

Type Dynamic

Stephen Chang

2/26/2010

Dynamic Typing in a Statically Typed Language [1]

```
@article{Abadi1991Dynamic,  
  author    = {Martin Abadi, Luca Cardelli, Benjamin C. Pierce, and Gordon D. Plotkin},  
  title     = {Dynamic Typing in a Statically Typed Language},  
  journal   = {ACM Trans. on Programming Languages and Systems (TOPLAS)},  
  volume    = {13},  
  number    = {2},  
  year      = {1991},  
  pages     = {237-268},  
}
```

Abstract

Statically typed programming languages allow earlier error checking, better enforcement of disciplined programming styles, and the generation of more efficient object code than languages where all type consistency checks are performed at run time. However, even in statically typed languages, there is often the need to deal with data whose type cannot be determined at compile time. To handle such situations safely, we propose to add a type **Dynamic** whose values are pairs of a value v and a type tag T , where v has the type denoted by T . Instances of **Dynamic** are built with an explicit tagging construct and inspected with a type safe **typecase** construct. This paper explores the syntax, operational semantics, and denotational semantics of a simple language that includes the type **Dynamic**. We give examples of how dynamically typed values can be used in programming. Then we discuss an operational semantics for our language and obtain a soundness theorem. We present two formulations of the denotational semantics of this language and relate them to the operational semantics. Finally, we consider the implications of polymorphism and some implementation issues.

Summary

A **Dynamic** value is essentially an infinite disjoint union. Previous languages such as Simula-67, CLU, Cedar/Mesa, Modula-2+, and Modula-3 have similar constructs but the behavior of the construct has never been formally specified. The main contribution of this paper is a formal specification of **Dynamic** values.

The authors add **Dynamic** values to the simply-typed lambda calculus, along with a **dynamic** construct (lowercase “d”) for creating **Dynamic** values, and a **typecase** construct for inspecting them. **Dynamic** values are a tuple (v, T) , where v is a value and T is its type. **typecase** statements consist of a series of patterns of the form $(\vec{X})(x : T) e$ where \vec{X} is a set of type variables, x is a variable expression, and T is a type expression that may contain the type variables in \vec{X} . If the **Dynamic** value (v, U) matches a pattern, then the result of the **typecase** expression is the result of evaluating e in an environment where the variable x is bound to v . A matching is defined such that any type variables in T can be used to represent any subexpression in the type U . If the **Dynamic** value matches more than one pattern, then the first match is used. One thing to note is that the set of types that **Dynamic** must handle cannot be computed statically because the **dynamic** constructor can create new types at run time. For example:

```
\dx:Dynamic.
  typecase dx of
    (X)(x:X) (dynamic (x,x) (X,X))
  else dx
end
```

The authors also give typechecking and evaluation rules for their language and prove the soundness of the type system. The authors also give a denotational semantics of their language and show that if an expression evaluates to a value in the operational semantics, then the meaning of the expression is equal to the meaning of the value in the domain they construct. In addition, the authors prove type soundness using the denotational semantics as well, where the main difficulty is in defining the meaning of **Dynamic**. To aid them, the authors use the ideal model of types from MacQueen, Plotkin, and Sethi [5]. The authors conclude by briefly describing several possible extensions to their language, such as polymorphism and abstract data types, which they elaborate on in a subsequent paper [2].

Situations where Dynamic can be used:

- multi-language programs
- accessing persistent data (picking)
- inter-process communication, RPC
- giving a type to functions like **eval** or **print** function where the type of the input or output is not known
- the authors also show that **Dynamic** can be used to implement a fixpoint operator in the simply-typed lambda calculus by hiding certain expressions inside a **Dynamic** value

Dynamic Typing in Polymorphic Languages [2]

```
@article{Abadi1995DynamicPolymorphic,
  author    = {Martin Abadi, Luca Cardelli, Benjamin C. Pierce, and Didier Remy},
  title     = {Dynamic Typing in Polymorphic Languages},
```

```

journal    = {Journal of Functional Programming},
volume     = {5},
number     = {1},
year       = {1995},
pages      = {111-130},
}

```

Abstract

There are situations in programming where some dynamic typing is needed, even in the presence of advanced static type systems. We investigate the interplay of dynamic types with other advanced type constructions, discussing their integration into languages with explicit polymorphism (in the style of system F), implicit polymorphism (in the style of ML), abstract data types, and subtyping.

