Teaching Programming Broadly and Deeply: The Kernel Language Approach

July 17, 2002

Peter Van Roy
Université catholique de Louvain (UCL)
Louvain-la-Neuve, Belgium

Seif Haridi
Royal Institute of Technology (KTH)
Stockholm, Sweden

Overview

- The goal
  - Programming as a unified discipline with a practical scientific foundation
- The kernel language approach
  - The declarative programming model
  - Extended models
    - With exceptions, security, state, concurrency, laziness, and search
  - Programming and reasoning within these models
  - Programming paradigms as epiphenomena
- Practical examples of the kernel approach
  - Concurrent programming
  - User interface programming
- Conclusions
  - The textbook and its use in education

Programming needs both technology and science

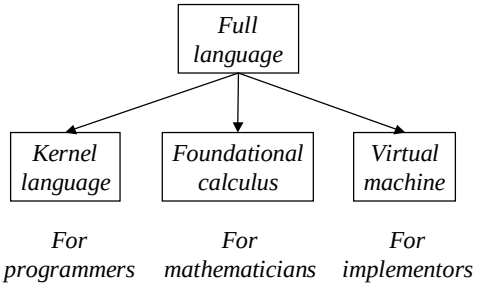
- We define **programming** broadly as the step from specification to running program, which consists in designing the architecture and its abstractions and coding them into a programming language
- Doing programming well requires two topics:
  - A **technology**: a set of practical techniques, tools, and standards
  - A **practical scientific foundation**: a scientific theory that explains the technology and that is useful for the practicing programmer
- Teaching programming requires teaching both the technology and the science
  - Surprisingly, programming is **not** taught in this way. It is taught as a **craft** in the context of current technology (e.g., Java and its tools). If there is any science, it is either limited to the tools or too theoretical.
- We propose a remedy, **the kernel language approach**

The kernel language approach (1)

- Modern programming languages have been developed for over 50 years and scale to millions of lines of code. Students should learn the fundamental concepts underlying this success.
- Modern languages have been designed to solve many problems, and are therefore rich and expressive. Unfortunately, they are superficially very different from each other. How can a student understand this variety in a simple and clear way?
- The kernel language approach is to translate these rich languages into small kernel languages. A wide variety of languages and programming paradigms can be modeled by a small set of closely-related kernel languages.
- For good reasons, the kernel language is not a foundational calculus (e.g., λ or π), although it does have a formal semantics

The kernel language approach (2)

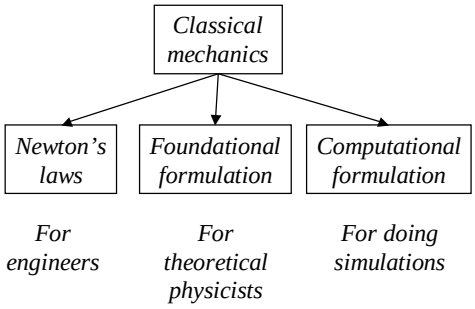
- Kernel languages have a small number of programmer-significant elements
- Their purpose is to understand programming from the programmer's viewpoint
- They are given a semantics which allows the practicing programmer to reason about correctness and complexity at a high level of abstraction
- All major programming paradigms are covered in a deep way



5

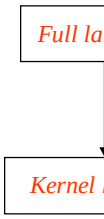
The kernel language approach (3): Analogy with classical mechanics

- Classical mechanics is a branch of physics that is widely used in engineering
- Classical mechanics is based on a small set of physical laws
- These laws can be formulated in three basically different ways, which are useful for different communities
- For engineers, the formulation based on Newton's laws (and its extensions) is the most useful in practice



6

The kernel language approach (4)




```
fun {Sqr X} X*X end
```

```
proc {Sqr X Y}
   {'*' X X Y}
end
local T in
   {Sqr A T}
   {Sqr T B}
end
```

- The full language provides useful linguistic abstractions and syntactic sugar for the programmer
- New linguistic abstractions can be added without changing the kernel language

- The kernel language contains a small set of intuitive concepts
- The kernel language has a formal semantics at high level of abstraction (reasoning about correctness and complexity)

