

# A typed store-passing translation for general references

François Pottier  
INRIA

## Abstract

We present a store-passing translation of System  $F$  with general references into an extension of System  $F_\omega$  with certain well-behaved recursive kinds. This provides a purely syntactic account of a possible worlds model.

## 1 Introduction

**Motivation** Building a semantic model of a programming language amounts to translating it into some mathematical meta-language. 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 [23] 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 seems to 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.

In so doing, we move the frontier between syntax and semantics. We suggest that a model of the imperative source language can 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  $F_\omega$ . 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, which means that the type system allows references to become aliased. As a consequence, an update to a reference must preserve its type, and explicit de-allocation is forbidden.

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 [11, 1, 6]. 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  $world \cong world \rightarrow \dots$  [6] or work with approximate solutions of it [1, 9]. An alternative is to define a world as a map of memory addresses to syntactic types [11]. 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  $world \cong world \rightarrow \dots$ , 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  $F_\omega$  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  $\lambda$ -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 [13, 14], 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  $world =$

$world \rightarrow \dots$  is in fact invalid, because it does not involve the modality. Fortunately, the modified equation  $world = \bullet world \rightarrow \dots$  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.

The ultrametric-space techniques of Birkedal *et al.* [6, 5] and of Schwinghammer *et al.* [20] 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 [20, Lemma 13] becomes a recursive definition that happens to be permitted in Nakano’s system. The fact that world composition is associative [20, Lemma 14] becomes an assertion about the equality of two Böhm trees and can be automatically checked (§4.4).

**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.

**Paper outline** We first recall Nakano’s system (§2), and build upon it in the definition of FORK (§3). Then, we encode general references into FORK (§4). A prototype implementation of FORK, as well as the source code for a large part of the encoding, are available online [18].

## 2 Nakano’s system

We now recall the definition and properties of Nakano’s system [13, 14]. Our version of the system is close to Nakano’s  $S\text{-}\lambda\bullet\mu^+$ . 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:

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

A kind  $\kappa$  is an infinite tree. In the prototype implementation [18], kinds are finitely represented via a set of mutually recursive defining equations.

A kind  $\kappa$  is *well-formed* iff every infinite path through  $\kappa$  crosses a  $\bullet$  constructor infinitely often. A kind is *finite* iff every infinite path through  $\kappa$  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 2.6, where it serves to rule out types

$$\frac{\kappa'_1 \leq \kappa_1 \quad \kappa_2 \leq \kappa'_2}{\kappa_1 \rightarrow \kappa_2 \leq \kappa'_1 \rightarrow \kappa'_2} \quad \frac{\kappa \leq \kappa'}{\bullet \kappa \leq \bullet \kappa'} \quad \kappa \leq \bullet \kappa$$

$$\bullet(\kappa_1 \rightarrow \kappa_2) \stackrel{\leq}{\geq} \bullet \kappa_1 \rightarrow \bullet \kappa_2$$

**Figure 1. Properties of the subkind relation**

$$\frac{K \vdash \alpha : K(\alpha)}{K \vdash \tau_1 : \kappa_1 \rightarrow \kappa_2} \quad \frac{K; \alpha : \kappa_1 \vdash \tau : \kappa_2}{K \vdash \lambda \alpha. \tau : \kappa_1 \rightarrow \kappa_2}$$

$$\frac{K \vdash \tau_1 : \kappa_1 \rightarrow \kappa_2 \quad K \vdash \tau_2 : \kappa_1}{K \vdash \tau_1 \tau_2 : \kappa_2} \quad \frac{K \vdash \tau : \kappa_1 \quad \kappa_1 \leq \kappa_2}{K \vdash \tau : \kappa_2}$$

**Figure 2. Kind assignment**

that produce an infinite stream of  $\lambda$ 's, and therefore do not have a head normal form.)

**Subkinding** Kinds come with a *subkind* relation  $\leq$ . We omit its definition, which is somewhat technical (the reader is referred to [17]) and is irrelevant as long as the following properties hold. The subkind relation is reflexive, transitive, and satisfies the laws in Figure 1, where  $\stackrel{\leq}{\geq}$  means that the relation holds in both directions. The subkind relation satisfies the following inversion lemma:

**Lemma 2.1** *If  $\kappa_1 \rightarrow \kappa_2 \leq \kappa'_1 \rightarrow \kappa'_2$  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.*  $\diamond$

(We write  $\bullet^n \kappa$  for  $n$  applications of  $\bullet$  to  $\kappa$ .) When kinds are finitely represented as a set of mutually recursive defining equations, it is decidable whether two kinds  $\kappa_1$  and  $\kappa_2$  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 [17]. The prototype implementation [18] takes advantage of this fact to perform bottom-up kind synthesis.

**Types** Types are pure  $\lambda$ -terms:

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

We write  $\tau_1 \longrightarrow \tau_2$  when  $\tau_1$   $\beta$ -reduces to  $\tau_2$ . 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  $\tau$  has kind  $\kappa$ . The rules that define it (Figure 2) are standard.

**Recursion** The judgement  $\vdash Y : (\bullet \kappa \rightarrow \kappa) \rightarrow \kappa$ , where  $Y$  is Curry's fixed point combinator, is derivable. 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 [18] 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 *left subtyping*, *degradation*, *substitution*, and *subject reduction* properties.

**Lemma 2.2**  *$K_1 \leq K_2$  and  $K_2 \vdash \tau : \kappa$  imply  $K_1 \vdash \tau : \kappa$ .*  $\diamond$

**Lemma 2.3**  *$K \vdash \tau : \kappa$  implies  $\bullet K \vdash \tau : \bullet \kappa$ .*  $\diamond$

**Lemma 2.4**  *$K_1; \alpha : \kappa_1; K_2 \vdash \tau_2 : \kappa_2$  and  $K_1 \vdash \tau_1 : \kappa_1$  imply  $K_1; K_2 \vdash [\alpha \mapsto \tau_1] \tau_2 : \kappa_2$ .*  $\diamond$

**Lemma 2.5**  *$K \vdash \tau_1 : \kappa$  and  $\tau_1 \longrightarrow \tau_2$  imply  $K \vdash \tau_2 : \kappa$ .*  $\diamond$

A type of the form  $\lambda \alpha_1 \dots \alpha_m. \alpha \tau_1 \dots \tau_n$  is a *head normal form*. Types are solvable [13, 14, 17]:

**Lemma 2.6** *If  $K \vdash \tau : \kappa$  then  $\tau$  has a head normal form.*  $\diamond$

It is worth noting that this holds under any environment  $K$ , that is, in the presence of type variables of arbitrary kind. Together, Lemmas 2.5 and 2.6 imply that every type admits a maximal Böhm tree, that is, one that does not contain any occurrence of  $\perp$  (the undefined Böhm tree). These two lemmas have been checked by the author using the Coq proof assistant [17].

