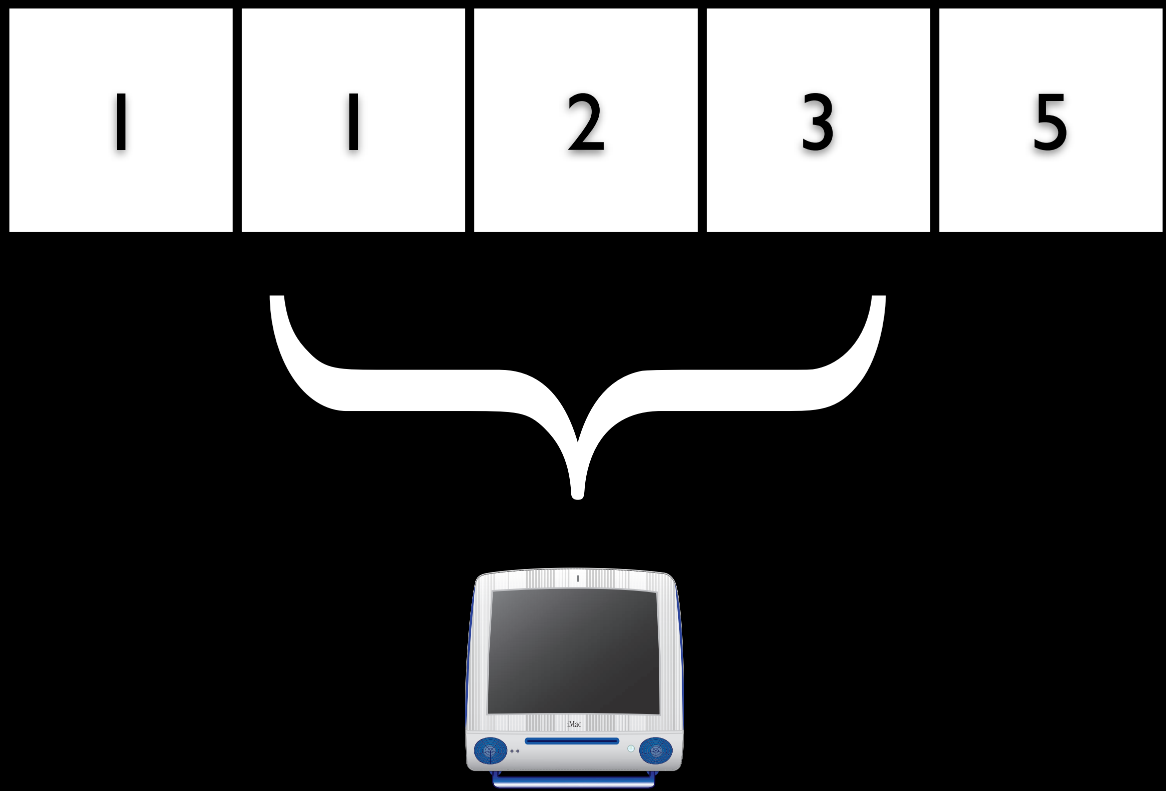
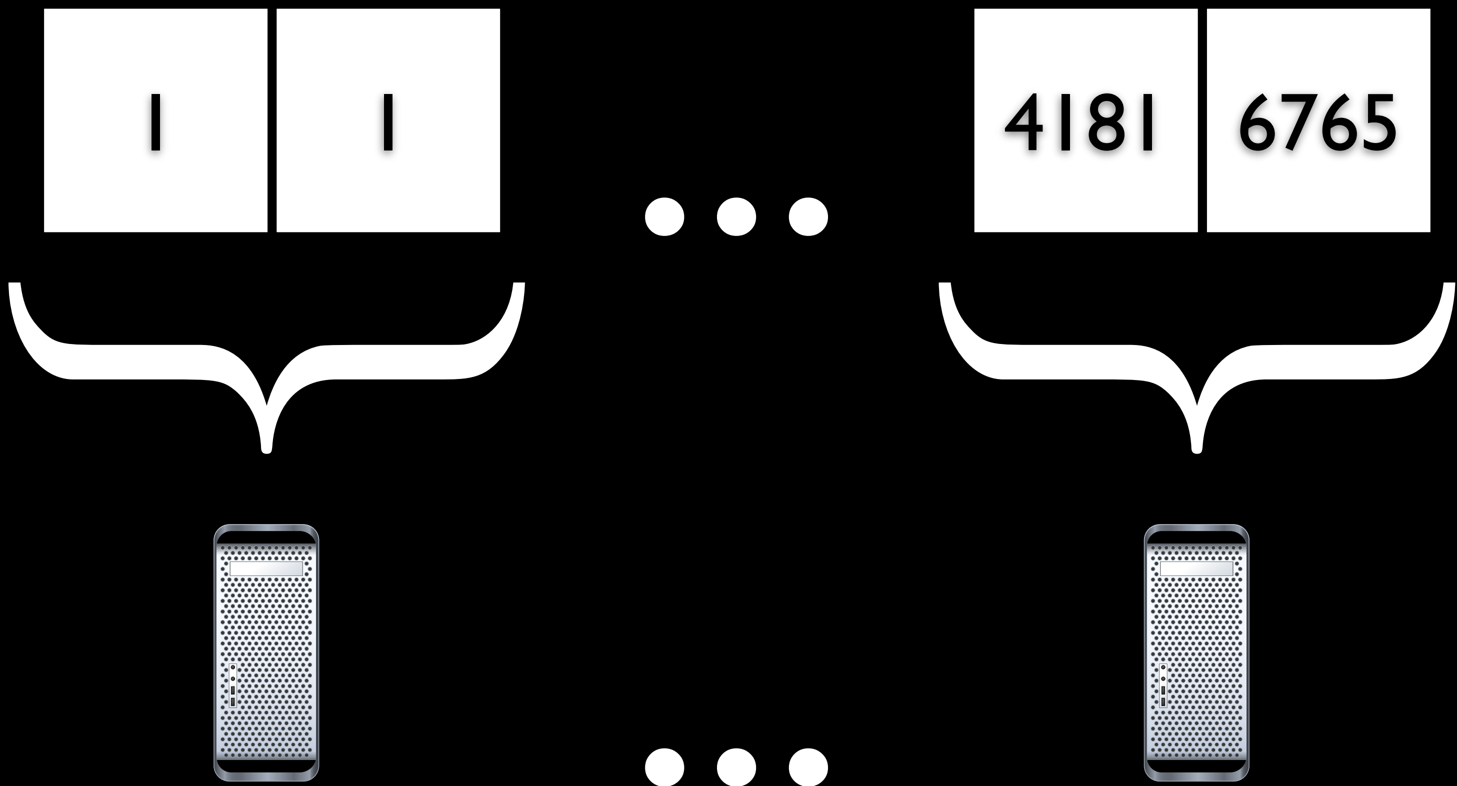
Dependent types for distributed arrays

