Design, Implementierung und Evaluierung einer virtuellen Maschine für Oz

Ralf Scheidhauer

PS Lab, DFKI
May 18, 1999

Oz

- Developed at DFKI since 1991
- DFKI Oz 1.0 (1995), DFKI Oz 2.0 (1998)
- Mozart 1.0 (1999)
  - 180 000 lines of C++
  - 140 000 lines of Oz
  - 65 000 lines documentation
- Since 1996 collaboration with SICS and UCL
- Application strength system:
  multi agents (DFKI, SICS), computer-bus scheduling (Daimler), gate scheduling (Singapore), NL (SFB), comp. biology (LMU),...

Related Work

- LP, CLP [Warren 77], [Jaffer Lassez 86]
- Concurrency [Saraswat 93]
- AKL [Janson Haridi 90, Janson 94]
- FP [Appel 92]

3

Overview

- Language L
- Virtual machine
- Implementation
- Evaluation

4

The Language L

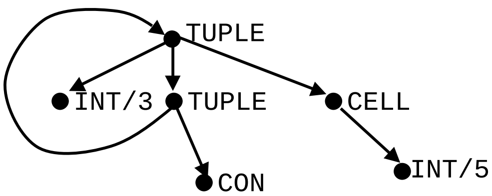
- Core language of Oz
- Presentation as extension of a sub language of SML
  - Logic variables
  - Threads
  - Synchronization
  - Dynamic type system
- Extensions via predefined functions

`lvar()`	*logic variable*
`unify(x,y)`	*unification*
`spawn(f)`	*thread creation*

5

Graph Model

- Integers
- Tuples
- Functions
- Cells (references)
- Constructors



- Strict evaluation of expressions

$e_0 \rightarrow e_1 \rightarrow \ldots$

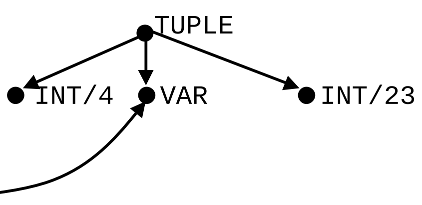
6

Why Logic Variables?

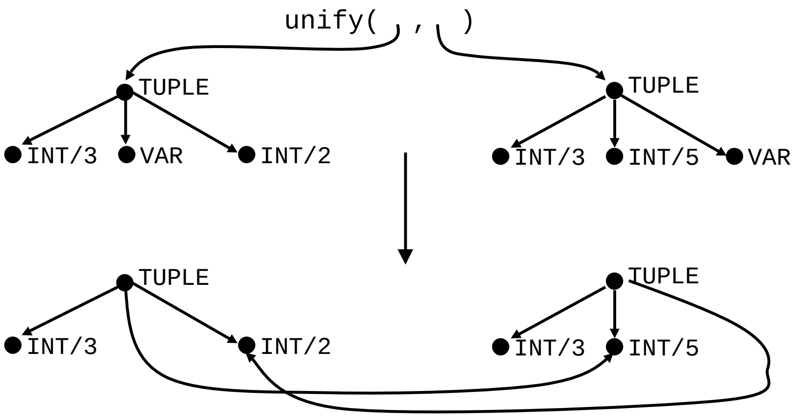
- Programming techniques: backpatching, difference lists, ...
- Cyclic data structures
- Tail recursive definition of many functions (`append`, `map`, ...)
- Synchronization of threads
- Search

Logic Variables: Creation and Representation

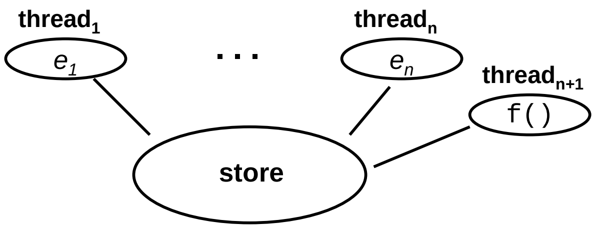
```
let val x = lvar()
in (4,x,23)
end
```



