Image taxonomy in Olena

Christophe Berger

LRDE – EPITA Research and Development Laboratory

May 24, 2006
Outline

1. Introduction

2. Image Taxonomy
   - Types
   - Hierarchies
   - Plugging rules
   - Morphers

3. Conclusion
About Olena:

- Generic image processing library.
- Developed at the LRDE [ole].
- Based on SCOOPv1 paradigm [BDLG^+03].
Image Type Taxonomy

**Taxonomy**

The classification of organisms in an ordered system that indicates natural relationships.

In our context, the organisms are image types.
Our goals are:

- Having many and efficient types in the next release.
- Enhancing type coherence within the hierarchy.
- Enforcing code factoring through a high level of genericity.
- Achieving to properly implement image abstractions.
Figure: SCOOP inheritance model [Ger06]

masked_image could be an image_2d, but we don’t know.
The SCOOP2 paradigm [Ger06].

Now *masked_image* inherits from *image_entry* which is plugged on the proper hierarchies.

**Figure:** SCOOP2 inheritance model [Ger06]
Modus operandi

- Bring together interesting and relevant image types.
- Find out appropriate abstract hierarchies.
- Bring forth general image properties.
- Define the pluging rules to link the image types with the abstractions.
We want to define some primitive image types,
- to test the hierarchies, SCOOP2 and “properties”.
- to give the user the basic types he needs.

They have some particularities that make them special and interesting to work on.
Basic 2D image

Basic 2D image: \textit{image\_2d< T>}

**Definition**
- \( T \) is the pixel value type.
- Bounding box.

**Representation**

\[
ima = \{ data = \ldots 2D \text{ buffer of } T \ldots, bbox \}
\]

**Example**

2D Bounding box example
- from (8, 8) with dimension (4, 6)
- from (8, 8) to (12, 18)
Functional image: \textit{fun\_image}\langle T, G\rangle

**Definition**

- $G$ is the image grid type.
- $T$ is the pixel value type.
- Data given by the function $f : point \rightarrow value$.
- $f$ is defined on a domain $D_f$, and $bbox \subseteq D_f$.
- A specific point in image cannot be modified.

**Representation**

$ima = \{ f : p \rightarrow v, bbox \}$

**Example**

$ima = \{ f = (r, c) \rightarrow r + c, bbox = \{(3, 3) \text{ to } (5, 7)\} \}$
point, value set image

Point, value set image: \textit{pvset\_image}< T, G >

**Definition**
- \( G \) is the image grid type.
- \( T \) is the pixel value type.
- Data stored in a \textit{vector}< \textit{pair}< \textit{point}, \textit{value}> >
- Bounding box.

**Representation**
\[
ima = \{ \text{vec} = \ldots(p, v)\ldots, \text{bbox} \}
\]

**Example**
\[
ima = \{ \text{vec} = [(0, 1), 2); (0, 2), 4); (1, 2), 3)], \text{bbox} = \\
{(0, 0) \text{ to } (2, 3)} \}\\
\]
Constant image: \( \text{cst\_image}\langle T, G \rangle \)

**Definition**
- \( G \) is the image grid type.
- \( T \) is the pixel value type.
- One unique pixel value but value wise mutable.
- Bounding box.

**Example**
\[
\text{ima} = \{ \text{val} = 3, \text{bbox} = \{(3, 5) \text{ to } (8, 7)\}\}
\]
We can write: \( \text{ima.value}(3) \leftarrow 5 \)
Abstract hierarchies

We specified basic types, we now have to characterise them.

We will use abstractions from image processing area.

Finer are the abstraction hierarchies, finer are the characterisation.

We are looking for orthogonal hierarchies.
Type of support / domain:

Figure: Image defined by its support type

Example

*image_2d*<int> is an *image_2d_rec*. 
Image contents:

Figure: Image defined by its contents

Example

image_2d<rgb3x8> is a color_image.
Nature of values:

![Image defined by its values](figure)

**Example**

- `image_2d<rgb3x8>` is a `vectorial_image`
- `image_2d<color_name>` is a `string_image` (`ima[p] = "red"`).
- `image_2d<color_wavenlen>` is a `scalar_image` (`ima[p] = 355`).
Point wise accessibility

Figure: Point wise accessibility

Example

<table>
<thead>
<tr>
<th>image_2d</th>
<th>ima[p] works because it provides a buffer access.</th>
</tr>
</thead>
<tbody>
<tr>
<td>pvset_image</td>
<td>ima[p] does not work because we do not want to browse the entire set of (p, v)!</td>
</tr>
</tbody>
</table>
Value wise accessibility

Figure: value wise accessibility

Consequences

\[
\text{ima.value(value&)} \leftarrow \text{value} \\
\text{ima.value(value&)} \text{ const} \rightarrow \{\text{values}\}
\]

Example: \(\text{ima.value}(3) = 5\)

This is not available with \textit{image\_2d} because we do not store the function \(f : v \rightarrow \{p\}/\text{ima}[p] = v\). It works with \textit{cst\_image}. 
Image constness

Figure: Constness

Consequences

- mutable_image
  - lvalue& op[](const site& s)
  - const rvalue& op[](const site& s) const
- const_image
  - const rvalue& op[](const site& s) const
Properties

What is a property?

