Diderot
A parallel domain-specific language for image analysis

John Reppy

University of Chicago

September 6, 2010

Diderot

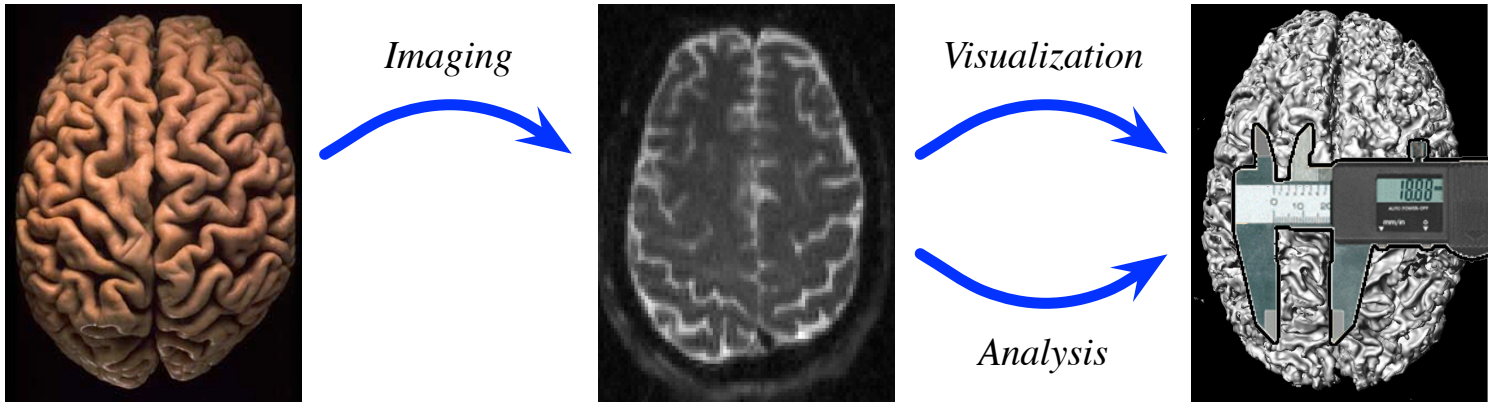
Diderot is a cross-discipline project involving

- Scientific visualization
- Programming languages

We are trying to use ideas from programming languages to improve the state of the art in image-analysis and visualization.

Joint work with Gordon Kindlmann and Lamont Samuels.

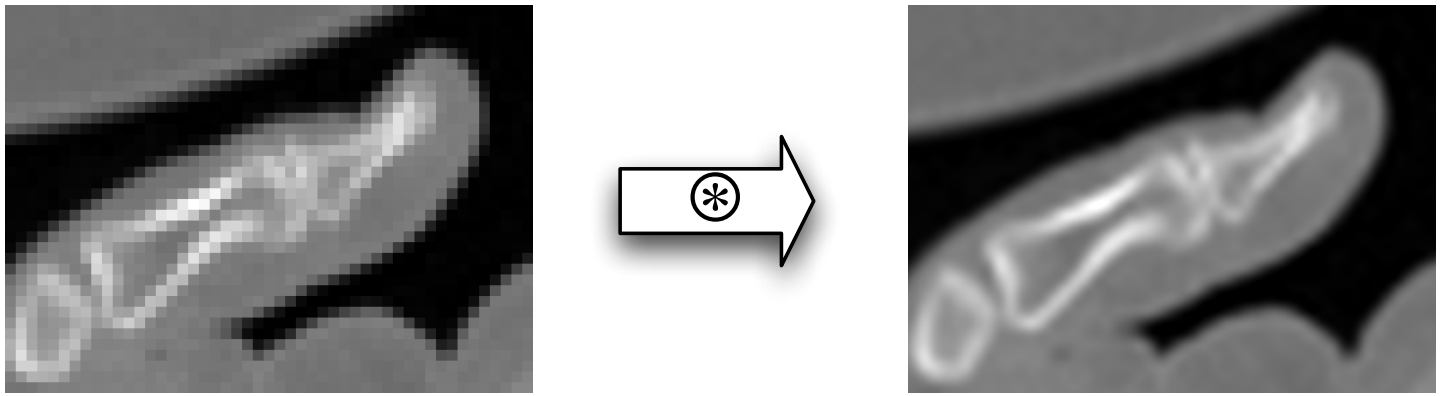
Why image analysis is important



Scientists need tools to extract structure from many kinds of image data.

Image analysis and visualization

- We are interested in a class of algorithms that compute a geometric properties of some object from imaging data.
- These algorithms compute over a continuous field that is reconstructed from discrete data.



Discrete image data *Continuous field*

Image analysis and visualization *(continued ...)*

Examples include

- Direct volume rendering (requires reconstruction, derivatives)
- Fiber tractography (requires tensor fields)
- Particle systems

Image analysis and visualization *(continued ...)*

Examples include

- Direct volume rendering (requires reconstruction, derivatives)
- Fiber tractography (requires tensor fields)
- Particle systems

Image analysis and visualization *(continued ...)*

Examples include

- Direct volume rendering (requires reconstruction, derivatives)
- Fiber tractography (requires tensor fields)
- Particle systems

