A Type System for Functional Imperative Programming (Technical Summary)

Shail Aditya Satyan Coorg

MIT Laboratory for Computer Science
545 Technology Square, Cambridge, MA 02139
{shail,satyan}@lcs.mit.edu

July 25, 1994

Abstract

In this paper, we explore the design of a powerful type system that provides a general mechanism to encapsulate low-level imperative program fragments into type-safe functional abstractions without imposing any single-threadedness constraint. Although confluence is the responsibility of the user, the type system guarantees that data-structures exported out of the functional abstraction are fully polymorphic and non-mutable and therefore can be safely embedded within functional programs without disturbing their functionality.

We present the static and the dynamic semantics of our system for a mini-language containing ML-like references. We describe the mechanism of converting mutable types into functional types and prove its soundness. Finally we discuss extensions to include arrays, tuples and general multi-level data-structures in our framework.

1 Introduction

In modern programming languages, high-level functional abstractions are often implemented using low-level imperative operations either directly within the compiler or separately within accompanying libraries. The imperative implementations have obvious advantages in terms of storage efficiency and avoiding data-copying overhead on every update. But arbitrary use of imperative constructs in a functional program destroys its nice functional properties that otherwise lead to powerful reasoning techniques and compiler optimizations. In this paper, we propose a type system which provides a reasonable solution to this problem in the context of the implicitly parallel and mostly functional programming languages Id [15] and pH [16] (a parallel version of Haskell [8]).

1.1 The Problem

Consider the problem of implementing *functional arrays* that are homogeneous, non-mutable, polymorphic arrays. The function `make_vector` creates a one dimensional functional array that memoizes a computation for a given index range. A direct and efficient way of implementing `make_vector` is to allocate memory and then fill it with the value of the computation function at each index:

Example 1:

make_vector :: $\forall t.(int \to t) \to (int, int) \to (t\ m\_vector)$

```
def make_vector f (l,u) =
   { a = alloc_vector (l,u);
     {for i <- l to u do
        a[i] := f i }
   in a };
```

The problem with providing the above definition in a standard library as opposed to hard-coding it into the compiler is the following. First, the above code is not functional and hence it is not expressible in a functional language. Second, it returns a mutable array of type $(t\ m\_vector)$ instead of a functional array of type $(t\ vector)$. There is no guarantee that the user will not tamper with the contents of the array once it is returned. Furthermore, in order to be sound, mutable objects must be restricted in polymorphism [22] which is not what we want for fully polymorphic functional arrays.

In order to express the above code in a standard library, the language must support an imperative kernel in which low-level imperative operations such as memory allocation and assignment are permitted. This means that the type system must have the capability to deal with imperative objects and functions that create and manipulate such objects in a sound manner. Furthermore, there must be a way to *encapsulate* such imperative programs that guarantees that the returned array `a` is non-mutable and is given a polymorphic functional type.

1.2 A Proposal for "Close"

There are many existing type systems in the literature that already deal with the problem of extending the Hindley/Milner type system to imperative objects [6, 10, 11, 14, 21, 22, 24]. Any one of these may be used to support an imperative kernel language in a type-safe manner. As for making the `make_vector` function return a functional array in Example 1, we introduce a *type conversion* operator called `close`:

Example 2:

make_vector :: $\forall t.(int \to t) \to (int, int) \to (t\ vector)$

```
def make_vector f (l,u) =  close { ... };
```

Now, the returned vector has a functional type which prohibits any subsequent assignments from being performed over it. In general, the `close` operator can be used to convert arbitrary non-functional data-structures like I-structures [1] and M-structures [2] into functional ones, retaining all the advantages of using these data-structures inside the `close`'d expression. The `close` operator delimits a well-defined *control block* within which the user has full flexibility and the raw power of an imperative language before exporting mutable data-structures across its boundary and rendering them functional.

1.3 Soundness Issues

However, simply converting the type of the returned result from an imperative program does not automatically ensure its non-mutability. If the returned result somehow escapes the scope of the control block in which it is being closed then that `close` operation is unsafe because the returned object may be mutated even after it is assigned a functional type. For example:

Example 3:

```
b = ref ...;
def escape_1 n =
  close { a = alloc_vector (1,n);
          ...
          b := a;                  % Storing into an external data structure
         in a };

def escape_2 n =
  close { a = alloc_vector (1,n);
          ...
          def g i v =              % Allowing a write handle to escape
            { a[i] := v;
              in v };
        in a,g };
```

In function `escape_1`, a reference to the locally allocated imperative array `a` is stored into an external object `b`. The type of the object `b` is constrained to be $((t\ m\_vector)\ ref)$ implying that the array `a` is still accessible in its open form through this indirection. The same effect is achieved in the function `escape_2`, although it is disguised in the form of a function that provides a write handle to the array being closed.

1.4 Our Work

The goal of this paper is to design a type system that automatically verifies the soundness of the type conversions implied by the `close` operations provided by the user. Our type system closes a polymorphic mutable data-structure only if it is guaranteed that no write handle pointing to that object remains accessible to the user after it is closed. Unsafe uses of the `close` operator such as those in Example 3 are flagged as compile-time "close-errors".

Note that `close` is purely a type domain operation, no run-time copying is involved while doing this type conversion. Therefore, there is no time or space overhead of using the `close` operation once it has been verified by the type system. Also, this approach may yield more parallelism because no single-threading constraint is imposed on the underlying imperative program encapsulated within the `close` operator. The following histogram example illustrates these issues:[1]

Example 4:

```
type tree t = leaf | node t (tree t) (tree t);

def histogram t n =
 close { a = alloc_vector (1,n);
         {for i <- 1 to n do
            a[i] := 0 };
         _ = accum t a n;
         ---
```

[1]We use pattern-matching to destructure and dispatch according to the shape of the given tree. In a parallel implementation, the array increment is performed atomically and the barrier (`---`) prevents the final array from being released before all the accumulations have finished.

```
        in a };

def accum leaf a n = ()
 |  accum (node x l r) a n =
   { i = hash x n;
     a[i] := a[i] + 1;
     _ = accum l a n;
     _ = accum r a n; };
```

In this example, a histogram of objects drawn from a binary tree is accumulated into an array `a` which is closed and returned at the end. No copying is involved during accumulations or at the time of returning the final array. Furthermore, the histogram initialization and the entire tree accumulation can potentially be done in parallel. Thus, the `close` construct enables returning a functional result without any copying or restricting the parallelism inherent in the underlying implementation.