Logic Variables: Unification



Threads



- **Creation**

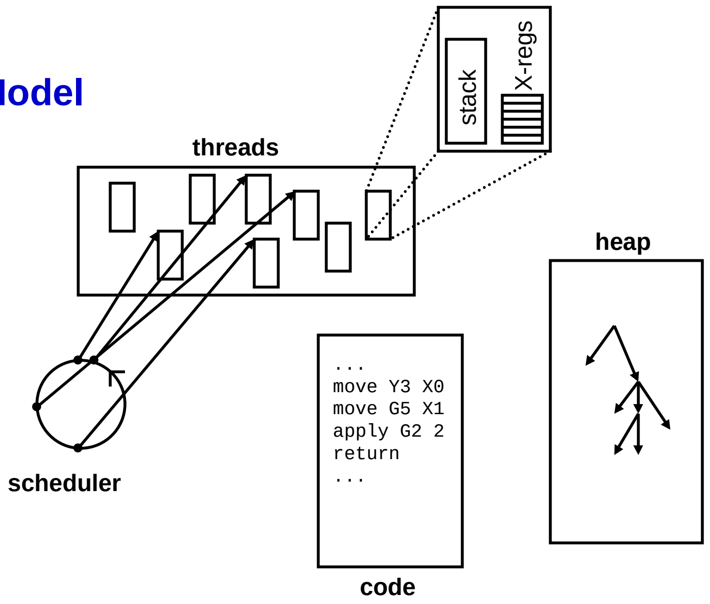
  `spawn(f)`

- **Synchronization: logic variables** `(x+y)`
- **Fairness**

Virtual Machine

11

Model



12

V-Addressing

- **Address toplevel variables via V-registers**
- **Loader builds data on the heap**

  $\Rightarrow$ **code contains direct references into heap**

- **Example**

```
fun f(l,u) = map(fn(x)=>h(x)+g(x)+u, l)
```

- `h` **and** `g` **in V-register** $\Rightarrow$ **reduced memory consumption**

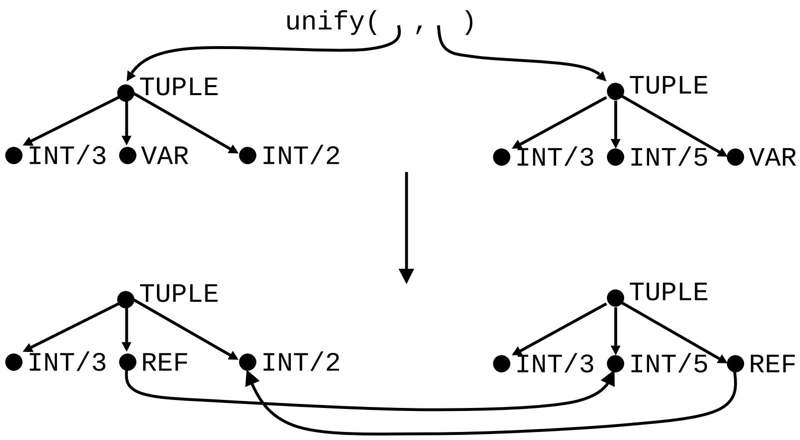
13

Dynamic Code Specialization

$$
\begin{tikzcd}
\texttt{apply } V_3 \texttt{ 2} \arrow[d] & \\
\texttt{specApply } V_3 \texttt{ 2} \arrow[r] & \texttt{fastApply } V_3
\end{tikzcd}
$$