Linguistic abstractions

- An abstraction is a tool or device (possibly conceptual) that solves a particular problem. An abstraction is often useful for widely different problems.
- A linguistic abstraction is a programming abstraction that has linguistic support in the language, i.e., it is visible as a construct in the language syntax
- Linguistic abstractions are defined by giving their syntax and their translation into the kernel language
- Practical languages contain a large number of useful linguistic abstractions
- Our kernel language is based on procedures; useful linguistic abstractions we build on top of this kernel language include functions, loops, lazy functions, list comprehensions, choice points, classes, components, and mailboxes
- Linguistic abstractions should not be confused with syntactic sugar, which reduces program size and improves program readability. Syntactic sugar is also defined by translation to the kernel language, but it does not define a new abstraction.

The first model: declarative programming

<s> ::=
  skip
  <x>$_1$=<x>$_2$
  <x>=<v>
  <s>$_1$ <s>$_2$
  local <x> **in** <s> **end**
  if <x> **then** <s>$_1$ **else** <s>$_2$ **end**
  case <x> **of** <p> **then** <s>$_1$ **else** <s>$_2$ **end**
  {<x> <y>$_1$ … <y>$_n$}

<v> ::= <number> | <record> | <procedure>
<number> ::= <int> | <float>
<record>, <p> ::= <lit>(<feat>$_1$:<x>$_1$ … <feat>$_n$:<x>$_n$)
<procedure> ::= **proc** {$ <x>$_1$ … <x>$_n$} <s> **end**
<lit> ::= <atom> | <bool>
<feat> ::= <atom> | <bool> | <int>
<bool> ::= **true** | **false**

- Nonterminals:
  - <s>: statement
  - <x>: variable identifier
  - <v>: value expression
  - <p>: pattern
- Single-assignment store
  - unbound variables
  - variables bound to partial value
- Semantics in terms of a simple abstract machine
- The model calculates functions over partial values, in procedural notation

9

Formal semantics (1)

- Basic concepts:
  - A *single-assignment store* σ is a set of store variables x_1, …, x_k, that are partitioned into sets of equal unbound variables and variables bound to a number, record, or procedure
  - An *environment* E is a mapping from variable identifiers to store variables, {<x>$_1$ → x_1, …, <x>$_n$ → x_n}
  - A *semantic statement* is a pair (<s>, E) where <s> is a statement and E is an environment
  - An *execution state* is a pair (ST, σ) where ST is a stack of semantic statements
  - A *computation* is a sequence of execution states starting from an initial state: (ST_0, σ_0) → (ST_1, σ_1) → (ST_2, σ_2) → ...

10

Formal semantics (2)

- Program execution
  - The initial execution state is ([(<s>,ϕ)], ϕ). The initial semantic statement is (<s>,ϕ) with an empty environment, and the initial store is empty.
  - At each execution step, the first element of *ST* is popped and execution proceeds according to the form of the element
  - The final execution state (if it exists) is one in which the semantic stack is empty.
- A semantic stack can be in one of three run-time states:
  - *running*: *ST* can do an execution step
  - *terminated*: *ST* is empty
  - *suspended*: *ST* is not empty but cannot do a step

Example: the **local** statement

- The semantic statement is (**local** <x> **in** <s> **end**, *E*)
- Execution consists of the following actions:
  - Create a new variable *x* in the store
  - Push (<s>, *E*+{<x>→*x*}) on the stack

Example: the **if** statement

- The semantic statement is (**if** <x> **then** <s>$_1$ **else** <s>$_2$ **end**, *E*)
- This statement has an activation condition: *E*(<x>) must be determined, i.e., bound to a number, record, or procedure
- Execution consists of the following actions:
  - If the activation condition is **true**, then do the following actions:
    - If *E*(<x>) is not a boolean (**true** or **false**), then raise an error condition
    - If *E*(<x>) is **true**, then push (<s>$_1$, *E*) on the stack
    - If *E*(<x>) is **false**, then push (<s>$_2$, *E*) on the stack
  - If the activation condition is **false**, then execution suspends
- If some other activity in the system makes the activation condition true, then execution can continue. This allows dataflow programming, which is at the heart of declarative concurrency.