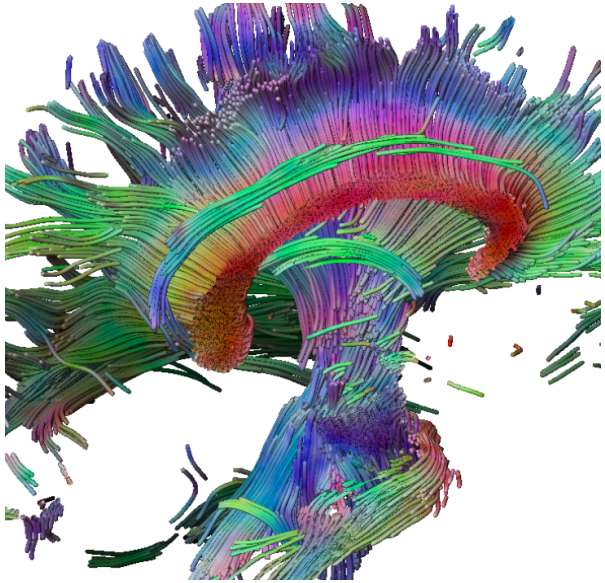
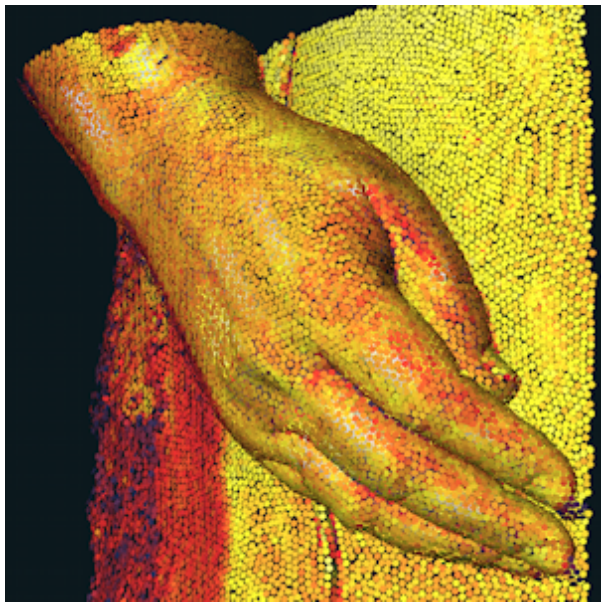


Image analysis and visualization *(continued ...)*

Examples include

- Direct volume rendering (requires reconstruction, derivatives)
- Fiber tractography (requires tensor fields)
- Particle systems



We want domain-specific support for image analysis

We have three design goals:

- Provide a high-level mathematical programming model that abstracts away from discrete image data and the target architecture.
- Use domain knowledge to get good performance on a range of parallel platforms.
- Make it possible for scientists to develop their own image analyses and visualizations ("Matlab" for image analysis).

We want domain-specific support for image analysis

We have three design goals:

- Provide a high-level mathematical programming model that abstracts away from discrete image data and the target architecture.
- Use domain knowledge to get good performance on a range of parallel platforms.
- Make it possible for scientists to develop their own image analyses and visualizations ("Matlab" for image analysis).

We want domain-specific support for image analysis

We have three design goals:

- Provide a high-level mathematical programming model that abstracts away from discrete image data and the target architecture.
- Use domain knowledge to get good performance on a range of parallel platforms.
- Make it possible for scientists to develop their own image analyses and visualizations ("Matlab" for image analysis).

We want domain-specific support for image analysis

We have three design goals:

- Provide a high-level mathematical programming model that abstracts away from discrete image data and the target architecture.
- Use domain knowledge to get good performance on a range of parallel platforms.
- Make it possible for scientists to develop their own image analyses and visualizations ("Matlab" for image analysis).

Approaches to domain-specific programming

There are three common approaches to supporting domain-specific programming:

1. Libraries; *e.g.*, VTK/ITK, OpenCV, Teem, ...
2. Embedded domain-specific languages (EDSL); *e.g.*, many Haskell examples, Ct (C++), Delite (Scala), ...
3. Domain-specific languages (DSL); *e.g.*, R, Renderman, TEX, ...

Approaches to domain-specific programming

There are three common approaches to supporting domain-specific programming:

1. Libraries; *e.g.*, VTK/ITK, OpenCV, Teem, ...
2. Embedded domain-specific languages (EDSL); *e.g.*, many Haskell examples, Ct (C++), Delite (Scala), ...
3. Domain-specific languages (DSL); *e.g.*, R, Renderman, TEX, ...

Approaches to domain-specific programming

There are three common approaches to supporting domain-specific programming:

1. Libraries; *e.g.*, VTK/ITK, OpenCV, Teem, ...
2. Embedded domain-specific languages (EDSL); *e.g.*, many Haskell examples, Ct (C++), Delite (Scala), ...
3. Domain-specific languages (DSL); *e.g.*, R, Renderman, TEX, ...

Approaches to domain-specific programming

There are three common approaches to supporting domain-specific programming:

1. Libraries; *e.g.*, VTK/ITK, OpenCV, Teem, ...
2. Embedded domain-specific languages (EDSL); *e.g.*, many Haskell examples, Ct (C++), Delite (Scala), ...
3. Domain-specific languages (DSL); *e.g.*, R, Renderman, TEX, ...

Comparing approaches

	Notation	Optimization	Parallelism	Impl. Effort	Client Effort
Library	−	0	−	+	−
EDSL	−	+	+	0	0
DSL	+	+	+	−	+

Comparing approaches

	Notation	Optimization	Parallelism	Impl. Effort	Client Effort
Library	−	0	−	+	−
EDSL	−	+	+	0	0
DSL	+	+	+	−	+

In our case, Diderot is a small language, so the extra implementation effort over an EDSL is small (front-end is 2,600 lines of SML code).

