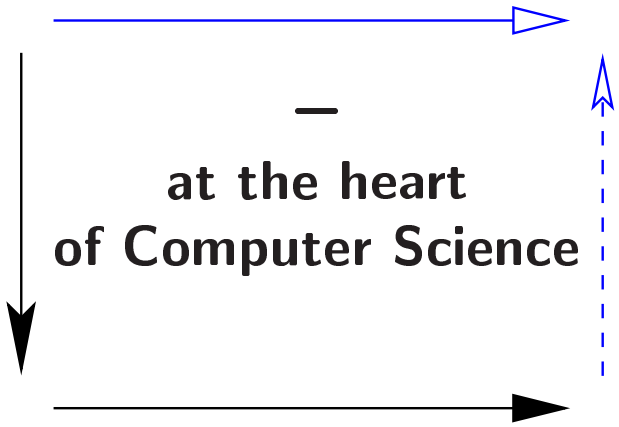
Abstract

In this talk, I will try to give an introduction to my views on language design, and why I think it should be a core topic of computer science.

The first part outlines and motivates the major role languages should play in computer science (in brief: languages are interfaces - between CS and other disciplines, and between users and the systems we offer them). The reasons why, so far, languages have not quite managed to take on that role, are largely historical and are discussed briefly. The discussion leads directly into the second part, which focuses on some of the changes that are necessary to remedy the current situation. This, in turn, sets the context for most of the things I hope to be working on over the next few years.

Language Design



Claus Reinke
Computing Laboratory, University of Kent

From the TCS www home page:

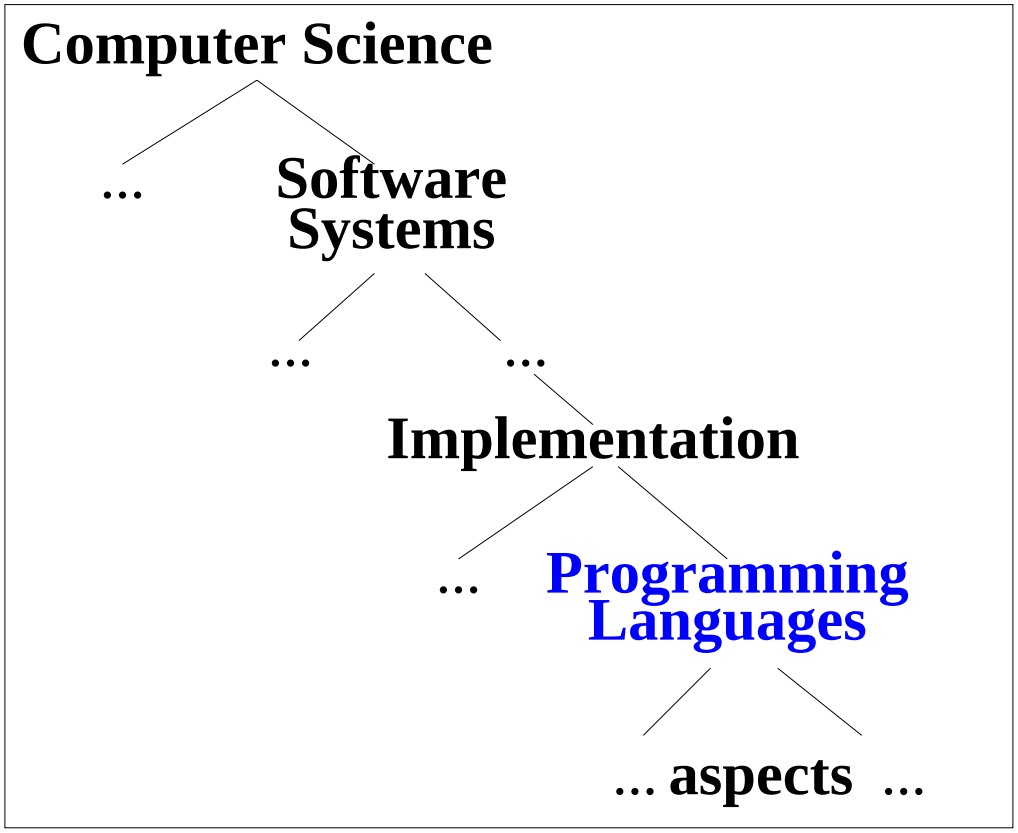
⋮

Programming languages are central to computer science, since they are the medium in which all software systems are ultimately written.
To make these systems reliable, extensible, modular and efficient an understanding of all aspects of programming languages is crucial. The study of programming languages covers a wide range of topics in computer science, including,

- the theory of programming language semantics and program verification using logic;
- the pragmatics of designing and using existing programming systems of various different sorts;
- understanding the foundations of programming;
- designing future programming systems;
- implementation aspects such as compile-time analysis and garbage collection.

