Diploma Thesis
An Abstract Machine for a Concurrent (and Parallel) Constraint Functional Programming Language

Florian Lorenzen
florenz@cs.tu-berlin.de

Fachgebiet Übersetzerbau und Programmiersprachen
Institut für Softwaretechnik und Theoretische Informatik
Fakultät IV – Elektrotechnik und Informatik
Technische Universität Berlin

December 5, 2006
Outline

1. Motivation
2. Core language FATOM
   - Syntax
   - Semantics
   - Examples
3. Abstract Machine ATAF
   - Graph reduction
   - Structure of ATAF
   - Machine semantics and instruction set
   - Compiling FATOM to ATAF
4. Implementation
5. Demonstration
6. Conclusion
Motivation

Concurrent programming

- Many difficulties: synchronisation, communication, process management
- Usually handled at a detailed level, e.g. by: lock variables, semaphores, explicit send- and receive operations
- Can be treated more abstract by a *constraint based coordination language* to express process dependencies

Parallel programming

- Same problems as concurrent programming
- Requires control over process placement to gain performance from simultaneous execution
- Can be tackled by the same means as concurrent problems if process placement obeys certain conditions
Aim of the thesis

What is the goal?
- Programming language for concurrent and parallel programming
- Target architecture: multi-processor systems usually programmed by message passing

How is it achieved?
- Specification of a simple core language: FATOM
- Design of a suitable abstract machine as target architecture: ATAF
- Definition of a compiler to translate FATOM programs into ATAF machine instructions
- Prototypical implementation of the compiler (fc) and a machine interpreter (ataf)
1 Motivation

2 Core language FATOM
   - Syntax
   - Semantics
   - Examples

3 Abstract Machine ATAF
   - Graph reduction
   - Structure of ATAF
   - Machine semantics and instruction set
   - Compiling FATOM to ATAF

4 Implementation

5 Demonstration

6 Conclusion
FATOM: two languages

**Computation language**

Lazy functional language with
- supercombinators,
- `let-` and `case-`expressions,
- constructor types,
- built-in types: natural and floating point numbers, and
- built-in arithmetic, boolean, and comparison functions.

**Coordination language**

Constraint language with
- guarded alternatives,
- equality constraints, conjunctions, and
- built-in constraints: constructor matching and boundness checks.
A **FATOM** program is a sequence of supercombinator definitions:

\[
\text{def } v_1 \ v_{11} \ldots v_{1n_1} = e_1 \\
\vdots \\
\text{def } \underbrace{v_m}_{\text{name}} \underbrace{v_{m1} \ldots v_{mn_m}}_{\text{parameters}} = \underbrace{e_m}_{\text{body expression}}
\]

**Expressions**
- Functional expressions
- Constraint expressions
- Guarded expressions
Local definitions, application, and built-ins

- **let-expression**: \texttt{let } \(v_1 = e_1; \ldots; \ v_n = e_n\) \texttt{in } \(e\)
  - let-expressions are non-recursive.

- **Application**: \(e_1 \ldots e_n\) and \(e_1 \oplus e_2\) for infix operators

- **Built-in operators**: +, −, *, /, <, ≤, ==, ≠, ≥, >, ||, &&

- **Built-in functions**: not, neg, noPE, error, natToFloat, floatToNat

- **Constants**: natural and floating point numbers
Functional expressions (2)

Case analysis and constructor types

- Constructor: $\text{Pack}\{t, a\} \ e_1 \ldots e_a$ with tag $t$ and arity $a$
- case-expression: \text{case} \ e \ \text{of} \\
  \{t_1\} v_{11} \ldots v_{1a_1} \rightarrow e_1;
  \vdots
  \{t_n\} v_{n1} \ldots v_{na_n} \rightarrow e_n$

Type representation

- Constructor types are represented by single constructor $\text{Pack}\{t, a\}$
  \Rightarrow the language is not type safe
- Examples:
  -- data \text{List} \ \alpha :: \text{Nil} | \text{Cons} \ \alpha (\text{List} \ \alpha)
  \text{def} \text{Nil} = \text{Pack}\{1, 0\}
  \text{def} \text{Cons} \ x \ xs = \text{Pack}\{2, 2\} \ x \ xs
  -- data \text{Bool} :: \text{True} | \text{False}
  \text{def} \text{True} = \text{Pack}\{1, 0\}
  \text{def} \text{False} = \text{Pack}\{2, 0\}
Constraint/guarded expressions

