# A type-preserving store-passing translation for general references


François Pottier


January 26, 2011

INRIA

# Contents

In this talk, I am concerned with a simple question:

> *How to translate a typed calculus equipped with general references down into a typed, pure $\lambda$-calculus?*

By *"general references"*, I mean: mutable memory cells that are dynamically allocated and hold a value of (fixed) arbitrary type.

By *"typed"*, I mean: well-typed programs must not go wrong.

# Store-passing translations

I am looking for a *store-passing translation*.

The idea is that the store should become an argument and a result of every computation.

> *"Commands can be considered as functions which transform [the store]." — Strachey, 1967*

This idea was initially *developed*, and is well-understood, in an *untyped* setting.

# The state monad

Moggi (1991) proposed *monads* as a way of structuring (and type-checking) imperative computations.

In particular, the *state monad* implements the store-passing machinery.

Is the state monad a typed store-passing translation? *Yes.*

Does it solve my problem? *No...*

The state monad is a solution to a simpler problem, where *the type s of the store is fixed.* There is just one global reference.

$$M\ a = s \longrightarrow (a, s)$$

$$\begin{aligned}
\text{return} \quad &: \forall a.a \longrightarrow M\ a \\
&= \lambda x.\lambda s.(x, s) \\
\text{bind} \quad &: \forall a.\forall \beta.(M\ a, a \longrightarrow M\ \beta) \longrightarrow M\ \beta \\
&= \lambda(f, g).\lambda s.\text{let } (x, s) = f\ s \text{ in } g\ x\ s
\end{aligned}$$

$$\begin{aligned}
\text{get} \quad &: \forall a.M\ a \\
&= \lambda s.(s, s) \\
\text{put} \quad &: \forall a.a \longrightarrow M\ () \\
&= \lambda x.\lambda s.((), x)
\end{aligned}$$

# A monadic presentation of System F with references

The calculus that I care about extends (say) System F with types for *computations* and for *references:*

$$T ::= \alpha \mid () \mid T \to T \mid (T, T) \mid \forall \alpha.T \mid M\ T \mid \mathit{ref}\ T$$

References are dynamically allocated, are first-class values, and can hold values of any type.

$$\begin{aligned}
\text{return} :&\quad \forall \alpha.\alpha \to M\ \alpha \\
\text{bind} :&\quad \forall \alpha.\forall \beta.(M\ \alpha, \alpha \to M\ \beta) \to M\ \beta \\[1ex]
\text{new} :&\quad \forall \alpha.\alpha \to M\ (\text{ref}\ \alpha) \\
\text{read} :&\quad \forall \alpha.\text{ref}\ \alpha \to M\ \alpha \\
\text{write} :&\quad \forall \alpha.(\text{ref}\ \alpha, \alpha) \to M\ ()
\end{aligned}$$

The problem again is to *find* a typed $\lambda$-calculus that supports an encoding of System $F$ with references, and to *define* this encoding.

# Is this really an open problem?

Is this an open problem?

- *Yes* — to the best of my knowledge, no type-preserving store-passing translation for general references has appeared earlier.

Really?

- *Well* — because a denotational semantics is a store-passing translation, many semanticists have confronted this problem before; solutions are implicit in their work.

In particular, the work by Schwinghammer, Birkedal, Reus and Yang [2009] has been a strong source of inspiration.

Why is it worth studying this problem?

- to explain in terms of *syntax and types* what semanticists have done in terms of mathematical meta-language;
- (perhaps) to offer a more modular approach to the construction of denotational semantic models;
- to discover, in the process, an extension of $F_\omega$ with *rich type-level recursion.*

# Contents

# Evolution in width and depth

*Dynamic memory allocation* and *higher-order store* cause the type of the store to change over time:

- because new cells appear, the store grows *in width;*
- because an older cell can hold a reference to a newer cell, the type of each cell changes (gets more specific) with time: the store evolves *in depth.*

In order to explain how the store evolves, we need *open-ended* descriptions of the store, known as *worlds.*

We need worlds to be open-ended both in *width* and in *depth.* A world should be a function of two parameters that produces a type.

We would like worlds to be *ordered*, so as to form a Kripke frame. The property $w_1 \leq w_2$ would then mean that $w_2$ is a possible evolution of $w_1$.

We would like worlds to support a well-behaved form of *composition*, so that the ordering can be defined simply via the axiom $w_1 \leq w_1 \circ w_2$.

We begin with *fragments* — store descriptions that are open-ended in width.