13

Example: procedures

- A procedure value is a pair (**proc** {$ <y>$_1$ … <y>$_n$} <s> **end**, *CE*) where *CE* (the « contextual environment ») is $E|_{\{<z>1, \ldots, <z>m\}}$, where *E* is the environment where the procedure is defined and {<z>$_1$, …, <z>$_m$} is the set of external identifiers of the procedure

- In a procedure call ({<x> <x>$_1$ … <x>$_n$}, *E*):
  - if *E*(<x>) has the form (**proc** {$ <y>$_1$ … <y>$_n$} <s> **end**, *CE*) , then
  - push (<s>, *CE*+ {<y>$_1$→*E*(<x>$_1$), …, <y>$_n$ →*E*(<x>$_n$)})

- This allows high-order programming as in functional languages

14

Relationship to other declarative paradigms

- The declarative model generalizes both strict functional programming and deterministic logic programming
- It is functional programming with logic variables
  - This increases expressiveness yet is still confluent, e.g., append is naturally tail recursive
  - There is a deeper reason for this model: it leads to declarative concurrency, which is both confluent and concurrent
- It is logic programming with higher-order procedures and without search
  - Higher-order is a powerful way to structure logic programs
  - Search is not necessary for most practical logic programs; in those areas where it is needed, constraint programming is a good fit

Importance of declarative programming

- Why is the declarative model important? Two reasons:
  - A declarative program can be **partitioned into components** that each be written, tested, and proved correct independently of the others
  - **Simple reasoning techniques** (e.g., reasoning with induction on data structures) can be used in the declarative model
- Proper role of the model
  - Partitioning does not work if intimate interaction between components is needed; it cannot be « legislated away » by limiting programs to a model that does not support it
  - More expressive models exist that support intimate interaction, but components written in them are harder to prove correct
    - Intimate interaction is supported by the concurrent stateful model, but this model is difficult to program in (reasoning on all possible interleavings)
  - An important rule is that intimate interaction should only be used where necessary and concentrated in as few components as possible
  - The programming model should support this rule by allowing both declarative and nondeclarative programs, with no syntactic or performance penalty for declarativeness

Limitations of declarative programming

- When is the declarative model appropriate and when is encapsulated state needed?
  - An important question that arouses strong religious feelings
  - Concurrency is not a limitation of the declarative model
- We find the following real limitations:
  - It leads to **modularity problems** in certain situations that need hidden state or that have observable nondeterminism. For example: an external database interface, an instrumented program, a client-server program with more than one independent client, and a function with memoization.
  - It can lead to **intricate code**. This follows because declarative programs impose more restrictions on how they are written. For example, a transitive closure algorithm.
- We find the following are not limitations in practice:
  - Programs that do incremental modifications of large data structures, e.g., for simulations, can be efficient if written in the right way.
  - Interfacing between declarative and non-declarative programs.
  - Problem specifications that mention state must be encoded in a declarative way.

Extension for exception handling

<s> ::=
  skip
  $\langle x \rangle_1 = \langle x \rangle_2$
  $\langle x \rangle = \langle v \rangle$
  $\langle s \rangle_1$ $\langle s \rangle_2$
  local <x> **in** <s> **end**
  if <x> **then** $\langle s \rangle_1$ **else** $\langle s \rangle_2$ **end**
  case <x> **of** <p> **then** $\langle s \rangle_1$ **else** $\langle s \rangle_2$ **end**
  $\{\langle x \rangle \langle y \rangle_1 \ldots \langle y \rangle_n\}$

  try $\langle s \rangle_1$ **catch** <x> **then** $\langle s \rangle_2$ **end**
  raise <x> **end**

Extension to the declarative model

- The declarative model does not allow handling exceptional situations
- We extend the model to add exception handling
- We add two new statements:
  - **try**: create new exception context
  - **raise**: raise an exception
- Exception contexts are nested dynamically; a raised exception transfers execution to the innermost context

Other extensions

<s> ::=
skip
<x>$_1$=<x>$_2$
<x>=<v>
<s>$_1$ <s>$_2$
local <x> **in** <s> **end**
if <x> **then** <s>$_1$ **else** <s>$_2$ **end**
case <x> **of** <p> **then** <s>$_1$ **else** <s>$_2$ **end**
{<x> <y>$_1$ … <y>$_n$}
try <s>$_1$ **catch** <x> **then** <s>$_2$ **end**
raise <x> **end**