Diderot programming model

- The Diderot programming model is based on a collection of mostly autonomous actors that are embedded in a continuous tensor field.
- Actors compute in a bulk-synchronous style.
- Diderot abstracts away from details such as the image-data format, the representation of reals (float vs double), and the target machine (*e.g.*, CPU vs GPU).

Diderot programming model

- The Diderot programming model is based on a collection of mostly autonomous actors that are embedded in a continuous tensor field.
- Actors compute in a bulk-synchronous style.
- Diderot abstracts away from details such as the image-data format, the representation of reals (float vs double), and the target machine (*e.g.*, CPU vs GPU).

Diderot programming model

- The Diderot programming model is based on a collection of mostly autonomous actors that are embedded in a continuous tensor field.
- Actors compute in a bulk-synchronous style.
- Diderot abstracts away from details such as the image-data format, the representation of reals (float vs double), and the target machine (*e.g.*, CPU vs GPU).

Value types

Diderot has several types of values, including booleans, integers, and strings. The most important type of values are tensors.

$$\texttt{tensor}\underbrace{[d_1, \ldots, d_n]}_{\textit{shape}}$$

This type describes an n-order tensor, where $n \geq 0$ and the $d_i > 1$.
We provide type aliases for common shapes; *e.g.*,

```
real = tensor[]
vec3 = tensor[3]
```

We also normalize `tensor[1]` to `tensor[]`, *etc.*

Image types

Images are used to represent the data sets that we are analyzing, as well as other array data, such as transfer functions.

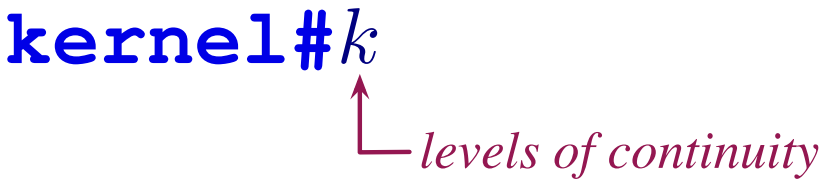
$$\texttt{image}(d)[d_1, \ldots, d_n]$$

dimension *shape*

where $d > 0$, $n \geq 0$, and the $d_i > 1$.
The image type abstracts away from the on-disk representation of the data.

Kernel types

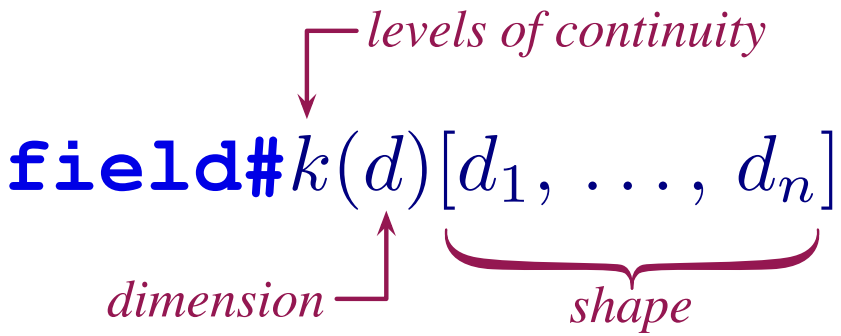
Kernels are used to reconstruct continuous fields from images.



where $k \geq 0$.
We currently restrict ourselves to separable, piecewise polynomial kernel functions with rational coefficients.

Field types

Fields are an abstract representation of functions from some vector space to tensors.



where $k \geq 0$, $d > 0$, and the $d_i > 1$.

Typechecking

The typing rules keep track of the dimensions, shapes, etc.
For example, here is the rule for convolution

$$\frac{\Gamma \vdash V : \texttt{image}(d)[\sigma] \qquad \Gamma \vdash h : \texttt{kernel\#}k}{\Gamma \vdash V \circledast h : \texttt{field\#}k(d)[\sigma]}$$

and for differentiation

$$\frac{\Gamma \vdash F : \texttt{field\#}k(d)[\sigma] \qquad k > 0 \qquad k' = k - 1}{\Gamma \vdash \nabla F : \texttt{field\#}k'(d)[\sigma, d]}$$

Probing tensor fields

- The key operation in Diderot is probing a continuous field to discover its value at some position.
- A Diderot programmer might write code like the following

```
(∇ (img ⊛ bspln3))@pos
```

  which probes a field at position `pos`, where the field is the first derivative of the field reconstructed by applying the convolution kernel `bspln3` to the image data `img`.
- The typing rule for probing a field is

$$\frac{\Gamma \vdash F : \texttt{field\#}k(d)[\sigma] \qquad \Gamma \vdash p : \texttt{tensor}[d]}{\Gamma \vdash F\texttt{@}p : \texttt{tensor}[\sigma]}$$

Probing tensor fields

- The key operation in Diderot is probing a continuous field to discover its value at some position.
- A Diderot programmer might write code like the following

$$(\nabla \ (\texttt{img} \circledast \texttt{bspln3}))@\texttt{pos}$$

  which probes a field at position `pos`, where the field is the first derivative of the field reconstructed by applying the convolution kernel `bspln3` to the image data `img`.
- The typing rule for probing a field is

$$\frac{\Gamma \vdash F : \texttt{field\#}k(d)[\sigma] \qquad \Gamma \vdash p : \texttt{tensor}[d]}{\Gamma \vdash F@p : \texttt{tensor}[\sigma]}$$