From a practical standpoint, our type system provides a sound semantic interface between high-level functional constructs of the language and their imperative implementations in libraries. Furthermore, the language is not restricted to a fixed set of functional constructs built into the system; the users can program their own imperative implementations of arbitrary functional data-structures and extend existing ones using the type system without disturbing the compiler at all. As illustrated above in Example 4, our type system also enables implementing functional computations using imperative algorithms that can not otherwise be expressed in a functional style efficiently.

From a technical standpoint, we believe we have the first type system which directly models imperative implementations of functional data-structures along with a complete proof of its soundness. The main result of this paper is the soundness theorem shown in Section 4 which provides the guarantee that well-typed terms in our system do not run into dynamic type-errors. This directly implies that in a type-correct program, it is not possible to mutate an object once it is converted into a functional type.

1.5 Related Work

The *Closure Typing* system proposed by Leroy [10] is the basis of our work. This powerful type system was designed to determine when type-generalization was safe in the presence of side-effects. Our contribution is adding *regions* to this type system and recognizing that such a system has all the information needed to express and verify type-coercions between imperative and functional data-structures.

Effect systems [12, 21, 24] are closely related to our work. *Effects* and *regions* in this type system describe the side effects that expression can have. Effect systems are able to determine when it is safe to evaluate expressions in parallel [12], and when to permit type-generalization [21, 24]. Although Talpin and Jouvelot [21] use regions to *mask* side-effects that are not visible from outside, they do not convert imperative data-structures into functional ones. However, as it seems that their system is at least as powerful as the closure typing system [10], we conjecture that our system can be adapted to work in their framework. We have chosen to work with the closure typing system because it provides a direct relationship between types and actual structure of objects and a simpler theoretical framework to deal with.

A different approach to integrate imperative and (lazy) functional programming has been the focus of many recent papers [5, 20, 23, 17, 3, 18, 7, 9]. These papers attempt to introduce assignments into a functional language without destroying its strong functional properties (*e.g.*, *confluence* and *referential transparency*). To preserve these properties, they ensure that assignments in a program are *single-threaded* [19] by means of type systems [5, 20, 23], reduction rules [17], dataflow analysis [3, 18], or data abstraction and monadic function composition [7, 9]. Side-effect analysis using regions, such as ours, is orthogonal to the techniques given in these papers and such analysis could be used to make these languages or type systems more permissive. Our system inherits the properties of the underlying imperative kernel and simply provides a sound mechanism to encapsulate and convert mutable objects into functional abstractions rather than enforcing additional functional properties (*e.g.*, confluence) on the underlying kernel.

1.6 Outline

The rest of the paper is organized as follows. Section 2 describes a small expression language and its operational semantics. Section 3 describes the type system proposed in this paper. Section 4 shows the soundness of the type system. Section 5 describes a few extensions to the basic type system and Section 6 concludes.

2 Dynamic Semantics of a Mini-Language with "Close"

2.1 Syntax

The expression syntax of our mini-language is given below. For simplicity, the language provides ML-like references [14] as the only means of creating mutable data structures. A brief description of possible extensions to other data structures is given in a later section.

EXPRESSIONS:	a	$::=$	c	constant
		$\mid$	x	identifier
		$\mid$	f `where` $f(x) = a$	recursive function
		$\mid$	$a_1\ a_2$	application
		$\mid$	`let` $x = a_1$ `in` a_2	`let` binding
		$\mid$	`close` a	`close` expression

The set of constants include the usual arithmetic primitive functions $(+, -, \ldots)$, and functions to allocate, read and write locations (`ref,!,:=`). We define $\mathcal{F}(a)$ to be the set of free identifiers of the expression a.

Entities used in the dynamic semantics are defined as follows:

VALUES:	v	$::=$	c	constant
		$\mid$	$\langle \mathsf{clsr}\ f, x, a, e\rangle$	function closure
		$\mid$	l	store location
ENVIRONMENTS:	e	$::=$	IDENTIFIERS $\rightarrow$ VALUES	
STORES:	s	$::=$	LOCATIONS $\rightarrow$ VALUES $\times$ TAG	

Each location in a store has a tag, rw or ro, indicating that the location has read/write or read-only semantics. This is useful in modeling the dynamic semantics of the `close` operation where the tag of a location is explicitly changed from rw to ro. We assume functions $value(s, l)$ and $tag(s, l)$ that read the value and tag respectively from a location.

CONST: $$e \vdash c/s \Rightarrow c/s$$

IDENT: $$\frac{x \in Dom(e)}{e \vdash x/s \Rightarrow e(x)/s}$$

ABS: $$\frac{Y = \mathcal{F}(f \texttt{ where } f(x) = a)}{e \vdash (f \texttt{ where } f(x) = a)/s \Rightarrow \langle \textsf{clsr}\ f, x, a, e \mid_Y \rangle / s}$$

APP: $$\frac{\begin{array}{c} e \vdash a_1/s \Rightarrow \langle \textsf{clsr}\ f, x, a_0, e_0 \rangle / s_1 \\ e \vdash a_2/s_1 \Rightarrow v_2/s_2 \\ e_0 + \{f \mapsto \langle \textsf{clsr}\ f, x, a_0, e_0 \rangle, x \mapsto v_2\} \vdash a_0/s_2 \Rightarrow v/s_3 \end{array}}{e \vdash (a_1\ a_2)/s \Rightarrow v/s_3}$$

