Type Dynamic

Stephen Chang

HOPL, Spring 2010

Dynamic Typing in a Statically Typed Language

```
@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}
}
```

Summary Even in the most expressive statically-typed languages, there are always programs that will require run-time type information. An example of such a program is the `eval` function, whose ouput cannot be assigned a type at compile time. Abadi et al. introduce `Dynamic` values to solve this problem. A `Dynamic` value is a pair that contains a value and a type tag for that value. The authors also introduce the `typecase` construct, which uses patterns to inspect `Dynamic` values. The authors formalize `Dynamic` values as an extension to the simply-typed lambda calculus and give both an operational semantics and a denotational semantics for their language. The authors independently prove the soundness of their type system using each semantics.

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 is also the first to give a soundness proof for a type system that includes the `Dynamic` type. In their conclusion, the authors briefly mention several possible extensions to their language such as polymorphism.

Dynamic Typing in Polymorphic Languages

```
@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}
}
```

Summary The authors add polymorphism to the language of their previous paper. To support polymorphism, second-order pattern variables must be added to the `typecase` construct (these variables can be instantiated to type operators). The 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.

Even though the authors describe languages with both explicit and implicit polymorphism, the authors only present typing and evaluation rules for the language with implicit polymorphism. Also, unlike in their previous paper, the authors do not prove soundness for any of the type systems in this paper.

Dynamics in ML

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

Summary Leroy and Mauny present two extensions to ML that use `Dynamic`. Both extensions integrate inspection of `Dynamic` values with ML's pattern matching. The first extension requires `Dynamic` values to only contain values with closed types. The authors devise 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 concerning the first extension.

The first extension is unable to capture some patterns such as "any pair" or "any function" so the authors develop a second extension that allows existential variables in patterns. The authors give new typing and evaluation rules for the second extension and present 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 second language in this paper differs from Abadi et al.'s language with implicit polymorphism in several ways. The main difference is that Leroy and Mauny's extension (LM) uses existential pattern variables, while Abadi et al.'s language (ACPP) uses second-order pattern variables. ACPP 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 simple interpretation in first-order logic.

Fine-Grained Interoperability Through Mirrors and Contracts

```
@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}
}
```

Summary Existing interoperability frameworks such as 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 [1], contracts [2], and the addition of a `dynamic` type to Java are the keys to achieving this "fine-grained" interoperability.

The primary interoperability vehicle introduced is dynamic method calls in Java. Specifically, the authors add a `dynamic` keyword that can be used in place of a type. The `dynamic` construct is implemented using mirrors [1]. When a method of type `dynamic` is called, 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 [2] in Java to check that the correct types are used (Java's type system is able to automatically infer such contracts 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. except Abadi et al.'s explicit `typecase` construct has been replaced with object-oriented method dispatch.

References

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

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