A typed store-passing translation for general references

François Pottier

▶ **To cite this version:**

François Pottier. A typed store-passing translation for general references. POPL 2011: 38th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, Jan 2011, Austin, United States. 10.1145/1925844.1926403 . hal-01081187

HAL Id: hal-01081187
https://inria.hal.science/hal-01081187

Submitted on 7 Nov 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Typed Store-Passing Translation for General References

François Pottier
INRIA
Francois.Pottier@inria.fr

Abstract

We present a store-passing translation of System F with general references into an extension of System F_ω with certain well-behaved recursive kinds. This seems to be the first type-preserving store-passing translation for general references. It can be viewed as a purely syntactic account of a possible worlds model.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Dynamic storage management; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type structure

General Terms Languages, Theory

1. Introduction

Motivation Building a semantic model of a programming language amounts to translating it into some mathematical metalanguage. An important constraint in the design of such an interpretation is that it must explain (that is, translate away) computational effects such as non-termination and state, which do not exist in mathematics. Non-termination is often handled using the tools of domain theory, which was created for this purpose. State is usually dealt with in the style of Strachey [32] by making the store explicit and interpreting commands as functions that map stores to stores. Another important consideration is that the model should exploit the type discipline of the programming language. This enables it to serve as a tool in establishing type soundness and in proving typed contextual equivalence laws.

As a result of these considerations, building a semantic model of a rich programming language can be a challenging task. It is common wisdom among compiler writers that when one is faced with a complex translation task, one should decompose it into a succession of independent phases, connected via suitably chosen intermediate languages.

In this paper, we study one instance where this wisdom might be applicable in the construction of a semantic model. We focus on a particular programming language, namely a version of System F equipped with general references, and on a particular aspect of its semantics, namely the store-passing transformation, whose purpose is to explain (translate away) references. We isolate this transformation, just as if it were a phase in a compiler, and present it as a translation of our typed, imperative source language into a typed, purely functional intermediate language.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
POPL'11, January 26–28, 2011, Austin, Texas, USA.
Copyright © 2011 ACM 978-1-4503-0490-0/11/01. . . $10.00

In so doing, we move the frontier between syntax and semantics. We suggest that a model of the imperative source language might be obtained through the composition of the store-passing transformation with a model of the purely functional intermediate language. This makes the construction of the model more modular, and may help explain it to researchers who are familiar with syntactic techniques. This also extends the family of known type-preserving transformations: to the best of our knowledge, until now, it was an open question how to define a type-preserving store-passing translation for general references.

Which intermediate language? In this endeavor, an important part of the difficulty is to find out what the typed intermediate language should be, and to keep it minimal. It would be nice if it could be a well-studied calculus, such as System F or F_ω. However, these calculi are strongly normalizing, whereas our source language is not: general references allow recursion through the store. Thus, one more ingredient is needed. In order to find out what this ingredient should be, let us first review some of the semantic models of general references.

Possible worlds models A reference is a dynamically-allocated memory cell, whose value can be read and modified at any time. We consider *general* references, which means that a reference can hold a value of any type, including a function, or the address of some other reference. Furthermore, we are interested in *weak* references. That is, we are interested in a type discipline whereby updates are type-invariant, de-allocation is forbidden, and aliasing is permitted.

The presence of references precludes a naïve interpretation of types as sets of values. A sentence such as: "the address 100 is an integer reference" does not make sense on its own. It implicitly relies on the assumption that the address 100 is currently allocated, that some integer value is currently stored there, and on the knowledge that these facts will continue to hold in the future.

To reflect this, several semantic models of weak references in the literature follow the "possible worlds" approach [2, 10, 16, 34]. There, the interpretation of a type is parameterized over a world, which represents the current state of the store. Worlds are equipped with a partial ordering: $w_1 \leq w_2$ means that w_2 is a possible future world of w_1, that is, every address that is allocated in w_1 is also allocated in w_2, with the same type. Bounded universal quantification over worlds is used to express the idea that a value that is valid now is also valid in every possible future world. Bounded existential quantification is used to express the idea that a command has the effect of transforming the current world into some possible future world.

At the heart of these possible worlds models of general references is a circularity. As stated above, semantic types are open-ended: they are parameterized over a world. However, worlds too must be open-ended: they must describe the store now and in every possible future. One way of achieving this is to define a world as a map of memory addresses to semantic types. This makes the definitions of worlds and semantic types mutually recursive,

and creates a need to either solve a recursive domain equation *world* $\cong$ *world* $\to \ldots$ [10] or work with approximate solutions of it [2, 14]. An alternative is to define a world as a map of memory addresses to syntactic types [16]. Then, the circularity appears when worlds and types must be interpreted as semantic objects, and again a recursive domain equation must be solved.

A calculus with recursive kinds In this paper, we are not interested in solving or approximating recursive domain equations. Our purpose is to argue that a certain syntactic program transformation (namely, a store-passing translation) produces well-typed terms. Drawing inspiration from the possible worlds models, we wish to parameterize and to quantify types over worlds. Thus, we need worlds to be types (of a particular kind). That is, *world* should be a kind. Because we need *world* $\cong$ *world* $\to \ldots$, we should look for an intermediate calculus that supports *recursive kinds.* This is the key ingredient that was alluded to above.

Should we propose an extension of F_ω with *arbitrary* recursive kinds? Such a system would have Turing-complete computation at the type level. That sounds rather wild. Do we need non-terminating computation at the type level? Yes. We do wish to allow certain forms of non-terminating computation at the type level, because we find it natural to view a recursive type as a λ-term whose infinite reduction produces, in the limit, an infinite tree. This is a form of non-terminating but *productive* computation.

Is there a way of ensuring that every type-level computation is productive in such a sense? Yes. Nakano [21, 22], for instance, presents a type system that controls recursion so as to guarantee productive computation. This system has recursive types but requires every cycle in the type structure to cross a "later" modality, written $\bullet$. We re-use this system off-the-shelf, at the kind level; therefore, Nakano's terms and types become our types and kinds.

By adopting Nakano's system, we rule out certain recursive kinds. For instance, the recursive equation *world* $=$ *world* $\to \ldots$ is in fact invalid, because it does not involve the modality. Fortunately, the modified equation *world* $=$ $\bullet$*world* $\to \ldots$ is permitted, and still fits our purposes. This equation states that a world is a *contractive* function: it is able to produce some output independently of its argument. This intuitively seems to correspond to the fact that if one attempts to describe the shape of the store, one will be able to provide a non-empty prefix of a description before one hits a self-reference to the shape of the store. For instance, one might say: *"the store currently contains one cell, which contains one function, whose argument must be a store that is a possible future of the current store..."*

The ultrametric-space techniques of Birkedal *et al.* [9, 10] and of Schwinghammer *et al.* [30] served as inspiration for the present work, so it is no surprise that there is a close analogy between these works and ours. Their $\frac{1}{2}$ scaling factor becomes the $\bullet$ modality. Their construction of world composition as the fixed point of a contractive map [30, Lemma 13] becomes a recursive definition that happens to be permitted in Nakano's system. The fact that world composition is associative [30, Lemma 14] becomes an assertion about the equality of two Böhm trees and can be automatically checked (§4.4).

In recent work, spurred in part by an earlier version of the present paper, Birkedal *et al.* [8] construct a metric model of a version of Nakano's system. This gives firm footing to the correspondence between $\bullet$ and $\frac{1}{2}$. Birkedal *et al.* note that their work provides the basis for a semantic model of the typed calculus that is presented here. Whether and how this model can be exploited in useful ways remains to be determined.

Contributions The contributions of this paper are: (i) the design of FORK, an extension of system F Omega with well-behaved Recursive Kinds; (ii) a type-preserving encoding of general references into this calculus.

$$\frac{\kappa_1' \le \kappa_1 \qquad \kappa_2 \le \kappa_2'}{\kappa_1 \to \kappa_2 \le \kappa_1' \to \kappa_2'} \qquad \frac{\kappa \le \kappa'}{\bullet\kappa \le \bullet\kappa'} \qquad \kappa \le \bullet\kappa$$

$$\bullet(\kappa_1 \to \kappa_2) \lesseqgtr \bullet\kappa_1 \to \bullet\kappa_2$$

Figure 1. FORK: properties of the subkind relation

$$K \vdash \alpha : K(\alpha) \qquad \frac{K; \alpha : \kappa_1 \vdash \tau : \kappa_2}{K \vdash \lambda\alpha.\tau : \kappa_1 \to \kappa_2}$$

$$\frac{K \vdash \tau_1 : \kappa_1 \to \kappa_2 \quad K \vdash \tau_2 : \kappa_1}{K \vdash \tau_1\ \tau_2 : \kappa_2} \qquad \frac{K \vdash \tau : \kappa_1 \qquad \kappa_1 \le \kappa_2}{K \vdash \tau : \kappa_2}$$

Figure 2. FORK: kind assignment

Paper outline We first recall Nakano's system (§2), and build upon it in the definition of FORK (§3). Then, we explain how a version of System F equipped with general references can be encoded into FORK. We prove that the encoding is type-preserving and semantics-preserving[1] (§4). A prototype implementation of a FORK type-checker, the complete source code for the encoding of references, as well as the machine-checked proof of semantic preservation, are available online [28]. Some proofs are omitted and can be found in the extended version of this paper [29].

2. Nakano's system

We now recall the definition and properties of Nakano's system [21, 22]. Our version of the system is close to Nakano's S-$\lambda\bullet\mu^+$. It is slightly simplified in that it is restricted to finite types (without distinction between positively and negatively finite types) and (as a result) it does not have a $\top$ type. Despite this difference, we refer to it as "Nakano's system".