LET: $$\frac{e \vdash a_1/s \Rightarrow v_1/s_1 \qquad e + \{x \mapsto v_1\} \vdash a_2/s_1 \Rightarrow v_2/s_2}{e \vdash (\texttt{let } x = a_1 \texttt{ in } a_2)/s \Rightarrow v_2/s_2}$$

ALLOC: $$\frac{e \vdash a/s \Rightarrow v/s_1 \qquad l \notin Dom(s_1)}{e \vdash \texttt{ref}(a)/s \Rightarrow l/(s_1 + \{l \mapsto v, \mathrm{rw}\})}$$

DEREF: $$\frac{e \vdash a/s \Rightarrow l/s_1 \qquad l \in Dom(s_1) \qquad value(s_1, l) = v}{e \vdash \texttt{!}a/s \Rightarrow v/s_1}$$

ASSIGN: $$\frac{e \vdash a/s \Rightarrow (l, v)/s_1 \qquad l \in Dom(s_1) \qquad tag(s_1, l) = \mathrm{rw}}{e \vdash \texttt{:=}(a)/s \Rightarrow \texttt{()}/(s_1 + \{l \mapsto v, \mathrm{rw}\})}$$

CLOSE: $$\frac{\begin{array}{c} e \vdash a/s \Rightarrow l/s_1 \qquad s_1(l) = v, \mathrm{rw} \\ L = Reachable(l, s_1) \cup Reachable(e, s_1) \cup \bigcup_{l' \in Dom(s)} Reachable(l', s_1) \end{array}}{e \vdash (\texttt{close } a)/s \Rightarrow l/(s_1 \mid_L + \{l \mapsto v, \mathrm{ro}\})}$$

Figure 1: The Dynamic Semantics of the Mini-Language.

For a value v, we define $\mathcal{L}(v)$ to be the set of locations directly contained in the value v. This is defined structurally as follows: $\mathcal{L}(c) = \phi$, $\mathcal{L}(l) = \{l\}$, and $\mathcal{L}(\langle \textsf{clsr}\ f, x, a, e \rangle) = \mathcal{L}(e)$. For an environment $e = \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$, we define $\mathcal{L}(e) = \bigcup_{1 \le i \le n} \mathcal{L}(v_i)$.

We also define $Reachable(v, s)$ to be the set of all locations within the domain of store s that are either directly contained within the value v or transitively contained in a value stored at such a location *via* the store s. This extends naturally (pointwise) to values present in an environment e.

2.2 Evaluation Rules

Figure 1 shows the axioms and inference rules for establishing EVALUATION JUDGMENTS of the form $e \vdash a/s \Rightarrow v/s'$. We do not provide "error" rules explicitly; a dynamic type-error occurs when the evaluation becomes "stuck". Most of the rules are fairly standard and self-explanatory. The rules for primitive reference operators are handled as special cases of the general APP rule. The ASSIGN rule makes sure that the location being written carries a read/write tag.

The CLOSE rule is the only place where the read/write tag of a location is explicitly changed to a read-only tag. This causes a dynamic "close-error" to be generated if that location is subsequently assigned. Ultimately, we would like to detect such dynamic errors at compile-time using the type system. We also restrict the domain of the final store in the CLOSE rule to the reachable locations of the location being closed, the current environment and the locations of the initial store. This operation "cleans" the store by removing spurious unreachable handles from its domain that point to the location being closed.[2]

Although the rules given in Figure 1 provide a strict, sequential, call-by-value semantics for the mini-language, our technique does not rely on preserving the sequentiality or the determinism of evaluation. This is seen from the fact that even if we make the language non-deterministic by adding an additional APP rule which interchanges the order in which the function and the argument are evaluated, our results on `close` still hold.[3]

3 Static Semantics

Now we will briefly describe Xavier Leroy's closure typing system [10] and our extensions to it. A type τ in our system is defined by:

$$\text{TYPES:} \qquad \tau ::= t \mid \iota \mid \tau_1 \!-\!\langle\, \pi \rangle\!\!\rightarrow \tau_2 \mid \tau\ \mathit{ref}(r) \mid \tau\ \mathit{ref}(\epsilon)$$

where t corresponds to an ordinary type variable and ι corresponds to a base type. Type schemes ($\sigma = \forall \alpha_1 \ldots \alpha_n . \tau$) are quantified types.

Functions in Leroy's system are decorated with a set of *closure types* (denoted by $\pi = \sigma_1, \sigma_2, \ldots, u$), which corresponds to the type schemes of the free identifiers of a function. The closure extension variable (u) allows us to extend the set of closure types in a meaningful manner during type unification. These closure types are useful in keeping track of the mutable objects hidden inside a dynamic function closure.

We extend the closure typing system by parameterizing type constructors with *regions*. A region variable parameter (r) on a reference type constructor serves two purposes. It identifies the reference type as being mutable and it also serves as a static abstraction for all the mutable locations that have that type. Non-mutable references are identified by a null region parameter (ϵ) which shows that they have been closed. Note that $\mathit{ref}(r)$ and $\mathit{ref}(\epsilon)$ are considered to be distinct type constructors for the purpose of type unification; they have a similar form only for syntactic uniformity.