**Type equality** We take *type equality*, a relation between types, to be *Böhm tree equivalence up to  $\eta$*  [4, §10.2.25] [10]. Several alternative characterizations of this relation are known. It is the greatest consistent  $\lambda$ -theory. It coincides with the equational theory of Scott's  $D_\infty$  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  $\tau_1$  and  $\tau_2$  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  $\kappa$ . This has the following beneficial consequence:

**Lemma 2.7** *The relation  $\equiv$ , restricted to well-kinded types, is semi-decidable.*  $\diamond$

**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  $\tau$ .

$$\begin{aligned} \langle \lambda \alpha. \tau_1 \equiv \lambda \alpha. \tau_2 \rangle & \text{ is } \langle \tau_1 \equiv \tau_2 \rangle \\ \langle \lambda \alpha. \tau_1 \equiv \tau_2 \rangle & \text{ is } \langle \tau_1 \equiv \tau_2 \ \alpha \rangle && \text{if } \alpha \# \tau_2 \\ \langle \tau_1 \equiv \lambda \alpha. \tau_2 \rangle & \text{ is } \langle \tau_1 \ \alpha \equiv \tau_2 \rangle && \text{if } \alpha \# \tau_1 \\ \langle \tau_1 \ \tau'_1 \equiv \tau_2 \ \tau'_2 \rangle & \text{ is } \langle \tau_1 \equiv \tau_2 \rangle \wedge \tau'_1 \downarrow \equiv \tau'_2 \downarrow \\ \langle \alpha \equiv \alpha \rangle & \text{ is true} \\ \langle \tau_1 \equiv \tau_2 \rangle & \text{ is false} \end{aligned}$$

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  $\tau_1$  and  $\tau_2$  have a finite Böhm tree, then it reports “yes”. If  $\tau_1 \equiv \tau_2$  holds and  $\tau_1$  and  $\tau_2$  have an infinite Böhm tree, then the semi-algorithm diverges.  $\square$

In the prototype implementation [18], 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  $\eta$  and up to context*, as studied by Lassen [10], 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.

**Remark** It is not known to the author whether the relation  $\equiv$ , restricted to well-kinded types, is decidable. It seems that Solomon’s reduction of the DPDA equality problem to

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

**Figure 3. Kinds, types, terms**

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.  $K \vdash \exists_{\kappa} : (\kappa \rightarrow \bullet^n \star) \rightarrow \bullet^n \star$

**Figure 4. Kind assignment: constants**

a type equality problem [22] can be adapted, up to an exponential increase in the size of the problem instance. This suggests that the decision problem remains hard.  $\diamond$

Why should types be well-kinded? If we were to remove this condition and work with unkinded types, the fact that  $\equiv$  is a consistent  $\lambda$ -theory would be sufficient to prove that FORK is type-safe (§3). However, we would lose the property that every type is solvable and the property that type equality is semi-decidable. Also, we would lose the ability (not investigated in this paper) to build a model of FORK where kinds are interpreted in a category of ultrametric spaces, in the style of Birkedal *et al.* [5].

### 3 FORK

In System  $F_{\omega}$ , 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  $F_{\omega}$ . 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  $F_{\omega}$ . These constants are assigned kinds by the axioms in Figure 4.

It may come as a surprise that the function and product type constructors have kind  $\bullet \star \rightarrow \bullet \star \rightarrow \star$ , as opposed to  $\star \rightarrow \star \rightarrow \star$  in  $F_{\omega}$ . This means that these constructors are

$\text{VAR}$ $\frac{}{\Gamma \vdash x : \Gamma(x)}$	$\text{ABS}$ $\frac{\Gamma \vdash \tau_1 : \circ\star \quad \Gamma; x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x.t : \tau_1 \rightarrow \tau_2}$	$\text{APP}$ $\frac{\Gamma \vdash t_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1 t_2 : \tau_2}$	$\text{UNIT}$ $\frac{}{\Gamma \vdash () : ()}$
$(\cdot, \cdot)\text{-INTRO}$ $\frac{\Gamma \vdash t_1 : \tau_1 \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (t_1, t_2) : (\tau_1, \tau_2)}$	$(\cdot, \cdot)\text{-ELIM}$ $\frac{\Gamma \vdash t_1 : (\tau_1, \tau_2) \quad \Gamma; x_1 : \tau_1; x_2 : \tau_2 \vdash t_2 : \tau}{\Gamma \vdash \text{let } (x_1, x_2) = t_1 \text{ in } t_2 : \tau}$	$\forall\text{-INTRO}$ $\frac{\Gamma; \alpha : \kappa \vdash t : \tau \quad \alpha \# \Gamma}{\Gamma \vdash \Lambda \alpha.t : \forall_\kappa (\lambda \alpha.\tau)}$	$\forall\text{-ELIM}$ $\frac{\Gamma \vdash t : \forall_\kappa \tau_1 \quad \Gamma \vdash \tau_2 : \circ\kappa}{\Gamma \vdash t \tau_2 : \tau_1 \tau_2}$
$\exists\text{-INTRO}$ $\frac{\Gamma \vdash t : \tau_1 \tau_2 \quad \Gamma \vdash \exists_\kappa \tau_1 : \circ\star \quad \Gamma \vdash \tau_2 : \circ\kappa}{\Gamma \vdash \text{pack } \tau_2, t \text{ as } \exists_\kappa \tau_1 : \exists_\kappa \tau_1}$	$\exists\text{-ELIM}$ $\frac{\Gamma \vdash t_1 : \exists_\kappa \tau_1 \quad \alpha \# \Gamma, \tau_2 \quad \Gamma; \alpha : \kappa; x : (\tau_1 \alpha) \vdash t_2 : \tau_2}{\Gamma \vdash \text{unpack } \alpha, x = t_1 \text{ in } t_2 : \tau_2}$	$\text{CONVERSION}$ $\frac{\Gamma \vdash t : \tau_1 \quad \Gamma \vdash \tau_2 : \circ\star \quad \tau_1 \equiv \tau_2}{\Gamma \vdash t : \tau_2}$	

Figure 5. Type assignment

*contractive* in both arguments. The FORK axioms are more liberal than their  $F_\omega$  counterparts: for instance, the former allow the recursive type  $\mu \alpha. \alpha \rightarrow \alpha$  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  $\bullet\star \rightarrow \bullet\star \rightarrow \star$ , one cannot expect the types that classify values to always have kind  $\star$ , as in  $F_\omega$ . Instead, in FORK, the types that classify values have kind  $\bullet^n \star$ , for some  $n \in \mathbb{N}$ . We write  $K \vdash \tau : \circ\kappa$  when  $K \vdash \tau : \bullet^n \kappa$  holds for some  $n \in \mathbb{N}$ . This notion is robust in the following sense:

**Lemma 3.1**  $K \vdash \tau_1 \rightarrow \tau_2 : \circ\star$  holds iff  $K \vdash \tau_1 : \circ\star$  and  $K \vdash \tau_2 : \circ\star$  hold. (Similarly for products.)  $\diamond$

**Proof.** Assume  $K \vdash \tau_1 \rightarrow \tau_2 : \bullet^n \star$ . An analysis of the derivation of this judgement reveals that we must have:

$$\begin{array}{l} K \vdash \tau_1 : \kappa_1 \\ K \vdash \tau_2 : \kappa_2 \\ \bullet\star \rightarrow \bullet\star \rightarrow \star \leq \kappa_1 \rightarrow \kappa_2 \rightarrow \bullet^n \star \end{array}$$

By Lemma 2.1, there exist  $i, j \in \mathbb{N}$  such that  $\kappa_1 \leq \bullet^{i+1} \star$  and  $\kappa_2 \leq \bullet^{j+1} \star$ . Thus,  $K \vdash \tau_1 : \circ\star$  and  $K \vdash \tau_2 : \circ\star$  hold.

Conversely, assume  $K \vdash \tau_1 : \bullet^i \star$  and  $K \vdash \tau_2 : \bullet^j \star$ . Let  $n$  be the maximum of  $i$  and  $j$ . Then, we have:

$$\begin{array}{l} \bullet\star \rightarrow \bullet\star \rightarrow \star \leq \bullet^{n+1} \star \rightarrow \bullet^{n+1} \star \rightarrow \bullet^n \star \\ \leq \bullet^i \star \rightarrow \bullet^j \star \rightarrow \bullet^n \star \end{array}$$

This allows deriving  $K \vdash \tau_1 \rightarrow \tau_2 : \bullet^n \star$ , which implies  $K \vdash \tau_1 \rightarrow \tau_2 : \circ\star$ .  $\square$

**Lemma 3.2**  $\Gamma \vdash \forall_\kappa \tau : \bullet^n \star$  holds iff  $\Gamma \vdash \tau : \kappa \rightarrow \bullet^n \star$  holds. (Similarly for existential quantifiers.)  $\diamond$

**Proof.** The right-to-left implication is an immediate consequence of the fact that  $\forall_\kappa$  has kind  $(\kappa \rightarrow \bullet^n \star) \rightarrow \bullet^n \star$ . Let us now consider the left-to-right implication. Assume  $\Gamma \vdash \forall_\kappa \tau : \bullet^n \star$ . An analysis of the derivation of this judgement shows that we must have:

$$\begin{array}{l} \Gamma \vdash \tau : \kappa_1 \\ (\kappa \rightarrow \bullet^m \star) \rightarrow \bullet^m \star \leq \kappa_1 \rightarrow \bullet^n \star \end{array}$$

for some  $m \in \mathbb{N}$ . By Lemma 2.1, there exists  $k \in \mathbb{N}$  such that:

$$\begin{array}{l} \kappa_1 \leq \bullet^k \kappa \rightarrow \bullet^{k+m} \star \\ \bullet^{k+m} \star \leq \bullet^n \star \end{array}$$

This implies  $\kappa_1 \leq \kappa \rightarrow \bullet^n \star$ , whence  $\Gamma \vdash \tau : \kappa \rightarrow \bullet^n \star$ .  $\square$

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  $(\kappa \rightarrow \bullet^n \star) \rightarrow \bullet^n \star$ , as opposed to just  $(\kappa \rightarrow \star) \rightarrow \star$  in  $F_\omega$ .

The properties stated in the previous section (§2) remain valid in the presence of type constants of arbitrary kind. The proof of Lemma 2.5 is trivially extended. The model that underlies the proof of Lemma 2.6 validates the fact that the variables inhabit every kind, so the proof is unchanged, provided constants are viewed as variables. Thus, every type admits a head normal form, where the head is either a variable or a constant.

**Type assignment** The syntax of terms  $t$  and type environments  $\Gamma$  is standard (Figure 3). A type environment  $\Gamma$  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  $F_\omega$  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  $\circ\star$ . This is reflected in the following definition and lemma:

**Definition 3.3** *The empty type environment is well-formed. The type environment  $\Gamma; \alpha : \kappa$  is well-formed if  $\Gamma$  is well-formed and  $\alpha \# \Gamma$ . The type environment  $\Gamma; x : \tau$  is well-formed if  $\Gamma$  is well-formed and  $\Gamma \vdash \tau : \circ\star$ .*  $\diamond$

**Lemma 3.4** *If  $\Gamma$  is well-formed, then  $\Gamma \vdash t : \tau$  implies  $\Gamma \vdash \tau : \circ\star$ .*  $\diamond$

**Proof.** By induction over the derivation of  $\Gamma \vdash t : \tau$ .

◦ *Case VAR.* One must check that, if  $\Gamma$  is well-formed, and if  $\Gamma(x)$  is defined, then  $\Gamma \vdash \Gamma(x) : \circ\star$  holds. This is easily checked by induction over  $\Gamma$ .

◦ *Case ABS.* The first premise is  $\Gamma \vdash \tau_1 : \circ\star$ . Thus,  $\Gamma; x : \tau_1$  is well-formed. This allows applying the induction hypothesis to the second premise. This yields  $\Gamma; x : \tau_1 \vdash \tau_2 : \circ\star$ , that is,  $\Gamma \vdash \tau_2 : \circ\star$ . By Lemma 3.1,  $\Gamma \vdash \tau_1 \rightarrow \tau_2 : \circ\star$  follows.

◦ *Case APP.* The induction hypothesis, applied to the first premise, yields  $\Gamma \vdash \tau_1 \rightarrow \tau_2 : \circ\star$ . By Lemma 3.1,  $\Gamma \vdash \tau_2 : \circ\star$  follows.

◦ *Case UNIT.* Immediate.

◦ *Cases  $(,)$ -INTRO,  $(,)$ -ELIM.* Analogous to the cases of ABS and APP above.

◦ *Case  $\forall$ -INTRO.* Because  $\alpha$  is fresh for  $\Gamma$ , the type environment  $\Gamma; \alpha : \kappa$  is well-formed. Thus, by the induction hypothesis,  $\Gamma; \alpha : \kappa \vdash \tau : \circ\star$  holds: that is, for some  $n \in \mathbb{N}$ ,  $\Gamma; \alpha : \kappa \vdash \tau : \bullet^n \star$  holds. This implies  $\Gamma \vdash \lambda \alpha. \tau : \kappa \rightarrow \bullet^n \star$ . By Lemma 3.2, this implies  $\Gamma \vdash \forall_{\kappa} (\lambda \alpha. \tau) : \bullet^n \star$ . Thus,  $\Gamma \vdash \forall_{\kappa} (\lambda \alpha. \tau) : \circ\star$  holds.

