

Use-Based Inference of Polymorphic References

Dave King
John Hannan

8th Symposium on Trends in Functional Programming
April 2nd, 2007

The Problem

- ▶ There are a lot of type systems which combine polymorphism and mutable state. (Tofte 90, Leroy 92, Wright 92, Wright 95, type variable strength from old SML/NJ)
- ▶ Nearly all integrate let-polymorphism with a store that holds only monomorphic values.
- ▶ We must be careful how we polymorphic values in the store for safety reasons:

```
let r = ref fn (x => x) in
  r := (fn n => n + 1);
  !r true
```

Our Goal

- ▶ A type system that lets us put polymorphic data in the store, so that we can later take it out and use it polymorphically.
- ▶ We also should be able to reassign polymorphic values in the store.
 - ▶ Let f be a polymorphic function and f' be an update of that function with the same type. We should be able to seamlessly update a reference pointing to f with the new value of f' .

Warning: Unsoundness Ahead

```
let r = ref fn (x => x) in
  r := (fn n => n + 1);
  !r true
```

- ▶ To restore soundness, systems usually restrict r from being assigned the type schema $\forall\alpha.\alpha\rightarrow\alpha$. (Tofte: imperative type variables, Leroy: dangerous type variables, Wright: value restriction)
- ▶ But: if you want to add polymorphism to the store, you need to be able to generalize r .
- ▶ Our approach: watch the behavior of what is *stored in* and *retrieved from* references.

Observation

- ▶ References have two major operations supported on them.
 - ▶ Storage: putting something into the reference cell (creation, assignment)
 - ▶ Retrieval: taking something out (dereference)
- ▶ We use aliases (Smith 00) to watch references after their creation.

Example

```
let Kref = polyref (fn x => fn y => x) in
  (polyderef !Kref) 3 4;
  polyasgn Kref := fn x => fn y => y;
  (polyderef !Kref) true false
```

- ▶ We tag `Kref` with the alias M .
- ▶ Two (polymorphic) types are put into M : $\forall\alpha\beta.\alpha\rightarrow\beta\rightarrow\alpha$ and $\forall\alpha\beta.\alpha\rightarrow\beta\rightarrow\beta$.
- ▶ Two types are taken out of M : $int\rightarrow int\rightarrow int$ and $bool\rightarrow bool\rightarrow bool$.
- ▶ Each value retrieved from reference with cell with alias M is an instance of every value stored in it, so this is a safe program.

An Example Revisited

```
let r = polyref fn (x => x) in
  polyasgn r := (fn n => n + 1);
  (polyderef !r) true
```

- ▶ We tag r with the alias N .
- ▶ Two types are put into N : $\forall\alpha.\alpha\rightarrow\alpha$ and $int\rightarrow int$.
- ▶ One type is taken out of N : $bool\rightarrow bool$
- ▶ As $bool\rightarrow bool$ is not an instance of $int\rightarrow int$, this is an unsafe program.

Some Technical Details

- ▶ References are given type $M \text{ pref}$.
- ▶ Constraint sets C keep track of how programs use polymorphic references.
- ▶ If e has type σ and e' has type $N \text{ pref}$, then:
 - ▶ The expression `polyref e` creates a storage constraint, $\sigma \succ M$.
 - ▶ The assignment `polyasgn $e' := e$` creates a storage constraint, $\sigma \succ N$.
 - ▶ The dereference `polyderef e'` creates a retrieval constraint, $N \succ \tau$, where τ is the type retrieved.
- ▶ Implication judgement: $C \supset c$ – under constraint set C , constraint c is true.
- ▶ A constraint set C is consistent if: for all pairs of storage and retrieval constraints on the same alias in C , $\sigma \succ M$ and $M \succ \tau$, then $\vdash \sigma \succ \tau$.

Alias Polymorphism

- ▶ As described, this system cannot properly deal with functions that may take polymorphic functions as arguments.
- ▶ For example, function f takes a reference to a function which is used inside of f at type $A \rightarrow A$.
- ▶ The most general type of f is $M \text{ pref} \rightarrow \alpha$ under the constraint $M \succ A \rightarrow A$.
- ▶ In order for f to also take a reference to a function of type $B \rightarrow A$ (with B is a subtype of A), we need to also generalize M .
- ▶ Generalized type of f is $\forall M: \{M \succ A \rightarrow A\}. M \text{ pref} \rightarrow \alpha$.
- ▶ We can only give arguments of type $N \text{ pref}$ to this function, where N satisfies the constraint $N \succ A \rightarrow A$.

Technical Changes

- ▶ With constraints inside the polytype, we need to change the polytype instantiate judgement: $C \vdash \sigma \succ \tau$.
- ▶ We can only safely generalize aliases for values.
- ▶ Let ρ be a map from locations l to aliases M .
- ▶ Type judgement for our system: $C; \gamma; \rho \vdash e : \sigma$.

Type Soundness

- ▶ Let μ be a store from locations to values.
- ▶ $\rho \vdash \mu : C$ if for all $l \in \text{dom}(\mu)$ and every retrieval constraint $\rho(l) \succ \tau \in C$, $C; \rho \vdash \mu(l) : \tau$.
- ▶ **Type soundness:** If $\mu \vdash e \hookrightarrow v, \mu'$, $C; \rho \vdash e : \sigma$, and $\rho \vdash \mu : C$, then there is a $\rho' \supseteq \rho$ such that $C; \rho' \vdash v : \sigma$.

Automatic Inference

- ▶ We have designed an algorithm for automatic type inference – details are our technical report (King 07).
- ▶ Critical task: ensuring that during inference, the same references are used in consistent ways.
- ▶ For example,
`let x = ref e in e1; e2`
- ▶ We need to ensure that e_1 and e_2 use x consistently.
- ▶ We use a non-structural subtyping algorithm (Palsberg 95) to check that storage constraints instantiate to retrieval constraints.

Future Work

- ▶ Implementation of this work as an add-on to an existing type system (SML, Objective Caml, Polyglot, etc)
- ▶ More generally, an inference-based Java-like language with polymorphic references to objects.

Any Questions?

`http://www.cse.psu.edu/~dhking`