Architecturing Software Using a Methodology for Language Development

Charles Consel

Compose Group

IRISA / University of Rennes 1 - INRIA

October 1998

Joint work with Renaud Marlet

Program Family

- Before developing a program:
  - Isolated problem?
  - Member of a program family?
- Program family:
  - A set of programs *sharing* enough characteristics to be studied / developed as a whole.

Program Family Examples

- Program analyzers.
  - Commonalities: equation solver.
  - Variations: languages, properties...
- Device drivers.
  - Commonalities: API, bit operations...
  - Variations: clock, parameters/registers…
- Graphic applications/libraries.
  - Commonalities: basic graphic objects.
  - Variations: layout, behavior...

Hypothesis: Program Family Development

- Given a recognized program family.
- How to develop it?
- Current approaches?

Program Family Development: Libraries

(of functions, objects, components, program patterns...)

- Use depends on the programmer.
  - No systematic re-use.
  - May require expertise.
  - Usability problems for large libraries.
- Properties local to components, not global to the application.
  - Unpredictable global behavior (performance, safety…)

Program Family Development: Genericity

Generic libraries / Generic applications:

- High parameterization.
  - Poor performance.
  - Difficult to use.
- Fast, hand-written specific components.
  - Difficult to maintain.
  - Does not scale up.

Program Family Development: Generators

Library generators / Application generators:

- Combination of building blocks.
- Few or no general-purpose techniques.
- Few or no general-purpose tools.

Program Family Development: General-Purpose Languages

- General-purpose abstractions.
  - “Too” expressive.
- Limited static verifications.
  - Unpredictable.
  - Undecidable.
- Need for dynamic checking.
  - Run-time tests.
  - Dynamic analyses.

Program Family Development: Domain-Specific Languages

- DSL = language with
  - Abstractions (data and control)
  - Notations

  specific to a domain.

- Often:
  - Small.
  - Less expressive than a GPL.
  - More declarative than imperative.

Various Facets of a DSL

- A programming/specification language.
- A dedicated interface to a library/application.
- A structured parameterization mechanism.
- A way to designate a program family member.

DSL: Advantages

- Productivity.
  - Easier programming.
  - Systematic re-use.
- Verification.
  - Easier analyses.
- Performance.
  - Similar to GPL.

DSL Examples (1) In Academia and Industry

- Not a toy concept.
  - Graphics.
  - Financial products.
  - Telephone switching systems.
  - Protocols.
  - Robotics.
  - ...

DSL Example (2)
GAL

Specification language for video device drivers.

- Productivity (compared to hand-coded C).
  - High level.
  - Close to hardware specification.
  - Specification 9 times smaller.
- Verifications.
  - No loop.
  - No bit overlap in register specification.

DSL Example (3)
PLAN-P

Application protocols for programmable networks (extension of PLAN / UPenn).

- Productivity (compared to C).
  - High level.
  - Specification 3 times smaller.
- Verifications (safety and security).
  - Restricted semantics.
  - Global termination.
  - No packet loss or exponential duplication.

DSL: Easier Programming

- Domain-specific abstractions and notations.
  - Conciseness.
  - Readability.
- Declarative (often).
  - What to compute, not how to compute it.
- ➡ Software engineering benefits.
  - Shorter development time.
  - Easier maintenance.

DSL: Systematic Re-Use

- Building blocks = libraries.
- Abstractions = common program patterns.
- Syntax = interface = glue.

➨ Software engineering benefits.

  – Expertise re-use (abstractions + notations).

  – Code re-use (building blocks).

  ➨ Systematic re-use.

DSL: Verification

- Restricted semantics.
  - Designed to make critical properties decidable.
  - Analyzability.
- Software engineering benefits.
  - Safety.
  - Predictability.

Why should you care about DSL?

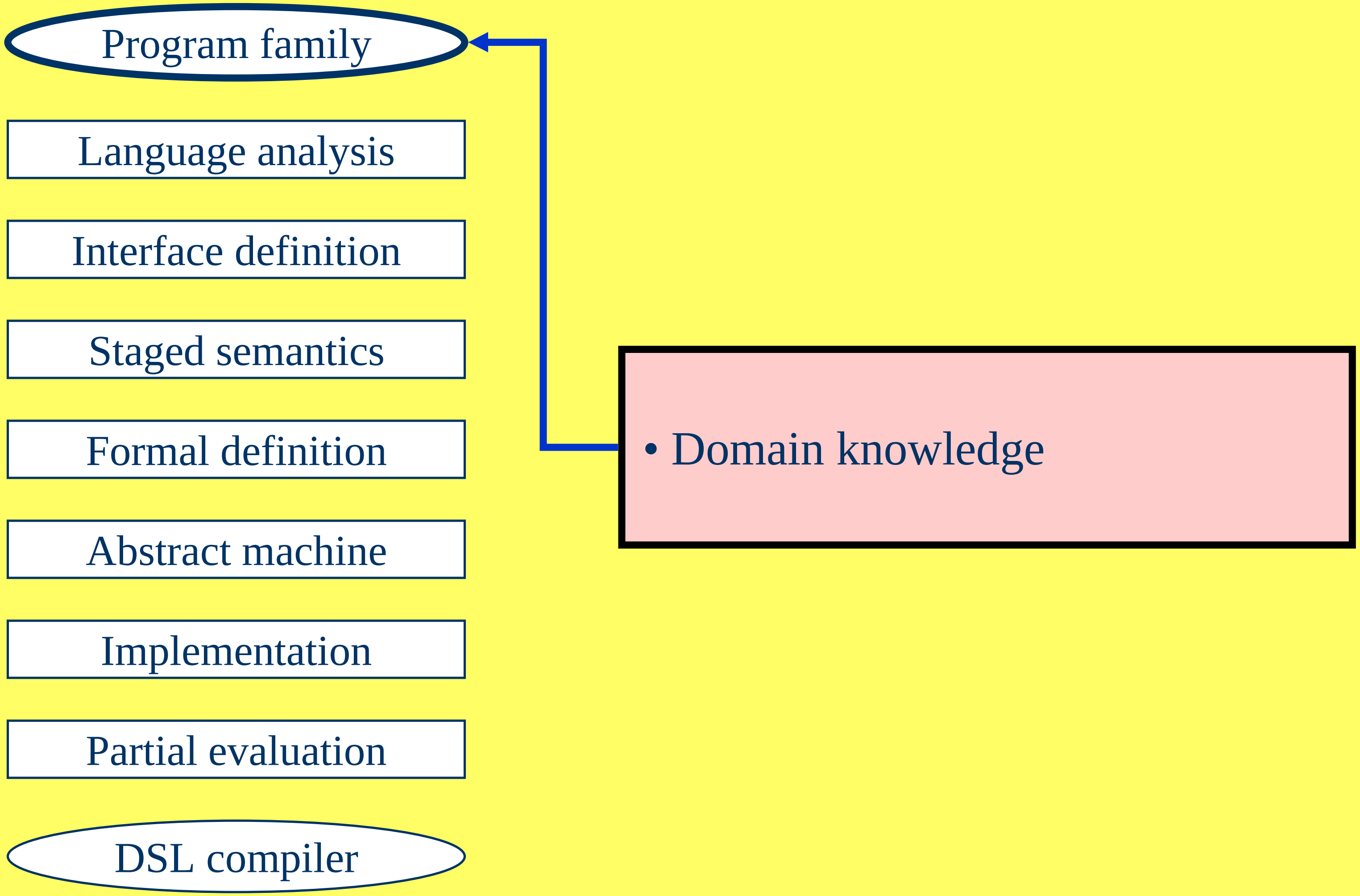
Developing DSLs: Our Potential Contribution

- Who should develop DSLs?
  - Few people have actually designed a language.
- How to develop a DSL?
  - Guidelines for design.
  - Support for implementation.
- ➨ Programming language community.
  - Design expertise.
  - Methodology and tools.

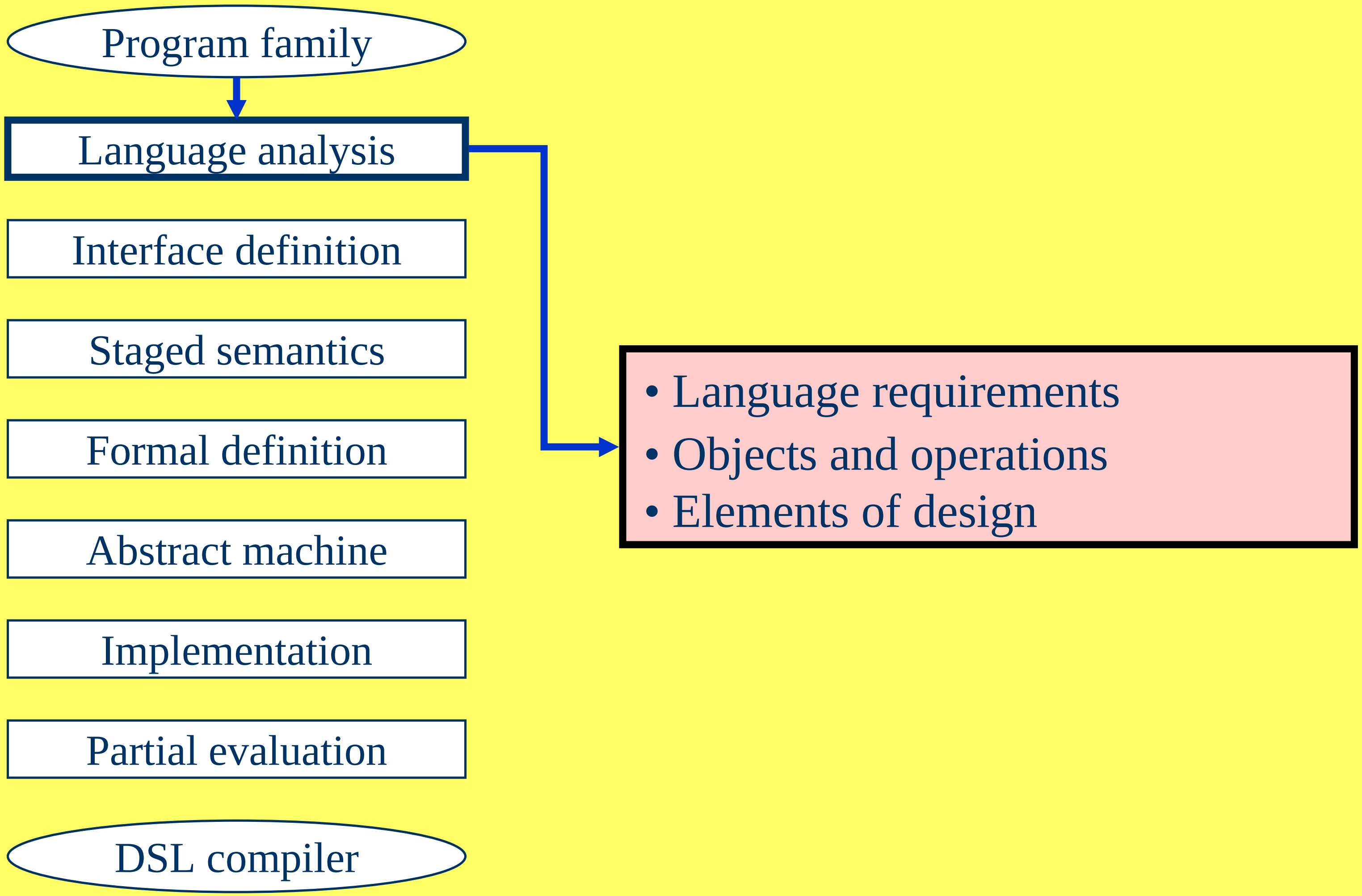
The ***Sprint*** Methodology: Basic Ingredients

- Denotational semantics
  - Key concepts of language design and semantics.
  - Techniques to derive implementations.
  - ➨ *Limitations alleviated by the nature of DSLs.*
- Software architectures
  - Domain expertise (design).
  - Building blocks (algebras).
  - Program patterns (constructs).