◦ *Case  $\forall$ -ELIM.* By the induction hypothesis, we have  $\Gamma \vdash \forall_{\kappa} \tau_1 : \circ\star$ , that is,  $\Gamma \vdash \forall_{\kappa} \tau_1 : \bullet^n \star$ , for some  $n \in \mathbb{N}$ . By Lemma 3.2, this implies  $\Gamma \vdash \tau_1 : \kappa \rightarrow \bullet^n \star$ . Now, we have  $\Gamma \vdash \tau_2 : \circ\kappa$ , that is,  $\Gamma \vdash \tau_2 : \bullet^j \kappa$ , for some  $j \in \mathbb{N}$ . There follows  $\Gamma \vdash \tau_1 \tau_2 : \bullet^{j+n} \star$ , which implies  $\Gamma \vdash \tau_1 \tau_2 : \circ\star$ .

◦ *Case  $\exists$ -INTRO.* Immediate.

◦ *Case  $\exists$ -ELIM.* As in the case of  $\forall$ -ELIM, one proves  $\Gamma \vdash \tau_1 : \kappa \rightarrow \bullet^n \star$ , for some  $n \in \mathbb{N}$ . There follows that the type environment  $\Gamma; \alpha : \kappa; x : (\tau_1 \alpha)$  is well-formed. This allows applying the induction hypothesis to the second premise, yielding  $\Gamma; \alpha : \kappa; x : (\tau_1 \alpha) \vdash \tau_2 : \circ\star$ , that is,  $\Gamma; \alpha : \kappa \vdash \tau_2 : \circ\star$ . Thanks to the premise  $\alpha \# \tau_2$ , this implies  $\Gamma \vdash \tau_2 : \circ\star$ .

◦ *Case CONVERSION.* Immediate.  $\square$

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  $\kappa$  can be instantiated with a type of kind  $\circ\kappa$ . The rule  $\exists$ -INTRO is similarly relaxed. This makes sense thanks to the following facts, which are later exploited in the proof of subject reduction:

**Lemma 3.5**  $\Gamma_1; \alpha : \kappa_1; \Gamma_2 \vdash \tau : \circ\kappa_2$  implies  $\Gamma_1; \alpha : \bullet \kappa_1; \Gamma_2 \vdash \tau : \circ\kappa_2$ .  $\diamond$

**Proof.** Assume  $\Gamma_1; \alpha : \kappa_1; \Gamma_2 \vdash \tau : \bullet^n \kappa_2$ , for some  $n \in \mathbb{N}$ . By Lemmas 2.2 and 2.3, this implies  $\Gamma_1; \alpha : \bullet \kappa_1; \Gamma_2 \vdash \tau : \bullet^{n+1} \kappa_2$ .  $\square$

**Lemma 3.6** *If  $\Gamma_1; \alpha : \kappa; \Gamma_2$  is well-formed, then  $\Gamma_1; \alpha : \bullet \kappa; \Gamma_2$  is well-formed. Furthermore,  $\Gamma_1; \alpha : \kappa; \Gamma_2 \vdash t : \tau$  implies  $\Gamma_1; \alpha : \bullet \kappa; \Gamma_2 \vdash t : \tau$ .*  $\diamond$

**Proof.** The first part of the statement is proved by induction over the structure of  $\Gamma_2$ , using Lemma 3.5 in the case where  $\Gamma_2$  grows with a term variable binding. The second part of the statement is proved by induction over the derivation of  $\Gamma_1; \alpha : \kappa; \Gamma_2 \vdash t : \tau$ , using Lemma 3.5 to deal with the well-kindedness premises.  $\square$

**Lemma 3.7** *If  $\Gamma_1 \vdash \tau : \kappa$  holds and  $\Gamma_1; \Gamma_2$  is well-formed, then  $\Gamma_1; \Gamma_2 \vdash \tau : \kappa$  holds.*  $\diamond$

**Proof.** By induction over  $\Gamma_2$ .  $\square$

**Lemma 3.8** *Let  $\Gamma_1 \vdash \tau_1 : \kappa$ . 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  $\Gamma_1; [\alpha \mapsto \tau_1] \Gamma_2 \vdash [\alpha \mapsto \tau_1] t : [\alpha \mapsto \tau_1] \tau_2$ .*  $\diamond$

**Proof.** The first part of the statement is proved by induction over the structure of  $\Gamma_2$ , using Lemma 3.7 and Lemma 2.4 in the case where  $\Gamma_2$  grows with a term variable binding. The second part of the statement is proved by induction over the derivation of  $\Gamma_1; \alpha : \kappa; \Gamma_2 \vdash t : \tau_2$ , using Lemma 2.4 to deal with the well-kindedness premises.  $\square$

**Lemma 3.9** *Let  $\Gamma_1 \vdash \tau_1 : \circ\kappa$ . 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  $\Gamma_1; [\alpha \mapsto \tau_1] \Gamma_2 \vdash [\alpha \mapsto \tau_1] t : [\alpha \mapsto \tau_1] \tau_2$ .*  $\diamond$

**Proof.** By Lemmas 3.6 and 3.8.  $\square$

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  $\lambda$ -bound variable carries an explicit kind and every  $\lambda$ -bound variable carries an explicit type, the knowledge of  $\Gamma$  and  $t$  is sufficient to reconstruct a type  $\tau$  (if one exists) such that  $\Gamma \vdash t : \tau$  holds. The prototype implementation of FORK follows this scheme.

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

**Figure 6. Reduction semantics**

**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 3.10**  $\Gamma_1; x : \tau_2; \Gamma_2 \vdash t_1 : \tau_1$  and  $\Gamma_1 \vdash t_2 : \tau_2$  imply  $\Gamma_1; \Gamma_2 \vdash [x \mapsto t_2]t_1 : \tau_1$ .  $\diamond$

**Proof.** By induction over the derivation of  $\Gamma_1; x : \tau_2; \Gamma_2 \vdash t_1 : \tau_1$ .  $\square$

**Lemma 3.11 (Subject reduction)**  $\Gamma \vdash t_1 : \tau$  and  $t_1 \longrightarrow t_2$  imply  $\Gamma \vdash t_2 : \tau$ .  $\diamond$

**Proof.** By induction over the derivation of  $t_1 \longrightarrow t_2$ . We assume, without loss of generality, that the derivation of  $\Gamma \vdash t_1 : \tau$  does not end with an instance of CONVERSION. Only two representative cases are treated.