- Extensions: threads, triggers (for demand-driven execution; lazy functions are a linguistic abstraction), and cells (encapsulated state; object-oriented programming is a linguistic abstraction)
- The extensions vastly increase the expressive power of the language
- For conciseness we leave out the extensions for constraint programming

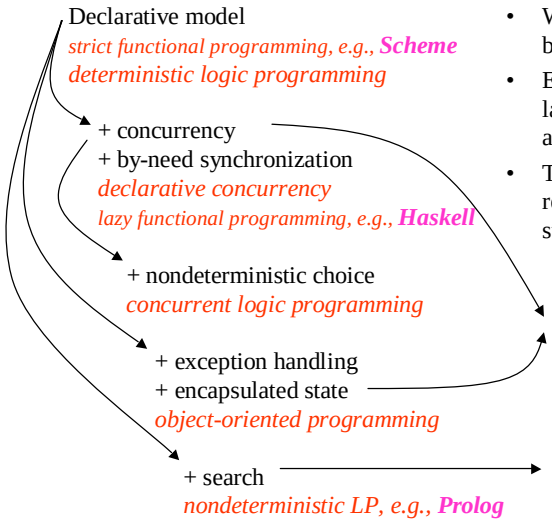
thread <s> **end**	*Thread creation*
{ByNeed <x>$_1$ <x>$_2$}	*Trigger creation*
{NewCell <x>$_1$ <x>$_2$}	*Cell creation*
{Exchange <x>$_1$ <x>$_2$ <x>$_3$}	*Cell exchange*

Extensions to the declarative model

Complete model (so far)

<s> ::=	
skip	*Empty statement*
<x>$_1$=<x>$_2$	*Variable-variable binding*
<x>=<v>	*Variable-value binding*
<s>$_1$ <s>$_2$	*Sequential composition*
local <x> **in** <s> **end**	*Variable creation*
if <x> **then** <s>$_1$ **else** <s>$_2$ **end**	*Conditional*
case <x> **of** <p> **then** <s>$_1$ **else** <s>$_2$ **end**	*Pattern matching*
{<x> <y>$_1$ … <y>$_n$}	*Procedure invocation*
thread <s> **end**	*Thread creation*
{ByNeed <x>$_1$ <x>$_2$}	*Trigger creation*
{NewName <x>}	*Name creation*
try <s>$_1$ **catch** <x> **then** <s>$_2$ **end**	*Exception context*
raise <x> **end**	*Raise exception*
{NewCell <x>$_1$ <x>$_2$}	*Cell creation*
{Exchange <x>$_1$ <x>$_2$ <x>$_3$}	*Cell exchange*
<space>	*Encapsulated search*

Computation models



- We show some of the relationships between the different models
- Each model has its own kernel language, its own reasoning techniques, and its own programming techniques
- The kernel languages are closely related, e.g., the declarative model is a subset of all of them

Some reasoning techniques for different models

- Correctness
  - Reasoning with induction on data structures (declarative model, including declarative concurrency)
  - Algebraic reasoning and related techniques (declarative model, including declarative concurrency)
  - Reasoning with induction on invariants (declarative and stateful models)
  - Reasoning on interleavings, use of atomic actions (concurrent stateful model)
- Execution time (asymptotic complexity)
  - Set up and solve recurrence equations
- Memory use
  - Active memory: reachability calculation on data structures
  - Memory consumption: set up and solve recurrence equations

Programming paradigms as epiphenomena

- We have seen that the kernel approach lets us organize programming in three levels:
  - **Concepts**: compositionality, encapsulation, lexical scoping, higher-orderness, capability property, concurrency, dataflow, laziness, state, inheritance, ...
  - **Techniques**: how to write programs with these concepts
  - **Computation models** (« paradigms »): a model is a subset of the concepts, realized with data entities, operations, and a language
- Programming paradigms *emerge in a natural way* when programming (as a kind of epiphenomenon), depending on which concepts one uses and which properties hold of the resulting model
- It is often advantageous for programs to use several paradigms together, as the examples we will give show

Practical examples of the kernel approach

- We have given an overview of the technical side of the kernel approach
- Now we will give examples to show that programming in terms of concepts and not languages or paradigms is advantageous in practice
- We give two detailed examples in this talk:
  - Concurrent programming
  - User interface programming
- Other good examples are given in the book:
  - Distributed programming
  - Constraint programming