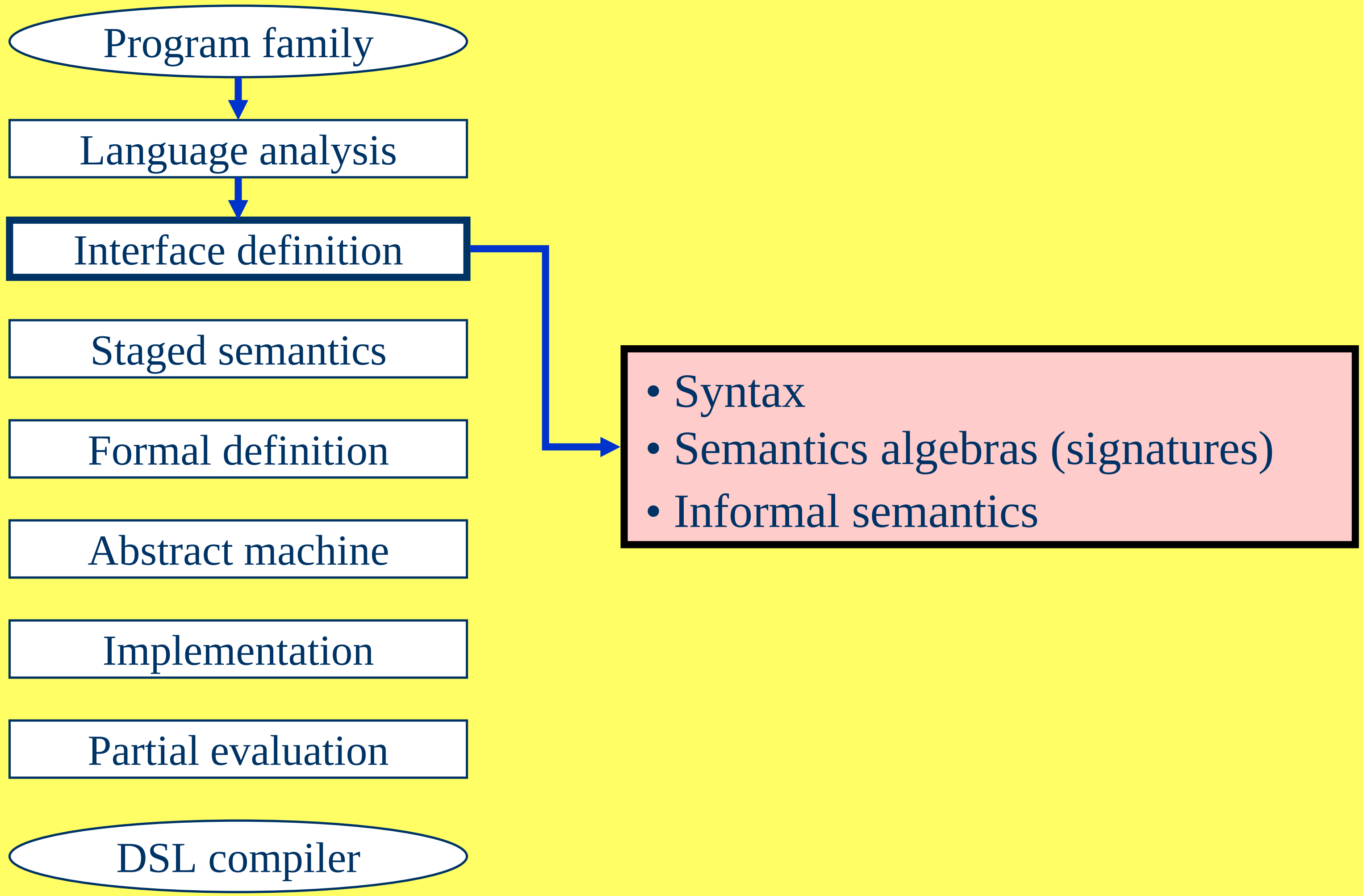
Sprint: An Overview



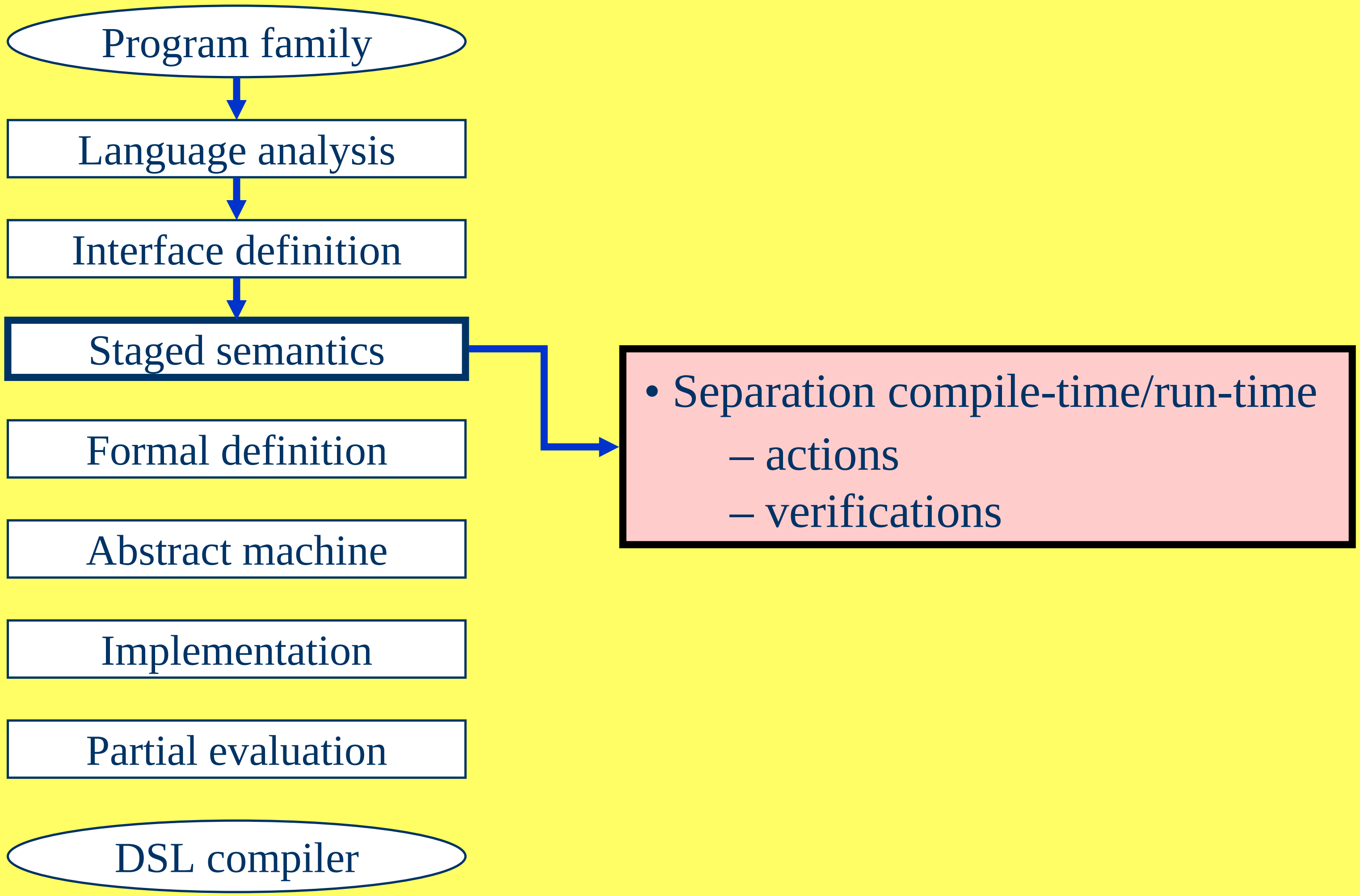
Sprint: An Overview



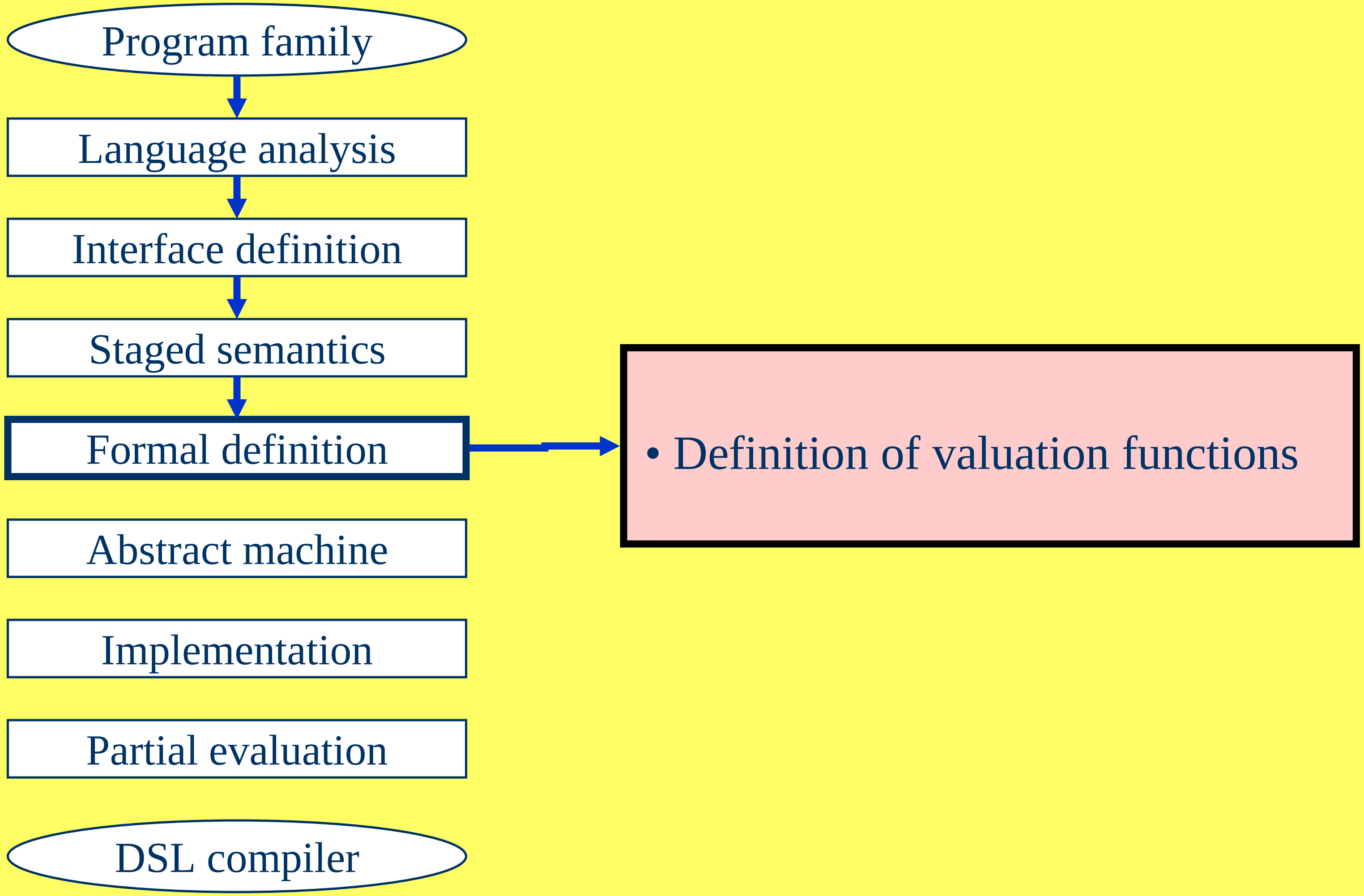
Sprint: An Overview



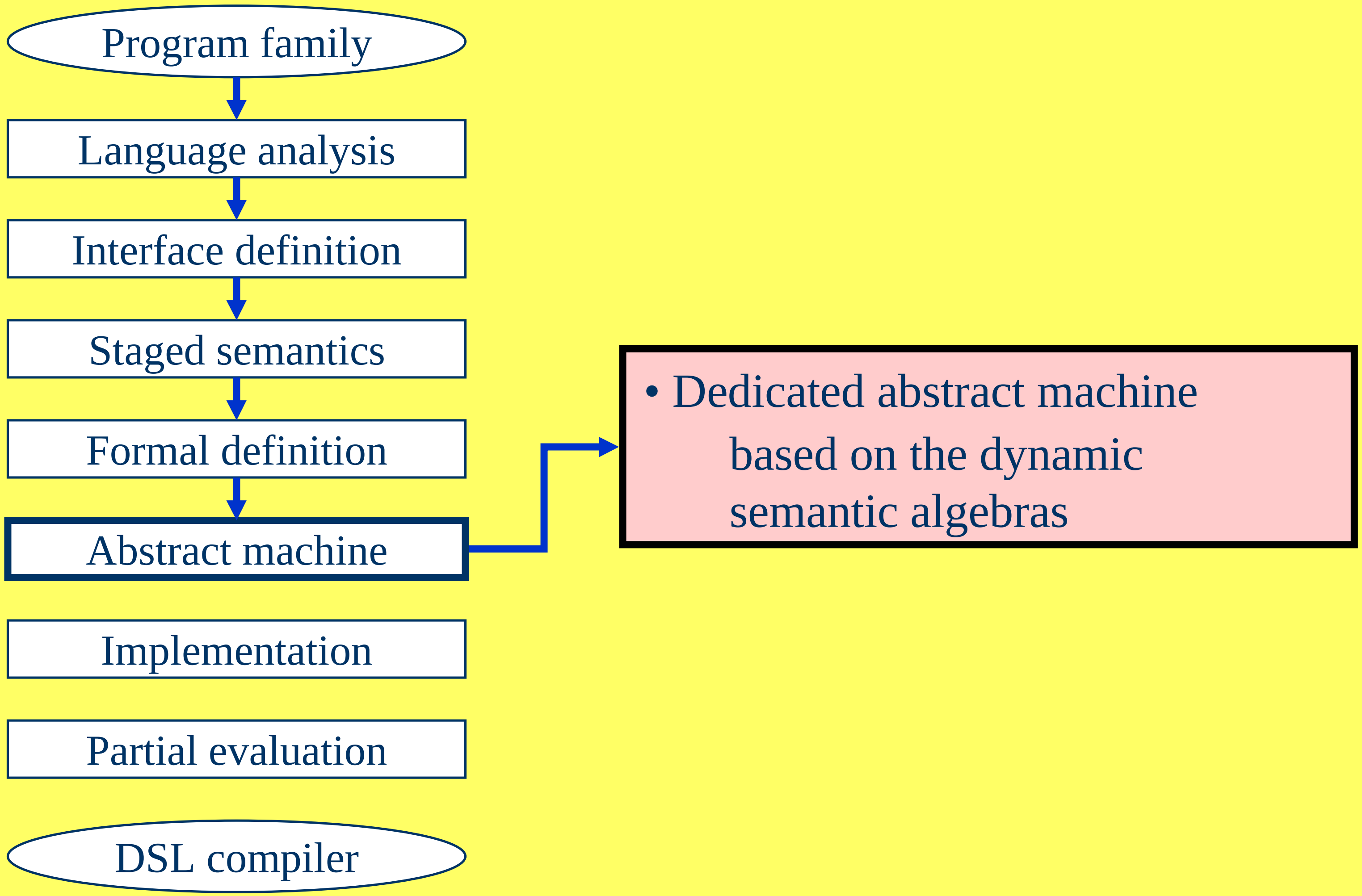
Sprint: An Overview



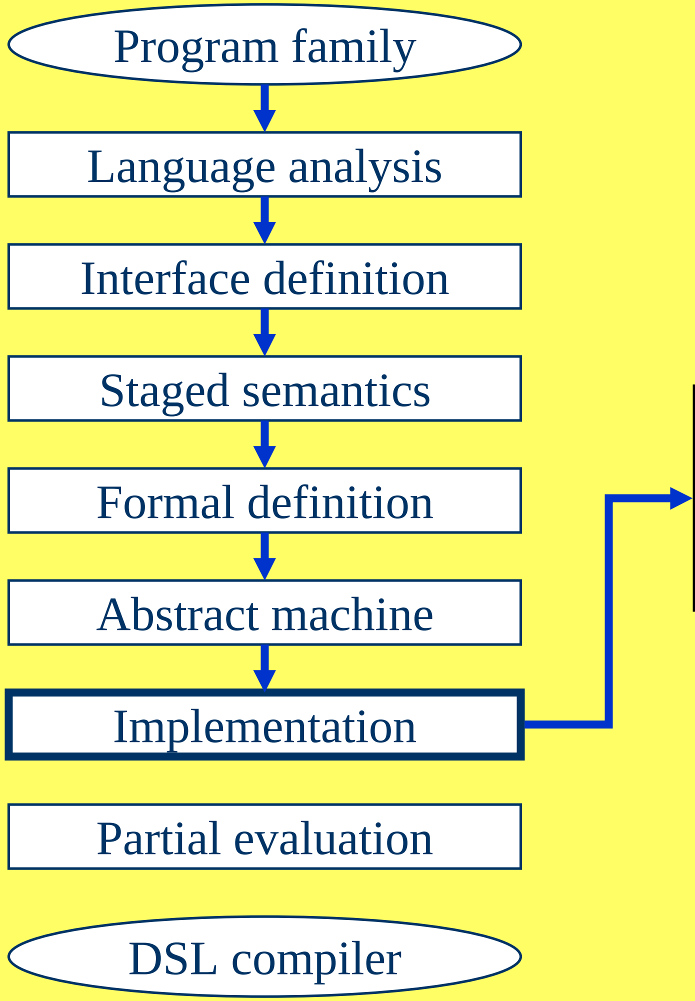
Sprint: An Overview



Sprint: An Overview

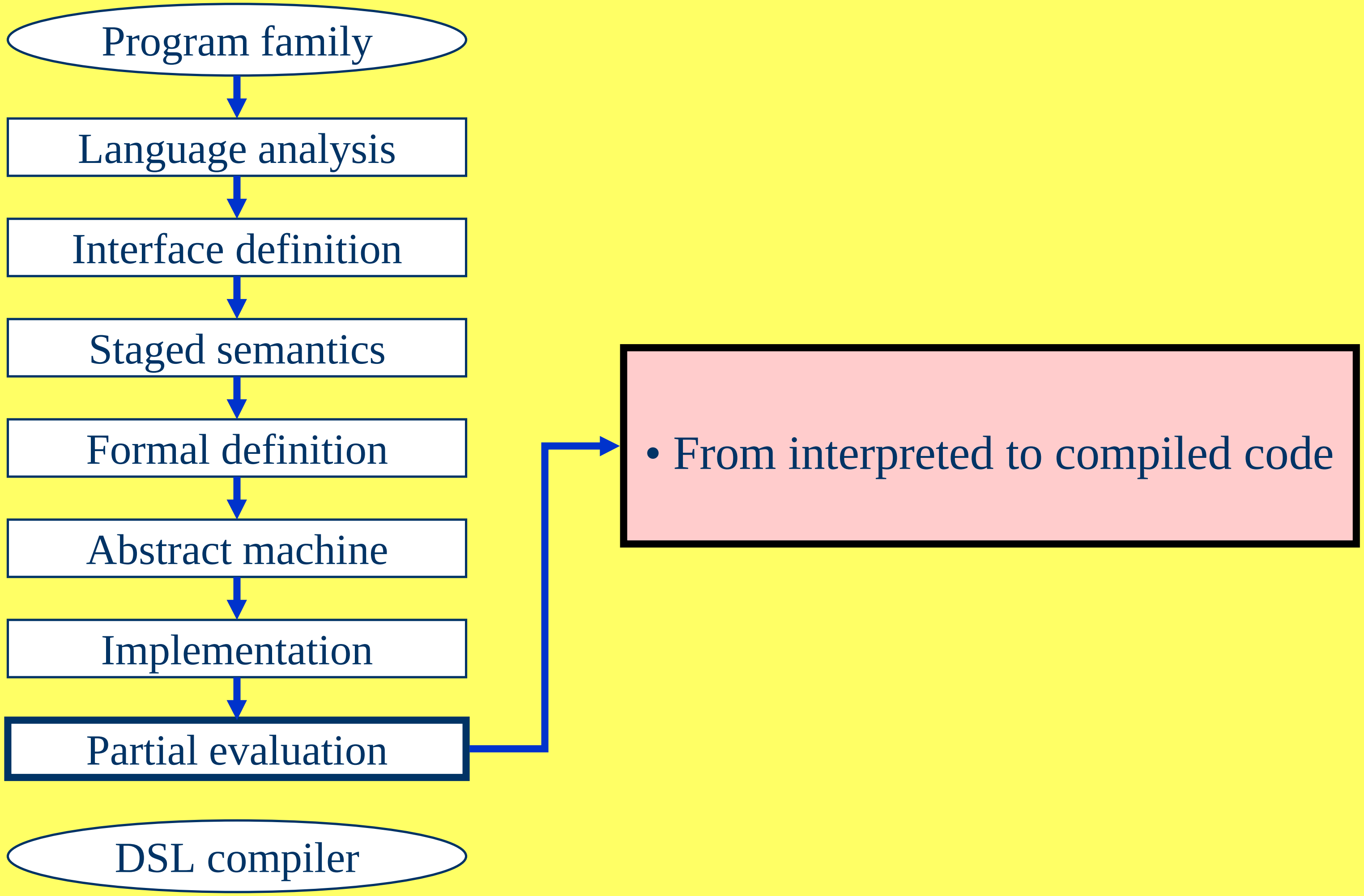


Sprint: An Overview

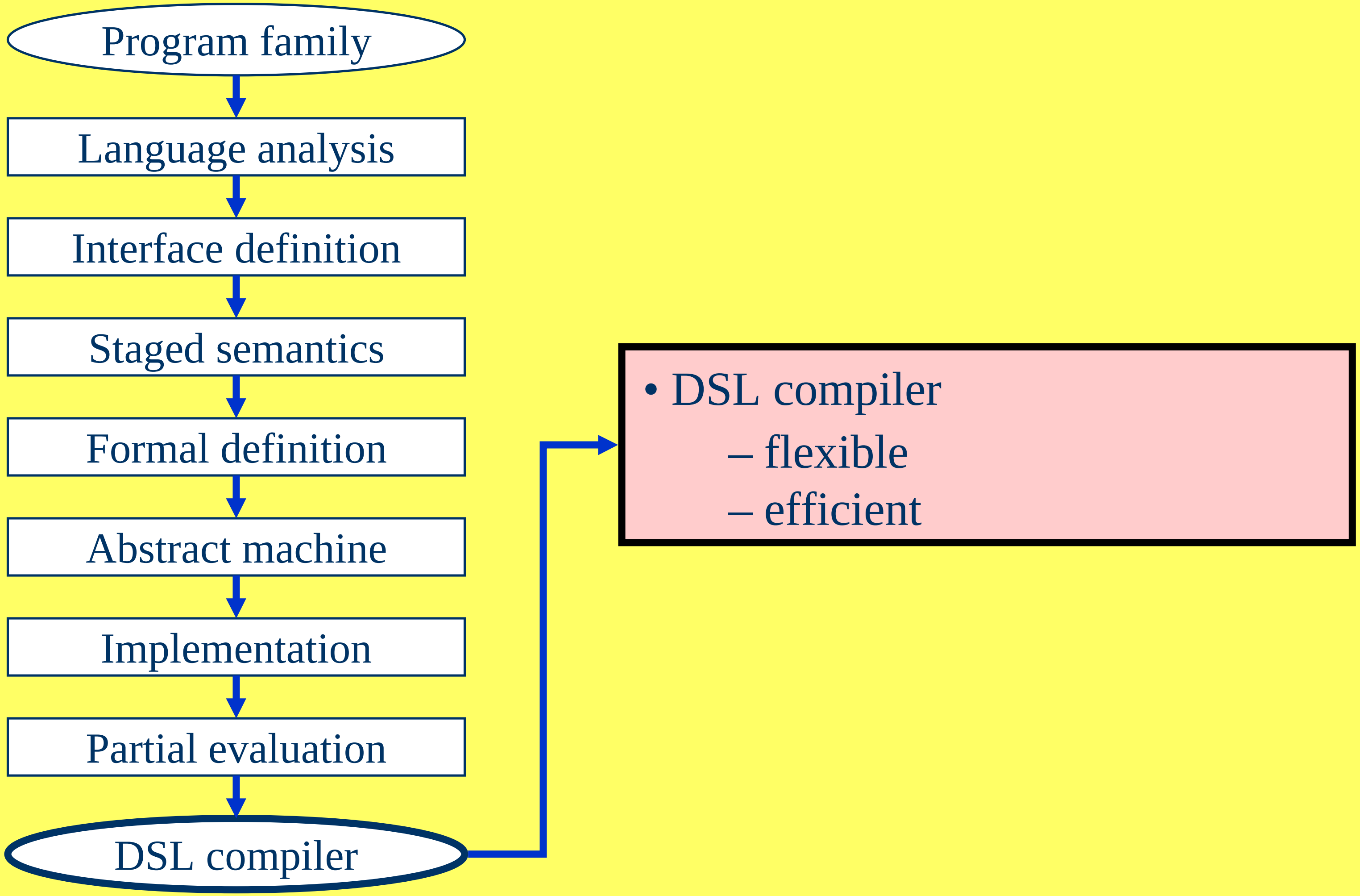


- Implementation of
  - abstract machine: library
  - valuat. functions: interpreter

Sprint: An Overview



Sprint: An Overview



Working Example: E-Mail Processing (1)

- Automatic treatment of incoming messages:
  - Dispatch mail to people or folders.
  - Filter out spam.
  - Automatic reply when absent.
  - Shell escape for specific treatments.
- Safety properties:
  - e.g., no loss of messages.

Working Example: E-Mail Processing (2)

- Program family:
  - Analysis of e-mail and decision making.
- Domain knowledge.
- Re-use opportunities.
- GPL $\Rightarrow$ no safety properties.

➡ Development of a DSL

- Inspired by `mh/slocal`, Unix mail delivery tool.

Language Analysis (1)

Program family

Interface definition

- Based on domain knowledge:
  - Technical literature and domain experts.
  - Existing programs.
  - Common patterns and variations.
  - Current and future requirements.
- Conducted using methodologies such as:
  - Domain analysis.
  - Commonality analysis.

Language Analysis (2)

Program family

Interface definition

- Language requirements.
  - Functionalities.
    - *Actions: copy, move, delete, forward, reply to a message.*
    - *Conditions: match message fields against string patterns.*
  - Language constraints (safety, security...).
    - *No loss or duplication of messages.*
    - *No loop when running or forwarding messages.*
  - Implementation constraints (resource bounds...).

Language Analysis (3)

Program family → Interface definition

- Objects and operations (building blocks).
  - *Messages: extract header fields, create messages...*
  - *Folders: add a message to a folder*
  - *Hierarchy of folders: associate a file to a folder path*
  - *Files of folders: read, write*
  - *Streams: inbound/outbound messages, pipe to shell*

Language Analysis (4)

Program family

Interface definition

- Elements of design.
  - Language paradigm and level.
    - *Hypothesis: shell programmers $\Rightarrow$ imperative like shell*
