A type-preserving store-passing translation for general references

François Pottier

November 2009

INRIA

Why study a store-passing translation?

A denotational semantics of an imperative language can be regarded as a *store-passing translation* into some mathematical meta-language.

In 1967, Strachey [2000] writes:

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

Strachey's semantics of commands is a *store-passing translation.*

Moggi [1991] views the store-passing translation as an instance of the *monadic translation*.

A *monad* is given by a type operator $M : \star \to \star$ together with two operators:

$$
\begin{aligned}
\text{return} &: \forall a. a \to M\ a \\
\text{bind} &: \forall a\, b. M\ a \to (a \to M\ b) \to M\ b
\end{aligned}
$$

which must satisfy the three monad laws (omitted).

For a fixed store type s, the type operator $(\mathsf{State}\ s)$ is a monad, known as the *state monad*.

$$\mathsf{State}\ s\ a = s \longrightarrow (a, s)$$

$$\begin{aligned}
\mathsf{return} &: \forall s\, a.\, a \longrightarrow \mathsf{State}\ s\ a \\
&= \lambda x.\lambda s.(x, s) \\
\mathsf{bind} &: \forall s\, a\, b.\, \mathsf{State}\ s\ a \longrightarrow (a \longrightarrow \mathsf{State}\ s\ b) \longrightarrow \mathsf{State}\ s\ b \\
&= \lambda f.\lambda g.\lambda s.\mathsf{let}\ (x, s) = f\ s\ \mathsf{in}\ g\ x\ s \\
\mathsf{get} &: \forall s\, a.\, \mathsf{State}\ s\ a \\
&= \lambda s.(s, s) \\
\mathsf{put} &: \forall s\, a.\, a \longrightarrow \mathsf{State}\ s\ () \\
&= \lambda x.\lambda s.((), x)
\end{aligned}$$

Is there a store-passing translation for general references?

We are missing:

- *dynamic memory allocation*, and
- *higher-order store.*

Each of these features poses significant difficulties...

Imagine the store has n cells of base type, where n grows with time.

The type of the store changes with time. So, what is the translation of the monad?

A computation may make assumptions about the existence and type of certain cells. Thus, it accepts any store that is *large enough* to satisfy these assumptions.

A computation may itself allocate new cells, so it returns a new store that is *larger* than its input store.

In summary, the translation of the monad involves *an extension ordering* over stores and *a bounded ∀∃ pattern,* roughly so:

$$[\![M\ \tau]\!] = \forall s_1 \geq s,\ s_1 \to \exists s_2 \geq s_1,\ ([\![\tau]\!], s_2)$$

But where is s bound? *The encoding of a type must be parameterized* with a store, roughly so:

$$[\![M\ \tau]\!]_s = \forall s_1 \geq s,\ s_1 \to \exists s_2 \geq s_1,\ ([\![\tau]\!]_{s_2}, s_2)$$

This implies the need for a *monotonicity* principle: a value that is valid now should remain valid later. That is,

$$\text{if } s_1 \leq s_2, \text{ then } [\![\tau]\!]_{s_1} \leq [\![\tau]\!]_{s_2}.$$

What is the extension ordering?

A store is a (tuple) type, of kind $\star$. Let a *fragment* be a store with a hole at the end, that is, an object of kind $\star \longrightarrow \star$.

Fragment concatenation is just function composition. A (prefix) ordering *over fragments* is defined in terms of concatenation.

A fragment can be turned into a store by applying it to the () type. Write $\mathrm{store}\, f = f\, ()$.

Dynamic memory allocation — the monad

With these conventions in mind, the translation of the monad could be written roughly so:

$$[\![M\ \tau]\!]_f = \forall f_1,\ \text{store}\ (f @ f_1) \longrightarrow \exists f_2,\ ([\![\tau]\!]_{f @ f_1 @ f_2}, \text{store}\ (f @ f_1 @ f_2))$$

The encoding of a type is now parameterized with a *fragment*.

The use of *concatenation* (an *associative* operation) allows expressing bounded quantification in terms of ordinary quantification.

This is roughly what is needed to deal with dynamic memory allocation with cells of base type.

In short, we need store descriptions that are *extensible in width* and a notion of *monotonicity*.

