A Generic Version of `scanf` Programmed 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 a generic version of `scanf` –the I/O standard library function of the C language– programmed 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 sequence of C constant expressions that can be attached to type expressions in a type declaration.

These extensions make C5 powerful enough to express in a generic form functions with dependent type arguments like `printf` and `scanf`.

The resulting version of `scanf` is a parser generator based on the Earley algorithm that for a given type constructs an object of such type according to the input string.

Keywords: *dynamic typing; dependent pair type ; generic programming ; Earley algorithm*

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 an infinite number of different tasks depending on the argument type and is powerful enough to include generic programs like parser generators (a top paradigm in generic programming).

In this paper, we present a generic version of `printf` and `scanf` programmed in the C5 language.

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).

C5 is a minimal C extension that express a wide class of generic programs where the generic versions of `printf` and `scanf` presented in this paper are two representative examples.

1.1 The type of `printf`

The C creators [14] 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 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` [9]) 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` and not necessarily valid for other functions with dependent type arguments. Furthermore, the C language must be extended with the format string syntax in order to develop a C compiler that typechecks the `printf` and `scanf` functions. This is a too restricted and rigid solution.

A better solution can be found in Cyclone [20], 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 needed 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.

The concept of *object with dynamic types* or *dynamics* for short, introduced by Cardelli [6] [1] provides an elegant and more generic 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 its type tag (`typecase`) at run time.

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 following 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 when the type tag is an atomic type.

Since the use of DPTs is limited to a special class of generic functions, there is a C5 statement called `DT_typedef` that allows 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 and defined types.

The following is an informal and 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, DPT errordp)`
  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 or `dp` is an atomic type, the third argument `errordp` is returned.

- `Type_enum C5_gtype(DPT)`
  The function yields an element of the enumeration {`CHAR`, `INT`, `DOUBLE`,

STRUCT, UNION, ARRAY, POINTER, TYPEDEF}, according to the type tag of the argument.

- `int C5_gsize(DPT)`
  If the type tag of the argument is an struct or union the function returns the field quantity and in case of arrays it returns their size. If the tagged type is an atomic type `C5_size` returns 0i, 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 name or label of the type tag of the argument.

- `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)`
  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,ErrorDp));
                           printf(" * ");
                           typetostring(C5_gos(dv,2,ErrorDp));
                           }
                        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.

We may now write in C5 a first 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`.

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

2.2 The Type Initialization Expression (TIE)

The syntax of a TIE is a comma-separated sequence of C constant expressions enclosed by brackets. A C constant expression is an arithmetic expression constructed from integers, floating point numbers and characters.

String notation in TIEs is accepted as a compressed notation for characters. For example, the TIE { `äbc,ï2`} is equivalent to the TIE { `'a','b','c','1','2'` }

There is a simple syntactical rule for inserting TIEs in 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:

- `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)`
  the function returns an element of the enumeration { `CHAR, INT, DOUBLE, NO_TIE` } that represents the type of the TIE of the type member of the dependent pair argument. If the TIE does not exist, the function returns `NO_TIE`.

- `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.

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.

3 A generic version of `printf`

Since `C5_printf` accepts type expressions (DPTs) as arguments, it is straightforward to extend the restricted argument types of C `printf` (strings and atomic types) to the entire C type system.

For example, the type definition with TIEs presented in Figure 3 is an acceptable argument for the `C5_printf` function.

