

# A Linear Type System for Multicore Programming

Rui Shi  
Yahoo Inc.

Hongwei Xi  
Boston University

Work partly funded by NSF grants no. CCR-0229480 and no. CCF-0702665

# *Resource Specification*

- Resource protection is crucial in concurrent programming.
- We need to properly specify resources.
- We need to strike a balance when specifying resources:
  - If specification is too weak, verification may underachieve.
  - If specification is too strong, verification may become too demanding.

# *Views for Classifying Capabilities (1)*

Given a type  $T$  and a memory location  $L$ ,  $T@L$  is a (primitive) view for describing that a value of the type  $T$  is currently stored at the location  $L$ .

Given types  $T_1$  and  $T_2$ , and a memory location  $L$ , the following view

$$T_1@L \otimes T_2@(L + 1)$$

describes that two values of the types  $T_1$  and  $T_2$  are stored at the locations  $L$  and  $L + 1$ , respectively.

# Views for Classifying Capabilities (2)

We can also declare recursive views (similar to datatypes in a functional language like ML). For instance, the following declaration introduces a view constructor *array\_v*:

```
dataview
array_v (type(*elt*), int(*sz*), addr(*loc*)) =
  | {a:type} {l:addr}
    array_v_nil (a, 0, l) of ()
  | {a:type} {n:nat} {l:addr}
    array_v_cons (a, n+1, l) of (a @ l, array_v (a, n, l+1))
// end of [dataview]
```

Given a type  $T$ , an integer  $I$  and a location  $L$ ,  $array\_v(T, I, L)$  is a view for an array containing elements of the type  $T$  that is of size  $I$  and located at  $L$ .

# Viewtypes

We can combine a view  $V$  and a type  $T$  to form a viewtype  $VT = V \otimes T$ .

- The following type can be assigned to a function  $ptr\_get_L$  that reads from a (fixed) address  $L$ :

$$(T@L, ptr(L)) \rightarrow (T@L) \otimes T$$

- The following type can be assigned to a function  $ptr\_set_L$  that writes to a (fixed) address  $L$ :

$$(T_1@L, ptr(L), T_2) \rightarrow (T_2@L) \otimes \mathbf{1}$$

# *Viewtypes for Read and Write*

- The type for the read function *ptr\_get* is

$$\forall \alpha \forall \lambda. (\alpha @ \lambda, ptr(\lambda)) \rightarrow (\alpha @ \lambda) \otimes \alpha$$

- The type for the write function *ptr\_set* is

$$\forall \alpha_1 \forall \alpha_2 \forall \lambda. (\alpha_1 @ \lambda, ptr(\lambda), \alpha_2) \rightarrow (\alpha_2 @ \lambda) \otimes \mathbf{1}$$

Note that we use  $\alpha$  for variables ranging over types and  $\lambda$  for variables ranging over locations.

# *Programming with Theorem-Proving*

ATS advocates a programming paradigm in which programming is combined with theorem programming.

- Proofs are manually constructed and then verified by the ATS typechecker automatically.
- Proofs are completely erased after typechecking.
- Proof erasure cannot alter the dynamic semantics of a program.

# *Array Subscripting (1)*

```
extern fun array_get
  {a:type} {n,i:nat | i < n} {l:addr}
  (pf: array_v (a, n, l) | p: ptr l, i: int i)
  : (array_v (a, n, l) | a)
```

```
extern fun array_set
  {a:type} {n,i:nat | i < n} {l:addr}
  (pf: array_v (a, n, l) | p: ptr l, i: int i, x: a)
  : (array_v (a, n, l) | unit)
```

# *Array Subscripting (2)*

```
fun array_getfst
  {a:type} {n:int | n > 0} {l:addr}
  (pf: array_v (a, n, l) | p: ptr l)
  : (array_v (a, n, l) | a) = let
  // pf1 : a @ l, pf2 : array (a, n-1, l+1)
  prval array_v_cons (pf1, pf2) = pf
  val (pf1' | x) = ptr_get (pf1 | p)
  prval pf' = array_v_cons (pf1', pf2)
in
  (pf' | x)
end // end of [array_getfst]

// after proof erasure
fun array_getfst (p) = ptr_get (p)
```

# *Array Subscripting (3)*

```
implement array_get (pf | p, i) =  
  if i = 0 then array_getfst (pf | p)  
  else let  
    prval array_v_cons (pf1, pf2) = pf  
    val (pf2' | x) = array_get (pf2 | p+1, i-1)  
    prval pf' = array_v_cons (pf1, pf2')  
  in  
    (pf' | x)  
  end // end of [if]  
// end of [array_get]
```

```
implement array_get (p, i) = // O(i)-time!  
  if i = 0 then array_getfst (p) else array_get (p+1, i-1)
```

# *Splitting/Unsplitting Array Views*

```
extern prfun array_v_split
  {a:type} {n,i:nat | i <= n} {l:addr}
  (pf: array_v (a, n, l) | i: int i)
  : (array_v (a, i, l), array_v (a, n-i, l+i))
```

```
extern prfun array_v_unsplit
  {a:type} {n1,n2:nat} {l:addr}
  (pf1: array_v (a, n1, l), array_v (a, n2, l+n1))
  : array_v (a, n1+n2, l)
```

# *Array Subscripting (4)*

```
implement array_get (pf | p, i) = let
  prval (pf1, pf2) = array_v_split (pf | i)
  val (pf2' | x) = array_getfst (pf2 | p + i)
  prval pf' = array_v_unsplit (pf1, pf2')
in
  (pf' | x)
end // end of [array_get]

// after proof erasure: clearly O(1)-time
implement array_get (p, i) =
  array_getfst (p+i) // = ptr_get (p+i)
```

# *Thread Creation*

A built-in function *thread\_create* is available for thread creation:

$$\mathit{thread\_create} \quad : \quad (\mathbf{1} \rightarrow_l \mathbf{1}) \rightarrow \mathbf{1}$$

Note that each thread created by *thread\_create* is immediately detached.