⋮

Central to CS?



Well, yes, if you are interested

in *software systems*,

. . . ultimately . . .

in their *implementations*,

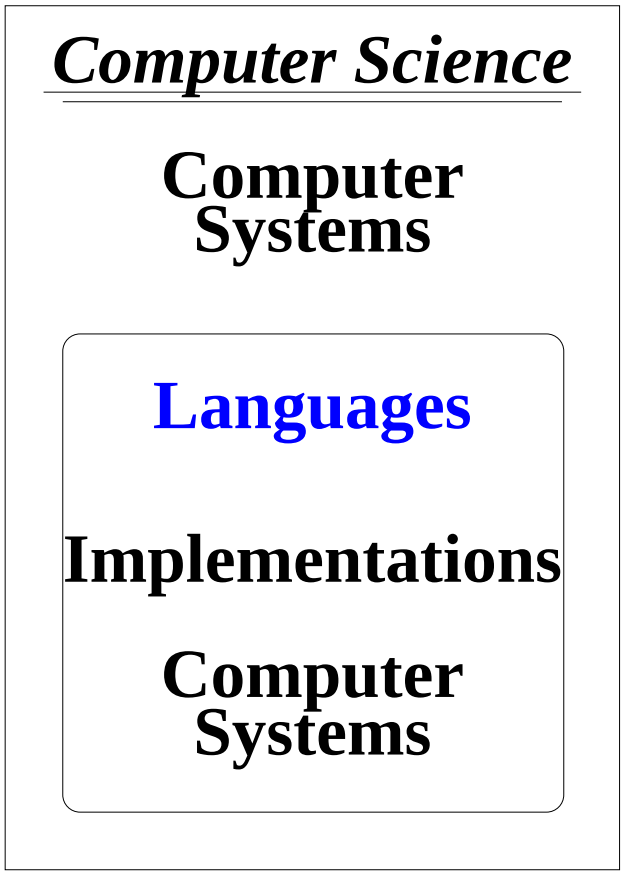
and in the *medium*

for producing those..

as well as its *aspects*

This view of programming languages is biased by CS-internal interests!

A second look at the picture



In Computer Science,

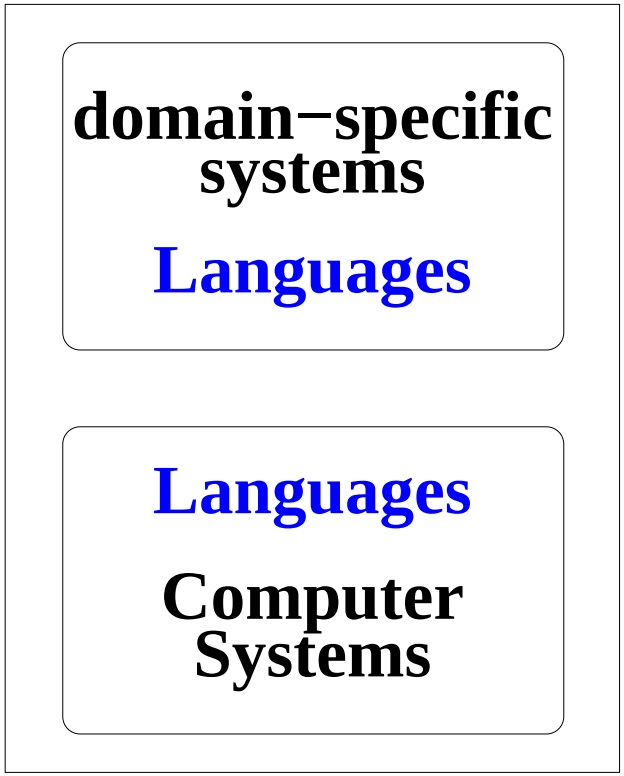
we are interested in computer systems (hard- and software). *Languages of various kinds are used to describe these systems*.

Languages with implementations we tend to call programming languages because they allow us to make use of/to program other computer systems.

If we specify a system in terms of a programming language, we get an implementation of that system on another.

