Safe Functional Abstractions in a Polymorphic, Imperative Language*

Shail Aditya
NE43-249, MIT Laboratory for Computer Science
shail@lcs.mit.edu

1 Introduction

Functional data structures are useful in optimizations and parallelization of programs because their value cannot be changed after it is defined. Functional behavior enables compiler optimizations such as fetch-elimination and common sub-expression elimination, enhances parallelism by eliminating redundant control dependence due to side-effects, and permits run-time caching of data without worrying about coherency. However, functional data structures need not be computed in a purely functional manner; it is sometime more appropriate or efficient to have the ability to perform side-effects while building a functional data structure. In this paper we describe a type-based approach for creating safe, polymorphic, functional data abstractions in an otherwise imperative language [2].

We start with a strongly-typed, polymorphic, imperative language such as Id and add a new data encapsulation construct to it called `close` (see Figure 1). This construct is used to convert locally allocated imperative objects into functional objects by restricting their side-effects to a given program block without unnecessary sequentialization. As an example, consider the following program in Id which constructs a histogram from a given list of values using side-effects, and then restricts the array to be functional before returning it.

Example 1:

```
def histogram l n =
  close { def zero i = 0;
           a = make_marray (1,n) zero;
           _ = accumulate a l;
          ---
          in a };
```

*Thesis Supervisor: Prof. Arvind. The research described in this paper was funded in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-92-J-1310.

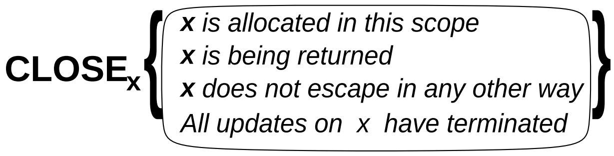

Figure 1: Definition of `close`.

The `make_marray` function allocates and initializes a mutable array. The `accumulate` function builds the histogram in parallel using side-effects. Once all the accumulations have finished (ensured by the barrier), the `close` construct changes the "view" of the array from imperative to functional and returns it. Note that `close` is a purely compile-time operation—no data copying or any other run-time overhead is involved. Furthermore, all activities within the scope of the `close` construct (including side-effects) are allowed to proceed in parallel—no single-threading is necessary as in update-in-place analysis for functional languages [1].

The key idea is that the `close` operation must be *dynamically* safe, *i.e.*, closed objects must never be updated again dynamically. The main contribution of this research is that the soundness of the `close` operation can be checked and enforced *statically* via a simple extension to the traditional Hindley/Milner type system. We present the key features of this type system below.

2 The Type System

Our type system is based on the *closure typing* system of Leroy [3] that safely extends the traditional polymorphic Hindley/Milner type system for imperative constructs. We further extend this system by parameterizing every object's type with one or more *region expressions* which statically model the memory locations reachable from that object. Region expressions denoted by *region variables* $(r_1, r_2, \ldots)$ within an object's type identify independent sets of mutable locations reachable from that object. A special *null* region expression (ϕ) identifies non-mutable locations. For example, the mutable histogram array `a` in Example 1 is typed as $array(r)\ int$, while the corresponding closed functional array is typed as $array(\phi)\ int$. The `close` operation converts mutable objects into functional objects by replacing the non-null regions in their type with the null region.

In order to ensure the dynamic safety of the `close` operation, the type system must guarantee that closed objects are never updated outside the scope of the `close` construct. This is enforced by ensuring that no write handles to the object being closed escape from the scope of the `close` construct either explicitly through the result or implicitly by storing the object into other external objects. This is where the region expressions prove useful. In our type system, the non-null regions within the type of an object provide a conservative static approximation for the set of mutable locations that are dynamically reachable from the given object (shown in [2]). Therefore, the type system is able to detect dynamically escaping write handles to an object by statically searching for its non-null region variables within the type of the result or the types of the objects present within the enclosing type environment. This intuition is captured in the following typing rule for the `close` operation on arrays.

$$\frac{E \vdash a :: array(r)\ \tau \qquad r \notin (FV(\tau) \cup FV(E))}{E \vdash \texttt{close}\ a :: array(\phi)\ \tau}$$

Here E stands for the type environment enclosing the expression a within the `close` construct. A region variable r in the type of a mutable array is allowed to be closed only if it is not visible both in the type of the array elements and within the enclosing type environment. Otherwise, the `close` operation is determined to be unsafe and a compile-time error is flagged.

Our main result is the type soundness theorem that ensures the consistency of the above typing rule (and similar rules for other data structures) with respect to the dynamic semantics of the language, relating mutable objects to their non-null region types and vice versa (proved in [2]). It immediately follows from this theorem that closed objects (possessing null region types) are never updated during the execution of a type-correct program and therefore can be treated like functional objects.

3 Applications

The `close` construct provides a general mechanism for defining arbitrary functional data abstractions within an imperative language. For instance, all the traditional functional objects such as arrays, lists, and tuples defined as primitives within a functional language, may now be defined in a system library or in the standard prelude using efficient imperative implementations. The users may also build arbitrary imperative data-structures and then convert them safely into functional objects using this construct.

Essentially, our type system performs aliasing and escape analysis on locally allocated data structures which is useful on its own for performing compiler optimizations in a conventional imperative language. Such information is traditionally obtained using dataflow analysis and abstract interpretation [1]. A type based approach such as ours may be more approximate than abstract interpretation, though it is much more efficient to compute. Furthermore, even though our type system uses closure types in order to deal with a polymorphic, higher-order language such as Id or ML, it naturally simplifies in the case of a first-order, monomorphic language such as C or Fortran—no closure types are needed for functions in the first-order case. This makes our approach highly attractive for practical use within compilers for conventional languages.

References

[1] A. Bloss. Update analysis and the efficient implementation of functional aggregates. In *Proc. Conf. on Functional Programming Languages and Computer Architecture*, September 1989.

[2] Shail Aditya Gupta. *Functional Encapsulation and Type Reconstruction in a Strongly-Typed, Polymorphic Language*. PhD thesis, MIT Laboratory for Computer Science, Cambridge, MA 02139, January 1995.

[3] Xavier Leroy. Polymorphic Typing of an Algorithmic Language. Rapports de Recherche 1778, INRIA, Rocquencourt, France, October 1992.