For any type object T, where T may be a type, a closure type, a region, or a type scheme, we can define its FREE VARIABLES $\mathcal{F}(T)$ as the set of all type, region, and closure extension variables contained in T. We also define the DANGEROUS VARIABLES $\mathcal{D}(T)$ as follows:

$$\begin{array}{rcl@{\qquad}rcl}
\mathcal{D}(t) & = & \phi & \mathcal{D}(u) & = & \phi \\
\mathcal{D}(\iota) & = & \phi & \mathcal{D}(\sigma, \pi) & = & \mathcal{D}(\sigma) \cup \mathcal{D}(\pi) \\
\mathcal{D}(\tau_1 \!-\!\langle\, \pi \rangle\!\!\rightarrow \tau_2) & = & \mathcal{D}(\pi) & \mathcal{D}(r) & = & \phi \\
\mathcal{D}(\tau\ \mathit{ref}(r)) & = & \mathcal{F}(\tau\ \mathit{ref}(r)) & \mathcal{D}(\forall \alpha_1 \ldots \alpha_n .\, \tau) & = & \mathcal{D}(\tau) \setminus \{\alpha_1 \ldots \alpha_n\} \\
\mathcal{D}(\tau\ \mathit{ref}(\epsilon)) & = & \mathcal{D}(\tau) & \mathcal{D}(\epsilon) & = & \phi
\end{array}$$

[2] This restriction is only necessary to simplify our soundness proof.

[3] Using a truly "parallel" semantics would be a more convincing argument, but we are not aware of a relational style semantics which captures parallelism. Adapting our system to a "parallel" semantics is a topic for future research.

CONST: $$\frac{typeof(c) \geq \tau}{E \vdash c : \tau}$$

IDENT: $$\frac{x \in Dom(E) \qquad E(x) \geq \tau}{E \vdash x : \tau}$$

ABS: $$\frac{\begin{array}{c}\{y_1 \ldots y_n\} = \mathcal{F}(f \texttt{ where } f(x) = a) \\ E + \{f \mapsto \tau_1 \text{-}\langle E(y_1), \ldots, E(y_n), \pi \rangle\!\!\to \tau_2, x \mapsto \tau_1\} \vdash a : \tau_2\end{array}}{E \vdash (f \texttt{ where } f(x) = a) : \tau_1 \text{-}\langle E(y_1), \ldots, E(y_n), \pi \rangle\!\!\to \tau_2}$$

APP: $$\frac{E \vdash a_1 : \tau_1 \text{-}\langle \pi \rangle\!\!\to \tau_2 \qquad E \vdash a_2 : \tau_1}{E \vdash a_1\ a_2 : \tau_2}$$

LET: $$\frac{E \vdash a_1 : \tau_1 \qquad E + \{x \mapsto Gen(E, \tau_1)\} \vdash a_2 : \tau_2}{E \vdash (\texttt{let } x = a_1 \texttt{ in } a_2) : \tau_2}$$

CLOSE: $$\frac{E \vdash a : \tau\ ref(r) \qquad r \notin (\mathcal{F}(E) \cup \mathcal{F}(\tau))}{E \vdash (\texttt{close } a) : \tau\ ref(\epsilon)}$$

Figure 2: The Static Semantics of the Mini-Language.

Finally, the DANGEROUS REGION VARIABLES $\mathcal{R}(T)$ are the region variables in $\mathcal{D}(T)$.

TYPE SUBSTITUTIONS (φ) over this type algebra are finite mappings from regular type variables to types, from closure extension variables to closure types, and from region variables to other region variables. We do not allow substituting region variables with the null region (ϵ) because that would convert a mutable reference type into a non-mutable reference type. This operation should only be performed when it is determined to be safe and is explicitly done using the `close` construct. Type substitutions extend naturally over types, closure types, and type schemes, being applied to their free variables in each case. The *instantiation* of a type scheme $\sigma = \forall \alpha_1 \ldots \alpha_n . \tau_0$ to a type τ, written as $\sigma \geq \tau$, is defined if there exists a type substitution φ with $Dom(\varphi) \subseteq \{\alpha_1 \ldots \alpha_n\}$ such that $\tau = \varphi(\tau_0)$.

The basic idea of our type system is to use the type of a composite object as a clue to the reachable mutable reference locations contained within it. Dangerous variables provide this clue directly from the overall type of an object. Intuitively, dangerous *type* variables model the polymorphic values stored in mutable data-structures and the dangerous *region* variables model the locations of those mutable data-structures.

3.1 Typing Rules

Figure 2 shows the axioms and inference rules for establishing ELABORATION JUDGMENTS of the form $E \vdash a : \tau$, where E is a type environment that maps identifiers to type schemes. The CONST rule establishes the type of a constant according to a predefined relation *typeof* that provides the type scheme associated with it. Most constants and operators have the obvious type schemes. We only show the predefined type schemes of the three reference operators below:

$$\begin{array}{rcl} \mathit{typeof}(\mathtt{ref}) & = & \forall t,u,r.\ t \multimap\langle u\rangle\!\!\rightarrow t\ \mathit{ref}(r) \\ \mathit{typeof}(\mathtt{!}_{\mathit{mutable}}) & = & \forall t,u,r.\ t\ \mathit{ref}(r) \multimap\langle u\rangle\!\!\rightarrow t \\ \mathit{typeof}(\mathtt{!}_{\mathit{non\text{-}mutable}}) & = & \forall t,u.\ t\ \mathit{ref}(\epsilon) \multimap\langle u\rangle\!\!\rightarrow t \\ \mathit{typeof}(\mathtt{:=}) & = & \forall t,u,r.\ (t\ \mathit{ref}(r), t) \multimap\langle u\rangle\!\!\rightarrow \mathit{unit} \end{array}$$

The CONST, IDENT, APP, and LET rules are standard Hindley/Milner typing rules [13]. The *generalization* operation in the LET rule excludes the dangerous variables $\mathcal{D}(\tau)$ from being generalized:

$$\mathit{Gen}(E,\tau) = \forall\alpha_1 \ldots \alpha_n.\tau \quad \text{where} \quad \{\alpha_1 \ldots \alpha_n\} = \mathcal{F}(\tau) \setminus \mathcal{D}(\tau) \setminus \mathcal{F}(E)$$

This ensures that the type variables corresponding to mutable objects are not generalized, making our type-generalization sound.

The ABS rule shows how closure types are created in this system. The type schemes of all the free identifiers of the function are stored in its closure type. This helps to expose the mutable locations hidden inside the closure environment of a function so that closing such data-structures may be caught as a compile-time close-error (Example 3).