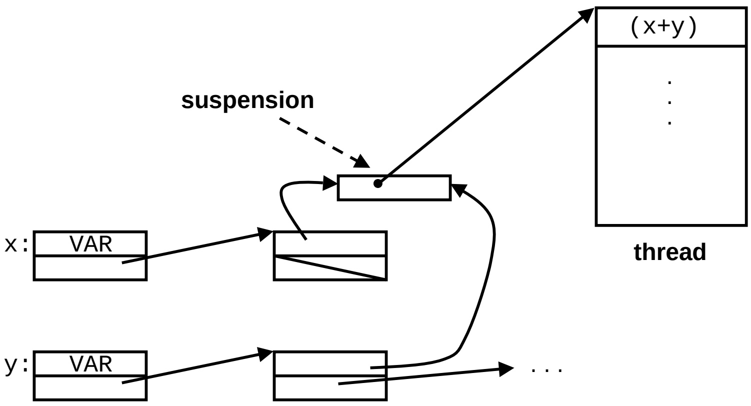
14

Unification in the Machine Model



15

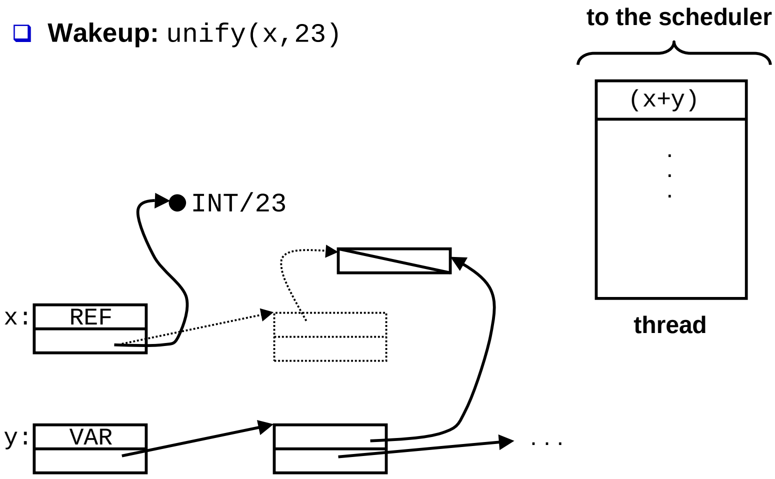
Synchronization = ***Suspension*** + Wakeup



16

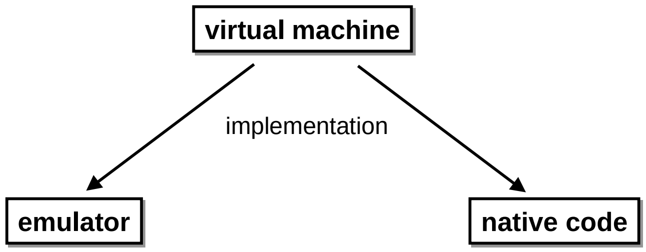
Synchronization = Suspension + ***Wakeup***

- **Wakeup:** `unify(x,23)`



Implementation

Emulator vs. Native Code



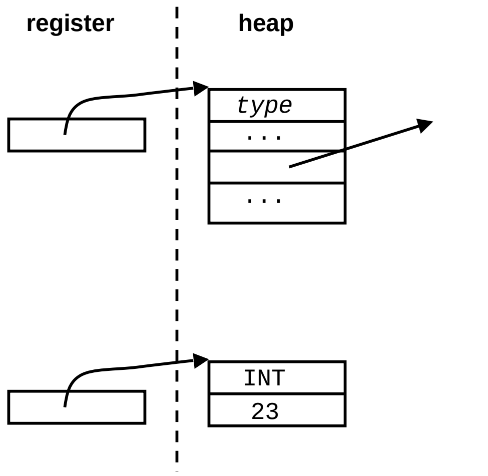
- portable
- flexible

- fast (?)