Constraint expressions

- Equality constraint: \( v = := e \)
- Conjunction (concurrent composition): \( e_1 \land \ldots \land e_n \)
- Introduction of fresh constraint variables: **with** \( v_1 \ldots v_n \) **in** \( e \)

Guarded expressions

- Guarded alternatives: \( g_1 \Rightarrow e_1 \)
  \[ \begin{array}{c}
  \vdots \\
  g_n \Rightarrow e_n
  \end{array} \]
- Guards: Conjunction of primitive constraint: \( g_i = c_{i1} \land \ldots \land c_{in_i} \)
- Primitive constraints: \( pack \ v \ t \ a \) (Constructor matching)
  \[ bound \ v \] (Check for boundness)
  \[ unbound \ v \] (Check for unboundness)
**Operational semantics**

**Configuration**

A *configuration* is a tuple

\[
CONF := PROC \times STORE
\]

with 

- **PROC** multiset of processes (expressions) and
- **STORE** set of equality constraints \( v := w \), 
  \( w \in VAL \cup Var \cup \{\diamond\} \).

**Transition relation**

The operational semantics of a *FATOM* program \( P \) is defined by a transition relation

\[
\rhd_P \subseteq CONF \times CONF
\]

with infix notation \( \gamma_1 \rhd_P \gamma_2 :\iff \langle \gamma_1, \gamma_2 \rangle \in \rhd_P \).
An Abstract Machine for a Concurrent (and Parallel) Constraint Functional Programming Language

Transition relation

- Defined by 11 rules along the syntax
- Implements normal order reduction for the functional part
- Helper functions:  
  - \( B \): evaluate built-in primitives
  - \( T \): add a constraint to the store (tell)
  - \( A \): retrieve information from the store (ask)
Transition rule $tellP$

**Strict context**

- Forces evaluation of $e$ in $v := e$
- Strictifies constructors to speed up concurrency

\[\begin{align*}
    e & \in p \\
    e &= r =:= b \\
    \langle\{b\}, s\rangle &\xrightarrow{P}^{+} \langle\{w\}, s\rangle \\
    w &= Pack\{t, a\} \, e_1 \ldots e_a \\
    s' &= s \cup \{v_1 =:= \diamond, \ldots, v_n =:= \diamond\} \\
    v_i &\notin vars(s), \ i = 1, \ldots, n \\
    T[r =:= Pack\{t, a\} v_1 \ldots v_n ]s' &= s'' \neq fail \\
    (tellP) \quad p' &= p \setminus \{e\} \uplus \{v_1 =:= e_1, \ldots, v_n =:= e_n\} \\
    \langle p, s \rangle &\xrightarrow{P} \langle p', s'' \rangle
\end{align*}\]
farm: a parallel map

- Applies $f$ to all elements of list $l$ and binds result to $r$
- Example:

  $\langle\text{farm } \text{succ} \ [1, 2] \ r, \{r =:= \Diamond\}\rangle$

Definition of farm

```haskell
-- farm :: (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta] \rightarrow C
def farm f l r =
case l of
  \{1\} \rightarrow r =:= \text{Nil};
  \{2\} x xs \rightarrow \textbf{with} \ rs \ \textbf{in} \ r =:= \text{Cons} \ (f \ x) \ rs \ \& \ \text{farm} \ f \ xs \ rs
```
**farm**: a parallel map

- Applies $f$ to all elements of list $l$ and binds result to $r$
- Example:

  \[
  \langle\text{farm succ} \ [1, 2] \ r, \{r =:= \Diamond\}\rangle
  \]

  \[
  \triangleright^+ \langle r =:= \text{succ} \ 1 : rs, \text{farm succ} \ [2] \ rs, \{rs =:= \Diamond, \ldots\}\rangle
  \]

**Definition of farm**

\[
\text{-- farm :: } (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta] \rightarrow C
\]

\[
\text{def farm f l r =}
\]

\[
\text{case l of}
\]

\[
\{1\} \rightarrow r =:= \text{Nil};
\]

\[
\{2\} \ x \ xs \rightarrow \textbf{with} \ rs \ \textbf{in} \ r =:= \text{Cons} (f \ x) \ rs \ & \text{farm f} \ xs \ rs
\]
**farm**: a parallel *map*

- Applies \( f \) to all elements of list \( l \) and binds result to \( r \)