Approaches to concurrency

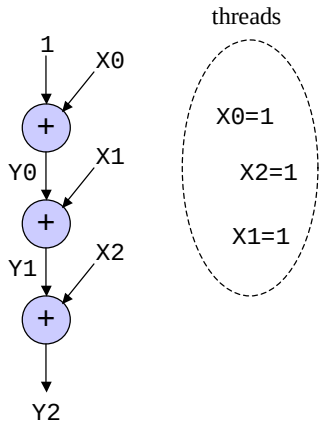
- We distinguish four forms of practical concurrency (in order of increasing difficulty):
  - Sequential programming + its variants
  - Declarative concurrency + lazy execution (add threads to a functional language and use **dataflow** to decouple independent calculations)
  - Message passing between active objects (Erlang style, each thread runs a functional program, threads communicate through asynchronous channels)
  - Atomic actions on shared state (Java style, using monitors and transactions)
- The Java style is the most popular, yet it is the most difficult to program
- We will give examples of declarative concurrency and message passing
  - Both avoid most of the complexities while still giving the advantages of concurrent execution
- Declarative concurrency especially is quite useful, yet is not widely known
  - All the programming and reasoning techniques of sequential declarative programming apply (concurrent programs give the same results as sequential ones)
  - Deep characterization: lack of observable nondeterminism

Declarative concurrency (1)

- Dataflow = block on data availability
- Add dataflow variables (i.e., logic variables) to a functional language

```
declare X0 X1 X2 X3
        Y0 Y1 Y2 in
Y0=1+X0
Y1=Y0+X1
Y2=Y1+X2

thread X0=1 end
thread X1=1 end
thread X2=1 end
```



Declarative concurrency (2)

- A system can be concurrent and still be purely functional
- Declarative concurrency has this property
- Here is a simple example:

```
fun {Fibo N}
   if N=<2 then 1
   else F1 F2 in
      thread F1={Fibo N-1} end
      F2={Fibo N-2}
      F1+F2
   end
end
```

- The results are the same, whether or not threads are used

27

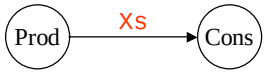
Declarative concurrency (3)

- Producer-consumer with dataflow

```
fun {Prod N Max}
   if N<Max then
      N|{Prod N+1 Max}
   else nil end
end
```

```
fun {Cons Xs A}
   case Xs of X|Xr then
      {Cons Xr A+X}
   [] nil then A end
end
```

```
local Xs S in
   thread Xs={Prod 0 1000} end
   thread S={Cons Xs 0} end
end
```



- Prod and Cons threads share list Xs
- Dataflow behavior of case statement (synchronizing on data availability) gives stream communication
- No other concurrency control needed

28

Declarative concurrency (4)

- Let us compare the sequential and concurrent versions
  - The result of the calculation is the same in both cases
  - So what is different?
- Sequential version:

```
local Xs S in
   Xs={Prod 0 1000}
   S={Cons Xs 0}
end
```

  Results are produced *in batch*: the whole calculation is done and then all results are given at once

- Concurrent version:

```
local Xs S in
   thread Xs={Prod 0 1000} end
   thread S={Cons Xs 0} end
end
```

  Results are produced *incrementally*, element by element

Declarative concurrency (5)

- Lazy producer-consumer with dataflow

```
fun lazy {Prod N}
   N|{Prod N+1}
end
```

(note « lazy » annotation)

```
fun {Cons Xs A Max}
   if Max>0 then
      case Xs of X|Xr then
         {Cons Xr A+X Max-1}
      end
   else A end
end
```

```
local Xs S in
   thread Xs={Prod 0} end
   thread S={Cons Xs 0 1000} end
end
```

- Lazy = demand-driven
- Flow control: the consumer decides how many list elements to create
- Dataflow behavior ensures concurrent stream communication

Active objects

- An active object is a concurrent entity to which any other active object can send messages
- The active object reads the messages in arrival order and sequentially executes an action for each message
- An active object's behavior is defined by a class, just like a passive object
- Active objects can be considered either as primitive or as defined with a thread, a passive object, and a communication channel
- Creation: `A={NewActive Class Init}`