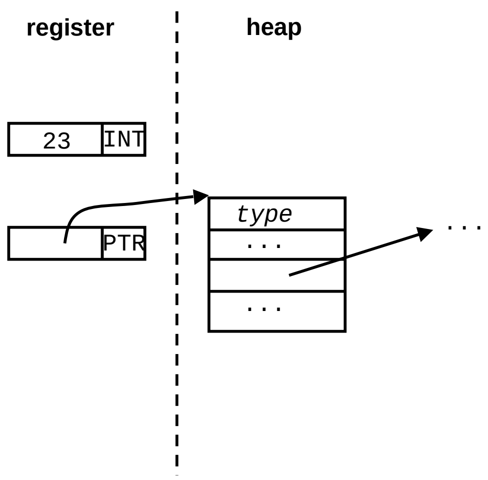
Threads

- X registers: once per machine, not per thread
  - Save live X registers upon preemption/suspension: pessimistic guess per function
  - Exact determination during GC by code interpretation

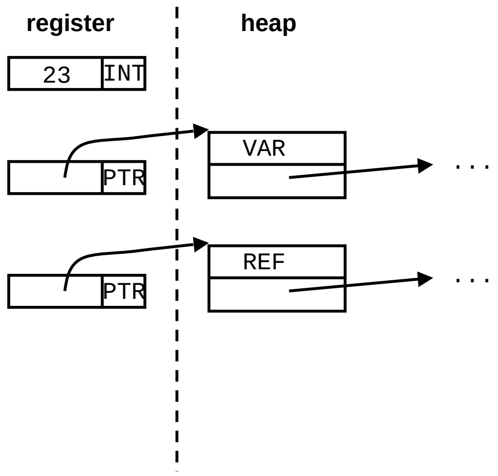
Representation of the Graph: Naiv



Representation of the Graph: Optimized

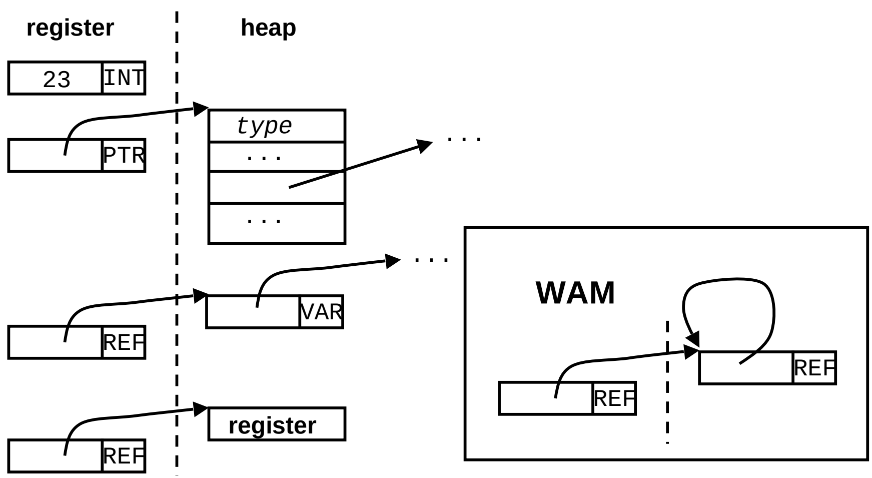


Representation of the Graph: Logic Variables



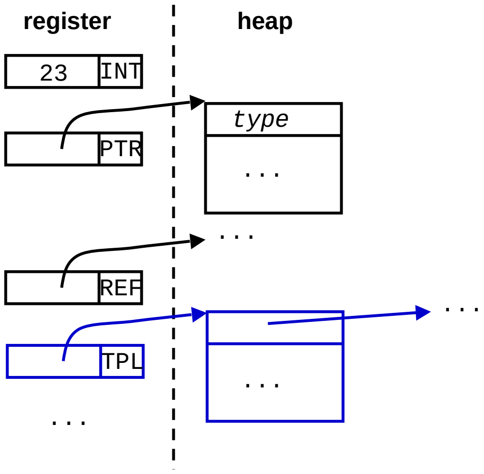
23

Logic Variables: Optimized



24

Moving More Tags



Evaluation

Comparison with Emulators

- Mozart is one of the fastest emulators
- Competitive with OCAML and Java
- Significantly faster than Moscow ML
- Twice as fast as Sicstus Prolog and Erlang

