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 a language with `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 type variables in T can be used to represent subexpressions in the type U. If a `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, in the function below, a new product type is created:

```
\dy:Dynamic.
  typecase dy of
    (Y)(y:Y) (dynamic (y,y) Y x Y)
    else dx
  end
```

The authors give typechecking and evaluation rules for their language and prove the soundness of the type system. The authors also give a denotational semantics for 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 denotational semantics. 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 address the difficulty, the authors use the ideal model of types from MacQueen, Plotkin, and Sethi [7]. 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 where one language is dynamically typed
- accessing persistent data (pickling)
- 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 [6].

Adding explicit polymorphism to a language with `Dynamic` requires allowing higher-order type 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(`Int`), where F is a type operator, can match the type `Int` by assigning F to be either $\Lambda X.X$, $\Lambda X.$`Int`. To address this problem, the allowed patterns are constrained to 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 [6]

```
@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

The contribution of this work is adding `Dynamic` values to a language with ML-style (implicit) polymorphism. The authors present two possible ways to add `Dynamic` values, the first of which is fully implemented in CAML, and the second of which is prototyped in CAML. The authors also give a calculus for their new language and they give a soundness proof for the type system. The authors also discuss various implementation issues they faced.

One difference between the first implementation in this paper and the caluclus of Abadi et. al [1, 2] is that in the ML implementation, the parameter to the `dynamic` creation construct does not need to be explicitly annotated with a type (because of type inference in ML). However, this places restrictions on when and what `Dynamic` values are allowed to be created. Specifically, the parameter to `dynamic` can only be values whose types are closed. Polymorphic types are still ok, so (`dynamic \x.x`) is acceptable, but the dynamic produced by `\x.(dynamic x)` is not ok because the type of `x` is unknown at compile time. The typechecker must be modifed to reject expressions like `\x.(dynamic x)` because the ML typechecker would infer a principal type of $\alpha \rightarrow$ `Dynamic`, which is not valid. It is interesting to note that the type system with `Dynamic` added no longer has the principal type property.

Inspection of `Dynamic` values is also different in the (first) language presented by the paper. The authors do not use a separate elimination construct like Abadi, et al.'s `typecase`, but instead use ML's existing pattern matching features. Also, the patterns in this paper are less expressive than that of [2], since this paper only allows the usual ML-style types, where type quantifiers are only allowed in the top level (ie - an ML type scheme). A type pattern in this paper will match any type in a `Dynamic` value that is more general than the pattern type. For example, a pattern `dynamic(x:int list)` will match a `Dynamic` value that contains the empty list (which has type $\forall\alpha.\alpha$ `list`) and a pattern `dynamic(f :` $\alpha \rightarrow \alpha$`)` will match any `Dynamic` value with internal type that is $\alpha \rightarrow \alpha$ or something more general like $\alpha \rightarrow \beta$. However, it's important to note that the type variables in the pattern are not normal pattern variables that can be instantiated during pattern matching (this is different from patterns in the `typecase` construct of [2]). So a pattern `dynamic(`$x : \alpha$ `list)` will only match the empty list. Therefore, in a case expression on a `Dynamic` value, the more general type patterns must come first, since they will match fewer types.

The authors acknowledge the limitations of their first type system that requires closedness of `dynamic` parameters. For example, in a `print` function, the first language cannot express a pattern that matches any product type. To address this limitation, he authors propose extend their language and type system to allow existential types.

Fine-Grained Interoperability Through Mirrors and Contracts [5]

```
@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

Existing frameworks for allowing different languages to interoperate, like Microsoft's COM, often only allow "course-grained" interoperability because they require that all languages use a universal format when exchanging data. The main contribution of this paper is to show that two specific languages (Scheme and Java) can interoperate on a "fine-grained" basis, meaning that by extending each language in specific ways, the languages can retain use of many of their native features. Mirrors [3] and contracts [4] and the addition of a `dynamic` construct in Java are the key to achieving this "fine-grained" interoperability.

The primary interoperability vehicle introduced is dynamic method calls in Java. Specifically, a `dynamic` keyword is added that can be used in place of a type. The `dynamic` construct is implemented using mirrors [3]. When a method call is made to an object of type `dynamic`, the existence of the method is not checked until run time. This differs from the standard Java behavior, where methods are statically associated with classes. However, now run-time checks are required to ensure that dynamic method calls accept arguments of the correct type and return a value of the expected type. The authors uses PLT-Scheme-style contracts [4] in Java to check that the correct types are used (and Java's type system is able to infer such contracts automatically from the surrounding context). In this way, Java's type safety is preserved when interoperating in Scheme. In addition, Scheme's contracts can also be preserved in Java programs. The `dynamic` construct also enables other Scheme-like features such as functions as values, which can be implemented using `dynamic` static methods.

The `dynamic` construct introduced in this paper is similar to the `Dynamic` values originally introduced by Abadi, et al [1] except Abadi's explicit `typecase` construct has been replaced with object-oriented method dispatch.

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] Bracha, G., and Ungar, D. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *OOPSLA* (2004), pp. 331–344.

[4] Findler, R. B., and Felleisen, M. Contracts for higher-order functions. In *ICFP* (2002), pp. 48–59.

[5] Gray, K. E., Findler, R. B., and Flatt, M. Fine-grained interoperability through mirrors and contracts. In *OOPSLA* (2005), pp. 231–245.

[6] Leroy, X., and Mauny, M. Dynamics in ml. In *FPCA* (1991), pp. 406–426.

[7] 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.