2005 Edition

THE C5 PROGRAMMING LANGUAGE

Juan José Cabezas

.

Preface

This is the third edition of the C5 manual. C5 is a programming language developed at the Instituto de Computación (InCo).

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. The goal of the C5 language is to experiment with generic programs based on functions with dependent arguments.

At the present the C5 framework includes the C5 compiler and a set of generic libraries that are the result of C5 related projects like the OPM machine, Typed Windows, Generic Fonts and the generic version of `scanf`.

Like the previous editions, the current edition is a recompilation of the reports of C5 projects extended with appendixes.

The main differences between the current edition with previous are:

1. a new rewriting of the introduction chapter.
2. the functions `C5_seq` and `C5_copy` are moved to a new chapter.
3. a new chapter is included with the generic version of `scanf`.
4. a new version of the Appendix C.
5. several errors of the previous edition detected by the students are corrected.

This manual is mainly used in C5 projects and the course *Introducción a la Programación para Diseño Gráfico* of the study programs *Ingeniería en Computación* and *Maestría en Informática* of PEDECIBA.

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

Montevideo, May 20th, 2004.

.

Contents

Chapter 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).

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` and the functions `C5_seq`, `C5_copy` and `opm_image_cons` presented in this paper are representative examples.

1.1 The type of `printf`

The C creators [17] 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 1.1 how `printf` works. The first argument of `printf`, called the *format string*, determines the type

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

Figure 1.1: A simple C `printf` example.

of the other two: the expressions `4s` and `6.2f` indicate that the type of the second 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 1.1 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` [11]) 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 [7] [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 1.2 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.

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.

```
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 1.2: The statement `typecase`

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.

1.2 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.

1.2.1 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 1.2:

```
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 1.1:

```
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.

1.2.2 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 and modified 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.

- `int C5_TIE_ass(DPT tieval, int, DPT dp)`
  The function assigns the value of `tieval` to the TIE of `dp` indexded 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 and the function returns 0.

After the introduction of TIEs, the C `printf` example presented in figure 1.1 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.

1.3 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 1.3 is an acceptable argument for the `C5_printf` function.

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

Figure 1.3: A type definition with TIEs.

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];
    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 1.3:

```
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 ]
}
}
```

Chapter 2

The equality and copy functions.

In this chapter we introduce the generic functions `C5_seq` and `C5_copy`. Both functions are defined using the DPT library.

2.1 Structural equality in C5

A regular C compiler is equipped with equality operators for constant expressions, variables of atomic types and pointers. In case of structured types like `struct` or arrays, the programmer must define an equality function for each declared type.

The definition of a generic structural equality, programmed using the dependent types provided in C5, is presented in figure 2.1.

The function `C5_type_eq` checks if the values of the first pair members are identical. Therefore, the function `C5_seq` will check the structural equality of the second DPT members only if they have the same type.

However, it is not possible to check the structural equality for pointers because any cyclic graph implementation with pointers does not terminate with the kind of traversal algorithm required. Therefore, the pointer equality is defined by checking if both pointers are referencing the same object. This is what `C5_ptr_eq` does.

Since C unions are not discriminated, i.e., the compiler does not know what field of the union is currently stored, a structural equality for unions is not decidable. Therefore, an union equality is not defined in `C5_seq`.

```
int C5_seq(DPT dpa, DPT dpb){
    int i;
    if(!C5_type_eq(dpa,dpb)) return(0);
    switch(C5_gtype(dpa)){
        case INT:    return(C5_gint(dpa)==C5_gint(dpb));
        case DOUBLE: return(C5_gdouble(dpa)==C5_gdouble(dpb));
        case CHAR:   return(C5_gchar(dpa)==C5_gchar(dpb));
        case STRUCT:
            for(i=1;i<=C5_gcant(dpa);i++)
                if(C5_seq(C5_gos(dpa,i),C5_gos(dpb,i))==0)
                    return(0);
            return(1);
        case ARRAY:
            for(i=0;i<C5_gcant(dpa);i++){
                if(C5_gtype(C5_gos(dpa,i))==VCHAR &&
                     C5_gchar(C5_gos(dpa,i))==NULL &&
                     C5_gchar(C5_gos(dpb,i))==NULL) break;
                if(C5_seq(C5_gos(dpa,i),C5_gos(dpb,i))==0)
                     return(0);
                }
            return(1);
        case DT_TYPEDEF:
            return(C5_seq(C5_gos(dpa,1),C5_gos(dpb,1)));
        case POINTER: return(C5_ptr_eq(dpa,dpb));
        default: return(0);
        }
    }
```

Figure 2.1: The C5_seq function

2.2 The copy function in C5

A *case-type* algorithm with dependent types is also used to define a generic copy function.

Figure 2.2 shows the function `C5_copy` which has similar restrictions than the equality: unions cannot be considered for this kind of generic copy and pointer copy is equivalent to the C pointer assignment, i.e., the target pointer will reference the same object than the source one does.

```
int C5_copy(DPT cpy,DPT src){
        int i;
        if(C5_gtype(src)==INT || C5_gtype(src)==DOUBLE ||
            C5_gtype(src)==CHAR || C5_gtype(src)==POINTER)
                return(C5_aass(cpy,src,0));
        switch(C5_gtype(cpy)){
            case CHAR: case DOUBLE: case INT: case POINTER:
                return(C5_aass(cpy,src,0));
            case STRUCT:
                if(!C5_type_eq(cpy,src)) return(0);
                for(i=1;i<=C5_gcant(cpy);i++)
                     if(C5_copy(C5_gos(cpy,i),C5_gos(src,i))==0)
                         return(0);
                return(1);
            case ARRAY:
                for(i=0;i<C5_gcant(cpy);i++){
                    if(i>=C5_gcant(src)) return(1);
                    if(C5_copy(C5_gos(cpy,i),C5_gos(src,i))==0)
                        return(0);
                    }
                return(1);
            case DT_TYPEDEF:
                return(C5_copy(C5_gos(cpy,1),C5_gos(src,1)));
            default: return(0);
            }
        }
```

Figure 2.2: The C5_copy function

The function

`int C5_aass(DPT, DPT, int)`

assigns dependent pairs of atomic and pointer types to depent pairs of any type in the following way:

```
DT_typedef struct{
              int {1} index;
              double {2,2} coef;
              } Rcd;
DT_typedef Rcd Structs1[3];
DT_typedef Rcd Structs2[2];
Structs1 sts1={ {0,0.2},{1,0.4},{2,0.8} };
Structs2 sts2;
Structs1 sts3;
main(){
    DPT dp_sts1, dp_sts2, dp_sts3;
    dp_sts1=DT_pair(Structs1,sts1);
    dp_sts2=DT_pair(Structs2,sts2);
    dp_sts3=DT_pair(Structs1,sts3);
    if(C5_copy(dp_sts2,dp_sts1)){
        C5_printf(dp_sts2);
        if(C5_seq(dp_sts2,dp_sts1))
            printf(" sts1 and sts2 are equal.\n");
        else printf(" sts1 and sts2 are not equal.\n");
        }
    else printf("Copy of sts1 failed\n");
    if(C5_copy(dp_sts3,dp_sts1)){
        C5_printf(dp_sts3);
        if(C5_seq(dp_sts3,dp_sts1))
            printf(" sts1 and sts3 are equal.\n");
        else printf(" sts1 and sts3 are not equal.\n");
        }
    else printf("Copy of sts1 failed\n");
    }
```

Figure 2.3: A C5_copy example

- If the first and second arguments are DPTs with equall atomic or pointer types, then `C5_aass` assigns the value of the second member of the second argument to the second member of the first argument. In this case, the third argument is not significant.

- If the first argument is a dependent pair with a `struct` type and the second argument is a DPT with an atomic or pointer type, then `C5_aass(struct_dp,dp,i)` assigns the value of the second member of the second argument to the *ith* field of the `struct` only if their types are the same.

- If the first argument is a dependent pair with an array type and the second argument is a DPT with an atomic or pointer type, then `C5_aass( array_dp, dp,i)` assigns the value of the second member of the second argument to the *ith* element of the array only if their types are the same.

`C5_copy` is an useful and safe copy function. In C, a function that copies arrays of characters cannot check if the source array size is less or equal than the target one allowing the occurrence of dangerous errors at run-time.

`C5_copy` checks array sizes avoiding copies outside the bounds of the arrays and this property is valid for arrays of any type. Figure 2.3 illustrates how struct arrays of different sizes can be safely copied by the function `C5_copy`.

```
array Structs2=[
struct Rcd={  0 0.20}
,
struct Rcd={  1 0.40}
]
sts1 and sts2 are not equal.

array Structs1=[
struct Rcd={  0 0.20}
,
struct Rcd={  1 0.40}
,
struct Rcd={  2 0.80}
]
sts1 and sts3 are equal.
```

Figure 2.4: The result of the copy example.

The result of the program (figure 2.4) shows that although the size of the arrays `sts1` and `sts2` are different (and their types too) the first and second elements of `sts1` are safely copied to `sts2`. The copy is successful because both arrays have elements of the same type (`Rcd`).

In the case of `sts3`, the target and source arrays belong to the type `Structs1` (i.e., the arrays have equal size) and therefore `sts1` is completely copied to `sts3` and consequently, the arrays become equal.

Chapter 3

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 atomic and string types. The results from such conversions are stored in the arguments of the function.

Like `printf`, we introduce a generic version of `scanf` in C5:

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

where the format string becomes in 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 [16] and a scanner like Lex [18].

We introduce `C5_scanf` 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.

3.1 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 with the following order:

1. REs from arrays of chars.
2. REs from characters.
3. REs from character pointers.
4. REs from `double` numbers.
5. REs from `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 rules which matched the same number of characters, the rule found first in the table is preferred.

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 }
 }
```

3.2 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.

3.2.1 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.

3.2.2 Pointers and type definitions

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

The next program shows a type (grammar) including the structure, 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)));
      }
```

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

3.2.3 Discriminated unions

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

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

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

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 [5].

3.2.4 Recursive declarations

Recursive type declarations of discriminated unions allow the expression of 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)));
        }
```

3.3 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 style:

```
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;
```

3.4 The parsing algorithm

The algorithm of the C5_scanf parser generator is an implementation of the Earley algorithm [10] 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.

Time response is n^3 where n is the quantity of symbols to be parsed.

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

C5_scanf will produce parsers even in the presence of conflicts. There are some disambiguating rules in the Yacc style solving, for example, the if-else conflict and the parsing of arithmetic expressions.

3.4.1 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}
    }
   }
  }
```

3.4.2 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 the associative and precedence rules respectively.

3.5 Semantic Actions

The TIE of a syntactic type may be used to code a semantic action to be performed after an object of this type is constructed or, in other words, the represented grammar rule is parsed.

These actions return a DTP, 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:

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

where `ACTION_ID` is the action identificator and $Mv_0,\ Mv_1,\ ...\ ,Mv_n$ ($n\ \geq\ 0$) 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.

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

3.6 Examples

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

3.6.2 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_dpSearch(C5_scanfArg(1,dp),"Error1")),
                C5_gstr(C5_dpSearch(C5_scanfArg(2,dp),"Error2")))){
              fprintf(stderr,"Tag unmatched.\n");
              exit(1);
              }
          else return(dp);
      else return(dp);
      }

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

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>
```

but rejects this input text with nested tags:

```
   <message>
       <subject>XML test </message>
   </subject>
```

Chapter 4

The C5 Standard Output Library

When a C library is presented, we usually expect the syntactical and semantical description of a set of functions.