◦ *Case*  $(\lambda x.t_1) t_2 \longrightarrow [x \mapsto t_2]t_1$ . An analysis of the typing derivation for  $(\lambda x.t_1) t_2$  reveals that we must have:

$$\begin{array}{ll}
\Gamma; x : \tau_2 \vdash t_1 : \tau_1 & \text{(1)} \\
\Gamma \vdash t_2 : \tau'_2 & \text{(2)} \\
\tau_2 \rightarrow \tau_1 \equiv \tau'_2 \rightarrow \tau'_1 & \text{(3)}
\end{array}$$

if the conclusion of this derivation is  $\Gamma \vdash (\lambda x.t_1) t_2 : \tau'_1$ . Because type equality  $\equiv$  is a consistent  $\lambda$ -theory, equality of two head normal forms implies equality of their components. That is, (3) implies:  $\tau_2 \equiv \tau'_2$  (4) and  $\tau_1 \equiv \tau'_1$  (5). By CONVERSION, (1) and (5) imply  $\Gamma; x : \tau_2 \vdash t_1 : \tau'_1$  (6), while (2) and (4) imply  $\Gamma \vdash t_2 : \tau_2$  (7). By Lemma 3.10, (6) and (7) imply  $\Gamma \vdash [x \mapsto t_2]t_1 : \tau'_1$ , which is the goal.

◦ *Case*  $(\Lambda \alpha.t) \tau \longrightarrow [\alpha \mapsto \tau]t$ . An analysis of the typing derivation for  $(\Lambda \alpha.t) \tau$  reveals that we must have:

$$\begin{array}{ll}
\Gamma; \alpha : \kappa \vdash t : \tau_1 & \text{(1)} \\
\Gamma \vdash \tau : \bigcirc \kappa & \text{(2)} \\
\forall \kappa (\lambda \alpha.\tau_1) \equiv \forall \kappa \tau_2 & \text{(3)}
\end{array}$$

if the conclusion of this derivation is  $\Gamma \vdash (\Lambda \alpha.t) \tau : \tau_2$ . By Lemma 3.9, (1) and (2) imply:

$$\Gamma \vdash [\alpha \mapsto \tau]t : [\alpha \mapsto \tau]\tau_1 \quad \text{(4)}$$

Because type equality  $\equiv$  is a consistent  $\lambda$ -theory, (3) implies:

$$\lambda \alpha.\tau_1 \equiv \tau_2 \quad \text{(5)}$$

Because type equality  $\equiv$  is a  $\lambda$ -theory, it is a congruence, and contains  $\beta$ -equivalence. That is, (5) implies:

$$[\alpha \mapsto \tau]\tau_1 \equiv \tau_2 \tau \quad \text{(6)}$$

By CONVERSION, (4) and (6) imply  $\Gamma \vdash [\alpha \mapsto \tau]t : \tau_2$ , which is the goal.  $\square$

**Definition 3.12** A term  $t$  is well-typed iff  $\Gamma \vdash t : \tau$  holds, where  $\Gamma$  binds only type variables (no term variables).  $\diamond$

The following lemma allows determining the shape of a value whose type is known. Note that exactly one of the cases listed in the lemma must be applicable.

**Lemma 3.13 (Classification)** Let  $\Gamma \vdash v : \tau$ . Then,

- if  $\tau \equiv \tau_1 \rightarrow \tau_2$ , then  $v$  is of the form  $\lambda x.t$ ;
- if  $\tau \equiv ()$ , then  $v$  is  $()$ ;
- if  $\tau \equiv (\tau_1, \tau_2)$ , then  $v$  is of the form  $(v_1, v_2)$ ;
- if  $\tau \equiv \forall \kappa \tau_1$ , then  $v$  is of the form  $\Lambda \alpha.v'$ ;
- if  $\tau \equiv \exists \kappa \tau_1$ , then  $v$  is of the form  $\text{pack } \tau_2, v' \text{ as } \tau_1$ .  $\diamond$

**Proof.** By induction over the derivation of  $\Gamma \vdash v : \tau$ . Recall that if two types in head normal form are equal, then their head constants are equal as well. Thus, at most one of the cases listed above is applicable. The result is then immediate.  $\square$

**Lemma 3.14 (Progress)** A well-typed term either reduces or is a value.  $\diamond$

**Proof.** By induction over the derivation of  $\Gamma \vdash t : \tau$ , where  $\Gamma$  binds only type variables. Only some representative cases are treated.

◦ *Case* VAR. Impossible.

◦ *Case* ABS. The term of interest,  $\lambda x.t$ , is a value.

◦ *Case* APP. The term of interest is  $t_1 t_2$ . The premises indicate that each of the terms  $t_1$  and  $t_2$  is well-typed, and, more precisely, that  $t_1$  admits a function type. By the induction hypothesis, each of  $t_1$  and  $t_2$  either reduces or is a value. If either reduces, then  $t_1 t_2$  reduces as well. If both are values, then, by Lemma 3.13,  $t_1$  must be a  $\lambda$ -abstraction, so  $t_1 t_2$  reduces.

◦ *Case*  $\forall$ -INTRO. The term of interest is  $\Lambda \alpha.t$ . The premise indicates that  $t$  is well-typed. By the induction hypothesis,  $t$  either reduces or is a value. In the former case,  $\Lambda \alpha.t$  reduces as well. In the latter case,  $\Lambda \alpha.t$  is a value.

◦ *Case*  $\forall$ -ELIM. The term of interest is  $t \tau$ . The premise indicates that  $t$  is well-typed, and, more precisely, that  $t$  admits a universal type. By the induction hypothesis,  $t$  either reduces or is a value. In the former case,  $t \tau$  reduces as well. In the latter case, by Lemma 3.13,  $t$  must be a  $\Lambda$ -abstraction, so  $t \tau$  reduces.  $\square$

**Theorem 3.15 (Type soundness)** *A well-typed term either diverges or reduces (in zero or more steps) to a value.*  $\diamond$

**Proof.** By Lemmas 3.14 and 3.11, a well-typed term either reduces in one step to a well-typed term, or is a value. By iteration, there follows that a well-typed term either admits an infinite reduction sequence or reduces (in zero or more steps) to a value.  $\square$

The fact that (thanks to the kind discipline) types have head normal forms is not exploited in the type soundness proof.

## 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. Its terms are the standard terms of System  $F$ , extended with the monadic constants *return* and *bind* and with the constants *new*, *read*, and *write* for allocating, reading and writing references. Its operational semantics (omitted) is standard. Its types are:

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

where  $M$  is the monad. Its typing judgement  $E \vdash e : T$  is standard; it is omitted as well. (In fact, it is contained in Figures 12 and 13, which describe the encoding.) This monadic treatment of references is due to Peyton Jones and Wadler [16, §5.3].