Event manager with active objects

- An event manager contains a set of event handlers
- Each handler is a triple `Id#F#S` where Id identifies it, F is the state update function, and S is the state
- Reception of an event causes all triples to be replaced by `Id#F#{F E S}` *(transition from S to {F E S})*
- The manager EM is an active object with four methods:
  - `{EM init}` initializes the event manager
  - `{EM event(E)}` posts event E at the manager
  - `{EM add(F S Id)}` adds new handler with F, S, and returns Id
  - `{EM delete(Id S)}` removed handler Id, returns state
- This example taken from real use in Erlang

Defining the event manager

- Mix of functional and object-oriented style

```
class EventManager
   attr handlers
   meth init handlers<-nil end
   meth event(E)
      handlers<-
         {Map @handlers fun {$ Id#F#S} Id#F#{F E S} end}
   end
   meth add(F S Id)
      Id={NewName}
      handlers<-Id#F#S|@handlers
   end
   meth delete(DId DS)
      handlers<-{List.partition
         @handlers fun {$ Id#F#S} DId==Id end [_#_#DS]}
   end
end
```

State transition done using functional programming

Using the event manager

- Simple memory-based handler keeps list of events

```
EM={NewActive EventManager init}

MemH=fun {$ E Buf} E|Buf end
Id={EM add(MemH nil $)}

{EM event(a1)}
{EM event(a2)}
...
```

- An event handler is purely functional, yet when put in the event manager, the latter is a concurrent imperative program. This is an example of impedance matching between paradigms.

Defining active objects

- Define `NewActive` in terms of existing `New` (passive object creation) by adding one port and one thread

```
fun {NewActive Class Init}
   S P Obj
in
   {NewPort S P}
   Obj={New Class Init}
   thread
      for M in S do {Obj M} end
   end
   proc {$ M} {Send P M} end
end
```

Port P is created together with stream S (dataflow list)

For loop does dataflow synchronization (like case statement)

Sending to a port causes message to appear on stream

35

Concurrency - conclusions

- There are two forms of concurrency that are simpler to program with than the shared state style: declarative concurrency and active objects
- Declarative concurrency is the simplest. Same results as sequential programming, yet the calculation is interleaved to make it incremental
  - It is useful for stream communication (sample application: circuit simulation)
  - Declarative concurrency is only usable in those situations where there is no observable nondeterminism
- Active objects are an extension to stream communication with many-to-one communication, which makes them as expressive as the passive object / monitor approach
  - Active objects can be used together with passive objects, where the active object is used as a serializer
  - Active objects require cheap threads to be practical

36

User interface design

- Three approaches:
  - *Imperative approach* (AWT, Swing, tcl/tk, …): maximum expressiveness but also maximum development cost
  - *Interface builders*: adequate for the part of the UI known before the application runs
  - *Declarative approach*: reduced development cost but limited expressiveness
- All are unsatisfactory for dynamic user interfaces, which change during execution

Mixed declarative/imperative approach to UI design

- Using both approaches together is advantageous:
  - Declarative specification is a data structure. It is concise and can be manipulated with all the power of the language.
  - Imperative specification is a program. It has maximum expressiveness.
- This makes creating dynamic user interfaces particularly easy
- This is important for *model-based UI design*, an important design methodology in the Human-Computer Interface research community

Mixed approach

- Declarative part
  - A widget is a record. A full UI specification is a nested record.
  - The nested record specifies interface structure and resize behavior, and all widget types with their initial states
- Imperative part
  - External events cause action procedures to be executed (sequentially, in the window's thread)
  - Widgets have handler objects, which allows the application to control them

Example user interface

Nested record with handler object E and action procedure P

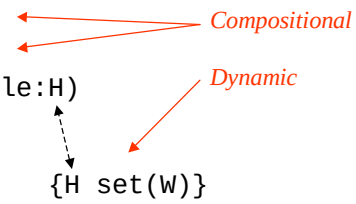
Construct interface (window & handler)

Call handler

```
W=td(lr(label(text:«Enter your name»)
        entry(handle:E))
     button(text:«Ok» action:P))
...
{Build W}
...
{E set(text:«Type here»)}
...
Result={E get(text:$)}
```

