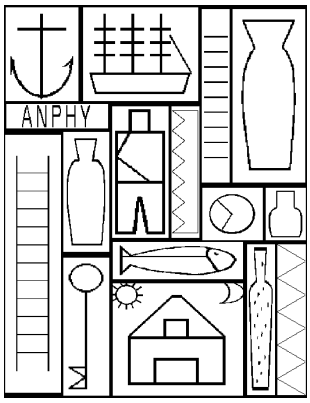
Typed Windows
An Implementation of a Programming Language for Graphics Design

Technical Report INCO-97-02



Master Thesis

Pablo J. Queirolo
Instituto de Computación
Facultad de Ingeniería, Universidad de la República
5º piso. Julio Herrera y Reissig 565
Montevideo, Uruguay
email: queirolo@fing.edu.uy

May 1997

The author has been supported by the Pedeciba and the Conicyt.

Abstract

This paper presents an implementation of ***TyWin***[1], a system for constructive 2D graphic design proposed by Juan José Cabezas [Cab91].

TyWin extends the usual concept of window in Computer Graphics, associating a type with every window in the system. Then, the displaying rules (graphic representation rules) for an object in a certain window depends of its type.

The displaying rules were designed inspired on the "Constructive Universalism" of the Uruguayan painter Joaquín Torres García and the type system of ***TyWin*** takes as a theoretical framework the Type Theory of the Swedish mathematician Per Martin-Löf.

Two main modules of the system were implemented, a programming language interpreter for 2D graphic design and a system to define libraries of "icons" (or ideograms).

In order to increase the power of the system, the original language proposed in [Cab91] was extended.

The result is an environment for 2D graphic design, with some original features which are shown through a collection of examples including paintings of Joaquín Torres García.

[1] "Typed Windows"

Contents

List of Figures

1 Introduction

[2]

1.1 Typed Windows

TyWin is a system for constructive 2D graphic design proposed by Juan José Cabezas [Cab91]. The main idea of the system is to generalize the usual concept of window in Computer Graphics.

This concept can be described as follows:

- " ... specifying a window in the world-coordinate space surrounding the information we wish displayed." [NS83]
- "In addition to the window, we can define a port (or viewport), a rectangle on the screen where we would like the windows contents displayed." [NS83]
- "We use the window to define what we want to display; we use the port to specify where on the screen to put it". [NS83]

However, the type of the objets of a certain window is usually limited to a cartesian product of subranges of real or integer numbers. As a consequence, when programming with graphic objects of different types, it is necessary to reduce (translate) them to a list of objects of the window type, before they could be displayed.

From a methodological point of view, it would be desirable that for every object of a certain type $\mathcal{A}$ in the 'graphics' universe, there exists a window of type $\mathcal{A}$ which accepts that object for displaying. A window system with this property can increase the quality of the programming environment in Computer Graphics, extending to the graphics area the power of type systems.

We call this generalized concept of window '*typed window*', and '***TyWin System***' a complete environment for graphic design based on *typed windows*.

Therefore the ***TyWin System*** associates graphic display rules with types so that the task of displaying an object of type $\mathcal{A}$ on a window (associated with the type $\mathcal{A}$) is determined by the graphic rule of $\mathcal{A}$.

The type system of ***TyWin*** has been designed taking as theoretical framework the Type Theory of the Swedish mathematician Per Martin-Löf.

Another expression of the constructive movement of this century was taken as guide when designing the graphic representation rules of ***TyWin***: the "Constructive Universalism" of the Uruguayan painter Joaquín Torres García.

[2] In order to introduce the foundations of the Typed Windows System, some paragraphs of the Juan José Cabezas's report are taken.

He defined an art conception that stands out for understanding the constructive painting like a structure of symbols. [Lin92]

As a consequence, ***TyWin*** does not support the concepts of foreground and background. Torres García said: "*In the unity of the composition, the idea of thing and background should disappear. ... Then, there are not the thing and the background, all is thing and all is background.*"[Gar69].

Furthermore the usual concept of port in Computer Graphics is also generalized: in the ***TyWin System*** a port is a rectangular oriented region including colour information that covers the usual concepts of port and pixel.

Then, when an object is displayed, the port defined for displaying is transformed into new ports. ***TyWin*** translate ports into ports, allowing that the output of a certain transformation could be the input of another.

In conclusion, the *Typed Windows* project takes two constructive expressions of the current century and based on them develops an original system to support 2D graphic design.

1.2 Martin-Löf's Type Theory

The Type Theory introduced by the Swedish mathematician Per Martin-Löf is a formalization of the constructive mathematics with concepts and properties relevant to Computer Science.

In this theory, the idea of specification and program can be associated with the idea of type and element respectively. The theory allows to express both using the same language and to formally verify the correctness of a program to its specification.

The base of the type theory is introduced in [NPS90] as follows:

"The judgement $a \in A$ in type theory can be read in at least the following ways:

- *a is an element in the set A.*
- *a is a proof object for the proposition A.*
- *a is a program satisfying the specification A.*
- *a is a solution to the problem A.*

The reason for this is that the concepts set, proposition, specification and problem can be explained in the same way."

Then, the *typed window* concept can be introduced by means of extending the previous list with:

- *a is an object that can be displayed in the window A.*

At the same time, this extension makes possible to apply this theory in Computer Graphics (Figure 1).

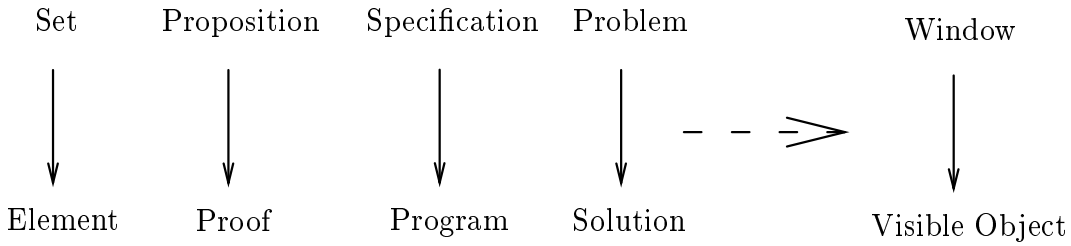


Figure 1: *Window - Visible Object* relation.

1.3 Torres García's Constructive Universalism

The Uruguayan painter Joaquín Torres García (July 1874, August 1949) defined an art conception that he called "Constructive Universalism".

This conception stands out for understanding the constructive work like a structure of symbols. [Lin92]

Torres García said: "*The concept of construction like symbolic operation, reducing every figure to sample 'ideograms'* [3] *that combined with archaic signs, numbers and other writings, tries to construct a synthesis of the universe like totality*". [Gar69]

His ideas made a constructive view of the "graphic universe" and can be interpreted (modelled) with reasonable simplicity from the mathematical point of view. Furthermore, they provide a conceptual guide to define a programming language for Computer Graphics.

Joaquín Torres García defined his art movement based on two concepts:

Structure : in order to give a unity to the construction ("*Colour planes and lines combined with art, will build a real structure.*" [Gar69]).

Abstraction : since he rejected the imitation of nature, he defined ideograms representing things and ideas in order to use universal representations ("*The painter is not interested in the object, he is interested in the colour plane and the geometry of his structure.*" [Gar69]).

Avoiding the hand drawing model, Torres García proposed a constructive painting based on a composition (structure) of rectangles (planes of colour) and stamped ideograms. This is the model followed in the ***TyWin System*** to define constructive 2D designs.

In his paintings we can find **structures**, **ideograms** and **planes of colour**. In the ***TyWin System*** these concepts are implemented by **windows (and views)**, **values (graphic objects)** and **ports**.

[3] He defines an ideogram like the simplest figure that represents a particular thing or idea.

1.4 The *TyWin* Implementation Project

This paper addresses the implementation of two main modules of the ***TyWin System***. These modules are a programming language interpreter for 2D graphic design and a system to define libraries of "figures" (or ideograms) to use in the language. Then, in this stage of the system implementation the user can edit "figures" for using in his designs and make programs to define and display 2D designs.

Outline of the paper

This paper is organised as follows. Section 2 addresses the graphical meaning of types. Sections 3 and 4 addresses the ***TyWin*** project. Section 5 addresses the ***TyWin Language*** and its different layers are presented in Sections 6, 7, 8, 9, 10 and 11. Section 12 tells how the ideograms were introduced into the system. Section 13 shows some examples. Section 14 presents the conclusions of this thesis.

2 Graphical Meaning of Types

The **Typed Windows System** associates graphic representation (display) rules with types. Then, if α is a canonical object of type $\mathcal{A}$, there is a graphic rule **R** that determines how to display α on a window (of type) $\mathcal{A}$.

In this section, the graphic rules are informally introduced with the help of figures. However, there are some differences between the basic rules introduced in this section and the rules implemented in the ***TyWin System***[4].

It was defined displaying rules for the following types in ***TyWin***:

- T
- Bool
- Nat
- Real
- List
- Set
- Disjoint Union
- Cartesian Product
- Record
- Function

The rules of representation were first defined for some of these types in [Cab91]. However, some rules were modified (Bool and List) and other were added (Real, Set and Record).

These display rules transform rectangular regions into new rectangular regions. Each rectangular region has associated an orientation[5] and colour information.

2.1 T

The rule corresponding to the **T** type is shown in Figure 2. For this type the result is the original rectangle.

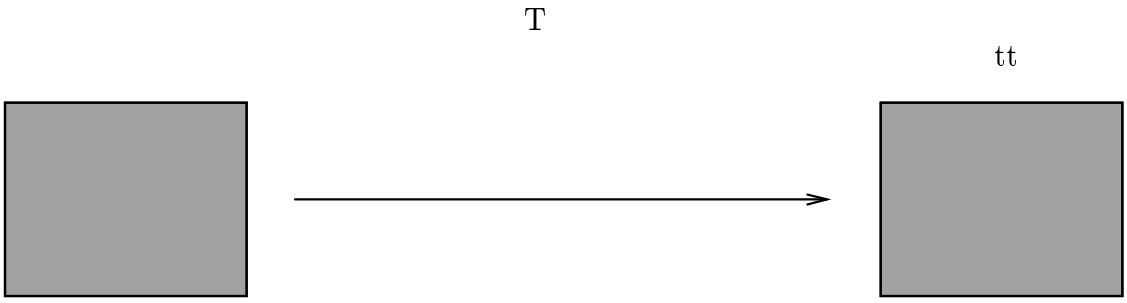


Figure 2: T display rule.

[4] See Section 10 on page 40 for a complete description of the rules implemented in the ***TyWin System***.

[5] There are four possible orientations: `NORTH`, `EAST`, `SOUTH` and `WEST`.

2.2 Bool

The rule corresponding to the **Bool** type is shown in Figure 3. The result is the original rectangle when the boolean value is *True* and a *null* rectangle[6] if the boolean value is *False*.

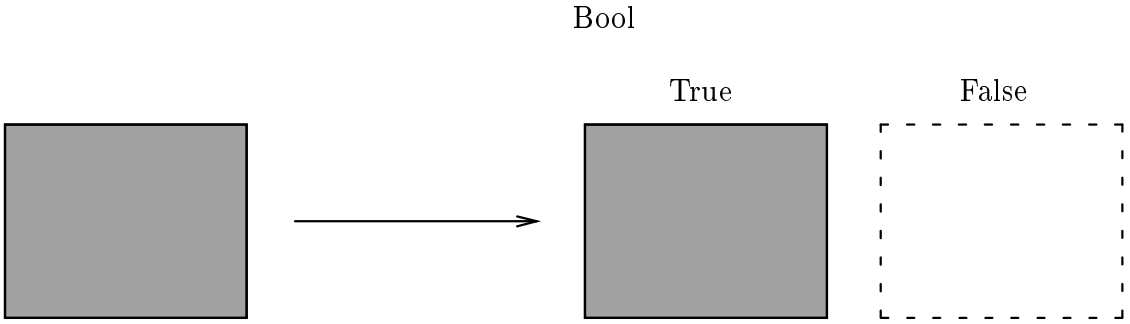


Figure 3: Bool display rule.

2.3 Nat

The rule corresponding to the **Nat** type is shown in Figure 4. In the ***TyWin System*** the **Nat** type has a range associated with it. This range defines which natural numbers can be visible. For example in the range (12 .. 18) only the natural numbers between 12 and 18 are visible. The operator splits the original rectangle in as many identical sub-rectangles as natural numbers accepts the range. The original rectangle is split in function of its orientation. Then, the sub-rectangle that matches the natural number is selected.

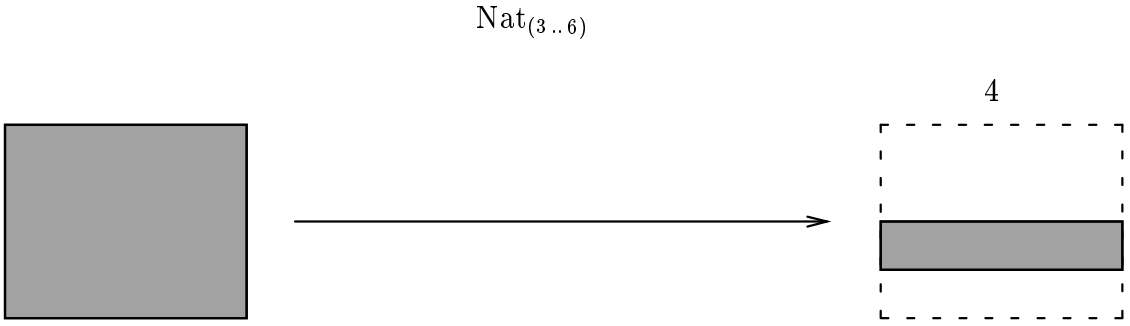


Figure 4: Nat display rule.

For displaying the natural number 4, the original rectangle is split in four identical sub-rectangles (since the range (3 .. 6) accepts four natural numbers) and the second one is returned (the natural number 4 matches the second natural number of the range). The original rectangle in the example has a `NORTH` orientation.

[6] A *null* rectangle is a rectangle with area 0. This concept is useful in order to have some properties in the operators defined in the system. See Section 6.2 on page 30.

2.4 Real

The rule corresponding to the **Real** type is shown in Figure 5. The operator splits the original rectangle proportionally with the real number in function of its orientation.

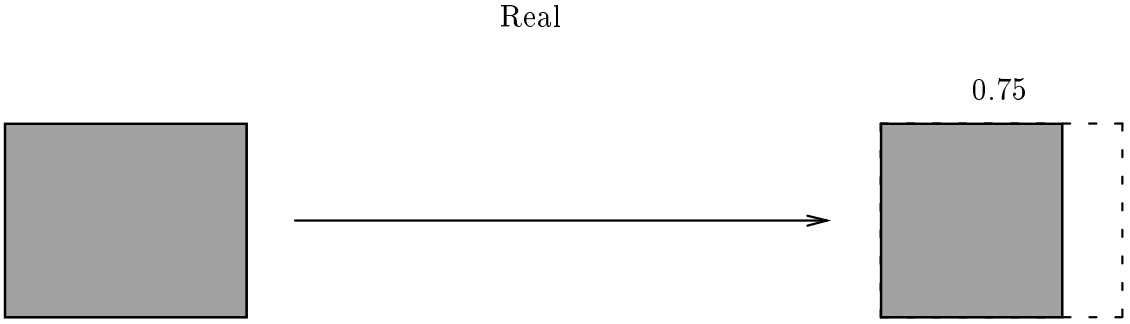


Figure 5: Real display rule.