This is hardly simple, but it gets worse with *higher-order store*...

The translation of a base type ignores its parameter. If a cell has base type at the source level, then its type in the target calculus remains fixed as the store grows.

If a cell has a computation type at the source level, then *the type of this cell* in the target calculus *changes* as the store grows.

To explain this, one needs store descriptions that are *extensible in width and in depth*.

Let us call a *world* such a doubly-open-ended store description.

$$W = ? \longrightarrow (\star \longrightarrow \star)$$

Worlds represent points in time. We intend to set up an *ordering* on worlds in terms of an appropriate notion of *world composition.*

But *of what kind* is the "depth" parameter of a world?

Let us call a *world* such a doubly-open-ended store description.

$$W = W \longrightarrow (\star \longrightarrow \star)$$

Worlds represent points in time. We intend to set up an *ordering* on worlds in terms of an appropriate notion of *world composition.*

But *of what kind* is the “depth” parameter of a world?

It is a point in time, that is, a world!

We seem to be looking at a *recursive kind.*

A semanticist would say: we are looking at *a recursive domain equation.*

An equation of the form $W = W \to \ldots$ arises in several semantic models of general references [Schwinghammer et al., 2009, Birkedal et al., 2009, Hobor et al., 2010].

Semanticists have dealt with this complexity for a long time, while "syntacticists" have remained blissfully ignorant of it, thanks to the miraculous notion of a *store typing* [Wright and Felleisen, 1994, Harper, 1994], an ad hoc recursive type that can be viewed as *simultaneously closed and open* and that grows with time.

The present work can be viewed as an attempt to *reverse-engineer* some of the semantic constructions in the literature and bring them back in the realm of syntax.

To put this another way, the idea is to build a semantic model of general references as the *composition* of:

- a type-preserving store-passing translation into some intermediate language;
- a semantic model of this intermediate language.

The present work can also be viewed as an answer to the challenge of defining a type-preserving store-passing translation,

with the proviso that well-typedness must guarantee: *no out-of-bounds accesses to the store.*

So, we need a calculus with recursive kinds.

So, we need a calculus with recursive kinds.

Do we want a calculus with *all* recursive kinds?

So, we need a calculus with recursive kinds.

Do we want a calculus with *all* recursive kinds?

No. This would lead to a rather wild system where types can exhibit arbitrary computational behavior.

So, we need a calculus with recursive kinds.

Do we want a calculus with *all* recursive kinds?

No. This would lead to a rather wild system where types can exhibit arbitrary computational behavior.

Do we *need* a calculus with all recursive kinds?

So, we need a calculus with recursive kinds.

Do we want a calculus with *all* recursive kinds?

No. This would lead to a rather wild system where types can exhibit arbitrary computational behavior.

Do we *need* a calculus with all recursive kinds?

No. It turns out that certain well-behaved recursive kinds suffice...

Nakano's system [2000, 2001] has *certain*, but not all, recursive types.

Well-typed terms are not necessarily strongly normalizing, but are *productive*.

It turns out that the patterns of recursion that are needed to define worlds, world composition, etc. are well-typed in Nakano's system.

In summary, the plan is:

1. *define* Fork, a variant of F_ω *equipped* with Nakano's system at the kind level;
2. *define* a type-preserving store-passing translation of *general* references into Fork;
3. (for semanticists only) build a model of Fork;
4. *get* filthy rich (!?).

Nakano's types are *co-inductively* defined by:

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

In fact, only *well-formed, finite* types are permitted...

A type is *well-formed* iff *every* infinite path infinitely *often enters* a bullet.

In a finite presentation, this means that *every cycle enters* a bullet.

A type is *finite* iff *every* rightmost path is finite.

In a finite presentation, this means that *every cycle enters the* left-hand side *of an arrow.*

Subtyping is a *pre-order* and additionally validates the following laws:

$$\frac{\kappa_1' \leq \kappa_1 \qquad \kappa_2 \leq \kappa_2'}{\kappa_1 \rightarrow \kappa_2 \leq \kappa_1' \rightarrow \kappa_2'} \qquad \frac{\kappa \leq \kappa'}{\bullet\kappa \leq \bullet\kappa'} \qquad \kappa \leq \bullet\kappa \qquad \bullet(\kappa_1 \rightarrow \kappa_2) \lesseqgtr \bullet\kappa_1 \rightarrow \bullet\kappa_2$$

When types are finitely represented by a set of mutually recursive equations, subtyping is (efficiently) decidable.