  - Terminology and notations.
    - *Shell notations for regular expressions*

Interface Definitions (1)

Language analysis

Staged semantics

- Based on a denotational framework.
- Semantic algebras (signatures).
  - *Domain: Message*
  - *Operations:*
    - *new-msg : Message*
    - *get-field : FieldName* $\rightarrow$ *Message* $\rightarrow$ *String*
    - ...
  - *Domains: InStream, OutStream*
  - *Operations:*
    - *next-msg : InStream* $\rightarrow$ *(Message* $\times$ *InStream)*
    - *send-msg : Message* $\rightarrow$ *OutStream* $\rightarrow$ *OutStream*

Language analysis

Interface Definitions (2)

Staged semantics

- Abstract syntax (kernel).

$B \in BoolExpr$
$C \in Command$
$F \in FolderPath$
$S \in String$

```
C = C1 ; C2
  | if B then C1 else C2
  | skip
  | delete
  | copy F
  | forward Sto
  | reply Sbody
  | pipe Scmd
```

```
B = match Sfield Spattern
  | not B
  | B1 and B2
  | B1 or B2
```

- Concrete syntax (graphic interface…).

```
move F ≡ copy F ; delete
if B then C ≡ if B then C else skip
```

IRISA

Interface Definitions (3)

Language analysis

Staged semantics

- Example:

```
if match “Subject” “DSL” then
    forward “jake”;
    move Research.Lang.DSL
else
if match “From” “hotmail.com” then
    reply “Leave me alone!”;
    delete
else
if match “Subject” “seminar” then
    pipe “agenda --stdin”;
    delete
```

Interface definitions

Formal definition

Staged Semantics (1)

- Separate static and dynamic semantics.

	Static	**Dynamic**
GPL	Actions performed by the compiler	Computations depending on input data
Concept	Determine member of program family	Produce answer for a family member
Implementation	Configure generic software	Execute customized software

➡ Reason about genericity: predict/control customisation.

Interface definitions

Formal definition

Staged Semantics (2)

- Initial staging constraints
  - *Static: DSL program, folder hierarchy, user's name*
  - *Dynamic: inbound messages*
- Staging of the semantic algebras
  - *Static: operations on folder hierarchy*
  - *Dynamic: streams and operations on streams*
- Staging of the language constraints
  - *Static: no loop (syntactic), no lost or duplicated message*
  - *Dynamic: no endless forwarding*

Staged semantics

Abstract machine

Formal Definition (1)

- Determine semantic arguments.
  - *Folder hierarchy, message being treated, streams...*
- Stage the semantic arguments.