Probing tensor fields

- The key operation in Diderot is probing a continuous field to discover its value at some position.
- A Diderot programmer might write code like the following

  $$(\nabla\ (\texttt{img} \circledast \texttt{bspln3}))\texttt{@pos}$$

  which probes a field at position `pos`, where the field is the first derivative of the field reconstructed by applying the convolution kernel `bspln3` to the image data `img`.
- The typing rule for probing a field is

$$\frac{\Gamma \vdash F : \textbf{field\#}k(d)[\sigma] \qquad \Gamma \vdash p : \textbf{tensor}[d]}{\Gamma \vdash F\textbf{@}p : \textbf{tensor}[\sigma]}$$

Diderot program structure

A Diderot program has three parts:

1. Global definitions, including inputs.
2. Actor definitions, which are the computational agents.
3. Initialization, which defines an initial collection of actors.

Diderot program structure

A Diderot program has three parts:

1. Global definitions, including inputs.
2. Actor definitions, which are the computational agents.
3. Initialization, which defines an initial collection of actors.

Diderot program structure

A Diderot program has three parts:

1. Global definitions, including inputs.
2. Actor definitions, which are the computational agents.
3. Initialization, which defines an initial collection of actors.

Diderot program structure

A Diderot program has three parts:

1. Global definitions, including inputs.
2. Actor definitions, which are the computational agents.
3. Initialization, which defines an initial collection of actors.

Actors

- Each actor has a state, which includes its position and an **`update`** method.
- Some state variables are annotated as outputs.
- The execution model is bulk synchronous: at each iteration, every actor's state is updated independently.
- The system iterates until a termination condition is met.

Actors

- Each actor has a state, which includes its position and an **`update`** method.
- Some state variables are annotated as outputs.
- The execution model is bulk synchronous: at each iteration, every actor's state is updated independently.
- The system iterates until a termination condition is met.

Actors

- Each actor has a state, which includes its position and an `update` method.
- Some state variables are annotated as outputs.
- The execution model is bulk synchronous: at each iteration, every actor's state is updated independently.
- The system iterates until a termination condition is met.

Actors

- Each actor has a state, which includes its position and an **`update`** method.
- Some state variables are annotated as outputs.
- The execution model is bulk synchronous: at each iteration, every actor's state is updated independently.
- The system iterates until a termination condition is met.

Example: Maximum intensity projection

MIP is a volume rendering technique that computes the maximum intensity value along rays that are cast into the field.

Example: Maximum intensity projection *(continued ...)*

```
input string dataFile;        // name of dataset
input real stepSz;            // size of steps
input vec3 eye;               // location of eye point
input vec3 orig;              // location of pixel (0,0)
input vec3 cVec;              // vector between pixels horizontally
input vec3 rVec;              // vector between pixels vertically

image(3)[] img = load (dataFile);
field#1(3)[] F = img ⊛ bspln3;
```

Maximum intensity projection *(continued ...)*

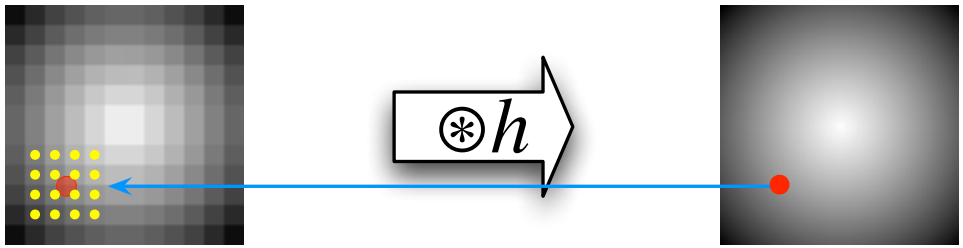
```
actor RayCast (int row, int col)
{
    vec3 pos = orig + real(row)*rVec + real(col)*cVec;
    vec3 dir = (pos - eye) / |pos - eye|;
    real t = 0.0;
    output real maxval = -∞;

    update
    {
        pos = pos + stepSz*dir;
        maxval = max (F@pos, maxval);
        if (t > 20)
            stabilize;
        t = t + stepSz;
    }
}

initially [ RayCast(r, c) | r in 0..199, c in 0..199 ];
```

Probing tensor fields

A probe gets compiled down into code that maps the real-valued coordinates to integers that index the image data. It then samples the image values in the neighborhood of the integer coordinates to compute the result of the probe.



Discrete image data *Continuous field*

This process is similar to texture filtering in Graphics, but it has stronger mathematical requirements.

Exploiting domain knowledge

There are many opportunities to exploit domain knowledge to gain better performance.

One example is the code

$$\begin{aligned} \mathtt{pos}_{i+1} &= \mathtt{pos}_i + \mathtt{stepSz} * \mathtt{dir}; \\ \mathtt{maxval}_{i+1} &= \max(\mathtt{F@pos}_{i+1}, \mathtt{maxval}_i); \end{aligned}$$

To probe the field $\mathtt{F}$, we need to map $\mathtt{pos}_{i+1}$ into image space. Using domain knowledge (linear algebra), we can apply strength reduction.

$$\begin{aligned} \mathtt{pos}_{i+1}^{img} &= \mathbf{M}^{-1}\mathtt{pos}_{i+1} \\ &= \mathbf{M}^{-1}(\mathtt{pos}_i + \mathtt{stepSz} * \mathtt{dir}) \\ &= \mathbf{M}^{-1}\mathtt{pos}_i + \mathbf{M}^{-1}(\mathtt{stepSz} * \mathtt{dir}) \\ &= \mathtt{pos}_i^{img} + \mathtt{delta} \end{aligned}$$

Exploiting domain knowledge

There are many opportunities to exploit domain knowledge to gain better performance.