Nakano's terms are pure λ-terms:

$$\tau ::= a \mid \lambda a.\tau \mid \tau\,\tau$$

The type-checking rules are standard:

$$K \vdash a : K(a) \qquad \frac{K; a : \kappa_1 \vdash \tau : \kappa_2}{K \vdash \lambda a.\tau : \kappa_1 \to \kappa_2} \qquad \frac{\begin{array}{c} K \vdash \tau_1 : \kappa_1 \to \kappa_2 \\ K \vdash \tau_2 : \kappa_1 \end{array}}{K \vdash \tau_1\ \tau_2 : \kappa_2}$$

$$\frac{K \vdash \tau : \kappa_1 \qquad \kappa_1 \leq \kappa_2}{K \vdash \tau : \kappa_2}$$

Theorem (Subject Reduction)

$K \vdash \tau_1 : \kappa$ *and* $\tau_1 \longrightarrow \tau_2$ *imply* $K \vdash \tau_2 : \kappa$.

The proof relies on the following unusual lemma:

Lemma (Degradation)

$K \vdash \tau : \kappa$ *implies* $\bullet K \vdash \tau : \bullet \kappa$.

Theorem (Productivity)

$K \vdash \tau : \kappa$ *implies that* τ *admits a head normal form.*

The proof, due to Nakano, uses realizability / logical relations...

Types are interpreted as sets of terms:

$$
\begin{array}{rcl}
[\![\kappa]\!]_0 & = & \{\tau\} \\
[\![\star]\!]_{j+1} & = & \{\tau \mid \tau \longrightarrow^\star a\, \tau_1\ \ldots\ \tau_n\} \\
[\![\kappa_1 \to \kappa_2]\!]_{j+1} & = & \{\tau_1 \mid \forall k \leq j+1 \quad \forall \tau_2 \in [\![\kappa_1]\!]_k \quad (\tau_1\ \tau_2) \in [\![\kappa_2]\!]_k\} \\
[\![\bullet\,\kappa]\!]_{j+1} & = & [\![\kappa]\!]_j
\end{array}
$$

This definition makes sense only because types are well-formed.

The sequence $[\![\kappa]\!]_j$ is monotonic and reaches a limit $[\![\kappa]\!] = \bigcap_j [\![\kappa]\!]_j$.

The interpretation validates subtyping:

Lemma

$\kappa_1 \leq \kappa_2$ *implies* $[\![\kappa_1]\!]_j \subseteq [\![\kappa_2]\!]_j$.

The interpretation validates type-checking:

Theorem

$K \vdash \tau : \kappa$ *implies* $\tau \in [\![\kappa]\!]$.

The fact that every well-typed term admits a head normal form follows.

As a consequence of Subject Reduction and Productivity, we have:

Corollary

Every well-typed term admits a maximal Böhm tree.

Let Y stand for $\lambda f.((\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)))$.

The judgement $\vdash Y : (\kappa \longrightarrow \kappa) \longrightarrow \kappa$ *cannot* be derived. Otherwise, the term $Y\ (\lambda x.x)$, which does not have a head normal form, would be well-typed, contradicting Productivity.

This judgement *can* be derived in simply-typed λ-calculus with recursive types. The self-application $(x\ x)$ requires $x : \kappa'$, where κ' satisfies $\kappa' \equiv \kappa' \longrightarrow \kappa$. In Nakano's system, this type is ill-formed.

Y stands for $\lambda f.((\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)))$.

In Nakano's system, *one* uses $x : \kappa'$, where $\kappa' \equiv \bullet(\kappa' \longrightarrow \kappa) \equiv \bullet\kappa' \longrightarrow \bullet\kappa$.

Thus, the self-application $(x\ x)$ has type $\bullet\kappa$.