  - *Static: folder hierarchy, user's name*
  - *Dynamic: message, folder files, streams, current date*
- Stage control.
  - Possibly introduce dynamic control combinators
    - *cond: for dynamic conditionals*

Formal Definition (2)

Staged semantics

Abstract machine

- Valuation functions.

$\boldsymbol{C} : Command \longrightarrow StaticState \longrightarrow DynamicState \longrightarrow DynamicState$

$\boldsymbol{C}[[\texttt{copy } F]]\ \rho\ \sigma =$
$\quad \text{let } \nu = \textit{get-filename}\ (\boldsymbol{F}[[F]])\ \rho_{\text{folder-hierarchy}}$
$\quad\quad \varphi = \textit{add-msg}\ (\textit{set-field}\ \text{“}\texttt{Delivery-Date}\text{”}\ \sigma_{\text{date}}\ \sigma_{\text{message}})$
$\quad\quad\quad (\textit{read-folder}\ \nu\ \sigma_{\text{folder-files}})$
$\quad \text{in } [\text{folder-files} \mapsto \textit{write-folder}\ \nu\ \varphi\ \sigma_{\text{folder-files}}]\ \sigma$

$\boldsymbol{B} : BoolExpr \longrightarrow DynamicState \longrightarrow DynamicState$

$\boldsymbol{B}[[\texttt{match } S_1\ S_2]]\ \sigma = \textit{match}\ (\textit{get-field}\ (\boldsymbol{S}[[S_1]])\ \sigma_{\text{message}})\ (\boldsymbol{S}[[S_2]])$

Abstract Machine (1)

Formal definition

Implementation

- A model of dynamic computations.
- Key to derive realistic implementation.
- Possibly shared between several DSLs.

Abstract Machine (2)

Formal definition

Implementation

- Single-threadedness:
  - Globalization of dynamic semantic arguments
    - *Globalize message being treated, folder files, streams...*
- Abstract machine entities (registers…)
    - *Dedicated register for message being composed (only one at a time)*

Formal definition

Implementation

Abstract Machine (3)

- Semantic definitions.

$$\boldsymbol{C} : Command \longrightarrow StaticState \longrightarrow AbsMachState \longrightarrow AbsMachState$$

$\boldsymbol{C}$ [[`copy` F]] ρ σ =

let ν = get-filename ($\boldsymbol{F}$ [[F]]) $\rho_{\text{folder-hierarchy}}$

in ((write-folder ν) ∘

(add-msg) ∘

(set-field$_i$ "`Delivery-Date`" σ_{date}) ∘

(read-folder ν)) σ

Implementation (1)

Abstract machine

Partial evaluation

<u>Valuation functions</u>

- Direct: interpretation.
  - Close to denotational definition.
  - Easy, flexible, slow.
  - Rapid prototyping.
  - Semantics-preserving extensions.
- Indirect: compilation.
  - Native code: expensive.
  - Abstract machine code: still expensive.