In C5, we can introduce a library by describing the meaning of types related to a certain task. The most remarkable property of this kind of C5 libraries is that we can use the library by doing type declarations instead of function callings. This is an important change of the programming methodology: type declarations can now be a very expressive member of a program. Furthermore, as we will see later, a type declaration can be the main code of a program.

In this chapter, we start presenting a small C graphics library and then we show how this library is used in C5 to achieve a powerful page-description language.

4.1 The oriented port machine

An oriented port is either a null port or a port representing a rectangular region of the *page*. The attributes of an oriented port are the coordinates, the color list and the orientation of the rectangular region. There are four different orientations: *Right*,*Down*, *Left* and *Up*. The current color of a non-null port is the first element of the color list. If the color list is null then the current color is *White*.

The *oriented port machine* is a C library based on the `Port_List` abstract data type:

- `Port_List opm_null()`
  The function returns a null port list.

- `Port_List opm_page(Color_List cl)`
  The function yields a one port list which is *Right* oriented, has the *page* size and the color list `cl`.

- `Port_List opm_inters(Port_List pl1, Port_List pl2)`
  The function yields a port list constructed from the intersections (of the rectangular regions) of the cross product of the lists `pl1` and `pl2`.

  The color and orientation of the resulting ports are taken from the corresponding `lp1` ports. If the intersection is a line, a point or empty, then a null port is constructed.

- `Port_List opm_rot(int rot_nr,Port_List pl)`
  The function applies the rotation function to every port of the list `pl`.

  The ports are rotated `r` times according to the rotation rules for port orientation: $rotate(Right) = Down$, $rotate(Down) = Left$, $rotate(Left) = Up$ and $rotate(Up) = Right$.

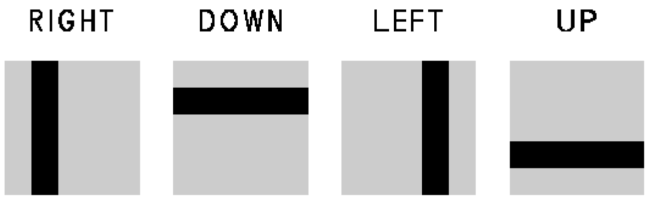

Figure 4.1: sel_split example for $n = 5$ and $i = 2$.

- `Port_List opm_selsplit(int n,int i,Port_List pl)`
  The function applies the function `sel_split` to every port of the list `pl`.

  If `n`$>$ 0 and `i` belongs to the range {`1`,`n`} then the function

  ```
  Port sel_split(int n,int i,Port p)
  ```

  splits the port `p` into `n` sub-rectangles and returns a port with the coordinates of the *ith* sub-rectangle. The orientation and color information are taken from `p`. Figure 4.1 shows the four different results of *sel_split* depending on the four possible orientations of `p`. If `i` is outside the range (`i`<0 or `n<i`) then the function returns a null port.

Finally, in case of `n`≤ 0, the function returns a port equal to `p` for all value of `i`.

- `Port_List opm_partition(float f,Port_List pl)`
  The function applies the function `partition` to every port of the list `pl`.

  If $0<$`f`<1 then the function `Port partition(float f, Port p)` divides the port `p` into two rectangles proportionally to the floating number f and returns a port representing the first sub-rectangle.

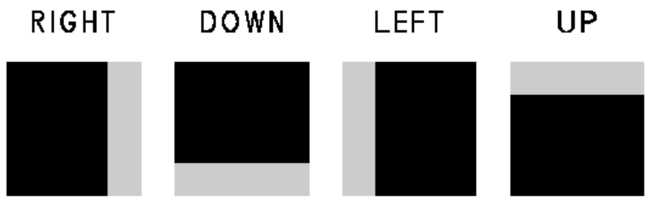

Figure 4.2: A partition example for $f = 0.75$

  The orientation and color information are taken from port `p`. Figure 4.2 shows the four different results depending on the four possible orientations of `p` (`f`$= 0.75$).

  The function returns a null port if `f`≤ 0 and a port equal to `p` if `f`≥ 1.

- `Port_List opm_set_color(int n, Port_List pl)`
  The function applies the function `drop_color` to every port of the list `pl`.

  The function `Port drop_color(int n, Port p)` yields a port with the `p` color list discarding the first `n` elements if `n`> 0. The other attributes of the returned port are equal to those of `p`.

4.1.1 An *opm* example

Figure 4.3 shows a C program using the *opm* library. The function `opm_cat` concatenates two port lists and `opm_print` prints the graphic representation

```
typedef struct{
              int y, x;
              } Ccoords;
typedef Ccoords Point_set[15];
Point_set points={
    { 6, 1 }, { 6, 2 }, { 5, 0 }, { 5, 3 },
    { 4, 3 }, { 3, 1 }, { 3, 2 }, { 3, 3 },
    { 2, 0 }, { 2, 3 }, { 1, 0 }, { 1, 3 },
    { 0, 1 }, { 0, 2 }, { 0, 4 }
    };
Port_List scale(double right, double down,
                double left, double up, Port_List lp){
    return(
        opm_inters(opm_partition(right,opm_rot(0,lp)),
        opm_inters(opm_partition(down ,opm_rot(1,lp)),
        opm_inters(opm_partition(left ,opm_rot(2,lp)),
             opm_partition(up   ,opm_rot(3,lp)))))
       );
    }
main(){
    int i;
    Port_List pl=opm_page(Gray85, Black, NULL), pl_2a;
    opm_print(pl);
    lp_2a=opm_cat(scale(0.8,0.8,0.4,0.4,opm_set_color(1,pl)),
                  scale(0.4,0.4,0.8,0.8,opm_set_color(1,pl))
                  );
    for(i=0;i<15;i++)
         opm_print(opm_inters(
            opm_selsplit(6+1,points[i].y+1,opm_rot(-1,pl_2a)),
            opm_selsplit(4+1,points[i].x+1,pl_2a))
            );
    }
```

Figure 4.3: A simple *opm* example

of the port list argument. The function `scale` is defined for scaling the letter `a` in the upper left and lower right corners.

The result of this example is an image (see figure 4.4) with two black `a` letters in a gray background.

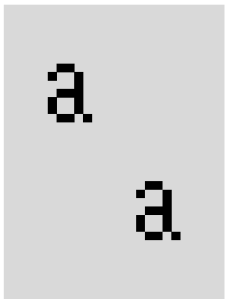

Figure 4.4: The letter `a` example.

4.2 The C5 Standard Output Library

Since a detailed description of the C5 Standard Output Library (see apendix C) is out of the scope of this chapter, we will concentrate our attention in the most important function of the library:

```
Port_List opm_image_cons(DPT dt, Port_List pl)
```

This function is an image constructor with a dependent pair argument. The semantics of the function `opm_image_cons` is informally explained by describing the graphic meaning of types with TIEs:

- Integer numbers: `DT_typedef int {m,n} Int_Def;`
  An `int` TIE is a two integers sequence `{m,n}` defining the visible range where `m` and `n` are the first and last visible integers respectively. If the dependent pair argument of `opm_image_cons` is `DT_pair(Int_Def,i)`, then the function returns the port list constructed by
  `opm_selsplit(n-m+1,i-m+1,pl)`.

- Floating point numbers: `DT_typedef float {s,t} Float_Def;`
  A `float` TIE is a two floating point numbers sequence `{s,t}` defining the visible range of the elements of this type. If the dependent pair argument of `opm_image_cons` is `DT_pair(Float_Def,f)` and `s<t`, then the function returns the port list constructed by
  `opm_partition((f-s)/(t-s),pl)`

  In case of `s`$\geq$`t` , the function returns a port list equal to `pl`.

- Characters: `DT_typedef char {c1,c2,...,c8} Char_Def;`
  If `dt` is a pair with a `char` type definition including a TIE of eight floating point numbers, then the function `opm_image_cons` yields a port list representing the character font defined by the TIE values.

- Structures: `DT_typedef struct`$\{f_0, f_1, ..., f_n\}$`{r} Struct_Def;`
  A `struct` TIE is an integer `{r}` that defines the field rotation. The function `opm_image_cons` returns the port list generated by the intersection of the graphic representation of the struct fields previously rotated $\mathtt{r} \times i$ times ($0 \leq i \leq n$), where i is the index of the *ith* field of a structure of $n+1$ fields.

  The intersections and rotations are implemented with `opm_inters` and `opm_rot` respectively.

  The next C5 program is a short example using a structure of an integer and a floating point number:

  ```
  DT_typedef int {0,2} Int_nr;
  DT_typedef double {0.0,1.0} Double_nr;
  DT_typedef struct{
                  Int_nr n;
                  Double_nr x;
                  } {1} Struct_nx;
  main(){
      Int_nr n=1;
      Double_nr x=0.6;
      Struct_nx nxs;
      nxs.n=n;
      nxs.x=x;
      opm_print(opm_image_cons(DT_pair(Int_nr,n),
          opm_page(Gray85,NULL)));
      opm_print(opm_image_cons(DT_pair(Double_nr,x),
          opm_rot(1,opm_page(Gray85,NULL)));
      opm_print(opm_image_cons(DT_pair(Struct_nx,nxs),
          opm_page(Black,NULL)));
      }
  ```

  Notice that `Int_nr` and `Double_nr` are printed in gray while the struct *Struct_nx* is represented in black. This makes easier to see that the struct is the intersection of the representation of `n` and a $\Pi/2$ rotated `x` (see figure 4.5).

- Arrays: `DT_typedef Elems_type Array_Def[Max] {r,m,n};`
  An array TIE is a three integer sequence `{r,m,n}` where `r` defines the

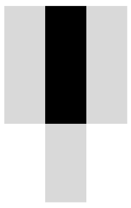

Figure 4.5: A simple struct declaration.

rotation of the elements of the array and `m` and `n` define the first and last visible array elements respectively.

If `m` and `n` belong to the range {`0,Max-1`} then the elements of the array are represented according to the following rule: the *ith* element of the array is graphically represented on the port list constructed by

```
opm_rot(r,opm_selsplit(n-m+1,i-m+1,pl))
```

- Unions: `DT_typedef union`$\{f_0, f_1, ..., f_n\}$`{c} Union_Def;`
  A `union` TIE is an integer {`c`} that defines the color of the fields. The function `opm_image_cons` returns the port list generated by the graphic representation of the valid union field with a current color defined by

  `opm_set_color(r`$\times$`i,pl)` $(0 \leq i \leq n)$

  where i is the index of the *ith* field of an union of $n + 1$ fields.

  Since C unions are not discriminated , the function `opm_image_cons` accepts a struct declaration with two fields, where the first is an union and the second an integer, like a discriminated union. In this case, the integer field is supposed to keep the information about the current field of the union. The discriminated union with TIE is the way to express in C5 the color structure of images.

- Pointers: `DT_typedef Ref_Obj * {r} Ptr_def;`
  A pointer TIE is an integer {`r`} that defines the rotation of the referenced object. The function `opm_image_cons` returns the graphic representation of the referenced object on a `r` times rotated `pl`.

- Type definitions: `DT_typedef Prev_Def {r,c} Def_Def;`
  Type definitions may include type declarations previously defined. In this case, the type definition TIE is a two integer sequence {`r,c`} where `r` and `c` set the rotation and current color of the defined type respectively.