The next program shows a simplified version of the `C5_printf` function defined for the `int`, `double`, `char`, `struct`, `DT_typedef`, pointer and array types. For the sake of readability, `printf` is used to print values of atomic types.

```
void C5_printf(DPT dp){
    int i;
    char format[100];
```

```
DT_typedef struct{
                  char ref[12];
                  double {2,3} *coef;
                  struct{
                        char name[40];
                        int {5} box_nrs[3];
                        } client;
                  } Client_Record;
```

Figure 3: A type definition with TIEs.

```
   switch(C5_gtype(dp)){
       case INT:
           sprintf(format,"%%%dd",C5_gTIE_int(dp,0,6));
           printf(format,C5_gint(dp,0)); break;
       case DOUBLE:
           sprintf(format,"%%%d.%df",C5_gTIE_int(dp,0,6),
                                     C5_gTIE_int(dp,1,6));
           printf(format,C5_gdouble(dp,0.0)); break;
       case CHAR: printf("%c",C5_gchar(dp,'!')); break;
       case STRUCT:
           printf("\n struct %s={ ",C5_gname(dp));
               for(i=1;i<=C5_gsize(dp);i++){
                   printf(" ");
                   C5_printf(C5_gos(dp,i,ErrorDp));
                   }
           printf("}\n"); break;
       case ARRAY:
           printf("\n array %s=[ ",C5_gname(dp));
               for(i=0;i<C5_gsize(dp);i++){
                   if(C5_gtype(C5_gos(dp,i,ErrorDp))==CHAR)
                       if(C5_gchar(C5_gos(dp,i,ErrorDp),'!')=='\0')
                           break;
                    else if(i>0) printf(" ,");
                    C5_printf(C5_gos(dp,i,ErrorDp));
                    }
               printf(" ]\n"); break;
       case POINTER: case TYPEDEF:
           C5_printf(C5_gos(dp,1,ErrorDp)); break;
       }
   }
```

The following `C5_printf` example prints an object of the type `Client_Record` presented in Figure 3:

```
main(){
      Client_Record cr;
      double r=2.8672;
```

```
strcpy(cr.ref,"0037731443");
cr.coef=&r;
cr.client.box_nrs[0]= 1204;
cr.client.box_nrs[1]= 82761;
cr.client.box_nrs[2]= 464;
strcpy(cr.client.name,"Carlos Gardel");
C5_printf(DT_pair(Client_Record,cr));
}
```

with the following result:

```
struct Client_Record={
array ref=[ 0037731443 ]
2.867
struct client={
array name=[ Carlos Gardel ]
array box_nrs=[  1204 ,82761 ,  464 ]
}
}
```

4 A generic version of `scanf`

The `scanf` function of the C language scans input according to the format string argument which specifies the type and conversion rules of the other arguments. The types specified in the format argument are restricted to (references to) atomic and string types. The results from these conversions are stored in the arguments of the function.

As we did with `printf`, we introduce a generic version of `scanf` in C5:

$$DPT\ C5\_scanf(DPT)$$

where the format string of the C `scanf` function is expressed by the dynamic type of the DPT argument.

`C5_scanf` interprets the dynamic type of the argument as the grammar for parsing the input and, if the parsing is successful, the object member of the argument pair is constructed accordingly to the input. If the input cannot be parsed, `C5_scanf` returns a dependent pair with information about the error.

The resulting program includes a parser generator that can be compared with Yacc [13] and a scanner like Lex [16].

We introduce the `C5_scanf` function by first explaining the *lexical* meaning of the C types that belong to the lexical analyzer and then the *grammatical* meaning of the types related to the syntax analyzer.

4.1 The lexical analyzer

Atomic and string types are the *lexical* or *token* elements of `C5_scanf`. The actual version of `C5_scanf` accepts the following *lexical* types: `int`, `double`, `char`, character pointer and array of characters.

These types are interpreted in `C5_scanf` as follows:

- `int` is interpreted as the regular expression (RE) `[0-9]+`. If the type is attached with { `Signed`} then the RE is `[+-]?[0-9]+`.
- `double` is interpreted as the RE `[0-9]+.[0-9]+`. If the type is attached with { `Signed`} then the RE is `[+-]?[0-9]+.[0-9]+`.
- `char` {`ch`} will match a character equal to `ch`.
- `char A[N]` {`Word`} will match a string equal to `Word` if its length is less than `N` and starts with a letter or punctuation char followed of printable (excluded space) chars. An error is reported if no TIE is declared.
- `char *`{`RE`} will match the input according with the regular expression `RE`. If the TIE is absent the default RE is `[A-Za-z][A-Za-z0-9_]*`.