(Declarative) Languages as interfaces

Languages are used to describe systems

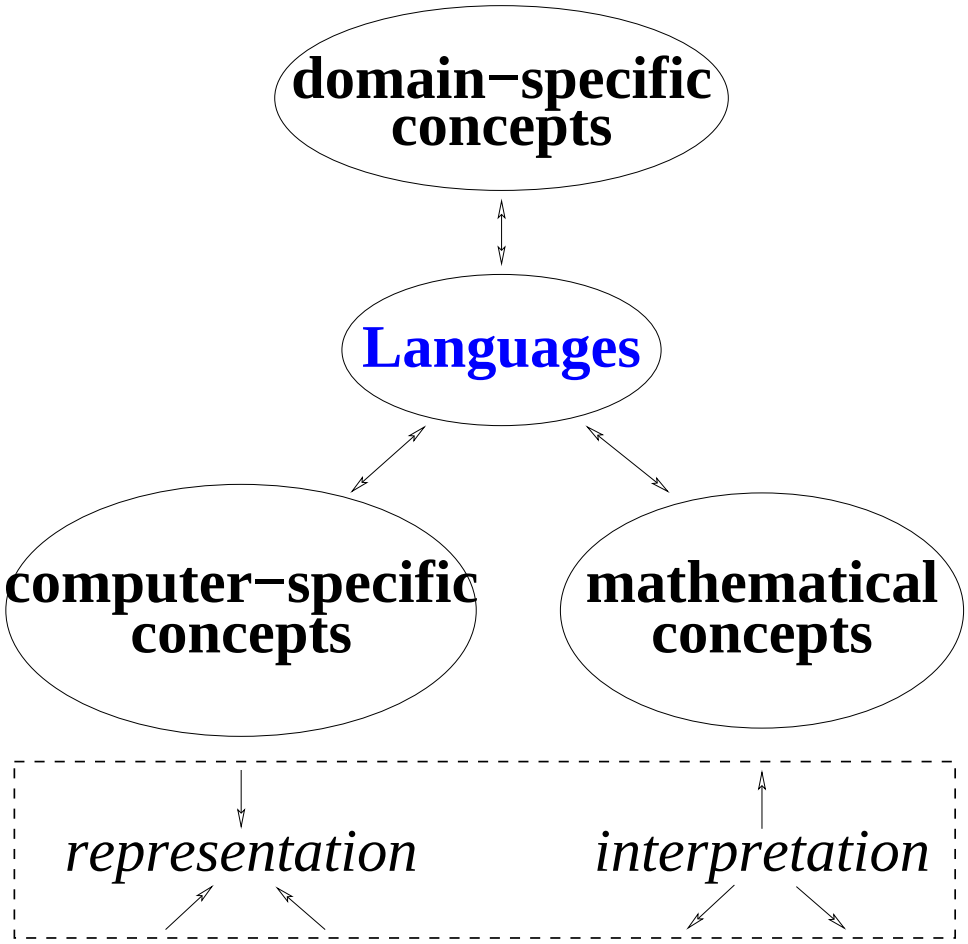


Computer scientists use languages to describe computer systems. They also use computer systems to implement languages.

It is just an accident that languages tend to be seen as internal to computer science – computer scientists have been their own customers for too long.

Instead, we should see *languages as interfaces between systems specific to an application domain and computer systems*.

Understanding languages – another accident



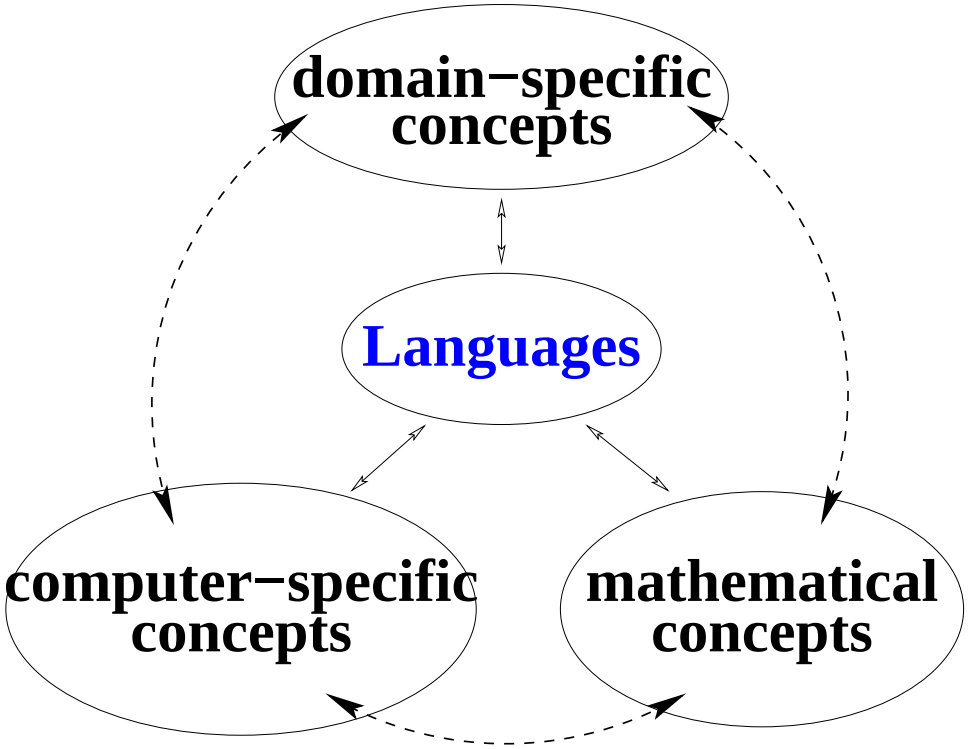
The idea of languages as interfaces builds on *common representations for concepts in different domains*.

