Dependent types for distributed arrays

Wouter Swierstra
joint work with Thorsten Altenkirch
Arrays

1 1 2 3 5
Distributed arrays
Golden Rule

Local access is quick;
remote access is slow.
Efficient code

- High-performance languages place restrictions on non-local array access.
- An operation that accidentally breaks the Golden Rule results in an exception.
- How can we avoid such exceptions?
Locality-aware IO

• Types with information about where the computation is executed.

• Think of \texttt{IO} \ a \ p as the type of a computation at location \ p returning a value of type \ a.
Consequences

read :: Int -> Array -> IO Int

- The location depends on the array and the index you are accessing.
Consequences

read :: Int -> Array -> IO Int

- The **type** of this function depends on the **value** of its arguments.
- Grothoff, Palsberg, and Saraswat have designed a type system for distributed arrays, based on a dependently typed lambda calculus.
\(\psi; \gamma; \Gamma, \text{here} : c : \text{int}\)

\(\psi, \gamma; \Gamma; \text{here} \vdash p : \text{pt}(p, \text{R})\) (where \(p \in \text{R}\))

(56)

(57)

\(\psi, \gamma; \Gamma; \text{here} \vdash R : \text{reg}(\text{R})\)

(58)

\(\psi, \gamma; \Gamma; \text{here} \vdash t : \psi(t)\)

(59)

\(\psi, \gamma; \Gamma; \text{here} \vdash p : \text{pl}(p)\)

(60)

\(\psi, \gamma; \Gamma; \text{here} \vdash \lambda x : \text{t}_x : \text{c}_x : \text{c}_x \rightarrow \text{t}_x\)

(61)

\(\psi, \gamma; \Gamma; \text{here} \vdash \lambda \alpha : \text{c}_x : \text{t}_x \rightarrow \text{t}_x\)

(62)

\(\psi, \gamma; \Gamma; \text{here} \vdash \text{constraint}(\kappa) : \Gamma; \text{unknown} \vdash \text{c}_x : \text{t}

(63)

\(\psi, \gamma; \Gamma; \text{here} \vdash \text{c}_x : \text{t}\)

(64)

\(\psi, \gamma; \Gamma; \text{here} \vdash \text{c}_x : \text{t}\)

(65)

\(\psi, \gamma; \Gamma; \text{here} \vdash \text{c}_x : \text{t}\)

(66)

\(\psi, \gamma; \Gamma; \text{here} \vdash \text{c}_x : \text{t}\)

(67)

\(\psi, \gamma; \Gamma; \text{here} \vdash \text{c}_x : \text{t}\)

(68)

\(\psi, \gamma; \Gamma; \text{here} \vdash \text{c}_x : \text{t}\)

(69)

\(\psi, \gamma; \Gamma; \text{here} \vdash \text{c}_x : \text{t}\)

(70)

\(\psi, \gamma; \Gamma; \text{here} \vdash \text{c}_x : \text{t}\)

(71)

\(\psi, \gamma; \Gamma; \text{here} \vdash \text{c}_x : \text{t}\)

(72)

\(\psi, \gamma; \Gamma; \text{here} \vdash \text{c}_x : \text{t}\)

(73)

\(\psi, \gamma; \Gamma; \text{here} \vdash \text{c}_x : \text{t}\)

(74)

\(\psi, \gamma; \Gamma; \text{here} \vdash \text{c}_x : \text{t}\)

(75)

\(\psi, \gamma; \Gamma; \text{here} \vdash \text{c}_x : \text{t}\)

(76)

\(\psi, \gamma; \Gamma; \text{here} \vdash \text{c}_x : \text{t}\)

(77)

\(\psi, \gamma; \Gamma; \text{here} \vdash \text{c}_x : \text{t}\)

(78)

\(\psi, \gamma; \Gamma; \text{here} \vdash \text{c}_x : \text{t}\)

(79)

\(\psi, \gamma; \Gamma; \text{here} \vdash \text{c}_x : \text{t}\)

(80)

\(\psi, \gamma; \Gamma; \text{here} \vdash \text{c}_x : \text{t}\)

(81)

\(\psi, \gamma; \Gamma; \text{here} \vdash \text{c}_x : \text{t}\)

(82)
A. Proof of Type Preservation

Here is the statement of Type Preservation (Theorem 1):

For a place \( P \), let \( Q \in \{P, unknown\} \). If \( \Psi; \varphi; \Gamma; Q \vdash e : t \), \( \models H : \Psi \), and \( P \vdash (H, e) \rightsquigarrow (H', e') \), then we have \( \Psi', t' \) such that \( \Psi \triangleleft \Psi', \Psi'; \varphi; \Gamma; Q \vdash e' : t' \), \( \models H' : \Psi' \), and \( \varphi \vdash t \equiv t' \).