There are two levels in Nakano's system, which are usually referred to as "types" and "terms". Nakano's "types" and "terms" are the *kinds* and *types* of FORK, respectively, so this is how we refer to them.

Kinds The kinds are *co-inductively* defined as follows:

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

A kind κ is a possibly infinite tree. In the prototype implementation [28], kinds are finitely represented via a set of mutually recursive defining equations. (Thus, only regular kinds can be defined.)

A kind κ is *well-formed* iff every infinite path through κ crosses a $\bullet$ constructor infinitely often. A kind is *finite* iff every infinite path through κ enters the domain of an arrow infinitely often. We restrict our attention to kinds that are well-formed and finite. (The finiteness condition is used in the proof of Lemma 2.4, where it serves to rule out types that produce an infinite stream of λ's, and therefore do not have a head normal form.) In the implementation, this requirement is enforced by checking that every occurrence of a kind name in the right-hand side of its defining equation(s) lies under a $\bullet$ constructor *and* in the domain of an arrow.

[1] More precisely, we prove that the encoding preserves convergence, and sketch how one might prove that it also preserves divergence.

Subkinding Kinds come with a *subkind* relation $\leq$. We omit its definition, which is somewhat technical (the reader is referred to [27]) and is irrelevant as long as the following properties hold. The subkind relation is reflexive, transitive, and satisfies the laws in Figure 1, where $\overset{\leq}{\geq}$ means that the relation holds in both directions. The subkind relation satisfies the following inversion lemma:

Lemma 2.1 *If $\kappa_1 \to \kappa_2 \leq \kappa_1' \to \kappa_2'$ holds, then, for some $n \in \mathbb{N}$, both $\kappa_1' \leq \bullet^n \kappa_1$ and $\bullet^n \kappa_2 \leq \kappa_2'$ hold.* ◇

(We write $\bullet^n \kappa$ for n applications of $\bullet$ to κ.) When kinds are finitely represented as a set of mutually recursive defining equations, it is decidable whether two kinds κ_1 and κ_2 are in the subkind relation. In fact, it is possible to decide whether there exists $n \in \mathbb{N}$ such that $\kappa_1 \leq \bullet^n \kappa_2$ holds, and to compute the least such n when one exists [27]. The prototype implementation [28] takes advantage of this fact to perform bottom-up kind synthesis.

Types Types are pure λ-terms:

$$\tau ::= \alpha \mid \lambda\alpha.\tau \mid \tau\ \tau$$

We write $\tau_1 \longrightarrow \tau_2$ when τ_1 β-reduces to τ_2. Reduction is permitted under arbitrary contexts.

A kind environment K is a sequence of bindings of the form $\alpha : \kappa$. The judgement $K \vdash \tau : \kappa$ means that, within such an environment K, the type τ has kind κ. The rules that define it (Figure 2) are standard.

Recursion The judgement $\vdash Y : (\bullet\kappa \to \kappa) \to \kappa$, where Y is Curry's fixed point combinator $\lambda f.((\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)))$, is derivable. (The variable x receives the recursive kind $\kappa' = \bullet(\kappa' \to \kappa)$.) We introduce the notation $\mu\alpha.\tau$ as syntactic sugar for $Y\ (\lambda\alpha.\tau)$. This gives rise to the derived reduction rule:

$$\mu\alpha.\tau \longrightarrow [\alpha \mapsto \mu\alpha.\tau]\tau$$

and to the derived kind assignment rule:

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

The prototype implementation [28] has built-in support for recursive type definitions, based on the above rules. In fact, it supports mutually recursive type definitions.

Properties The system enjoys the following *degradation* and *subject reduction* properties.

Lemma 2.2 *$K \vdash \tau : \kappa$ implies $\bullet K \vdash \tau : \bullet\kappa$.* ◇

Lemma 2.3 *$K \vdash \tau_1 : \kappa$ and $\tau_1 \longrightarrow \tau_2$ imply $K \vdash \tau_2 : \kappa$.* ◇

A type of the form $\lambda\alpha_1 \ldots \alpha_m.\alpha\ \tau_1 \ldots \tau_n$ is a *head normal form.* Types are solvable: they admit head normal forms [21, 22, 27].

Lemma 2.4 *If $K \vdash \tau : \kappa$ then τ has a head normal form.* ◇

It is worth noting that this holds under any environment K, that is, in the presence of type variables of arbitrary kind. Together, Lemmas 2.3 and 2.4 imply that every type admits a maximal Böhm tree, that is, one that does not contain any occurrence of $\bot$ (the undefined Böhm tree). These two lemmas have been checked by the author using the Coq proof assistant [27].

Type equality We take *type equality,* a relation between types, to be *Böhm tree equivalence up to η* [7, §10.2.25] [15]. Several alternative characterizations of this relation are known. It is the greatest consistent λ-theory. It coincides with the equational theory of Scott's D_∞ model. It is the greatest compatible semi-sensible relation, that is, it coincides with the observational congruence obtained by observing solvability.

We write $\tau_1 \equiv \tau_2$ when τ_1 and τ_2 are in the type equality relation. In this paper, this relation is used only when both $K \vdash \tau_1 : \kappa$ and $K \vdash \tau_2 : \kappa$ hold for some environment K and kind κ. This has the following beneficial consequence:

Lemma 2.5 *The relation $\equiv$, restricted to well-kinded types, is semi-decidable.* ◇

Proof. We outline a simple semi-algorithm. This semi-algorithm maintains a conjunction G of goals, where a *goal* is an equation $\tau_1 \equiv \tau_2$ between two head normal forms. The free variables of a goal are implicitly viewed as universally quantified.

A goal g can be decomposed into a conjunction of sub-goals, written $\langle g \rangle$, as per the following equations. A case applies only if no prior case applies. We write $\tau\!\downarrow$ for the principal head normal form of τ.

$$\begin{array}{lll}
\langle \lambda\alpha.\tau_1 \equiv \lambda\alpha.\tau_2 \rangle \text{ is } \langle \tau_1 \equiv \tau_2 \rangle & & \\
\langle \lambda\alpha.\tau_1 \equiv \tau_2 \rangle \text{ is } \langle \tau_1 \equiv \tau_2\ \alpha \rangle & & \text{if } \alpha \# \tau_2 \\
\langle \tau_1 \equiv \lambda\alpha.\tau_2 \rangle \text{ is } \langle \tau_1\ \alpha \equiv \tau_2 \rangle & & \text{if } \alpha \# \tau_1 \\
\langle \tau_1\ \tau_1' \equiv \tau_2\ \tau_2' \rangle \text{ is } \langle \tau_1 \equiv \tau_2 \rangle \wedge \tau_1'\!\downarrow \equiv \tau_2'\!\downarrow & & \\
\langle \alpha \equiv \alpha \rangle \text{ is true} & & \\
\langle \tau_1 \equiv \tau_2 \rangle \text{ is false} & &
\end{array}$$

We write $\langle G \rangle$ for the conjunction of sub-goals obtained by applying $\langle\cdot\rangle$ to every goal in G.

Consider the problem of deciding whether $\tau_1 \equiv \tau_2$ holds. Enumerate the potentially infinite sequence defined by $G_0 = (\tau_1\!\downarrow \equiv \tau_2\!\downarrow)$ and $G_{k+1} = \langle G_k \rangle$. If some G_k is found to be empty, report "yes". If some G_k is found to be false, report "no".

If $\tau_1 \not\equiv \tau_2$ holds, then this semi-algorithm reports "no". If $\tau_1 \equiv \tau_2$ holds and τ_1 and τ_2 have a finite Böhm tree, then it reports "yes". If $\tau_1 \equiv \tau_2$ holds and τ_1 and τ_2 have an infinite Böhm tree, then the semi-algorithm diverges. □

In the prototype implementation [28], this semi-algorithm is improved in two ways.

First, G_{k+1} is defined as $\langle G_k \rangle \setminus \bigcup_{j \leq k} G_j$. That is, any goal that has already appeared during an earlier step is considered valid. Goals are compared up to renaming of their free variables.

Second, an equation of the form $\tau_1\ \tau_1' \equiv \tau_2\ \tau_2'$, where one of the two sides is *not* a head normal form, is heuristically decomposed into $\tau_1 \equiv \tau_2 \wedge \tau_1' \equiv \tau_2'$. (This is done in addition to the default behavior, which is to reduce both sides to head normal form before decomposing them.) This can have the effect of replacing a difficult goal, which would lead the semi-algorithm into a sequence of ever-growing goals, with a conjunction of simpler goals that the semi-algorithm is able to prove.

The semi-algorithm thus improved remains sound. Indeed, when it succeeds, the set of goals that have been examined forms an *hnf bisimulation up to η and up to context,* as studied by Lassen [15], which implies that these goals are valid. The semi-algorithm remains incomplete, but is strong enough to prove all of the equations required by the encoding presented in §4.

Why do we require types to be well-kinded? If we were to remove this condition, every type would be permitted. Still, the fact that $\equiv$ is a consistent λ-theory would be sufficient to prove that FORK is type-safe (§3). However, type equality would then become undecidable, and (as a result) type-checking would become undecidable as well. This would make it pragmatically more difficult to build well-typed FORK terms, as we do in §4. If we do insist that types are well-kinded, on the other hand, then every type is solvable and (as a result) type equality and type-checking are semi-decidable, a pragmatically valuable property.