In the example, the real number is 0.75, then the result is a sub-rectangle which size is three quarters of the original one. The original rectangle in the example has an `EAST` orientation.

2.5 List

The rule corresponding to the **List** type is shown in Figure 6. First the original rectangle is split in as many identical sub-rectangles as elements are in the list value (*length*(*list-value*)) in function of its orientation. Finally, the first element of the list is displayed in the first sub-rectangle; the second element of the list is displayed in the second sub-rectangle; and so on.[7]

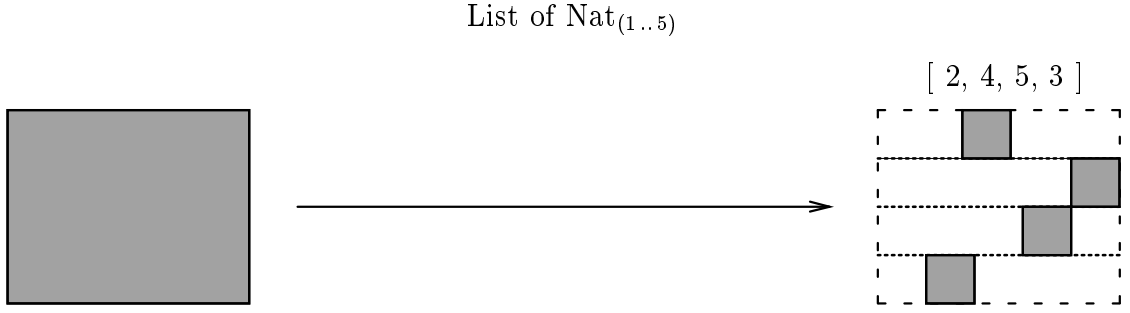


Figure 6: List display rule.

In the example, the list has four elements, so the original rectangle is split in four identical sub-rectangles. Then, the first element of the list (the natural number 2)) is displayed in the first sub-rectangle using the range (1 .. 5), the second element

[7] This rule was extended during the implementation of the ***TyWin System***. See Section 10.2.1 on page 41 for a complete description of the rule implemented for the **List** type in the ***TyWin System***.

of the list (the natural number 4) is displayed in the second sub-rectangle using the range (1 .. 5) and so on. The original rectangle in the example has a `NORTH` orientation and the sub-rectangles have an `EAST` orientation.

2.6 Set

The rule corresponding to the **Set** type is shown in Figure 7. Each element of the set is displayed in the original rectangle. Then the result are these sub-rectangles. When the set is empty, the result is a *null* rectangle.

Figure 7: Set display rule.

In the example, the four natural numbers of the set are displayed in the original rectangle using the range (1 .. 8) and the result are the four sub-rectangles. The original rectangle in the example has a `SOUTH` orientation.

2.7 Disjoint Union

The rule corresponding to the **Disjoint Union** type is shown in Figure 8. This rule is applied to two rectangles[8]. When the union value corresponds to the left sub-type of the union, the result is the representation of the sub-value in the first rectangle. When the union value corresponds to the right sub-type of the union, the results is the representation of the sub-value in the second rectangle.

In the example, the first case displays the natural number 2 using the range (1 .. 4) and the first rectangle and the second case displays the real number 0.75 using the second rectangle. Both of the original rectangles in the example have a `NORTH` orientation.

2.8 Cartesian Product

The rule corresponding to the **Cartesian Product** type is shown in Figure 9. This rule is applied to two rectangles. First, the sub-rectangle associated with the left element of the pair is found in the first rectangle, then the sub-rectangle associated

[8]Both rectangles could be the same.

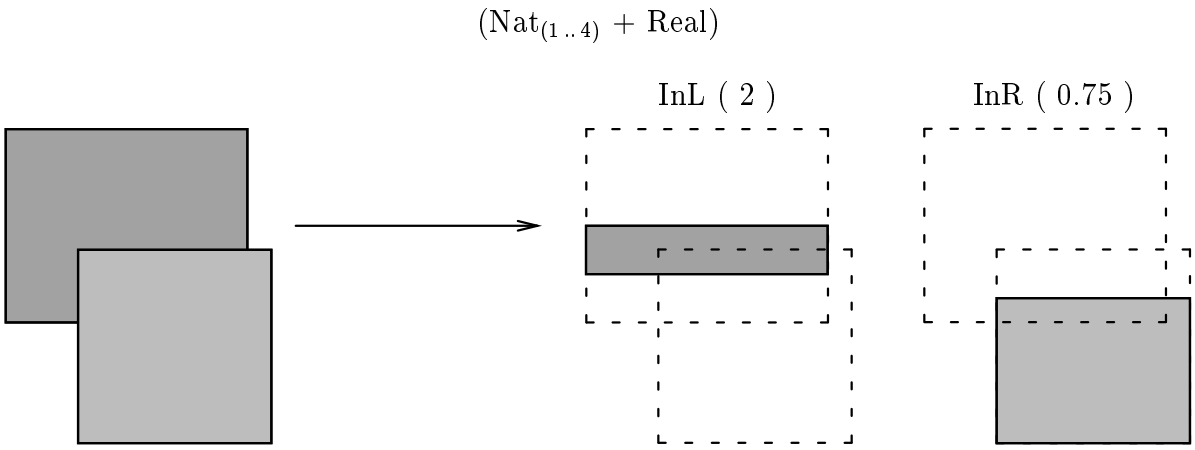


Figure 8: Disjoint Union display rule.

with the right element of the pair value is found in the second rectangle and finally the result is the intersection of both sub-rectangles.

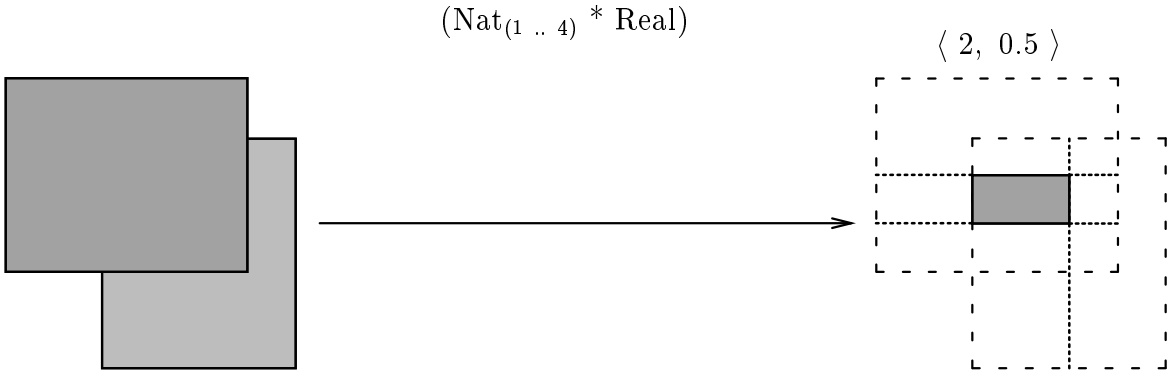


Figure 9: Cartesian Product display rule.

In the example, the first element of the pair, the natural number 2 is displayed using the range (1 .. 4) and the first rectangle. Then, the second element of the pair, the real number 0.5 is displayed in the second rectangle. Finally, both sub-rectangles are intersected. In the example the first original rectangle has a `NORTH` orientation and the second has an `EAST` orientation.

2.9 Record

The rule corresponding to the **Record** type is shown in Figure 10. This rule is applied to a collection of rectangles of different colours but the same position, size and orientation. In this case, the first element of the record is displayed in the first rectangle of the collection, the second element of the record is displayed in the second rectangle of the collection, and so on.

In the example, each element of the record is displayed using the range (1 .. 3)

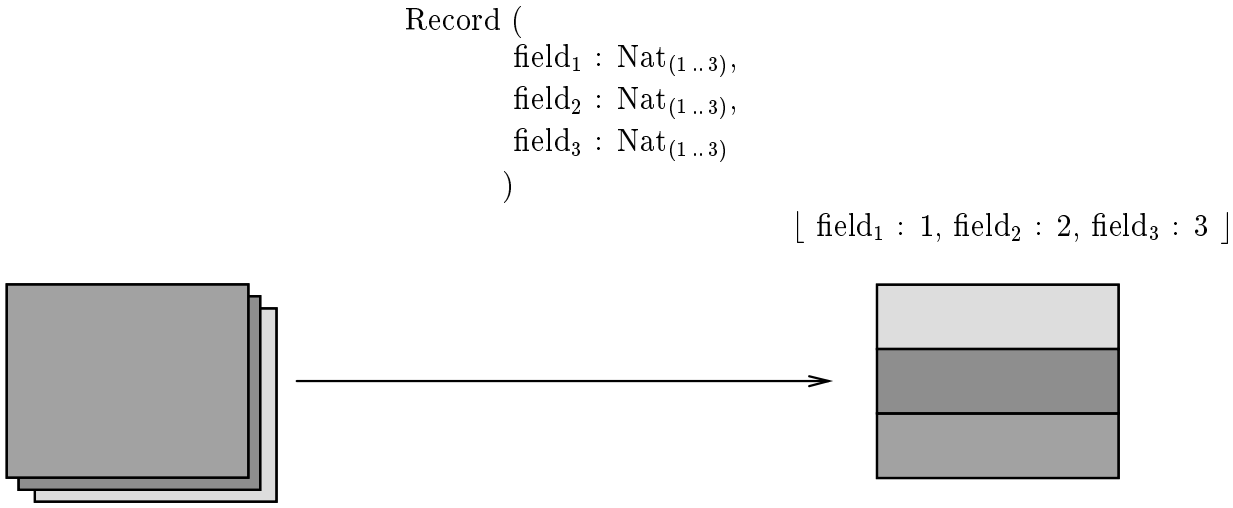


Figure 10: Record display rule.

and the corresponding coloured rectangle of the collection. The first element of the record, the natural number 1, is displayed using the range (1 .. 3) and the first rectangle of the collection; the second element of the record, the natural number 2, is displayed using the range (1 .. 3) and the second rectangle of the collection and finally the third element of the record, the natural number 3, is displayed using the range (1 .. 3) and the third rectangle of the collection. Finally, the result are these three sub-rectangles. The original rectangles in the example have a `NORTH` orientation.

2.10 Function

The rule corresponding to the **Function** type is shown in Figure 11. For this type the result is a *null* rectangle.

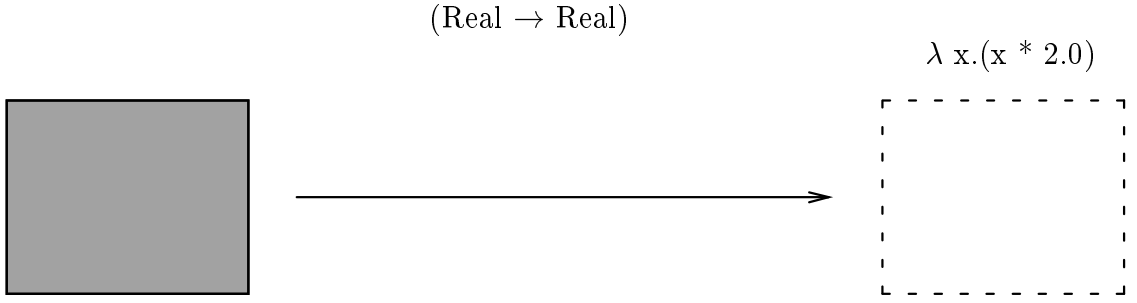


Figure 11: Function display rule.

No rule was found natural to be associated with functions. This type is absolutely different from the ones introduced previously and any rule associated to it will be extremely artificial. These are the reasons for selecting so simple rule.

3 The *TyWin* Project

The ***TyWin System*** is a 2D graphic design environment, that includes graphic editors and languages in order to allow the user to comfortably define constructive designs.

The architecture of the system is composed by the modules shown in Figure 12.

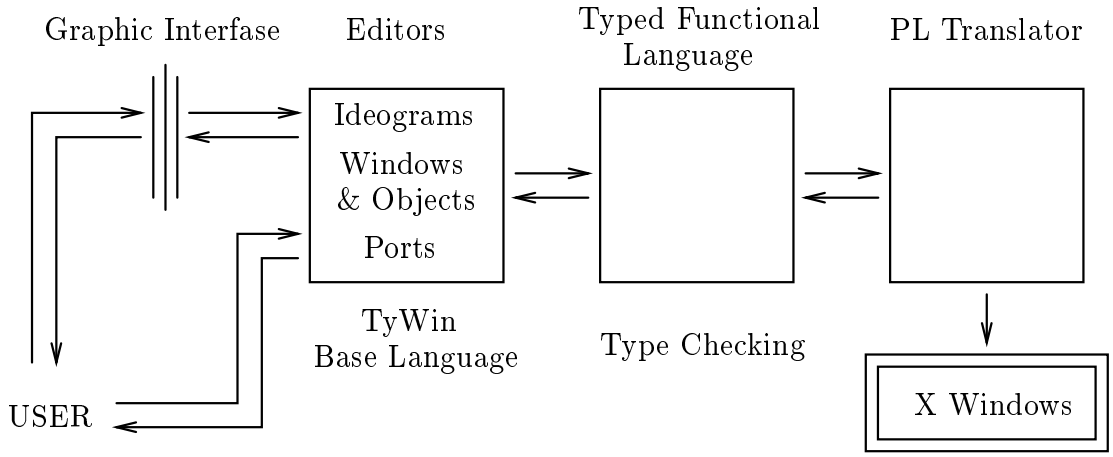


Figure 12: The ***TyWin System*** viewed by the implementator (taken from the system report of Cabezas).

The user defines 2D designs using graphic editors that offer a comfortable development environment. These editors define types and values of a functional language in order to use them in programs. The types of these objects are checked by the *Typed Functional Language* module, then the checked program is interpreted by the *PL Translator*[9] module and the graphic output is sent to the X-Windows System.

The modules considered are:

Ideograms Editor : to define icons (*ideograms*) and store them in the ***TyWin*** format.

Windows and Objects Editor : to define types and values of the Typed Functional Language.

Ports Editor : to define rectangular regions (*ports*) of the main output window.

Typed Functional Language : to check that types of windows and values match, and to add functionality to the language with statements which are not available in the ***TyWin Language***[10].

PL Translator : to convert the graphic data of the ***TyWin System*** into graphic output in the X-Windows System.

[9] Port Language Translator

[10] Since 1994 Juan José Cabezas has been implementing a *Typed Functional Language* (called **"bamba"**) for the ***TyWin System***.

4 The *TyWin* Implementation Project

In order to check some hypotheses about the ***TyWin System*** a first prototype was implemented.

These hypotheses are:

- The rules defined for the graphic representation of the different types work and interact each other well.
- The system can represent 2D figures and designs like Torres García's paintings.

Two modules of the system were implemented, the *PL Translator* module and the *Ideograms Editor* module. The subjects of this document are these implementations and the evaluation of the obtained 2D graphic design environment.