`C5_scanf` uses *token* type declarations to construct a regular expression table (in the Lex style) with the following order:

1. arrays of chars
2. characters
3. character pointers.
4. `double` numbers
5. `int` numbers

There are also special functions to extend the table with comments and spacing characters. The default table has no comments and the spacing characters are $'\backslash r'$,$'\backslash t'$,$'\ '$ and $'\backslash n'$.

In case of ambiguous specifications, `C5_scanf` chooses the longest match. If there are more than one RE matching the same number of characters, the RE found first in the table is selected.

The example below shows how a string can be scanned according to the RE `[AB]+`:

```
DT_typedef char * {'[','A','B',']','+'} AB;
main(){
      AB ab;
      addComment("/*","*/");
      C5_printf(C5_scanf(DT_pair(AB,ab)));
      }
```

The function `addComment` enables comments with the declared start and ending strings. The program accepts the following input

```
AABBBAAAA  /* A C5_scanf example */
```

and the output will be

```
"AABBBAAAA"
```

The next input string

```
AA12xy   /* this string is not acceptable by the scanner */
```

cannot be parsed and therefore the output will be an error message:

```
struct ErrorMessage={ "Syntax error"
 struct near_at_line={ "AA"      1 }
 }
```

4.2 The syntax analyzer

The types with a *syntactic* meaning in `C5_scanf` are: structures, arrays (array of char is excluded), type definitions , discriminated unions, pointers (char pointer is excluded) and recursive declarations.

4.2.1 Structures and arrays

A `struct` or an array type is a sequence of syntactic or lexical types. The set of strings accepted by this grammar (type) is the cartesian product

$$< S_0,\ S_1,\ \ldots\ , S_n >$$

where $S_0, S_1, ..., S_n$ are the sets of strings of the fields or elements of a given structure or array respectively.

4.2.2 Pointers and definition types

The set of strings accepted by pointer and definition types are the same than the referenced and the defined type respectively.

The next program shows a type (grammar) that includes the structured, pointer and defined types:

```
DT_typedef double Real;
DT_typedef struct{ int n; Real r; } *IntReal[2];
main(){
      IntReal ir;
      C5_printf(C5_scanf(DT_pair(IntReal,ir)));
      }
```

For example, the string `"123 0.432 21 0.55"` is an acceptable input for this program.

4.2.3 Discriminated unions

C unions cannot be used to express alternative grammars because they are not discriminated, that is, the compiler does not know which field of the union is currently stored.

By convention, we will represent alternative grammars in `C5_scanf` by the following type:

$$
\begin{array}{l}
DT\_typedef\ struct\{ \\
\qquad union\{\ d_0, .., d_i, .., d_n\ \}\ < id >; \\
\qquad int\ < id >; \\
\qquad \}\ < id >;
\end{array}
$$

where $d_0, .., d_i, .., d_n$ are the fields of the union and the integer field is called the *union discriminator* and is supposed to keep the information about the current field of the union. Thus, the discriminator field has no grammatical meaning.

The discriminated union type represents in `C5_scanf` the union of the sets of strings accepted by the fields (grammars) $d_0, .., d_i, .., d_n$.

The concept of empty rule is implemented in the fields of discriminated unions through a special nullable *token* called `emptyProd` and defined as follows:

```
DT_typedef char {'\0'} emptyProd;
```

This implementation is based on the proposal of Aycock and Horspool [4].

4.2.4 Recursive declarations

Recursive type declarations of discriminated unions allow us to express unbounded sets of strings.

For example, the program below accepts sequences of numbers and the constructed object will be a linked list of integers:

```
DT_typedef struct IntL{
              union{
                   int n;
                   struct{ struct IntL *next; int n; } RecProd;
                   } UU;
              int discriminator;
              } * Int_List;
main(){
      Int_List il;
      C5_printf(C5_scanf(DT_pair(Int_List,il)));
      }
```

4.3 BNF notation

In most parser generators, grammars are expressed in BNF (Backus-Naur notation) or EBNF (Extended BNF).