Understanding languages is not always necessary, but typically builds on interpretation in some domain.

Early on, CS interpreted languages internally, in terms of the systems they represent (meta-circular interpreters).

Maths is one useful target domain for external interpretations. It is *not* the only one, it is *not always* appropriate.

Language-mediated isomorphisms



Ideally, *representations and interpretations should be isomorphic*.

We could then move freely between corresponding concepts in different domains (none of which needs to be more fundamental than the others).

Different interpretations, implied by different backgrounds, would then be known to be equivalent.

⇒ *Languages not only mediate, they also isolate – they act as interfaces.*

It is good practice in modular systems design to combine isolated components via well-defined interfaces. Unfortunately, CS has a strong tendency to incorporate more and more of maths and of other domains, creating unmanageable complexity, e.g., for software engineers.

Computing systems, a micro-history

2. machine abstraction *(soft everything)*
1. control abstraction, data abstraction *(soft structures, fixed machine)*
0. stored-program machines *(soft data, soft control, fixed structures)*
   control sequences/graphs
   data sequences/graphs
-1. reconfigurable machines *(soft data, soft parameters, hard control)*
-2. build-your-own machines *(soft data, hard control)*
-3. one-job-one-machine *(hard everything)*

The ***software aspect*** went from *none at all*, to *direct representations* of machine aspects, to control and data *structures indirectly representable* on the machine, to *virtual machines* representable on the real machine!

The confusion of "hard" and "real"

At this point in our story, a very unfortunate factor began to dominate the further development: ***the confusion of "hard" and "real"***.

Today, we think of the *"hard"-ware as just another virtual machine*, built on top of further abstractions, down to physical laws. Virtual machines in soft- and hardware can be just as real as hardware alone, and much easier to adapt to specific application domains.

But hardware used to be substantial, expensive, and rare. Software was just a cheap and insubstantial add-on to the "real" thing.

Interactive systems (virtual machines) became associated with *interpreters* (*slow*). *Fast execution* became associated with *compilers* and *machine-level executables*.

Virtual machines became just a means to specify *intermediate languages*, to be *translated away* into code that could run on the "real" machine.

The decline and fall of abstraction?

Orientation downwards, towards lower, more "real" levels of abstraction:

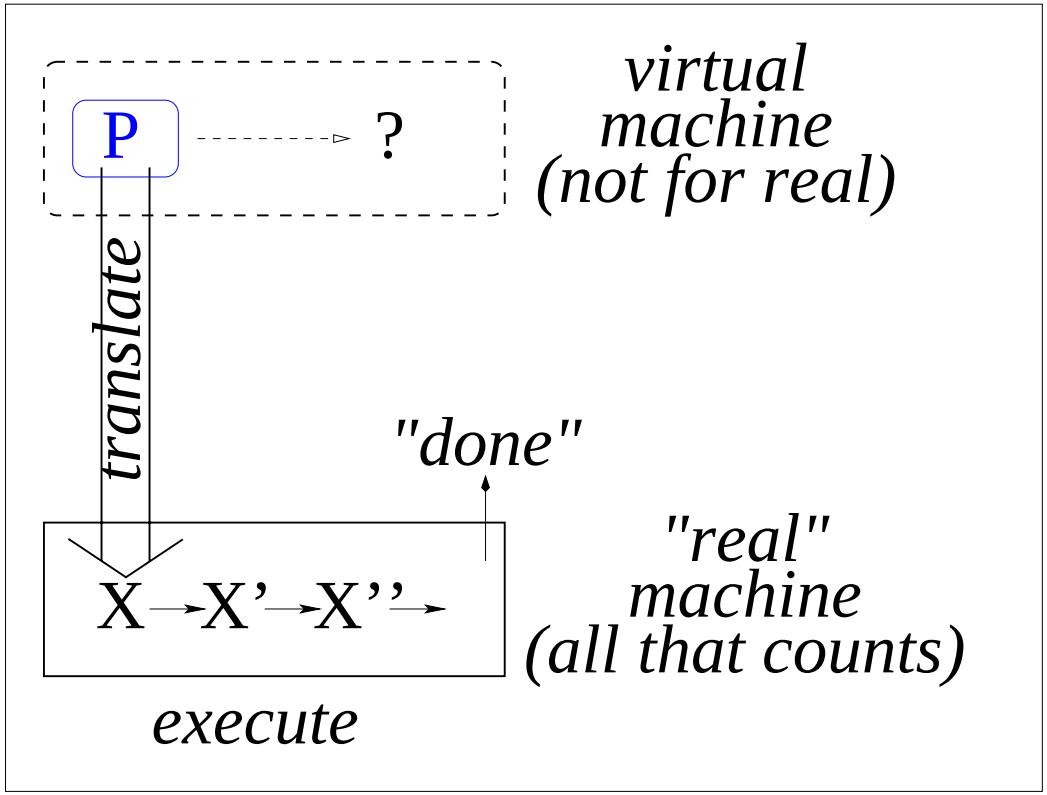
Instead of interacting directly with virtual machines and their languages, these abstractions were translated away ($\rightarrow$ compiler backends).