4.2.1 A C5 version of the *opm* example

```
DT_typedef struct{
                   int {0,6} y;
                   int {0,4} x;
                   } Ccoords;
DT_typedef Ccoords {3,1} Point_set[15] {0,1,0};
Point_set pts={
    { 6, 1 }, { 6, 2 }, { 5, 0 }, { 5, 3 },
    { 4, 3 }, { 3, 1 }, { 3, 2 }, { 3, 3 },
    { 2, 0 }, { 2, 3 }, { 1, 0 }, { 1, 3 },
    { 0, 1 }, { 0, 2 }, { 0, 4 }
    };
DT_typedef struct{
           double {0.0,1.0} right, down, left, up;
           } Scale_2[2] {0,1,0};
Scale_2 scs={{0.8,0.8,0.4,0.4},{0.4,0.4,0.8,0.8}};
main(){
    Port_List lp=opm_page(Gray85, Black, NULL);
    opm_print(lp);
    opm_print(opm_image_cons(DT_pair(Point_set,pts),
            opm_image_cons(DT_pair(Scale_2,scs),lp)));
    }
```

Figure 4.6: A C5 version of the *opm* example.

Figure 4.6 shows a C5 version of the *opm* example presented in figure 4.3.

The most relevant difference between both programs is that what the C5 program really do is mainly specified (programmed) in three `DT_typedef` declarations, while the rest of the program itself deals with the variables `pts` and `scs` construction and the image printing. Furthermore, the type definition `Scale_2` is enough expressive to substitute for the function `scale` of the C program.

4.3 Programming images in C5

The Standard Output Library transforms C5 in a high level page-description language, i.e., a language capable of describing the appearance of text, graphic shapes and images on a page. The use of DPTs and TIEs increase the abstraction level of the language producing a readable and compact code for graphics programs.

The image presented in figure 4.7 is generated by a 5 Kb C5 source program.

Some statistics will help us to show why this library produce such a compact code:

The program uses twelve graphics functions of the Standard Output Library in 103305 callings where 92 of them are invoked explicitly in the program and the others 103213 are called implicitly through 15 invocations of `opm_image_cons` which is the only Standard Output Library function with a DPT argument. Seven of these callings answer for the font construction involving 60% of the total quantity of callings.

Eight `DT_typedef` declarations were required by the 15 `opm_image_cons` invocations, remarking that types with TIEs are the heart of the design of programs that use this kind of libraries.

Finally, C5 translates this program into a 22 Kb C code which produces, when compiled and executed, a 3.1 Mb PostScript [2] file.

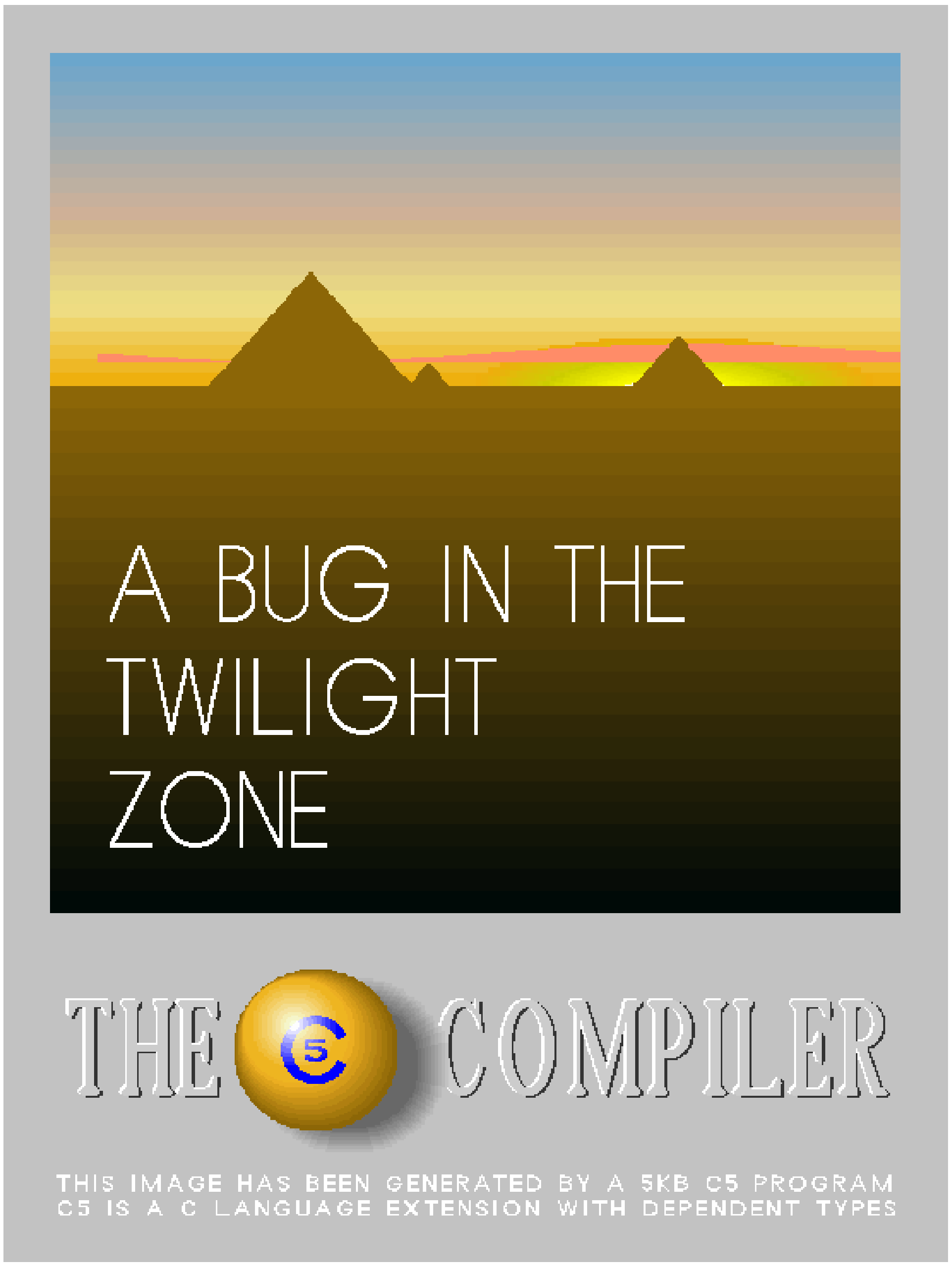
A BUG IN THE TWILIGHT ZONE
THE 5 COMPILER
THIS IMAGE HAS BEEN GENERATED BY A 5KB C5 PROGRAM
C5 IS A C LANGUAGE EXTENSION WITH DEPENDENT TYPES

Figure 4.7: An image programmed in C5

Chapter 5

The C5 Tutorial

Let us begin with a quick introduction to the C5 Standard Output Library. Our aim is to show the use of DPTs and TIEs in real programs, but without getting bogged down in details, formal rules or theoretical concepts.

5.1 Getting Started

The first program to write is the same for all languages:

Print the words HELLO WORLD

In C5, the program to print *HELLO WORLD* is

```
#define Max_Char 40
#define Str_TIE {0,0,Max_Char-1}
DT_typedef char Arial String[Max_Char] Str_TIE;
main(){
    String str="HELLO WORLD";
    opm_print(opm_image_cons(DT_pair(String,str),
        opm_page(Black,NULL)));
    }
```

and the resulting image is showed in figure 5.1. The type definition has two TIEs: `Arial` and `Str_TIE`. The first is attached to the `char` type specifying the font to be used when a character is printed and the second to the array type. The array TIE is a three integer expression with the following form:

{*rotation,first visible element,last visible element*}

where *rotation* is an integer specifying a clockwise rotation of the array elements with an angle of *rotation* $\times\ \Pi/2$, and the *first visible element* and *last*

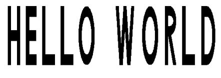

Figure 5.1: Hello World

visible element are integers defining the sequence of array elements that will be printed. In our program the TIE values {`0,0,Max_Char-1`} mean that the characters will be rotated 0 degree and all the array characters are visible for printing.

In the body of the main function, the function `opm_page` is a page constructor whose arguments are the colors (see apendix A) required by the image to be printed on that page. When the page is created , the current color is specified by the first argument (`Black` in our program). The type of the range of this function is `Port_List`. This is the type of the second argument of `opm_image_cons` –the *case-type* function of the library– and the argument of `opm_print` too. Notice that pages and images are represented by the same data structure.

5.2 Integer numbers

The C `printf` translates integer numbers to a digit character sequence starting with the minus character if the number is a negative integer.

Instead of this character oriented translation, the image constructor `opm_image_cons` represents integer numbers by simple geometric images based on color rectangles.

Let us see a short C5 program that prints the number 2 for a quick understanding of the way C5 produce the graphic representation of an integer:

```
DT_typedef int {0,3} intnr;
main(){
    intnr n=2;
    opm_print(opm_page(Gray85,NULL)); /* A gray background */
    opm_print(opm_image_cons(DT_pair(intnr,n),
                             opm_page(Black,NULL)));
    }
```

The `int` TIE is a two integer sequence with the following form:

{ *first visible integer* , *last visible integer* }

where the *first visible integer* and *last visible integer* are integers defining the printable range of numbers. The default TIE for the `int` type is {0,1}.

In our program the TIE values {`0,3`} mean that the integers 0, 1, 2 and 3 are visible for printing.

In this case, the page is virtually divided in four vertical rectangles. Starting from the left side of the page , the first rectangle represents the number 0, the second the number 1, the third the number 2 and the last rectangle the number 3. Accordingly, when printing the number 2, `opm_print` will print the third rectangle painted in black.

Figure 5.2 shows the resulting page.

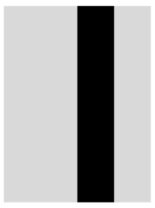

Figure 5.2: An integer number representation.

What would this program do if we try to print a number outside the range {0,3}? Suppose we have the number `11` for printing. Just the gray background will be printed because `11` is not a visible number in this range.

There is a way to express the range $\{ -\infty, +\infty \}$ by declaring a `int` TIE of the form $\{m, n\}$ where $m > n$. For example, the TIE $\{1, 0\}$ specifies that all the integer numbers are visible. The graphic representation of an integer with infinite visibility is the complete page painted with the current color.

5.3 Floating point numbers

A visible floating point number is graphically represented by the first (left) rectangle of a proportional partition of the page.

A program that prints the number 2.5 shows how C5 produce the graphic representation of floating point numbers:

```
DT_typedef double {0.0,4.0} float_nr;
main(){
    float_nr f=2.5;
opm_print(opm_page(Gray85,NULL)); /* A gray background */
opm_print(opm_image_cons(DT_pair(float_nr,f),
    opm_page(Black,NULL)));
}
```

The `double` or `float` type TIE is a two floating point number sequence with the following form:

{ *first visible float* , *last visible float* }

where the *first visible float* and *last visible float* are floating point numbers delimiting the printable range of numbers. The default TIE for `double` or `float` types is {0.0,1.0}.

In our program the TIE values {`0.0,4.0`} mean that a floating point number `f` is visible if `f`$\geq$ 0.0 and `f`$\leq$ 4.0. In this case, the page is partitioned in two rectangles with a `f` dependent size. The graphic representation of `f` is the left rectangle painted with the current color. If $f > 4.0$ the graphic representation will be the complete page painted with the current color and if $f < 0.0$ no action is produced and the resulting image is the page painted with the background color.