4.1 The PL Translator module

The *PL Translator* module interacts with the *Typed Functional Language* module and produces the graphic output of the system (as shown in Figure 12).

Since the *PL Translator* module interacts with the *Typed Functional Language* module, the ***TyWin Language*** implemented is really simple (the language includes four different statements and it only deals with canonical expressions), since in the *Typed Functional Language* module the user has available functions, conditionals and any common primitive of a functional language.

This module works in function of the rules defined to the graphic representation of the types considered in this implementation.

These types are:

- T
- Boolean
- Integer
- Real
- List
- Set
- Disjoint Union
- Cartesian Product [11] [12]
- Record

4.2 The Ideograms Editor module

The *Ideograms Editor* module is composed by two sub-modules:

[11] Even when we introduce the Nat type in Section 2.3 on page 14, during the implementation of the modules the Integer type was selected (then the system deals with negative values as in the usual programming languages).

[12] The Function type was not introduced in this implementation.

Graphic Ideogram Editor :
is a basic graphic image editor implemented to define ideograms with five-planes of colour. The editor asks for the height and width of the ideogram, and shows a grid where the user switches on the elements of the image.

Image Translator :
converts X-images (X-Bitmap or X-Pixmap) to the ***TyWin*** format. Then, the user can use his icon editors, and can export pre-defined images to ***TyWin***.

5 The *TyWin Language*

The ***TyWin Language*** is built up as a herarchical structure of four languages. The structure of the ***TyWin Language*** is shown in Figure 13.

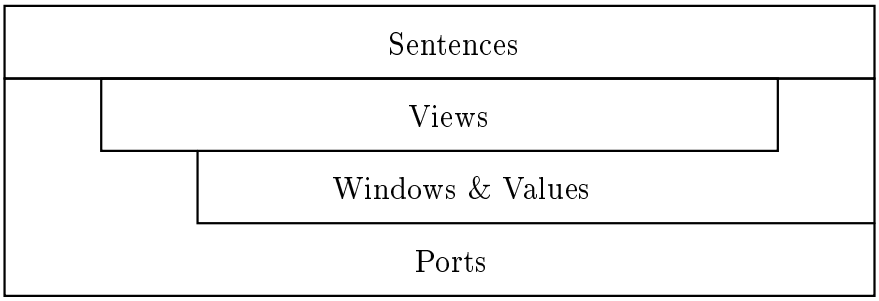


Figure 13: The structure of the ***TyWin Language***

Each layer of the language is introduced in the following sections.

5.1 The Ports Layer

The concept of *port* is the core of the ***TyWin System*** and this layer is really the kernel of the language. Every other object will be reduced to a *port* in order to be displayed. See Section 6 on page 28 for a complete description of this layer.

In ***TyWin*** a *port* object can be an instance of a *simple port* or an instance of a *port list*.

- A *simple port* in the language is a rectangular region (with its edges aligned with the screen edges) with an orientation and some colour information. A *simple port* is defined by a vector (which defines the region and orientation of the *simple port*) and the associated colors.
- A *port list* is a list of *simple ports*.

Observations

1. The rectangular region and the orientation of a simple port is defined by a vector. A vector in the system is defined as a pair of coordinates. For example the pair `(<0, 0>|<WIDTH, HEIGHT>)` defines the vector where the coordinate `<0, 0>` (the top-left corner of the screen) and the coordinate`<WIDTH, HEIGHT>` (the bottom-right corner of the screen). Then the vector `(<0, 0>|<WIDTH, HEIGHT>)` defines a rectangle that covers all the screen.
2. The colour of a simple port is defined by the RGB[13] model, for example the red colour is `<MAX, MIN, MIN>` and the blue colour is `<MIN, MIN, MAX>`.

[13] Red-Green-Blue.

Examples

- `port((<0, 0>|<WIDTH, HEIGHT>), [<MID, MID, MID>])`: a simple port that covers all the screen (the vector `(<0, 0>|<WIDTH, HEIGHT>)`) and with the grey colour associated (`[<MID, MID, MID>]`).
- `port((<WIDTH, HEIGHT>|<(WIDTH/2), (HEIGHT/2)>), [<MAX, MIN, MIN>])`: a simple port that covers the bottom-right quarter of the screen (the vector `(<WIDTH, HEIGHT>|<(WIDTH/2), (HEIGHT/2)>)`) and with the red colour associated (`[<MAX, MIN, MIN>]`).
- `Nil`: an empty port list.
- `(port((<(WIDTH/2), 0>|<WIDTH, (HEIGHT/2)>), [<MIN, MAX, MIN>]) # (port((<(WIDTH/2), HEIGHT>|<0, (HEIGHT/2)>), [<MIN, MAX, MAX>]) # Nil))`: a port list composed by two simple ports:
  - The first simple port covers the up-right quarter of the screen (the vector `((WIDTH/2), 0>|<WIDTH, (HEIGHT/2))`) and with the green colour associated (`[<MIN, MAX, MIN>]`).
  - The second simple port covers the bottom-left quarter of the screen (the vector `(<(WIDTH/2), HEIGHT>|<0, (HEIGHT/2)>)`) and with the yellow colour associated (`[<MIN, MAX, MAX>]`).

5.2 The Windows & Values Layer

5.2.1 The Values

The ***TyWin System*** assigns a method to display any object of the types introduced in Section 4.1 on page 20. These objects are canonical values of the *Typed Functional Language* module. See Section 7 on page 35 for a complete description of this sublayer.

Examples

- `tt`: the object of the T type.
- `True`: an object of the Boolean type.
- `7`: an object of the Integer type.
- `0.45`: an object of the Real type.
- `[ 3 : 2 : 8 ]`: an object of the List type.
- { `True : True : False` }: an object of the Set type.
- `InR(-5)`: an object of the Disjoint Union type.

- `(tt, 9.3)`: an object of the Cartesian Product type.
- `<12::2.8::True>`: an object of the Record type.

5.2.2 The Windows

In ***TyWin*** the method used to display graphically an object (or value) is associated with the type of the object. The windows are defined as extensions of the different types in the language. Then, the methods to display objects are associated with the windows. See Section 8 on page 37 for a complete description of this sub-layer.

Observations

1. The Integer type is extended with a range as it was introduced in Section 2.3 on page 14 for the Nat type. The window `Integer(1..5)` can display any integer, but only the values between 1 and 5 should be visible in the output. See Section 10.1.4 on page 40 for a complete description of how the integer objects are displayed.
2. The List type is extended with a range and an integer in order to define which sub-values of the list are displayed and wich orientation is used. The window `List of ((2..4), 0, Boolean)` can display the second, third and fourth sub-values of the list using the same orientation of the original port. See Section 10.2.1 on page 41 for a complete description of how the list objects are displayed.

Examples

- `T`: a window for the T type.
- `Boolean`: a window for the Boolean type.
- `Integer (3..4)`: a window for the Integer type with the range `(3..4)`.
- `Real`: a window for the Real type.
- `List of ((2..5), 0, Boolean)`: a window for the List type.
- `Set of Real`: a window for the Set type.
- `(Boolean + Integer (1..100))`: a window for the Disjoint Union type.
- `(T * Real)`: a window for the Cartesian Product type.
- `Record (Integer (1..10) :: Real :: Boolean>`: a window for the Record type.

5.3 The Views Layer

In the ***TyWin Language*** a view is a pair $\langle window,\ port \rangle$. Then views have the information needed to represent graphically every object or value. These objects are canonical values of the *Typed Functional Language* module[14]. See Section 9 on page 38 for a complete description of this layer.

Examples

- `ToView(Boolean, port((<0, 0>|<WIDTH, HEIGHT>), [<MID, MID, MID>]))`: a view to display boolean values in the port `port((<0, 0>|<WIDTH, HEIGHT>), [<MAX, MAX, MIN>]))`.
- `ToView(Integer(1..10), port((<WIDTH, HEIGHT>|<(WIDTH/2), (HEIGHT/2)>), [<MAX, MIN, MAX>]))`: a view to display integer numbers in the port `port((<WIDTH, HEIGHT>|<(WIDTH/2), (HEIGHT/2)>), [<MAX, MIN, MAX>]))`.

5.4 The Sentences Layer

The sentences included in the system are:

Assignment :
associates a new value with an identifier.

Graphics Display :
sends the graphic data to the X-Windows System.

File Inclusion :
includes and processes a pre-defined ***TyWin*** program.

Free Identifier :
deletes an identifier of the environment and frees its associated memory.

See Section 11 on page 45 for a complete description of this layer.

Examples

- `a1 = port((<0, 0>|<WIDTH, HEIGHT>), [<MIN, MAX, MIN>]);`

  It assigns the port `port((<0, 0>|<WIDTH, HEIGHT>), [<MIN, MAX, MIN>])` to the identifier `a1`.

- `#include "figure1.fig";`

  It includes the definitions in the `"figure1.fig"` file.

[14] The view has the method of displaying in the window and the place of the screen where to display in the port.

- `display(port((<0, 0>|<WIDTH, HEIGHT>), [<MAX, MAX, MIN>]));`

  It displays the port `port((<0, 0>|<WIDTH, HEIGHT>), [<MAX, MAX, MIN>])`.

- `free(a1);`

  It frees the memory assigned to the identifier `a1`.

Example of a complete program

A simple example is introduced to show how the language is structured.

```
display(                                          || Display a port.
       ToPort(                                    || Define a port.
             ToView(                              || Define a view.
                   Integer(1..5),                 || Window (or type).
                   port(                          || Create a port.
                       (<0, 0>|<WIDTH, HEIGHT>),  || Vector (or Region).
                       [<MID, MID, MID>]          || Colour.
                       )
                   ),
             2                                    || Value.
             )
        );
```

The screen is split in five rectangles and the second is shown in grey, as can be seen in Figure 14.

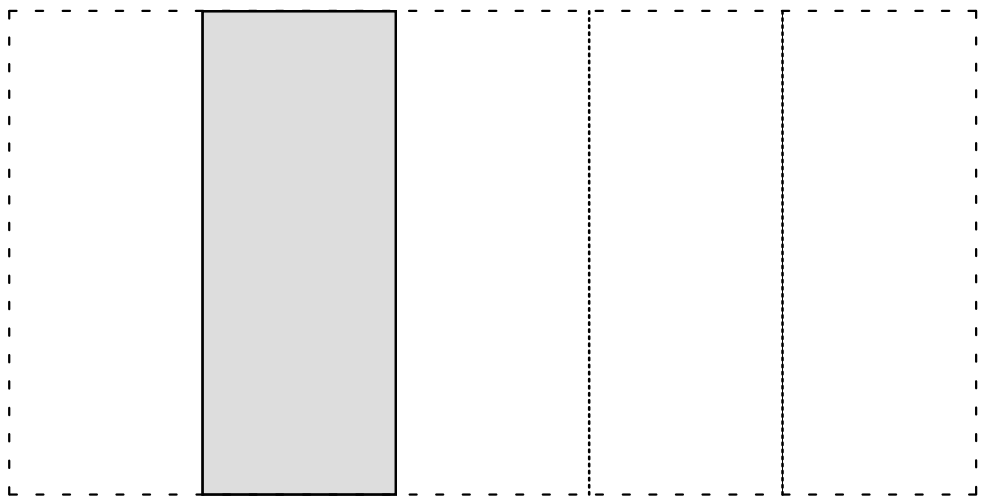


Figure 14: Example output.

Observations

1. The `ToView` operator receives a port and a window and returns a view.
2. The `ToPort` operator receives a view and a value and returns a port. This is the main operator of the system. It translates the value into graphic information in function of the view.

6 The Ports

The concept of *port* is the core of the ***TyWin System*** and this layer is really the kernel of the language. Every other object will be reduced to a *port* in order to be displayed.

In ***TyWin*** a *port* object can be an instance of a *simple port* or an instance of a *port list*. A *simple port* in the language is a rectangular region with an orientation and some colour information. A *port list* is a list of *simple ports*.

6.1 Definition of ports

6.1.1 Simple ports

A *simple port* in the language is a rectangular region (with its edges aligned with the screen edges [15]) with an orientation and some colour information. The region and the orientation are defined by a vector (a pair of coordinates). How the vector represents the region and one of the four possible orientations (`NORTH`, `EAST`, `SOUTH` or `WEST`) is shown in Figure 15. The orientation of a *simple port* is used in the semantic of the main operators.

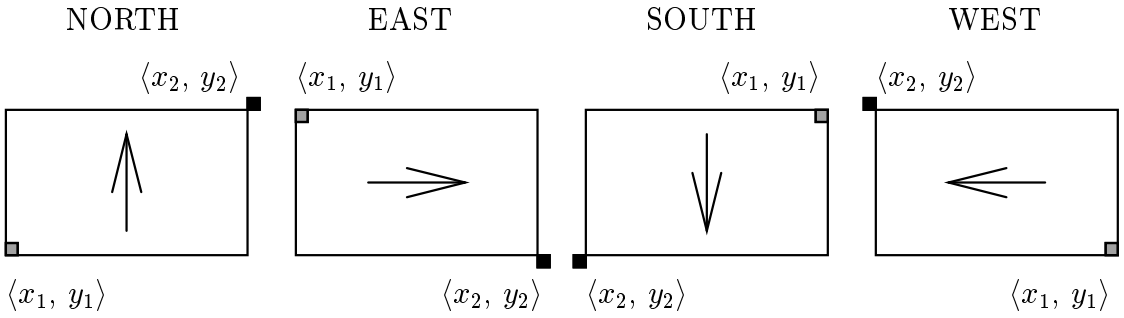


Figure 15: The Ports Orientations defined in function of the different variations of the vector $(\langle x_1, y_1\rangle|\langle x_2, y_2\rangle)$

The colour information is defined by a record of five different planes of colour. Then five colours allow the user to manage five planes in a *simple port*. The display of a port on screen is characterized by its associated planes of colour.

In conclusion, a *simple port* is composed by a vector (or rectangular region with an orientation) and the planes of colour.

[15] Joaquín Torres García said about the wall paintings: *"The dominant lines in the architecture are always the vertical and the horizontal. Circles, semicircles, parabolas, angles and oblique lines are always generated in function of them."* [Gar69]

simple_port	Constructor:	○	Port (*vector* , *planes*)
vector	Constructor:	○	(*position* | *position*)
position	Constructor:	○	⟨ *integer* , *integer* ⟩
planes	Constructors:	○	[*colour* : *colour* : *colour* : *colour* : *colour*]
		○	[*colour* : *colour* : *colour* : *colour*]
		○	[*colour* : *colour* : *colour*]
		○	[*colour* : *colour*]
		○	[*colour*]
colour	Constructor:	○	< *complevel* , *complevel* , *complevel* >
complevel	Constructors:	○	Min
		○	Mdn
		○	Mid
		○	Mdx
		○	Max

Observations

1. In order to deal with *simple ports* whose regions are only one pixel but with any of the four orientations, the vector origin is in the defined region but the destiny is out, as it is shown in Figure 15.

   When both positions in a vector have the same value, the *simple port* defined is null, since it has not got a region.

2. The language has five variants of colour planes definition. Then, the user can use less than five colour planes even when the ***TyWin System*** manages five planes of colours associated with a port.