Summary

The main contribution of this work is incorporating polymorphism into a language with **Dynamic** values. The authors first add explicit polymorphism to their language but encounter a problem where some **Dynamic** values cannot be matched uniquely using **typecase**. The authors work around the problem by restricting the allowable **typecase** patterns. The authors then show that their solution can also be used to add abstract data types and subtyping to their language. They also explore implicit polymorphism and compare their work to that of Leroy and Mauny [4].

Adding explicit polymorphism to a language with **Dynamic** requires allowing higher-order variables in a **typecase** pattern. Instead of being able to represent only type expressions, the pattern type variables must be allowed to represent abstractions over type expressions (ie - type operators). The authors add this new **typecase** construct to System F_ω . However, allowing unconstrained type operators leads to the problem where some **typecase** patterns may not have unique matches for some types. For example, the pattern $F(\text{Int})$, where F is a type operator, can match the type **Int** by assigning F to be either $\lambda X.X$, $\lambda X.\text{Int}$. To address this problem, the only allowable patterns are those that can uniquely match every type in at most one way (the authors call this property definiteness). Requiring definiteness introduces a new problem of how to incorporate the requirement into the typechecker, since there is not even a decidable algorithm to match higher-order patterns to types. The authors address the second problem by restricting their language to second-order polymorphism (System F) and they make a reasonable argument that this restriction satisfies their requirements, although they do not give an explicit matching algorithm.

Dynamics in ML [4]

```

@inproceedings{Leroy1991Dynamics,
  author    = {Xavier Leroy and Michel Mauny},
  title     = {Dynamics in ML},
  booktitle = {FPCA},
  year      = {1991},
  pages     = {406-426},
}

```

Abstract

Objects with dynamic types allow the integration of operations that essentially require run-time type-checking into statically-typed languages. This paper presents two extensions of the ML language with dynamics, based on what has been done in the CAML implementation of ML, and discusses their usefulness. The main novelty of this work is the combination of dynamics with polymorphism.

Summary

Fine-Grained Interoperability Through Mirrors and Contracts [3]

```
@inproceedings{Gray2005FineGrained,  
  author    = {Kathryn E. Gray, Robert Bruce Findler, and Matthew Flatt},  
  title     = {Fine-grained interoperability through mirrors and contracts},  
  booktitle = {OOPSLA},  
  year      = {2005},  
  pages     = {231-245},  
}
```

Abstract

As a value flows across the boundary between interoperating languages, it must be checked and converted to fit the types and representations of the target language. For simple forms of data, the checks and coercions can be immediate; for higher order data, such as functions and objects, some must be delayed until the value is used in a particular way. Typically, these coercions and checks are implemented by an ad-hoc mixture of wrappers, reflection, and dynamic predicates. We observe that 1) the wrapper and reflection operations fit the profile of mirrors, 2) the checks correspond to contracts, and 3) the timing and shape of mirror operations coincide with the timing and shape of contract operations. Based on these insights, we present a new model of interoperability that builds on the ideas of mirrors and contracts, and we describe an interoperable implementation of Java and Scheme that is guided by the model.

Summary

References

- [1] ABADI, M., CARDELLI, L., PIERCE, B. C., AND PLOTKIN, G. D. Dynamic typing in a statically typed language. *ACM Trans. Program. Lang. Syst.* 13, 2 (1991), 237–268.
- [2] ABADI, M., CARDELLI, L., PIERCE, B. C., AND RÉMY, D. Dynamic typing in polymorphic languages. *J. Funct. Program.* 5, 1 (1995), 111–130.
- [3] GRAY, K. E., FINDLER, R. B., AND FLATT, M. Fine-grained interoperability through mirrors and contracts. In *OOPSLA* (2005), pp. 231–245.
- [4] LEROY, X., AND MAUNY, M. Dynamics in ml. In *FPCA* (1991), pp. 406–426.

- [5] MACQUEEN, D. B., PLOTKIN, G. D., AND SETHI, R. An ideal model for recursive polymorphic types. *Information and Control* 71, 1/2 (1986), 95–130.