Instead of directly supporting and implementing domain-specific notations, software engineering advocated the complete recoding and translation of domain knowledge into software architectures and programming languages that remained foreign to the domain experts.

Language design was described as compiler design [Hoare73, Wirth74]!

Only input programs were written in high-level notation. Program results amounted to status codes, execution took place at machine level, and the responsibility for intermediate interactions was shifted to the code.

Downwards only . . .



Against the trend

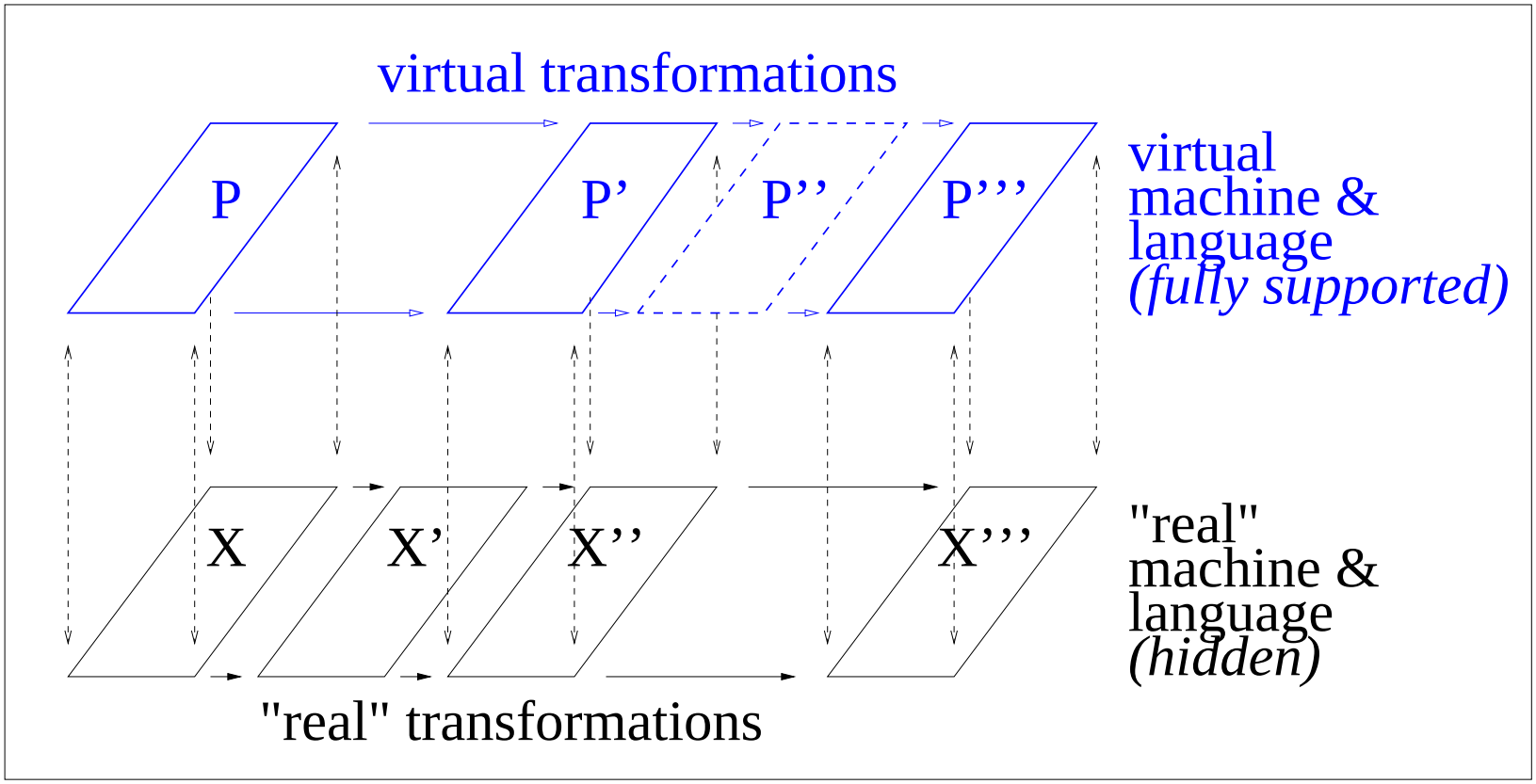
Established exceptions *(virtual machines for domain-specific languages)*