Finally, the CLOSE rule converts a mutable reference type into a non-mutable reference type by *erasing* its region variable r and replacing it with the null region (ϵ). By checking that r does not occur free in the type environment E or in the exported type τ, we ensure that the corresponding reference location l being closed is not escaping from the current scope. Informally, $r \notin \mathcal{F}(E)$ implies that l is not reachable from the value environment (e) and $r \notin \mathcal{F}(\tau)$ implies that l is not recursively reachable from the value stored in the location l. Thus, no write handle to the location l is escaping and it can be safely closed. A compile-time close-error is flagged if this check fails.

As an aside, we note that a type inference algorithm can be designed for this type system along the lines of [10]. We do not discuss that here, the reader is referred to [10] for details.

4 Type Soundness

4.1 Semantic Consistency

In order to show the soundness of the typing judgments generated by the above type system with respect to its evaluation rules, we must precisely characterize a "consistent" semantic relationship between value-domain entities and their corresponding type-domain entities. Since values may contain reference locations from the store, we introduce STORE TYPINGS (S) as finite mappings from store locations to types. We also define that a store typing S' *extends* another store typing S if $\mathit{Dom}(S) \subseteq \mathit{Dom}(S')$ and $S(l) = S'(l)$ for all $l \in \mathit{Dom}(S)$. Semantic consistency is now defined below:

Definition 1 (Semantic Model) *Let s be a store, S be a store typing, e be an environment, E be a type environment, v be a value, τ be a type, and σ be a type scheme. Define the following relations:*

Case *1: $S \models v : \tau$ — The value v belongs to the type τ under the store typing S. The various cases are as follows:*

SubCase *1.1: $S \models c : \mathit{typeof}(c)$, where typeof is a predefined relation between predefined constants and their types.*

SubCase *1.2:* $S \models \langle \mathsf{clsr}\ f, x, a, e\rangle : \tau_1 \neg\langle \pi \rangle\!\!\rightarrow \tau_2$, *if there exists a type environment* E *such that* $S \models e : E$ *and* $E \vdash (f\ \mathtt{where}\ f(x) = a) : \tau_1 \neg\langle \pi \rangle\!\!\rightarrow \tau_2$.

SubCase *1.3:* $S \models l : \tau\ ref(r)$, *if* $l \in Dom(S)$ *and* $S(l) = \tau\ ref(r)$.

SubCase *1.4:* $S \models l : \tau\ ref(\epsilon)$, *if* $l \in Dom(S)$ *and there exists a substitution* φ *with* $Dom(\varphi) \subseteq \mathcal{F}(S(l)) \setminus \mathcal{D}(S(l))$ *such that* $\varphi(S(l)) = \tau\ ref(\epsilon)$.

Case *2:* $S \models v : \sigma$ — *The value* v *belongs to the type scheme* $\sigma = \forall \alpha_1 \ldots \alpha_n.\tau$ *under the store typing* S, *if none of* α_i *belong to* $\mathcal{D}(\tau)$ *and if* $S \models v : \varphi(\tau)$ *for all substitutions* φ *with* $Dom(\varphi) \subseteq \{\alpha_1 \ldots \alpha_n\}$.

Case *3:* $S \models e : E$ — *The values contained in the environment e belong to the corresponding type schemes in the type environment E (pointwise) under the store typing* S, *if* $Dom(e) = Dom(E)$ *and for all* $x \in Dom(E)$ *we have* $S \models e(x) : E(x)$.

Case *4:* $\models s : S$ — *The values contained in the store s belong to the corresponding types in the store typing S (pointwise), if* $Dom(s) = Dom(S)$ *and for all* $l \in Dom(S)$ *we have,*

SubCase *4.1: If* $S(l) = \tau\ ref(r)$ *then* $s(l) = v, \mathrm{rw}$ *and* $S \models v : \tau$.

SubCase *4.2: If* $S(l) = \tau\ ref(\epsilon)$ *then* $s(l) = v, \mathrm{ro}$ *and* $S \models v : \tau$.

The above definition models the type polymorphism and the dynamic mutability of closed and mutable locations in a consistent manner. SubCase 1.4 allows a closed location (typed using the null-region ϵ) to be typed polymorphically, while a mutable location in SubCase 1.3 (typed using a region variable r) is allowed to have only a monomorphic type. Similarly, closed locations in a consistent store are defined to possess the read-only tag (SubCase 4.2), while mutable locations must have the read/write tag (SubCase 4.1).

4.2 Type Soundness and Non-Mutability

The following proposition establishes the essential correspondence between the dangerous regions of a type and the mutable locations that are reachable from a value possessing that type. This allows us to use dangerous regions as a safe static abstraction for mutable locations. The proof is done using induction on the *depth* of reachability of a location l in the value v.

Proposition 2 (Region Abstraction) *Let s be a store and S be a store typing such that* $\models s : S$. *If* $S \models v : \tau$, $l \in Reachable(v, s)$ *and* $r \in \mathcal{R}(S(l))$, *then* $r \in \mathcal{R}(\tau)$. *That is, the dangerous regions contained in the types of reachable locations of a value are dangerous in the type of the value. This proposition can be extended pointwise to environments.*

The semantic consistency between the static and the dynamic semantics can now be stated in the form of the soundness theorem given below. It is proved using induction on the size of evaluation derivation, doing a case analysis on a and hence on the last rule used in the typing derivation.