The following example is a BNF grammar in Yacc syntax:

```
exp      :  NUMBER
         |  exp '+' exp
         ;
```

where `exp` is a nonterminal symbol and `NUMBER` and `'+'` are terminals (tokens). In `C5_scanf`, this BNF grammar can be expressed by the next type declaration:

```
DT_typedef struct EXP{
             union{
               int number;
               struct{
                 struct EXP *e1; char{'+'} pl; struct EXP *e2;
                 } RecP;
               } UU;
             int discriminator;
             } *exp;
```

4.4 The parsing algorithm

The algorithm of the `C5_scanf` parser generator is an implementation of the Earley algorithm [8] with a lookahead of $k = 1$. This algorithm is a chart-based top-down parser that accepts any context free grammar (CFG) and avoids the left-recursion problem.

The algorithm runs in $O(n^3)$ time order where n is the quantity of symbols to be parsed.

The algorithm has been modified to construct an object of the type that represents the grammar. This is done by programming the recognizer so that it builds an object during the recognition process.

`C5_scanf` will produce parsers even in the presence of conflicts. There are some disambiguating rules in the Yacc style. For example, the `if-else` and the arithmetic expression conflicts are solved in `C5_scanf`.

4.4.1 The if-else conflict

The program below is an example of the `if-else` conflict in `C5_scanf`:

```
DT_typedef  char Else[5] {'e','l','s','e'};
DT_typedef  char If[3]   {'i','f'};
DT_typedef struct IFE{
              union{
                char {'e'} exp;
                struct{ If i; struct IFE *e; } If_stmt;
                struct{ If i; struct IFE *e1;
                        Else s; struct IFE *e2;} If_Else_stmt;
                } UU;
              int  discriminator;
```

```
                } * Stat;
  main(){
        Stat il;
        C5_printf(C5_scanf(DT_pair(Stat,il)));
        }
```

The input `if if e else e` produces two possible outputs for the same input `if (if e else e)` and `if (if e) else e`.

The ambiguity is detected by `C5_scanf` returning a diagnostic message:

```
C5_scanf: Disc. union "Stat" ambiguous in
   field 3 "If_Else_stmt" and
   field 2 "If_stmt".
   Suggestion: attach an int TIE to the "Stat" discriminator
   specifying the preferred alternative ({3} or {2}).
```

If we attach the TIE {2} to the discriminator field of `Stat` then the ambiguity is solved and the output will be

```
 struct If_stmt={
  array If=[ if ]
  d_union Stat={
   struct If_Else_stmt={
    array If=[ if ]
    d_union Stat={ e}
    array Else=[ else ]
    d_union Stat={ e}
    }
   }
  }
```

4.4.2 Arithmetic expressions

The next token declaration in Yacc:

```
  %left  '+'  '-'
  %left  '*'  '/'
```

describes the precedence and associativity of the four arithmetic operators. The four tokens are left associative, and plus and minus have lower precedence than star and slash.

The next type declaration is the `C5_scanf` version of the above Yacc token declaration:

```
  DT_typedef char {'+'} PLUS;
  DT_typedef char {'-'} MINUS;
  DT_typedef char {'*'} TIMES;
  DT_typedef char {'/'} DIV;
```

```
DT_typedef PLUS   {LeftAss, 1} Plus;
DT_typedef MINUS  {LeftAss, 1} Minus;
DT_typedef DIV    {LeftAss, 2} Div;
DT_typedef TIMES  {LeftAss, 2} Times;
```

These disambiguating rules are declared in TIEs attached to type definitions related to token (or lexical) types. The first and second members of the TIE are respectively the associative and precedence rules.

4.5 Semantic Actions

The TIE of a syntactic type may be used to code a semantic action so that when an object of this type is constructed, the semantic action is performed.