Abstract machine

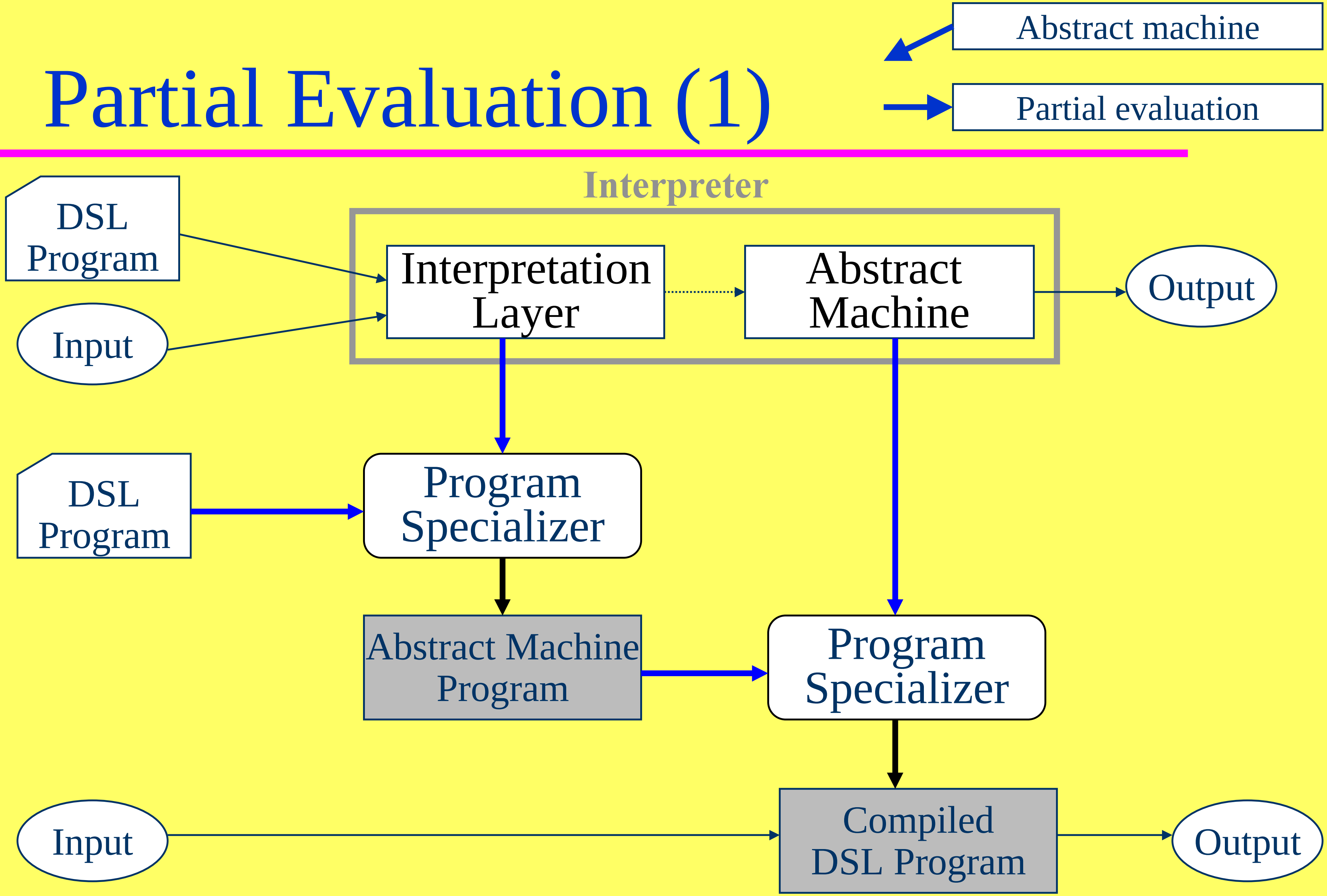
Partial evaluation

Implementation (2)

<u>Abstract machine</u>

- Little overhead:
  - Each instruction = coarse-grain operation.
  - Efficient compiler = efficient instruction.
- API: several implementations
  - *Folder as single file (Netscape, emacs)*
  - *Folder as directory, one file per mail (exmh)*

Partial Evaluation (1)



Implementation

DSL compiler

Partial Evaluation (2)

```
if match "Subject" "DSL" then
    forward "jake";
    ...
```

```
cond (match (get-field_i "Subject") "DSL")
  (new-msg;
   set-field_c "Date" (date)
   set-field_c "To" "jake";
   set-field_c "From" "bob";
   set-body_c  (msg-to-string_i);
   set-field_c "Subject"
      (concat "Fwd: " (get-field_i "Subject"));
   set-field_c "Resent-by"
      (concat "bob" (get-field_i "Resent-by"));
   send-msg;
   ...
```

Partial Evaluation (3)

Implementation → DSL compiler

- Experiments with Tempo, a specializer for C.
- GAL performance:
  - As fast as existing hand-coded C drivers.
- PLAN-P performance (*runtime specialization*):
  - PLAN-P *JIT* twice as fast as a Java *JIT*.
  - 100% of the throughput of hand-written C bridge.
  - 100% of the bandwidth.

Sprint: Assessment

- Based on well-studied ingredients.
- Careful structuring.
  - Design / definition / implementation.
  - Analyzability: source / abstract-machine level.
- Development cost.
  - Interpreter *vs* compiler.
  - Off-line or *JIT*.
- Maintenance.
  - Flexible, extensible.

Conclusion: A Revival

- DSL ≠ GPL: many things become possible.
- Dig up your old theories.
  - Paradigm.
  - (Denotational) semantics.
  - Implementation.
  - Verification.
- The programming language community can (must) play an important role.

More information

Prototypes

- DSLs: GAL, PLAN-P
- Specializer: Tempo

are

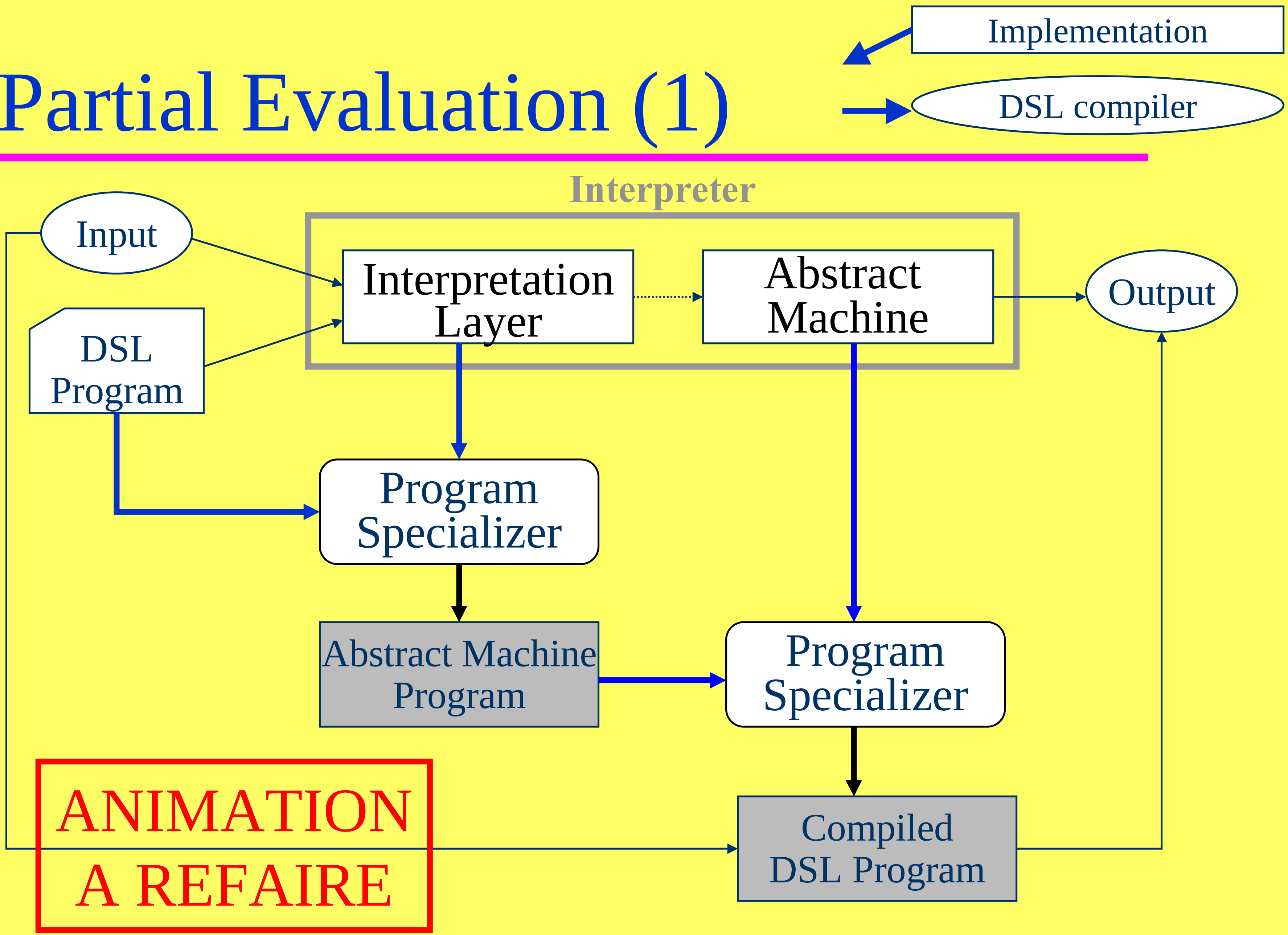
- described (papers),
- available (distribution).

at

`http://www.irisa.fr/compose`

Ce qui suit est du trash.

Partial Evaluation (1)



Trash

$\mathcal{C}$ [[`if` B `then` C_1 `else` C_2]] ρ = *cond* ($\mathcal{B}$ [[B]]) ($\mathcal{C}$ [[C_1]] ρ) ($\mathcal{C}$ [[C_2]] ρ)

$\mathcal{C}$ [[`pipe` S]] $\rho\ \sigma = [\text{cmd-stream} \mapsto \textit{pipe-msg}\ \sigma_{\text{message}}\ (\mathcal{S}\ [[S]])\ \sigma_{\text{cmd-stream}}]\ \sigma$

DSL Example (1)
`make`

A utility to maintain programs.

- Small, mainly declarative.
  - Expressive power: dependency updates.
  - Actions delegated to a shell.
- Domain abstractions:
  - File suffixes, implicit compilation rules.
- Verifications:
  - No cycles in dependencies.