One example is the code

$$\texttt{pos}_{i+1} = \texttt{pos}_i + \texttt{stepSz} * \texttt{dir};$$
$$\texttt{maxval}_{i+1} = \max(\texttt{F@pos}_{i+1}, \texttt{maxval}_i);$$

To probe the field $\texttt{F}$, we need to map $\texttt{pos}_{i+1}$ into image space. Using domain knowledge (linear algebra), we can apply strength reduction.

$$\begin{aligned}
\texttt{pos}_{i+1}^{img} &= \mathbf{M}^{-1}\texttt{pos}_{i+1} \\
&= \mathbf{M}^{-1}(\texttt{pos}_i + \texttt{stepSz} * \texttt{dir}) \\
&= \mathbf{M}^{-1}\texttt{pos}_i + \mathbf{M}^{-1}(\texttt{stepSz} * \texttt{dir}) \\
&= \texttt{pos}_i^{img} + \texttt{delta}
\end{aligned}$$

Exploiting domain knowledge

There are many opportunities to exploit domain knowledge to gain better performance.

One example is the code

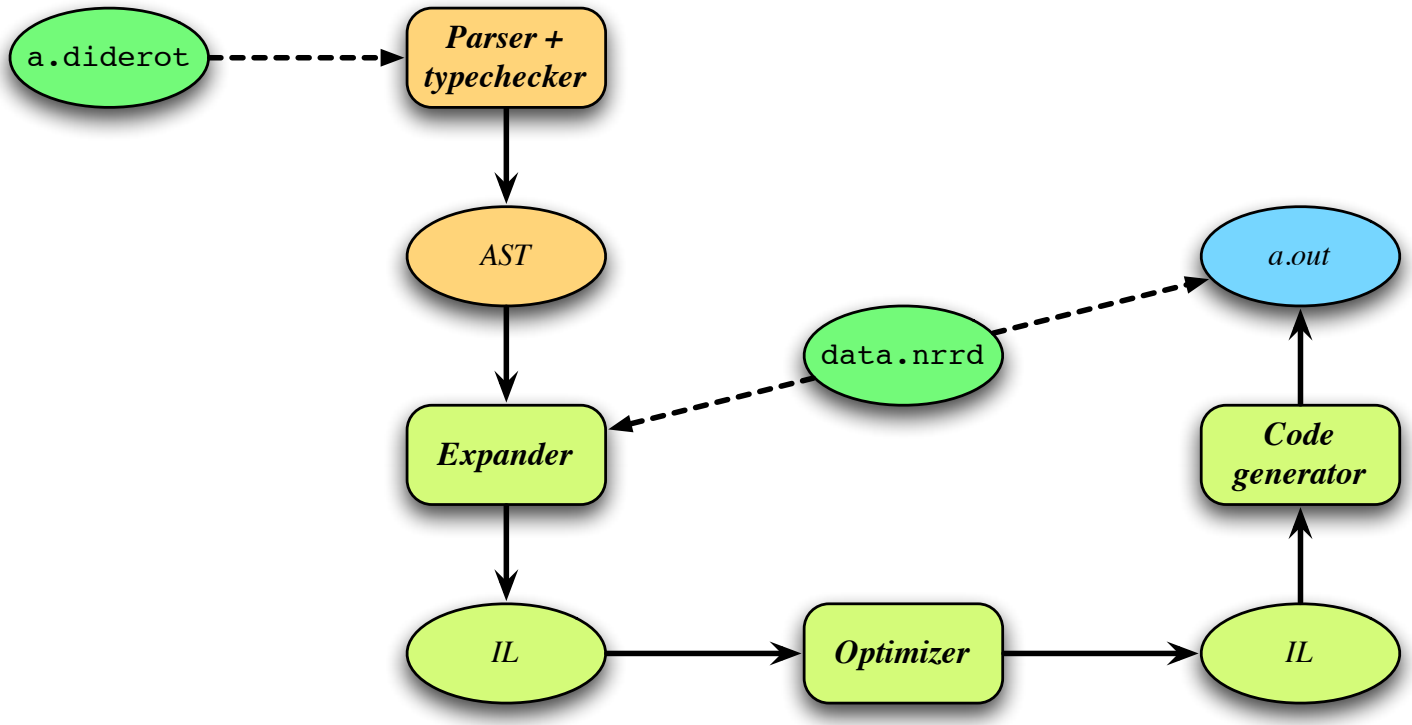
$$\begin{aligned}\texttt{pos}_{i+1} &= \texttt{pos}_i + \texttt{stepSz} * \texttt{dir};\\ \texttt{maxval}_{i+1} &= \max(\texttt{F@pos}_{i+1}, \texttt{maxval}_i);\end{aligned}$$

To probe the field `F`, we need to map $\texttt{pos}_{i+1}$ into image space. Using domain knowledge (linear algebra), we can apply strength reduction.

$$\begin{aligned}\texttt{pos}_{i+1}^{img} &= \mathbf{M}^{-1}\texttt{pos}_{i+1}\\ &= \mathbf{M}^{-1}(\texttt{pos}_i + \texttt{stepSz} * \texttt{dir})\\ &= \mathbf{M}^{-1}\texttt{pos}_i + \mathbf{M}^{-1}(\texttt{stepSz} * \texttt{dir})\\ &= \texttt{pos}_i^{img} + \texttt{delta}\end{aligned}$$

Staged computation

Using staged computation (*e.g.*, partial evaluation) to specialize the code for properties of the image-data.



