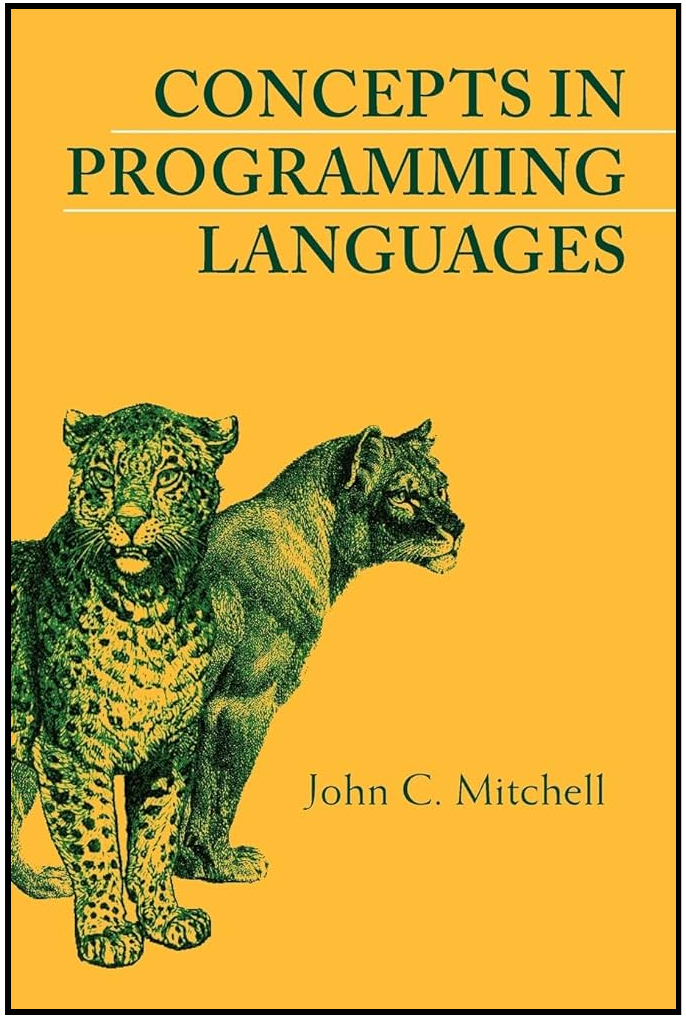
NPRG077

TinySelf: Tiny prototype-based object-oriented language

Tomáš Petříček, 309 (3rd floor)
petricek@d3s.mff.cuni.cz
https://tomasp.net | @tomaspetricek
https://d3s.mff.cuni.cz/teaching/nprg077



Object-orientation

Dynamic lookup - object chooses how to respond

Abstraction - object state can be hidden from user

Subtyping - any compatible object can be used

Inheritance - reuse to implement a new object

Charles University

Brief history

Origins of object orientation (1960s-70s)

Algol-based and scientific Simula

Tools for thought and messaging in Smalltalk

Conceptual development (1980s)

Rigorous Eiffel and "serious" C++

Prototypes and materialized objects in Self

Commercial refinement (1990s-2000s)

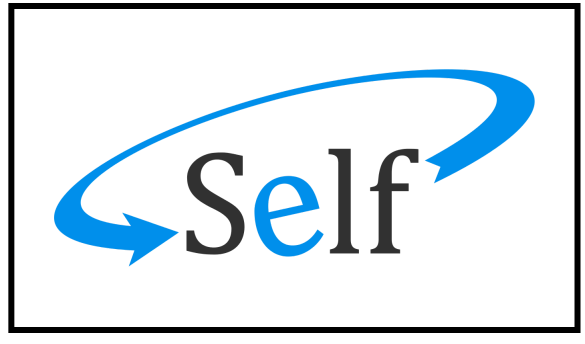
Class-based safe Java and C#

Prototypes in JavaScript and typed TypeScript

Why TinySelf?