- Spreadsheets, WYSIWYG editors *(dynamic documents)*
- The desktop metaphor *(your office in your computer)*
- Game machines *(. . .)*
- Smalltalk *(everything is Smalltalk)*
- Petri nets *(program execution at graphical level)*
- Reduction systems *(high-level program execution)*

More recent developments

- Data visualisation *(how to get program results and outputs back to a more appropriate level of abstraction)*
- Extreme programming *(how to keep the software engineering process and its products in touch with customers)*

Two levels of abstraction



Beware of the one-sided arrow!

Whenever you see a computer scientist using a picture with uni-directional arrows, ask about the reverse direction (actually, that's a more general idea – see, e.g., invariants and conservation laws in physics).

- *In implementation terms*: the downward mapping is essential, but **without the inverse mapping, the abstraction-level breaks down**.
- *In theory terms*: **there is no absolute semantics!**.
  But attempts to establish two-way mappings between structures in different domains can reveal a need to modify languages *(→ semantic design principles)* or candidate models, *balancing out influences.*
- *In software engineering terms*: alienating customers from the product made for them complicates the development process and places an impossible burden of responsibilities and workload on software engineers! **Supporting domain-experts ≠ engineering applications!**

Implications for the study of language design

1. ***The central issue is to identify useful general concepts and structures***. Maths is not the only acceptable source of ideas (chemistry, biology, physics, economy, etc.), but formal representations are necessary.

2. (Partial) ***correspondences between related structures in different domains*** help to identify the common structures behind the different partial views.

3. ***There is room and need for a variety of domain-specific languages.*** The idea of a single, ultimate language might well be an illusion.

4. ***In spite of differences, languages should share common foundations.*** Many languages, but only few foundational calculi and concepts.

5. ***Programming calculi*** are an appropriate medium for the investigation of foundational questions in language design. A spartan form of languages, free from syntactical irrelevancies, but not representation-free!

Semantic principles for language design

1. define ***objects of interest*** *(semantic domains)*
2. define ***general services*** *(language framework)*
3. services should include ***abstraction*** *(names for known and unknown things)*
4. the rules for using names should be ***consistent*** *(no special cases)*
5. the general services should be ***complete*** *(uniform definition)*

Principles involved: *abstraction*, *correspondence*, *data type completeness*.
Design method: we *start with the minimal set of objects and general services* for our purpose, then we *close the language under the principles*.

Other principles are known, e.g., we prefer *orthogonal* sets of general services and, as a general guideline, we strive for "*simplicity through generality*".

Towards an algebra of languages?

$L = \mu X.B + S(X),\ \ S \supseteq Abs,\ \ Abs \supseteq Decl + Param,\ \ Decl \leftrightarrow Param$

0. Assume $L \neq \emptyset$ (to bootstrap) $\Rightarrow L \sim \lambda$-calculus + let *(biased!)*
1. $B =$ some data types with cf operations $\Rightarrow L \sim$ functional language
2. $B =$ cs operations on implicit global state $\Rightarrow L \sim$ imperative language
3. $B =$ cf + cs operations $\Rightarrow L \sim$ imperative functional language
4. $B =$ cf, $S \supseteq$IO $\Rightarrow L \sim$ functional language with first-class i/o
5. $B =$ cf, $S \supseteq$records $\Rightarrow L \sim$ functional language with first-class modules

⋮

Surprisingly useful already, but needs some refinements.

Different kinds of transformations

$C[\ ], l[\ ], r[\ ]$ contexts, i.e., terms with a hole; $\vec{v}$ sequences of variables.