Other issues

- Type inference and dimension polymorphism.
- Compiler architecture and internal representations that support multiple parallel targets.
- Compiler and runtime techniques to automatically handle very large images.

Other issues

- Type inference and dimension polymorphism.
- Compiler architecture and internal representations that support multiple parallel targets.
- Compiler and runtime techniques to automatically handle very large images.

Other issues

- Type inference and dimension polymorphism.
- Compiler architecture and internal representations that support multiple parallel targets.
- Compiler and runtime techniques to automatically handle very large images.

Long-term goals

In the future, we would like to generalize this work in two directions:

1. Extend Diderot to other classes of algorithms (*e.g.*, object recognition).
2. Generalize approach to other domains.

Status

- First version of language design is done
- Parser and typechecker have been implemented
- Working on naïve code generator with limited optimization that targets OpenCL.

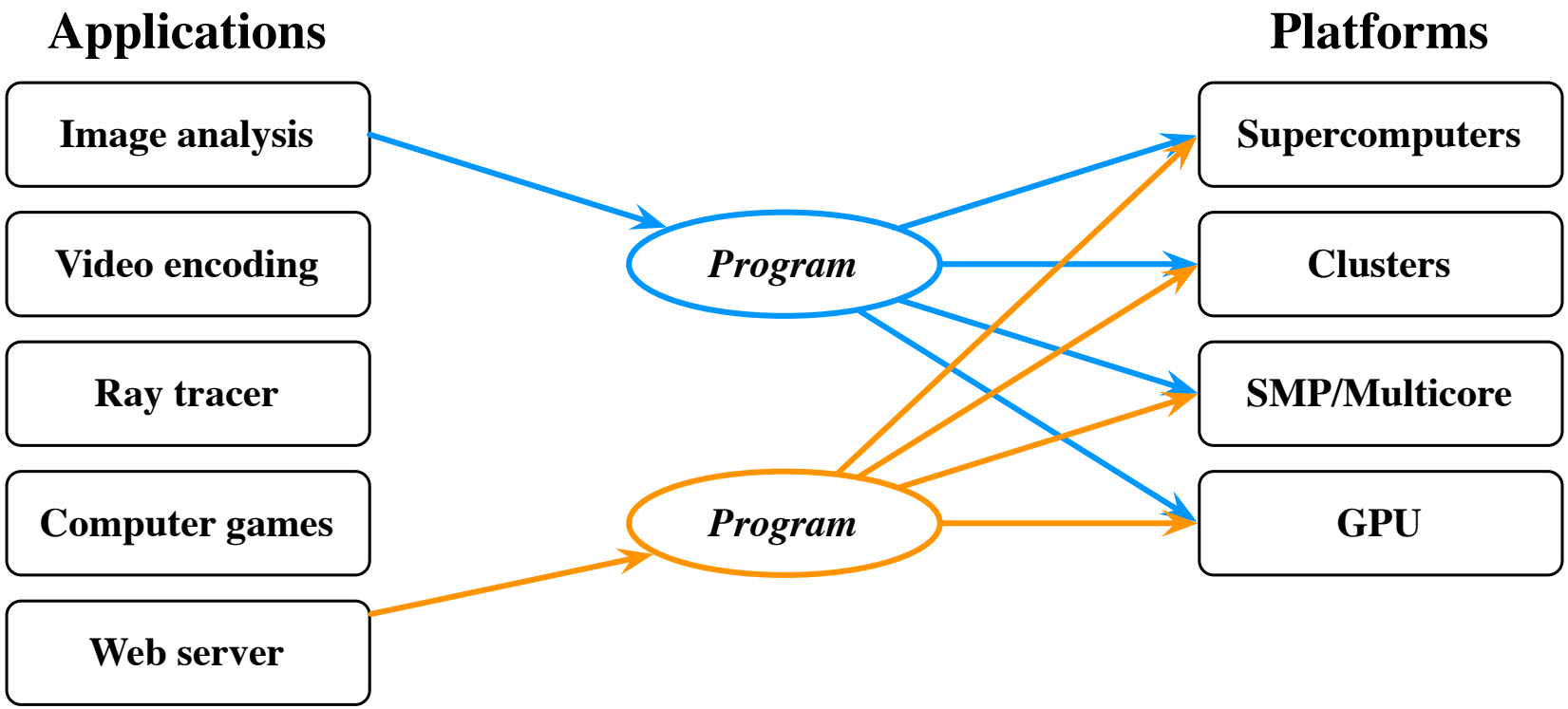
Status

- First version of language design is done
- Parser and typechecker have been implemented
- Working on naïve code generator with limited optimization that targets OpenCL.