Another potential reason why kinds should be well-formed and types should be well-kinded is that these conditions make it possible to interpret kinds in a category of ultrametric spaces and to interpret types as inhabitants of kinds, in the style of Birkedal *et al.* [8].

$$
\begin{array}{lll}
\kappa ::= \star \mid \kappa \to \kappa \mid \bullet\,\kappa & & \text{(co-inductively; see §2)} \\
\tau ::= \alpha \mid \lambda\alpha.\tau \mid \tau\ \tau & & \text{(see §2)} \\
\quad \mid \to \mid () \mid (\,,\,) \mid \forall_\kappa \mid \exists_\kappa & & \text{(type constants)} \\
t ::= x \mid \lambda x.t \mid t\ t & & \text{(functions)} \\
\quad \mid () & & \text{(unit)} \\
\quad \mid (t,t) \mid \text{let } (x,x) = t \text{ in } t & & \text{(pairs)} \\
\quad \mid \Lambda\alpha.t \mid t\ \tau & & \text{(universals)} \\
\quad \mid \text{pack } \tau, t \text{ as } \tau & & \text{(existentials)} \\
\quad \mid \text{unpack } \alpha, x = t \text{ in } t & & \\
\Gamma ::= \emptyset \mid \Gamma; \alpha : \kappa \mid \Gamma; x : \tau & &
\end{array}
$$

Figure 3. FORK: kinds, types, terms

$$
\begin{array}{ll}
1. & K \vdash\ () : \star \\
2. & K \vdash\ \to : \bullet\star \to \bullet\star \to \star \\
3. & K \vdash (\,,\,) : \bullet\star \to \bullet\star \to \star \\
4. & K \vdash\ \forall_\kappa : (\kappa \to \bullet^n \star) \to \bullet^n \star \\
5. & K \vdash\ \exists_\kappa : (\kappa \to \bullet^n \star) \to \bullet^n \star
\end{array}
$$

Figure 4. FORK: kind assignment: constants

$$
\begin{array}{lll}
\textsc{Var}\ \ \Gamma \vdash x : \Gamma(x) & \textsc{Abs}\ \ \dfrac{\Gamma \vdash \tau_1 : \bigcirc\star \qquad \Gamma; x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x.t : \tau_1 \to \tau_2} & \\
\textsc{App}\ \ \dfrac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1\ t_2 : \tau_2} & & \textsc{Unit}\ \ \Gamma \vdash () : () \\
(\,,\,)\text{-}\textsc{Intro}\ \ \dfrac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (t_1, t_2) : (\tau_1, \tau_2)} & (\,,\,)\text{-}\textsc{Elim}\ \ \dfrac{\Gamma \vdash t_1 : (\tau_1, \tau_2) \qquad \Gamma; x_1 : \tau_1; x_2 : \tau_2 \vdash t_2 : \tau}{\Gamma \vdash \text{let } (x_1, x_2) = t_1 \text{ in } t_2 : \tau} & \\
\forall\text{-}\textsc{Intro}\ \ \dfrac{\Gamma; \alpha : \kappa \vdash t : \tau \qquad \alpha \# \Gamma}{\Gamma \vdash \Lambda\alpha.t : \forall_\kappa\ (\lambda\alpha.\tau)} & & \forall\text{-}\textsc{Elim}\ \ \dfrac{\Gamma \vdash t : \forall_\kappa\ \tau_1 \qquad \Gamma \vdash \tau_2 : \bigcirc\kappa}{\Gamma \vdash t\ \tau_2 : \tau_1\ \tau_2} \\
\exists\text{-}\textsc{Intro}\ \ \dfrac{\Gamma \vdash t : \tau_1\ \tau_2 \qquad \Gamma \vdash \exists_\kappa\ \tau_1 : \bigcirc\star \qquad \Gamma \vdash \tau_2 : \bigcirc\kappa}{\Gamma \vdash \text{pack } \tau_2, t \text{ as } \exists_\kappa\ \tau_1 : \exists_\kappa\ \tau_1} & & \\
\exists\text{-}\textsc{Elim}\ \ \dfrac{\Gamma \vdash t_1 : \exists_\kappa\ \tau_1 \qquad \alpha \# \Gamma, \tau_2 \qquad \Gamma; \alpha : \kappa; x : (\tau_1\ \alpha) \vdash t_2 : \tau_2}{\Gamma \vdash \text{unpack } \alpha, x = t_1 \text{ in } t_2 : \tau_2} & & \textsc{Conversion}\ \ \dfrac{\Gamma \vdash t : \tau_1 \qquad \Gamma \vdash \tau_2 : \bigcirc\star \qquad \tau_1 \equiv \tau_2}{\Gamma \vdash t : \tau_2}
\end{array}
$$

Figure 5. FORK: type assignment

3. FORK

In System F_ω, a system of simple finite kinds is used to classify types. In FORK, instead, Nakano's system is used. As a result, FORK is an extension of F_ω. FORK retains type safety, but abandons strong normalization.

Kinds and types Kinds and types (Figure 3) are as presented previously (§2), except a number of type constants are introduced, as is standard in F_ω. These constants are assigned kinds by the axioms in Figure 4.

$$
\begin{array}{rcl}
(\lambda x.t_1)\ t_2 & \longrightarrow & [x \mapsto t_2]t_1 \\
\text{let } (x_1, x_2) = (t_1, t_2) \text{ in } t & \longrightarrow & [x_1 \mapsto t_1, x_2 \mapsto t_2]t \\
(\Lambda\alpha.t)\ \tau & \longrightarrow & [\alpha \mapsto \tau]t \\
\text{unpack } \alpha, x = & & \\
\quad (\text{pack } \tau_2, t_1 \text{ as } \tau_1) \text{ in } t_2 & \longrightarrow & [\alpha \mapsto \tau_2, x \mapsto t_1]t_2 \\
C[t_1] & \longrightarrow & C[t_2] \\
 & & \text{if } t_1 \longrightarrow t_2
\end{array}
$$

Figure 6. FORK: reduction semantics

It may come as a surprise that the function and product type constructors have kind $\bullet\star \to \bullet\star \to \star$, as opposed to $\star \to \star \to \star$ in F_ω. This means that these constructors are *contractive* in both arguments. An intuitive reason why they should be viewed as contractive is precisely that they are type *constructors*, as opposed to type *operators:* they produce syntax. For instance, an application of the function type constructor to two arbitrary types τ_1 and τ_2 yields a type that is well-defined down to depth 1—it has an arrow at its root— regardless of how τ_1 and τ_2 might behave. The FORK axioms are more liberal than their F_ω counterparts, and crucially so. For instance, the former allow the recursive type $\mu\alpha.\alpha \to \alpha$ to have kind $\star$, while the latter would make this type ill-kinded. Naturally, the encoding of general references into FORK (§4) relies on the existence of such recursive types.

Because the function and product type constructors have kind $\bullet\star \to \bullet\star \to \star$, one cannot expect the types that classify values to always have kind $\star$, as in F_ω. Instead, in FORK, the types that classify values have kind $\bullet^n\star$, for some $n \in \mathbb{N}$. We write $K \vdash \tau : \bigcirc\kappa$ when $K \vdash \tau : \bullet^n\,\kappa$ holds for some $n \in \mathbb{N}$.

The universal and existential type constructors $\forall_\kappa$ and $\exists_\kappa$ are not considered contractive. Because the types that classify values can have kind $\bullet^n\star$ for any $n \in \mathbb{N}$, it is natural for these two constants to admit every kind of the form $(\kappa \to \bullet^n\star) \to \bullet^n\star$, as opposed to just $(\kappa \to \star) \to \star$ in F_ω.

The properties stated in the previous section (§2) remain valid in the presence of type constants of arbitrary kind.

Type assignment The syntax of terms t and type environments Γ is standard (Figure 3). A type environment Γ can play the role of a kind environment K.

The type assignment judgement $\Gamma \vdash t : \tau$ is defined in Figure 5. The typing rules are identical to their F_ω counterparts up to a few details, which we now discuss.

The type conversion rule (CONVERSION) relies on the notion of type equality that was defined earlier (§2).

The types that classify values have kind $\bigcirc\star$. This is reflected in the following definition and lemma:

Definition 3.1 *The empty type environment is well-formed. The type environment $\Gamma; \alpha : \kappa$ is well-formed if Γ is well-formed and $\alpha \# \Gamma$. The type environment $\Gamma; x : \tau$ is well-formed if Γ is well-formed and $\Gamma \vdash \tau : \bigcirc\star$.* ◇

Lemma 3.2 *If Γ is well-formed, then $\Gamma \vdash t : \tau$ implies $\Gamma \vdash \tau : \bigcirc\star$.* ◇

In the following, we restrict our attention to well-formed type environments.