context-free $\forall C, \vec{v} :\ C[\ l[\ \vec{v}\ ]\ ] \to C[\ r[\ \vec{v}\ ]\ ]$

context-sensitive I $\forall \vec{v} :\ l[\ \vec{v}\ ] \to r[\ \vec{v}\ ]$

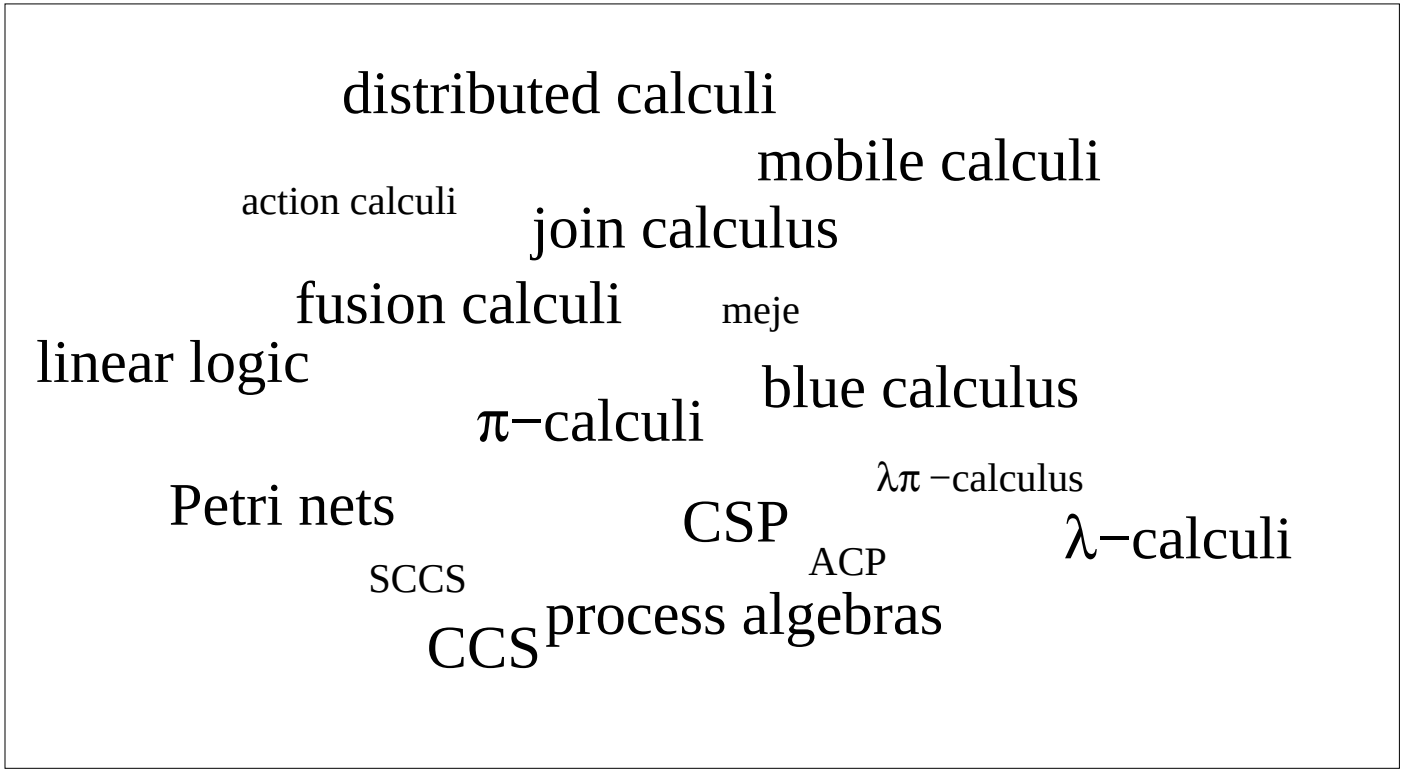
context-sensitive II $\forall \vec{v} :\ C_\kappa[\ l[\ \vec{v}\ ]\ ] \to C'_\kappa[\ r[\ \vec{v}\ ]\ ]$

The advantage of this second form is that even context-sensitive transformations cannot change everything – the kind κ of the contexts is fixed (structured contexts $\leftrightarrow$ structured data).

context-sensitive III *multiple interacting foci of transformation*

Instead of a single hole, corresponding to a single agent, contexts may contain other agents.

Foundations for non-sequential systems?



Towards functional Petri nets

$L = \mu X.B + S(X), \quad S \supseteq Abs, \quad Abs \supseteq Decl + Param, \quad Decl \leftrightarrow Param$

-1. B =functional language, S = Petri net structures

$\Rightarrow L \sim$ coloured Petri nets

Problem: "only" CPN – base language lacks composition operators!

0. B =Petri nets + ?

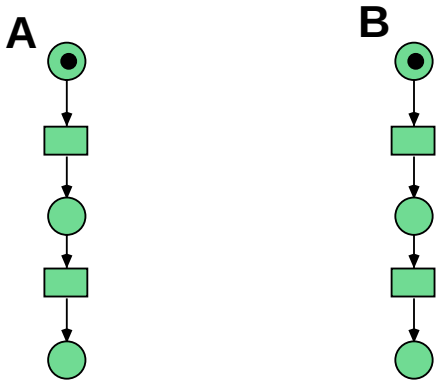
$\Rightarrow L \sim$ hierarchical Petri nets

Problem: the "standard" PN composition operators bear little resemblance to those of other calculi . . .