Proof. We proceed by induction on the structure of the derivation of \( \Psi; \varphi; \Gamma; Q \vdash e : t \). There are now twenty-five subcases depending on which one of the type rules was the last one used in the derivation of \( \Psi; \varphi; \Gamma; Q \vdash e : t \).
Domain-specific embedded type systems

- Designing a type system is a lot of work!
- Can’t we use enforce these invariants using a general purpose dependently typed host language, such as Agda?
- Implementation and meta-theory for free!
Overview

- Embed the syntax and semantics of distributed array operations in a dependently typed language host language;
- statically enforce locality constraints;
- extract efficient code from our specification.
Terminology

• Any processor that executes code and stores data is referred to as a **place**.

• We will call an index in the array a **Point**

• We postulate a global **distribution**:

  \[
  \text{distr} : \text{Array} \rightarrow \text{Point} \rightarrow \text{Place}
  \]
Syntax - I

data IO (a : Set) : Place -> Set

  Return : a -> IO p a

  Read : (a : Array)

      -> (i : Point)

      -> (Int -> IO (distr a i) a)

      -> IO (distr a i) a
Syntax - II

data IO (a : Set) : Place -> Set
...

Write : (a : Array)
  -> (i : Point) -> Int
  -> IO (distr a i) a
  -> IO (distr a i) a
Syntax - III

data IO (a : Set) : Place -> Set
...

At : (q : Place)
  -> IO q ()
  -> IO p a
  -> IO p a
Auxiliary definitions

• We can define smart constructors:

\[
\text{read} : (a : \text{Array}) \\
\rightarrow (i : \text{Point}) \\
\rightarrow \text{IO} \ (\text{distr} \ a \ i) \ \text{Int}
\]

• and show that the IO data type is a monad.
Example: for

for : (Point -> IO p ())
    -> Array -> IO p ()

for io a = worker 0

  where worker i =

    if i == (size a) - 1 then
      then return ()
    else io i >> worker (i+1)
Example: dmap

dmap : (Int -> Int)
    -> Array -> IO p ()

dmap f a =
    for (\i -> at (distr a i)
        (read a i >> \x ->
            write a i (f x)))
Heap

data Heap = List (List Int)
type Array = Int
type Point = Int
run : (p : Place) -> IO a p
    -> Heap -> (a, Heap)
run p (Return x) h = (x,h)
run p (At q io1 io2) h =
    run p io2 (snd (run io1 h))
run : (p : Place) -> IO a p
    -> Heap -> (a, Heap)
run ? (Write a i x wr) h =
    run ? wr (updateHeap a i x h)
run ? (Read a i rd) h =
    run ? (rd (h !! a !! i))
Semantics - II

\[
\text{run} : (p : \text{Place}) \to \text{IO a p} \\
\rightarrow \text{Heap} \rightarrow (a, \text{Heap}) \\
\text{run ? (Write a i x wr) h =} \\
\text{run ? wr (updateHeap a i x h)}
\]

How do can we be sure we are not breaking the Golden Rule?
Why is this Haskell program well-typed?

data EQ a b where
  Refl :: EQ a a

coerce :: EQ a b -> a -> b
coerce Refl x = x
Learning from pattern matching

run .(distr a i)

(Write a i x wr) h

= run (distr a i)

wr (updateHeap a i x h)
Limitations

- This semantics is partial – that is, the lookup functions may fail...
- No allocation of new arrays
- Both of these points are solved in the paper.
Even more limitations

- No multi-dimensional arrays;
- Arrays may only store integers;
- A fixed, global distribution;
- Synchronous semantics;
- And we need a lot of these things to do interesting examples!
Conclusions

• Plenty of limitations – but the approach seems viable.

• *Domain-specific embedded type systems* are the way to go!