   Each colour plane is defined by its [Red · Green · Blue] components. A colour component can take one of five different values[16]. Then, the system can manage $5^3 = 125$ colours.

6.1.2 Port lists

Port lists are introduced in the language for two reasons.

1. In order to allow the user to manage more complex figures modelled by ports. Every graphic image can be built using some coloured rectangles (if the image uses only the colours supported by the system).

2. If we work only with *simple ports* some basic operations considered in the language are not closed. For example the union or difference of two simple ports is not always another simple port. On the other hand, a list of rectangles is closed under these operations [17].

[16] Min:minimum; Mdn:middle-minimum; Mid:middle; Mdx:middle-maximum; Max:maximum

[17] This is the reason for not introducing other shapes in the port scheme. Think about the different results of the intersection between other shapes, for example two circles.

Then, a *port list* is a list of *simple ports*.

port_list	Constructors:	∘	Nil	
		∘	(*simple_port* # *port_list*)	
	Selectors:	∘	Head (*port_list*)	→*simple_port*
		∘	Tail (*port_list*)	→*port_list*
		∘	Length (*port_list*)	→*integer*

Some operators, like the selectors implemented on *port lists* are not needed in the prototype, but make the language more friendly for developing tests.

6.2 Operators of ports

The following table introduce the operators defined on the *port* class (composed by *simple ports* and *ports lists*).

port	Operators:	∘	Rotate (*port*)	→*port*
		∘	UnRotate (*port*)	→*port*
		∘	BiRotate (*port*)	→*port*
		∘	SplitInt (*port* , *range* , *integer*)	→*port*
		∘	SplitReal (*port* , *real*)	→*port*
		∘	(*port* + *port*)	→*port*
		∘	(*port* ∗ *port*)	→*port*
		∘	(*port* − *port*)	→*port*
		∘	SetColour (*port* , *planes*)	→*port*
range	Constructor:	∘	(*integer* .. *integer*)	

[18]

6.2.1 Rotate · BiRotate · UnRotate

The Rotate operator modifies the vector of the port, in order to change the orientation but preserves the region. How it works is shown in Figure 16.

The operator BiRotate applies two rotations and UnRotate applies three. Since the result of apply the operator Rotate four times is the *Identity*, the following equation is true in the system:

$$(\text{Rotate} \circ \text{UnRotate}) \equiv (\text{UnRotate} \circ \text{Rotate}) \equiv (\text{BiRotate} \circ \text{BiRotate}) \equiv \mathit{Identity}$$

[18] A range represents an interval defined by two integers, the first is the lower bound and the second the upper bound. However, if the second integer is lower than the first, an empty range is defined, and any integer will be out of the interval.

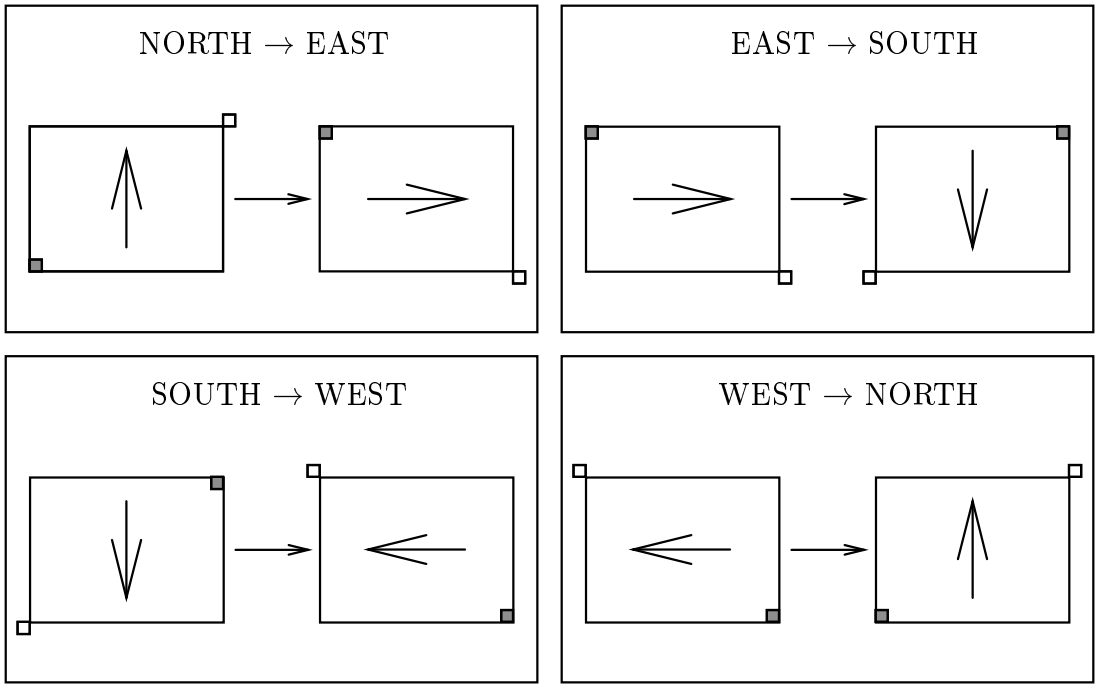


Figure 16: Port rotation examples

6.2.2 SplitInt

The SplitInt operator splits the original port in as many sub-ports as integer numbers are in the range and selects the port whose number matches with the integer number of the third argument. Figure 17 shows the four different variations of the operator in function of the four possible orientations [19].

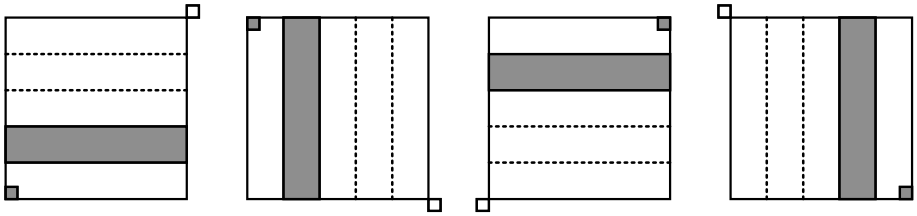


Figure 17: SplitInt examples. In this figure, the range represents the values $\{v_1, v_2, v_3, v_4, v_5\}$ and the integer is v_2.

When the range is null or the integer number is not in the range, there is not matching and then the operator returns a null simple port or an empty port list.

6.2.3 SplitReal

The SplitReal operator splits the port proportionally to the real argument. For example the real number 0.5 returns a port whose size is half of the original, the

[19] For example, the arguments in this case could be the range (3..7) and the integer 4.

real number 0.75 returns a port whose size is three quarters of the original and any real argument greater than 1.0 returns the whole port. Figure 18 shows the four different variations of the operator, in function of the four possible orientations.

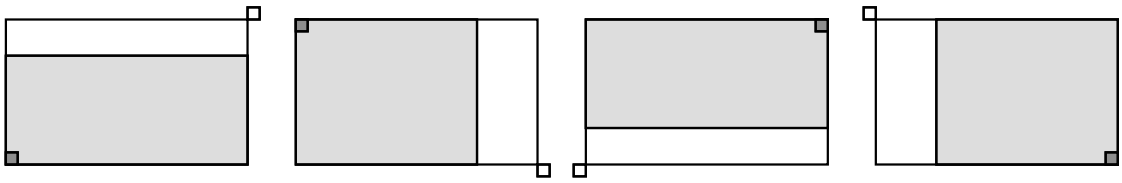


Figure 18: SplitReal examples. In this figure, the real value is 0.75

6.2.4 Constructive operators

The **union** $(+)$, **intersection** $(*)$ and **difference** $(-)$ of two simple ports are shown in Figure 19.

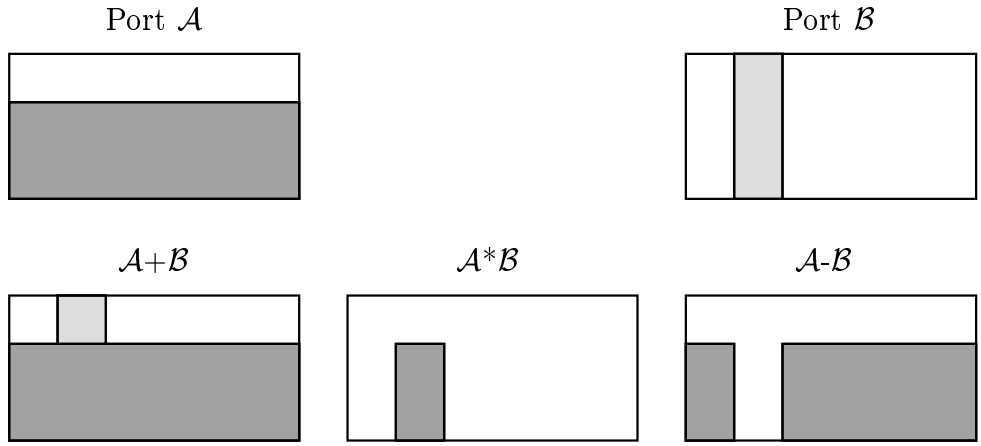


Figure 19: Constructive Ports Operators

The union of two ports concatenates the *port lists* or creates a new one; then this operation returns a *port list*.

Some problems were found with the difference of two ports. Think for example in the difference of two *simple ports* when the second is included in the first. In this case, the result can not be modelled by another simple port, and it does not exist a intuitive or direct solution using *port lists*. The language returns a *port list* composed by four sub-ports as is shown in Figure 20. This method adopted during the implementation is only one of some possibilities. During the analysis, no case was found to be better than the one used.

Property

The colour planes associated with the result port are the colour planes of the first argument. Then, the following equation is true in the system:

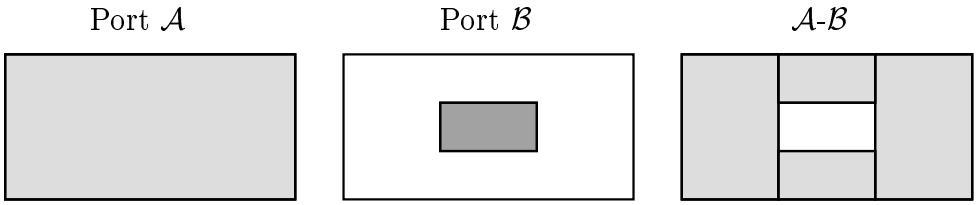


Figure 20: Difference example

$$\text{display}(\mathcal{A}) \equiv \text{display}(\mathcal{A}-\mathcal{B}) \cup \text{display}(\mathcal{A}\cap\mathcal{B})$$

However, we have to remark that the equation is false if we omit the display operator. This point is shown in Figure 21.

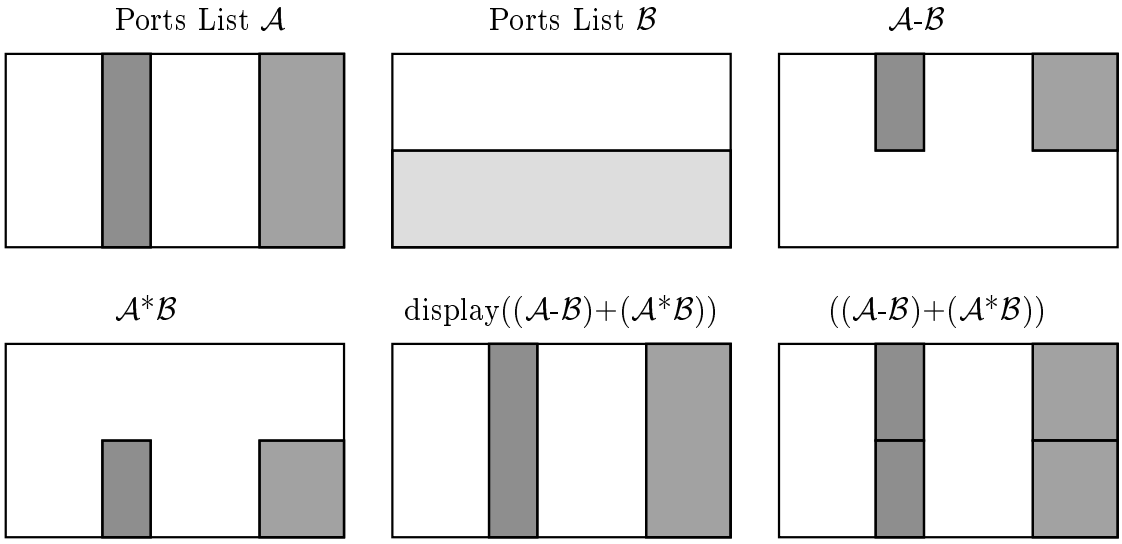


Figure 21: $\mathcal{A}\simeq(\mathcal{A}-\mathcal{B}) \cup (\mathcal{A}\cap\mathcal{B})$ example

Observation

Unfortunately, the ***TyWin System*** does not hold more basic properties. For example, the commutative property is false for the product (intersection) in the system, since $(\mathcal{A}*\mathcal{B})$ takes the colour planes from $\mathcal{A}$ and $(\mathcal{B}*\mathcal{A})$ from $\mathcal{B}$.

$$(\mathcal{A}*\mathcal{B}) \not\equiv (\mathcal{B}*\mathcal{A})$$

In spite of the fact that this last equation is false in the system, the alternatives of combining the colours of both arguments in the different operators were refused, since these methods make the result of applying some operations to different ports unpredictable.

For example, if the average of colours were used, the result of applying some operations to ports with different colours, would result in grey coloured ports.

In conclusion, the tests showed that the implemented method is better than other considered alternatives even when it lacks some basic properties.

7 The Values

The ***TyWin System*** assigns a method to display objects of the types introduced in Section 4.1 on page 20. These objects are canonical values of the *Typed Functional Language*.

7.1 Simple values

Although in the complete ***TyWin System*** we handle only canonical values, in the *PL Translator* module the simpler types have some operators defined (addition, product, etc.) in order to make the language interpreter more friendly for test performing.

T	Constructor:	○	tt	
boolean	Constructors:	○	True	
		○	False	
real	Constructors:	○	*real_constant*	
		○	Aureate	
	Operators:	○	($real + real$)	$\rightarrow real$
		○	($real * real$)	$\rightarrow real$
		○	($real - real$)	$\rightarrow real$
		○	($real \,/\, real$)	$\rightarrow real$
		○	($-\, real$)	$\rightarrow real$
integer	Constructors:	○	*integer_constant*	
		○	Width	
		○	Height	
	Operators:	○	($integer + integer$)	$\rightarrow integer$
		○	($integer * integer$)	$\rightarrow integer$
		○	($integer - integer$)	$\rightarrow integer$
		○	($integer \,/\, integer$)	$\rightarrow integer$
		○	($-\, integer$)	$\rightarrow integer$

The "aureate" section[20] is very often present in Torres García's paintings.

7.2 Collections

Two different classes of collections are introduced in ***TyWin***, lists and sets.

[20] The value of this real is $\frac{1}{\sqrt{2}}$.

list	Constructors:	○	[ ]
		○	[*collection_values*]
set	Constructors:	○	{ }
		○	{ *collection_values* }
collection_values	Syntax:	○	*value*
		○	*collection_values* : *value*

7.3 Composed values

Finally, we have the values of the types record, cartesian product and disjoint union.

record	Constructor:	○	< *record_values* >
record_values	Syntax:	○	*value*
		○	*record_values* :: *value*
pair	Constructor:	○	(*value* , *value*)
union	Constructors:	○	InL (*value*)
		○	InR (*value*)

Observation

The values presented are the objects to be displayed in function of the view selected by the system.