DSL Example (2)
Shell

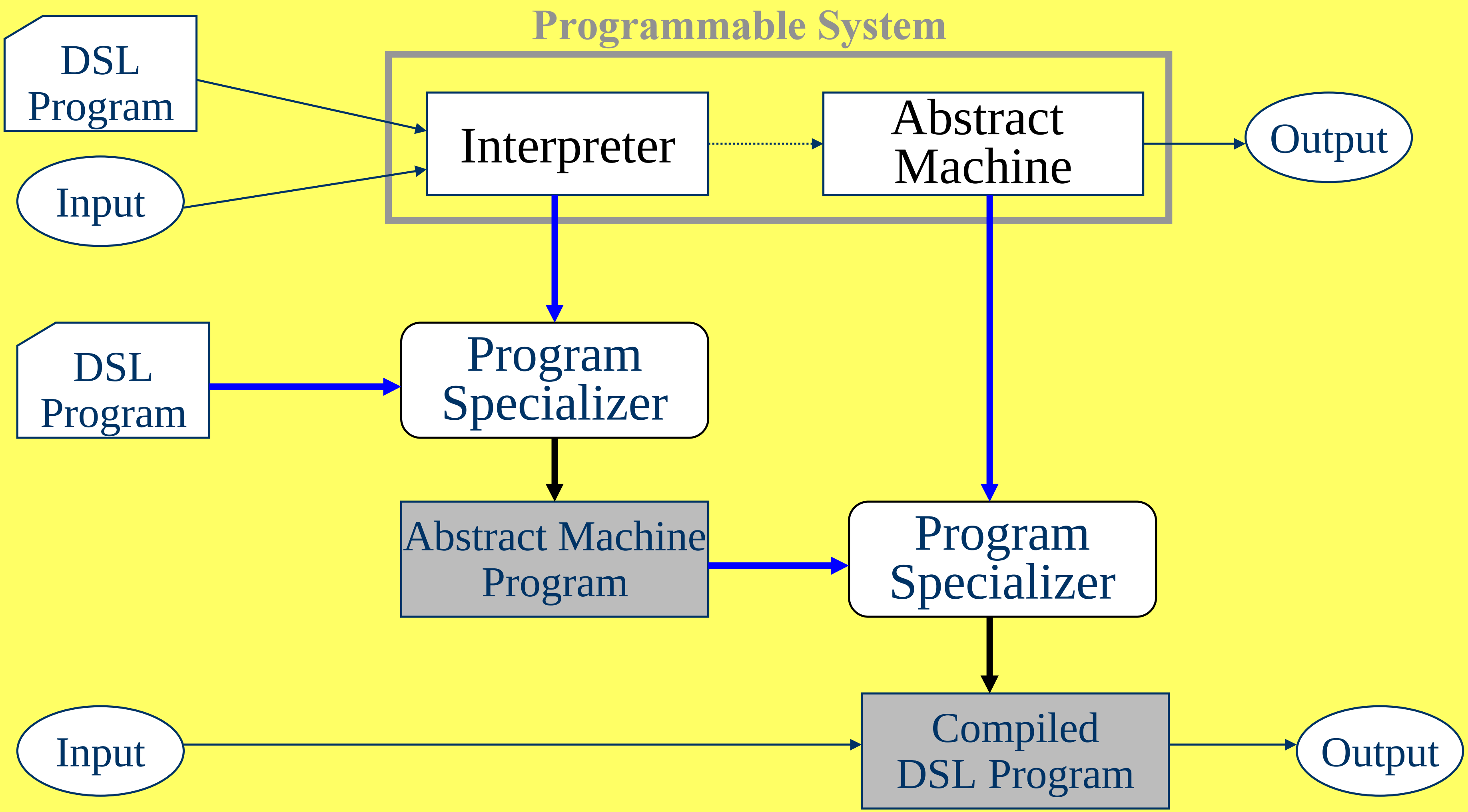
A command programming language.

- Domain abstractions:
  - stdin/stdout/stderr.
  - Command line facilities.
- Expressive power:
  - Run/control processes.
  - Some string manipulations.
- Interface to standard system libraries.

Current Approaches to Deal with Program Family (2)

- Patterns of programs: often unexploited
  - Readability
    - » **???**
  - Redundancy
    - » Development.
    - » Maintenance.

Adaptation Process



Our Methodology: An Overview

Problem family

Language analysis

Interface definition

Staged semantics

Formal definition

Abstract machine

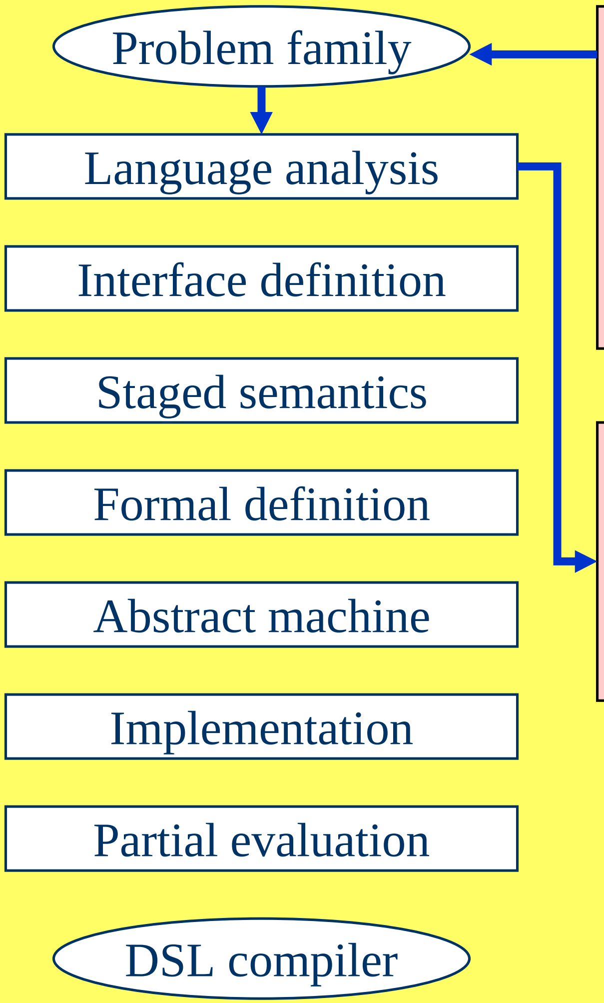
Implementation

Partial evaluation

DSL compiler

- Domain knowledge:
- Commonalities and variations
- Definition of the syntax of the DSL.
- Informal semantics relating
  - Syntactic constructs, and
- Splitting compile-time and run-time actions.
- Making explicit stages of configuration.
- Dedicated abstract machine based on dynamic semantic algebras
- Abstract machine implementation(s): library.
- Valuation function implementation: interpreter.
- From interpreted to compiled code.

Our Methodology: An Overview



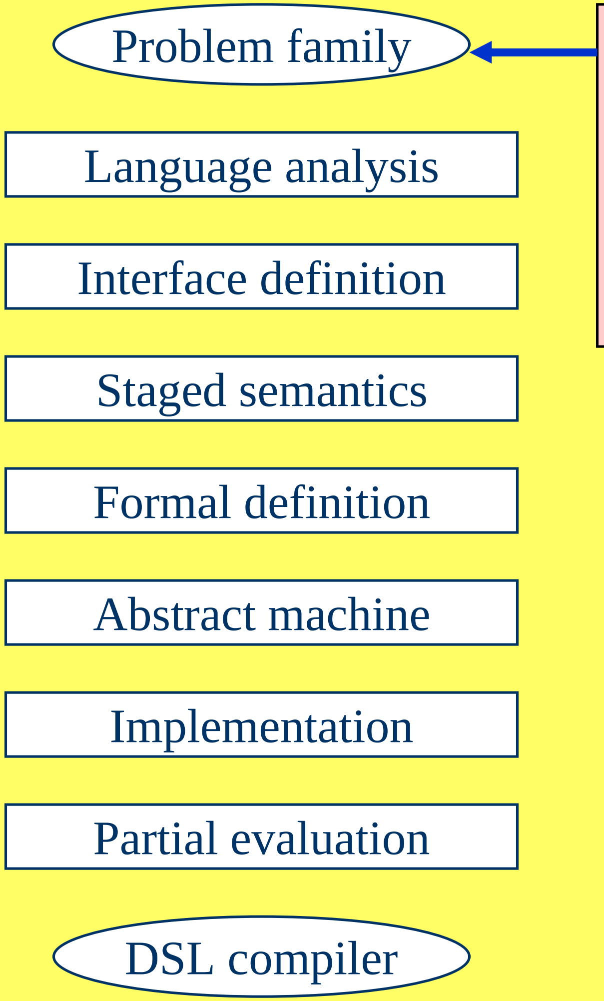
Input: Domain Knowledge.

- Technical literature.
- Existing programs.
- Current and future requirements.
- Common patterns and variations.

Output:

- Description of objects and operations
- Language requirements.
- Elements of design.

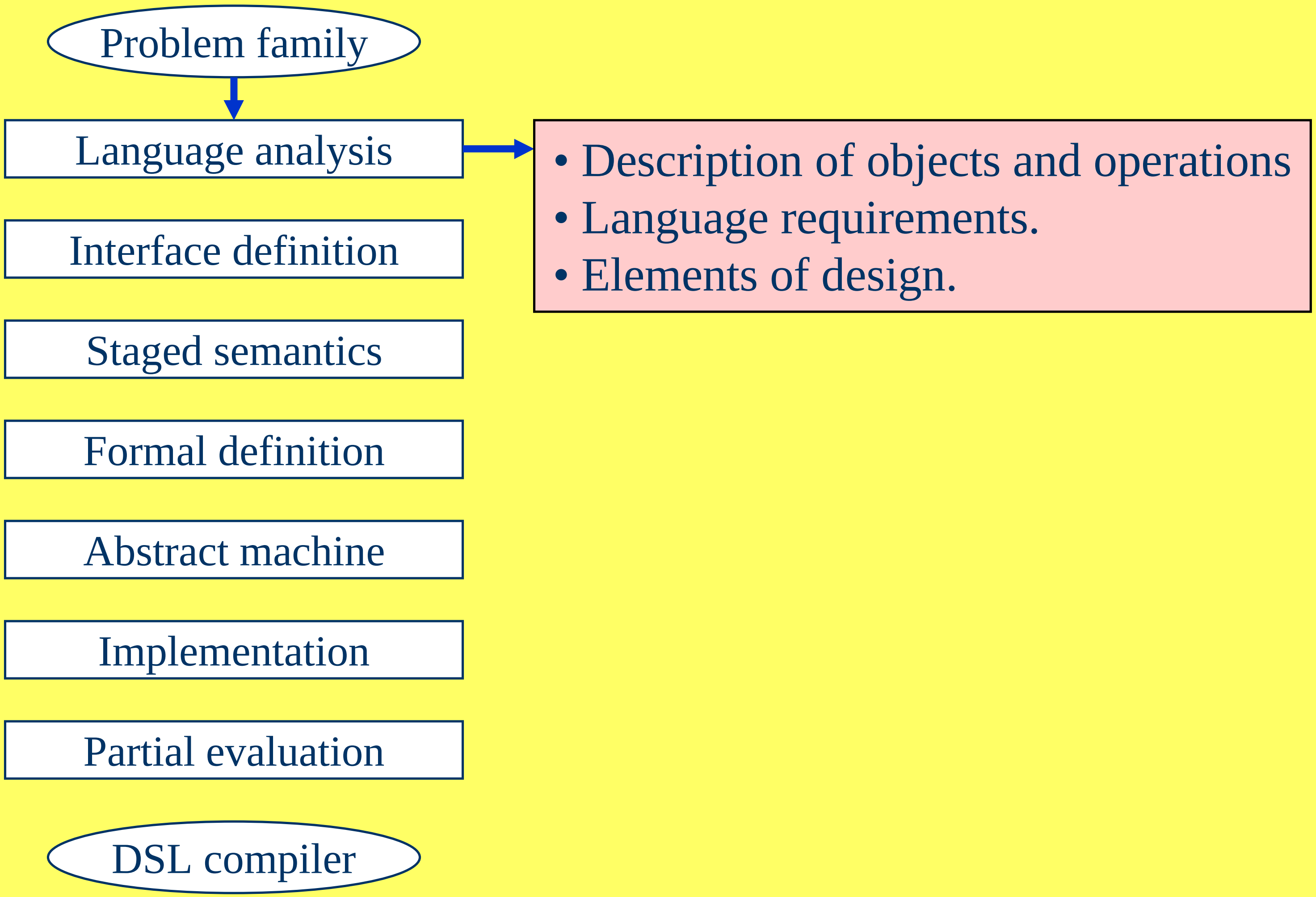
Our Methodology: An Overview



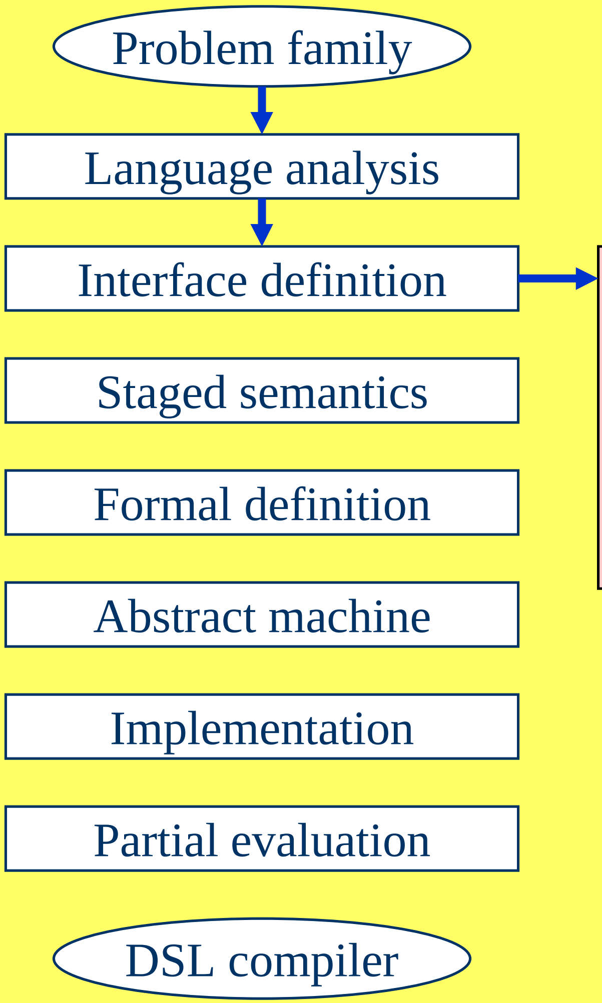
Domain knowledge:

- Technical literature.
- Existing programs.
- Current and future requirements.
- Common patterns and variations.