We now progress, in several steps, towards an encoding of this calculus into FORK. The well-kindedness and well-typedness of the types and terms involved in this encoding have been machine-checked using the prototype implementation of FORK [18]. Only the final meta-theorem (Theorem 4.3) cannot be checked in this way and has been checked on paper [19].

### 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 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*.

```

kind fragment =
   $\star \rightarrow \star$ 
type fnil : fragment =
   $\lambda \text{tail}. \text{tail}$ 
type @ : fragment  $\rightarrow$  fragment  $\rightarrow$  fragment =
   $\lambda f_1 f_2 \text{tail}. f_1 (f_2 \text{tail})$ 
type snoc : fragment  $\rightarrow$   $\star \rightarrow$  fragment =
   $\lambda f. \lambda \text{data}. \lambda \text{tail}. f (\text{data}, \text{tail})$ 

```

Figure 7. Fragments

```

type array :  $\bullet$  fragment  $\rightarrow$   $\star$ 
type index :  $\bullet$  fragment  $\rightarrow$   $\bullet \star \rightarrow \star$ 
term array_empty : array fnil
term array_extend :
   $\forall f \text{data}. \text{array } f \rightarrow \text{data} \rightarrow \text{array } (f \text{ 'snoc' data})$ 
term array_read :
   $\forall f \text{data}. \text{array } f \rightarrow \text{index } f \text{ data} \rightarrow \text{data}$ 
term array_write :
   $\forall f \text{data}. \text{array } f \rightarrow \text{index } f \text{ data} \rightarrow \text{data} \rightarrow \text{array } f$ 
term array_end_index :
   $\forall f. \text{array } f \rightarrow \forall \text{data}. \text{index } (f \text{ 'snoc' data}) \text{ data}$ 
term index_monotonic :
   $\forall f_1 f_2 \text{data}. \text{index } f_1 \text{ data} \rightarrow \text{index } (f_1 \text{ '@' } f_2) \text{ data}$ 

```

Figure 8. Arrays

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 7). The kind *fragment* is  $\star \rightarrow \star$ . 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*  $f \text{ data}$  or more pleasantly  $f \text{ 'snoc' data}$ , is  $\lambda \text{tail}. f (\text{data}, \text{tail})$ . The type  $(\text{data}, \text{tail})$  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 Blain Levy puts it, it “is unnatural [...] to specify what happens when we read a non-existent cell, something that can never occur in reality” [11].) 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 8 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 constructors  $array$  and  $index$  are contractive.

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 [18]. It does not use any recursion: it is in fact expressed within  $F_\omega$ . 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 8 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

```

kind world =
  • world → fragment
type nil : world =
  λx. fnil
type o : world → world → world =
  λw1 w2 x. w1 (w2 'o' x) '@' w2 x

