

Saarland University  
Computer Science Department  
Programming Systems Lab, Software Engineering Chair  
Seminar: Advanced Functional Programming

# Dynamic Typing in a Statically Typed Language

[M. Abadi, L. Cardelli, B. Pierce, G. Plotkin]

Matthias Berg  
Advisor: Andreas Rossberg

March 22, 2006

# Outline

## Motivation

## Sketch

## $\lambda$ -calculus with dynamic and typecase

Syntax

Typing rules

Evaluation

Example

## Pattern Variables

## Extensions

Polymorphism

Higher-Order Pattern Variables

Subtyping

## Abstract Data Types

## Example Languages

## References

# Motivation

What is dynamic typing?

- ▶ some type checks are performed at run-time
- ▶ language remains statically typed

# Motivation

What is dynamic typing?

- ▶ some type checks are performed at run-time
- ▶ language remains statically typed

What is it good for?

- ▶ Save storing and loading of values (pickling) and inter-process communication:
- ▶ build heterogeneous data structures such as lists without having polymorphism
- ▶ type-dependent functions such as `eval` and `print`

# Outline

Motivation

## Sketch

$\lambda$ -calculus with dynamic and typecase

Syntax

Typing rules

Evaluation

Example

Pattern Variables

Extensions

Polymorphism

Higher-Order Pattern Variables

Subtyping

Abstract Data Types

Example Languages

References

# Sketch

bundle values with their types

- ▶ To bundle value  $v$  with its type  $\tau$ , write `dynamic( $v:\tau$ )`
- ▶ such bundles are of type `Dyn`
- ▶ this allows uniform handling (All bundles have the same type)
- ▶ use expressions like “save  $e$ ” and “load  $e$ ” to handle persistent data
- ▶ run-time verification of dynamic values

# Sketch

bundle values with their types

- ▶ To bundle value  $v$  with its type  $\tau$ , write `dynamic(v: $\tau$ )`
- ▶ such bundles are of type `Dyn`
- ▶ this allows uniform handling (All bundles have the same type)
- ▶ use expressions like “save  $e$ ” and “load  $e$ ” to handle persistent data
- ▶ run-time verification of dynamic values

Use a typecase-construct to inspect the type tag of dynamics:

- ▶ provides type-dependent branching
- ▶  `$\lambda e:\text{Dyn} . \text{typecase } e \text{ of } x:\text{Nat} . x + 1 \text{ else } 0$`   
checks whether its argument represents a number.

# Outline

Motivation

Sketch

$\lambda$ -calculus with dynamic and typecase

Syntax

Typing rules

Evaluation

Example

Pattern Variables

Extensions

Polymorphism

Higher-Order Pattern Variables

Subtyping

Abstract Data Types

Example Languages

References



# $\lambda$ -calculus with dynamic and typecase: Syntax & Typing

$$\tau \in Typ ::= X \mid \tau \rightarrow \tau$$

| Dyn

$$e \in Exp ::= x \mid \lambda x:\tau . e \mid e e$$

| dynamic( $e:\tau$ ) | typecase  $e$  of  $x:P . e$  else  $e$

where  $P$  is a pattern (here, just a type).

# $\lambda$ -calculus with dynamic and typecase: Syntax & Typing

$$\tau \in Typ ::= X \mid \tau \rightarrow \tau$$
$$\mid \text{Dyn}$$
$$e \in Exp ::= x \mid \lambda x:\tau . e \mid e e$$
$$\mid \text{dynamic}(e:\tau) \mid \text{typecase } e \text{ of } x:P . e \text{ else } e$$

where  $P$  is a pattern (here, just a type).

$$\Gamma \vdash e : \tau$$

---

$$\Gamma \vdash \text{dynamic}(e:\tau) : \text{Dyn}$$
$$\Gamma \vdash e_1 : \text{Dyn} \quad \Gamma, x:P \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau$$

---

$$\Gamma \vdash \text{typecase } e_1 \text{ of } x:P . e_2 \text{ else } e_3 : \tau$$