Figure 5.3 shows the resulting page.

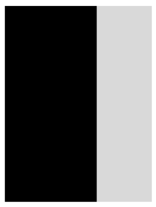

Figure 5.3: A floating point number representation.

5.4 Structures

The type `struct` is represented by the intersection of its fields rotated with an angle determined by the field index and the struct TIE.

Let be the following struct declaration of $n+1$ fields:

$$DT\_typedef\ struct\{\ d_0,..,d_i,..,d_n\ \}\ \{r\}\ sd;$$

where $d_0,..,d_i,..,d_n$ are C5 type declarations. The graphic representation for the struct type is the intersection of the rotated graphic representation of $d_0,..,d_i,..,d_n$. The rotation angle for the i^{th} field of the struct sd is

$$r \times i \times \Pi/2$$

where r is the rotation declared in the struct TIE. The default struct TIE is 1. The struct representation will be painted with the current color of the first field.

The next program shows how the graphic representation of the type `struct` is generated by `opm_image_cons`:

```
DT_typedef int {0,2} int_nr;
DT_typedef double {0.0,1.0} double_nr;
DT_typedef struct{
                int_nr n;
                double_nr x;
                } {1} nx_Struct;
main(){
    int_nr n=1;
    double_nr x=0.6;
    nx_Struct nxs;
    nxs.n=n;
    nxs.x=x;
    opm_print(opm_image_cons(DT_pair(int_nr,n),
        opm_page(Gray85,NULL)));
    opm_print(opm_image_cons(DT_pair(double_nr,x),
        opm_rot(1,opm_page(Gray85,NULL)));
    opm_print(opm_image_cons(DT_pair(Struct_nx,nxs),
        opm_page(Black,NULL)));
    }
```

Notice that `int_nr` and `double_nr` are printed in gray while the struct *Struct_nx* is represented in black. This makes easier to see that the struct is the intersection of the representation of `n` and a $\Pi/2$ rotated `x` (see figure 5.4).

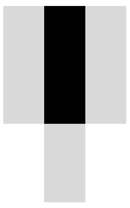

Figure 5.4: A simple struct declaration.

The struct type with TIEs is an expressive programming resource of the C5 Standard Output Library. The code of `opm_scale` illustrates how a member of this library has been programmed:

```
DT_typedef struct{
                double {0.0,1.0} x2,y1,x1,y2;
                } {1} dp2;

Port_List opm_scale(double left, double right,
                   double up,   double down,
                   Port_List pl){
        dp2  margs;
        margs.x1= left ;
        margs.x2= right;
        margs.y2= up;
        margs.y1= down;
        if(pl==NULL) return(NULL);
        else return(opm_cat(
                opm_image_cons(DT_pair(dp2,margs),
                        opm_cons(opm_hd(pl),NULL)),
                opm_scale(left,right,up,down,opm_tl(pl))));
        }
```

The function `opm_cat` is the concatenation operator for port lists and the functions `opm_hd` and `opm_tl` return the head and the tail of a port list respectively.

Let us look closer at the struct declaration because there are two interesting things to note here. First, since the rectangles representing the variables $x2, y1, x1$ and $y2$ are rotated 0, 1, 2 and 3 times $\Pi/2$ respectively, the intersection produced by the struct `dp2` will be a scaled rectangle defined by the values of the `x2,y1,x1` and `y2` variables. Second, what the function `opm_scale` really do is mainly specified (programmed) in the struct declaration while the body of the function itself deals with the `margs` variable assigning and the port list `pl` recursive handling.

5.5 Arrays

In the next program the function *sin* is visualized using an array of floating point numbers. This example is interesting because it shows the graphic power of this type for function visualization:

```
#define Max 100
DT_typedef double {-1.0,1.0} func_visual[Max] {3,0,Max-1};
main(){
      func_visual fn;
      double rn=0.0;
      int i;
```

```
        for(i=0;i<Max;i++){
              fn[i]= sin(rn);
              rn= rn + 2.0*M_PI/Max;  /* M_PI=3.1416... */
              }
        opm_print(opm_page(Gray85,NULL)); /* A gray background */
        opm_print(opm_image_cons(DT_pair(func_visual,fn),
              opm_page(Black,NULL)));
        }
```

The array TIE specifies that all the array elements are visible and will be rotated $3 \times \Pi/2$ before printing. The array `fn` is assigned with `sin` values in the range $\{0, 2\Pi\}$ and then visualized.

Figure 5.5 shows the *sin* function visualization.

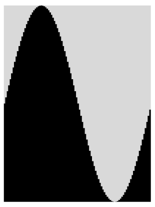

Figure 5.5: The sin function visualization.

5.5.1 The Set mode

As we did for the integer TIE, it is possible to define an infinite range TIE for arrays.

We call the *Set Mode* representation to an array declaration with TIEs of the form

$$\{\ rot,\ m\ ,n\ \}$$

where *rot* specifies the rotation of the array elements and m and n are integers so that $m > n$.

In this case , the elements of the array are represented directly on the page following the order indexed by the array starting from 0.

The next program shows the *Set Mode* representation in an array declaration of integer structures:

```
  DT_typedef struct{
              int {0,6} y;
              int {0,4} x;
              } ccoords;
```

```
DT_typedef ccoords point_set[15] {0,1,0};

point_set pts={
    { 6, 1}, { 6, 2}, { 5, 0}, { 5, 3}, { 4, 3},
    { 3, 1}, { 3, 2}, { 3, 3}, { 2, 0}, { 2, 3},
    { 1, 0}, { 1, 3}, { 0, 1}, { 0, 2}, { 0, 4} };

main(){
    opm_print(opm_page(Gray85,NULL));
    opm_print(opm_image_cons(DT_pair(point_set,pts),
        opm_rot(3,opm_page(Black,NULL)));
    }
```

This is an important array declaration because it simulates a two dimensional Cartesian Coordinate System. In the program, the points are pairs of integers where the first element is the Y axis coordinate and the second, the X axis. Notice again that the kernel of the program is the type declaration.

The resulting image is a 15 points Cartesian representation of the letter a.

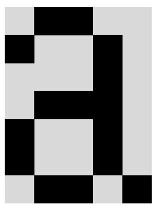
Figure 5.6: The Set Mode Representation.

5.5.2 Matrices.

Matrix representation is obtained in C5 by the double array declaration with finite range TIEs. The following program produce the same output than the previous but now the letter a is represented by a 7×5 matrix:

```
DT_typedef int {1,1} matrix[5] {0,0,4} [7] {3,0,6};
matrix mtx={  0,1,1,0,0,
              1,0,0,1,0,
              0,0,0,1,0,
              0,1,1,1,0,
              1,0,0,1,0,
              1,0,0,1,0,
              0,1,1,0,1 };
```

```
main(){
    opm_print(opm_page(Gray85,NULL));
    opm_print(opm_image_cons(DT_pair(matrix,mtx),
        opm_rot(1,opm_page(Black,NULL))));
    }
```

There are some interesting details here. First, since the integer TIE range is $\{1, 1\}$, only one number –the number one– is visible for printing. Second, the way the double array `mtx` is initialized makes easy the graphic design of the matrix. Third, the port list `opm_page(Black,NULL)` is rotated by `opm_rot` so that the printed matrix (in this case , the letter a) is coincident with the `mtx` initialization.

5.6 Unions

C Unions are not interesting for C5 programs because there is no way to know the current field of an union.

Instead of this kind of union , C5 recognizes discriminated unions as a special case of the struct type.

5.6.1 Discriminated unions

A struct declaration with two fields where the first is an union and the second is an integer is recognized as a discriminated union. In this case, the integer field is supposed to keep the information about the current field of the union. The discriminated union with TIEs is the way we express the color structure of the images printed by `opm_printf`.

The graphic representation of the discriminated union follows the struct rules and the representation of the union field is the representation of the current field painted with a color determined by the union TIE and the place of the field.

The form of a discriminated union is:

$$\begin{array}{ll} \mathit{DT\_typedef} & \mathit{struct}\{ \\ & \mathit{union}\{\ d_0, .., d_i, .., d_n\ \}\ \{c\}\ \ <\mathit{id}>; \\ & \mathit{int}\ \{m, n\}\ \ <\mathit{id}>; \\ & \}\ \{\ r\ \}\ \ <\mathit{id}>; \end{array}$$

where $d_0, .., d_i, .., d_n$ are the fields of the union, and c is an integer number defining the color factor of the union.

The current color of the *ith* field of the union is the $(c \times i + 1)^{th}$ element of the color list of the page.

The next program shows a discriminated union example:

```
DT_typedef struct{
              union{
                  int {0,9} foreground;
                  double {0.0,1.0} background;
                  } {2} cu; /* the color factor is 2 */
              int {1,0} idx;
              } disc_union ;
main(){
    Port_List pl=opm_page(Black,Red,Gray85,NULL);
    disc_union du1,du2;
    du1.idx=1;
    du1.cu.background=0.75; /* Gray85 */
    du2.idx=0;
    du2.cu.foreground=4;    /* Black  */
    opm_print(opm_image_cons(DT_pair(disc_union,du1),pl));
    opm_print(opm_image_cons(DT_pair(disc_union,du2),pl));
    }
```

The variable `foreground` –the first field of the union `cu`– is printed with the color `Black` because the equation $field_place \times color_factor + 1$ is $0 \times 2 + 1$ and this implies that the current color is the first color of the page. In the case of the variable `background`, the resulting equation is $1 \times 2 + 1$, that is, the third color of the page (`Gray85`).

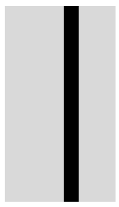

Figure 5.7: A discriminated union representation.

The output of `opm_print` is presented in Figure 5.7.

5.7 Pointers and recursion

The rule for the graphic representation of a pointer type declaration is the representation of the pointed object with a rotation specified by the pointer

TIE. The default TIE for pointers is {0}. If the referenced object is a `char`, C5 will try to represent a NULL terminated string likewise an array of characters.

The form of a pointer TIE is { *rotation* } where *rotation* is an integer.

In the C language, recursive type declarations are expressed by a struct declaration including a field with a pointer to itself. This kind of recursive declarations is required when implementing lists, trees or any other dynamic data structure.

When C5 recognizes a recursive struct declaration, it represents the objects of this type in Set Mode, i.e., the fields of the struct are printed in an ordered sequence, discarding the intersection operator that is applied in non recursive struct declarations.

The program below shows a recursive type declaration for implementing a list of points:

```
DT_typedef struct{
              double {-500.0,10000.0} y;
              int {-100,100} x;
              } point;
DT_typedef struct NODE{
              point pt;
              struct NODE *next;
              } {0} *node_list;
DT_typedef struct NODE * {3} Node_List;

node_list ucons(double y, int x, node_list l){
    node_list p;
    p= (node_list) malloc(sizeof(struct NODE));
    p->pt.y=y;
    p->pt.x=x;
    p->next=l;
    return(p);
    }

main(){
    Node_List nl=NULL;
    int x;
    for(x=-100;x<=100;x++) nl=ucons((double) x*x,x,nl);
    opm_print(opm_page(Gray55,NULL));
    opm_print(opm_image_cons(DT_pair(Node_List,nl),
        opm_page(Black,NULL)));
    }
```

The `point` struct produce the intersection of `y` and `x` while the struct `NODE` is recursive and therefore it generates the image of the fields without intersections.