Comparison with Native Code Systems

- Few memory accesses (i.e. arithmetics)
  ⇒ Mozart is easily one order of magnitude slower
- Memory intensive (symbolic computation)
  - Difference only approx. factor 2-3
  - Mozart in single cases faster than native ML or C++

Threads

- Threads in Mozart are very light weight
- Leading position both for creation and communication
- Up to nearly 2 orders of magnitude faster than Java (creation)

Summary

- Extended sub language of SML by logic variables and threads
- Machine model
  - V - registers
  - Dynamic code specialization
  - Synchronization
- Implementation
  - Efficient implementation of threads
  - Tagging scheme
- Evaluation
  - Mozart is one of the fastest emulators
  - Compares well with native code systems on its target applications
  - Mozart has very light weight threads

Backup Slides for the Discussion

31

Logic Variables vs. Functions

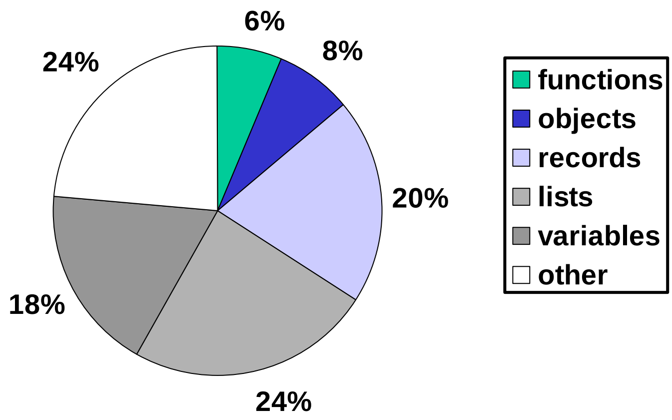
- Runtime

	fibonacci	takeushi
speedup	1.18	1.45

- Memory (large scale applications)
  - Use approx. 18 % of heap memory
  - Approx. twice as much as objects
  - Approx. as much as records

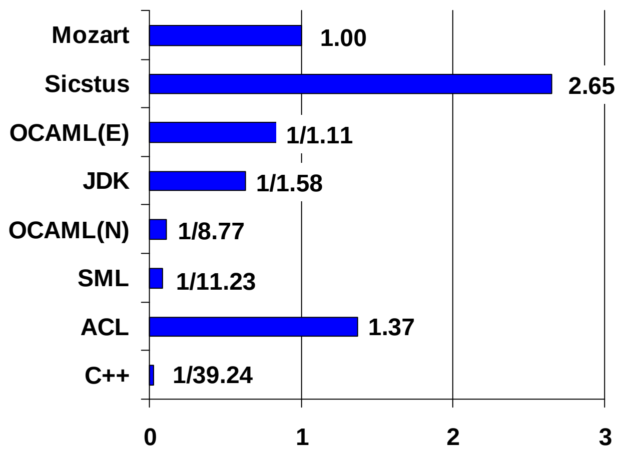
32

Memory Profile



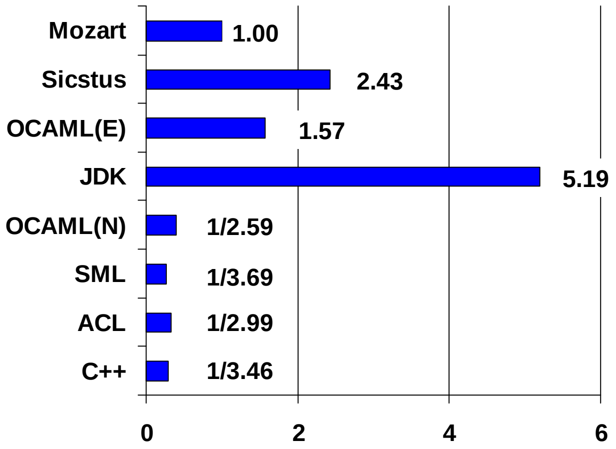
33

Mandelbrot (Floats)