The type application rule ($\forall$-ELIM) states that a universally quantified variable of kind κ can be instantiated with a type of kind $\bigcirc\kappa$. The rule $\exists$-INTRO is similarly relaxed. This makes sense thanks to the following type substitution lemma, which is later exploited in the proof of subject reduction:

$$
\begin{array}{rl}
C ::= & \lambda x.[] \mid [] \; e \mid e \; [] \mid \\
& ([], e) \mid (e, []) \mid \pi_i \; [] \\
S ::= & [] \mid S[\text{bind } ([], e)]
\end{array}
$$

$$
\begin{array}{rcl}
(\lambda x.e_1) \; e_2 & \to & [x \mapsto e_2]e_1 \\
\pi_i \; (e_1, e_2) & \to & e_i \\
C[e] & \to & C[e'] \\
& & \text{if } e \to e' \\
\\
(s, S, e_1) & \to & (s, S, e_2) \\
& & \text{if } e_1 \to e_2 \\
(s, S, \text{bind } (e_1, e_2)) & \to & (s, S[\text{bind } ([], e_2)], e_1) \\
(s, S[\text{bind } ([], e_2)], \text{return } e_1) & \to & (s, S, e_2 \; e_1) \\
(s, S, \text{new } e) & \to & (s\{\ell \mapsto e\}, S, \text{return } \ell) \\
& & \text{where } \ell = |s| \\
(s, S, \text{read } \ell) & \to & (s, S, s(\ell)) \\
& & \text{if } \ell < |s| \\
(s, S, \text{write } (\ell, e)) & \to & (s\{\ell \mapsto e\}, S, \text{return } ()) \\
& & \text{if } \ell < |s|
\end{array}
$$

Figure 7. System F: untyped reduction semantics

Lemma 3.3 *Let* $\Gamma_1 \vdash \tau_1 : \bigcirc\kappa$. *If* $\Gamma_1; \alpha : \kappa; \Gamma_2$ *is well-formed, then* $\Gamma_1; [\alpha \mapsto \tau_1]\Gamma_2$ *is well-formed. Furthermore,* $\Gamma_1; \alpha : \kappa; \Gamma_2 \vdash t : \tau_2$ *implies* $\Gamma_1; [\alpha \mapsto \tau_1]\Gamma_2 \vdash [\alpha \mapsto \tau_1]t : [\alpha \mapsto \tau_1]\tau_2$. ◇

It is possible to give a syntax-directed presentation of the type system, where the conversion rule is merged with the other rules. This allows type-checking to be performed in a standard bottom-up fashion. That is, provided every Λ-bound variable carries an explicit kind and every λ-bound variable carries an explicit type, the knowledge of Γ and t is sufficient to reconstruct a type τ (if one exists) such that $\Gamma \vdash t : \tau$ holds. The prototype implementation of FORK follows this scheme.

Type soundness FORK is equipped with a standard reduction semantics (Figure 6). Reduction is permitted under an arbitrary context. *Values* are defined as follows:

$$v ::= \lambda x.t \mid () \mid (v, v) \mid \Lambda\alpha.v \mid \text{pack } \tau, v \text{ as } \tau$$

The system enjoys the following properties:

Lemma 3.4 (Subject reduction) $\Gamma \vdash t_1 : \tau$ *and* $t_1 \longrightarrow t_2$ *imply* $\Gamma \vdash t_2 : \tau$. ◇

Definition 3.5 *A term* t *is* well-typed *iff* $\Gamma \vdash t : \tau$ *holds, where* Γ *binds only type variables (no term variables).* ◇

Lemma 3.6 (Progress) *A well-typed term either reduces or is a value.* ◇

Theorem 3.7 (Type soundness) *A well-typed term either diverges or reduces (in zero or more steps) to a value.* ◇

4. Encoding general references into FORK

4.1 The source calculus

The source language of the encoding is a monadic presentation of System F with general references. It is due to Peyton Jones and Wadler [25, §5.3].

The terms are the standard terms of System F, extended with the monadic constants return and bind; with the constants new, read, and write for allocating, reading and writing references; and with memory locations ℓ, which we take to be natural numbers.

$$
\begin{array}{l}
e ::= x \mid \lambda x.e \mid e \; e \mid \Lambda\alpha.e \mid e \; T \mid () \mid (e, e) \mid \pi_i \; e \mid \\
\quad \text{return} \mid \text{bind} \mid \text{new} \mid \text{read} \mid \text{write} \mid \ell
\end{array}
$$

$$\text{F-VAR} \quad E_1; x : T; E_2 \vdash x : T$$

$$\text{F-ABS} \quad \frac{E; x : T_1 \vdash e : T_2}{E \vdash \lambda x.e : T_1 \to T_2}$$

$$\text{F-APP} \quad \frac{E \vdash e_1 : T_1 \to T_2 \qquad E \vdash e_2 : T_1}{E \vdash e_1 \; e_2 : T_2}$$

$$\text{F-}\forall\text{-INTRO} \quad \frac{E; \alpha \vdash e : T}{E \vdash \Lambda\alpha.e : \forall\alpha.T}$$

$$\text{F-}\forall\text{-ELIM} \quad \frac{E \vdash e : \forall\alpha.T_1}{E \vdash e \; T_2 : [\alpha \mapsto T_2]T_1}$$

$$\text{F-UNIT} \quad E \vdash () : ()$$

$$\text{F-}(,)\text{-INTRO} \quad \frac{E \vdash e_1 : T_1 \qquad E \vdash e_2 : T_2}{E \vdash (e_1, e_2) : (T_1, T_2)}$$

$$\text{F-}(,)\text{-ELIM} \quad \frac{E \vdash e : (T_1, T_2)}{E \vdash \pi_i \; e : T_i}$$

$$\text{F-RETURN} \quad E \vdash \text{return} : \forall\alpha.\alpha \to M \; \alpha$$

$$\text{F-BIND} \quad E \vdash \text{bind} : \forall\alpha.\forall\beta.(M \; \alpha, \alpha \to M \; \beta) \to M \; \beta$$

$$\text{F-NEW} \quad E \vdash \text{new} : \forall\alpha.\alpha \to M \; (\text{ref } \alpha)$$

$$\text{F-READ} \quad E \vdash \text{read} : \forall\alpha.\text{ref } \alpha \to M \; \alpha$$

$$\text{F-WRITE} \quad E \vdash \text{write} : \forall\alpha.(\text{ref } \alpha, \alpha) \to M \; ()$$

Figure 8. System F: typing rules

The types are:

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

where M is the monad. The definition of the typing judgement $E \vdash e : T$ appears in Figure 8. This definition concerns programs whose execution has not begun and (hence) that do not contain memory locations. A definition of the encoding of locations is needed only as part of the semantics preservation argument (§4.10).

The operational semantics is defined for type-erased terms, that is, terms where the type abstraction and type application constructs have been erased. The semantics, which appears in Figure 7, is organized in two layers, following Moggi and Sabry [19]. First, *simplification* of terms is defined: it is the compatible closure of β-reduction. Then, *reduction* of configurations is defined, where a configuration is a triple of a store s, an evaluation stack S, and a term e, which represents a computation (that is, it presumably has a type of the form $M \; T$). Because memory locations are natural numbers, a store is just a list of terms. We write *nil* for the empty store. We write $|s|$ for the length of the store s. We write $s(\ell)$ for the term found in the store s at location ℓ. We write $s\{\ell \mapsto e\}$ for the extension or update of the store s at location ℓ with the term e. An evaluation stack is just a list of suspended applications of bind.

In the following, we present an encoding of this calculus into FORK. We begin with a series of definitions of FORK kinds, types, and terms (§4.2–§4.6), whose well-formedness has been machine-checked by the prototype implementation of FORK [28]. Then, we define the encoding (§4.7 and §4.8), a type-directed transformation of the source calculus into FORK. Finally, we prove that the encoding is type-preserving (§4.9) and semantics-preserving (§4.10).

4.2 Fragments

In order to type-check the store, which is basically a sequence of values, we need sequences of base types, where a *base type* is a type of kind $\star$. Because the store grows with time, we often use

kind *fragment* =
  $\star \to \star$
type *fnil* : *fragment* =
  λ*tail*. *tail*
type @ : *fragment* $\to$ *fragment* $\to$ *fragment* =
  $\lambda f_1\, f_2$ *tail*. f_1 (f_2 *tail*)
type *snoc* : *fragment* $\to \star \to$ *fragment* =
  λ*f*. λ*data*. λ*tail*. *f* (*data*, *tail*)

Figure 9. Fragments

type *array* : $\bullet$ *fragment* $\to \star$
type *index* : $\bullet$ *fragment* $\to \bullet \star \to \star$
term *array_empty* : *array fnil*
term *array_extend* :
  $\forall f$ *data*. *array f* $\to$ *data* $\to$ *array* (*f* ‘*snoc*‘ *data*)
term *array_read* :
  $\forall f$ *data*. *array f* $\to$ *index f data* $\to$ *data*
term *array_write* :
  $\forall f$ *data*. *array f* $\to$ *index f data* $\to$ *data* $\to$ *array f*
term *array_end_index* :
  $\forall$*f*. *array f* $\to \forall$*data*. *index* (*f* ‘*snoc*‘ *data*) *data*
term *index_monotonic* :
  $\forall f_1\, f_2$ *data*. *index* f_1 *data* $\to$ *index* (f_1 ‘@‘ f_2) *data*

Figure 10. Arrays

such a sequence to describe only part of the store, and concatenate multiple such sequences to obtain a description of the complete store. For this reason, we refer to such a sequence as a *fragment.*

Because we intend to parameterize types over fragments, to quantify over fragments, etc., fragments must be types (of a suitable kind), and the basic operations over fragments must be type operators.

This is done as follows (Figure 9). The kind *fragment* is $\star \to \star$. The empty fragment *fnil* is the identity function. Fragment concatenation @ is function composition. It is easy to check, with respect to type equality $\equiv$, that *fnil* is a left unit and right unit for @, and that @ is associative. The extension of a fragment *f* with a base type *data*, written *snoc f data* or more pleasantly *f* ‘*snoc*‘ *data*, is λ*tail*. *f (data, tail)*. The type *(data, tail)* is the application of the “product” type constructor (,) (Figure 4) to the base types *data* and *tail*.

4.3 Arrays

We wish to represent the store as an array. Thus, FORK must support arrays. We need *heterogeneous* arrays: it must be permitted for different array elements to have different types. We need *safe* arrays: out-of-bound array accesses must be statically forbidden. (As Levy puts it, it “is unnatural [...] to specify what happens when we read a non-existent cell, something that can never occur in reality” [16].) We need *extensible* arrays: there must be a way of creating a new array by appending a new cell at the end of an existing array, and any valid index into the earlier array must remain a valid index into the new array.