8 The Windows

A window in the ***TyWin System*** is defined as an extension of a type with some display information.

The definitions of windows or extended types in the system are:

type	Constructors:	○ T	
		○ Boolean	
		○ Real	
		○ Integer *range*	
		○ List of (*range* , *integer* , *type*)	
		○ Set of *type*	
		○ Record (*record_types*)	
		○ (*type* + *type*)	
		○ (*type* ∗ *type*)	
	Operator:	○ ToView (*type* , *port*)	→*view*
record_types	Constructors:	○ *type*	
		○ *record_types* :: *type*	

The *ToView* operator, translates a type structure to a view. This operation associates the port with the information necessary to build the corresponding view.

9 The Views

A view represents a pair $\langle window,\ port \rangle$. It contains the information needed to represent graphically every object, since it has the method in the window and the place in the port. A view constructor is defined in the language for each type.

view	Constructors:	○	VT (*port*)	
		○	VBoolean (*port*)	
		○	VReal (*port*)	
		○	VInteger (*range* , *port*)	
		○	VList (*range* , *integer* , *view*)	
		○	VSet (*view*)	
		○	VRecord (*record_views*)	
		○	VUnion (*view* , *view*)	
		○	VProd (*view* , *view*)	
	Operator:	○	ToPort(*view* , *value*)	→*portlist*
record_views	Constructors:	○	*view*	
		○	*record_views* :: *view*	

Observations

1. The *RotatePlane* operation is applied to the *ports* in the *record_views* argument when they are saved in a *Record View*. The idea is that the first sub-view of the record is not modified, the *VRotatePlane* operation is applied once to the second sub-view of the record, twice to the third sub-view of the record and so on.

   In this way, the first colour in the port is activated in the first sub-view, the second colour in the port is activated in the second sub-view, etc. Then, the first object in the record will be displayed in the first colour, the second object in the record will be displayed in the second colour, and so on.

2. The *Rotate* operation is applied to the *port* in the second *view* argument when it is saved in a *Product* view. The idea is that the second object is considered rotated when the intersection is evaluated.

The core of the language is the concept of *port*. In order to produce the graphic representation of any object, the *port* in the view is transformed into another port. This transformation is developed by the *ToPort* operator.

Observation

A particular conditional can be introduced easily in the language:

$$cond : Boolean \times Port \longrightarrow Port$$

$$cond(True, ToPort(ToView(\mathcal{T}, \mathcal{P}), \mathcal{V}_{\mathcal{T}})) \doteq ToPort(ToView(\mathcal{T}, \mathcal{P}), \mathcal{V}_{\mathcal{T}}))$$
$$cond(False, ToPort(ToView(\mathcal{T}, \mathcal{P}), \mathcal{V}_{\mathcal{T}})) \doteq Nil$$

The result of this conditional is the port defined by the value $\mathcal{V}_{\mathcal{T}}$ and the view made by the type $\mathcal{T}$ and the port $\mathcal{P}$, when the Boolean value is True, and the null port when the Boolean value is False.

The semantic associated with the display of the Boolean and Product types allows the user to use the conditional in the language. The following equation explain it:

$$cond(\mathcal{V}_{Boolean}, ToPort(ToView(\mathcal{T}, \mathcal{P}), \mathcal{V}_{\mathcal{T}})) \doteq ToPort(ToView((\mathcal{T} * Boolean), \mathcal{P}), (\mathcal{V}_{\mathcal{T}}, \mathcal{V}_{Boolean}))$$

Thus the display of a product where the second subtype is Boolean works like the conditional introduced above.

10 The ToPort Operator

The ToPort operator translates a value into graphic information (a port). It works in function of the category of views considered.

10.1 Simple values

10.1.1 ToPort(VT(port), tt)

Since the type T has only one value, the result is the same port received as argument.

Example

- $ToPort(VT(p_{view}), tt)$ - Figure 22.A

10.1.2 ToPort(VBoolean(port), bool_value)

The result is an empty port list if the *bool_value* is $False$ and the original *port* when the *bool_value* is $True$.

Examples

- $ToPort(VBoolean(p_{view}), True)$ - Figure 22.A
- $ToPort(VBoolean(p_{view}), False)$ - Figure 22.B

10.1.3 ToPort (VReal (port), real_value)

The operator applies the *SplitReal* operation to the port with the argument *real_value*.

Example

- $ToPort(VReal(p_{view}), 0.75)$ - Figure 22.C

10.1.4 ToPort (VInteger(range , port), int_value)

The operator applies the *SplitInt* operation to the port with the arguments *range* and *int_value*.

Examples

- $ToPort(VInteger((15..15), p_{view}), 15)$ - Figure 22.A
- $ToPort(VInteger((1..0), p_{view}), 3)$ - Figure 22.B
- $ToPort(VInteger((7..9), p_{view}), 12)$ - in Figure 22.B
- $ToPort(VInteger((3..6), p_{view}), 4)$ - in Figure 22.D

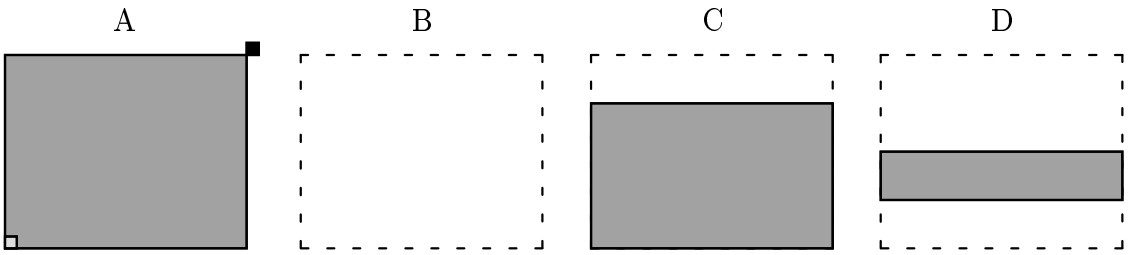


Figure 22: ToPort examples. Simple values cases. The original port p_{view} is defined in A.

The rules for simple values are introduced in Section C.1 on page 67.

10.2 Collections

10.2.1 ToPort (VList (range , integer, sub-view), list_value)

The display method associated with the lists is not so simple. The operation works in function of the range in the view.

- When the range is empty, the port in the *sub-view* is split in as many sub-ports as elements are in the *list_value* (*length*(*list_value*)). Then, each split sub-port is rotated the number of times defined by the *integer*. Finally, the first element of the list is displayed in the *sub-view* with the first sub-port split and rotated associated; the second element of the list is displayed in the *sub-view* with the second sub-port split and rotated associated; and so on.

- When the range is not empty, the port in the *sub-view* is split in as many sub-ports as elements are in the *range*. Then, each split sub-port is rotated the number of times defined by the *integer*. Finally, the element of the list whose ordinal corresponds with the first integer of the range is displayed in the *sub-view* with the first sub-port split and rotated associated; then the next element of the list is displayed in the *sub-view* with the next sub-port split and rotated associated; and so on.

Examples

- $ToPort(VList((1..0), 1, VInteger((1..5), p_{view}), [2 : 4 : 5 : 3])$ - Figure 23.B
- $ToPort(VList((2..5), 1, VInteger((1..5), p_{view}),$
  $[4 : 2 : 4 : 5 : 3 : 7])$ - Figure 23.B
- $ToPort(VList((3..6), 3, VReal(p_{view}),$
  $[0.1 : 0.05 : 0.6 : 0.4 : 0.0 : 0.8 : 0.9 : 0.7])$ - Figure 23.C
- $ToPort(VList((1..0), 1, VReal(p_{view}), [0.8 : 1.4 : 0.2 : 0.4])$ - Figure 23.D
- $ToPort(VList((1..4), 0, VInteger((5..8), p_{view}), [5 : 3 : 6 : 7])$ - Figure 23.E
- $ToPort(VList((3..7), 1, VInteger((5..8), p_{view}), [1 : 8])$ - Figure 23.H
- $ToPort(VList((0..1), 1, VInteger((5..8), p_{view}), [])$ - Figure 23.H

10.2.2 ToPort (VSet (sub-view), set_value)

The operator works on each element of *set_value*, using *sub-view*. By this way as many ports as elements have *set_value* are obtained. The final result is the union of all these port lists. When the *set_value* is empty, the result is an empty port list.

Examples

- $ToPort(VSet(VInteger((1..8), p_{view})), \{1 : 3 : 6 : 8\})$ - Figure 23.F
- $ToPort(VSet(VInteger((3..6), p_{view})), \{2 : 4 : 6 : 10\})$ - Figure 23.G
- $ToPort(VSet(VReal(p_{view})), \{\})$ - Figure 23.H

The rules for collections are introduced in Section C.2 on page 67.

10.3 Composed values

10.3.1 ToPort (VRecord (sub-view$_1$::...::sub-view$_n$), <val$_1$::...::val$_n$ >)

In this case, *ToPort* is invoked with $sub\text{-}view_1$ and val_1, then with $sub\text{-}view_2$ and val_2, and so on. Finally, the operation returns the union of all the previous results.

Remember that the *VRecord* operator modifies the colour planes in the sub-views. In this way, each element in the record can be displayed in a different colour.

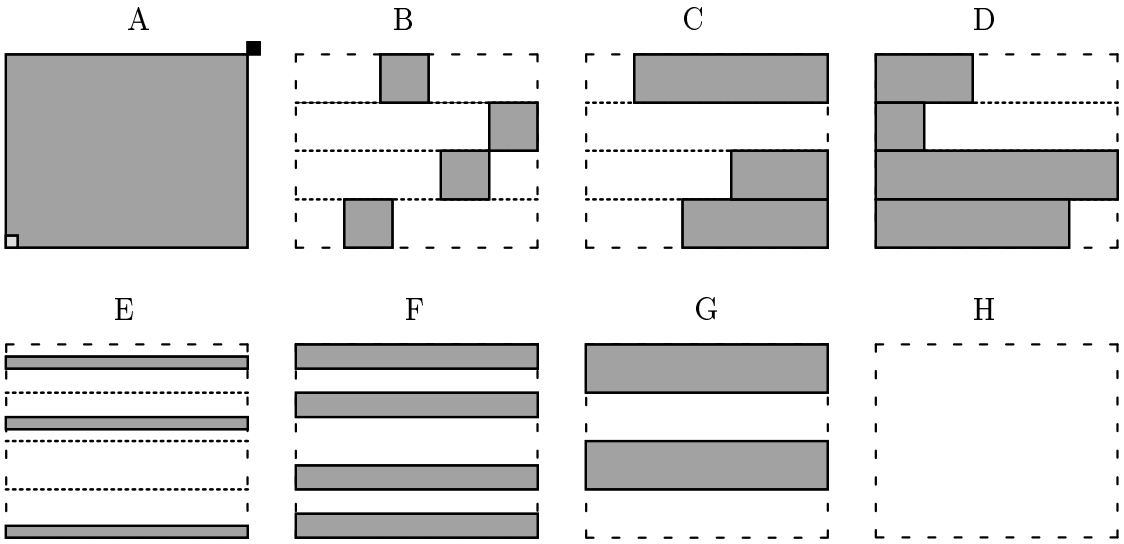


Figure 23: ToPort examples. Collection values cases. The original port p_{view} is defined in A.

Examples

- $ToPort(VRecord(VInteger((1..3), p_{view}) ::$
  $VInteger((1..3), p_{view}) ::$
  $VInteger((1..3), p_{view})),$
  $< 1 :: 2 :: 3 >)$ - Figure 24.B

- $ToPort(VRecord(VReal(p_{view}) ::$
  $VInteger((1..4), p_{view})),$
  $< 0.75 :: 2 >)$ - Figure 24.C

10.3.2 ToPort (VUnion (sub-view$_1$, sub-view$_2$), union_value)

When the *union_value* corresponds to the left subtype of the union, the result is the application of ToPort to *sub-view*$_1$ and this value. When *union_value* corresponds to the right subtype of the union, the result is the application of ToPort to *sub-view*$_2$ and this value.

Examples

- $ToPort(VUnion(VReal(p_{view}),$
  $VInteger((1..4), p_{view})),$
  $InL(0.75))$ - Figure 24.D

- $ToPort(VUnion(VReal(p_{view}),$
  $VInteger((1..4), p_{view})),$
  $InR(2))$ - Figure 24.E

10.3.3 ToPort (VProd (sub-view$_1$, sub-view$_2$), pair_value)

First, ToPort is recursively invoked with the left element of *pair_value* and *sub-view*$_1$, then with the right element of *pair_value* and *sub-view*$_2$, and finally the result is the intersection of both ports.

Remember that the $VProd$ operator rotates the port associated with the second sub-view.

Examples

- $ToPort(VProd(VInteger((1..4), p_{view}),$
  $VReal(p_{view})),$
  $(2, 0.8))$ - Figure 24.F
- $ToPort(VSet(VProd(VInteger((1..2), p_{view}),$
  $VInteger((1..5), p_{view}))),$
  $\{(1,2) : (1,5) : (2,1) : (2,4)\})$ - Figure 24.G
- $ToPort(VRecord(VSet(VProd(VInteger((1..2), p_{view}),$
  $VInteger((1..5), p_{view}))) ::$
  $VSet(VProd(VInteger((1..2), p_{view}),$
  $VInteger((1..5), p_{view}))) ::$
  $VSet(VProd(VInteger((1..2), p_{view}),$
  $VInteger((1..5), p_{view}))),$
  $<\{(1,5) : (2,1)\} :: \{(1,3) : (2,3)\} :: \{(1,2) : (2,4)\} >)$ - Figure 24.H

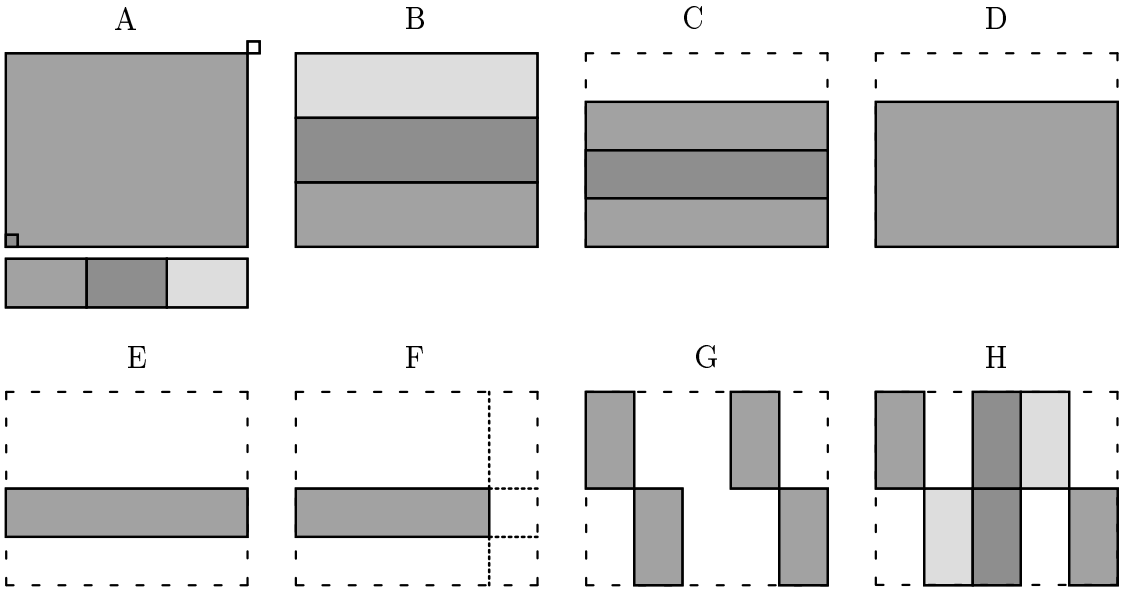


Figure 24: ToPort examples. Composed values cases. The original port p_{view} is defined in A with the three colour planes showed.