Assuming $f : \bullet\kappa \longrightarrow \kappa$, we find that $f\ (x\ x)$ has type κ. Thus, $\lambda x.f\ (x\ x)$ has type $\kappa' \longrightarrow \kappa$. By subtyping, it also has type $\bullet(\kappa' \longrightarrow \kappa)$, that is, κ'. Thus, the self-application $(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$ has type κ.

This leads to $\vdash Y : (\bullet\kappa \longrightarrow \kappa) \longrightarrow \kappa$.

An intuition is, *only the contractive functions* have fixed points.

Let $\mu a.\tau$ be sugar for $Y\,(\lambda a.\tau)$. Then, the following rule is derivable:

$$\frac{K; a : \bullet\kappa \vdash \tau : \kappa}{K \vdash \mu a.\tau : \kappa}$$

F_ω with recursive kinds, Fork for short, uses Nakano's system at the kind and type levels.

This yields a system with very expressive recursive types: a recursive type is produced by a non-terminating, but productive, type-level computation.

$$
\begin{array}{lll}
\kappa ::= \dots & & \text{(as before)} \\
\tau ::= \dots & & \text{(as before)} \\
\quad \mid \to \mid () \mid (\,,\,) \mid \forall_\kappa \mid \exists_\kappa & & \text{(type constants)} \\
t ::= x \mid \lambda x.t \mid t\ t & & \text{(functions)} \\
\quad \mid () & & \text{(unit)} \\
\quad \mid (t, t) \mid \mathsf{let}\ (x, x) = t\ \mathsf{in}\ t & & \text{(pairs)} \\
\quad \mid \Lambda a.t \mid t\ \tau & & \text{(universals)} \\
\quad \mid \mathsf{pack}\ \tau, t\ \mathsf{as}\ \tau \mid \mathsf{unpack}\ a, x = t\ \mathsf{in}\ t & & \text{(existentials)}
\end{array}
$$

The kind assignment judgement $K \vdash \tau : \kappa$ is Nakano's, *extended* with axioms for the type constants:

1. $K \vdash () : \star$
2. $K \vdash \rightarrow : \bullet\star \rightarrow \bullet\star \rightarrow \star$
3. $K \vdash (\,,\,) : \bullet\star \rightarrow \bullet\star \rightarrow \star$
4. $K \vdash \forall_\kappa : (\kappa \rightarrow \bullet^n\star) \rightarrow \bullet^n\star$
5. $K \vdash \exists_\kappa : (\kappa \rightarrow \bullet^n\star) \rightarrow \bullet^n\star$

Axioms 2 & 3 reflect the idea that $\rightarrow$ and $(\,,\,)$ are type *constructors*, as opposed to arbitrary type operators. They build structure *on top of* their arguments, that is, they are contractive.

Due to this decision, *the types that classify values have kind* $\bullet^n\star$ in general. Axioms 4 & 5 reflect this.

I write $\Gamma \vdash \tau : \bigcirc\kappa$ when $\Gamma \vdash \tau : \bullet^n\kappa$ holds for some n.

Var
$$\Gamma \vdash x : \Gamma(x)$$

Abs
$$\frac{\Gamma \vdash \tau_1 : \bigcirc\star \qquad \Gamma; x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x.t : \tau_1 \to \tau_2}$$

App
$$\frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1\ t_2 : \tau_2}$$

Unit
$$\Gamma \vdash () : ()$$

(,)-Intro
$$\frac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (t_1, t_2) : (\tau_1, \tau_2)}$$

(,)-Elim
$$\frac{\begin{array}{c}\Gamma \vdash t_1 : (\tau_1, \tau_2) \\ \Gamma; x_1 : \tau_1; x_2 : \tau_2 \vdash t_2 : \tau\end{array}}{\Gamma \vdash \text{let } (x_1, x_2) = t_1 \text{ in } t_2 : \tau}$$

$\forall$-Intro
$$\frac{\Gamma; a : \kappa \vdash t : \tau \qquad a \# \Gamma}{\Gamma \vdash \Lambda a.t : \forall_\kappa\ (\lambda a.\tau)}$$

$\forall$-Elim
$$\frac{\begin{array}{c}\Gamma \vdash t : \forall_\kappa\ \tau_1 \\ \Gamma \vdash \tau_2 : \bigcirc\kappa\end{array}}{\Gamma \vdash t\ \tau_2 : \tau_1\ \tau_2}$$