Theorem 3 (Type Soundness) *Let a be an expression,* τ *be a type, E be a type environment, e be an evaluation environment, s be an initial store, and S be a store typing such that:*

$$E \vdash a : \tau \qquad \textit{and} \qquad S \models e : E \qquad \textit{and} \qquad \models s : S$$

If there exists a result r such that $e \vdash a/s \Rightarrow r$, *then* $r \neq \mathtt{err}$, *instead* $r = v/s'$ *for some value v and a resulting store* s', *and there exists a store typing* S' *such that:*

$$S' \textit{ extends } S \qquad \textit{and} \qquad S' \models v : \tau \qquad \textit{and} \qquad \models s' : S'$$

The soundness of the `close` operation relies on the fact that it only closes *fresh* and *non-escaping* locations, *i.e.*, locations that are neither present in the initial store s nor are accessible from the environment e or hidden inside the result v. The former is a property of the dynamic rules and the latter is ensured by the side condition on the static CLOSE rule and Proposition 2. A proof sketch for this case is provided in the appendix. The details appear in [4].

The soundness theorem immediately leads us to the following corollary that guarantees that objects with a closed functional type can not be mutated at run-time.

Corollary 4 (Non-Mutability) *Let a be an expression fragment within a type correct program p such that $E \vdash a : \tau$ where $\mathcal{R}(\tau) = \phi$ and $e \vdash a/s \Rightarrow v/s'$. Then, no location $l \in Reachable(v, s')$ is mutated during the evaluation of the rest of the program.*

Proof: Using the soundness theorem we know that the evaluation of p (and hence a) does not lead to error and there exists a store typing $S' \models v : \tau$ and $\models s' : S'$. We claim that for all locations $l \in Reachable(v, s')$ we must have $tag(s', l) = \text{ro}$. Otherwise, from Definition 1 Case 4 it follows that there exists a region variable r_1 such that $S'(l) = \tau_1\ ref(r_1)$; then using Proposition 2 it follows that $r_1 \in \mathcal{R}(\tau)$, which contradicts the hypothesis $\mathcal{R}(\tau) = \phi$.

Now, sound uses of the ASSIGN rule in Figure 1 require that the tag of the location being assigned should be rw. Furthermore, there is no rule that converts the tag of a location from ro to rw. Therefore, no assignments are possible on any location $l \in Reachable(v, s')$ in the rest of the program. $\square$

5 Extensions

In this section we describe how to extend the use of the `close` construct to mutable arrays and arbitrary, multi-level data-structures involving a combination of references, tuples, functions and arrays.[4]

5.1 Arrays

The treatment of individual reference locations in Sections 2 and 3 directly extends to arrays of locations. A 1-dimensional array value is modeled as a pair $\langle \mathsf{vect}\ l, \underline{n} \rangle$ which denotes n consecutive array locations starting from l. Since arrays are considered to be homogeneous data-structures, a single region variable suffices to represent the imperative properties of all the locations within an array. For example, a mutable vector containing values of type τ is typed as $(\tau\ vector(r))$ (*c.f.* mutable references with type $(\tau\ ref(r))$).

The primitive dynamic rules for array allocation (`alloc_vector (l,u)`), array dereference (`a[i]`) and array assignment (`a[i]:=b`) follow the corresponding rules for references (see Figure 1). During vector allocation, n fresh locations are added to the domain of the store all of which possess the read/write tag. The dynamic CLOSE rule for a vector closes all its locations simultaneously. The safety of this operation is ensured by the static CLOSE rule (see Figure 2) that allows converting a mutable vector type $(\tau\ vector(r))$ into a non-mutable vector type $(\tau\ vector(\epsilon))$ only if the region r is not present in the enclosing type environment E and in the exported element type τ.

[4]Extensions to include general algebraic datatypes will be discussed in the full paper. We have dropped that discussion from this technical summary due to lack of space.

The above machinery allows us to finally answer the problem in Section 1.1 regarding functional arrays. The imperative implementations for functions `make_vector` (Example 2) and `histogram` (Example 4) can now be verified and closed automatically.

5.2 Multi-level Data-Structures

In our system, for each primitive or user-defined mutable datatype we have specific rules regarding which regions may be closed and how to verify the soundness of the closing operation. But in general, the result of an arbitrary `close` expression may be a multi-level data-structure involving a combination of references, arrays, tuples, and functions. The user may want all or some of the locations present in the result to be closed. This can be specified in our type system using a *type annotation*:

$$\texttt{close}\ a :: \tau$$

The user annotates the expression to be closed with an appropriately decorated type τ that shows all qualified type constructors along with their region parameters. Mutable types that need to be closed are annotated with the null region (ϵ).

In order to verify the soundness of such arbitrary `close` expressions, the type system effectively synthesizes specialized dynamic and static CLOSE rules on the lines of the semantics presented in Sections 2 and 3. First, the annotation type is matched against the inferred type of the expression to determine the exact set of region variables being closed. In order for the overall close operation to be sound, none of the region variables being closed should be present in the current type environment E or the annotation type τ (*c.f.* CLOSE rule in Figure 2).

6 Conclusions

In this paper, we have presented a powerful type system which provides a general mechanism to encapsulate low-level, imperative program fragments into sound, high-level, functional abstractions without imposing any single-threadedness constraint. We achieved this by adding a new construct to the kernel language called `close`, that changes the *view* of a mutable data-structure from imperative to a functional one. The type system statically verifies the soundness of such a change and guarantees dynamic functional behavior for the closed object.

We also briefly described how to extend the use of the `close` construct to arrays and general multi-level data-structures. Our proposal for syntactically specifying imperative expressions that may be closed using type annotations provides complete flexibility to the user while enabling the type system to automatically verify the soundness of this operation.

Our type system can be viewed as a safe trap-door to embedded imperative programming within an overall functional setting. The user is allowed full flexibility in constructing arbitrary functional abstractions using a controlled imperative approach. We believe that this is the best way to obtain the potential benefits of imperative programming in a functional language without obscuring parallelism and expressive power of the underlying imperative kernel.

7 Acknowledgments

The research described in this paper was funded in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-89-J-1988.

The authors would like to thank Prof. Arvind, Xavier Leroy and Yuli Zhou for their helpful comments and suggestions.

A Proof Sketch for the Soundness Theorem

First, we state some useful properties of the evaluation rules (Figure 1), static rules (Figure 2) and the semantic model (Definition 1). The details of all these proofs appear in [4].