In order to keep the `y` and `x` coordinates vertical and horizontal respectively, the pointer `Node_List` is declared including a TIE that specifies a rotation $3 \times \Pi/2$.

The visualization of the function $f(x) = x^2$ is predented in Figure 5.8.

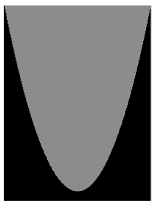

Figure 5.8: A recursive type representation.

5.7.1 Color expressions

Dynamic data structures like lists or binary trees with nodes including unions are natural implementations for color expressions in C5. As a good example, let us write the type declaration of `Color_Series` which is the output type of the function `opm_colors`, the color expression constructor of the C5 Standard Output Library.

```
DT_typedef struct OPMTON{
              dp2 scale;
              struct{ /* discr union  */
                      union{
                             int bg;
                             struct OPMTON *next;
                             } un;
                      int{1,0} idx;  /* infinite range */
                      } du;
              }{0} *Color_Series;
```

The next program shows how color tones are structured with text using `opm_colors`:

```
DT_typedef struct{
              Color_Series * {3} c;
              char Antique_Draft_S string[20] {0,0,19};
              } {0} Color_String;
main(){
```

```
Port_List lp1,lp2;
Color_String cst;
Color_Series obj=opm_colors(4,2*TONES_NR,1.0,0.0);
cst.c=&obj;
strcpy(cst.string,"AB");
lp1=opm_page(opm_col2col( White,  Gray50,    TONES_NR),
             opm_col2col(Gray50,   White,    TONES_NR),
             NULL);
lp2=opm_page(opm_col2col( Black,   White,    TONES_NR),
             opm_col2col( White,   Black,    TONES_NR),
             NULL);
opm_print(opm_image_cons(DT_pair(Color_Series,obj),
        opm_scale(0.75,0.75,0.90,0.90,opm_rot(1,lp1))));
opm_printf(opm_image_cons(DT_pair(Color_String,cst),lp2));
}
```

The function `opm_col2col` is a compressed notation for color series. For example, the color serie denoted by

`opm_col2col(White,Black, 4)`

is, when expanded, equivalent to the color series

`(White,Gray67,Gray33,Black)`

.

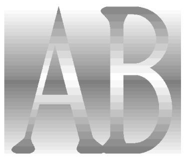

Figure 5.9: Color tones and text.

Figure 5.9 shows the output of `opm_print` .

5.8 Type definitions and enumerations

Type definitions may include type declarations previously defined. In this case, the TIE is of the form:

$$\{rotation, color\}$$

where *rotation* and *color* are integers that work similar to the struct and union TIE respectively. The default TIE for type definitions is $\{0, 0\}$.

The rule for representing enumerations is quite simple. Let be the type declaration

$$DT_typedef enum\{id_0, id_1, ..., id_n\}enum\_id;$$

where id_i are $n + 1$ non equal identifiers. The objects of this type will be represented in a similar way to the type declaration

```
DT_typedef int {0,n} enum_index;
```

Chapter 6

The C5 compiler

The current version of the C5 compiler is written in Yacc, Lex and C and translates C5 to C code. The C5 parser is a reused C parser with few gramatical 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 DTP 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 DTP library. Both C files are then compiled and link-edited with the DTP library.

The C5 Standard Output Library and the C5 Font Library are written in C5 and have a size of about 2500 and 1200 lines respectively.

The current implementation of C5 can be found on the Web at

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

6.1 The C5 type checker

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

Notice that there is here an important restriction for `DT_pair` arguments: they must be just identifiers. Any other C expression is not valid. This restriction makes DTP type checking very simple.

Chapter 7

Related work and conclusions

7.1 Related work

We can find logical frameworks and functional programming languages (or extensions to existing ones) with dependent types.

Logical frameworks are proof checking systems mainly designed for formal program verification. Some examples, among many, are Coq [9], Alf [3] and Elf [12].

There are few functional programming languages with dependent types. Cayenne [4], Dependent ML [21] and Cardelli's Quest [8] are three of them. Cayenne, a Haskell-like [14] language developed in 1998 by Augustsson is powerful enough to encode predicate logic at the type level. DML –a SML [19] extension with a restricted form of dependent types– has been primarily designed for better program error detection.

Xanadu [22] is an imperative programming language with dependent types following the same design concepts than DML. In Xanadu, types depend on constant expressions of natural numbers, allowing the programmer to express invariants more accurately and thus detect more program errors at compile-time. The language includes a weak form of dependent record type that do not support field dependency in the DTP style.

An important contribution of Xanadu is to give some feel in to what dependent types can do in practice. However, Xanadu is actually a rudimentary prototype of a new programming language.

Another interesting approach is being developed by the Generic Programming community [6][13]. With generic programming one achieves an expressive power similar to that of dependent types by parameterizing function definitions with respect to data type signatures. PolyP [15] is an example of a generic programming Haskell extension.

7.2 Conclusions and future work

We have presented C5, a minimal C extension with a restricted form of dependent types. Although C5 has very few syntactical differences with C, the resulting language increases C expressiveness in a dramatic form. C5 is enough powerful to express a generic form of the `printf`, the equality and the copy functions. And this power does not come at the cost of lower code quality: unlike C, `C5_printf` is type checked and `C5_copy` copies ,for example, arrays in a safe way.

Programming with DTPs and TIEs modifies also C programming methodologies in a radical way. A type declaration with TIEs can now concentrate a big amount of information in few lines of readable and safe code. This allows, for example, to introduce a function library by explaining the meaning ot types related to a certain task and, the most important, to use implicitly the library through types declarations with TIEs.

In the future, we will continue to gain more experience in programming with dependent types. We will also explore the properties and power of TIEs which seem to be an interesting research area.

Another line of future work is to experiment with this kind of extensions in other existing programming languages.

Bibliography

[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] Adobe. *Postcript Language Reference Manual.* Adisson Wesley, second edition, 1990.

[3] L. Augustsson, T. Coquand, and B. Nordström. A short description of another logical framework. In *Proceedings of the First Workshop on Logical Frameworks*, pages 39–42, Antibes, 1990.

[4] 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.

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

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

[7] 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.

[8] Luca Cardelli. The Quest Language and System. Technical report, DEC SRC, 1994.

[9] Thierry Coquand and Gérard Huet. The Calculus of Constructions. Technical Report 530, INRIA, Centre de Rocquencourt, 1986.

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

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

[12] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *JACM*, pages 40(1):143–184, 1993.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

[19] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[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.

[21] H. Xi and F. Plenning. Dependent Types in Practical Programming. In *Proceedings of ACM SIGPLAN, Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, 1999.

[22] Hongwei Xi. Imperative Programming with Dependent Types. In *Proceedings of 15th IEEE Symposium on Logic in Computer Science (LICS'00)*, pages 375–387, Santa Barbara, 2000.

Appendix A

Colors

```
AliceBlue
AntiqueWhite
AntiqueWhite1
AntiqueWhite2
AntiqueWhite3
AntiqueWhite4
Aquamarine
Aquamarine1
Aquamarine2
Aquamarine3
Aquamarine4
Azure
Azure1
Azure2
Azure3
Azure4
Beige
Bisque
Bisque1
Bisque2
Bisque3
Bisque4
Black
BlanchedAlmond
Blue
Blue1
Blue2
Blue3
Blue4
BlueViolet
Brown
Brown1
Brown2
Brown3
Brown4
Burlywood
Burlywood1
Burlywood2
Burlywood3
Burlywood4
CadetBlue
CadetBlue1
CadetBlue2
CadetBlue3
CadetBlue4
Chartreuse
Chartreuse1
Chartreuse2
Chartreuse3
Chartreuse4
Chocolate
Chocolate1
Chocolate2
Chocolate3
Chocolate4
Coral
Coral1
Coral2
Coral3
```

```
Coral4
CornflowerBlue
Cornsilk
Cornsilk1
Cornsilk2
Cornsilk3
Cornsilk4
Cyan
Cyan1
Cyan2
Cyan3
Cyan4
DarkGoldenrod
DarkGoldenrod1
DarkGoldenrod2
DarkGoldenrod3
DarkGoldenrod4
DarkGreen
DarkKhaki
DarkOliveGreen
DarkOliveGreen1
DarkOliveGreen2
DarkOliveGreen3
DarkOliveGreen4
DarkOrange
DarkOrange1
DarkOrange2
DarkOrange3
DarkOrange4
DarkOrchid
DarkOrchid1
DarkOrchid2
DarkOrchid3
DarkOrchid4
DarkSalmon
DarkSeaGreen
DarkSeaGreen1
DarkSeaGreen2
DarkSeaGreen3
DarkSeaGreen4
DarkSlateBlue
DarkSlateGray
DarkSlateGray1
DarkSlateGray2
DarkSlateGray3
DarkSlateGray4
DarkSlateGrey
DarkTurquoise
DarkViolet
DeepPink
DeepPink1
DeepPink2
DeepPink3
DeepPink4
DeepSkyBlue
DeepSkyBlue1
DeepSkyBlue2
DeepSkyBlue3
DeepSkyBlue4
DimGray
DimGrey
DodgerBlue
DodgerBlue1
DodgerBlue2
DodgerBlue3
DodgerBlue4
Firebrick
Firebrick1
Firebrick2
Firebrick3
Firebrick4
FloralWhite
ForestGreen
Gainsboro
GhostWhite
Gold
Gold1
Gold2
Gold3
Gold4
Goldenrod
Goldenrod1
```

```
Goldenrod2
Goldenrod3
Goldenrod4
Gray
Gray0
Gray1
Gray10
Gray100
Gray11
Gray12
Gray13
Gray14
Gray15
Gray16
Gray17
Gray18
Gray19
Gray2
Gray20
Gray21
Gray22
Gray23
Gray24
Gray25
Gray26
Gray27
Gray28
Gray29
Gray3
Gray30
Gray31
Gray32
Gray33
Gray34
Gray35
Gray36
Gray37
Gray38
Gray39
Gray4
Gray40
Gray41
Gray42
Gray43
Gray44
Gray45
Gray46
Gray47
Gray48
Gray49
Gray5
Gray50
Gray51
Gray52
Gray53
Gray54
Gray55
Gray56
Gray57
Gray58
Gray59
Gray6
Gray60
Gray61
Gray62
Gray63
Gray64
Gray65
Gray66
Gray67
Gray68
Gray69
Gray7
Gray70
Gray71
Gray72
Gray73
Gray74
Gray75
Gray76
Gray77
Gray78
```