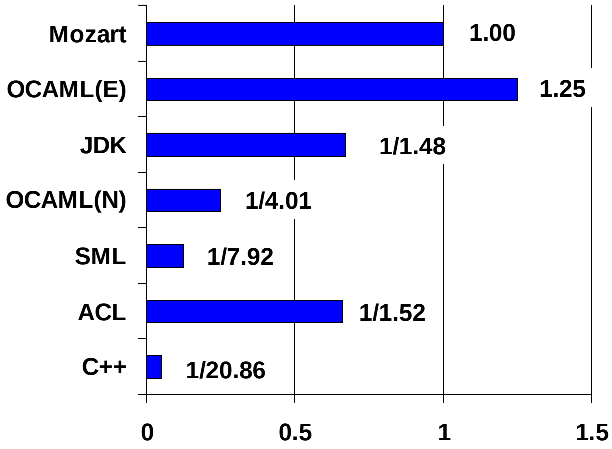
34

Quicksort with Lists



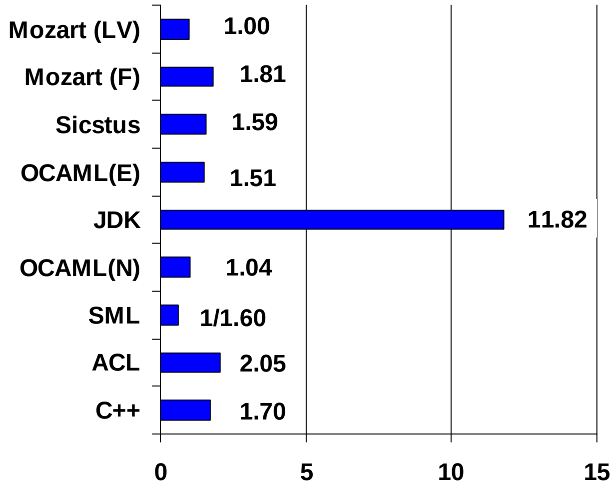
35

Quicksort with Arrays



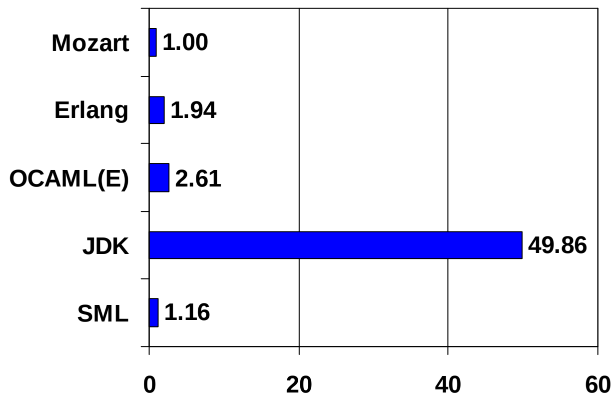
36

Naiv Reverse



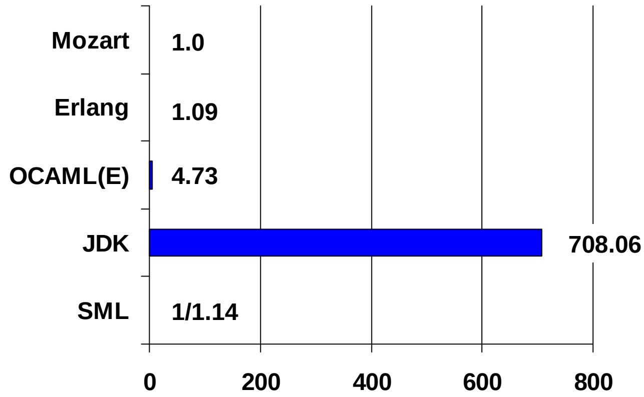
37

Threads: Creation



38

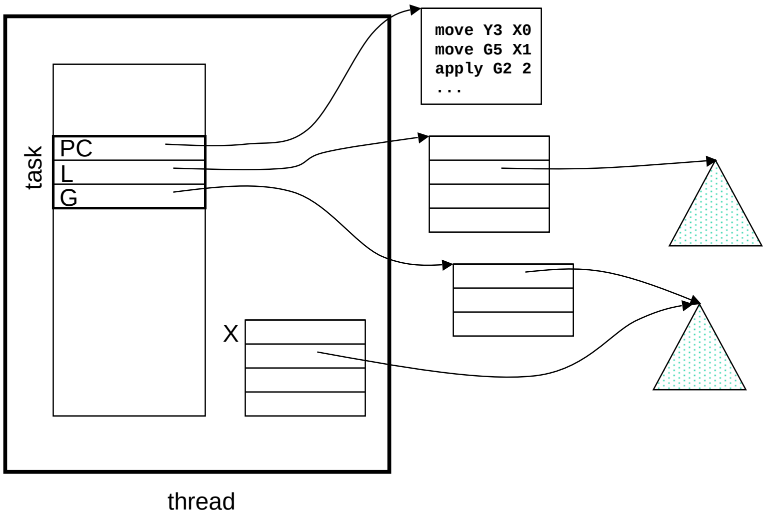
Threads: fib(20)



Tagging Scheme of Mozart

- **4 bit tag, but only 2 bit loss for address space (=1GB): align structures on word boundaries**
- **Lists, tuples: no need to unmask before type test**
- REF **- tag**
  - **no unmask before test necessary**
  - **no unmask before deref**

Threads



Emulators: Optimization Techniques

- Threaded code
- Instruction collapsing
- Register access
- Specialization
- Example

$$\left.\begin{array}{l}\texttt{move}\ Y_5\ X_3 \\ \texttt{move}\ Y_6\ X_1\end{array}\right\}$$ **34 → 11 (SPARC)**

Address Modes (Registers)

name	liveness	notation	usage
X	thread	X_i	temp. values, parameters
local	fct-body	L_i	local variables
global	function	G_i	free variables
virtual	program	V_i	constants

43

Threads

- Fairness: status-register

PRE	GC	IO	....

check on every function call (and return)

44

L

e	::=	*x*	*variable*
	|	*n*	*integer*
	|	$(e_1,\ldots,e_n)$	*tuple*
	|	**fn** $(x_1,\ldots,x_n)$ **=>** *e*	*function*
	|	$e_0(e_1,\ldots,e_n)$	*application*
	|	**let val** *x* = *e* **in** *e* **end**	*variable declaration*
	|	**let con** *x* **in** *e* **end**	*constructor declaration*