The rules for records, disjoint unions and cartesian products, are introduced in Section C.3 on page 68.

11 The Sentences

The statements included in the system are:

assignment	◦	*identifier* = *expression* ;
file_inclusion	◦	#include "*filename*";
display	◦	display (*port*) ;
free	◦	free (*identifier*) ;

Then the language allows the user:

- To use identifiers to store *simple ports*, *port lists*, *values*, *views*, etc.
- To include complex values (like ideograms) and types stored in "*libraries*" created by the user (for example using the *Ideogram Editor*).
- To display *ports*.
- To remove a local identifier from the environment when it becomes useless. Since the scope of an identifier in the language is all the session after its definition, it is useful to free the memory assigned to an identifier when it will not be used again.

In order to allow the programmer to document his programs, two types of comments are included in the language:

1. ... *sentence* ... || ... *comment* ... <EOL> : allows to write a comment at the end of the sentence.
2. /* ... *comment* ... */ : allows to write C style comments.

No other sentences are implemented because more complex structures (like functions) will be available in the *Functional Programming Language* module of the ***TyWin System***, where the input of this *PL Translator* module is generated. In conclusion, this module only translates canonical objects and types into ports.

12 The Ideograms Module

12.1 Introduction

The ideograms are implemented in the system as pairs $\langle ideogram\_value, ideogram\_type \rangle$. The following steps should be followed to display an ideogram:

- Include the library-file where the ideogram is defined.
- Define the view where the ideogram will be displayed using the *ideogram_type* and the selected port.
- Define the port associated with the display of the ideogram using the *ideogram_value* and the previouslly defined view.
- Display this port.

The model to design an ideogram is a common matrix of "pixels". This model is easily implemented in ***TyWin*** like a set of "*TWpixels*", where a *TWpixels* is a cartesian product of integers. Where the ideogram has more than one colour plane, the ideogram is a record where each field is defined like the previous set.

Then, an ideogram results in something very similar to a classic rectangular image. The differences between the behaviour of both models are:

- An ideogram has many planes of colour, but the colour associated with each plane is defined in the port where it is stamped and is not in the ideogram.
- Even when the ideograms have a resolution associated (the number of rows and columns where it is defined), the display of an ideogram uses automatically the size of the port where it is stamped.

12.2 The Ideogram Editor

A simple graphic editor was implemented to define five-planes ideograms. The editor asks for the ideogram name and resolution and shows a grid to active the *TWpixels* with each of the five planes.

However, it is better to use the usual icon editors and translate the output to ***TyWin*** ideograms. Then the user would not be required to learn a new editor. Further, the user can translate pre-defined icons and images to ***TyWin***.

12.3 The Ideogram Translator

The translator produces the same output as the editor, but the input can be a standard *X Bitmap* or a *Colour X Pixmap*.

12.4 The Ideogram Compression

Unfortunately, some problems appeared when dealing with large ideograms (more than 10.000 *TWpixels*):

1. the library-file size was huge.
2. the process for displaying an ideogram used too many resources (time and memory).

In order to solve these problems a compression method was designed and implemented. This method is similar to the **Constant Area Coding** compression [GW93]. The compression process defines variable-size windows on the image in order to detect rectangular areas with the same plane associated.

The number of rows and columns of this windows have to be divisors of the number of rows and columns of the complete matrix respectively. When one of this rectangular areas are found a sub-value is defined.

Then, the method covers each plane in the image, with variable-size rectangles.

Example

In order to better understand the compression method an example is introduced. Figure 25 shows a 16×16 matrix associated with a two colour ideogram that we called *example*.

This ideogram has 104 "pixels" (82 in the first plane and 22 in the second). Then the simple method without compression would define two sets of TWpixels, one with 82 elements and the other with 22.

However, the largest square in the upper left corner (8×8) could be modelled by the pair $\langle$**(0, 0), (Integer(0..1) * Integer(0..1))**$\rangle$. Then, only one value defines this area, not 64 like in the simple method. You can see that the type (Integer(0..1) * Integer(0..1)) is associated with the type of the first plane and the value (0, 0) associated with the value of the first plane in the code of this example.

This idea is used with windows [(16×16) : (16×8) : (8×16) : (8×8) : $\cdots$: (2×2) : (2×1) : (1×2) : (1×1)] considering the different combinations of divisors of 16.

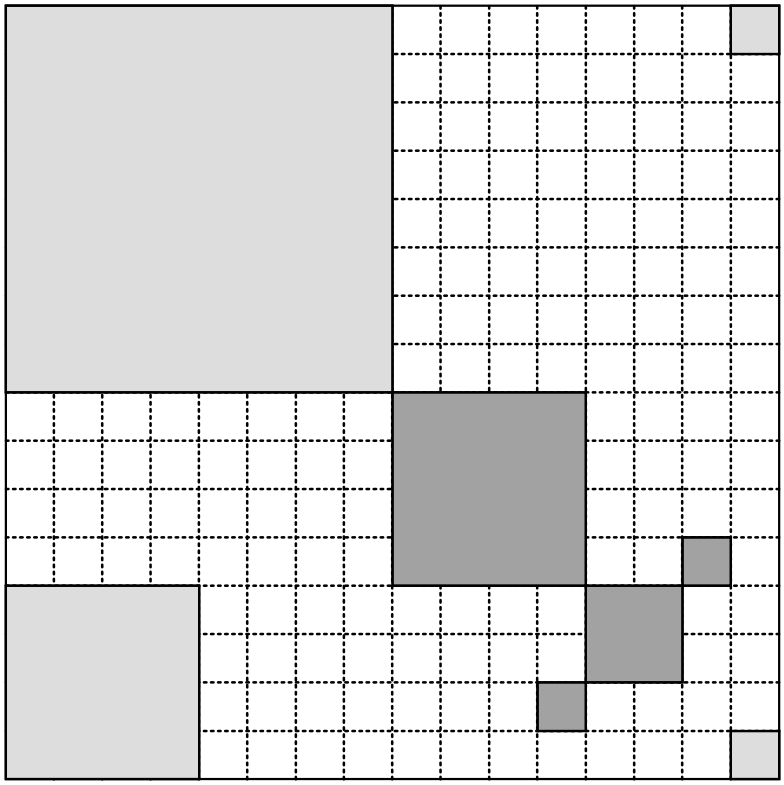


Figure 25: The ideogram matrix

Finally, the following ideogram file is obtained:

```
plane_type_1 = Set of
                      (Set of (Integer(0..15) * Integer(0..15))
                    +        ((Integer(0..3)  * Integer(0..3))
                    +         (Integer(0..1)  * Integer(0..1))));

plane_1 = {InL({(15,15):(15,0)}):InR(InL((0,3))):InR(InR((0,0)))};

plane_type_2 = Set of
                      (Set of (Integer(0..15) * Integer(0..15))
                    +        ((Integer(0..7)  * Integer(0..7))
                    +         (Integer(0..3)  * Integer(0..3))));

plane_2 = {InL({(14,11):(11,14)}):InR(InL((6,6))):InR(InR((2,2)))};

example = < plane_1 :: plane_2 >;

example_type = Record ( plane_type_1 :: plane_type_2);

free (plane_1);
free (plane_2);
free (plane_type_1);
free (plane_type_2);
```

13 Examples

13.1 Hello World

The first example introduced is the classic *Hello World* where it is taken advantage of the fact that the language works translating ports into ports.

Figure 26: Hello World.

Figure 26 shows the result of the following simple program.

```
#include "$TW_LIBRARY/char.5.12.all";

lport = (port((<0, 0> | <WIDTH, HEIGHT>), [<MIN, MIN, MID>]) # Nil);

hello_val = [ _H: _E: _L: _L: _O ];
world_val = [ sp: _W: _O: _R: _L: _D ];
text_type = List of ((1..0), 1, char_type);

text_view = ToView(text_type, lport);
lport = ToPort(text_view, hello_val);

text_view = ToView(text_type, lport);
lport = ToPort(text_view, world_val);

display (lport);
```

First, the font library file *char.5.12.all* is loaded (each character is defined in a matrix of 5×12). Then, a port that covers all the ***TyWin*** output window, the values of each string (implemented as a list of characters) and the type associated with these strings are defined. After it, the views and ports for each string are defined, using the port obtained by the first string in the view of the second. Finally the result port is displayed.

13.2 Text Fonts

13.2.1 Text Orientation I

This example shows how the orientation of the ports does not affect the way that the operations are applied.

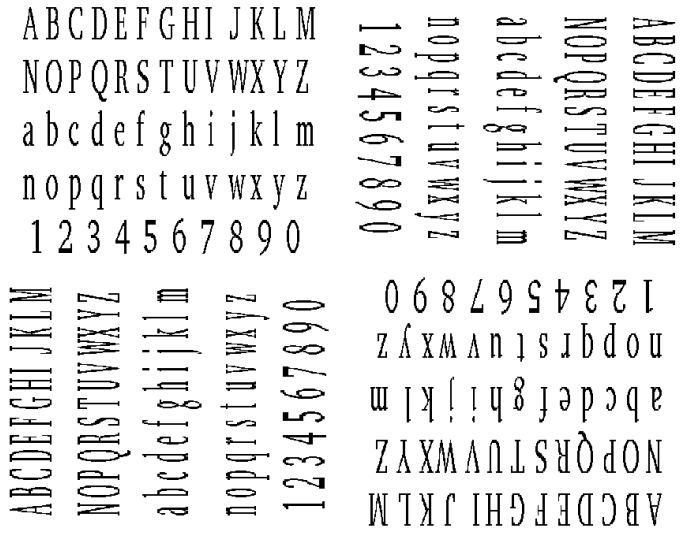


Figure 27: Text Fonts.

Figure 27 shows the result of the following program.

```
#include "$TW_LIBRARY/char.20.24.all";

lport = (port((<0, 0> | <WIDTH, HEIGHT>), [ <MDX, MDX, MDN> ]) # Nil);

sub_view = ToView((Integer(0..1) * Integer(0..1)), lport);

p1 = ToPort(sub_view, (0, 0));          p2 = ToPort(sub_view, (1, 0));
p3 = ToPort(sub_view, (1, 1));          p4 = ToPort(sub_view, (0, 1));

p1 = rotate(p1);          p2 = birotate(p2);        p3 = unrotate(p3);

wport = (p1+(p2+(p3+p4)));

text_val =
      [
       [ sp: _A: _B: _C: _D: _E: _F: _G: _H: _I: _J: _K: _L: _M: sp ]:
       [ sp: _N: _O: _P: _Q: _R: _S: _T: _U: _V: _W: _X: _Y: _Z: sp ]:
       [ sp: _a: _b: _c: _d: _e: _f: _g: _h: _i: _j: _k: _l: _m: sp ]:
       [ sp: _n: _o: _p: _q: _r: _s: _t: _u: _v: _w: _x: _y: _z: sp ]:
       [ sp: _1: _2: _3: _4: _5: _6: _7: _8: _9: _0: sp ]
      ];
```

```
view = ToView(List of ((1..0),3,List of ((1..0),0,char_type)), wport);
lport = ToPort(view, text_val);
display (lport);
```

First, the font library file *char.20.24.all* is loaded (each character is defined in a matrix of 20 × 24). Then, a port that covers all the ***TyWin*** output window is defined and split in four identical sub-ports using a cartesian product of integers. Each sub-port is re-oriented in order to obtain the four possible orientations and the four sub-ports are associated with *wport*. Then, a list of strings (a string is implemented as a list of characters) is defined. After that, the view and port for the list of string are assigned to *view* and *lport* respectively. Finally the result port is displayed.

13.2.2 Text Orientation II

This example shows how the orientation of the strings characters can be managed.

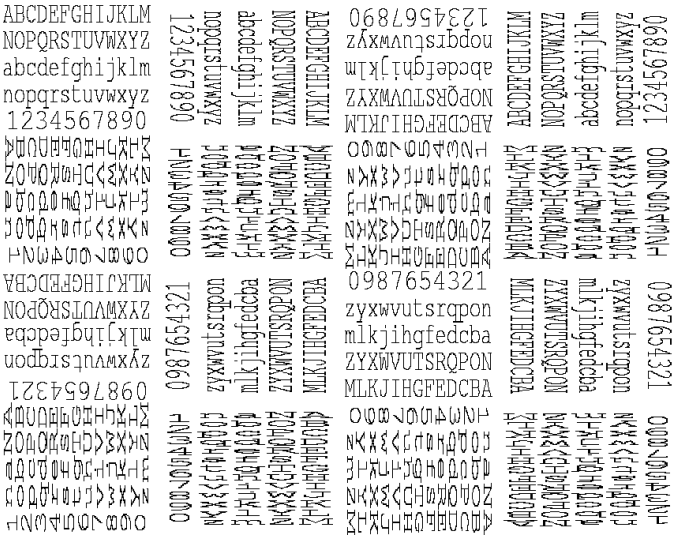


Figure 28: Text Fonts.