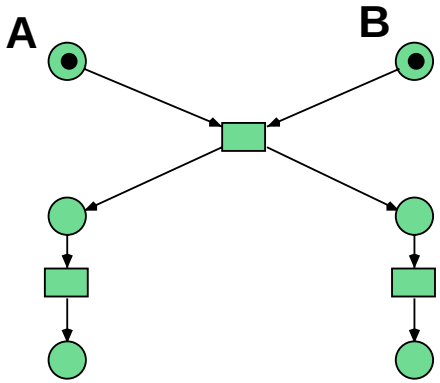
1. B =FL + PN + PN composition, S = Petri net structures

$\Rightarrow L \sim$ functional Petri nets

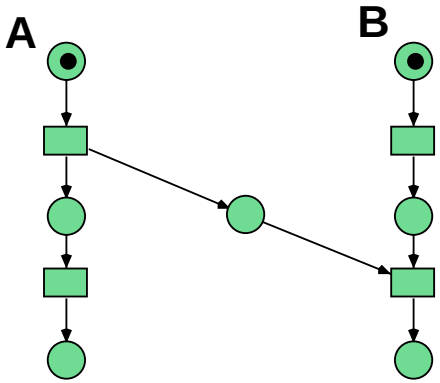
From FAs to CPNs – a micro-introduction



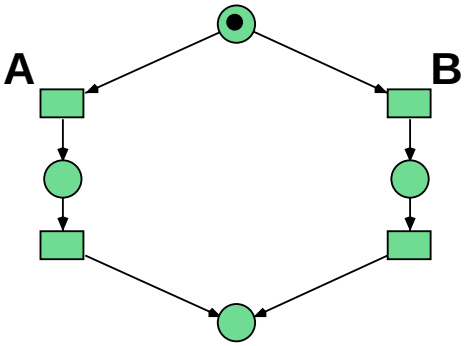
concurrency



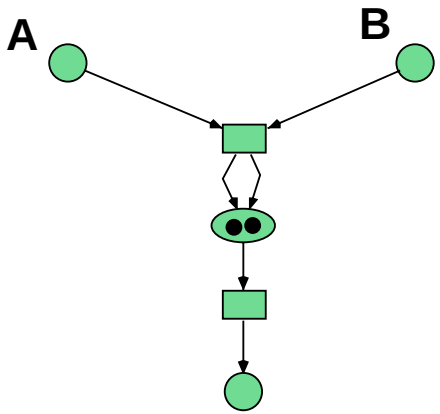
synchronisation



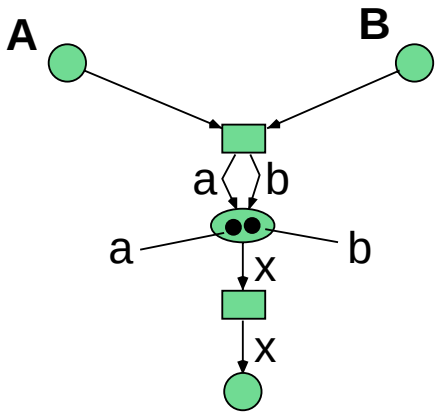
communication



conflict/choice



multiplicity



colour/individuality

Summary

- If you get a language implementation without upwards mapping, you are being sold short! *(the abstraction level breaks down for you)*
- Compilers, etc. shouldn't be visible outside CS, only computer-supported languages/notations should. *(languages as interfaces)*
- Some of the trouble in software engineering is home-made: we are trying to run the show alone. If we would provide domain experts with support for their domain-specific languages, they could do what they are good at instead of waiting for us to deliver! *(applications considered harmful)*
- More languages mean more language designers: can't remain an art for a few. We need to develop language design from an art to an engineering discipline. *(language design as a core CS topic)*
  - *prototype*: demonstrate my ideas about languages and their design
  - *theory*: what is the core of concurrency? Petri nets $\leftrightarrow$ linear logic.

"Formal" maths and "the" semantics of programming languages

- historically, *languages* were often *defined by their implementation*.
- as a counter-reaction, *mathematical semantics* was promoted as the only known way towards an *implementation-independent, formal semantics*.
- a useful development, but many evangelists went over the top:
  - most *mathematicians* turn out to be *not* half as *formal* as we -naively- thought them to be. Nor is maths itself static or always built on firm and absolute foundations.
  - the *structures* usually studied *in maths* are *not always* the most *appropriate for* the study of *CS phenomena*.
  - what is a *mathematical semantics* but *yet another abstract implementation*? Outsiders (better: experts in other domains) find both equally (un-)helpful.