Wouter Swierstra
joint work with Thorsten Altenkirch

Arrays



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 **`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.

$$\Psi;\varphi;\Gamma;\mathit{here} \vdash c :\ \texttt{int} \tag{56}$$

$$\Psi;\varphi;\Gamma;\mathit{here} \vdash p :\ \texttt{pt}\ (p, R) \quad (\text{where } p \in R) \tag{57}$$

$$\Psi;\varphi;\Gamma;\mathit{here} \vdash R :\ \texttt{reg}\ R \tag{58}$$

$$\Psi;\varphi;\Gamma;\mathit{here} \vdash l :\ \Psi(l) \tag{59}$$

$$\Psi;\varphi;\Gamma;\mathit{here} \vdash P :\ \texttt{pl}\ P \tag{60}$$

$$\frac{\Psi;\varphi;\Gamma[x : t_1];\mathit{unknown} \vdash e :\ t_2}{\Psi;\varphi;\Gamma;\mathit{here} \vdash \lambda x : t_1.e :\ t_1 \to t_2} \tag{61}$$

$$\frac{\Psi;\varphi \wedge \mathit{constraint}(k);\Gamma;\mathit{unknown} \vdash e :\ t}{\Psi;\varphi;\Gamma;\mathit{here} \vdash \texttt{lam}\ \alpha : k.e :\ \Pi\alpha : k.t} \tag{62}$$