The signature in Figure 10 fulfills these requirements. It introduces a couple of abstract type constructors for arrays and indices, as well as a number of operations over arrays.

The type *array f* describes a heterogeneous array whose elements are described by the fragment *f*. The type *index f data* represents the address of a cell of base type *data* within an array of type *array f*. The type operators *array* and *index* are contractive: as far as the user of the array abstraction is concerned, they can be thought of as type constructors.

kind *world* =
  $\bullet$ *world* $\to$ *fragment*
type *nil* : *world* =
  λ*x*. *fnil*
type *o* : *world* $\to$ *world* $\to$ *world* =
  $\lambda w_1\, w_2\, x.\ w_1\ (w_2$ ‘*o*‘ *x*) ‘@‘ $w_2\ x$

Figure 11. Worlds

The zero length array, *array_empty*, is described by the fragment *fnil*. Array extension, *array_extend*, is described using the fragment extension operation, *snoc*. Reading and writing are permitted by *array_read* and *array_write*. Writing is type-invariant: the old and new array elements both have type *data*. The operation *array_end_index* returns the end index of an array of type *array f*. It is not a valid index into this array, but becomes a valid index once the array is extended with a new cell: this is expressed by the type $\forall$*data. index (f ‘snoc‘ data) data*. The operation *index_monotonic* witnesses the fact that a valid index into a smaller array is also a valid index into a larger array: this is expressed in terms of fragment concatenation. This operation is a coercion: its semantics is the identity.

There are two ways of making the types and operations described by this signature available in FORK.

The first way, which we follow, is to implement this signature. This can be done by representing array data as a tagless singly-linked list (that is, as a sequence of nested pairs) and by representing an array index as a pair of functions for reading and writing at this index. (The read function, for instance, encapsulates a sequence of pair projections.) The code is about a hundred lines [28]. It does not use any recursion: it is in fact expressed within F_ω. This implementation of arrays shows that, in principle, it is not necessary to extend FORK with primitive arrays. It is of course inefficient: reading and writing have linear time complexity.

The second way would be to extend FORK with primitive arrays, that is, to consider the signature of Figure 10 as a set of axioms, to extend the operational semantics of FORK with new reduction rules for arrays, and to extend its type soundness proof. This would allow a more efficient implementation of array access, although efficiency seems of little concern here.

4.4 Worlds

Our arrays are extensible *in width.* This helps us model dynamic allocation, that is, the fact that the store grows with time. There remains to model higher-order store, that is, the fact that the store can contain references and functions, whose type, in the encoding, depends on the shape of the store. This dependency means that, as the store grows in width, the type of an existing store cell evolves. We say that the store also grows *in depth*.

An example may help illustrate this phenomenon. Consider the ML program “let x_1 = ref $(\lambda x.x)$ in let x_2 = ref 0 in $x_1 := (\lambda x.!x_2)$”. When the reference cell x_1 is first allocated, it contains a function that has type int $\to$ int unconditionally. However, x_1 is later updated with a function that has type int $\to$ int only under a store where the cell x_2 exists and holds an integer value. Thus, the type of the contents of the cell x_1 evolves with time.

In order to reflect this, we introduce *worlds* (Figure 11). A world is an open-ended description of a store fragment. More precisely, a world is a fragment that is itself parameterized over a world. The kind *world* is recursive. It is well-formed, because the recursion goes through the “later” constructor $\bullet$. A world is a *contractive* function of a world: it produces some structure before it uses its argument.

The empty world *nil* is the constant function that returns the empty fragment. World composition, w_1 ‘*o*‘ w_2, can be described as

```
kind stype =
  • world → ⋆
type box : stype → stype =
  λa x. ∀y. a (x `o` y)
type unit : stype =
  λx. ()
type pair : stype → stype → stype =
  λa b x. (a x, b x)
type univ : (stype → stype) → stype =
  λbody x. ∀a. body a x
type arrow : stype → stype → stype =
  λa b x. box a x → box b x
type monad : stype → stype =
  λa x. store x → outcome a x
type outcome : stype → stype =
  λa x. ∃y. (box a (x `o` y), store (x `o` y))
type store : • world → ⋆ =
  λx. ∀y. array (x y)
type ref : stype → stype =
  λa x. ∀y. index (x y) (a (x `o` y))
```

Figure 12. Semantic types

the result of extending w_1 *in depth and in width* with w_2. Naturally, its definition is recursive. We invite the reader to check that it is well-typed in Nakano's system, using the derived kind assignment rule for recursive type definitions.

The empty world *nil* is a left unit and right unit for world composition. Furthermore, world composition is associative. This fact is non-obvious; fortunately, the semi-algorithm for type equality (§2) proves it (and others like it) without assistance.

In the following, by convention, the variable w has kind *world*, while the variables x and y have kind • *world*.

4.5 Semantic types

A type in the source language is translated to a *semantic type*, that is, a contractive function of worlds to base types (Figure 12).

Let a be a semantic type and x be a world. A value v of type $a\ x$ is valid now, in world x. Is it valid also in every future world? That is, is it the case that v also has type $a\ (x\ `o`\ y)$ for every y? In general, there is no guarantee that this is so. Where we need this to be the case, we are explicit about this requirement and use a value of type *box a x*, where the semantic type operator *box* builds in a universal quantification over future worlds. This operator is known as the *necessity modality* [5]. A semantic type of the form *box a* is known as *necessary* [5] or *hereditary* [14].

Remark There are semantic models (see e.g. [31]) where necessity is built into the world equation, so that (the analogue of) every type is hereditary. Here, this is not the case. FORK does not have a "monotonic arrow" at the kind level, so there seems to be no way of building necessity into worlds. We follow Appel *et al.* [5] and Hobor *et al.* [14] and explicitly use the *box* modality to keep track of which types are hereditary. ◇

It is worth noting that bounded quantification over all future worlds z (that is, $\forall z \geq x.\ \tau$) is expressed here in terms of ordinary quantification over a world extension y (that is, $\forall y.\ [z \mapsto x\ `o`\ y]\tau$). In this encoding, the associativity of world composition expresses the transitivity of the world ordering.

A hereditary value is valid in every future world, hence is hereditary in every future world. This is expressed by defining a coercion *forward* of type $\forall a\ x\ y.\ box\ a\ x \to box\ a\ (x\ `o`\ y)$.

The encoding of unit, pairs, and quantifiers is straightforward: the world parameter x is just passed down. The encoding of arrows states that functions require a hereditary argument and produce a hereditary result. This is expressed by the type $box\ a\ x \to box\ b\ x$.

Remark The reader might have expected instead the type $box\ (\lambda x.\ a\ x \to b\ x)$, which guarantees that the function itself is hereditary. This type is more general than $box\ a\ x \to box\ b\ x$.

However, this choice would cause a difficulty in the construction of certain functions, such as *new* (Figure 15). There, we must be able to argue that the argument v is hereditary, because this value will be written into the store, and the store holds hereditary values. If v has type $box\ a\ x$, the argument is trivial, whereas if it has type $a\ x$, it is not clear that v is hereditary.

Our current choice, on the other hand, does not seem to cause any difficulty. It leads to a style where every variable in the type environment has a boxed type (see Definition 4.2) and every expression has a boxed type (see the statement of Theorem 4.3). In short, the fact that the encoding $[\![T]\!]$ of a type T is not always hereditary does not hurt us: we use explicit boxes where needed.

Another route would be to set things up so that the encoding $[\![T]\!]$ of every type T is a hereditary semantic type. One would then build this information into the translation via coercions: wherever one abstracts over a semantic type variable a, one would also abstract over a coercion of type $\forall x.\ a\ x \to box\ a\ x$. It seems plausible that this would succeed; it could be investigated as part of future work.◇

The encoding of the monad is standard [16]. A computation requires a store in world x, and produces a pair of a (hereditary) result and a new store in some future world $x\ `o`\ y$. We use the abbreviation *outcome a x* for the type of such a pair.

A store in world w contains values that are valid in world w and in future worlds. That is, a store is an array of hereditary values. This is expressed by defining *store w* as $\forall x.\ array\ (w\ x)$. In other words, a store is of course of fixed *width,* but is polymorphic in *depth.*

A reference of type *ref a* in world x is, roughly speaking, an index that allows reading or writing data of type $a\ x$ within an array of type *store x*. More precisely, universal quantification over a world extension y is again used to guarantee that references are hereditary. By direct appeal to *index_monotonic*, it is possible to define a coercion *box_ref* of type $\forall a.\ ref\ a\ x \to box\ (ref\ a)\ x$.

We have reviewed the encoding of every type constructor of the source language. Thus, any source-level type T is translated to a semantic type $[\![T]\!]$. (The inductive definition of this translation cannot be expressed within FORK.)

Definition 4.1 *The encoding* $[\![T]\!]$ *of a type* T *is defined as follows:*

$$\begin{aligned}
[\![\alpha]\!] &= \alpha \\
[\![()]\!] &= unit \\
[\![T_1 \to T_2]\!] &= arrow\ [\![T_1]\!]\ [\![T_2]\!] \\
[\![(T_1, T_2)]\!] &= pair\ [\![T_1]\!]\ [\![T_2]\!] \\
[\![\forall \alpha.T]\!] &= univ\ (\lambda \alpha.[\![T]\!]) \\
[\![M\ T]\!] &= monad\ [\![T]\!] \\
[\![ref\ T]\!] &= ref\ [\![T]\!]
\end{aligned}$$

where the combinators that appear in the right-hand sides of these equations are defined in Figure 12. ◇

4.6 Memory allocation

When a new reference is allocated, the width of the array that represents the store is increased by one. If the allocation takes place in world x, and if the newly created reference is initialized with a value v of type $box\ a\ x$, what is the new world after allocation?