`C5_scanf` actions return a DPT, and may obtain the DPTs returned by previous actions.

A semantic action in `C5_scanf` is an integer TIE attached to a syntactic type with the form:

$$\{\ ACTION\_ID,\ Mv_0,\ Mv_1,\ \ldots\ , Mv_n\ \}$$

where `ACTION_ID` is the action identifier and $Mv_0,\ Mv_1,\ \ldots\ , Mv_n\ \ (n \geq 0)$ are references to the elements of the syntactic type.

The code of an action TIE is interpreted by a user-defined function called

$$DPT\ C5\_scanfActions(\ DPT\ )$$

which may access DPTs of previous actions through the function

$$DPT\ C5\_scanfArg(\ int\ TIE\_idx,\ DPT\ dp\ )$$

where `TIE_idx` is an element of $Mv_0,\ Mv_1,\ \ldots\ , Mv_n$.

The nexr program shows the use of an action TIE in a simple grammar:

```
DT_typedef struct{ char {'<'} l; char *id; char {'>'} g; }
                              {SELECT, 2} IdExp[2] {SELECT, 0};
DPT C5_scanfActions(DPT dp){
      if(C5_gTIE_int(dp,0,0) == SELECT )
            return(C5_scanfArg(1,dp));
      else return(dp);
      }

main(){
      IdExp ie;
      C5_printf(C5_scanf(DT_pair(IdExp,ie)));
      }
```

The TIE { `SELECT, 0` } selects the first element of the array and { `SELECT , 2` } selects the second field of the structure.

For example, this program accepts the string `" < one > < two > "` and the output is `"one"`.

Programming with `C5_scanf`.

The following C5 programs are three motivating examples that illustrate the use of the `C5_scanf` function.

Matrix

The example below prints an element of a 2×3 matrix constructed by `C5_scanf`:

```
DT_typedef int Matrix[2][3];
main(){
      Matrix mtx;
      if(C5_scanfError(C5_scanf(DT_pair(Matrix, mtx)))
            printf("Cannot read the matrix.\n");
      else printf("mtx[1][2]=%d\n",mtx[1][2]);
      }
```

Notice the way the variable `mtx` is used to communicate the dynamic and the static universeas. This is an useful programming methodology in C5: the user constructs an object in the dynamic universe which is *processed* in the static universe.

A desk calculator

The next program shows a desk calculator that includes associative and precedence rules to avoid ambiguous grammars:

```
/*  Tokens  */
DT_typedef char {'+'} PLUS;
DT_typedef char {'-'} MINUS;
DT_typedef char {'*'} TIMES;
DT_typedef char {'/'} DIV;

/* Association and precedence rules */
DT_typedef PLUS    {LeftAss, 2}  Plus;
DT_typedef MINUS   {LeftAss, 2}  Minus;
DT_typedef DIV     {LeftAss, 3}  Div;
DT_typedef TIMES   {LeftAss, 3}  Times;
DT_typedef MINUS   {LeftAss, 4}  Uminus;

#define Ae_ struct Aexp *
DT_typedef struct Aexp{    /* Grammar rules */
    union{
   int number;
   struct{Ae_ e1; Plus  a; Ae_ e2;} {SUM,1,3} PlusProd;
   struct{Ae_ e1; Times t; Ae_ e2;} {MUL,1,3} TimesProd;
   struct{Ae_ e1; Minus m; Ae_ e2;} {SUB,1,3} MinusProd;
   struct{Ae_ e1; Div   d; Ae_ e2;} {DVD,1,3} DivProd;
   struct{ Uminus um; Ae_ e;}       {UNA,2  } UminusProd;
   } uu;
```

```
        int disc;
        } *AritihmeticExp;

int Arg(int n, DPT dp){
return(C5_gint(C5_scanfArg(n,dp),0));
}
DT_typedef int SC_Int;

DPT  C5_scanfActions(DPT dp){   /*  Semantic actions  */
        SC_Int result;
        switch(C5_gTIE_int(dp,0,0)){
            case SUM: result= Arg(1,dp) + Arg(2,dp); break;
            case SUB: result= Arg(1,dp) - Arg(2,dp); break;
            case MUL: result= Arg(1,dp) * Arg(2,dp); break;
            case DVD: result= Arg(1,dp) / Arg(2,dp); break;
            case UNA: result= - Arg(1,dp);           break;
            default: return(dp);
            }
        return(DT_pair(SC_Int,result));
        }
main(){
    AritihmeticExp aexp;
    C5_printf(C5_scanf(DT_pair(AritihmeticExp,aexp)));
    }
```