$$\Psi;\varphi;\Gamma;\mathit{here} \vdash x :\ \Gamma(x) \tag{63}$$

$$\frac{\Psi;\varphi;\Gamma;\mathit{here} \vdash e_1 :\ t_1 \to t_2 \qquad \Psi;\varphi;\Gamma;\mathit{here} \vdash e_2 :\ t_1}{\Psi;\varphi;\Gamma;\mathit{here} \vdash e_1\ e_2 :\ t_2} \tag{64}$$

$$\frac{\Psi;\varphi;\Gamma;\mathit{here} \vdash e_1 :\ \Pi\alpha : k.t_1 \qquad \Psi;\varphi;\Gamma;\mathit{here} \vdash e_2 : t_2 \qquad \vdash t_2 :\ k \rhd W \qquad \varphi \models (\mathit{constraint}(k))[\alpha := W]}{\Psi;\varphi;\Gamma;\mathit{here} \vdash e_1\texttt{<}e_2\texttt{>}\ :\ t_1[\alpha := W]} \tag{65}$$

$$\frac{\Psi;\varphi;\Gamma[x : t_1];\mathit{here} \vdash e :\ t_2 \qquad \mathit{here} \neq \mathit{unknown}}{\Psi;\varphi;\Gamma;\mathit{here} \vdash \lambda^{\bullet} x : t_1.e :\ t_1 \to t_2} \tag{66}$$

$$\frac{\Psi;\varphi \wedge \mathit{constraint}(k);\Gamma;\mathit{here} \vdash e :\ t \qquad \mathit{here} \neq \mathit{unknown}}{\Psi;\varphi;\Gamma;\mathit{here} \vdash \mathrm{lam}^{\bullet}\alpha : k.e :\ \Pi\alpha : k.t} \tag{67}$$

$$\frac{\Psi;\varphi;\Gamma;\mathit{here} \vdash e :\ \texttt{reg}\ r}{\Psi;\varphi;\Gamma;\mathit{here} \vdash \texttt{new}\ t[e] :\ t[r]} \tag{68}$$

$$\frac{\begin{array}{c}\Psi;\varphi;\Gamma;\mathit{here} \vdash y_1 :\ t[r_1] \qquad \Psi;\varphi;\Gamma;\mathit{here} \vdash y_2 :\ \texttt{pt}\ (\sigma, r_2) \\ \varphi \models r_2 \subseteq_t r_1 \qquad \varphi \models \sigma \in_t r_2 \qquad \varphi \vdash \mathit{here} \equiv r_1[@_t(\sigma, r_2)]\end{array}}{\Psi;\varphi;\Gamma;\mathit{here} \vdash y_1[y_2] :\ t} \tag{69}$$

$$\frac{\begin{array}{c}\Psi;\varphi;\Gamma;\mathit{here} \vdash y_1 :\ t[r_1] \qquad \Psi;\varphi;\Gamma;\mathit{here} \vdash y_2 :\ \texttt{pt}\ (\sigma, r_2) \qquad \varphi \models r_2 \subseteq_t r_1 \\ \varphi \models \sigma \in_t r_2 \qquad \varphi \vdash \mathit{here} \equiv r_1[@_t(\sigma, r_2)] \qquad \Psi;\varphi;\Gamma;\mathit{here} \vdash e :\ t\end{array}}{\Psi;\varphi;\Gamma;\mathit{here} \vdash y_1[y_2] = e :\ t} \tag{70}$$

$$\frac{\Psi;\varphi;\Gamma;\mathit{here} \vdash e :\ t[r]}{\Psi;\varphi;\Gamma;\mathit{here} \vdash e.\texttt{reg}\ :\ \texttt{reg}\ r} \tag{71}$$

$$\frac{\Psi;\varphi;\Gamma;\mathit{here} \vdash y_1 :\ t[r_1] \qquad \Psi;\varphi;\Gamma;\mathit{here} \vdash y_2 :\ \texttt{pt}\ (\sigma, r_2) \qquad \varphi \models r_2 \subseteq_t r_1 \qquad \varphi \models \sigma \in_t r_2}{\Psi;\varphi;\Gamma;\mathit{here} \vdash y_1[@_s y_2]\ :\ \texttt{pl}\ r_1[@_t(\sigma, r_2)]} \tag{72}$$