A property is a notion, an abstraction.

Properties are seen in implementation as concrete types.

Example

image \rightarrow \text{value} \\
image2d<int> \rightarrow \text{value=int}
Plugging to the appropriate hierarchies

Properties examples

Properties are deduced from types.

- image → value
- image → grid
- grid → point
- grid → iterator

We need:

- to list and define relevant properties for our types.
- to express the rules to allow the abstraction of a given hierarchy to be deduced from a set of image properties.
- a static mechanism that plugs to the appropriate hierarchies.
First example

Properties of $image_{2d}<T>$

- **value**: $T$
- **grid**: Grid2d
- **point**: Point2d
- **site**: Point2d
- **size**: BBox2d
- **mutable**: true
- **pwra**: true
- **vwra**: false

Abstract hierarchies describing $image_{2d}<T>$

- $image_{2d}_{\text{rec}}$
- $pw_{\text{accessible}}_{image}$
- $mutable_{image}$
- case of $T$ is int: data_image and scalar_image
- case of $T$ is rgb3x8: color_image and vectorial_image
Second example

Properties of $cst\_image<T, G>$

- **value**: $T$
- **grid**: $G$
- **site**: $G::site$
- **point**: $G::point$
- **mutable**: true
- **pwra**: false
- **vwra**: true

Abstract hierarchies describing $cst\_image<T, G>$

- **vw_accessible_image**
- **mutable_image**
- case of $T$ is int: **data_image** and **scalar_image**
- case of $G$ is Grid2d: **image_2d_rec**
What is a morpher?

Morphers are:

- Special types of images.
- Modify sometimes properties of an image.

**Example**

A morpher $m$ applied on an image $ima$ gives a new image $ima' = m(ima)$.

A morpher can be seen as a function, but creates a new image using the properties of $ima$.

Exemple: *a_particular_morpher<l, aux_params>*.
Morpher example: image seen through a function

Definition: `image_through_fun<l, fun>`

\[ ima' = \{ima; f : value \rightarrow value\}. \]

Properties

- **output**: `output(F, l::value)`
- **input**: `l`
- **mutable**: `false`
- **pwra**: `l::pwra`

... and so on...

Example

```
image<int> ima = [2, 4, 6, 1, 0, 8]
function f(i) = i + 3
ima' = [5, 7, 9, 4, 3, 11]
```
Morpher example: image seen through a bijective function

**Definition:** \( \text{image	extunderscore through	extunderscore bijective	extunderscore fun}<I, \text{bijfun}> \)

\[ \text{ima}' = \{ \text{ima} ; f = \{ \text{f}\text{dir} : \text{value} \rightarrow \text{value}, \text{f}\text{inv} : \text{value} \rightarrow \text{value} \} \}. \]

**Properties**

- **output** \( \text{output}(F, l::\text{value}) \)
- **input** \( I \)
- **mutable** \( \text{true} \)
- **pwra** \( l::\text{pwra} \)
  - ... and so on...

**Example**

\( \text{ima} = [2, 4, 6, 1], \text{f}\text{dir}(i) = i + 3, \text{f}\text{inv}(i) = i - 3 \)

consequently \( \text{ima}' = [5, 7, 9, 4] \)

then \( \text{ima}'[p] = 5/p = 2^{\text{nd}} \text{element} \Rightarrow \text{ima}' = [5, 5, 9, 4] \) but \( \text{ima} = [2, 2, 6, 1] \).
Noticed ambiguities

This taxonomy led to:

- ambiguities detection
- more relevant and precise definitions

Example:

- Grids: regular v.s. graph.
- Image contents v.s. storage type.
- Image contents v.s. Contents nature.
- Point wise, value wise and constness.
Work relevance

To conclude:

- Innovative and proheminent work.
- Starting point for the next release.
- Documentation.
- Study of the best hierarchy before starting the code.
- Assurance of global coherance.
A static C++ object-oriented programming (SCOOP) paradigm mixing benefits of traditional OOP and generic programming.
In Proceedings of the Workshop on Multiple Paradigm with OO Languages (MPOOL), Anaheim, CA, USA, October 2003.

Thierry Geraud.