For example, this calculator accepts the input `10 + 2 * 4 / - 2 - 2` and produces the output `4`.

XML checker.

The example below shows a partial and simplified version of a well-formed XML document checker.

```
DT_typedef char *{'[','^','<','&','>',']','+'} charD;
DT_typedef struct{ char {'<'} l; char *id; char {'>'} r; } STag;
DT_typedef struct{ char l[3] {'<','/'}; char *id; char {'>'} r;} ETag;
DT_typedef struct{ char {'<'}l; char *id; char r[3]{'/','>'};
                }EmptyElemTag;

DT_typedef struct{
              union{ charD chd; char * id; } UU;
              int discriminator;
              } CharData;

DT_typedef  struct CharDL{
              union{
                      emptyProd nil;
                      struct{ struct CharDL *c; CharData cd; } CDls;
                      } DU;
              int discriminator;
```

```
              } *CharDataList;

DT_typedef  struct{
              CharDataList cdl;
              struct XML_EL_LS *els;
              } XMLcontent;

DT_typedef  struct XML_EL{
              union{
                      EmptyElemTag eet;
                      struct{ STag s; XMLcontent c; ETag e; }
                                  {CHECK_NAMES,1,3}  elem;
                      } DU;
              int discriminator;
              } *XMLelement;

DT_typedef  struct XML_EL_LS{
              union{
                emptyProd nil;
                struct{struct XML_EL_LS *next; struct XML_EL *el;} els;
                } DU;
              int discriminator;
              } *XMLelementL;

DPT C5_scanfActions(DPT dp){
      if(C5_gTIE_int(dp,0,0)==CHECK_NAMES)
        if(strcmp(C5_gstr(C5_gos(C5_scanfArg(1,dp),2),"error"),
                  C5_gstr(C5_gos(C5_scanfArg(2,dp),2),"error"))){
          fprintf(stderr,"Tag unmatched.\n");
          exit(1);
          }
        else return(dp);
      else return(dp);
      }

main(){
      XMLelement xmldoc;
      C5_printf(C5_scanf(DT_pair(XMLelement,xmldoc)));
      }
```

This program accepts the following XML document

```
  <message>
       <to>juanma@adinet.com</to>
       <from>marcos@adinet.com</from>
       <subject>XML test </subject>
       <text>
          --Can you check this with C5_scanf? ...
       </text>
   </message>.
```

However, it rejects this input text with nested tags:

```
<message>
    <subject>  XML test of nested tags.  </message>
</subject>.
```

Notice that in the case of a successful check, the variable `xmldoc` contains a structured XML document that can easily be inspected or processed.

5 About the implementation.

5.1 The C5 compiler

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

The C5 parser is a reused 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 two C files: one of them is the translation of the C5 source program while the other is a type database required by the DPT library.

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 with a sample of `C5_scanf` programs can be found on the Web at

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

5.2 The function `C5_scanf`.

`C5_scanf` has been implemented in C5 and consists on about 1600 lines where 600 of them belong to the lexical analyzer.

The Earley algorithm and the lexical analyzer have been implemented in a naive way discarding efficiency considerations. In this first approach, we are centered in the methodological aspects of the problem. On the other hand, there is an important amount of work related with fast Earley parsing [4] that can be incorporated in the future.