$\exists$-Intro
$$\frac{\begin{array}{c}\Gamma \vdash t : \tau_1\ \tau_2 \\ \Gamma \vdash \exists_\kappa\ \tau_1 : \bigcirc\star \qquad \Gamma \vdash \tau_2 : \bigcirc\kappa\end{array}}{\Gamma \vdash \text{pack } \tau_2, t \text{ as } \exists_\kappa\ \tau_1 : \exists_\kappa\ \tau_1}$$

$\exists$-Elim
$$\frac{\begin{array}{c}\Gamma \vdash t_1 : \exists_\kappa\ \tau_1 \qquad a \# \Gamma, \tau_2 \\ \Gamma; a : \kappa; x : (\tau_1\ a) \vdash t_2 : \tau_2\end{array}}{\Gamma \vdash \text{unpack } a, x = t_1 \text{ in } t_2 : \tau_2}$$

Conversion
$$\frac{\begin{array}{c}\Gamma \vdash t : \tau_1 \\ \Gamma \vdash \tau_2 : \bigcirc\star \qquad \tau_1 \equiv \tau_2\end{array}}{\Gamma \vdash t : \tau_2}$$

Definition (Well-formedness)

The empty typing environment is well-formed. The typing environment $\Gamma; a : \kappa$ is well-formed if Γ is well-formed and $a \# \Gamma$. The typing environment $\Gamma; x : \tau$ is well-formed if Γ is well-formed and $\Gamma \vdash \tau : \bigcirc\star$.

Lemma (Well-formedness)

If Γ is well-formed, then $\Gamma \vdash t : \tau$ implies $\Gamma \vdash \tau : \bigcirc\star$.

Lemma (Degradation)

$\Gamma_1; a : \kappa; \Gamma_2 \vdash t : \tau$ *implies* $\Gamma_1; a : \bullet\kappa; \Gamma_2 \vdash t : \tau$.

Lemma (Type substitution)

$\Gamma_1; a : \kappa; \Gamma_2 \vdash t : \tau_2$ *and* $\Gamma_1 \vdash \tau_1 : \kappa$ *imply*
$\Gamma_1; [a \mapsto \tau_1]\Gamma_2 \vdash [a \mapsto \tau_1]t : [a \mapsto \tau_1]\tau_2$.

Corollary (Type substitution with degradation)

$\Gamma_1; a : \kappa; \Gamma_2 \vdash t : \tau_2$ *and* $\Gamma_1 \vdash \tau_1 : \bigcirc\kappa$ *imply*
$\Gamma_1; [a \mapsto \tau_1]\Gamma_2 \vdash [a \mapsto \tau_1]t : [a \mapsto \tau_1]\tau_2$.

Lemma (Value substitution)

...

Lemma (Subject Reduction)

...

Lemma (Progress)

...

Contents

A world is a *contractive* function of worlds to fragments:

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

That is, a world must produce some structure before invoking its world argument.

There is a direct analogy with the metric approach used by some semanticists [Schwinghammer et al., 2009, Birkedal et al., 2009].

World composition is *recursively defined.*

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

Composition is *associative:*

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

The proof is automatic. In fact, it is not necessary to state the lemma — the type-checker could find out that it must prove this.

Just like a world, a *semantic type* is a contractive function of a world.

```
kind stype = later world -> *
```

Let our *source language* be a monadic presentation of System F with general references:

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

We encode each of these connectives as a semantic type transformer...

Here are the easy ones:

```
type unit : stype =
  \x. ()

type pair : stype -> stype -> stype =
  \a b. \x. (a x, b x)

type univ : (stype -> stype) -> stype =
  \body : stype -> stype. \x.
    forall a. body a x
```

The future world x is not used, only transmitted down.

A value is *necessary* if it is valid not only now, but also in every future world:

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

Functions require necessary arguments and produce necessary results:

```
type arrow : stype -> stype -> stype =
  \a b. \x. box a x -> box b x
```

A computation requires a *current* store and produces a value and store in some *future* world:

```
type monad : stype -> stype =
  \a. \x.
    store x -> outcome a x

type outcome : stype -> stype =
  \a. \x.
    exists y. (box a (x `o` y), store (x `o` y))
```

A store in world *w* is an array of necessary values:

```
type store : world -> * =
  \w. forall x. array (w x)
```

A reference in world *w* is an index into such an array:

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

What are arrays, what are indices?