A.1 Properties of the Evaluation Rules

The following propositions are shown by induction on the length of the evaluation derivation for the expression a, doing a case analysis on the last evaluation rule used in the derivation.

Proposition 5 (Fresh Locations) *Let a be an expression, v be a value, e be an environment, and s_0, s_1 be initial and final stores respectively such that $e \vdash a/s_0 \Rightarrow v/s_1$. Then,*

$$l' \in \mathit{Reachable}(v, s_1) \Longrightarrow l' \notin \mathit{Dom}(s_0) \bigvee l' \in \mathit{Reachable}(e, s_0)$$

and for all locations $l \in \mathit{Dom}(s_0)$,

$$l' \in [\mathit{Reachable}(l, s_1) \setminus \mathit{Reachable}(l, s_0)] \Longrightarrow l' \notin \mathit{Dom}(s_0) \bigvee l' \in \mathit{Reachable}(e, s_0)$$

That is, locations reachable from a value computed in an evaluation are reachable from the environment or they are fresh locations.

Proposition 6 (Changed Locations) *Let a be an expression, v be a value, e be an environment, and s_0, s_1 be initial and final stores respectively such that $e \vdash a/s_0 \Rightarrow v/s_1$. Then for any location $l \in \mathit{Dom}(s_0)$,*

$$\mathit{value}(s_0, l) \neq \mathit{value}(s_1, l) \Longrightarrow l \in \mathit{Reachable}(e, s_0)$$

That is, only the locations reachable from the environment may change their value during an evaluation.

A.2 Properties of the Typing Rules

The following proposition states that typing is stable under type substitution. The proof of this proposition is done using structural induction over a, doing a case analysis on the last typing rule used in the typing derivation of a.

Proposition 7 (Stability under Type Substitution) *Let a be an expression, τ be a type, E be a type environment, and φ be a substitution. If $E \vdash a : \tau$, then $\varphi(E) \vdash a : \varphi(\tau)$.*

A.3 Properties of the Semantic Model

The following proposition shows that a semantic relation such as $S \models v : \tau$ that holds true at some point during evaluation, remains true under an extended store typing.

Proposition 8 (Store Typing Extension) *If S' extends S, then $S \models v : \tau$ implies $S' \models v : \tau$. Similarly, $S \models e : E$ implies $S' \models e : E$.*

The following proposition establishes the fact that it is semantically safe to generalize the non-dangerous variables of a type. It is proved using structural induction over v using the definition of $\models$ (Definition 1).

Proposition 9 (Semantic Generalization) *Let v be a value, τ be a type and S be a store typing such that $S \models v : \tau$. Let $\alpha_1, \ldots, \alpha_m$ be type variables such that for all i, $\alpha_i \notin \mathcal{D}(\tau)$. Then, for all substitutions φ with $Dom(\varphi) \subseteq \{\alpha_1 \ldots \alpha_m\}$, we have $S \models v : \varphi(\tau)$. As a consequence, $S \models v : \forall \alpha_1 \ldots \alpha_m . \tau$.*

A.4 The Soundness Theorem

Now, we sketch the proof of the CLOSE case of the soundness theorem.

Proof: The proof is by induction on the size of evaluation derivation. The argument is by case analysis on a. Refer to the static (Figure 2) and dynamic (Figure 1) CLOSE rules. Using the induction hypothesis on a, we obtain $e \vdash a/s \Rightarrow l/s_1$ with the store typing S_1 such that $S_1 \models l : \tau\ ref(r)$, $\models s_1 : S_1$ and S_1 extends S. From the first two clauses and the definition of $\models$ for mutable locations, we obtain, $l \in Dom(S_1)$, $S_1(l) = \tau\ ref(r)$, $s_1(l) = v, \mathrm{rw}$ and $S_1 \models v : \tau$. Thus, the CLOSE evaluation rule applies. Define $s' = s_1 \mid_L + \{l \mapsto v, \mathrm{ro}\}$ and $S' = S_1 \mid_L + \{l \mapsto \tau\ ref(\epsilon)\}$. Now, we have to show the following:

$$S' \models l : \tau\ ref(\epsilon) \qquad \text{and} \qquad \models s' : S' \qquad \text{and} \qquad S' \text{ extends } S \tag{1}$$

The first clause follows directly from the definition of $\models$ for non-mutable locations since we have chosen $l \in Dom(S')$ and $S'(l) = \tau\ ref(\epsilon)$.

Next, we show that $l \notin Reachable(e, s)$. If not, $r \in \mathcal{R}(S_1(l)) = \mathcal{R}(S(l))$ (as S_1 extends S) $\subseteq \mathcal{R}(E)$ (by Proposition 2) $\subseteq \mathcal{F}(E)$, which contradicts the condition $r \notin \mathcal{F}(E)$ of the typing rule. Similarly, we can show that $l \notin Reachable(e, s_1)$.

Now, we show that S' extends S. Note that S_1 extends S if and only if $l \notin Dom(S)$, since that is the only location at which S_1 and S' differ. As we know that $l \notin Reachable(e, s)$, Proposition 5 on the evaluation $e \vdash a/s \Rightarrow l/s_1$ implies that $l \notin Dom(s) = Dom(S)$.

As the final step in proving Equation 1, we have to show $\models s' : S'$. By construction of S', the tags of s' are consistent with S'. To show consistency for values in the store, consider a location l' such that $value(s', l') = v'$ and $S'(l') = \tau'$. If v' does not contain l, we can show that $S' \models v' : \tau'$ by a simple structural induction on v' (using the fact that $S_1 \models v' : \tau'$ and that l is the only location in which S_1 and S' differ). To complete the proof, we show that l is not *reachable* from any v' (in s_1 and hence in s') which implies the above property.