Figure 28 shows the result of the following program.

```
#include "$TW_LIBRARY/char.20.24.all";

lport = (port((<0, 0> | <WIDTH, HEIGHT>), [ <MDX, MDX, MDN> ]) # Nil);

sub_view = ToView((Integer(1..4) * Integer(1..4)), lport);

p01 = ToPort(sub_view, (1, 1));        p02 = ToPort(sub_view, (1, 2));
p03 = ToPort(sub_view, (1, 3));        p04 = ToPort(sub_view, (1, 4));
p05 = ToPort(sub_view, (2, 1));        p06 = ToPort(sub_view, (2, 2));
p07 = ToPort(sub_view, (2, 3));        p08 = ToPort(sub_view, (2, 4));
p09 = ToPort(sub_view, (3, 1));        p10 = ToPort(sub_view, (3, 2));
```

```
p11 = ToPort(sub_view, (3, 3));        p12 = ToPort(sub_view, (3, 4));
p13 = ToPort(sub_view, (4, 1));        p14 = ToPort(sub_view, (4, 2));
p15 = ToPort(sub_view, (4, 3));        p16 = ToPort(sub_view, (4, 4));

p01 =   rotate(p01);     p02 =   rotate(p02);     p03 =   rotate(p03);
p04 =   rotate(p04);     p05 = birotate(p05);     p06 = birotate(p06);
p07 = birotate(p07);     p08 = birotate(p08);     p09 = unrotate(p09);
p10 = unrotate(p10);     p11 = unrotate(p11);     p12 = unrotate(p12);

pI   = (((p01 + p05) + p09) + p13);
pII  = (((p02 + p06) + p10) + p14);
pIII = (((p03 + p07) + p11) + p15);
pIV  = (((p04 + p08) + p12) + p16);

text_val =
      [
       [ sp: _A: _B: _C: _D: _E: _F: _G: _H: _I: _J: _K: _L: _M: sp ]:
       [ sp: _N: _O: _P: _Q: _R: _S: _T: _U: _V: _W: _X: _Y: _Z: sp ]:
       [ sp: _a: _b: _c: _d: _e: _f: _g: _h: _i: _j: _k: _l: _m: sp ]:
       [ sp: _n: _o: _p: _q: _r: _s: _t: _u: _v: _w: _x: _y: _z: sp ]:
       [ sp: _1: _2: _3: _4: _5: _6: _7: _8: _9: _0: sp ]
      ];

view = ToView(List of ((1..0), 3, List of ((1..0), 0, char_type)), pI);
lportI = ToPort(view, text_val);

view = ToView(List of ((1..0), 3, List of ((1..0), 1, char_type)), pII);
lportII = ToPort(view, text_val);

view = ToView(List of ((1..0), 3, List of ((1..0), 2, char_type)), pIII);
lportIII = ToPort(view, text_val);

view = ToView(List of ((1..0), 3, List of ((1..0), 3, char_type)), pIV);
lportIV = ToPort(view, text_val);

display (lportI);
display (lportII);
display (lportIII);
display (lportIV);
```

First, the font library file *char.20.24.all* is loaded. Then, a port that covers all the ***TyWin*** output window is defined and split in sixteen identical sub-ports using a cartesian product of integers. Some sub-ports are re-oriented in order to obtain the four possible orientations in groups of four ports. The sixteen sub-ports are associated in four groups where each group has four ports with four different orientations. Then, a list of strings (a string is implemented as a list of characters) is defined. After that, the string is processed in each group of ports, using different rotations for each view. Finally the result ports are displayed.

13.3 Coloured Butterfly

This example shows how the colours are associated with the ports and not to the ***TyWin*** images (ideograms).

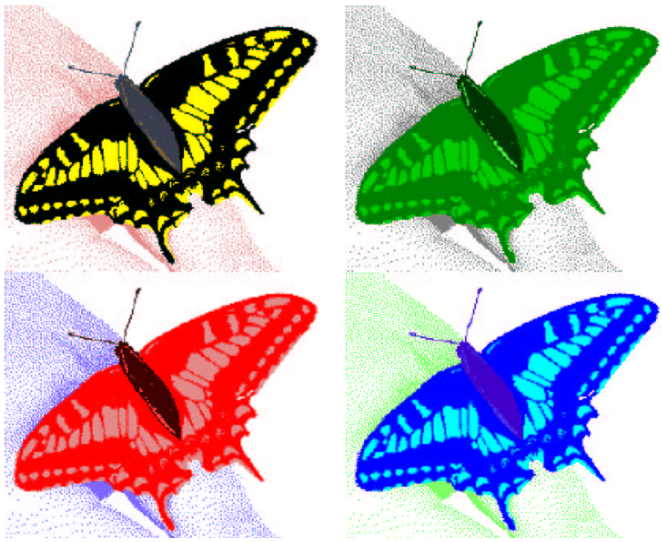


Figure 29: Butterfly.

Figure 29 shows the result of the following program.

```
pp = (port((<0, 0> | <WIDTH, HEIGHT>), [ <MDX, MDX, MDN> ]) # Nil);

sub_view = ToView((Integer(1..2) * Integer(1..2)), pp);

p1 = ToPort(sub_view, (1, 1));        p2 = ToPort(sub_view, (1, 2));
p3 = ToPort(sub_view, (2, 1));        p4 = ToPort(sub_view, (2, 2));

p1 = SetColour(p1, [ <MAX,MDX,MDX>:<MDX,MDX,MDN>:<MDN,MDN,MDN>:
<MIN,MIN,MIN>:<MAX,MAX,MIN> ]);
p2 = SetColour(p2, [ <MDX,MDX,MAX>:<MAX,MDN,MDN>:<MDN,MIN,MIN>:
<MAX,MIN,MIN>:<MAX,MDX,MDX> ]);
p3 = SetColour(p3, [ <MDX,MDX,MDX>:<MDN,MAX,MDN>:<MIN,MDN,MIN>:
<MIN,MDX,MIN>:<MIN,MID,MIN> ]);
p4 = SetColour(p4, [ <MDX,MAX,MDX>:<MDN,MDN,MAX>:<MDN,MIN,MID>:
<MIN,MIN,MAX>:<MIN,MAX,MAX> ]);

p_all = (((p1 + p2) + p3) + p4);

#include "$TW_LIBRARY/butterfly";
view = ToView(butterfly_type, p_all);
lport = ToPort(view, butterfly);
display (lport);
```

13.4 Torres García's Paintings

The following pictures show four examples of designs where Torres García's paintings were taken as models.

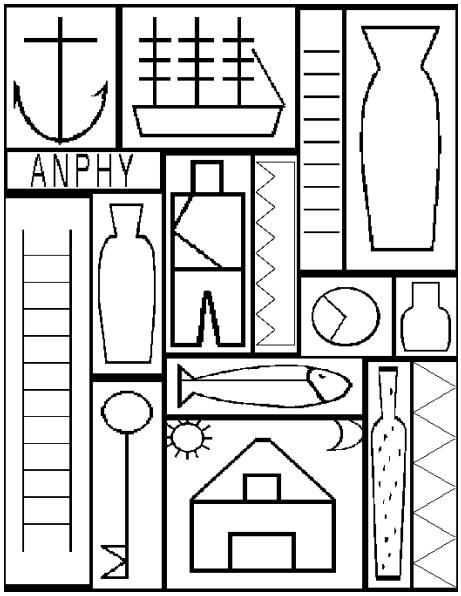


Figure 30: "Constructivo con ancla y barco - Anphy" (1932 - 12 × 9 cm.).

[21]

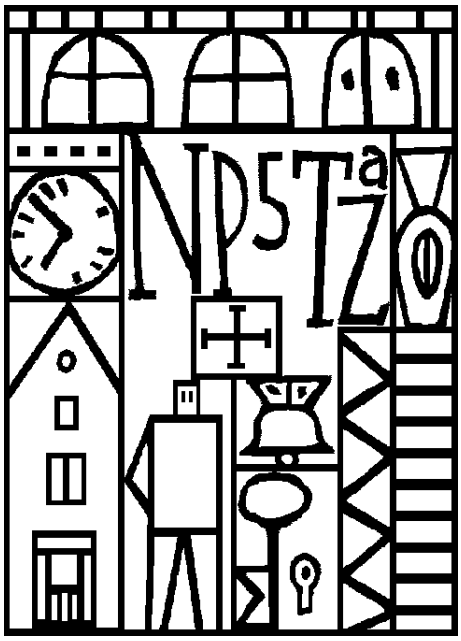


Figure 31: "Composición constructiva" (1932 - 73 × 60 cm.).

[22]

[21] Constructive with anchor and ship - Anphy
[22] Constructive composition

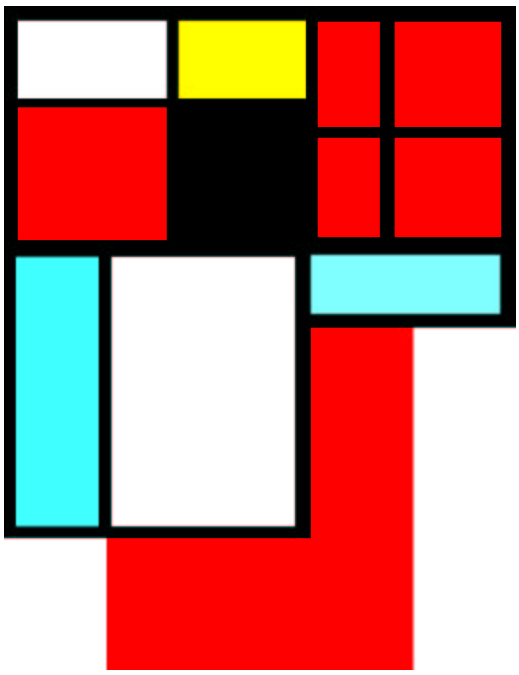


Figure 32: "Planos de color - Madera constructiva" (1929 - 28 × 21.5 cm.).

23

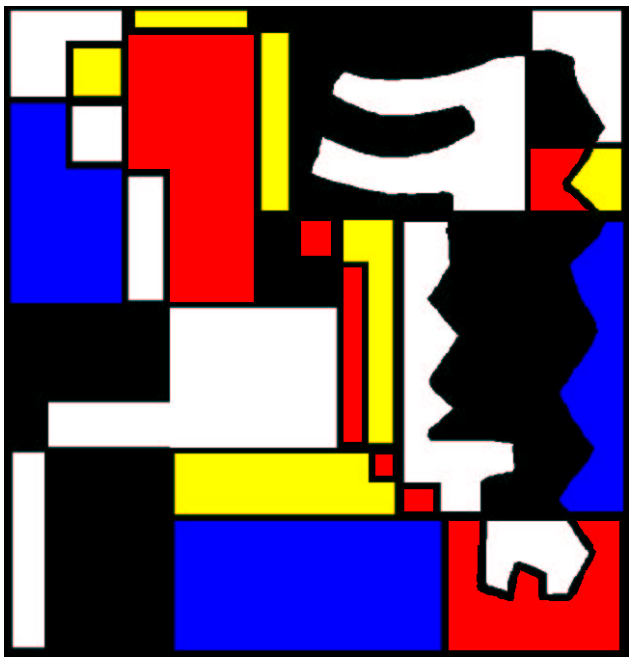


Figure 33: "Estructura a cinco tonos con dos formas intercaladas" (1948 - 52 × 50 cm.).

24

[23] Planes of colour - Constructive wood

[24] Structure at five tones with two shapes intercalated

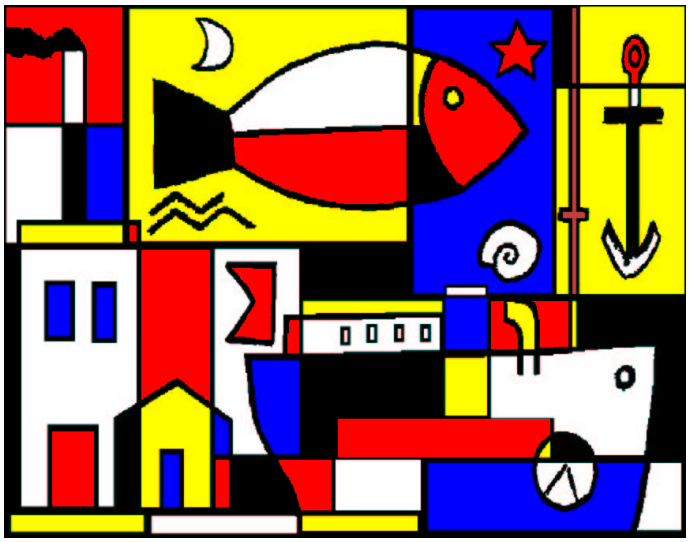


Figure 34: "Construcción" (1944 - 54 × 82 cm.).

[25]

13.5 Screen design

Figure 35 shows a different example, a possible screen on a graphic station.

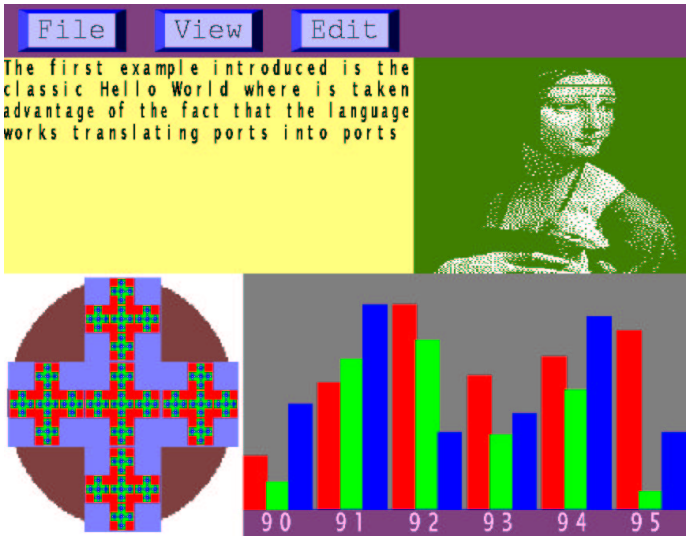


Figure 35: Screen Design.

[25] Construction

14 Conclusions and Future Work

One of the main questions of the ***TyWin*** research project was:

*Could a **TyWin** programmer easily define constructive style designs, defining structures and stamping ideograms stored in libraries?*

The tests performed with the *Language Interpreter* indicate that we can answer the previous question in a positive form:

After a learning period of the different display rules, it was possible to design and program in a natural and easy way 2D designs like the examples introduced in this paper.

We have to remark that the ***TyWin*** programmer does not need a deep knowledge in Computer Graphics. The programmer just needs to learn the graphic representation rules of the different types and to understand the simple model <**port-window-view**> defined in the system.

From this point of view, the system could be useful for beginners in Computer Graphics since they would mainly use the usual programming language concepts.

The task of programming in the ***TyWin System*** the examples gave us more information about the behavior of the system:

- The examples revealed that the ***TyWin*** programmer can easily design fractals like in the *Hello World* or *Screen Design* examples.
- When the graphic editors of the system are available, the task of programming the examples presented in this paper will take few minutes [26].
- In ***TyWin***, 2D graphic designs are independent of the size of the port where the designs will be mapped. Then, an ideogram stored in a library does not include any restriction of the port size where it can be displayed.
- Text in ***TyWin*** was implemented in a easy way by means of ideograms representing fixed pitch (or character-cell) fonts.
- Even when this implementation of the system can use only a reduced set of colours (125 colours), this problem could be overcome in future extensions using different primitives to manage the colours in the X-Windows System.

However, we can not evaluate completely the two finished modules until all the modules of the ***TyWin System*** are developed. We hope that the *Typed Functional*

[26] If all the ideograms of the design are defined previously.

Language module will enhance the potential of the system [27].

In balance, the implementation of the *Language Interpreter* module and the *Ideograms* sub-system allows the user to program designs using a really different conception that needs some practice to work loosely.

Unfortunately at this stage of the development the prototype does not provide the user with primitives to manage some simple shapes. However, we think that the *Typed Functional Language* module will provide the primitives to manage diagonal lines, triangles, arcs and other primitive shapes and also will provide the primitives to deal with variable-pitch (or proportional) fonts implementing libraries of functions to deal with some shapes and fonts.

Then, for a definitive evaluation of these implemented modules, it should be necessary to implement all the ***TyWin System*** modules and evaluate how each module interacts with the others in the complete system.

[27] Juan José Cabezas has been implementing the *Typed Functional Language* (**"bamba"**) for the system since 1994.