Arrays and indices can be implemented in F_ω:

```
type array : later fragment -> *
type index : later fragment -> later * -> *
term array_empty : array fnil
term array_extend :
  forall f data. array f -> data -> array (f ‘snoc‘ data)
term array_read :
  forall f data. array f -> index f data -> data
term array_write :
  forall f data. array f -> index f data -> data -> array f
term array_end_index :
  forall f. array f -> forall data. index (f ‘snoc‘ data) data
term index_monotonic :
  forall f1 f2 data. index f1 data -> index (f1 ‘@‘ f2) data
```

When a new reference is allocated, the world grows by one *cell:*

```
type cell : stype -> later world -> world =
  \a x y tail. (a (x `o` cell a x `o` y), tail)
```

Here is how the world changes when a new memory cell is allocated.

The following two functions respectively return the new store and the address of the new cell:

```
term store_extend :
  forall x a. store x -> box a x -> store (x ‘o‘ cell a x)
term store_end_index :
  forall x a. store x -> ref a (x ‘o‘ cell a x)
```

Up to type abstractions and applications, The former is just **array_extend**, while the latter is just **array_end_index**...

For instance:

```
term store_extend : forall x a. store x -> box a x -> store (x ‘o‘ cell
  /\ x a.
  \s : store x.
  \v : box a x.
  /\ y.
    type cy = cell a x ‘o‘ y in
    array_extend [x cy] [a (x ‘o‘ cy)] (s [cy]) (v [cy])
```

Mechanized type-checking is essential.

It is now straightforward to encode these operations:

```
term new :
  box (univ (\a. a `arrow` monad (ref a))) nil
term read :
  box (univ (\a. ref a `arrow` monad a)) nil
term write :
  box (univ (\a. (ref a `pair` a) `arrow` monad unit)) nil
```

Encoding the monadic combinators

It is also straightforward to *encode* the monadic combinators:

```
term return :
  box (univ (\a. a ‘arrow‘ monad a)) nil
term bind :
  box (univ (\a. univ (\b.
    (monad a ‘pair‘ (a ‘arrow‘ monad b)) ‘arrow‘ monad b
  ))) nil
```

The complete encoding is about 800 lines of kind/type/term definitions, lemmas, and comments.

It is checked in 0.1 seconds.

Contents

What conclusions can be drawn?

- The encoding *exists*.
- Semanticists do this on paper; here, it is *machine-checked.*
- You wouldn't want to implement this in a compiler!
- Fork is an *economical* and *expressive* calculus.

(Most titles are clickable links to online versions.)

- Birkedal, L., Støvring, K., and Thamsborg, J. 2009.
  Realizability semantics of parametric polymorphism, general references, and recursive types.
  In *International Conference on Foundations of Software Science and Computation Structures (FOSSACS)*. Lecture Notes in Computer Science, vol. 5504. Springer, 456–470.

- Harper, R. 1994.
  A simplified account of polymorphic references.
  Information Processing Letters 51, 4, 201–206.

- Hobor, A., Dockins, R., and Appel, A. W. 2010.
  A theory of indirection via approximation.
  In *ACM Symposium on Principles of Programming Languages (POPL)*.

Bibliography]Bibliography

Moggi, E. 1991.
Notions of computation and monads.
Information and Computation 93, 1.

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

[][

Nakano, H. 2001.
Fixed-point logic with the approximation modality and its Kripke completeness.
In *International Symposium on Theoretical Aspects of Computer Software (TACS)*. Lecture Notes in Computer Science, vol. 2215. Springer, 165–182.

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.

[][

Strachey, C. 2000.
Fundamental concepts in programming languages.
Higher-Order and Symbolic Computation 13, 1–2 (Apr.), 11–49.

Wright, A. K. and Felleisen, M. 1994.
A syntactic approach to type soundness.
Information and Computation 115, 1 (Nov.), 38–94.