This new world must be the composition of x and of a world of width one, which we refer to as a *cell.* That is, the new world must be $x\ `o`\ cell\ a\ x$, for an appropriate definition of *cell*, an operator that maps a and x to a world.

```
type cell : stype → • world → world =
  λa x y tail. (a (x `o` cell a x `o` y), tail)
term store_extend :
  ∀x a. store x → box a x → store (x `o` cell a x)
term store_end_index :
  ∀x a. store x → ref a (x `o` cell a x)
```

Figure 13. Memory allocation

The definition of *cell* appears in Figure 13. As above, the parameter x represents the world before allocation, or a past world. The parameter y represents a depth extension, or a future world, while *tail* represents a width extension. The type *cell a x y tail*, a type of kind $\star$, is a product of the types *a (. . .)* and *tail*. (This is consistent with our definition of fragment extension, *snoc*, in Figure 9.) The semantic type a is applied to the composite world *x `o` (cell a x) `o` y*, reflecting the manner in which the final world is obtained: starting in world x, first a memory cell is allocated, then the world is extended with y. Thus, the definition of *cell* is recursive.

It is worth noting that a value v of type *box a x* also has every type of the form *a (x `o` cell a x `o` y)*, simply because the latter is a polymorphic instance of the former. Thus, the value that is used to initialize the cell is indeed a suitable value for the cell in every future world y.

How do we ascertain that this definition of *cell* is right? The proof is in the fact that the terms *store_extend* and *store_end_index*, which respectively construct the new store and the address of the new reference, have the types shown in Figure 13. The definitions of these terms [28] are a couple lines each. Up to a number of suitable type abstractions and applications, *store_extend* is just *array_extend*, while *store_end_index* is just *array_end_index*.

4.7 Encoding the monadic constants

All of the infrastructure is now in place. We are in a position to encode the constants *return*, *bind*, *new*, *read*, and *write*. To do so, we must define five terms that admit the types shown in Figure 14 and that adequately implement the store-passing machinery.

We present and explain only the definitions of the term *new* (Figure 15). The definitions of *return*, *bind*, *read*, and *write* are omitted and can be found online [28]. We believe that the definition of *new* is representative, so that the reader who has studied it could (if he or she so wished!) reconstruct the other definitions as an exercise.

The structure of the definition of *new* is in large part imposed by its type. By definition of *box*, *univ*, *arrow*, and *monad* (Figure 12), the desired type for *new*:

$$box\ (univ\ (\lambda a.\ a\ `arrow`\ monad\ (ref\ a)))\ nil$$

is equal to:

$$\forall x.\ \forall a.\ box\ a\ x \to \forall y.\ store\ (x\ `o`\ y) \to outcome\ (ref\ a)\ (x\ `o`\ y)$$

Thus, the definition of *new* begins with abstractions over a world x, a semantic type a, a value v of type *box a x*, a world y, and a store s of type *store xy*. The type *xy* is defined on the fly as a local abbreviation for *x `o` y*.

It might seem strange that we must abstract over *two* successive world extensions, x and y. This is required by the monadic structure of the source language: the world x corresponds to the point in time where *ref* is applied to a value v, while the world *x `o` y* corresponds to the point in time where the computation *ref v* is run.

After accepting these parameters, *new* must produce a result of type *outcome (ref a) xy*. By definition of *outcome* (Figure 12), this is equal to:

$$\exists z.\ (box\ (ref\ a)\ (xy\ `o`\ z),\ store\ (xy\ `o`\ z))$$

```
term return :
  box (univ (λa. a `arrow` monad a)) nil
term bind :
  box (univ (λa. univ (λb.
    (monad a `pair` (a `arrow` monad b)) `arrow` monad b
  ))) nil
term new :
  box (univ (λa. a `arrow` monad (ref a))) nil
term read :
  box (univ (λa. ref a `arrow` monad a)) nil
term write :
  box (univ (λa. (ref a `pair` a) `arrow` monad unit)) nil
```

Figure 14. Encoding the monadic constants: declarations

```
term new : box (univ (λa. a `arrow` monad (ref a))) nil =
  Λ x a.
  λv : box a x.
  Λ y.
  type xy = x `o` y in
  λs : store xy.
    type c = cell a xy in
    pack c, (
      box_ref [a] [xy `o` c] (store_end_index [xy] [a] s),
      store_extend [xy] [a] s (forward [a] [x] [y] v)
    )
    as outcome (ref a) xy
```

Figure 15. Encoding the monadic constants: definition of *new*

This type is existentially quantified over a world extension z, which represents the new storage allocated by the computation. Here, this new storage is a single cell, which holds a value of semantic type a. So, according to our earlier discussion (§4.6), the concrete witness for z should be *cell a xy*. Thus, c is defined as a local abbreviation for *cell a xy*, and the construct **pack** *c, ...* **as** *outcome (ref a) xy* is used in order to build an existential package.

Inside this package should be a pair of a newly allocated memory location, at type *box (ref a) (xy `o` c)*, and an extended store, at type *store (xy `o` c)*.

The next available memory location is obtained by applying *store_end_index* (Figure 13) to suitable type arguments and to the current store s. This application has type *ref a (xy `o` c)*. This is what was required, except a leading *box* is missing: we must justify that this memory location is valid not only now, but also in the future. This is achieved via an application of the coercion *box_ref*, which was mentioned earlier (§4.5).

It seems that the new store should be obtained by applying *store_extend* (Figure 13) to the current store s and to the initial value v of the newly allocated cell. However, there is a slight difficulty: s and v do not inhabit the same world. Indeed, s has type *store (x `o` y)*, while v has type *box a x*. Thus, before *store_extend* can be applied, the value v must be moved from the world x into the world *x `o` y*. Fortunately, v is hereditary, so this is easily achieved by applying the coercion *forward* (§4.5).

The definition of *new* is somewhat complex due to the many type annotations. However, by erasing all type abstractions, applications, and abbreviations, as well as all coercion applications, one finds that *new* is just $\lambda v.\lambda s.(store\_end\_index\ s, store\_extend\ s\ v)$. Thus, it is easy to informally convince oneself that the untyped translation that underlies our encoding is indeed a standard store-passing translation. A formal argument of semantics preservation appears further on (§4.10).

ENCODE-VAR

$$E_1; x : T; E_2 \vdash x \rightsquigarrow \Lambda y.x\ (w_{E_2}\ `o`\ y) : T$$

ENCODE-ABS

$$\frac{E; x : T_1 \vdash e \rightsquigarrow t : T_2}{E \vdash \lambda x.e \rightsquigarrow \Lambda w_x.\lambda x.t : T_1 \to T_2}$$

ENCODE-APP

$$\frac{E \vdash e_1 \rightsquigarrow t_1 : T_1 \to T_2 \qquad E \vdash e_2 \rightsquigarrow t_2 : T_1}{E \vdash e_1\ e_2 \rightsquigarrow t_1\ \mathit{nil}\ t_2 : T_2}$$

ENCODE-∀-INTRO

$$\frac{E; \alpha \vdash e \rightsquigarrow t : T}{E \vdash \Lambda\alpha.e \rightsquigarrow \Lambda y.\Lambda(\alpha : \mathit{stype}).(t\ y) : \forall\alpha.T}$$

ENCODE-∀-ELIM

$$\frac{E \vdash e \rightsquigarrow t : \forall\alpha.T_1}{E \vdash e\ T_2 \rightsquigarrow \Lambda y.(t\ y\ [\![T_2]\!]) : [\alpha \mapsto T_2]T_1}$$

ENCODE-UNIT

$$E \vdash () \rightsquigarrow \Lambda y.() : ()$$

ENCODE-(,)-INTRO

$$\frac{E \vdash e_1 \rightsquigarrow t_1 : T_1 \qquad E \vdash e_2 \rightsquigarrow t_2 : T_2}{E \vdash (e_1, e_2) \rightsquigarrow \Lambda y.(t_1\ y, t_2\ y) : (T_1, T_2)}$$

ENCODE-(,)-ELIM

$$\frac{E \vdash e \rightsquigarrow t : (T_1, T_2)}{E \vdash \pi_i\ e \rightsquigarrow \Lambda y.\text{let } (x_1, x_2) = t\ y \text{ in } x_i : T_i}$$

Figure 16. System F-to-FORK encoding: pure fragment

4.8 Encoding terms

We have defined the encoding of the five monadic constants. There remains to encode the pure fragment of the source language, that is, the terms of System F. The encoding is type-directed. It takes the form of an encoding judgement $E \vdash e \rightsquigarrow t : T$, which enriches the System F typing judgement: that is, $E \vdash e : T$ holds iff $E \vdash e \rightsquigarrow t : T$ holds for a certain t. The definition of this judgement appears in Figure 16. Over pure terms, the encoding is essentially the identity. It introduces type abstractions and applications in order to introduce and/or eliminate the *box* modality.

4.9 Type preservation

The following definition and theorem state precisely in what way the encoding is type-preserving.

