr,n));
      }
```

Note that the declared types `String` and `Fnr` are the arguments of the function `DT_pair`.

Note that this is not a complete version of `printf` because the numeric information of the format argument is absent.

2.2 DPT list and atomic DPT constructors.

The DPT library includes an ADT of DPT list and a set of atomic DPT constructors to simplify the use of DPTs:

- `DPT_list dpnil()`
  The null list constructor.
- `DPT_list dpcons(DPT, DPT_list)`
  The inductive list constructor.
- `int dpempty(DPT_list)`
  Returns 1 if the argument is a null list.
- `DPT dphd(DPT_list)`
  It returns the head of the list. If the argument is the null list, the function returns the null DPT.
- `DPT_list dptl(DPT_list)`
  It returns the tail of the list. If the list is empty returns the null list.
- `int dplen(DPT_list)`
  It returns the length of the list.

- `DPT_list dpappend(DPT ,DPT_list)`
  It appends the DPT argument to the end of the list.

- `DPT dp_In(int)`
  `DPT dp_Ch(char)`
  `DPT dp_Do(double)`
  `DPT dp_St(char * )`
  The functions construct dynamic pairs using predefined types and the value of the argument. For example, `dpIn(124)` is equivalent to the following C5 code:

```
DT_typedef int IntType;
...
IntType vn=124;
DT_pair(IntType,vn);
```

The next example presents a DPT list constructed with elements of different types:

```
dpcons( dp_Ch('A'), dpcons( dp_Do(0.57),
        dpcons( dp_St("Hello"), dpnil())));
```

2.3 The Type Initialization Expression (TIE)

A TIE is a DPT list attached to a C5 type.

The syntax of a TIE is a comma-separated sequence of DPTs enclosed by brackets.

Constant expressions of atomic types do not need DPT constructors in a TIE declaration.

For example, the TIE { `'A', 0.57, "Hello"` } is correct and is translated by the C5 compiler to

```
dpcons(dp_Ch('A'),dpcons(dp_Do(0.57),dpcons(dp_St("Hello"),dpnil())));
```

This TIE declaration is equivalent to the TIE { `dp_Ch('A'), dp_Do(0.57), dp_St("Hello")` }.

There is a simple syntactical rule for inserting TIEs into a type declaration:
a TIE is placed on the right of the related type.

The next example shows two type definitions with TIEs:

```
DT_typedef int{1} Numbers[10]{2} [20]{3};
DT_typedef struct{
                  Numbers{4} nrs;
                  char{5} *{6} String_ptr;
                  }{7} Rcrd;
```

In the first type definition, the TIE {1} is attached to an `int` type and the TIEs {2} and {3} are attached to a double array. In the second definition, the TIEs

{4}, {5}, {6} and {7} are attached to the types `Numbers`, `char`, pointer of `char` and `struct` respectively.

TIEs can be inspected at run time using the following functions of the DPT library:

- `DPT_list C5_gtie(DTP)`
  It returns the DPT list in order to be directly manipulated. If the dynamic pair has no TIE, the null DPT list is returned.

- `int C5_gTIE_length(DPT)`
  the function returns the size of the TIE of the type tag of the dependent pair argument. If the TIE does not exist, the function returns 0.

- `int C5_gTIE_type(DPT, int idx)`
  the function applies `C5_gtype` to the TIE element indexed by `idx`. If the TIE does not exist, the function returns 0.

- `int C5_gTIE_int(DPT, int, int)`
  `double C5_gTIE_double(DPT, int, double)`
  `char C5_gTIE_char(DPT, int, char)`
  The functions yield the value of the TIE element indexed by the second argument. If the TIE element to be read does not exist, the function returns the third argument. In case of type mismatch a warning message is printed.

- `int C5_TIE_ass(DPT dp, int, DPT tieval)`
  The function assigns the value of `tieval` to the TIE of `dp` indexed by the second argument. If the assignment is successful the returned value is 1. If the TIE does not exist or the index is out of range or in case of type mismatch a warning message is printed.

After the introduction of TIEs, the C `printf` example presented in figure 1 can be completely expressed in C5 as follows:

```
DT_typedef char String[5] {4};
DT_typedef float {6,2} Fnr;
main(){
      String st="coef";
      Fnr n=42.56;
      c5_printf(DT_pair(String,st));
      c5_printf(DT_pair(Fnr,n));
      }
```

The TIEs {4} and {6,2} are respectively attached to the array and `float` types. Notice that TIE declarations are optional: in this program, for example, the `char` type of the first type definition has no TIE.

2.4 C5 Type equality.

In most cases, C5 functions (e.g. `C5_fapply`) require structural type equality.

In the next type definitions, the types `T1` and `T2` are not equally defined.