Our Methodology: An Overview

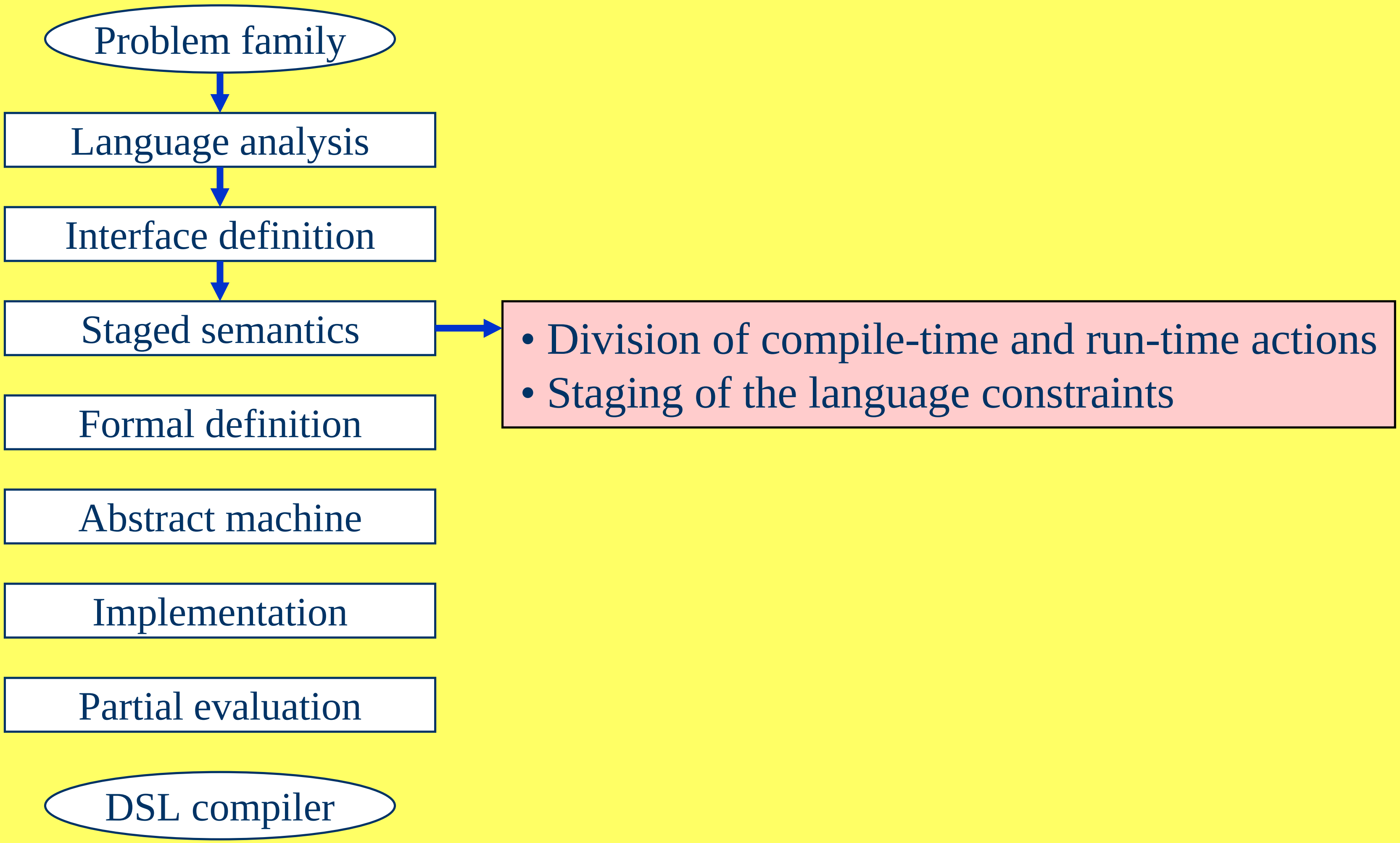


Our Methodology: An Overview

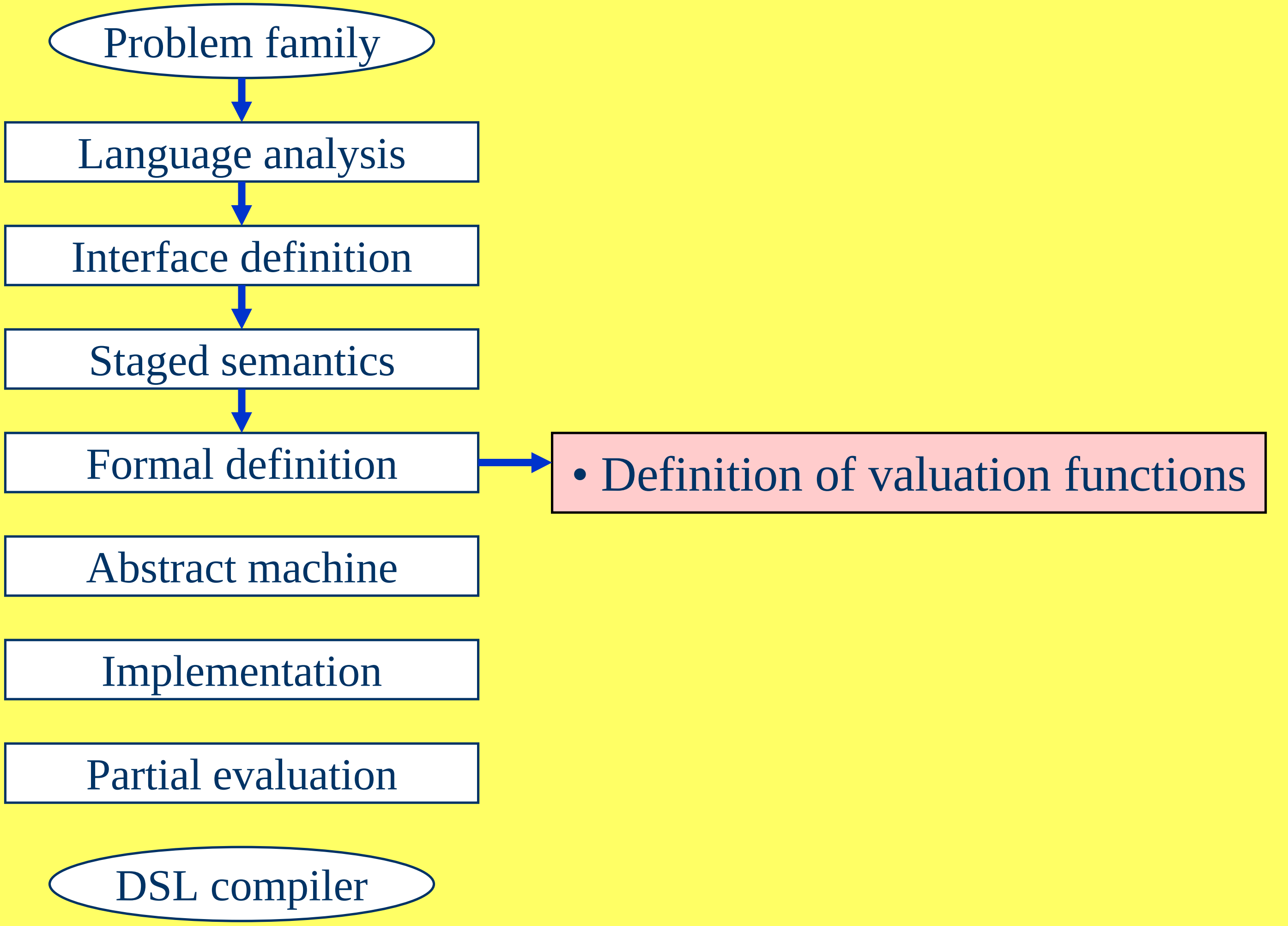


- Definition of the syntax of the DSL
- Informal semantics relating
  - syntactic constructs
  - the objects and operations
- Signature of semantics algebras