"Pure" object-orientation

- Simple, uniform system
- Everything is an object (for real)
- Simpler than class-based Smalltalk



Shows the potential of objects

- Not Java-style organization of code
- Objects and graphical interfaces!
- Objects with introspection and debugging!

Charles University

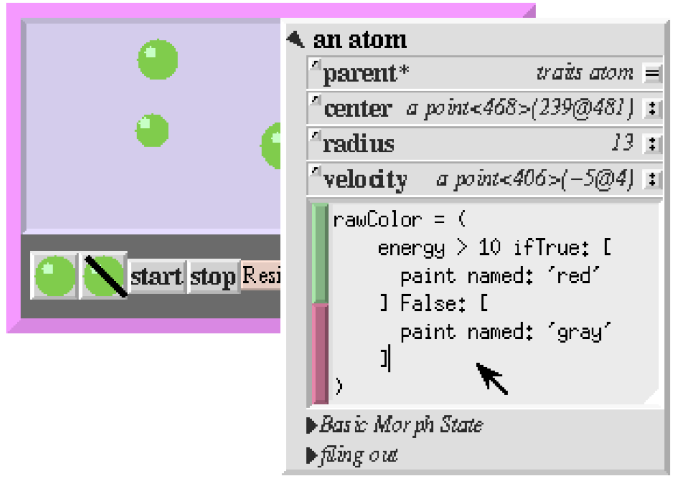


Figure 9. The user has selected one atom on which to experiment. The user changes the "rawColor" slot from a computed to a stored value by editing directly in the atom's outliner.

Self & Morphic user interface framework

Visual programming

Programming by graphically manipulating objects on screen

Direct programming

Objects on screen *are* objects in the system

Charles University

Demo

(Not so) Tiny Smalltalk

TinySelf

Scope of the implementation

- Prototype-based multiple inheritance
- Methods with simple interpreted code
- Explaining basic runtime structures
- Inaccurate interpreter in "Self style"
- Sketch of what UI framework might look like

Charles University

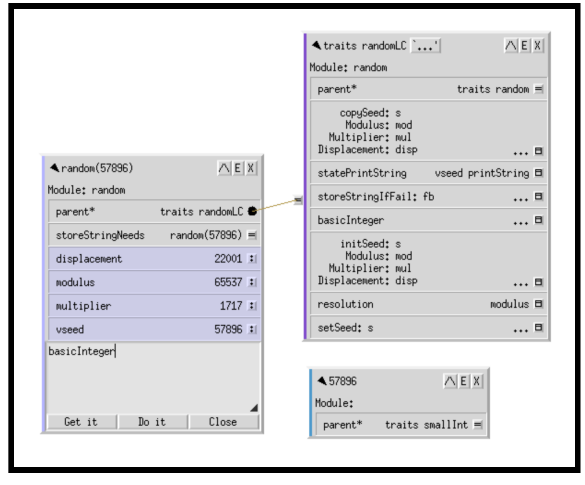
TinySelf

Self programming paradigm

Everything is an object

Really everything

- Objects, methods, lambdas, expressions, activation records
- Object has lists of slots and optionally contains code



Obect = slots* + code?

- Data object has just slots
- Method object has code
- Closure has code and slots!
- Data object has methods as slots

Charles University

```
// Object consists of zero or more
// slots and optionally code
type Objekt =
  { mutable Code : Objekt option
    mutable Slots : Slot list }

// A slot has name and contents;
// Some slots are parents
and Slot =
  { Name : string
    Contents : Objekt
    IsParent : bool }
```