```
  DT_typedef struct AT {
int nr1, nr2;
struct{ char *String; struct AT *link; } ST;
} * T1;

  DT_typedef int Integer;
  DT_typedef struct BT {
                Integer cod;
int age;
                struct{ char * name; struct BT * next; } ns;
                } * T2;
```

However, they are structurally equal (`struct{int,int, struct{char ptr, rec ptr}}ptr`). This is the kind of equality used by C5 functions.

The structural equality of C5 types is checked by the `C5_type_seq` function. It returns 1 if the types of two dynamic pairs are at least structurally equal: Other wise, the function returns 0.

```
  int C5_type_seq(DPT d1, DPT d2){
        if(C5_gpin(d1)==C5_gpin(d2)) return(1);/* identical types */
/*  skip typedefs */
if(C5_gtype(d1)==VTYPEDEF)
              return(C5_type_seq(C5_gos(t1,1),t2));
if(C5_gtype(d2)==VTYPEDEF)
              return(C5_type_seq(t1,C5_gos(t2,1)));
        if(C5_gtype(d1)==C5_gtype(d2))a  /* same type class */
            switch(C5_gtype(d1)){
            case INT: case CHAR: case DOUBLE: return(1);
            case STRUCT:  case UNION:
                 if(C5_gsize(d1)==C5_gsize(d2)){
                       int i;
                       for(i=1;i<=C5_gsize(d1);i++)
                             if(C5_type_seq(C5_gos(t1,i),
                                   C5_gos(t2,i))==0) return(0);
                        return(1);
                        }
                 return(0);
            case VARRAY:
                 if(C5_gsize(d1)==C5_gsize(d2))
                        return(C5_type_seq(C5_gos(t1,0),
                              C5_gos(t2,0)));
                 return(0);
            case POINTER:
                 if(C5_rec_ptr(t1) && C5_rec_ptr(t2)){
                       int p1=
```

```
                        is_ptr_checked(C5_gpin(t1),ptr_table);
                    int p2=
                        is_ptr_checked(C5_gpin(t2),ptr_table);
                    if(p1==NChk || p2==NChk) /* not checked */
                          return(C5_type_seq(V_gfield(t1,1),
                                V_gfield(t2,1)));
                    if(p1==Chkd && p2==Chkd) /*both checked*/
                          return(1);
                    }
 else if(!C5_rec_ptr(t1) && !C5_rec_ptr(t2))
                    return(C5_type_seq(C5_gos(t1,1),
                          C5_gos(t2,1)));
                return(0);
            default:return(0);
            }
        return(0);
        }
```

Note that in the case of pointers, the function avoids infinite loops by using a table to assure that recursive pointers are checked once.

3 The function type.

Functions are not first class members in the C language. It is not possible to declare a variable of a function type or assign a function to variables. Instead, the C language accepts function pointers and this is the way functions are handled as objects in a C program.

Function types are also declared through type definition of function pointers.

The C program presented in Figure 3 is a C program including the definition and construction of a function variable.

```
  typedef int (* FunctionType)( int , char );

  int my_func(int n, char c){
        if(c=='0') return(0); else return(n);
        }

  main(){
        FunctionType mf;
mf= &my_func;
printf("%d", &mf(123, '5'));
        }
```

Figure 3: A function type in C.

3.1 Function DPTs

The version 0.98 (September, 2006) of the C5 compiler includes function pointers definitions for DPT construction and we can express in C5 the C presented in Figure 3:

```
  DT_typedef int (* FunctionType)( int , char );

  int my_func(int n, char c){
      if(c=='0') return(0);  else return(n);
      }

  main(){
DPT fdp;
FunctionType mf;
        mf= & my_func;
fdp= DT_pair(FunctionType, mf);
C5_printf(C5_fapply(fdp,
              dpcons(dp_In(123),dpcons(dp_Ch('5'),
                     dpnil())));
}
```

Note the use of the function `C5_fapply)`. This is the only way to use (destroy) a function DPT.

3.1.1 `C5_fapply`

The function `C5_fapply` performs functional application in C5:

$$C5\_fapply: \ DPT \ \times \ DPT\_List \ \rightarrow \ DPT\_List$$

If the first argument is a function pointer, `C5_fapply` type checks (`C5_type_seq`) the function against the argument list contained in the second argument of `C5_fapply`.

If type checking is successful, `C5_fapply` applies the function of the first argument to the n arguments and returns a DPT with the result value. Otherwise, the return DPT includes error information.

3.1.2 `dp_Fn`

However, when the name of a function starts with `c5`, it is possible to construct function DPTs avoiding the `DT_typedef` declaration. In this case the C5 constructor `dp_Fn` takes the type of the function from their signature and constructs a DPT.

The constructor `dp_Fn` allows a compact C5 version of the C program presented in Figure 3:

```
  int c5_my_func(int n, char c){
      if(c=='0') return(0);  else return(n);
      }

  main(){
C5_printf(C5_fapply(dp_Fn(c5_my_func),
dpcons(dp_In(123),dpcons(dp_Ch('5'),
                        dpnil()))));
}
```

3.2 C5_compil

The function `C5_compil` is a good example about the use of `C5_fapply` in generic programs.

$$C5\_compil: \ DPT \ \rightarrow \ DPT$$

The function is a generic translation program where the translation rules are included in the TIEs of the argument of `C5_compil`.

If the type of a dynamic pair is atomic (`int`, `char`,`double`, `char *` or array of char) or its type name starts with "Token", `C5_compil` returns its argument without evaluation. Otherwise, the dynamic pair is evaluated as follows:

1. If the type member of the pair has a TIE, then `C5_compil` evaluates the TIE as follows:
   (a) the elements of the TIE are evaluated (in sequence, from the first to the last) and `C5_compil` returns the result of the first one.
   (b) if the first element of the TIE is a function, the remaining elements are supposed to be the arguments of the function and the output of `C5_fapply` is returned.
   (c) if a member of a TIE attached to a `struct` is a string (`char *`) then `C5_compil` compares the string with the field names of the structure. If the string matches, a evaluation of the matched field is performed. Otherwise, the result pair is the original string.
   (d) if a member of a TIE attached to an array is a integer with a value within the bounds of the array, `C5_compil` evaluates the indexed element. Otherwise, the result pair is the original integer.
2. In case of a pair of `struct` or array type without TIE, the result of the evaluation is the pair itself.
3. In case of disjoint unions, pointers or definitions with no TIEs the result of the evaluation is the respective evaluation of the valid field, the referenced value or or the defined object..
4. In case of Disjoint Unions with no TIEs, the result is the evaluation of the selected member of the union.

5. pairs of function type are returned without changes.

The next C5 program is a simplified version of `C5_compil`.

```
DPT C5_compil(DPT dp){
    int i;
    if(!strncmp("Token",C5_gname(dp),5)) return(dp);
    switch(C5_gtype(dp)){
        case CHAR: case DOUBLE: case INT: return(dp);
        case STRUCT:
              if(isDUnion(dp)){   /* Disjoint Union */
                      if (C5_TIE_length(dp)==0)  /* No TIE */
                              return(C5_compil(C5_gos(C5_gos(dp,1),
                                      C5_gint(C5_gos(dp,2),0)+1  )));
                      else return(C5_scanfActions(dp));
                      }
              else{
                      if (C5_TIE_length(dp)==0) return(dp);
                      else return(C5_scanfActions(dp));
                      }
        case ARRAY:
              if (C5_TIE_length(dp)==0 ||
                      C5_gtype(C5_gos(dp,0))==CHAR) return(dp);
              else return(C5_scanfActions(fp,dp));
        case POINTER:
              if(C5_is_ptr_nul(dp) ||
                      C5_gtype(C5_gos(dp,1))==CHAR) return(dp);
        case TYPEDEF:
              if (C5_TIE_length(dp)>0) return(C5_scanfActions(dp));
              else return(C5_compil(C5_gos(dp,1)));
        case FUNCTION: return(dp);
        default: fprintf(fp,"Unknown type %s.\n",C5_gname(dp)); return(dp);
        }
    }

DPT  C5_scanfActions(DPT dp){     /*  */
       DPT tie1= dphd(C5_gtie(dp));  /* first element of the TIE */
       if(C5_isFunction(tie1))  /* function application */
             return(C5_fapply(tie1, C5_map_tie(fp,dp, dptl(C5_gtie(dp)))));
       else return(dphd(C5_map_tie(fp,dp,C5_gtie(dp))));
       }

DPT_list  C5_map_tie(DPT dp, DPT_list tie_ls){
       if(tie_ls==NULL) return(NULL);
       else{
             DPT auxdp= C5_compil(C5_gPtrdp(dp,dphd(tie_ls)));
             return(dpcons(C5_map_tie(dp, dptl(tie_ls))));
             }
        }
```