```

Figure 9. Worlds

```

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 → ∃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 10. Semantic types

the store also grows *in depth*.

In order to reflect this, we introduce *worlds* (Figure 9). 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,  $w_1 \text{ 'o' } w_2$ , can be described as 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  $\bullet$  *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 10).

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  $a\ (x \text{ 'o' } y)$  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  $\text{box } a\ x$ , where the semantic type operator *box* builds in a universal quantification over future worlds. This operator is known as the *necessity modality* [2]. A semantic type of the form  $\text{box } a$  is known as *necessary* [2] or *hereditary* [9].

**Remark** There are semantic models (see e.g. [21]) 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.* [2] and Hobor *et al.* [9] and explicitly use the *box* modality to keep track of which types are hereditary.  $\diamond$

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,  $\forall y. [z \mapsto x \text{ 'o' } y] \tau$ ). In this encoding, the associativity of world composition expresses the transitivity of the world ordering.

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  $\text{box } a\ x \rightarrow \text{box } b\ x$ .

**Remark** The reader might have expected instead the type  $\text{box } (\lambda x. a\ x \rightarrow b\ x)$ , which guarantees that the function itself is hereditary, and is more general. This choice would make the construction of a function more difficult, and its use easier. The construction of certain functions, such as *write* (Figure 12), requires that the argument be hereditary. If the argument has type  $\text{box } a\ x$ , this is immediate, whereas if the argument has type  $a\ x$ , this is not obvious. This explains our current choice. One could set things up so that the encoding  $a$  of every source-level type is a hereditary semantic type and argue that the types  $\text{box } a\ x$  and  $a\ x$  are then equivalent. This, however, would require recording the fact that  $a$  is in the image of the encoding. Our current solution is perhaps less elegant, but more lightweight.  $\diamond$

The encoding of the monad is standard [11]. A computation requires a store in world  $x$ , and produces a pair of a (hereditary) result and a new store in some future world  $x \text{ 'o' } y$ .

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  $\forall x. \text{array } (w\ x)$ . 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 of type  $\forall a\ x. \text{ref } a\ x \rightarrow \text{box } (\text{ref } a)\ x$ .

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  $\llbracket T \rrbracket$ . (The inductive definition of this translation cannot be expressed within FORK.)

**Definition 4.1** *The encoding  $\llbracket T \rrbracket$  of a System  $F$  type  $T$  is defined as follows:*

$$\begin{aligned} \llbracket \alpha \rrbracket &= \alpha \\ \llbracket () \rrbracket &= \text{unit} \\ \llbracket T_1 \rightarrow T_2 \rrbracket &= \text{arrow } \llbracket T_1 \rrbracket \llbracket T_2 \rrbracket \\ \llbracket (T_1, T_2) \rrbracket &= \text{pair } \llbracket T_1 \rrbracket \llbracket T_2 \rrbracket \\ \llbracket \forall \alpha. T \rrbracket &= \text{univ } (\lambda \alpha. \llbracket T \rrbracket) \\ \llbracket M\ T \rrbracket &= \text{monad } \llbracket T \rrbracket \\ \llbracket \text{ref } T \rrbracket &= \text{ref } \llbracket T \rrbracket \end{aligned}$$

where the combinators that appear in the right-hand sides of these equations are defined in Figure 10.  $\diamond$

**type**  $cell : stype \rightarrow \bullet world \rightarrow world =$   
 $\lambda a x y tail. (a (x 'o' cell a x 'o' y), tail)$   
**term**  $store\_extend :$   
 $\forall x a. store x \rightarrow box a x \rightarrow store (x 'o' cell a x)$   
**term**  $store\_end\_index :$   
 $\forall x a. store x \rightarrow ref a (x 'o' cell a x)$

**Figure 11. Memory allocation**

## 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.

The definition of *cell* appears in Figure 11. 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  $a(\dots)$  and *tail*. (This is consistent with our definition of fragment extension, *snoc*, in Figure 7.) 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 11. The definitions of these terms [18] 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 The encoding

Once this infrastructure is in place, the definition of the encoding is fairly simple. The most important (and by now straightforward) part is to encode the constants *return*, *bind*, *new*, *read*, and *write*. That is, one must define five terms that admit the types shown in Figure 12 and that adequately

**term**  $return :$   
 $box (univ (\lambda a. a 'arrow' monad a)) nil$   
**term**  $bind :$   
 $box (univ (\lambda a. univ (\lambda b.$   
 $(monad a 'pair' (a 'arrow' monad b)) 'arrow' monad b$   
 $))) nil$   
**term**  $new :$   
 $box (univ (\lambda a. a 'arrow' monad (ref a))) nil$   
**term**  $read :$   
 $box (univ (\lambda a. ref a 'arrow' monad a)) nil$   
**term**  $write :$   
 $box (univ (\lambda a. (ref a 'pair' a) 'arrow' monad unit)) nil$

**Figure 12. Encoding the monadic constants**

implement the store-passing machinery. These definitions are omitted and can be found online [18].

Then, 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 13. 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.

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:*

$$\begin{aligned} w_\emptyset &= nil \\ w_{E;\alpha} &= w_E \\ w_{E;x:T} &= w_E 'o' w_x \end{aligned}$$

The encoding  $\llbracket E \rrbracket$  of a type environment  $E$  is given by:

$$\begin{aligned} \llbracket \emptyset \rrbracket &= \emptyset \\ \llbracket E; \alpha \rrbracket &= \llbracket E \rrbracket; \alpha : stype \\ \llbracket E; x : T \rrbracket &= \llbracket E \rrbracket; w_x : world; x : box \llbracket T \rrbracket w_{E;x:T} \quad \diamond \end{aligned}$$

**Theorem 4.3 (Type preservation)**  $E \vdash e \rightsquigarrow t : T$  implies  $\llbracket E \rrbracket \vdash t : box \llbracket T \rrbracket w_E$ .  $\diamond$

**Proof.** In this proof, we do not consider the cases of the constants *return*, *bind*, *new*, *read*, and *write*, which have been machine-checked [18]. There remains one case for each of the rules in Figure 13.

◦ *Case ENCODE-VAR.* By hypothesis, in the environment  $\llbracket E_1; x : T; E_2 \rrbracket$ ,  $x$  has type:

$$box \llbracket T \rrbracket w_{E_1;x:T}.$$

$$\begin{array}{c}
\text{ENCODE-VAR} \\
E_1; x : T; E_2 \vdash x \rightsquigarrow \Lambda y.x (w_{E_2} 'o' y) : T \\
\\
\text{ENCODE-ABS} \\
\frac{E; x : T_1 \vdash e \rightsquigarrow t : T_2}{E \vdash \lambda x.e \rightsquigarrow \Lambda w_x.\lambda x.t : T_1 \rightarrow T_2} \\
\\
\text{ENCODE-APP} \\
\frac{E \vdash e_1 \rightsquigarrow t_1 : T_1 \rightarrow T_2 \quad E \vdash e_2 \rightsquigarrow t_2 : T_1}{E \vdash e_1 e_2 \rightsquigarrow t_1 \text{ nil } t_2 : T_2} \\
\\
\text{ENCODE-UNIT} \\
E \vdash () \rightsquigarrow \Lambda y.() : () \\
\\
\text{ENCODE-}(,)-\text{INTRO} \\
\frac{E \vdash e_1 \rightsquigarrow t_1 : T_1 \quad E \vdash e_2 \rightsquigarrow t_2 : T_2}{E \vdash (e_1, e_2) \rightsquigarrow \Lambda y.(t_1 y, t_2 y) : (T_1, T_2)} \\
\\
\text{ENCODE-}(,)-\text{ELIM} \\
\frac{E \vdash e \rightsquigarrow t : (T_1, T_2)}{E \vdash \pi_i e \rightsquigarrow \Lambda y.\text{let } (x_1, x_2) = t y \text{ in } x_i : T_i} \\
\\
\text{ENCODE-}\forall\text{-INTRO} \\
\frac{E; \alpha \vdash e \rightsquigarrow t : T}{E \vdash \Lambda \alpha.e \rightsquigarrow \Lambda y.\Lambda(\alpha : \text{stype}).(t y) : \forall \alpha.T} \\
\\
\text{ENCODE-}\forall\text{-ELIM} \\
\frac{E \vdash e \rightsquigarrow t : \forall \alpha.T_1}{E \vdash e T_2 \rightsquigarrow \Lambda y.(t y \llbracket T_2 \rrbracket) : [\alpha \mapsto T_2]T_1}
\end{array}$$

**Figure 13. Encoding the pure fragment of F**

By definition of  $\text{box}$ , this is:

$$\forall y.(\llbracket T \rrbracket (w_{E_1; x:T} 'o' y)).$$

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

$$\llbracket T \rrbracket (w_{E_1; x:T} 'o' w_{E_2} 'o' y),$$

that is,

$$\llbracket T \rrbracket (w_{E_1; x:T; E_2} 'o' y).$$

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

$$\forall y.(\llbracket T \rrbracket (w_{E_1; x:T; E_2} 'o' y)),$$

that is,