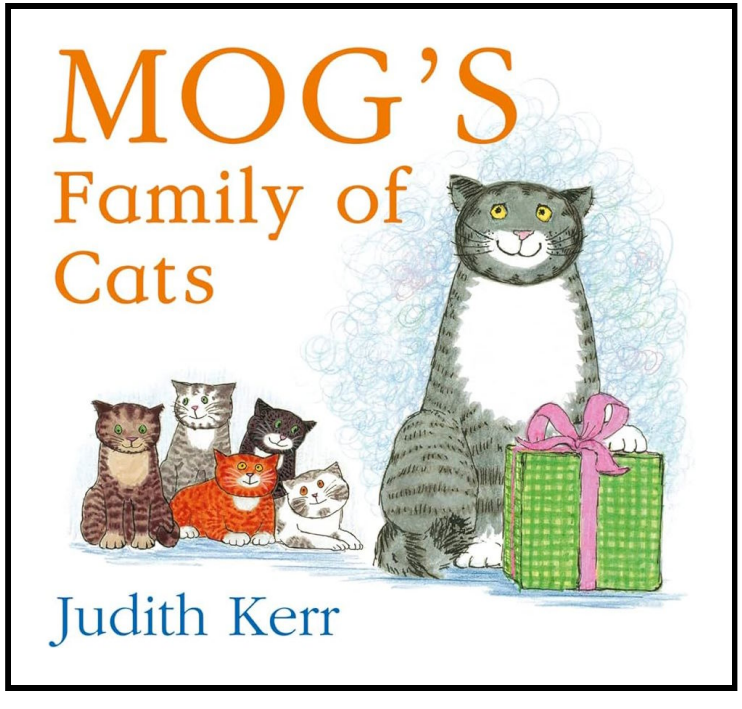
TinySelf objects

Object consists of zero or more slots and optional code!

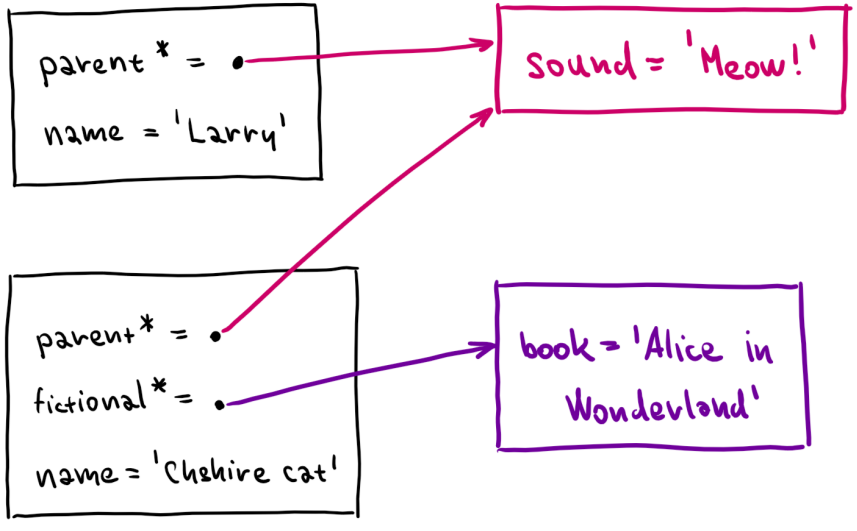
In Self parent slot names end with *

TinySelf objects can also be special things

Charles University



Charles University



Prototypes and slots

Message send looks at target object first, then searches parents

```
cheshire name   // OK
cheshire book   // OK

larry name   // OK
larry book   // Fail
```

Message send fail if none or multiple slots found

Charles University

Demo

Representing cats in Self

```
"""Data object with name"""
(| book = 'Alice in Wonderland' |)

"""Method with  some code"""
( self name printLine )

"""Data object with parent
 slot and a 'speak' method"""
(| parent* = cat
   name = 'Cheshire Cat'
   speak = (
     self sound printLine
   )
|)

"""Data access or method call"""
cheshire name
cheshire speak
```

Message sending

Lookup slot with a matching name, then:

- If it contains data object, it is returned
- If it contains method, the method is called

Assignment slots and special calls differ...