```
DPT  C5_gPtrdp(DPT dp, DPT tie){
        DPT out;  int i;
        char *st;
        if(C5_gtype(dp)==ARRAY && C5_gtype(tie)==INT){
                int tie_idx=C5_gint(tie,-1);
                if(tie_idx<0 || tie_idx>=C5_gsize(dp)) return(dp);
                else return(C5_gos(dp,tie_idx));
                }
        if(C5_gtype(dp)==STRUCT && !isDUnion(dp) && C5_isString(tie)){
char *str= C5_gstr(tie, "C5_gPtrdp error");
                for(i=1;i<=C5_gcant(dp);i++)
                        if(!strcmp(str,C5_gname(C5_gos(dp,i))))
                                return(C5_gos(dp,i));
                return(tie);  /* no match */
                }
        return(tie); /* no string */
        }
```

3.3 Examples

The examples presented below show the use of `C5_compil` and C5 functions.

3.3.1 Field selector

The following example shows how `C5_compil` is used to select a certain value of a data structure. The TIE `id` selects the second field of the structure and the TIE {1} selects the second element of the array.

```
DT_typedef struct{
                char {'<'} l;
                char *id;
                char {'>'} g;
                } {"id"} IdExp[2] {1};
  main(){
        IdExp ie;
        C5_printf(C5_compil(C5_scanf(DT_pair(IdExp,ie))));
        }
```

The program returns `"two"` for the input `< one > < two >`.

3.3.2 The sum of a integer list.

The next example shows how `C5_compil` uses a c5 function:

```
  int c5_add(int number, int recProd){ return(number + recProd);}

  DT_typedef struct IntL{
                int number;
                struct{
```

```
                        union{
                                emptyProd {0} nil;
                                struct IntL *next;
                                } UU;
                        int discriminator;
                        } recProd;
                } {dp_Fn(c5_add),"number","recProd"} *Word_List;
main(){
      Int_List nrls;
      C5_printf(C5_compil(C5_scanf(DT_pair(Int_List,nrls))));
      }
```

Note that the functional dynamic pair is constructed with `dp_Fn` and the arguments of the function are the fields `number` and `recProd` of the structure `Int_List`.

`C5_printf` is the C5 generic print function and `C5_scanf` is a scanner that interprets the dynamic type of the argument as the grammar for parsing the standard input and, if the parsing is successful, the object member of the argument pair is constructed accordingly to the input.

`C5_scanf` constructs a list of integers, `C5_compil` computes the sum of the list which is printed by `C5_printf`. For example, he input `11 22 33` produces the output `66`.

3.3.3 Word count.

The following example changes the type of the linked list of the previous example to *word* and the first argument of `c5_add` to the constant 1. .

```
int c5_add(int one, int recProd){ return(one + recProd); }

DT_typedef struct WordL{
              char * word;
              struct{
                        union{
                                emptyProd {0} nil;
                                struct WordL *next;
                                } UU;
                        int discriminator;
                        } recProd;
              } {dp_Fn(c5_add), 1 ,"recProd"} *Word_List;
main(){
      Word_List wls;
      C5_printf(C5_compil(C5_scanf(DT_pair(Word_List,wls))));
      }
```

For example, if the input of this program is `one two three`, it returns `3`.

3.3.4 The reverse function.

The next example applies the simple `c5_reverse` function to the input list to print it in reverse order.

```
char * c5_reverse(char *word, char *recProd){
       printf("%s ",word);
       return(recProd);
       }