Gray79
Gray8
Gray80
Gray81
Gray82
Gray83
Gray84
Gray85
Gray86
Gray87
Gray88
Gray89
Gray9
Gray90
Gray91
Gray92
Gray93
Gray94
Gray95
Gray96
Gray97
Gray98
Gray99
Green
Green1
Green2
Green3
Green4
GreenYellow
Grey
Grey0
Grey1
Grey10
Grey100
Grey11
Grey12
Grey13
Grey14
Grey15
Grey16
Grey17
Grey18
Grey19
Grey2
Grey20
Grey21
Grey22
Grey23
Grey24
Grey25
Grey26
Grey27
Grey28
Grey29
Grey3
Grey30
Grey31
Grey32
Grey33
Grey34
Grey35
Grey36
Grey37
Grey38
Grey39
Grey4
Grey40
Grey41
Grey42
Grey43
Grey44
Grey45
Grey46
Grey47
Grey48
Grey49
Grey5
Grey50
Grey51
Grey52
Grey53
Grey54

```
Grey55
Grey56
Grey57
Grey58
Grey59
Grey6
Grey60
Grey61
Grey62
Grey63
Grey64
Grey65
Grey66
Grey67
Grey68
Grey69
Grey7
Grey70
Grey71
Grey72
Grey73
Grey74
Grey75
Grey76
Grey77
Grey78
Grey79
Grey8
Grey80
Grey81
Grey82
Grey83
Grey84
Grey85
Grey86
Grey87
Grey88
Grey89
Grey9
Grey90
Grey91
Grey92
Grey93
Grey94
Grey95
Grey96
Grey97
Grey98
Grey99
Honeydew
Honeydew1
Honeydew2
Honeydew3
Honeydew4
HotPink
HotPink1
HotPink2
HotPink3
HotPink4
IndianRed
IndianRed1
IndianRed2
IndianRed3
IndianRed4
Indianred
Ivory
Ivory1
Ivory2
Ivory3
Ivory4
Khaki
Khaki1
Khaki2
Khaki3
Khaki4
Lavender
LavenderBlush
LavenderBlush1
LavenderBlush2
LavenderBlush3
LavenderBlush4
LawnGreen
```

LemonChiffon
LemonChiffon1
LemonChiffon2
LemonChiffon3
LemonChiffon4
LightBlue
LightBlue1
LightBlue2
LightBlue3
LightBlue4
LightCoral
LightCyan
LightCyan1
LightCyan2
LightCyan3
LightCyan4
LightGold
LightGoldenrod
LightGoldenrod1
LightGoldenrod2
LightGoldenrod3
LightGoldenrod4
LightGoldenrodYellow
LightGray
LightGrey
LightPink
LightPink1
LightPink2
LightPink3
LightPink4
LightSalmon
LightSalmon1
LightSalmon2
LightSalmon3
LightSalmon4
LightSeaGreen
LightSkyBlue
LightSkyBlue1
LightSkyBlue2
LightSkyBlue3
LightSkyBlue4
LightSlateBlue
LightSlateGray
LightSlateGrey
LightSteelBlue
LightSteelBlue1
LightSteelBlue2
LightSteelBlue3
LightSteelBlue4
LightYellow
LightYellow1
LightYellow2
LightYellow3
LightYellow4
LimeGreen
Linen
Magenta
Magenta1
Magenta2
Magenta3
Magenta4
Maroon
Maroon1
Maroon2
Maroon3
Maroon4
MediumAquamarine
MediumBlue
MediumOrchid
MediumOrchid1
MediumOrchid2
MediumOrchid3
MediumOrchid4
MediumPurple
MediumPurple1
MediumPurple2
MediumPurple3
MediumPurple4
MediumSeaGreen
MediumSlateBlue
MediumSpringGreen
MediumTurquoise

```
MediumVioletRed
MidnightBlue
MintCream
MistyRose
MistyRose1
MistyRose2
MistyRose3
MistyRose4
Moccasin
NavajoWhite
NavajoWhite1
NavajoWhite2
NavajoWhite3
NavajoWhite4
Navy
NavyBlue
OldLace
OliveDrab
OliveDrab1
OliveDrab2
OliveDrab3
OliveDrab4
Orange
Orange1
Orange2
Orange3
Orange4
OrangeRed
OrangeRed1
OrangeRed2
OrangeRed3
OrangeRed4
Orangered
Orchid
Orchid1
Orchid2
Orchid3
Orchid4
PaleGoldenrod
PaleGreen
PaleGreen1
PaleGreen2
PaleGreen3
PaleGreen4
PaleTurquoise
PaleTurquoise1
PaleTurquoise2
PaleTurquoise3
PaleTurquoise4
PaleVioletRed
PaleVioletRed1
PaleVioletRed2
PaleVioletRed3
PaleVioletRed4
PapayaWhip
PeachPuff
PeachPuff1
PeachPuff2
PeachPuff3
PeachPuff4
Peru
Pink
Pink1
Pink2
Pink3
Pink4
Plum
Plum1
Plum2
Plum3
Plum4
PowderBlue
Purple
Purple1
Purple2
Purple3
Purple4
Red
Red1
Red2
Red3
Red4
```

```
RosyBrown
RosyBrown1
RosyBrown2
RosyBrown3
RosyBrown4
RoyalBlue
RoyalBlue1
RoyalBlue2
RoyalBlue3
RoyalBlue4
SaddleBrown
Saddlebrown
Salmon
Salmon1
Salmon2
Salmon3
Salmon4
SandyBrown
SeaGreen
SeaGreen1
SeaGreen2
SeaGreen3
SeaGreen4
Seashell
Seashell1
Seashell2
Seashell3
Seashell4
Sienna
Sienna1
Sienna2
Sienna3
Sienna4
SkyBlue
SkyBlue1
SkyBlue2
SkyBlue3
SkyBlue4
SlateBlue
SlateBlue1
SlateBlue2
SlateBlue3
SlateBlue4
SlateGray
SlateGray1
SlateGray2
SlateGray3
SlateGray4
SlateGrey
Snow
Snow1
Snow2
Snow3
Snow4
SpringGreen
SpringGreen1
SpringGreen2
SpringGreen3
SpringGreen4
SteelBlue
SteelBlue1
SteelBlue2
SteelBlue3
SteelBlue4
Tan
Tan1
Tan2
Tan3
Tan4
Thistle
Thistle1
Thistle2
Thistle3
Thistle4
Tomato
Tomato1
Tomato2
Tomato3
Tomato4
Turquoise
Turquoise1
Turquoise2
```

```
Turquoise3
Turquoise4
Violet
VioletRed
VioletRed1
VioletRed2
VioletRed3
VioletRed4
Wheat
Wheat1
Wheat2
Wheat3
Wheat4
White
WhiteSmoke
Yellow
Yellow1
Yellow2
Yellow3
Yellow4
YellowGreen
```

Appendix B

Fonts

In C5, fonts are represented by port lists. Fonts are the images associated to the `char` type. The `char` type TIE selects the desired font in a C5 program. There is a small set of predefined fonts (TIEs).

The rules for font names are

```
<Font>_<Draft|Letter|Ultra>_<UC|C|S|D|UD>
```

where `Font` is the font name (for example `Roman`), `Draft`, `Letter` and `Ultra` is the font resolution (quality) and the separation between fonts is represented by `UC` ultracompact, `C` compact, `S` standard, `D` disperse and `UD` ultra disperse.

The following is the form of a `char` type TIE:

```
{recnr,ftype,vert1,vert2,hor,serif,incl,disp}
```

B.1 A list of font TIES

```
#define Roman          {-1.0,1.0,0.1,0.4,0.15,0.2,0.0,0.7 }
#define Roman_Draft_C  {-1.0,1.0,0.1,0.4,0.15,0.2,0.0,0.7 }
#define Roman_Letter_C {-2.0,1.0,0.1,0.4,0.15,0.2,0.0,0.7 }
#define Roman_Ultra_C  {-3.0,1.0,0.1,0.4,0.15,0.2,0.0,0.7 }

#define Roman_Draft_UC  {-1.0,1.0,0.1,0.4,0.15,0.2,0.0,0.8 }
#define Roman_Letter_UC {-2.0,1.0,0.1,0.4,0.15,0.2,0.0,0.8 }
#define Roman_Ultra_UC  {-3.0,1.0,0.1,0.4,0.15,0.2,0.0,0.8 }

#define Roman_Draft_S  {-1.0,1.0,0.15,0.4 ,0.15,0.25,0.0,0.6 }
```