Charles University

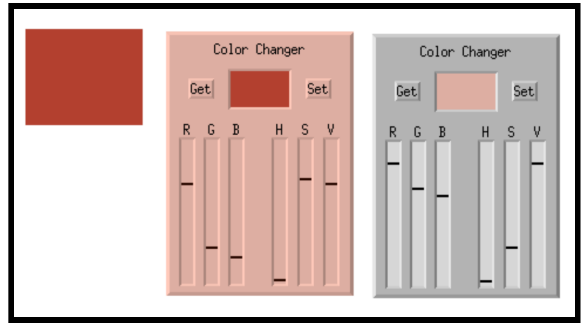
Demo

Hello world, traits and cloning

The power of simplicity...

Simplicity and uniformity

- All objects can be opened!
- Activation records for debugging
- Self-sustainable system



Morphic framework

- Things on screen are objects!
- Object with a morph prototype can draw itself
- User interface is just morphs - no special code!

Charles University

Demo

Morphic and graphical objects

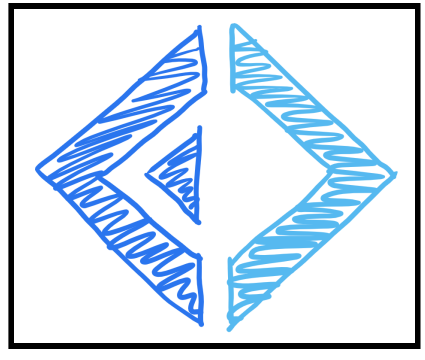
The F# language

What we need for TinySelf

Mutable records in F#

Defining mutable objects

- Records with `mutable` fields
- We could use classes too



Equality and records

- Still use structural equality by default
- Not if records (can) contain functions!
- `ReferenceEquality` attribute to override

Charles University

Mutable records

```
type Person =
  { mutable Name : string
    mutable Book : string option }

let setName n p =
  p.Name <- n
let setBook b p =
  p.Book <- Some b

let x = { Name = "Bill"; k = None }
x |> setName "William"
x |> setBook "Alice in Wonderland"

match x with
| { Book = Some book } ->
  printfn "%s likes %s" x.Name book
| _ ->
  printfn "%s is sad :-(" x.Name
```

Helper functions

Make code a bit nicer

Can support **|>** pipe

Pattern matching

Same as immutable

Nice data extraction!

Charles University

Demo

Working with mutable records

TinySelf programming style

Different than before!

Everything is an `Objekt`
Type definition stays
We change what we put in!

Uniformity has drawbacks
Everything type checks!

Helper methods

Simplify object construction

```
let makeString s =
  makeDataObject [
    makeParentSlot "parent*"
      stringPrototype
    makeSlot "value"
      (makeSpecial(String s))
    makeAssignmentSlot "value"
  ]
```

Charles University

TinySelf

Key implementation tricks

TinySelf

Key implementation tricks


How Self puts things on screen!