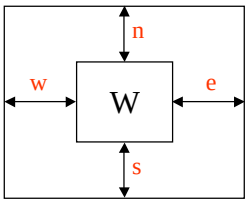
Widgets

- Widget = rectangular area with particular behavior
- Examples (as records):
  - `label(text:«Hello»)`
  - `text(handle:H tdscrollbar:true)`
  - `button(text:«Ok» action:P)`
  - `lr(W1 W2 ... Wn)`
  - `td(W1 W2 ... Wn)`
  - `placeholder(handle:H)`

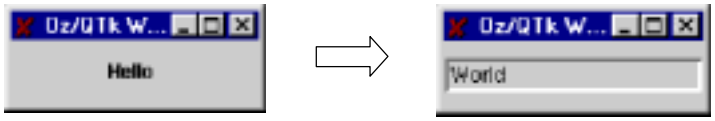


Declarative resize behavior

- Resizing is dynamic but specified declaratively
- Declarative specification with « glue »
- Consider widget W inside another:
- W = <type>(... glue:<g>)
  - nswe: stretch horizontal and vertical
  - we: stretch horizontal, centered vertical
  - w: left justified, centered vertical
  - (none): centered, keeps natural size



Example dynamic interface




```
W=placeholder(handle:P)
...
{P set(label(text:«Hello»))}
...
{P set(entry(text:«World»))}
```

- Any UI specification can be put in the placeholder at run-time

Calculating interfaces (1)

- Calculate interface directly from internal data representation

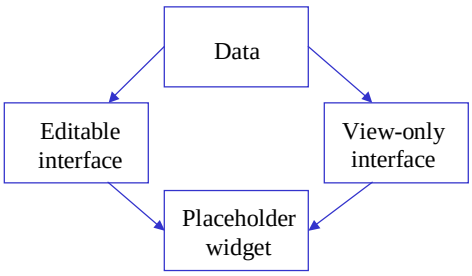
```
Data=[«Name»#«Roger»
     «Surname»#«Rabbit»]

Result=
  {ListToRecord td
    {Map Data
      fun {$ L#E}
         lr(label(text:L) entry(init:E))
      end}}

Result=td(lr(label(text:«Name») entry(init:«Roger»))
          lr(label(text:«Surname») entry(init:«Rabbit»)))
```

Calculating interfaces (2)

- Calculate several interfaces from the same data
- Choose between representations according to run-time condition



- With language support (syntax and implementation) for record operations and functional programming this is very concise
- This is another example of impedance matching between paradigms

The book and the formalism

- We have written most of a textbook that realizes the kernel language approach
  - The latest draft is always available at **`http://www.info.ucl.ac.be/people/PVR/book.html`**
- The book uses subsets of the Oz language for the different computation models, for the following reasons:
  - Oz was designed to integrate programming concepts into a coherent whole
  - Oz incorporates ten years of application development experience: its concepts and techniques have been tested in real use (industrial projects, deployed applications)
  - Oz has a complete and simple formal semantics
  - Oz has a high-quality fully-featured implementation, the Mozart Programming System (see **`http://www.mozart-oz.org`**)
- Oz does not have a Java-compatible syntax
  - Java can easily be translated into Oz; see for example the CC-Java work
  - In any case, computer scientists should be familiar with several notations
- We do not know any other formalism that covers so much ground so well
  - But we could be mistaken: please let us know!

Conclusions

- Programming has traditionally been taught as a fragmented discipline
  - There is too much emphasis on language idiosyncrasies to the detriment of fundamental concepts
  - Paradigms are considered in isolation
  - The science is either limited to the paradigms or is too theoretical
- The kernel language approach is intended to remedy this situation
  - Practical languages are understood by translating them to simple kernel languages based on small sets of programmer-significant concepts
  - The kernel languages have much in common with each other, which allows them to show clearly the deep relationships between different languages and programming paradigms
  - We give a semantics at the right level of abstraction for the practicing programmer, to allow reasoning about correctness and complexity
- We are finishing a textbook that realizes this approach and we are teach-testing it in three universities in the Fall 2001 and Spring 2002 semesters. We consider the textbook as a worthy successor to the book « Structure and Interpretation of Computer Programs », by Abelson and Sussman.