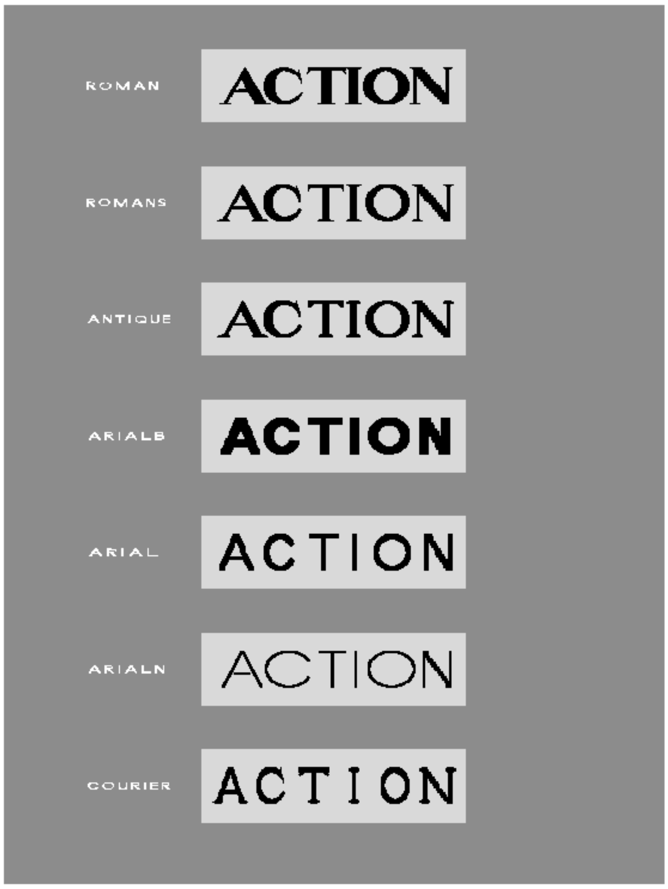

Figure B.1: C5 fonts

```
#define Roman_Letter_S {-2.0,1.0,0.15,0.4 ,0.15,0.25,0.0,0.6 }
#define Roman_Ultra_S  {-3.0,1.0,0.15,0.4 ,0.15,0.25,0.0,0.6 }

#define Roman_Draft_D  {-1.0,1.0,0.2,0.4,0.2,0.3,0.0,0.4 }
#define Roman_Letter_D {-2.0,1.0,0.2,0.4,0.2,0.3,0.0,0.4 }
#define Roman_Ultra_D  {-3.0,1.0,0.2,0.4,0.2,0.3,0.0,0.4 }

#define Roman_Draft_UD  {-1.0,1.0,0.25,0.4,0.25,0.35,0.0,0.1 }
#define Roman_Letter_UD {-2.0,1.0,0.25,0.4,0.25,0.35,0.0,0.1 }
#define Roman_Ultra_UD  {-3.0,1.0,0.25,0.4,0.25,0.35,0.0,0.1 }

#define Romans_Draft_UC  {-1.0,1.0,0.1 ,0.25,0.1 ,0.15,0.0,0.8}
#define Romans_Letter_UC {-2.0,1.0,0.1 ,0.25,0.1 ,0.15,0.0,0.8}
#define Romans_Ultra_UC  {-3.0,1.0,0.1 ,0.25,0.1 ,0.15,0.0,0.8}

#define Romans_Draft_C  {-1.0,1.0,0.1 ,0.3,0.1 ,0.2,0.0,0.7 }
#define Romans_Letter_C {-2.0,1.0,0.1 ,0.3,0.1 ,0.2,0.0,0.7 }
#define Romans_Ultra_C  {-3.0,1.0,0.1 ,0.3,0.1 ,0.2,0.0,0.7 }

#define Romans          {-1.0,1.0,0.1 ,0.3,0.15,0.25,0.0,0.6 }
#define Romans_Draft_S  {-1.0,1.0,0.1 ,0.3,0.15,0.25,0.0,0.6 }
#define Romans_Letter_S {-2.0,1.0,0.1 ,0.3,0.15,0.25,0.0,0.6 }
#define Romans_Ultra_S  {-3.0,1.0,0.1 ,0.3,0.15,0.25,0.0,0.6 }

#define Romans_Draft_D  {-1.0,1.0,0.2,0.3,0.2,0.3,0.0,0.4 }
#define Romans_Letter_D {-2.0,1.0,0.2,0.3,0.2,0.3,0.0,0.4 }
#define Romans_Ultra_D  {-3.0,1.0,0.2,0.3,0.2,0.3,0.0,0.4 }

#define Romans_Draft_UD  {-1.0,1.0,0.2,0.3,0.2,0.3,0.0,0.2 }
#define Romans_Letter_UD {-2.0,1.0,0.2,0.3,0.2,0.3,0.0,0.2 }
#define Romans_Ultra_UD  {-3.0,1.0,0.2,0.3,0.2,0.3,0.0,0.2 }

#define Antique_Draft_C  {-1.0,1.0,0.13,0.3,0.13,-0.28,0.0,0.76}
#define Antique_Letter_C {-2.0,1.0,0.13,0.3,0.13,-0.28,0.0,0.76}
#define Antique_Ultra_C  {-3.0,1.0,0.13,0.3,0.13,-0.28,0.0,0.76}

#define Antique          {-1.0,1.0,0.13,0.3,0.13,-0.31 ,0.0,0.6}
#define Antique_Draft_S  {-1.0,1.0,0.13,0.3,0.13,-0.31 ,0.0,0.6}
#define Antique_Letter_S {-2.0,1.0,0.13,0.3,0.13,-0.31 ,0.0,0.6}
#define Antique_Ultra_S  {-3.0,1.0,0.13,0.3,0.13,-0.31 ,0.0,0.6}

#define Antique_Draft_D  {-1.0,1.0,0.13,0.3 ,0.13,-0.32,0.0,0.4}
#define Antique_Letter_D {-2.0,1.0,0.13,0.3 ,0.13,-0.32,0.0,0.4}
```

```
#define Antique_Ultra_D  {-3.0,1.0,0.13,0.3 ,0.13,-0.32,0.0,0.4}

#define Antique_Draft_UD  {-1.0,1.0,0.13,0.3,0.13,-0.33,0.0,0.1}
#define Antique_Letter_UD {-2.0,1.0,0.13,0.3,0.13,-0.33,0.0,0.1}
#define Antique_Ultra_UD  {-3.0,1.0,0.13,0.3,0.13,-0.33,0.0,0.1}

/* Arial gorda bold */
#define Arialb_Draft_UC  {-1.0,0.0,0.4,0.4,0.4,0.0,0.0,0.85}
#define Arialb_Letter_UC {-2.0,0.0,0.4,0.4,0.4,0.0,0.0,0.85}
#define Arialb_Ultra_UC  {-3.0,0.0,0.4,0.4,0.4,0.0,0.0,0.85}

#define Arialb_Draft_C  {-1.0,0.0,0.4,0.4,0.4,0.0,0.0,0.70}
#define Arialb_Letter_C {-2.0,0.0,0.4,0.4,0.4,0.0,0.0,0.70}
#define Arialb_Ultra_C  {-3.0,0.0,0.4,0.4,0.4,0.0,0.0,0.70}

#define Arialb          {-1.0,1.0,0.4,0.4,0.4,0.0,0.0,0.50}
#define Arialb_Draft_S  {-1.0,1.0,0.4,0.4,0.4,0.0,0.0,0.50}
#define Arialb_Letter_S {-2.0,1.0,0.4,0.4,0.4,0.0,0.0,0.50}
#define Arialb_Ultra_S  {-3.0,1.0,0.4,0.4,0.4,0.0,0.0,0.50}

#define Arialb_Draft_D  {-1.0,1.0,0.42,0.42,0.42,0.00,0.0,0.3 }
#define Arialb_Letter_D {-2.0,1.0,0.42,0.42,0.42,0.00,0.0,0.3 }
#define Arialb_Ultra_D  {-3.0,1.0,0.42,0.42,0.42,0.00,0.0,0.3 }

#define Arialb_Draft_UD  {-1.0,1.0,0.42,0.42,0.42,0.00,0.0,0.0 }
#define Arialb_Letter_UD {-2.0,1.0,0.42,0.42,0.42,0.00,0.0,0.0 }
#define Arialb_Ultra_UD  {-3.0,1.0,0.42,0.42,0.42,0.00,0.0,0.0 }

/* Arial standard  */
#define Arial_Draft_UC  {-1.0,0.0,0.2,0.2,0.2,0.0,0.0,0.90}
#define Arial_Letter_UC {-2.0,0.0,0.2,0.2,0.2,0.0,0.0,0.90}
#define Arial_Ultra_UC  {-3.0,0.0,0.22,0.22,0.20,0.0,0.0,0.90}

#define Arial_Draft_C  {-1.0,0.0,0.2,0.2,0.2,0.0,0.0,0.7 }
#define Arial_Letter_C {-2.0,0.0,0.2,0.2,0.2,0.0,0.0,0.7 }
#define Arial_Ultra_C  {-3.0,0.0,0.22,0.22,0.20,0.0,0.0,0.7 }

#define Arial_Draft_S  {-1.0,1.0,0.2,0.2,0.2,0.0,0.0,0.60}
#define Arial_Letter_S {-2.0,1.0,0.2,0.2,0.2,0.0,0.0,0.60}
#define Arial_Ultra_S  {-3.0,1.0,0.2,0.2,0.2,0.0,0.0,0.60}

/* This is the default font  */
#define Arial          {-1.0,1.0,0.22,0.22,0.22,0.00,0.0,0.3 }
```

```
#define Arial_Draft_D  {-1.0,1.0,0.22,0.22,0.22,0.00,0.0,0.3 }
#define Arial_Letter_D {-2.0,1.0,0.22,0.22,0.22,0.00,0.0,0.3 }
#define Arial_Ultra_D  {-3.0,1.0,0.21,0.21,0.21,0.00,0.0,0.3 }

#define Arial_Draft_UD  {-1.0,0.0,0.30,0.30,0.30,0.00,0.0,0.1}
#define Arial_Letter_UD {-2.0,0.0,0.30,0.30,0.30,0.00,0.0,0.1}
#define Arial_Ultra_UD  {-3.0,0.0,0.30,0.30,0.30,0.00,0.0,0.1}

/* Arial delgada  */
#define Arialn_Draft_C  {-1.0,0.0,0.09,0.09,0.08,0.0,0.0,0.75}
#define Arialn_Letter_C {-2.0,0.0,0.09,0.09,0.08,0.0,0.0,0.75}
#define Arialn_Ultra_C  {-3.0,0.0,0.07,0.07,0.06,0.0,0.0,0.75}

#define Arialn          {-1.0,0.0,0.09,0.09,0.08,0.0,0.0,0.55}
#define Arialn_Draft_S  {-1.0,0.0,0.09,0.09,0.08,0.0,0.0,0.55}
#define Arialn_Letter_S {-2.0,0.0,0.09,0.09,0.08,0.0,0.0,0.55}
#define Arialn_Ultra_S  {-3.0,0.0,0.07,0.07,0.06,0.0,0.0,0.55}

#define Arialn_Draft_D  {-1.0,0.0,0.09,0.09,0.08,0.0,0.0,0.25}
#define Arialn_Letter_D {-2.0,0.0,0.09,0.09,0.08,0.0,0.0,0.25}
#define Arialn_Ultra_D  {-3.0,0.0,0.07,0.07,0.06,0.0,0.0,0.25}

/* Courier  */
#define Courier_Draft_C  {-1.0,3.0,0.2 ,0.2 ,0.2 ,-0.4,0.0,0.5}
#define Courier_Letter_C {-2.0,3.0,0.2 ,0.2 ,0.2 ,-0.4,0.0,0.5}
#define Courier_Ultra_C  {-3.0,3.0,0.22,0.22,0.20,-0.4,0.0,0.5}

#define Courier_Draft_S  {-1.0,3.0,0.2,0.2,0.2,-0.47,0.0,0.4 }
#define Courier_Letter_S {-2.0,3.0,0.2,0.2,0.2,-0.47,0.0,0.4 }
#define Courier_Ultra_S  {-3.0,3.0,0.2,0.2,0.2,-0.47,0.0,0.4 }

#define Courier          {-1.0,3.0,0.24,0.24,0.24,-0.5,0.0,0.2}
#define Courier_Draft_D  {-1.0,3.0,0.24,0.24,0.24,-0.5,0.0,0.2}
#define Courier_Letter_D {-2.0,3.0,0.24,0.24,0.24,-0.5,0.0,0.2}
#define Courier_Ultra_D  {-3.0,3.0,0.24,0.24,0.24,-0.5,0.0,0.2}

#define Courier_Draft_UD  {-1.0,3.0,0.30,0.30,0.30,-0.5,0.0,0.1}
#define Courier_Letter_UD {-2.0,3.0,0.30,0.30,0.30,-0.5,0.0,0.1}
#define Courier_Ultra_UD  {-3.0,3.0,0.30,0.30,0.30,-0.5,0.0,0.1}
```

Appendix C

The Standard Output Library

1. opm_page

   ```
   Port_List opm_page(Color, Color, Color, ..., NULL)
   ```

   `opm_page` returns a port list with a `Right` oriented port. The size of the port is the entire page and the color list is constructed with the `Color` arguments that are not equal to `NULL`. The function accepts up to 12 arguments. (see `opm_Page` for constructing a page with an unbounded color list).

2. opm_Page

   ```
   Port_List opm_Page(Color_List)
   ```

   `opm_Page` returns a port list with a `Right` oriented port. The size of the port is the entire page and the color list is the argument. The constructor of color lists is `LCC` so `LCC(Red, LCC(Green,NULL))` is a valid color list of two elements.

3. opm_print

   ```
   void opm_print(Port_List)
   ```

   `opm_print` translates the port list argument into a PostScript format file using the standard output.

4. opm_scale

```
Port_List opm_scale(double left, double right,
             double up, double down,Port_List)
0.0 <= left  <=1.0
0.0 <= right <=1.0
0.0 <= up    <=1.0
0.0 <= down  <=1.0
```

`opm_scale` scales the ports of the argument according to the values of the arguments `left`, `right`, `up` and `down`.

5. *opm_rot*

```
Port_List opm_rot(int rotnr,Port_List)
```

`opm_rot` rotates `rotnr` $\times\Pi/2$ the ports of the list argument.

6. `opm_image_cons`

```
Port_List opm_image_cons(DTP,Port_List)
```

`opm_image_cons` returns an image (a port list) which is the graphic representation of the object member of the DPT argument printed on a page (the port list argument).

7. `opm_set_color`

```
Port_List opm_set_color(int color_idx,Port_List)
```

`opm_set_color` sets the current color of the ports of the list argument according to the `color_idx` value. if `color_idx` is greater than the color list length of the port then the current color is `White`.