# $\lambda$ -calculus with dynamic and typecase: Evaluation

$$\frac{e \Rightarrow v}{\text{dynamic}(e:\tau) \Rightarrow \text{dynamic}(v:\tau)}$$

$$\frac{e_1 \Rightarrow \text{dynamic}(v_1:\tau) \quad e_2[x := v_1] \Rightarrow v_2}{\text{typecase } e_1 \text{ of } x:P . e_2 \text{ else } e_3 \Rightarrow v_2} \quad \tau = P$$

$$\frac{e_1 \Rightarrow \text{dynamic}(v_1:\tau) \quad e_3 \Rightarrow v_3}{\text{typecase } e_1 \text{ of } x:P . e_2 \text{ else } e_3 \Rightarrow v_3} \quad \tau \neq P$$

## $\lambda$ -calculus with dynamic and typecase: Example

A function which, given two dynamic values, applies the first to the second:

```
 $\lambda df:\text{Dyn} . \lambda dx:\text{Dyn} .$   
  typecase  $df$  of  
     $f:\text{Nat} \rightarrow \text{Nat} .$   
      typecase  $dx$  of  
         $x:\text{Nat} .$   
           $f(x)$   
        else 0  
      else 0
```

## $\lambda$ -calculus with dynamic and typecase: Example

A function which, given two dynamic values, applies the first to the second:

```
 $\lambda df:\text{Dyn} . \lambda dx:\text{Dyn} .$   
  typecase  $df$  of  
     $f:\text{Nat} \rightarrow \text{Nat} .$   
      typecase  $dx$  of  
         $x:\text{Nat} .$   
           $f(x)$   
        else 0  
      else 0
```

How to write such a function, which works with arbitrary types and not only Nat?

# Outline

Motivation

Sketch

$\lambda$ -calculus with dynamic and typecase

Syntax

Typing rules

Evaluation

Example

## Pattern Variables

Extensions

Polymorphism

Higher-Order Pattern Variables

Subtyping

Abstract Data Types

Example Languages

References

## Pattern Variables

Allow the pattern in the typecase-construct to contain pattern variables.

- ▶ pattern variables match parts of the type tag of the dynamic argument.
- ▶ pattern  $U \rightarrow V$  with pattern variables  $U$  and  $V$  matches any functional type.
- ▶ A match binds  $U$  to the argument type and  $V$  to the result type of the function.

## Pattern Variables

Allow the pattern in the typecase-construct to contain pattern variables.

- ▶ pattern variables match parts of the type tag of the dynamic argument.
- ▶ pattern  $U \rightarrow V$  with pattern variables  $U$  and  $V$  matches any functional type.
- ▶ A match binds  $U$  to the argument type and  $V$  to the result type of the function.

```
 $\lambda df:\text{Dyn} . \lambda dx:\text{Dyn} .$   
  typecase  $df$  of  
     $\{U, V\} f:U \rightarrow V .$   
      typecase  $dx$  of  
         $\{x:U .$   
          dynamic( $f(x):V$ )  
        else dynamic(...)  
      else dynamic(...)
```



# Outline

Motivation

Sketch

$\lambda$ -calculus with dynamic and typecase

Syntax

Typing rules

Evaluation

Example

Pattern Variables

**Extensions**

Polymorphism

Higher-Order Pattern Variables

Subtyping

Abstract Data Types

Example Languages

References

## Extensions: Polymorphism

Polymorphism is modelled by functions which take types as argument (System F [Gir71]):

- ▶ written  $\lambda X . e$
- ▶ polymorphic identity function:  $\lambda X . \lambda x:X . x$
- ▶ has type  $\forall X . X \rightarrow X$
- ▶  $(\lambda X . \lambda x:X . x) [\text{Nat}] \Rightarrow \lambda x:\text{Nat} . x$

## Extensions: Polymorphism

Polymorphism is modelled by functions which take types as argument (System F [Gir71]):

- ▶ written  $\lambda X . e$
- ▶ polymorphic identity function:  $\lambda X . \lambda x:X . x$
- ▶ has type  $\forall X . X \rightarrow X$
- ▶  $(\lambda X . \lambda x:X . x) [\text{Nat}] \Rightarrow \lambda x:\text{Nat} . x$

Integration into previous formulation of dynamic types is straightforward:

- ▶ allow  $\forall$ -quantified type variables in patterns
- ▶  $\lambda df:\text{Dyn} .$

    typecase  $df$  of

        {  $f:\forall X . \text{List } X \rightarrow \text{List } X .$

$\lambda Y . \lambda x:\text{List } Y . f [Y] (\text{reverse } [Y] x)$

    else ...