Definition 4.2 *With each variable x, we associate a world variable w_x. Then, with a System F type environment E, we associate a world w_E, as follows:*

$$\begin{aligned} w_\emptyset &= \mathit{nil} \\ w_{E;\alpha} &= w_E \\ w_{E;x:T} &= w_E\ `o`\ w_x \end{aligned}$$

The encoding $[\![E]\!]$ of a type environment E is given by:

$$\begin{aligned} [\![\emptyset]\!] &= \emptyset \\ [\![E; \alpha]\!] &= [\![E]\!]; \alpha : \mathit{stype} \\ [\![E; x : T]\!] &= [\![E]\!]; w_x : \mathit{world}; x : \mathit{box}\ [\![T]\!]\ w_{E;x:T} \end{aligned}$$ ◇

Every abstraction λx in the source program gives rise to a sequence of two abstractions, of the form $\Lambda w_x.\lambda x$, in the translated program. This accounts for the fact that the world at the time a function is defined and the world at the time this function is applied are distinct: the latter is in general an extension of the former. The parameter w_x represents this extension.

Theorem 4.3 (Type preservation) *$E \vdash e \rightsquigarrow t : T$ implies $[\![E]\!] \vdash t : \mathit{box}\ [\![T]\!]\ w_E$.* ◇

Proof. In this proof, we do not consider the cases of the constants return, bind, new, read, and write, which have been machine-checked [28]. There remains one case for each of the rules in Figure 16.

∘ *Case* ENCODE-VAR. By hypothesis, under the environment $[\![E_1; x : T; E_2]\!]$, x has type:

$$\mathit{box}\ [\![T]\!]\ w_{E_1;x:T}.$$

By definition of *box*, this is:

$$\forall y.([\![T]\!]\ (w_{E_1;x:T}\ `o`\ y)).$$

Thus, the type application $x\ (w_{E_2}\ `o`\ y)$ has type:

$$[\![T]\!]\ (w_{E_1;x:T}\ `o`\ w_{E_2}\ `o`\ y),$$

that is,

$$[\![T]\!]\ (w_{E_1;x:T;E_2}\ `o`\ y).$$

Thus, the term $\Lambda y.x\ (w_{E_2}\ `o`\ y)$ has type:

$$\forall y.([\![T]\!]\ (w_{E_1;x:T;E_2}\ `o`\ y)),$$

that is,

$$\mathit{box}\ [\![T]\!]\ w_{E_1;x:T;E_2}.$$

∘ *Case* ENCODE-ABS. By the induction hypothesis, under the environment $[\![E; x : T_1]\!]$, the term t has type $\mathit{box}\ [\![T_2]\!]\ w_{E;x:T_1}$. That is, under the type environment:

$$[\![E]\!]; w_x : \mathit{world}; x : \mathit{box}\ [\![T_1]\!]\ (w_E\ `o`\ w_x),$$

the term t has type:

$$\mathit{box}\ [\![T_2]\!]\ (w_E\ `o`\ w_x).$$

There follows that the term $\Lambda w_x.\lambda x.t$ has type:

$$\forall w_x.(\mathit{box}\ [\![T_1]\!]\ (w_E\ `o`\ w_x) \to \mathit{box}\ [\![T_2]\!]\ (w_E\ `o`\ w_x)),$$

By definition of the encoding and by definition of *box*, this type is:

$$\mathit{box}\ [\![T_1 \to T_2]\!]\ w_E.$$

∘ *Case* ENCODE-APP. By the induction hypothesis, under the type environment $[\![E]\!]$, the term t_1 has type $\mathit{box}\ [\![T_1 \to T_2]\!]\ w_E$ and the term t_2 has type $\mathit{box}\ [\![T_1]\!]\ w_E$. By definition of *box* and by definition of the encoding, the former of these types is:

$$\forall w_x.(\mathit{box}\ [\![T_1]\!]\ (w_E\ `o`\ w_x) \to \mathit{box}\ [\![T_2]\!]\ (w_E\ `o`\ w_x)).$$

As a result, the application $t_1\ \mathit{nil}$ has type:

$$\mathit{box}\ [\![T_1]\!]\ w_E \to \mathit{box}\ [\![T_2]\!]\ w_E,$$

and the application $t_1\ \mathit{nil}\ t_2$ has type

$$\mathit{box}\ [\![T_2]\!]\ w_E.$$

∘ *Case* ENCODE-∀-INTRO. By the induction hypothesis, under the type environment $[\![E]\!]; \alpha : \mathit{stype}$, the term t has type

box $[\![T]\!]$ w_E. Thus, the term:

$$\Lambda y.\Lambda(\alpha : \mathit{stype}).(t\ y)$$

has type

$$\forall y.\forall(\alpha : \mathit{stype}).([\![T]\!]\ (w_E\ \text{`}o\text{`}\ y)).$$

By definition of *univ*, this type is:

$$\forall y.(\mathit{univ}\ (\lambda\alpha.[\![T]\!])\ (w_E\ \text{`}o\text{`}\ y)),$$

that is, by definition of *box*:

$$\mathit{box}\ (\mathit{univ}\ (\lambda\alpha.[\![T]\!]))\ w_E,$$

that is, by definition of the encoding,

$$\mathit{box}\ [\![\forall\alpha.T]\!]\ w_E.$$

◦ *Case* ENCODE-∀-ELIM. By the induction hypothesis, under the type environment $[\![E]\!]$, the term t has type:

$$\mathit{box}\ [\![\forall\alpha.T_1]\!]\ w_E.$$

As we saw in the previous case, this type is:

$$\forall y.\forall(\alpha : \mathit{stype}).([\![T_1]\!]\ (w_E\ \text{`}o\text{`}\ y)).$$

Thus, the term $\Lambda y.(t\ y\ [\![T_2]\!])$ has type:

$$\forall y.(([\alpha \mapsto [\![T_2]\!]][\![T_1]\!])\ (w_E\ \text{`}o\text{`}\ y)).$$

Because the encoding of types is compositional (i.e., commutes with type substitution), this type is:

$$\forall y.([\![[\alpha \mapsto T_2]T_1]\!]\ (w_E\ \text{`}o\text{`}\ y)),$$

that is, by definition of *box*:

$$\mathit{box}\ [\![[\alpha \mapsto T_2]T_1]\!]\ w_E.$$

◦ *Cases* ENCODE-UNIT, ENCODE-(,)-INTRO, ENCODE-(,)-ELIM. Left to the reader. □

4.10 Semantics preservation

The encoding is semantics-preserving. This is true even in the absence of a well-typedness hypothesis. This is not surprising: one might be tempted to say that the encoding is "obviously" a store-passing translation, and that a store-passing translation is "obviously" semantics-preserving. Nevertheless, it is worth checking this fact. The results presented in this section have been machine-checked using Coq; the development can be found online [28].

Because the rules ENCODE-∀-INTRO and ENCODE-∀-ELIM in Figure 16 encode type abstractions and type applications in terms of type abstractions and type applications, it is possible to define an untyped version of the encoding, which transforms untyped terms into untyped terms. This untyped encoding is a function. In the following, we write e for an untyped term of the source calculus, and we write $[\![e]\!]$ for its untyped encoding.

It is easy to check, by examination of Figure 16, that the untyped encoding function is the identity over the pure fragment of System F. As an immediate corollary, every simplification step in the source calculus is simulated by one reduction step in the target calculus:

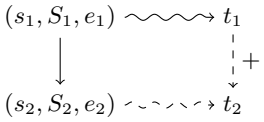
Lemma 4.4 $e_1 \to e_2$ *implies* $[\![e_1]\!] \longrightarrow [\![e_2]\!]$. ◇

We now wish to prove an analogous simulation diagram about the reduction of configurations. This requires defining the encoding of a memory location $[\![\ell]\!]$, the encoding of a store, the encoding of an evaluation stack, and the encoding of a configuration. These somewhat technical definitions are omitted, but can be found online [28].

It is then a matter of routine to check that one step of reduction in the source calculus is simulated by one or more steps of reduction in the target calculus. This is stated as follows.

In order to allow for some slack in the administrative reductions, the encoding of configurations is a relation, rather than a function. We write $(s, S, e) \rightsquigarrow t$ to indicate that the term t is an encoding of the configuration (s, S, e). Then, we have:

Lemma 4.5 (Simulation) *For a closed configuration* (s_1, S_1, e_1)*, the following diagram holds:*



◇

It is worth noting that, in order to establish $t_1 \longrightarrow^+ t_2$, we exploit the fact that reduction of FORK terms is permitted under arbitrary contexts. We use this flexibility, in particular, to perform reduction inside the components of a pair.

As an immediate consequence, we find that convergence is preserved by the encoding.

Lemma 4.6 (Convergence) *Let* e_1 *be a closed term of the source calculus. If the start configuration* $(\mathit{nil}, [], e_1)$ *reduces (in many steps) to some configuration of the form* $(s, [], \mathit{return}\ e_2)$*, then the term* $[\![e_1]\!]\ [\![\mathit{nil}]\!]$ *reduces (in many steps) to the pair* $([\![e_2]\!], [\![s]\!])$. ◇

Proving that divergence is also preserved by the encoding is more difficult. Of course, by Lemma 4.5, the existence of an infinite reduction sequence out of the start configuration $(\mathit{nil}, [], e_1)$ implies the existence of an infinite reduction sequence out of its encoding $[\![e_1]\!]\ [\![\mathit{nil}]\!]$. However, this is not the desired property. Because reduction of FORK terms is permitted under arbitrary contexts, the relation $\longrightarrow$ is non-deterministic, and the existence of an infinite reduction sequence is not an appropriate definition of divergence. Instead, we would like to consider that the term $[\![e_1]\!]\ [\![\mathit{nil}]\!]$ diverges if and only if it does not have a head normal form. An analogous phenomenon arises in the source calculus, where simplification is permitted under arbitrary contexts.