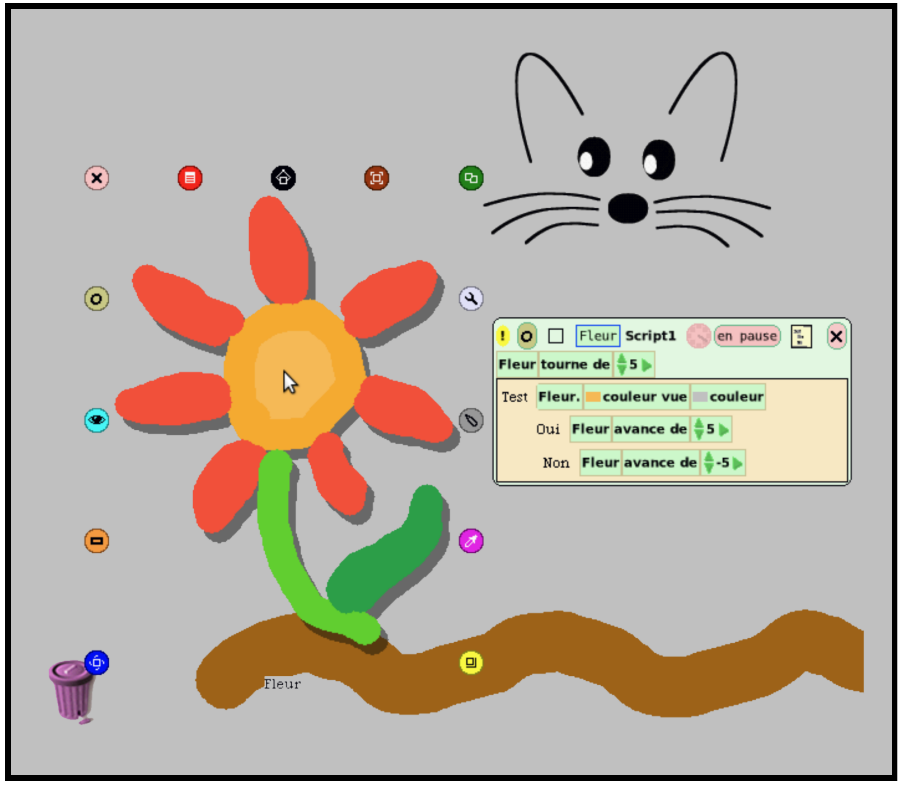

Slot lookup in parent objects


Message send to method/data slots


Activation records and calling


How TinySelf represents expressions

Charles University



How Self-like systems put things on screen?

Escape hatch is a must
Smalltalk system calls
Self primitive calls
(primitives primitiveList)

TinySelf special objects
Primitive string values
Native F# methods

Charles University

```
// Special TinySelf objects!
type Special =
  | String of string
  | Native of (Objekt -> Objekt)

// Optionally special object
and Objekt =
  { mutable Code : Objekt option
    mutable Special : Special option
    mutable Slots : Slot list }

// Code to clone an object
let cloneCode =
  { Slots = []; Code = None
    Special = Some(Native(fun arcd ->
      lookupSlotValue "self*" arcd
      |> cloneObject )) }

// Method with special code object
let cloneMethod =
  { Slots = []; Special = None;
    Code = Some cloneCode }
```

Special objects

String values

No other way to represent strings!

Native methods

F# function taking "activation record" and returning the result

Used as method code

Charles University

TinySelf

Lookup and message sending

Input:

`obj`, the object being searched for matching slots
`sel`, the message selector
`V`, the set of objects already visited along this path

Output:

`M`, the set of matching slots

Algorithm:

```
if obj ϵ V
then M ← ∅
else M ← {s ϵ obj | s.name = sel}
    if M = ∅ then M ← parent_lookup(obj, sel, V) end
end
return M
```

Where *parent_lookup(obj, sel, V)* is defined as follows:

```
P ← {s ϵ obj | s.isParent}
M ← ∪ lookup(s.contents, sel, V ∪ {obj})
   sϵP
return M
```

Slot lookup logic

1) Search target object

2) Search parents and union the results

3) Avoid infinite loops!

Charles University

Message sending logic

Self handbook

A normal send does a look-up to obtain the target slot;