Status

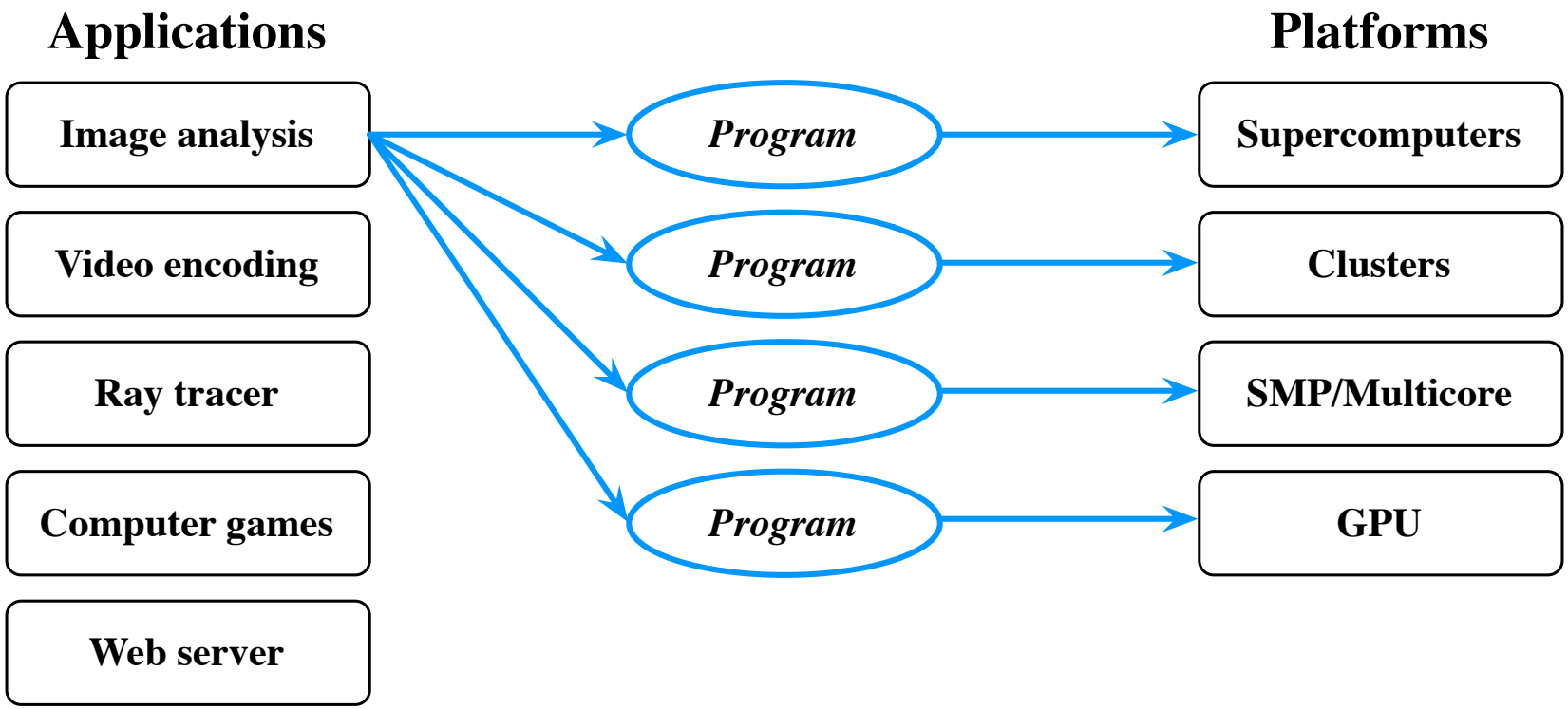
- First version of language design is done
- Parser and typechecker have been implemented
- Working on naïve code generator with limited optimization that targets OpenCL.

Portable parallel programming



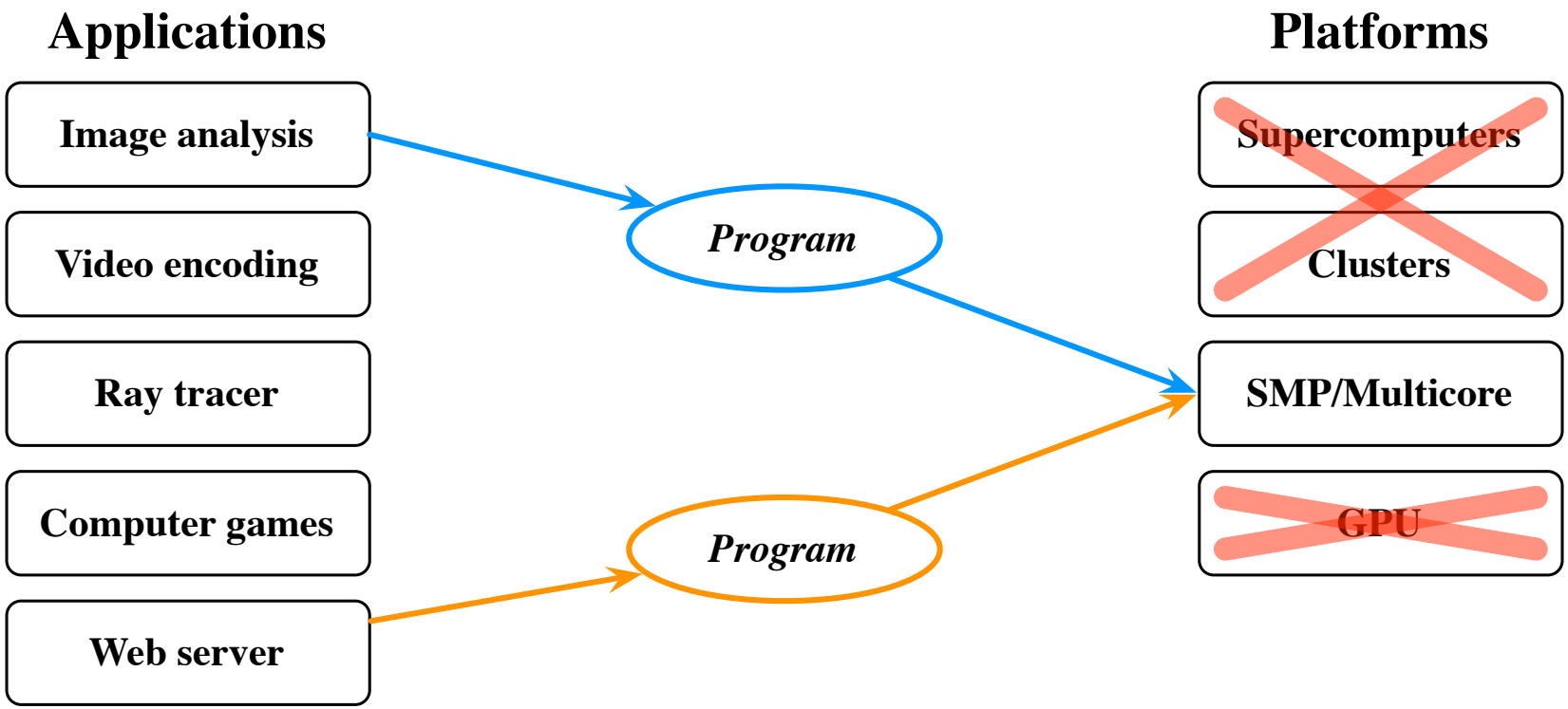
The Ideal: write once, run everywhere, for any application