# *Linear Locks and Tickets for Uploading*

*uplock\_create* :  $\forall \hat{\alpha}. \mathbf{1} \rightarrow \mathbf{uplock0}(\hat{\alpha})$

*uplock\_destroy* :  $\forall \hat{\alpha}. \mathbf{uplock1}(\hat{\alpha}) \rightarrow \hat{\alpha}$

*upticket\_create* :  $\forall \hat{\alpha}. \mathbf{uplock0}(\hat{\alpha}) \rightarrow \mathbf{uplock1}(\hat{\alpha}) \otimes \mathbf{upticket}(\hat{\alpha})$

*upticket\_destroy* :  $\forall \hat{\alpha}. \mathbf{upticket}(\hat{\alpha}) \otimes \hat{\alpha} \rightarrow \mathbf{1}$

# *Joinable Threads*

Each thread created by *thread\_create* is detached. We can readily implement joinable threads on the top of detached ones.

$$\mathit{thread\_create\_join} \quad : \quad \forall \hat{\alpha}. (\mathbf{1} \rightarrow_l \hat{\alpha}) \rightarrow \mathbf{tid}(\hat{\alpha})$$
$$\mathit{thread\_join} \quad : \quad \forall \hat{\alpha}. \mathbf{tid}(\hat{\alpha}) \rightarrow \hat{\alpha}$$

# *Implementing Joinable Threads*

```
viewtypedef tid (a: viewtype) = uplock1 (a)

implement thread_create_join (f) = let
  val lock0 = uplock_create ()
  val (lock1, tick) = upticket_create (lock0)
  val () = thread_create
    (lam () => upticket_destroy (tick, f ()))
in
  lock1
end // end of [thread_create_join]

implement thread_join (lock1) = uplock_destroy (lock1)
```

# fib\_mt1

```
//  
// CUTOFF is some fixed integer constant >= 2  
// [fib] computes Fibonacci numbers sequentially  
//  
fun fib_mt1 (n: int): int =  
  if n < CUTOFF then fib (n)  
  else let  
    val tid1 =  
      thread_create_join (lam () => fib_mt1 (n-1))  
    val tid2 =  
      thread_create_join (lam () => fib_mt1 (n-2))  
    val res1 = thread_join (tid1)  
    val res2 = thread_join (tid2)  
  in  
    res1 + res2  
  end // end of [if]  
// end of [fib_mt1]
```

# *Scheduled Spawning and Synchronizing*

*spawn* :  $\forall \hat{\alpha}. (\mathbf{1} \rightarrow_l \hat{\alpha}) \rightarrow \mathbf{spawn}(\hat{\alpha})$

*sync* :  $\forall \hat{\alpha}. \mathbf{spawn}(\hat{\alpha}) \rightarrow \hat{\alpha}$

# fib\_mt2

```
fun fib_mt2 (n: int): int =
  if n < CUTOFF then fib (n)
  else let
    val tid1 = spawn (lam () => fib_mt2 (n-1))
    val tid2 = spawn (lam () => fib_mt2 (n-2))
    val res1 = sync (tid1)
    val res2 = sync (tid2)
  in
    res1 + res2
  end // end of [if]
// end of [fib_mt2]
```

# *Parallel Let-Binding*

The following syntax

```
let
  val par x1 = e1 and x2 = e2
in
  ...
end
```

translates into

```
let
  val tid1 = spawn (lam () => e1)
  and tid2 = spawn (lam () => e2)
  val x1 = sync (tid1) and x2 = sync (tid2)
in
  ...
end
```

# fib\_mt3

```
fun fib_mt3 (n: int): int =
  if n < CUTOFF then fib (n)
  else let
    // the keyword [par] indicates parallel let-binding
    val par res1 = fib_mt3 (n-1) and res2 = fib_mt3 (n-2)
  in
    res1 + res2
  end // end of [if]
// end of [fib_mt3]
```

# *Conclusion*

- Combining programming with theorem-proving (as is done in ATS) yields a promising approach to the construction of safer and more reliable programs.
- With dependent types and linear types, the programmer can accurately describe resources and then rely on the type system of ATS to preclude them being misused.
- We have formalized a type system to support safe concurrent programming and proven its type soundness.
- We have added support for concurrent programming in ATS and done some experimental benchmarking, providing a proof of concept.

# *More Information on ATS*

The programming language ATS is freely available to the public (GPL 3.0):

*<http://www.ats-lang.org>*

The source code for some programs used in benchmarking can be found at

*<http://www.ats-lang.org/EXAMPLE/MULTICORE/>*