	|	**case** *e* **of** p_1 **=>** e_1 | ... | p_n**=>**e_n	*pattern matching*

Operators

lvar	**: () ->** α	*logic variable*
unify	**:** $\alpha \times \alpha$ **-> ()**	*unification*
spawn	**: ((() ->** α**) -> ()**	*thread creation*

45

add X_i X_k X_n

```
Tagged Xi = X[*(PC+1)];                                  2         0  (2)
DEREF(Xi);                                               2         0
if (isInt(getTag(Xi))) {                                 1+2       0
  Tagged Xk = X[*(PC+2)];                                2         2
  DEREF(Xk);                                             2         0
  if (isInt(getTag(Xk))) {                               1+2       0
    int aux = intValue(Xi)+intValue(Xk);                 1+1+1     2
    XPC(3) = oz_int(aux);   ovflw+shifttag+store         3+2+2     0  (2)
    DISPATCH(4);                                         3         3
  }                                                      ---------------
}                                                        27        7 (11)
                                        no derefs        23
                                        no type tests    17
                                        overflow          6
```

46

Java: JIT vs. Emulator

	speedup
quicksort (array)	18.8
fib (int)	14.2
fib (float)	4.9
queens	6.1
nrev	2.0
quicksort (list)	2.3
fib (thread)	1.1
mandelbrot	5.4
deriv (virtual)	1.9

47