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

There are some situations that demand run-time type information in a statically-typed language. The authors address this issue by introducing `Dynamic` values to the simply typed lambda calculus. A `Dynamic` value is a pair that contains a value and a type tag for that value. `Dynamic` values are inspected using a `typecase` construct, which uses patterns. The authors give both an operational

semantics and a denotational semantics for their language and use both to prove that their type system is sound.

Other languages have constructs similar to `Dynamic` but this paper is the first to present a formal description of a language with `Dynamic` values and the first give a soundness proof for their type system that includes `Dynamic`. The authors briefly mention several possible extensions to their language such as polymorphism but to not go into much detail.

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 authors add polymorphism to their language from their previous paper [1]. To support polymorphism, second-order pattern variables are used in the `typecase` construct (these variables can be instantiated to type operators). Their new language is equivalent to (a restricted form of) System F_ω extended with `Dynamic`. The authors present two flavors of polymorphism, explicit and implicit.

The authors do not give a specification for their language with explicit polymorphism. No typing rules or evaluation rules are given either. The authors give typechecking and evaluation rules for their language with implicit polymorphism but they do not prove soundness for their type system.

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

Leroy and Mauny give two extensions to ML that use `Dynamic`. They integrate inspection of `Dynamic` values with ML's pattern matching. The first extension requires that `Dynamic` values only contain values with closed types. The authors give typechecking and evaluation rules for this language and prove soundness for their type system. They also explain how to modify the unification algorithm and discuss various implementation issues (the authors implement the first extension in CAML).

The first extension is unable to capture some patterns such as "any pair" or "any function" so the authors present a second extension that allows existential variables in patterns. The authors give new typing and evaluation rules for the second extension, and give a modified unification algorithm, but they do not prove soundness for this type system. The authors have a working prototype for the second extension and they also discuss various implementation issues.

The language in this paper (second extension) differs from Abadi, et al.'s language with implicit polymorphism in several ways. The main difference is that Leroy and Mauny's (LM) extension uses existential pattern variables, while Abadi, et al.'s (ACPP) language uses second-order pattern variables. ACPP's language is more expressive since it allows arbitrary dependencies between pattern variables, while LM only allows linear dependencies between quantified variables in their patterns. However, ACPP must impose ad-hoc restrictions on their language to address various issues that arise with second-order pattern variables, while LM has a very simple interpretation in first-order logic.

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.