A type-preserving store-passing translation for general references

François Pottier

January 26, 2011
Contents

- Introduction
- Technical elements
- Conclusion
- Bibliography
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 λ-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.
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.
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.

\[
M a = s \rightarrow (a, s)
\]

**return** : \( \forall a.a \rightarrow M a \)
= \( \lambda x.\lambda s.(x, s) \)

**bind** : \( \forall a.\forall \beta.(M a, a \rightarrow M \beta) \rightarrow M \beta \)
= \( \lambda (f, g).\lambda s.\text{let } (x, s) = f s \text{ in } g x s \)

**get** : \( \forall a.M a \)
= \( \lambda s.(s, s) \)

**put** : \( \forall a.a \rightarrow M () \)
= \( \lambda x.\lambda s.()(, x) \)
The calculus that I care about extends (say) System F with types for \textit{computations} and for \textit{references}:

\[
T ::= a \mid () \mid T \to T \mid (T, T) \mid \forall a. T \mid M T \mid \text{ref } T
\]

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

\begin{align*}
\text{return } & : \forall a. a \to M a \\
\text{bind } & : \forall a. \forall \beta. (M a, a \to M \beta) \to M \beta \\
\text{new } & : \forall a. a \to M \text{ (ref } a) \\
\text{read } & : \forall a. \text{ref } a \to M a \\
\text{write } & : \forall a. \text{(ref } a, a) \to M ()
\end{align*}
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 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

- Introduction
- Technical elements
- Conclusion
- Bibliography
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.
Towards worlds

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

\[ \text{kind world} = \text{world} \to \text{fragment} \] (* to be revisited *)

We would then like to define world composition as follows:

\[ \text{type o} : \text{world} \to \text{world} \to \text{world} = \] \[ \lambda w1 w2 x. w1 (w2 'o' x) '@' w2 x \]
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:

\[ \kappa ::= \star \mid \kappa \to \kappa \mid \bullet \kappa \]

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:

\[
\begin{align*}
K_1' \leq K_1 & \quad K_2 \leq K_2' \\
K_1 \to K_2 \leq K_1' \to K_2' & \\
K \leq K' & \\
\bullet K \leq \bullet K' & \\
K \leq \bullet K & \\
\bullet (K_1 \to K_2) \leq \bullet K_1 \to \bullet K_2 &
\end{align*}
\]

All of the magic lies in here. Types are ordinary \( \lambda \)-terms, as in \( F_\omega \), and the kind assignment rules are standard.
Nakano’s system allows deriving \( \vdash Y : (\bullet \kappa \to \kappa) \to \kappa \).

That is, only \textit{contractive functions} have fixed points.
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.
My earlier definition of worlds is illegal in Fork, but can be fixed:

\[
\text{kind world} = \text{later world} \rightarrow \text{fragment}
\]

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):

\[
\begin{align*}
type\ o\ :\ world\to\ world\to\ world\ &= \\
\w1\ w2\ x.\ w1\ (w2\ 'o'\ x)\ '@'\ w2\ x
\end{align*}
\]

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

\[
\begin{align*}
\text{lemma compose_associative:} \\
\forall\ w1\ w2\ w3.\quad (w1\ 'o'\ w2)\ 'o'\ w3\ &=\ w1\ 'o'\ (w2\ 'o'\ w3)
\end{align*}
\]
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 \( \text{box } a \ x \), where:

\[
\text{type } \text{box} : \text{stype } \to \text{stype } = \\
\lambda a. \lambda x. \\
\quad \forall y. a \ (x 'o' y)
\]

Associativity of composition is required for this to work smoothly.
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.
General references can be translated down into pure λ-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

- Introduction
- Technical elements
- Conclusion
- Bibliography
A modality for recursion. 
In IEEE Symposium on Logic in Computer Science (LICS). 
255–266.

Nested Hoare triples and frame rules for higher-order store. 
In Computer Science Logic. Lecture Notes in Computer Science, 