$$\frac{\Psi;\varphi;\Gamma;\mathit{here} \vdash e_1 :\ \texttt{reg}\ r_1 \qquad \Psi;\varphi;\Gamma;\mathit{here} \vdash e_2 :\ \texttt{reg}\ r_2}{\Psi;\varphi;\Gamma;\mathit{here} \vdash e_1 \cup_s e_2 :\ \texttt{reg}\ r_1 \cup_t r_2} \tag{73}$$

$$\frac{\Psi;\varphi;\Gamma;\mathit{here} \vdash e_1 :\ \texttt{reg}\ r_1 \qquad \Psi;\varphi;\Gamma;\mathit{here} \vdash e_2 :\ \texttt{reg}\ r_2}{\Psi;\varphi;\Gamma;\mathit{here} \vdash e_1 \cap_s e_2 :\ \texttt{reg}\ r_1 \cap_t r_2} \tag{74}$$

$$\frac{\Psi;\varphi;\Gamma;\mathit{here} \vdash e :\ \texttt{reg}\ r}{\Psi;\varphi;\Gamma;\mathit{here} \vdash e +_s c :\ \texttt{reg}\ r +_t c} \tag{75}$$

$$\frac{\Psi;\varphi;\Gamma;\mathit{here} \vdash e :\ \texttt{pt}\ (\sigma, r)}{\Psi;\varphi;\Gamma;\mathit{here} \vdash e \mathbin{++_s} c :\ \texttt{pt}\ (\sigma \mathbin{++_t} c, r +_t c)} \tag{76}$$

$$\frac{\Psi;\varphi;\Gamma;\mathit{here} \vdash y_1 :\ \texttt{reg}\ r \qquad \Psi;\varphi;\Gamma;\mathit{here} \vdash y_2 :\ \texttt{pl}\ \pi}{\Psi;\varphi;\Gamma;\mathit{here} \vdash y_1 \mathbin{\%_s} y_2 :\ \texttt{reg}\ r \mathbin{\%_t} \pi} \tag{77}$$

$$\frac{\Psi;\varphi;\Gamma;\mathit{here} \vdash e_1 :\ \texttt{reg}\ r \qquad \Psi;\varphi \wedge (\alpha \in_t r);\Gamma[x :\ \texttt{pt}\ (\alpha, r)];\mathit{here} \vdash e_2 :\ \texttt{int} \qquad \mathit{here} \neq \mathit{unknown}}{\Psi;\varphi;\Gamma;\mathit{here} \vdash \texttt{for}\ (x\ \texttt{in}\ e_1)\{e_2\} :\ \texttt{int}} \quad (\text{where } \alpha \text{ is fresh}) \tag{78}$$

$$\frac{\Psi;\varphi;\Gamma[x : \texttt{pl}\ \alpha];\mathit{here} \vdash e :\ \texttt{int} \qquad \mathit{here} \neq \mathit{unknown}}{\Psi;\varphi;\Gamma;\mathit{here} \vdash \texttt{forallplaces}\ x\{e\} :\ \texttt{int}} \quad (\text{where } \alpha \text{ is fresh}) \tag{79}$$

$$\frac{\Psi;\varphi;\Gamma;\mathit{here} \vdash e_1 :\ t_1 \qquad \Psi;\varphi;\Gamma;\mathit{here} \vdash e_2 :\ t_2}{\Psi;\varphi;\Gamma;\mathit{here} \vdash e_1; e_2 :\ t_2} \tag{80}$$

$$\frac{\Psi;\varphi;\Gamma;\mathit{here} \vdash y :\ \texttt{pl}\ \pi \qquad \Psi;\varphi;\Gamma;\pi \vdash e :\ t}{\Psi;\varphi;\Gamma;\mathit{here} \vdash \texttt{at}\ (y)\ \{e\} :\ t} \tag{81}$$

$$\frac{\Psi;\varphi;\Gamma;\mathit{here} \vdash e :\ t \qquad \varphi \vdash t \equiv t'}{\Psi;\varphi;\Gamma;\mathit{here} \vdash e :\ t'} \tag{82}$$

A. Proof of Type Preservation

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

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

```
distr : Array -> Point -> 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:

```
read : (a : Array)
    -> (i : Point)
    -> IO (distr a i) 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
```

Semantics - 1

```
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))
```

Semantics - II

```
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

```
run : (p : Place) -> IO a p
   -> Heap -> (a, Heap)
run ? (Write a i x wr) h =
   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!