$$\text{box } \llbracket T \rrbracket w_{E_1; x:T; E_2}.$$

◦ *Case ENCODE-ABS.* By the induction hypothesis, under the type environment  $\llbracket E; x : T_1 \rrbracket$ , the term  $t$  has type  $\text{box } \llbracket T_2 \rrbracket w_{E; x:T_1}$ . That is, under the type environment:

$$\llbracket E \rrbracket; w_x : \text{world}; x : \text{box } \llbracket T_1 \rrbracket (w_E 'o' w_x),$$

the term  $t$  has type:

$$\text{box } \llbracket T_2 \rrbracket (w_E 'o' w_x).$$

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

$$\forall w_x.(\text{box } \llbracket T_1 \rrbracket (w_E 'o' w_x) \rightarrow \text{box } \llbracket T_2 \rrbracket (w_E 'o' w_x)),$$

By definition of the encoding and by definition of  $\text{box}$ , this type is:

$$\text{box } \llbracket T_1 \rightarrow T_2 \rrbracket w_E.$$

◦ *Case ENCODE-APP.* By the induction hypothesis, under the type environment  $\llbracket E \rrbracket$ , the term  $t_1$  has type  $\text{box } \llbracket T_1 \rightarrow T_2 \rrbracket w_E$  and the term  $t_2$  has type  $\text{box } \llbracket T_1 \rrbracket w_E$ . By definition of  $\text{box}$  and by definition of the encoding, the former of these types is:

$$\forall w_x.(\text{box } \llbracket T_1 \rrbracket (w_E 'o' w_x) \rightarrow \text{box } \llbracket T_2 \rrbracket (w_E 'o' w_x)).$$

As a result, the application  $t_1 \text{ nil } t_2$  has type:

$$\text{box } \llbracket T_1 \rrbracket w_E \rightarrow \text{box } \llbracket T_2 \rrbracket w_E,$$

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

$$\text{box } \llbracket T_2 \rrbracket w_E.$$

◦ *Case ENCODE-}\forall\text{-INTRO.}* By the induction hypothesis, under the type environment  $\llbracket E \rrbracket; \alpha : \text{stype}$ , the term  $t$  has type  $\text{box } \llbracket T \rrbracket w_E$ . Thus, the term:

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

has type

$$\forall y.\forall(\alpha : \text{stype}).(\llbracket T \rrbracket (w_E 'o' y)).$$

By definition of  $\text{univ}$ , this type is:

$$\forall y.(\text{univ } (\lambda \alpha. \llbracket T \rrbracket) (w_E 'o' y)),$$

that is, by definition of  $\text{box}$ :

$$\text{box } (\text{univ } (\lambda \alpha. \llbracket T \rrbracket)) w_E,$$

that is, by definition of the encoding,

$$\text{box } \llbracket \forall \alpha.T \rrbracket w_E.$$

◦ *Case ENCODE-}\forall\text{-ELIM.}* By the induction hypothesis, under the type environment  $\llbracket E \rrbracket$ , the term  $t$  has type:

$$\text{box } \llbracket \forall \alpha.T_1 \rrbracket w_E.$$

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

$$\forall y. \forall (\alpha : \text{stype}). (\llbracket T_1 \rrbracket (w_E \text{ 'o' } y)).$$

Thus, the term  $\Lambda y. (t \ y \ \llbracket T_2 \rrbracket)$  has type:

$$\forall y. (([\alpha \mapsto \llbracket T_2 \rrbracket] \llbracket T_1 \rrbracket) (w_E \text{ 'o' } y)).$$

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

$$\forall y. (\llbracket [\alpha \mapsto T_2] T_1 \rrbracket (w_E \text{ 'o' } y)),$$

that is, by definition of *box*:

$$\text{box } \llbracket [\alpha \mapsto T_2] T_1 \rrbracket w_E.$$

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

We conjecture that the encoding is also adequate (we have not attempted to prove this fact):

**Claim 4.4 (Adequacy)** *If  $\emptyset \vdash e \rightsquigarrow t : ()$  holds, then the System  $F$  term  $e$  and the FORK term  $t$  have the same semantics – that is,  $e$  diverges iff  $t$  diverges.*  $\diamond$

Let us recall that we use a type-erasure interpretation: that is, in the semantics of the source and target calculi, type annotations do not influence reduction, or, more accurately, erasure of the type annotations commutes with reduction.

## 5 Related work

Several store-passing translations have appeared in the literature. Moggi’s state monad [12] relies on a fixed store type: it does not support dynamic memory allocation. Parameterised monads [3] 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. O’Hearn and Reynolds [15] 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. Chaguéraud and Pottier [7] translate an expressive type-and-capability calculus, which supports strong references, into a purely functional calculus. 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 [24, 8] 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, but it does not suggest how to design a type-preserving store-passing translation.

## 6 Directions for future work

It would be desirable to verify our claim that the translation is adequate.

It should be possible to enrich FORK with linear types and to refine the store-passing translation so as to prove that the store is treated linearly.

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 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.

In the future, we hope to investigate the encoding of a separation logic with higher-order frame and anti-frame rules [20, 21] into a separation logic that lacks these rules but has recursive features in the style of FORK.

## Acknowledgments

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

## References

- [1] A. J. Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, 2004.
- [2] A. W. Appel, P.-A. Melliès, C. D. Richards, and J. Vouillon. [A very modal model of a modern, major, general type system](#). In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 109–122, Jan. 2007.
- [3] R. Atkey. [Parameterised notions of computation](#). *Journal of Functional Programming*, 19(3–4):355–376, 2009.
- [4] H. P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*. Elsevier Science, 1984.
- [5] L. Birkedal, K. Støvring, and J. Thamsborg. [The category-theoretic solution of recursive metric-space quations](#). Technical Report ITU-2009-119, IT University of Copenhagen, 2009.
- [6] L. Birkedal, K. Støvring, and J. Thamsborg. [Realizability semantics of parametric polymorphism, general references, and recursive types](#). In *International Conference on Foundations of Software Science and Computation Structures (FOSSACS)*, volume 5504 of *Lecture Notes in Computer Science*, pages 456–470. Springer, Mar. 2009.
- [7] A. Chaguéraud and F. Pottier. [Functional translation of a calculus of capabilities](#). In *ACM International Conference on Functional Programming (ICFP)*, pages 213–224, Sept. 2008.
- [8] R. Harper. [A simplified account of polymorphic references](#). *Information Processing Letters*, 51(4):201–206, 1994.
- [9] A. Hobor, R. Dockins, and A. W. Appel. [A theory of indirection via approximation](#). In *ACM Symposium on Principles of Programming Languages (POPL)*, Jan. 2010.

- [10] S. 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, Apr. 1999.
- [11] P. B. 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.
- [12] E. Moggi. [Notions of computation and monads](#). *Information and Computation*, 93(1), 1991.
- [13] H. Nakano. [A modality for recursion](#). In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 255–266, June 2000.
- [14] H. 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, Oct. 2001.
- [15] P. W. O’Hearn and J. C. Reynolds. [From Algol to polymorphic linear lambda-calculus](#). *Journal of the ACM*, 47(1):167–223, 2000.
- [16] S. Peyton Jones and P. Wadler. [Imperative functional programming](#). In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 71–84, Jan. 1993.
- [17] F. Pottier. A formalization of Nakano’s type system. <http://gallium.inria.fr/~fpottier/fork/>.
- [18] F. Pottier. The electronic FORK, Jan. 2010. <http://gallium.inria.fr/~fpottier/fork/>.
- [19] F. Pottier. An encoding of general references into  $F_\omega$  with certain recursive kinds (extended version). <http://gallium.inria.fr/~fpottier/publis/fpottier-fork-x.pdf>, Jan. 2010.
- [20] J. Schwinghammer, L. Birkedal, B. Reus, and H. 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, Sept. 2009.
- [21] J. Schwinghammer, H. Yang, L. Birkedal, F. Pottier, and B. Reus. [A semantic foundation for hidden state](#). In *International Conference on Foundations of Software Science and Computation Structures (FOSSACS)*, Mar. 2010. To appear.
- [22] M. H. Solomon. [Type definitions with parameters](#). In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 31–38, Jan. 1978.
- [23] C. Strachey. [Fundamental concepts in programming languages](#). *Higher-Order and Symbolic Computation*, 13(1–2):11–49, Apr. 2000.
- [24] A. K. Wright and M. Felleisen. [A syntactic approach to type soundness](#). *Information and Computation*, 115(1):38–94, Nov. 1994.