# Extensions: Higher-Order Pattern Variables

First-order pattern variables are not expressive enough:

- ▶ consider previous function which applies a dynamic value to another.
- ▶ The types  $\forall X . X \rightarrow X$  and  $\forall X . \text{List } X \rightarrow \text{List } X$  are incompatible, i.e. there is no common pattern.
- ▶ typecase *df* of  
     $\{V\} f: \forall X . V \rightarrow V .$   
    dynamic(*f* [Nat]:  $V \rightarrow V$ )  
    else ...
- ▶ Reason: argument and result types contain  $\forall$ -quantified variables

# Extensions: Higher-Order Pattern Variables

Solution: Allow higher-order pattern variables

- ▶ typecase  $df$  of  
 $\{F\} f:\forall X . F X \rightarrow F X .$   
...
- ▶  $F$  is a second-order pattern variable.
- ▶ A match binds it to a function mapping types to types.
  - ▶ e.g.  $\lambda X . X$  or  $\lambda X . \text{List } X$

## Extensions: Subtyping

A type tag  $T$  should match a pattern  $P$ , if  $T \leq P$ .

- ▶ typecase dynamic(5:Nat) of  
     $x:\text{Int} . \dots$   
    else ...
- ▶ matches, since  $\text{Nat} \leq \text{Int}$

## Extensions: Subtyping

A type tag  $T$  should match a pattern  $P$ , if  $T \leq P$ .

- ▶ typecase dynamic(5:Nat) of  
     $x:\text{Int} . \dots$   
    else ...
- ▶ matches, since  $\text{Nat} \leq \text{Int}$
- ▶ Problem:  $\text{Int} \rightarrow \text{Nat}$  matches pattern  $V \rightarrow V$ , but how?
- ▶  $\text{Int} \rightarrow \text{Int}$  and  $\text{Nat} \rightarrow \text{Nat}$  are both instances of  $V \rightarrow V$  and supertypes of  $\text{Int} \rightarrow \text{Nat}$ .

## Extensions: Subtyping

A type tag  $T$  should match a pattern  $P$ , if  $T \leq P$ .

- ▶ typecase dynamic(5:Nat) of  
     $x:\text{Int} \dots$   
    else ...
- ▶ matches, since  $\text{Nat} \leq \text{Int}$
- ▶ Problem:  $\text{Int} \rightarrow \text{Nat}$  matches pattern  $V \rightarrow V$ , but how?
- ▶  $\text{Int} \rightarrow \text{Int}$  and  $\text{Nat} \rightarrow \text{Nat}$  are both instances of  $V \rightarrow V$  and supertypes of  $\text{Int} \rightarrow \text{Nat}$ .
- ▶ Solutions:
  - a. Linear patterns (each pattern variable occurs at most once)



## Extensions: Subtyping

A type tag  $T$  should match a pattern  $P$ , if  $T \leq P$ .

- ▶ typecase dynamic(5:Nat) of  
   $x:\text{Int} . \dots$   
  else ...
- ▶ matches, since  $\text{Nat} \leq \text{Int}$
- ▶ Problem:  $\text{Int} \rightarrow \text{Nat}$  matches pattern  $V \rightarrow V$ , but how?
- ▶  $\text{Int} \rightarrow \text{Int}$  and  $\text{Nat} \rightarrow \text{Nat}$  are both instances of  $V \rightarrow V$  and supertypes of  $\text{Int} \rightarrow \text{Nat}$ .
- ▶ Solutions:
  - Linear patterns (each pattern variable occurs at most once)
  - Add subtyping constraints and perform exact matching. check constraints after matching.
    - ▶ typecase  $dx$  of  
   $\{V \leq \text{Int}\} x:V . \dots$   
  else ...