Fragments can be defined in $F_\omega$ as functions from types to types. They admit an associative notion of concatenation.

```
kind fragment = * -> *

type @ : fragment -> fragment -> fragment =
  \f1 f2 tail. f1 (f2 tail)
```

# Worlds — a tentative definition

Walking in the footsteps of semanticists, we would like worlds to be functions of one parameter — *itself a world* — to fragments.

```
kind world = world -> fragment (* to be revisited *)
```

We would then like to define world composition as follows:

```
type o : world -> world -> world =
  \w1 w2 x. w1 (w2 `o` x) `@` w2 x
```

Wait, wait! We are no longer in $F_\omega$.

We just tried to define a *recursive kind* and a *recursive type function!*

It is not surprising that $F_\omega$ does not fit our purposes — after all, System $F$ with references is not normalizing. But *in which extension of $F_\omega$* do these recursive definitions make sense?

$F_\omega$ has simple (finite) kinds, so that types are strongly normalizing.

Extending it with arbitrary recursive kinds would lead to a calculus where types can diverge and type equality is undecidable.

Fortunately,

- we don't need arbitrary non-terminating type-level computations, only *productive* computations;
- we can use an off-the-shelf system, known as Nakano's system [2000], for determining which computations are productive.

I take Fork ($F_\omega$ with Recursive Kinds) to be a version of $F_\omega$ where Nakano's system replaces the simply-typed $\lambda$-calculus at the kind level.

Thus, Nakano's types and terms become my kinds and types.

Kinds are *co-inductively* defined by:

$$\kappa ::= \star \mid \kappa \longrightarrow \kappa \mid \bullet\kappa$$

with the proviso that *every* infinite path must infinitely *often* *enter* *a* "later" ($\bullet$) *constructor*.

As per Nakano's papers, *subkinding* is a pre-order and additionally validates the following laws:

$$\frac{\kappa_1' \leq \kappa_1 \qquad \kappa_2 \leq \kappa_2'}{\kappa_1 \to \kappa_2 \leq \kappa_1' \to \kappa_2'} \qquad \frac{\kappa \leq \kappa'}{\bullet\kappa \leq \bullet\kappa'} \qquad \kappa \leq \bullet\kappa \qquad \bullet(\kappa_1 \to \kappa_2) \mathrel{\substack{\leq \\ \geq}} \bullet\kappa_1 \to \bullet\kappa_2$$

All of the magic lies in here. Types are ordinary $\lambda$-terms, as in $F_\omega$, and the kind assignment rules are standard.

# Fixed points in Nakano's system

Nakano's system allows deriving $\vdash Y : (\bullet\kappa \longrightarrow \kappa) \longrightarrow \kappa$.

That is, only *contractive functions* have fixed points.

## Properties of Nakano's system

Every well-kinded type admits a *head normal form*, hence (by repeated application of this result) admits a *maximal Böhm tree.*

In other words, *types are productive.*

As a result, *type equality is semi-decidable.*

# Worlds — a correct definition

My *earlier* definition of worlds is illegal in Fork, but can be fixed:

```
kind world = later world -> fragment
```

There is an obvious connection between “later” and the $\frac{1}{2}$ factor used in metric space approaches.

The definition of world composition is *well-kinded* because the recursive occurrence of `o` is used at kind `later (world -> world -> world)`:

```
type o : world -> world -> world =
  \w1 w2 x. w1 (w2 `o` x) `@` w2 x
```

Associativity of composition, *a type equality fact*, is automatically proved by the semi-algorithm in the Fork type-checker:

```
lemma compose_associative:
  forall w1 w2 w3.
  (w1 `o` w2) `o` w3 = w1 `o` (w2 `o` w3)
```

*Quantification over future worlds* is expressed directly in terms of composition, so bounded quantification is not required.

For instance, a value that has type **a** not only in world **x**, but also in *every* possible future world, is denoted by the type **box a x**, where:

```
type box : stype -> stype =
  \a. \x.
    forall y. a (x `o` y)
```

Associativity of composition is required for this to work smoothly.

# And so on...

One can continue in this way and produce about *800 lines of kind/type/term definitions,* lemmas, and comments, culminating in the definitions of the terms that correspond to return, bind, new, read, and write.

They are checked by the Fork type-checker in 0.1 seconds.

# Contents

General references *can be translated* down into pure $\lambda$-calculus in a type-preserving manner.

Although the encoding is somewhat complex, the target calculus is *"just about as simple"* as one might hope, and quite expressive.

One take-home idea?

Recursive types in Fork are not just inert infinite syntax — they are possibly non-terminating *processes* that produce type structure as they go.

# Contents

(Most titles are clickable links to online versions.)


Nakano, H. 2000.
A modality for recursion.
In *IEEE Symposium on Logic in Computer Science (LICS)*.
255–266.

Schwinghammer, J., Birkedal, L., Reus, B., and Yang, H. 2009.
Nested Hoare triples and frame rules for higher-order store.
In *Computer Science Logic*. Lecture Notes in Computer Science, vol. 5771. Springer, 440–454.