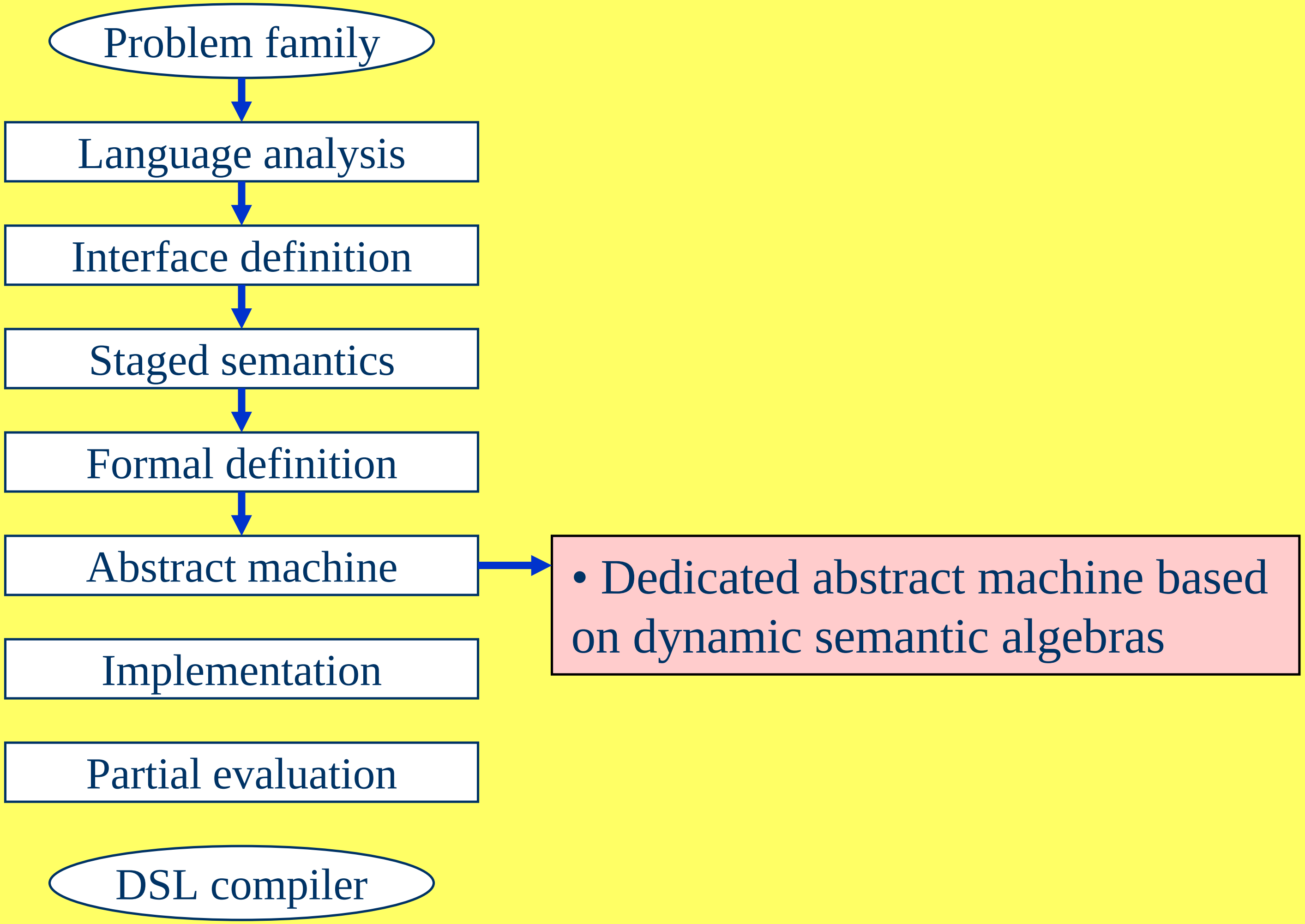
Our Methodology: An Overview



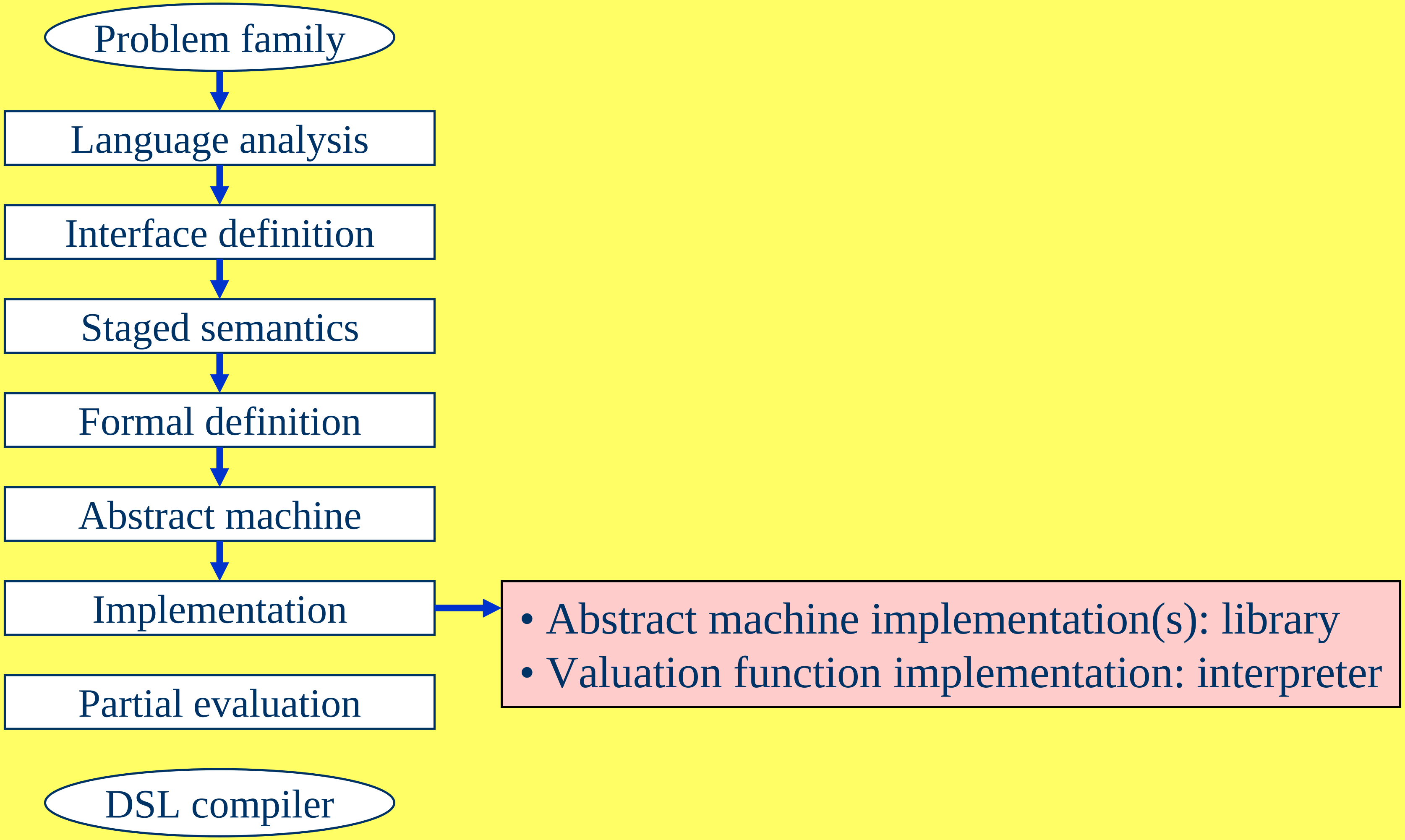
Our Methodology: An Overview



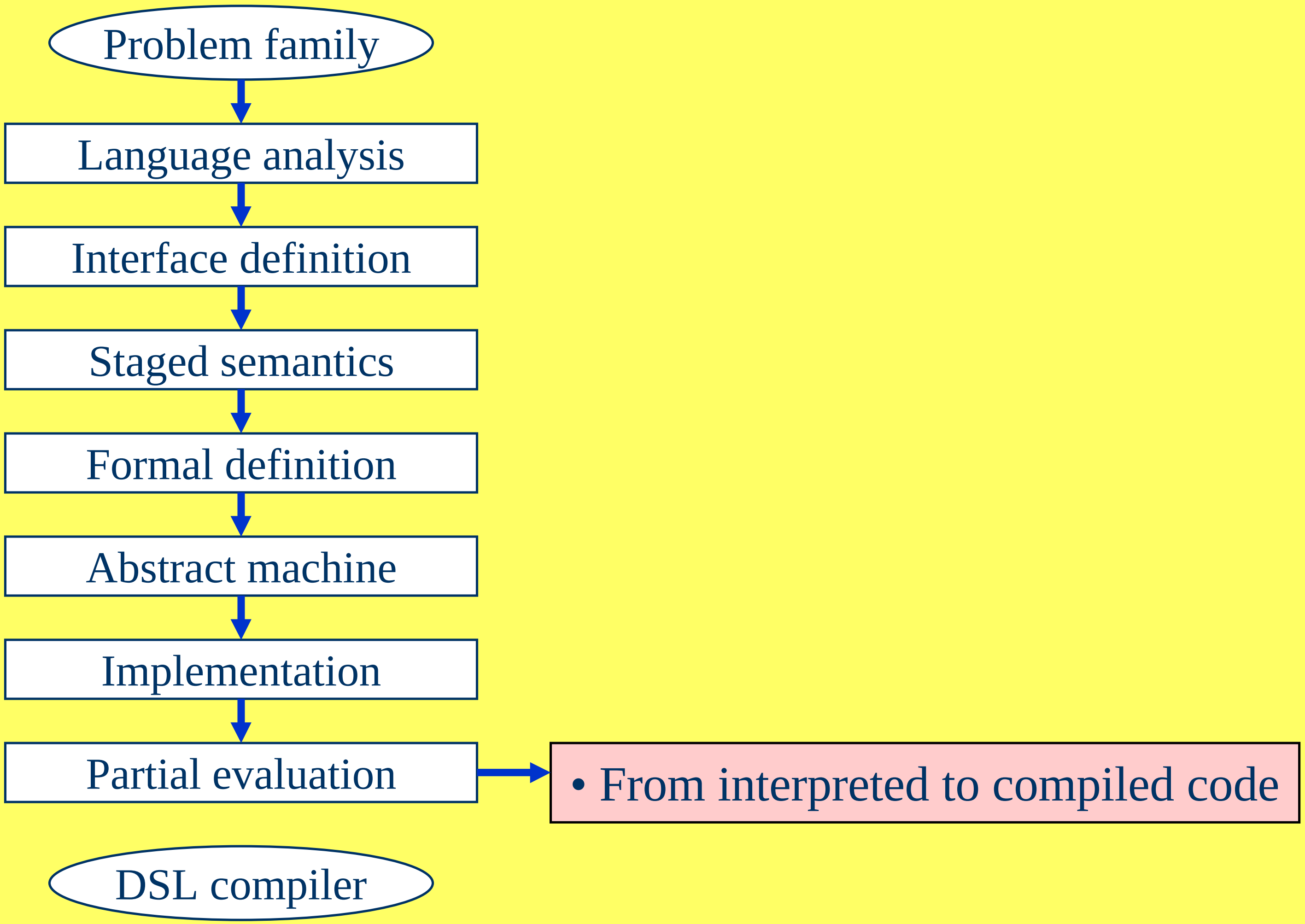
Our Methodology: An Overview



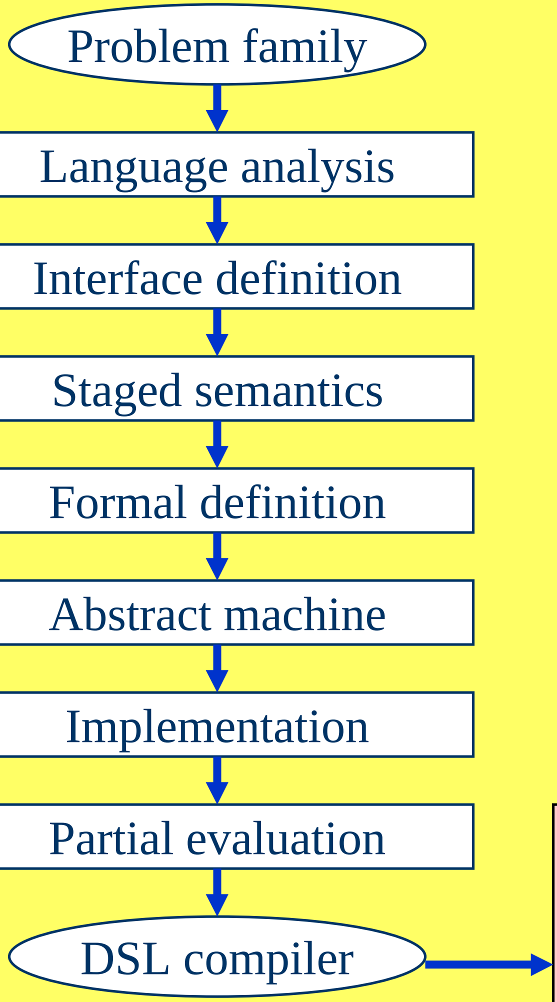
Our Methodology: An Overview



Our Methodology: An Overview



Our Methodology: An Overview



- DSL programming environment:
  - efficient
  - flexible