 DT_typedef struct WordL{
               char *word;
               struct{
                       union{
                               emptyProd {""} nil;
                               struct WordL *next;
                               } UU;
                       int discriminator;
                       } recProd;
               } {dp_Fn(c5_reverse),"word","recProd"} *Word_List;
 main(){
       Word_List wls;
       C5_compil(C5_scanf(DT_pair(Word_List,wls)));
       }
```

For example, if the input of this program is `one two three`, it prints `three two one`.

4 About the C5 compiler.

The C5 compiler has been developed in 1999 at the *Instituto de Computación (InCo)* in Montevideo, Uruguay. The prototype translates C5 programs into C code.

The C5 parser is a extended C parser with few grammatical modifications. The compiler consists on about 3500 lines where 500 of them are the actual type checker. The compiler parses C5, does type checking of DPT construction and translates the resulting code into C.

Since the language keeps types during run-time, the compiler generates three C files:

1. `C5_defs.h`
   Type definitions.

2. `C5_out.c`
   It includes the functions `C5_gos` and `C5_fapply`, and the type database required by the DPT library.

3. `C5_prog.c`
   The translated C5 source program.

The C5 type checker is trivial: for every `DT_pair` invocation C5 checks statically if the first argument is a `DT_typedef` type definition and if the second is a variable of the same type than the value of the first argument.

The current implementation of C5 (Version 0.98, September 2006) with a sample of C5 programs can be found on the Web at

`http://www.fing.edu.uy/~jcabezas/c5`

5 Related work

The statically typed programming languages Amber [4] and Modula-3 [5] include notions of a dynamic type and a typecase statement. This idea can also be seen in functional programming [12] [14] [6] and in type-safe C dialects like Ccured [13] where *dynamics* are used for converting C in a type safe language.

Although C5 may assign accurate types to untyped C programs like `printf`, it is not a type-safe C dialect but rather a C-based framework for experimenting with generic programming methodologies. In our knowledge, C5 is the first C extension with dynamics developed for generic programming.

The extension of functional languages with dependent types is another interesting alternative for generic programming: Cayenne [2] –a Haskell-like [9] language with dependent types– is powerful enough to encode predicate logic at the type level and thus express generic functions like `printf` without restrictions.

In a close research line to dependent types, the Generic Programming community [3][8]. is developing another approach. PolyP [10] is an example of this work that achieves an expressive power similar to that of dependent types by parameterizing function definitions with respect to data type signatures.

6 Conclusions

The generic functions `C5_printf` and `C5_compil` show that a static typed language extended with DPTs (dynamics) and TIEs can be powerful enough to express a wide class of generic functions in a straightforward, compact and safe way.

TIEs seem to be an useful way of providing parameters for generic functions without affecting the static C type system. In the case of `C5_compil`, the use of TIEs with C5 functions allows C5 to express a generic function in a very compact and readable way.

Finally, we will remark that even though the communication between static and dynamic types is restricted to avoid typing conflicts, we have not detected practical limitations when implementing generic functions like `C5_compil`.

References

[1] Martin Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic Typing in a Statically Typed Language. In *16th POPL*, pages 213–227, 1989.

[2] Lennart Augustsson. Cayenne - a Language with Dependent Types. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, pages 239–250, USA, 1998. ISBN 0-58113-024-4.

[3] R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic Programming - An Introduction -. In *Advanced Functional Programming, LNCS 1608*. Springer-Verlag, 1999.

[4] Luca Cardelli. "amber". In Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet, editors, *"Combinators and functional programming languages : Thirteenth Spring School of the LITP, Val d'Ajol, France, May 6-10, 1985"*, volume 242. Springer-Verlag, 1986.

[5] "Luca Cardelli, James Donahue, and Lucille Glassman". Modula-3 report (revised). Technical report, DEC SRC-RR-52, 1989.

[6] "James Cheney and Ralf Hinze". A Lightweight Implementation of Generics and Dynamics;. In *Haskell Workshop 2002*, October 2002.

[7] GNU. *Using and Porting the GNU Compiler Collection (GCC)*. Free Software Foundation, http://gcc.gnu.org/onlinedocs/gcc-2.95.3/gcc.html.

[8] R. Hinze. Polytypic Programming with Ease. In *4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99), Tsukuba, Japan.*, Lecture Notes in Computer Science Vol. 1722, pages 21–36. Springer-Verlag, 1999.

[9] Paul Hudak, Simon L. Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairnbairn, Joseph H. Fasel, Maria M. Guzman, Kevin Hammond, John Hughes, Thomas Johnsson, Richard B. Kieburtz, Rishiyur S. Nikhil, Will Partain, and John Peterson. Report on the Programming Language Haskell, A Non-strict, Purely Functional Language . *SIGPLAN Notices*, 27(5):R1–R164, 1992.

[10] P. Jansson and J. Jeuring. PolyP - A Polytypic Programming Language Extension. In *POPL 97: The 24th ACM SIGPLAN-SIGACT Symposium of Principles of Programming Languages* , pages 470–482. ACM Press, 1997.

[11] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*, page pg. 71. Prentice Hall, 1977. ISBN 0-13-1101-63-3.

[12] Xavier Leroy and Michel Mauny. Dynamics in ML. *Journal of Functional Programming*, 3(4):431–463, 1993.

[13] "George C. Necula, Scott McPeak, and Westley Weimer". "CCured: type-safe retrofitting of legacy code". In *Symposium on Principles of Programming Languages*, pages 128–139, 2002.

[14] Mark Shields, Tim Sheard, and Simon Peyton Jones. Dynamic Typing as Staged Type Inference. In *POPL 98: The 25th ACM SIGPLAN-SIGACT Symposium of Principles of Programming Languages*, pages 289–302, Jan 1998.

[15] Jim Trevor, Greg Morriset, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe Dialect of C. In *The USENIX Annual Technical Conference* , Monterrey , CA, 2002.

.