# Outline

Motivation

Sketch

$\lambda$ -calculus with dynamic and typecase

Syntax

Typing rules

Evaluation

Example

Pattern Variables

Extensions

Polymorphism

Higher-Order Pattern Variables

Subtyping

**Abstract Data Types**

Example Languages

References

# Abstract Data Types

Problem: typecase destroys parametricity and type abstraction:

- ▶ Dynamically, abstract types are just their representation types.
- ▶ Therefore, typecase can expose representation types.
- ▶ `abstype Number = Int`  
    `implementation : ...`
- ▶ `λdn:Dyn .`  
    `typecase dn of`  
        `{ } n:Int .`  
        `"implementation uses integers"`  
    `else "implementation unknown"`

# Abstract Data Types

Problem: typecase destroys parametricity and type abstraction:

- ▶ Dynamically, abstract types are just their representation types.
- ▶ Therefore, typecase can expose representation types.
- ▶ `abstype Number = Int`  
`implementation : ...`
- ▶ `λdn:Dyn .`  
`typecase dn of`  
`{ } n:Int .`  
`"implementation uses integers"`  
`else "implementation unknown"`
- ▶ Solution: Generate new type names dynamically [Ros03, Ber04]

# Outline

Motivation

Sketch

$\lambda$ -calculus with dynamic and typecase

Syntax

Typing rules

Evaluation

Example

Pattern Variables

Extensions

Polymorphism

Higher-Order Pattern Variables

Subtyping

Abstract Data Types

Example Languages

References

# Example Languages

- ▶ Mercury [HCSJ96]
  - ▶ type `univ`
  - ▶ predicates `type_to_univ` and `univ_to_type`
- ▶ GHC [MPO02]
  - ▶ type `Dynamic`, deals with monomorphic types only
  - ▶ type class `Typeable` of types with known representation
    - ▶ `toDyn`  $:: \text{Typeable } a \Rightarrow a \rightarrow \text{Dynamic}$
    - ▶ `fromDynamic`  $:: \text{Typeable } a \Rightarrow \text{Dynamic} \rightarrow \text{Maybe } a$
- ▶ Clean [Pil97]
  - ▶ type `Dynamic`
  - ▶ pattern matching including first-order pattern variables and polymorphism
- ▶ Alice [Ali05]
  - ▶ uses packages
  - ▶ `dynamic` and `typecase` correspond to `pack` and `unpack`
  - ▶ module subtyping rules apply to `unpack` type check

# Outline

Motivation

Sketch

$\lambda$ -calculus with dynamic and typecase

Syntax

Typing rules

Evaluation

Example

Pattern Variables

Extensions

Polymorphism

Higher-Order Pattern Variables





Subtyping

Abstract Data Types

Example Languages

References

# References I

-  Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin.  
Dynamic typing in a statically typed language.  
*ACM Trans. Program. Lang. Syst.*, 13(2):237–268, 1991.
-  Martín Abadi, Luca Cardelli, Benjamin C. Pierce, and Didier Rémy.  
Dynamic typing in polymorphic languages.  
*Journal of Functional Programming*, 5(1):111–130, 1995.
-  Alice Team.  
*The Alice System*.  
Programming System Lab, Universität des Saarlandes,  
<http://www.ps.uni-sb.de/alice/>, 2005.
-  Matthias Berg.  
Polymorphic lambda calculus with dynamic types.  
Bachelor's thesis, Programming Systems Lab, Saarland University,  
October 2004.



# References II



Jean-Yves Girard.

Une extension de l'interprétation de Gödel à l'analysis, et son application à l'élimination des coupures dans l'analysis et la théorie des types.

In J. E. Fenstad, editor, *Proceedings 2nd Scandinavian Logic Symposium*. North-Holland, 1971.



F. Henderson, T. Conway, Z. Somogyi, and D. Jeffery.

*The Mercury language reference manual*.

University of Melbourne, <http://www.cs.mu.oz.au/mercury>, 1996.



Simon Marlow, Simon Peyton Jones, and Others.

*The Glasgow Haskell Compiler*.

University of Glasgow, <http://www.haskell.org/ghc/>, 2002.

## References III



Benjamin C. Pierce.

*Types and Programming Languages.*

MIT Press, February 2002.



Marco Pil.

First class file i/o.

In *IFL '96: Selected Papers from the 8th International Workshop on Implementation of Functional Languages*, pages 233–246, London, UK, 1997. Springer-Verlag.



Andreas Rossberg.

Generativity and dynamic opacity for abstract types.

In Dale Miller, editor, *Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, Uppsala, Sweden, August 2003. ACM Press.