Acknowledgements

I specially thank Juan José Cabezas for suggesting this project and for helping me along the development of this work.

I want to thank Gustavo Betarte and Vera Goldim for reading drafts of this document and for making suggestions.

I would also want to thank the members of the Bid-Conicyt project number 043 "Estructuras para Sistemas Lógicos (ESL)"[28] who allowed me to use their workstations and laser printer.

[28] Logical Framework

References

[Cab91] Juan José Cabezas. *An approach to typed windows.*
Proceedings of the Compugraphics91 Conference, Portugal, 1991.

[Gar69] Joaquín Torres García. *Lo aparente y lo concreto en el arte (1947).*
Capítulo Oriental 41 - Centro Editor de América Latina, 1969.

[GW93] Rafael C. Gonzáles and Richard E. Woods. *Digital Image Processing.*
Addison Wesley, 1993.

[Lin92] Gabriel Paluffo Linari. *Historia de la Pintura Uruguaya. Torres García : de Barcelona a París.*
Ediciones de la Banda Oriental, 1992.

[NS83] William M. Newman and Robert F. Sprouil. *Principles of Interactive Computer Graphics.*
Mc Graw Hill, 1983.

[NPS90] Bengt Nordström, Kent Petersson and Jan M. Smith. *Programming in Martin-Löf's Type Theory. An Introduction.*
Oxford University Press, 1990.

APPENDIX

A The Implementation

All the modules were programmed in *C*. *Lex* and *Yacc* were used to parse the ***TyWin Language*** and the X-images (X Bitmaps or colour X Pixmaps) in the ideogram's translator. The *Xlib* library was used to manage the X-Windows graphic output and the *XView* library was used in the *Ideogram Editor*.

The ***TyWin*** interpreter takes more than 9000 code lines and the ideogram translators more than 2000.

A.1 The Interpreter Architecture

The interpreter main modules are shown in Figure 36.

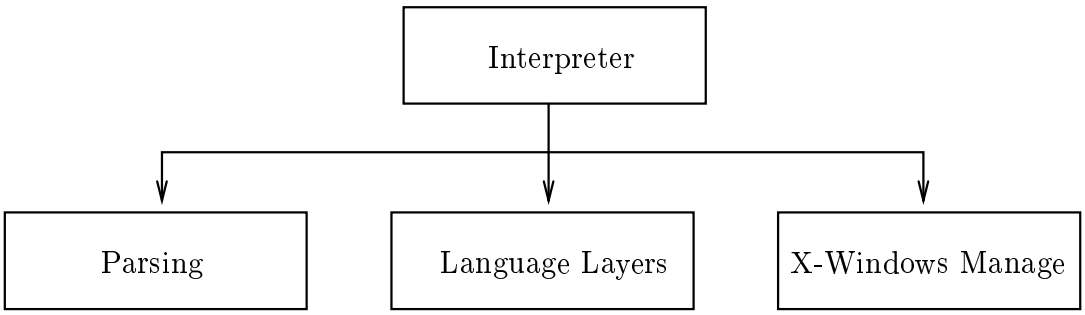


Figure 36: Interpreter Modules - Architecture.

Then, the interpreter is composed by three sub-systems.

A.1.1 Parsing

This module is composed by three sub-modules as is shown in Figure 37. Then, the parsing module has:

- a Lex component to make the lexicographic analysis.
- a Yacc component to make the syntax analysis.
- a Layers component to define the syntax functions invoked in the Yacc module. This module has one sub-module for each component of the language.

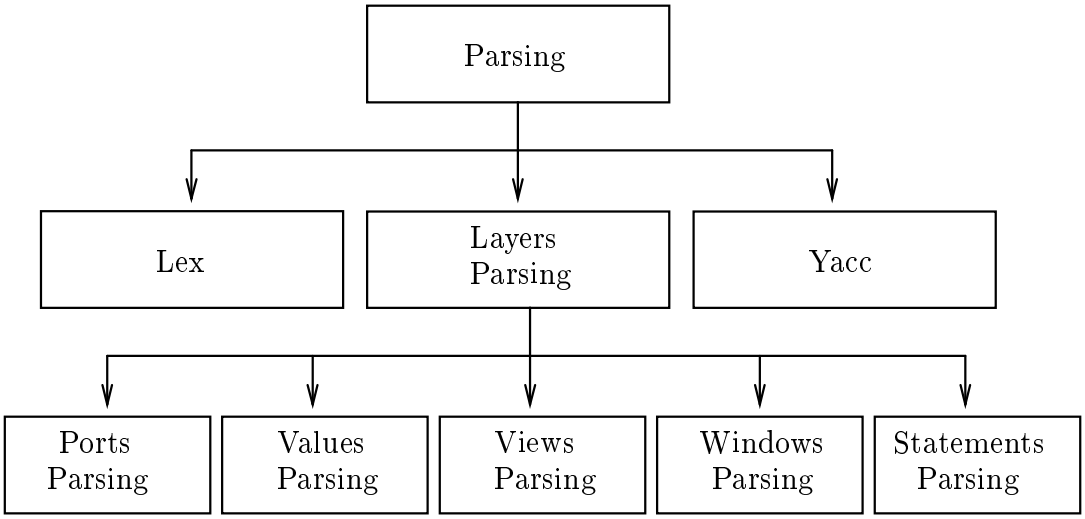


Figure 37: Parsing sub-modules.

A.1.2 Language Layers

This module is composed by five sub-modules as is shown in Figure 38. Then, a sub-module is defined for each component of the language in order to define the services for each layer (constructors, selectors, operators, etc.).

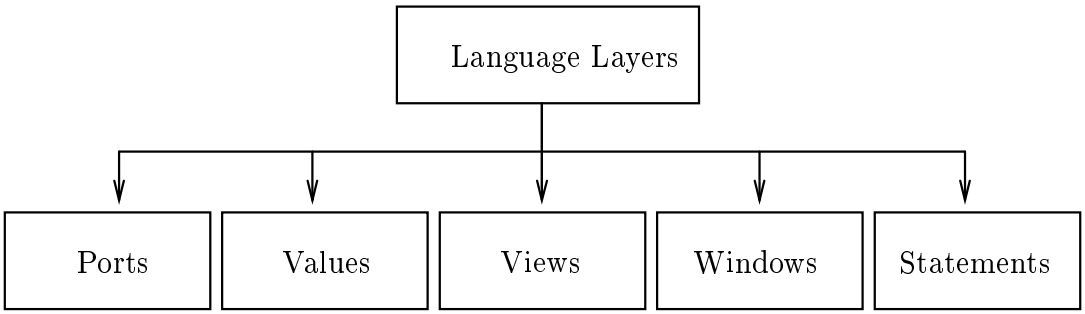


Figure 38: Language Layers sub-modules.

A.1.3 X-Windows Manage

This module manages the graphic output to the X-Windows System.

A.2 The Image Translator Architecture

The image translator main modules are shown in Figure 39.

Then, the image translator is composed by three sub-systems.

- The Parsing module loads the X-image (X Bitmap or colour X Pixmap) in an internal format in order to be processed.

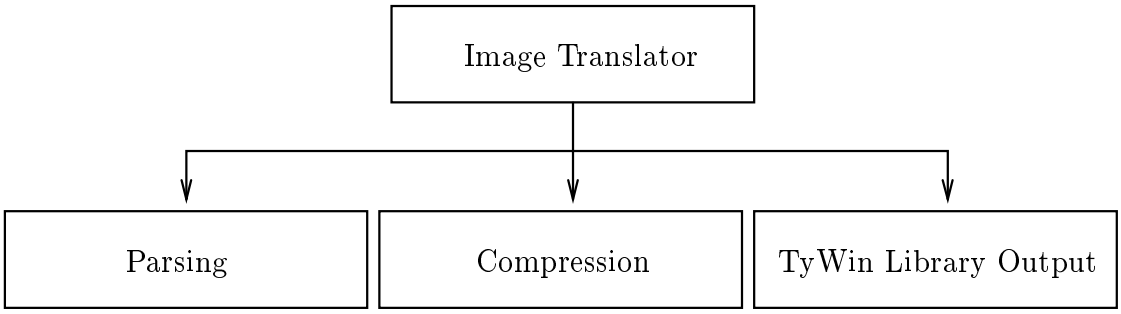


Figure 39: Image Translator Modules - Architecture.

- The Compression module compresses the image.
- The ***TyWin*** Library Output module saves the new ***TyWin*** ideogram file in the directory of libraries.

B The *TyWin Language* Syntax

B.1 Ports

Simple Ports

simple_port	Constructor:	∘	Port (*vector* , *planes*)
vector	Constructor:	∘	(*position* | *position*)
position	Constructor:	∘	⟨ *integer* , *integer* ⟩
planes	Constructors:	∘	[*colour* : *colour* : *colour* : *colour* : *colour*]
		∘	[*colour* : *colour* : *colour* : *colour*]
		∘	[*colour* : *colour* : *colour*]
		∘	[*colour* : *colour*]
		∘	[*colour*]
colour	Constructor:	∘	< *complevel* , *complevel* , *complevel* >
complevel	Constructors:	∘	Min
		∘	Mdn
		∘	Mid
		∘	Mdx
		∘	Max

Port lists

port_list	Constructors:	∘	Nil	
		∘	(*simple_port* # *port_list*)	
	Selectors:	∘	Head (*port_list*)	→*simple_port*
		∘	Tail (*port_list*)	→*port_list*
		∘	Length (*port_list*)	→*integer*

Port's operations

port	Operators:	∘	Rotate (*port*)	→*port*
		∘	UnRotate (*port*)	→*port*
		∘	BiRotate (*port*)	→*port*
		∘	SplitInt (*port* , *range* , *integer*)	→*port*
		∘	SplitReal (*port* , *real*)	→*port*
		∘	(*port* + *port*)	→*port*
		∘	(*port* ∗ *port*)	→*port*
		∘	(*port* − *port*)	→*port*
		∘	SetColour (*port* , *planes*)	→*port*
range	Constructor:	∘	(*integer* .. *integer*)	

B.2 Values

Simple values

T	Constructor:	◦	tt	
boolean	Constructors:	◦	True	
		◦	False	
real	Constructors:	◦	*real_constant*	
		◦	Aureate	
	Operators:	◦	(*real* + *real*)	→*real*
		◦	(*real* ∗ *real*)	→*real*
		◦	(*real* − *real*)	→*real*
		◦	(*real* / *real*)	→*real*
		◦	(− *real*)	→*real*
integer	Constructors:	◦	*integer_constant*	
		◦	Width	
		◦	Height	
	Operators:	◦	(*integer* + *integer*)	→*integer*
		◦	(*integer* ∗ *integer*)	→*integer*
		◦	(*integer* − *integer*)	→*integer*
		◦	(*integer* / *integer*)	→*integer*
		◦	(− *integer*)	→*integer*

Collections

list	Constructors:	◦	[ ]
		◦	[*collection_values*]
set	Constructors:	◦	{ }
		◦	{ *collection_values* }
collection_values	Syntax:	◦	*value*
		◦	*collection_values* : *value*

Composed values

record	Constructor:	◦	< *record_values* >
record_values	Syntax:	◦	*value*
		◦	*record_values* :: *value*
pair	Constructor:	◦	(*value* , *value*)
union	Constructors:	◦	InL (*value*)
		◦	InR (*value*)

B.3 Views

view	Constructors:	∘	VT (*port*)	
		∘	VBoolean (*port*)	
		∘	VReal (*port*)	
		∘	VInteger (*range* , *port*)	
		∘	VList (*range* , *integer* , *view*)	
		∘	VSet (*view*)	
		∘	VRecord (*record_views*)	
		∘	VUnion (*view* , *view*)	
		∘	VProd (*view* , *view*)	
	Operator:	∘	ToPort(*view* , *value*)	→*portlist*
record_views	Constructors:	∘	*view*	
		∘	*record_views* :: *view*	

B.4 Windows

type	Constructors:	∘	T	
		∘	Boolean	
		∘	Real	
		∘	Integer *range*	
		∘	List of (*range* , *integer* , *type*)	
		∘	Set of *type*	
		∘	Record (*record_types*)	
		∘	(*type* + *type*)	
		∘	(*type* ∗ *type*)	
	Operator:	∘	ToView (*type* , *port*)	→*view*
record_types	Constructors:	∘	*type*	
		∘	*record_types* :: *type*	

B.5 Sentences

assignment	∘	*identifier* = *expression* ;
file_inclusion	∘	#include "*filename*";
display	∘	display (*port*) ;
free	∘	free (*identifier*) ;

C ToPort operator specification

C.1 Simple values

The rules for simple values are:

$ToPort(VT(port), tt) \doteq port$
$ToPort(VBoolean(port), False) \doteq Nil$ $ToPort(VBoolean(port), True) \doteq port$
$ToPort(VReal(port), real\_val) \doteq SplitReal(port, real\_val)$
$ToPort(VInteger(range, port), int\_val) \doteq SplitInt(port, range, int\_val)$

C.2 Collections

The rules for collections are:

$ToPort(VList(range_{empty}, rot\_int, sub\text{-}view), [val_1 : ... : val_n]) \doteq$ $ToPort(VRot_{rot\_int}(VSplit(sub\text{-}view, (1..n), 1))), val_1)$ $+ \ldots +$ $ToPort(VRot_{rot\_int}(VSplit(sub\text{-}view, (1..n), n))), val_n)$ $ToPort(VList((r_1..r_2), rot\_int, sub\text{-}view), [val_1 : ... : val_n]) \doteq$ $ToPort(VRot_{rot\_int}(VSplit(sub\text{-}view, (1..(r_2 - r_1)), 1))), val_{r_1})$ $+ \ldots +$ $ToPort(VRot_{rot\_int}(VSplit(sub\text{-}view, (1..(r_2 - r_1)), (r_2 - r_1)))), val_{r_2})$
$ToPort(VSet(sub\text{-}view), \{\}) \doteq Nil$ $ToPort(VSet(sub\text{-}view), \{val_1 : val_2 : ... : val_n\}) \doteq$ $ToPort(sub\text{-}view, val_1) + ToPort(VSet(sub\text{-}view), \{val_2 : ... : val_n\})$

Where:

- $VRot_{rot\_int}(view)$ rotates the ports in the view *rot_int* times.
- $VSplit(view, range, int\_index)$ splites the port in the view in function to *range* and *int_index*.

C.3 Composed values

The rules for records, disjoint unions and cartesian products are:

$ToPort(VRecord(sub\text{-}view_1 :: ... :: sub\text{-}view_n), < val_1 :: ... :: val_n >) \doteq$ $ToPort(sub\text{-}view_1, val_1) + \ldots + ToPort(sub\text{-}view_n, val_n)$
$ToPort(VUnion(sub\text{-}view_1, sub\text{-}view_2), InL(val)) \doteq ToPort(sub\text{-}view_1, val)$ $ToPort(VUnion(sub\text{-}view_1, sub\text{-}view_2), InR(val)) \doteq ToPort(sub\text{-}view_2, val)$
$ToPort(VProd(sub\text{-}view_1, sub\text{-}view_2), (val_1, val_2)) \doteq$ $ToPort(sub\text{-}view_1, val_1)) * ToPort(sub\text{-}view_2, val_2)$