However, small examples as those presented in this paper are immediately processed in a Pentium III under Linux. Furthermore, we tested `C5_scanf` with the type

```
DT_typedef struct IntL{
              union{
                   emptyProd nil;
                   struct{ struct IntL *next; int n; } RecProd;
```

```
    } UU;
int discriminator;
} * Int_List;
```

and an input file of 10000 integers and the parser constructed the integer list of 10000 nodes in about 0.5 second.

6 Related work

6.1 About C5

The statically typed programming languages Amber [6] and Modula-3 [7] include notions of a dynamic type and a typecase statement. This idea can also be seen in functional programming [15] [19] and in type-safe C dialects like Ccured [17] 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 [11] 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 [5][10]. is developing another approach. PolyP [12] 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.

6.2 About `C5_scanf`

Most parsers in use today are based on efficient linear-time algorithms that accept a subset of CFGs (LL,LR or LALR) [13].

The primary objection to the Earley's algorithm is not functionality but with its run-time response.

Nevertheless, the practical use of Earley parsing has become an interesting alternative in the last years: Accent [18] is the first Earley parser generator along the lines of Yacc and DEEP [3] is an efficient directly-executable Earley parsing.

Finally, we did not found parser generators that accept grammars denoted with C types in the `C5_scanf` style.

7 Conclusions

7.1 About C5

The generic functions `C5_printf` and `C5_scanf` 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.

Although TIEs are very restricted (constant) expressions, they seem to be an useful way of providing parameters for generic functions without affecting the static C type system.

Even though the communication between static and dynamic types is also restricted to avoid typing conflicts, we have not detected practical limitations when implementing generic functions like `C5_scanf`.

7.2 About `C5_scanf`

In contrast with YACC, grammar rules are written within the same programming language as the rest of the program. There is no gap between the grammar formalism and the actual programming language used.

Type declarations with TIEs contain all the information required by the parsing process: the lexical, syntactic, disambiguating and semantic action components. However, semantic actions coded with TIEs has become a more complex task than we use to have in Yacc.

Since BNF or EBNF are clearly better notations than C types for grammar description, `C5_scanf` is not a YACC alternative. On the other hand, when dealing with small or medium size grammars, `C5_scanf` can be an attractive option.

The most remarkable property of `C5_scanf` is the object construction. The user just need to define a grammar which is a type. The parsing result is an object of that type. Thus, the user may inspect or process the resulting object according to the defined type. The parsing process becomes transparent to the user in `C5_scanf`.

8 Acknowledgments

The support and suggestions of many colleagues and students have added greatly to the developing of C5 and the pleasant writing of this paper. In particular: Pablo Queirolo, Gustavo Betarte, Alberto Pardo, Hector Cancela, Bengt Nordstriöm and Alfredo Viola.

Special thanks to the 238 computer engineering students of InCo who tested (and suffered) the successive versions of the C5 prototype.

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] "John Aycock and Nigel Horspool". "directly-executable Earley parsing". *Lecture Notes in Computer Science*, 2027:229+, 2001.

[4] John Aycock and R. Nigel Horspool. Practical Earley Parsing. *The Computer Journal*, 45(6):620–630, 2002.

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

[6] 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.

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

[8] Jay Earley. An Efficient Context-Free Parsing Algorithm. *Communications of the ACM*, 13(2):94–102, February 1970.

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

[10] 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.

[11] 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.

[12] 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.

[13] Steven C. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.

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

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

[16] Michael E. Lesk and Eric Schmidt. "lex: A lexical analyzer generator". In *UNIX Programmer's Manual*, volume 2, pages 388–400. Holt, Rinehart, and Winston, New York, NY, USA, 1979. AT&T Bell Laboratories Technical Report in 1975.

[17] "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.

[18] "Friedrich Wilhelm Schrer". The ACCENT Compiler Compiler. Technical report, GMD Report 101, 2000.

[19] 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.

[20] 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.

.