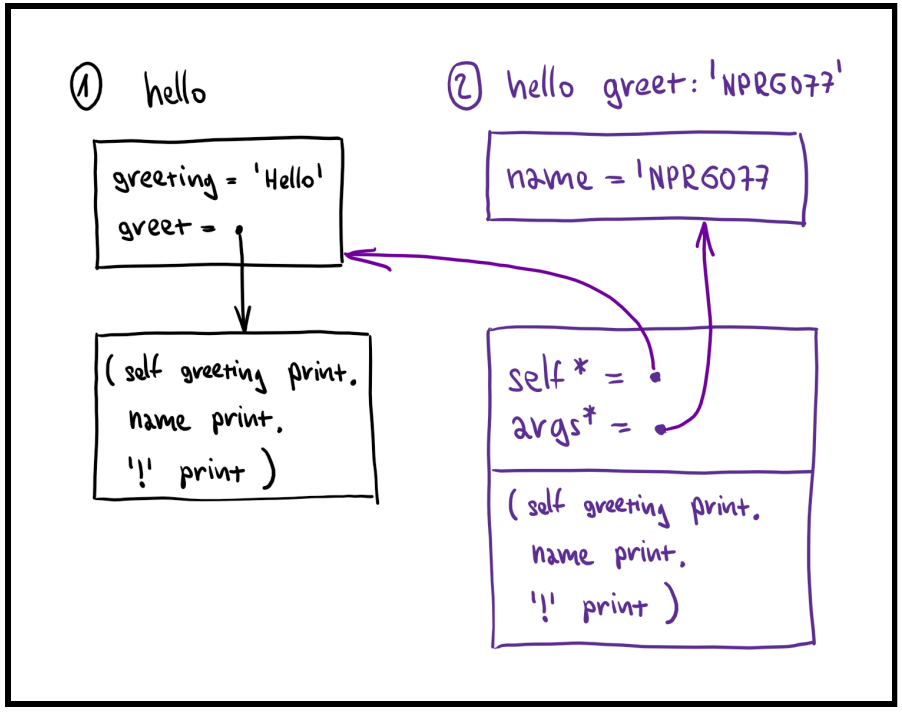
If the slot contains a data object, then the data object is simply returned.

If the slot contains a method, an activation is created and run.

TinySelf translation

1. Find slot using lookup!
2. Check it is exactly one
3. If there is no code, return it
4. If there is code, run it...
    - Create activation record
    - Run (non-)native code

Charles University



Activation record

Lookup in activation record to get all our code needs!

Clone of method

It could have data!

Self as parent

Access target's slots!

Arguments as parent

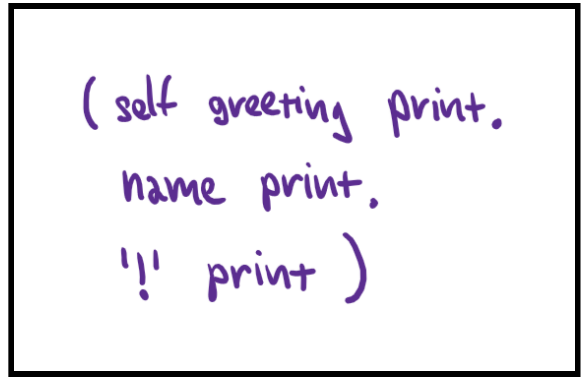
Access arguments!

Charles University

Representing TinySelf code

AST is a tree of objects

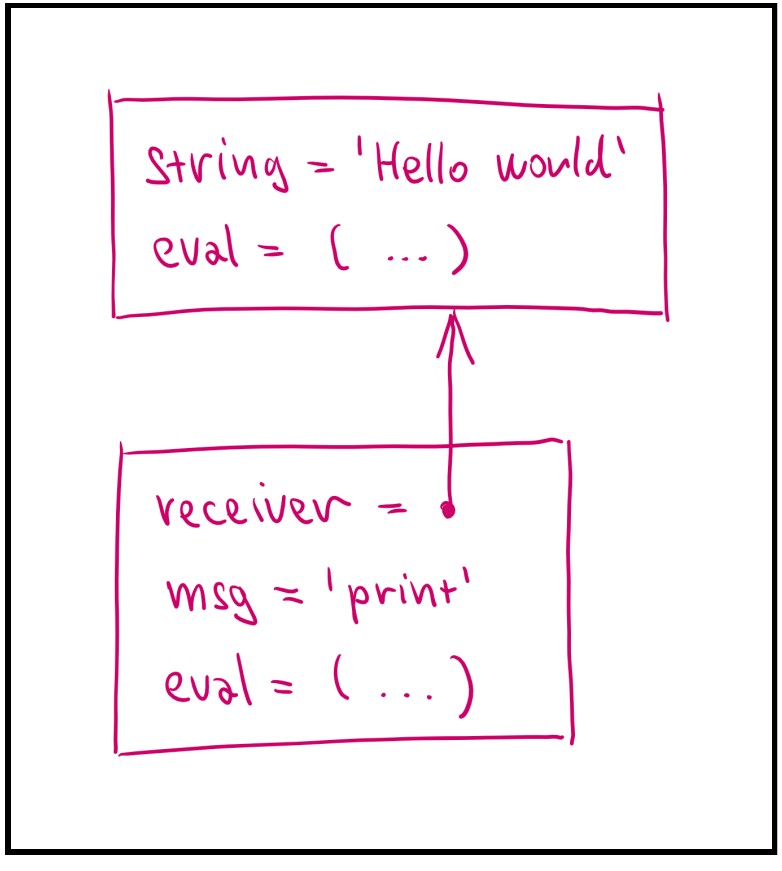
- All nodes have **`eval`** method
- Called with **`activation`** as argument
- Objects store sub-expressions etc.