Assume that $l \in Reachable(v', s_1)$. Looking at the components of $Dom(s')$, there are three possibilities for l':

1. $l' = l$ — Then $v' = v$. Given $l \in Reachable(v, s_1)$, we apply Proposition 2 to l and v to conclude that $r \in \mathcal{R}(S_1(l)) \subseteq \mathcal{R}(\tau) \subseteq \mathcal{F}(\tau)$ which contradicts the condition $r \notin \mathcal{F}(\tau)$ in the typing rule.
2. $l' \in Reachable(e, s_1)$ — This implies $l \in Reachable(e, s_1)$, a contradiction.

3. $l' \in \mathit{Reachable}(\mathit{Dom}(s), s_1)$ — We know that l was not reachable from any value present in the domain s initially, *i.e.*, $l \notin \mathit{Reachable}(\mathit{Dom}(s), s)$ because we have already shown that $l \notin \mathit{Dom}(s)$. Thus, the only way l could become reachable from $\mathit{Dom}(s)$ after the evaluation $e \vdash a/s \Rightarrow v/s_1$ is if some location in $\mathit{Dom}(s)$ was assigned a new value from which l was reachable. Without loss of generality, let us assume that location is l' and the newly assigned value is v', *i.e.*,

$$\exists\, l' \in \mathit{Dom}(s): \quad \mathit{value}(s, l') \neq \mathit{value}(s_1, l') = v' \quad \text{and} \quad l \in \mathit{Reachable}(v', s_1) \tag{2}$$

By Proposition 6, $l' \in \mathit{Reachable}(e, s)$. Applying Proposition 2 to l and v', we obtain $r \in \mathcal{R}(S_1(l'))$. Thus, $r \in \mathcal{R}(S(l'))$. Applying Proposition 2 to l' and e, we obtain $r \in \mathcal{R}(E) \subseteq \mathcal{F}(E)$, a contradiction.

□

References

[1] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-Structures: Data Structures for Parallel Computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, 1989.

[2] Paul S. Barth. Atomic Data Structures for Parallel Computing. Technical Report MIT/LCS/TR-532, Laboratory for Computer Science, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139, March 1992.

[3] A. Bloss. Update analysis and the efficient implementation of functional aggregates. In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture, London, UK*. ACM Press, September 1989.

[4] Shail Aditya Gupta. *A Typed Approach to Layered Programming Language Design*. PhD thesis, Massachusetts Institute of Technology, December 1994. (In Preparation).

[5] J. Guzman and P. Hudak. Single-threaded polymorphic lambda calculus. In *Proceedings of Fifth Annual Symposium on Logic in Computer Science*, pages 333–343. ACM Press, June 1990.

[6] My Hoang, John Mitchell, and Ramesh Viswanathan. Standard ML weak polymorphism and imperative constructs. In *Proceedings of the Eighth Annual Symposium on Logic in Computer Science*, pages 15–25. ACM Press, June 1993.

[7] P. Hudak and D. Rabin. Mutable abstract datatypes - or - how to have your state and munge it too. Technical Report YALEU/DCS/RR-914, Yale University, Department of Computer Science, July 1992.

[8] P. Hudak and P. Wadler (editors). Report on the programming language Haskell, a non-strict purely functional language (Version 1.0). Technical Report YALEU/DCS/RR777, Yale University, Department of Computer Science, April 1990.

[9] John Launchbury and Simon L. Peyton Jones. Lazy Functional State Threads. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, Orlando, Florida, USA*. ACM Press, June 1994.

[10] Xavier Leroy. Polymorphic Typing of an Algorithmic Language. Rapports de Recherche 1778, INRIA, Rocquencourt, France, October 1992. English translation of the author's Ph.D. thesis originally in French.

[11] Xavier Leroy and Pierre Weis. Polymorphic type inference and assignment. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 291–302. ACM Press, January 1991.

[12] John M. Lucassen and David K. Gifford. Polymorphic Effect Systems. In *Proceedings of the Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming languages, San Diego, California*, pages 47–57, January 1988.

[13] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[14] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* The MIT Press, Cambridge, Massachusetts, 1990.

[15] Rishiyur S. Nikhil. Id Language Reference Manual Version 90.1. Technical Report CSG Memo 284-2, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, July 15 1991.

[16] Rishiyur S. Nikhil, Arvind, and James Hicks. pH Language Proposal (Preliminary). Circulated on the pH mailing list, September 1993.

[17] M. Odersky, D. Rabin, and P. Hudak. Call by Name, Assignment, and the Lambda Calculus. In *Proceedings of the 1993 ACM Conference on Principles of Programming Languages*, pages 43–56. ACM Press, 1993.

[18] A. V. S. Sastry, William Clinger, and Zena Ariola. Order-of-evaluation Analysis for Destructive Updates in Strict Functional Languages with Flat Aggregates. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark*, pages 266–275. ACM Press, June 1993.

[19] D. Schmidt. Detecting global variables in denotational specifications. *ACM Transactions on Programming Languages and Systems*, 5(2):299–310, 1985.

[20] V. Swarup, U. S. Reddy, and E. Ireland. Assignments for applicative languages. In *Functional Programming Languages and Computer Architecture*, pages 192–214. Springer-Verlag, February 1991. Lecture Notes in Computer Science, volume 523.

[21] Jean-Pierre Talpin and Pierre Jouvelot. The Type and Effect Discipline. In *Proceedings of the ACM Symposium on Logic in Computer Science*, pages 162–173. ACM Press, 1992.

[22] Mads Tofte. Type Inference for Polymorphic References. *Information and Computation*, 89:1–34, 1990.

[23] Philip Wadler. Linear types can change the world! In *Proceedings of the Working Conference on Programming Concepts and Methods, Israel*, pages 385–407. North-Holland, 1990.

[24] Andrew K. Wright. Typing References by Effect Inference. In *Proceedings of the 4th European Symposium on Programming, Rennes, France*, pages 473–491. Springer-Verlag, February 1992. Lecture Notes in Computer Science, volume 582.