8. `opm_colors`

```
DT_typedef struct{double x2,y1,x1,y2;} dp2;
DT_typedef struct OPMTON {
          dp2 scale;
          struct{ /* discr union  */
                  union{
                  int{0,10} bg;
                  struct OPMTON *next;
                  } un;
          int{1,0}  idx;
```

```
            } du;
    }{0} *Color_Serie;

  Color_Serie opm_colors(int mode,int tones_nr,
                         double coef1,double coef2)
```

`opm_colors` constructs a list of color scaled rectangles. The `mode` argument sets the scaling mode:

- `mode=0` sets the four sides of the rectangle for decreasing concentric scaling.
- `mode=1` sets the right side of the rectangle for decreasing concentric scaling.
- `mode=2` sets the right and down sides of the rectangle for decreasing concentric scaling.
- `mode=3` sets the right, down and up sides of the rectangle for decreasing concentric scaling.
- `mode=4` sets the left and right sides of the rectangle for left to right sequencing scaling.
- `mode=5` sets the up and down sides of the rectangle for up to down sequencing scaling.
- `mode=10` sets the right and left sides of the rectangle for decreasing concentric scaling.
- `mode=20` sets the up and down sides of the rectangle for decreasing concentric scaling.

The `tones_nr` argument defines the length of the color list and the `coef1` and `coef2` arguments are the scaling factor of the active and inactive sides respectively. The standard values of these arguments are 1.0 and 0.0.

9. `opm_col2col`

```
  Color opm_col2col(Color from, Color to, int tones_nr)
```

`opm_col2col` is a compressed notation for the colors of a port. The function represents a color serie starting from `from` to `to` of `tones_nr` tones.

10. `rgb`

```
Color  rgb(int, int, int)
```

The function `rgb` constructs a color object according to the values of the red, green and blue arguments. The range of the arguments is (0,255).

The White color is represented by `rgb(0,0,0)` and Black by `rgb(255,255,255)`.

11. `opm_bcurv4`

```
Port_List opm_bcurv4(int rec_nr,
            double y1_U,double y2_U,double y3_U,double y4_U,
            double y1_D,double y2_D,double y3_D,double y4_D,
            Port_List pl)
```

The function `opm_bcurv4` paints the shape limited by two Bezier curves of four points where the upper curve is defined by (0.0,y1_U), (0.25,y2_U), (0.5,y3_U), (0.75,y4_U) and the lower curve by (0.0,y1_D), (0.25,y2_D), (0.50,y3_D),(0.75,y4_D). The visible range of the `y_...` arguments is (0.0,1.0). `rec_nr` is the resolution level where 10 is the top manual resolution and -1, -2 , -3 and -4 are the automatic resolution for the draft, standard, high and ultra level respectively.

Figure C.1 shows the image produced by the four `opm_bcurv4` examples of the next program:

```
main(){
    Port_List lp=opm_page(Black,Gray65,NULL),lp1,lp2,lp3,lp4;
    lp1=opm_scale(1.0,0.45,0.9,0.45,lp);
    lp2=opm_scale(0.45,1.0,0.9,0.45,lp);
    lp3=opm_scale(1.0,0.45,0.45,0.9,lp);
    lp4=opm_scale(0.45,1.0,0.45,0.9,lp);
    opm_print(opm_set_color(1,opm_concat(lp1,opm_concat(lp2,
       opm_concat(lp3,lp4)))));
    opm_print(opm_bcurv4( 4, 1.0, 0.2, 1.0, 0.0,
                            1.0, 0.0, 0.8, 0.0, lp1));
    opm_print(opm_bcurv4(-4, 1.0, 0.2, 1.0, 0.0,
                            1.0, 0.0, 0.8, 0.0, lp2));
    opm_print(opm_bcurv4(-3, 0.4, 1.2, 1.2, 0.4,
                            0.2, 1.0, 1.0, 0.2, lp3));
    opm_print(opm_bcurv4(-4, 1.0, 0.35,0.35,1.0,
                            0.5, 0.5, 0.5, 0.5, lp4));
    }
```

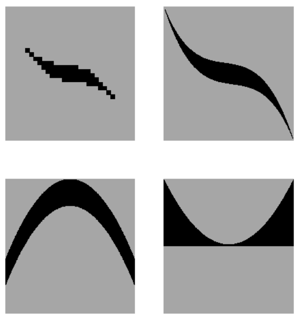

Figure C.1: 4 opm_bcurv4 examples

12. `opm_bcurv4p`

```
Port_List opm_bcurv4p(int rec_nr,
            double r1_U,double r2_U,double r3_U,double r4_U,
            double r1_D,double r2_D,double r3_D,double r4_D,
            Port_List pl)
```

The function `opm_bcurv4p` paints the surface limited by two Bezier curves of four points where the upper curve is defined in polar coordinates by (Π/2,r1_U), (0.66 × Π/2,r2_U), (0.33 × Π/2,r3_U), (0.0,r4_U) and the lower curve by (Π/2,r1_D), (0.66×Π/2,r2_D), (0.33×Π/2,r3_D),(0.0,r4_D). The visible range of the radius `r_...` arguments is (0.0,1.0). `rec_nr` is the resolution level where 10 is the top manual resolution and -1, -2 , -3 and -4 are the automatic resolution for the draft, standard, high and ultra level respectively.

Figure C.2 shows the image produced by the four `opm_bcurv4p` examples of the next program:

```
main(){
    Port_List lp=opm_page(Black,Gray65,NULL),lp1,lp2,lp3,lp4;
    lp1=opm_scale(1.0,0.45,0.9,0.45,lp);
    lp2=opm_scale(0.45,1.0,0.9,0.45,lp);
    lp3=opm_scale(1.0,0.45,0.45,0.9,lp);
    lp4=opm_scale(0.45,1.0,0.45,0.9,lp);
opm_print(opm_set_color(1,opm_concat(lp1,opm_concat(lp2,
   opm_concat(lp3,lp4)))));
opm_print(opm_bcurv4p( 4, 1.0, 0.2, 1.0, 0.0,
                        1.0, 0.0, 0.8, 0.0, lp1));
opm_print(opm_bcurv4p(-4, 1.0, 0.2, 1.0, 0.0,
                        1.0, 0.0, 0.8, 0.0, lp2));
opm_print(opm_bcurv4p(-3, 0.4, 1.2, 1.2, 0.4,
                        0.2, 1.0, 1.0, 0.2, lp3));
opm_print(opm_bcurv4p(-4, 1.0, 0.35,0.35,1.0,
                        0.5, 0.5, 0.5, 0.5, lp4));
}
```

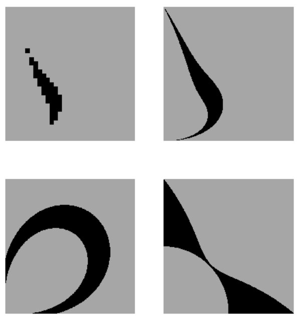

Figure C.2: 4 opm_bcurv4p examples

13. `opm_ellipsis`

```
Port_List opm_ellipsis(int rec_nr,double ring,double elipse,
                       double incl1,double incl2,Port_List pl)
    -3 <= rec_nr <= 12
   0.0 <= ring   <= 1.0
   0.0 <= elipse <= 1.0
  -1.0 <= incl1  <= 1.0
  -1.0 <= incl2  <= 1.0
```

`opm_ellipsis` paints the surface delimited by the maximal ellipsis of the port and the ellipsis defined by `ring`. The `elipse` argument is an elliptical factor. `incl1` and `incl2` sets the inclination of the major and minor ellipsis respectively. `rec_nr` is the resolution level where 10 is the top manual resolution and -1, -2 , -3 and -4 are the automatic resolution for the draft, standard, high and ultra level respectively.

Figure C.3 shows the image produced by the four `opm_ellipsis` examples of the next program:

```
main(){
  Port_List lp=opm_page(Black,Gray65,NULL),lp1,lp2,lp3,lp4;
  lp1=opm_scale(1.0,0.45,0.9,0.45,lp);
  lp2=opm_scale(0.45,1.0,0.9,0.45,lp);
  lp3=opm_scale(1.0,0.45,0.45,0.9,lp);
  lp4=opm_scale(0.45,1.0,0.45,0.9,lp);
  opm_print(opm_set_color(1,opm_concat(lp1,opm_concat(lp2,
      opm_concat(lp3,lp4)))));
  opm_print(opm_ellipsis(-3, 0.0, 0.0, 0.0, 0.0,  lp1));
  opm_print(opm_ellipsis( 5, 0.90,0.40,0.90,0.20, lp2));
  opm_print(opm_ellipsis(-2, 0.55,0.20,0.30,-0.7, lp3));
  opm_print(opm_ellipsis(-4, 0.60,0.20,-0.8,1.00, lp4));
  }
```

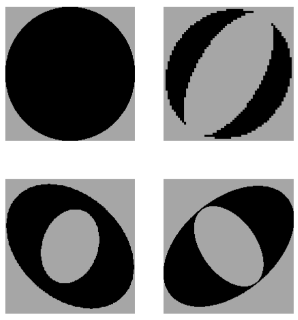

Figure C.3: 4 opm_ellipsis examples

14. `opm_sector`

```
Port_List opm_sector(int recnr,
                double angle1, double x1,
                double angle2, double x2,
                Port_List lp);
0.0 <= x1 <= 1.0
0.0 <= x2 <= 1.0
```

`opm_sector` paints the intersection of the right plane sector defined by the line `line1` and the left plane sector defined by the line `line2`. `line1` and `line2` are defined by the X coordenates `x1` and `x2` with the angles in radians `angle1` and `angle2` respectively. `rec_nr` is the resolution level where 10 is the top manual resolution and -1, -2 , -3 and -4 are the automatic resolution for the draft, standard, high and ultra level respectively.

Figure C.4 shows the image produced by the four `opm_sector` examples of the next program:

```
main(){
Port_List lp;
lp=opm_page(Black,Gray85,Beige, NULL);
    opm_print(opm_set_color(1,lp));
opm_print(opm_sector(6, M_PI_2-0.07,0.00,
                        M_PI_4     ,0.00,
                        opm_scale(1.0,0.5,1.0,0.5,lp)));
opm_print(opm_sector(-1,M_PI_2+0.3, 0.50,
                        M_PI_2-0.3, 0.50,
                        opm_scale(0.5,1.0,1.0,0.5,lp)));
opm_print(opm_sector(6, M_PI_2+M_PI_4, 1.00,
                        M_PI_2+0.1   , 1.00,
                        opm_scale(1.0,0.5,0.5,1.0,lp)));
opm_print(opm_sector(-1,0.95, 0.00,
                        0.95, 0.05,
                        opm_scale(0.5,1.0,0.5,1.0,lp)));
}
```

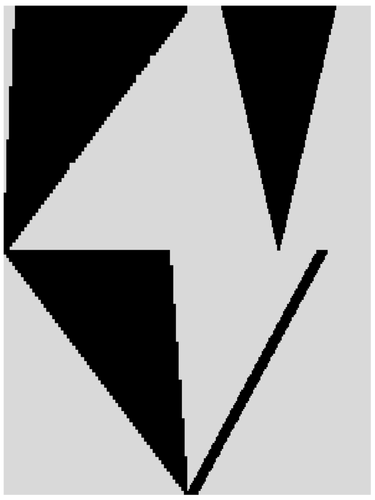

Figure C.4: 4 opm_plane_sector examples

-

-