Benefits and drawbacks

- Differs from normal Self or Smalltalk!
- Much simpler than compiled methods
- Beware! Values and expressions are **`Objekt`**!

Charles University



Simple expression

```
'Hello world' print
```

Send expression

Receiver, message, arguments to be used

String expression

String value to be returned

To make this nicer, put eval code into prototypes...

Charles University

Lab overview

TinySelf system step-by-step

Charles University

TinySelf - Basic tasks

1. Implementing slot lookup
   Traversing the prototype hierarchy to find slots
2. Implementing (basic) message sending
   Handling of data objects and (native) methods
3. Cloning and mutating TinySelf objects
   Assignment slots and clonable trait
4. Representing & interpreting TinySelf expressions
   Creating expression objects with `eval` method

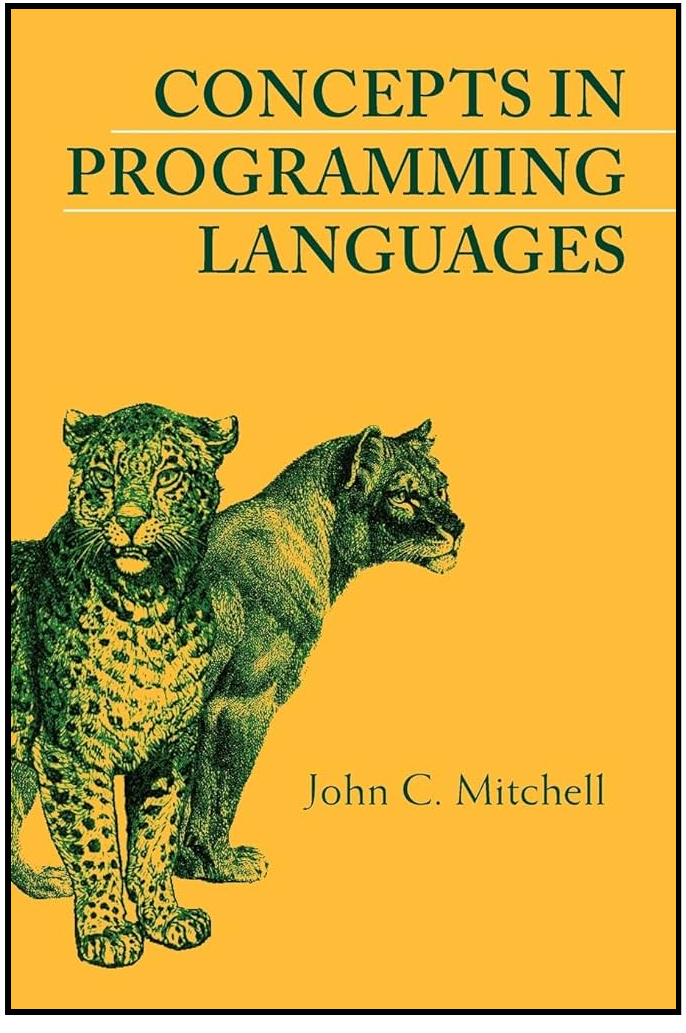
Charles University

TinySelf - Bonus and super tasks

1. Arguments and sequencing of expressions
   Adding more types of expressions to TinySelf
2. Booleans and 'if' as a message send!
   Booleans are just objects with an `if` method
3. Objects as lists and more expressions
   Adding more infrastructure before the next step...
4. Creating web-based visualizers
   A small step towards TinyMorphic framework