Portable parallel programming



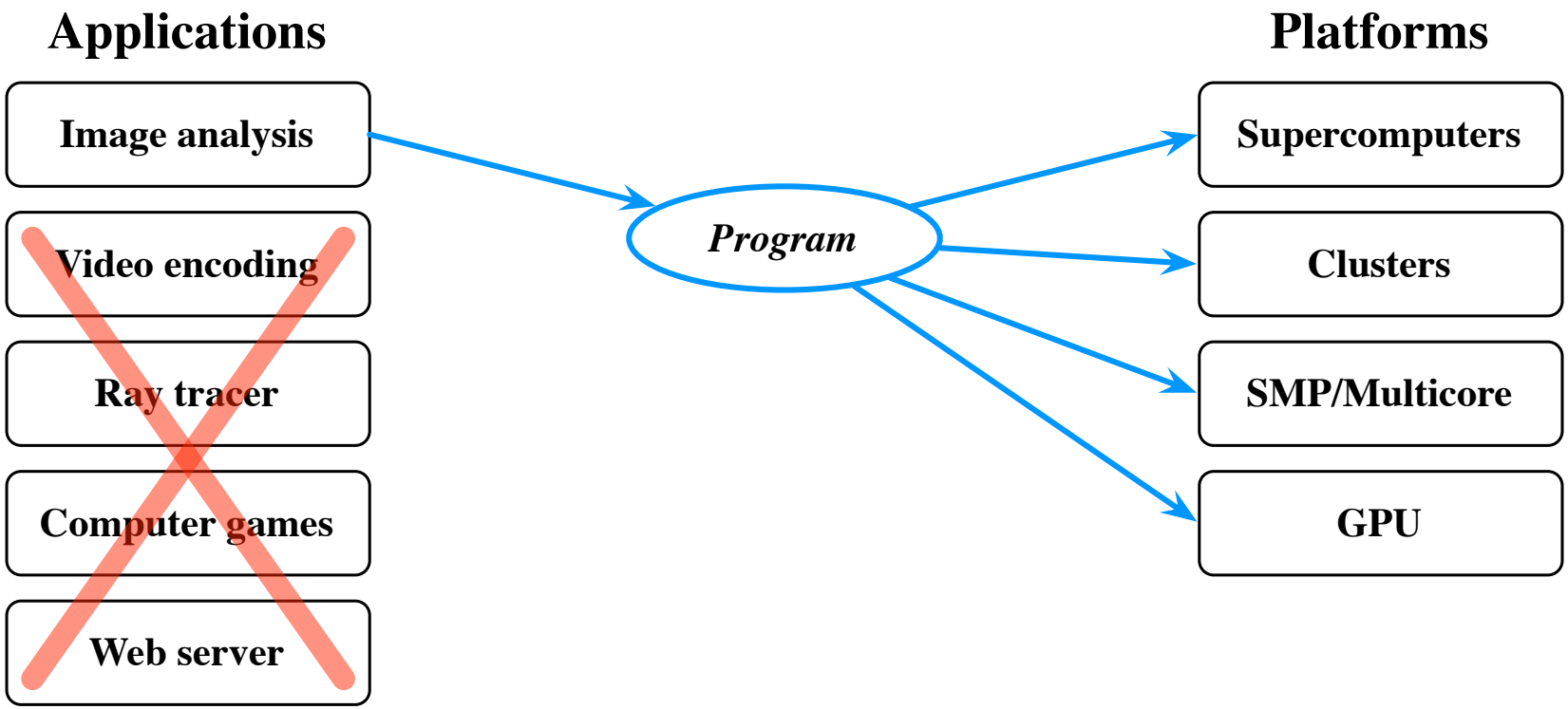
The Reality: write once per platform per application

Portable parallel programming



Manticore: restrict platforms

Portable parallel programming



Diderot: restrict applications

Questions?