Here is an informal sketch of how this problem might be solved. In each of the source and target calculi, define notions of standard (that is, leftmost) and internal reduction. Prove that divergence (that is, the absence of a head normal form) is equivalent to the existence of an infinite standard reduction sequence. In the target calculus, prove that standard reduction and internal reduction commute. Takahashi [33] proves these facts in the pure λ-calculus.

Then, refine Lemma 4.5, by showing that one standard reduction step in the source calculus is simulated by a mixture of at least one standard reduction step and an arbitrary number of internal reduction steps in the target calculus. Use this fact, together with the property that standard reduction and internal reduction commute, to conclude that, if a source configuration admits an infinite standard reduction sequence, then so does its encoding. We have not yet attempted to machine-check this development.

5. Related work

Several store-passing translations have appeared in the literature. Moggi's state monad [18] relies on a fixed store type: it does not support dynamic memory allocation. Parameterised monads [6] allow the type of the store to vary with time and can be used to model systems of strong references with memory allocation and de-allocation. Similarly, Hoare Type Theory [23] extends type theory with a monad that is indexed with pre- and post-conditions: $\{P\}x : A\{Q\}$ is the type of computations that expect a store in state P and produce a value x of type A together with a new store in state Q. Hoare Type Theory is very expressive, yet does not support weak references. O'Hearn and Reynolds [24] translate two variations of Algol 60 into a purely functional calculus with

polymorphic and linear types, and compose this translation with a model of the target calculus to obtain models of the source languages. Charguéraud and Pottier [12] translate an expressive type-and-capability calculus, which supports strong references, into a purely functional calculus[2]. Pottier [26] extends Charguéraud and Pottier's work with an *anti-frame* rule that allows weak references to be defined in terms of strong references. However, it is not clear how Charguéraud and Pottier's store-passing translation could be extended to support the anti-frame rule. To the best of our knowledge, no typed store-passing translation for weak references has appeared in the literature.

The syntactic approach to type soundness [13, 35] deals with weak references via store types, which map memory addresses to types. The store type grows with time (this is part of the statement of subject reduction) and simultaneously describes the current store as well as all future stores. This is probably the simplest approach to type soundness for general references. However, it does not suggest how to design a type-preserving store-passing translation.

FORK is an ad hoc extension of System F_ω that allows non-terminating yet productive computation at the type level and (as a result) is able to express rich recursive types, far beyond the view of recursive types as regular trees that is commonly encountered in simpler type systems [4, 11]. There exist other calculi that permit productive computation at the type level: ΠΣ [3] and MiniAgda [1] come to mind. Perhaps these calculi could serve as target languages for a type-preserving store-passing translation of general references.

6. Directions for future work

In this paper, we have equipped the source and target calculi with pairs. Is it possible to equip both calculi with sums and extend the definition of the encoding? We believe so, up to a technical difficulty. In the definition of the encoding, we have exploited the commutation of *box* with respect to pairs: that is, a polymorphic pair can be transformed into a pair whose components are polymorphic. The term that performs this transformation can be defined in System F, and, a fortiori, in FORK; up to type erasure, it is the identity. If the calculi were extended with sums, then, analogously, we would need a coercion that transforms a polymorphic sum into a sum whose summands are polymorphic. Unfortunately, such a coercion cannot be defined, it seems, in System F. Instead, it must be added as an axiom, and one must move to a version of FORK equipped with subtyping, in the style of System F_η [17].

The typed store-passing translation that we have presented is not fully abstract: there are terms that inhabit the encoding of a source type, but do not encode any source term. In particular, there is "snapback:" it is permitted to duplicate or discard the store. Following O'Hearn and Reynolds [24], one could enrich FORK with linear types and refine the translation so as to encode the fact that the store is treated linearly. One might then hope to prove that the refined translation is fully abstract. Møgelberg and Staton [20] prove such a result for a store-passing translation that deals with local state. Their translation is simpler than the one considered here insofar as they fix the type of the store: all locations have the same type, and all locations are considered allocated (they initially hold a default value).

In this paper, only a limited meta-theoretic study of FORK has been carried out: we have established its type soundness with respect to an operational semantics. To go further, we suggest building semantic models of FORK, perhaps by following Birkedal *et al.* [8], and determining whether useful models of System F with general references can in fact be obtained by composition with the store-passing translation presented in this paper.

[2] In the absence of group regions in the source calculus, Charguéraud and Pottier's translation is type-preserving in a strong sense. When the source calculus has group regions, however, their translation is type-preserving only in a weaker sense: it uses map lookup and map update operations whose success is guaranteed by the type system of the source calculus but not by the type system of the target calculus. The success of these operations depends on the fact that the population of a group region can only grow with time. Thus, achieving type preservation in a strong sense might require a possible worlds machinery, as in the present paper.

Acknowledgments

I wish to thank Lars Birkedal, Paul-André Melliès, Bernhard Reus, Jan Schwinghammer, and Hongseok Yang for pleasant and inspiring discussions.

References

[1] Andreas Abel. MiniAgda: Integrating sized and dependent types. In *Workshop on Partiality And Recursion in Interactive Theorem Provers (PAR)*, July 2010.

[2] Amal Jamil Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, 2004.

[3] Thorsten Altenkirch, Nils Anders Danielsson, Andres Löh, and Nicolas Oury. ΠΣ: Dependent types without the sugar. In *Functional and Logic Programming*, volume 6009 of *Lecture Notes in Computer Science*, pages 40–55. Springer, April 2010.

[4] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, September 1993.

[5] Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. A very modal model of a modern, major, general type system. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 109–122, January 2007.

[6] Robert Atkey. Parameterised notions of computation. *Journal of Functional Programming*, 19(3–4):355–376, 2009.

[7] Henk P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*. Elsevier Science, 1984.

[8] Lars Birkedal, Jan Schwinghammer, and Kristian Støvring. A metric model of lambda calculus with guarded recursion. Presented at FICS 2010, July 2010.

[9] Lars Birkedal, Kristian Støvring, and Jacob Thamsborg. The category-theoretic solution of recursive metric-space quations. Technical Report ITU-2009-119, IT University of Copenhagen, 2009.

[10] Lars Birkedal, Kristian Støvring, and Jacob Thamsborg. Realizability semantics of parametric polymorphism, general references, and recursive types. *Mathematical Structures in Computer Science*, 2010. To appear.

[11] Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticæ*, 33:309–338, 1998.

[12] Arthur Charguéraud and François Pottier. Functional translation of a calculus of capabilities. In *ACM International Conference on Functional Programming (ICFP)*, pages 213–224, September 2008.

[13] Robert Harper. A simplified account of polymorphic references. *Information Processing Letters*, 51(4):201–206, 1994.

[14] Aquinas Hobor, Robert Dockins, and Andrew W. Appel. A theory of indirection via approximation. In *ACM Symposium on Principles of Programming Languages (POPL)*, January 2010.

[15] Soren B. Lassen. Bisimulation in untyped lambda calculus: Böhm trees and bisimulation up to context. In *Mathematical Foundations of Programming Semantics*, volume 20 of *Electronic Notes in Theoretical Computer Science*, pages 346–374. Elsevier Science, April 1999.

[16] Paul Blain Levy. Possible world semantics for general storage in call-by-value. In *Computer Science Logic*, volume 2471 of *Lecture Notes in Computer Science*. Springer, 2002.

[17] John C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 76(2–3):211–249, 1988.

[18] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1), 1991.

[19] Eugenio Moggi and Amr Sabry. An abstract monadic semantics for value recursion. *Informatique théorique et applications*, 38(4):377–400, 2004.

[20] Rasmus Ejlers Møgelberg and Sam Staton. Full abstraction in a metalanguage for state. In *Workshop on Syntax and Semantics of Low Level Languages*, July 2010.

[21] Hiroshi Nakano. A modality for recursion. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 255–266, June 2000.

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

[23] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Hoare type theory, polymorphism and separation. *Journal of Functional Programming*, 18(5–6), 2008.

[24] Peter W. O'Hearn and John C. Reynolds. From Algol to polymorphic linear lambda-calculus. *Journal of the ACM*, 47(1):167–223, 2000.

[25] Simon Peyton Jones and Philip Wadler. Imperative functional programming. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 71–84, January 1993.

[26] François Pottier. Hiding local state in direct style: a higher-order anti-frame rule. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 331–340, June 2008.

[27] François Pottier. A formalization of Nakano's type system. `http://gallium.inria.fr/~fpottier/fork/`, October 2009.

[28] François Pottier. The electronic FORK, July 2010. `http://gallium.inria.fr/~fpottier/fork/`.

[29] François Pottier. A typed store-passing translation for general references (extended version). November 2010.

[30] Jan Schwinghammer, Lars Birkedal, Bernhard Reus, and Hongseok Yang. Nested Hoare triples and frame rules for higher-order store. In *Computer Science Logic*, volume 5771 of *Lecture Notes in Computer Science*, pages 440–454. Springer, September 2009.

[31] Jan Schwinghammer, Hongseok Yang, Lars Birkedal, François Pottier, and Bernhard Reus. A semantic foundation for hidden state. In *International Conference on Foundations of Software Science and Computation Structures (FOSSACS)*, volume 6014 of *Lecture Notes in Computer Science*, pages 2–17. Springer, March 2010.

[32] Christopher Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1–2):11–49, April 2000.

[33] Masako Takahashi. Parallel reductions in λ-calculus. *Information and Computation*, 118(1):120–127, April 1995.

[34] Robert D. Tennent and Dan Ghica. Abstract models of storage. *Higher-Order and Symbolic Computation*, 13:119–129, 2000.

[35] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994.