Closing

A tiny prototype-based OO language



TinySelf and OO

Dynamic lookup

Find method using lookup

Abstraction

No private slots in TinySelf

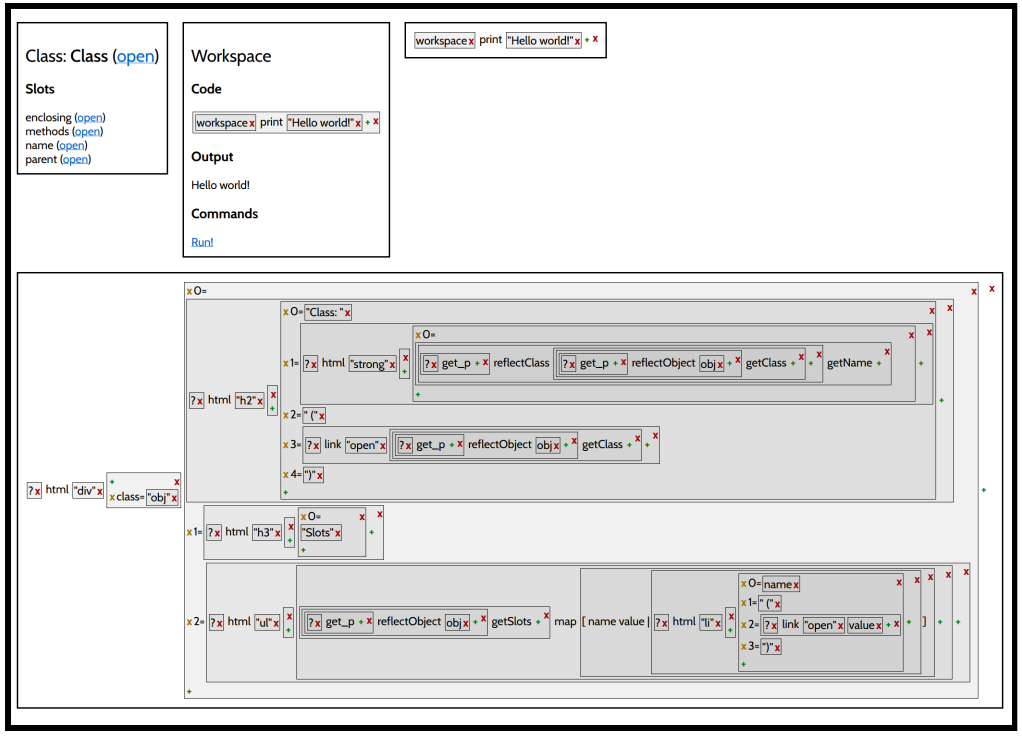
Subtyping

Object with required slots

Inheritance

By setting a parent slot

Charles University



What is missing

Self-sustainable

Complete basic library

Reflection capabilities

Reflection via mirrors

Mirror objects

Inspect & modify

Done in Nanospeak

Charles University

Conclusions

A tiny prototype-based object-oriented language

- Basic logic of object-oriented languages
- Shows how to build self-sustainable system
- Different implementation - everything is object
- Hard to implement! Need debuggers, not types

Tomáš Petříček, 309 (3rd floor)
petricek@d3s.mff.cuni.cz
https://tomasp.net | @tomaspetricek
https://d3s.mff.cuni.cz/teaching/nprg077

Charles University