**Example:**

\[
\langle \text{farm succ } [1, 2] \ r, \{r \ =:= \ \Diamond}\rangle
\]

\[
\triangleright^+ \langle r \ =:= \ \text{succ } 1 : \ rs, \ \text{farm succ } [2] \ rs, \{rs \ =:= \ \Diamond, \ldots\}\rangle
\]

\[
\triangleright^+ \langle r \ =:= \ \text{succ } 1 : \ rs, \ rs \ =:= \ \text{succ } 2 : \ rs', \{rs' \ =:= \ [] , \ldots\}\rangle
\]

**Definition of farm**

\[
\text{-- farm} :: (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta] \rightarrow C
\]

\[
\text{def farm } f \ l \ r =
\]

\[
\text{case } l \ of
\]

\[
\{1\} \rightarrow r =:= Nil;
\]

\[
\{2\} \ x \ xs \rightarrow \textbf{with } rs \ \textbf{in } \ r =:= \text{Cons } (f \ x) \ rs \ & \ \text{farm } f \ xs \ rs
\]
**farm: a parallel map**

- Applies \( f \) to all elements of list \( l \) and binds result to \( r \)

**Example:**

\[
\langle farm\ succ\ [1,\ 2]\ r,\ \{ r =:= \diamond \} \rangle \\
\triangleright^+ \langle r =:= succ\ 1:\ rs,\ farm\ succ\ [2]\ rs,\ \{ rs =:= \diamond,\ \ldots \} \rangle \\
\triangleright^+ \langle r =:= succ\ 1:\ rs,\ rs =:= succ\ 2:\ rs',\ \{ rs' =:= [],\ \ldots \} \rangle \\
\triangleright^+ \langle \emptyset,\ \{ r =:= 2:\ rs,\ rs =:= 3:\ rs',\ rs' =:= [] \} \rangle
\]

**Definition of farm**

\[
←← farm :: (α → β) → [α] → [β] → C
\]

```haskell
def farm f l r =
    case l of
        \{1\} → r =:= Nil;
        \{2\} x xs → with rs in r =:= Cons (f x) rs \& farm f xs rs
```
Non-deterministic merge

Definition of $merge$

\[ def \ merge \ l1 \ l2 \ r = \text{pack} l1 \ 1 \ 0 \ \& \]
\[ pack \ l2 \ 1 \ 0 \Rightarrow r =: Nil \]
\[ | \ pack \ l1 \ 2 \ 2 \Rightarrow \text{with} \ rs \ l1' \]
\[ \text{in} \ r =: \text{Cons} \ (hd \ l1) \ rs \ \& \]
\[ l1' =: tl \ l1 \ \& \]
\[ merge \ l1' \ l2 \ rs \]
\[ | \ pack \ l2 \ 2 \ 2 \Rightarrow \text{with} \ rs \ l2' \]
\[ \text{in} \ r =: \text{Cons} \ (hd \ l2) \ rs \ \& \]
\[ l2' =: tl \ l2 \ \& \]
\[ merge \ l1 \ l2' \ rs \]
We are here

1. Motivation

2. Core language FATOM
   - Syntax
   - Semantics
   - Examples

3. Abstract Machine ATAF
   - Graph reduction
   - Structure of ATAF
   - Machine semantics and instruction set
   - Compiling FATOM to ATAF

4. Implementation

5. Demonstration

6. Conclusion
Lazy evaluation

Ingredients

- Normal order reduction: arguments to functions are passed unevaluated:
  \[(\lambda x. 2 \times x) (1 + 2) \leadsto 2 \times (1 + 2)\]
  not \[(\lambda x. 2 \times x) (1 + 2) \leadsto (\lambda x. 2 \times x) 3\]

- Updates: identical expressions are not evaluated twice
  Example: \texttt{let } \texttt{x = e in f x x}
  \[\Rightarrow x \text{ is evaluated only once}\]

Implementation

- Graph reduction is an efficient technique
- Expressions are represented as graphs in a heap
- Graphs are transformed until normal form is reached
Graph reduction (1)

- Finding the next redex by unwinding the spine
- Instantiating the supercombinator
- Updating

**Example program**

\[
\begin{align*}
\text{compose } f & \quad g \quad x = f (g \ x) \\
\text{twice } f & \quad x = \text{compose } f \quad f \quad x \\
\text{double } x & = x + x
\end{align*}
\]

**twice double 2**

Stack

<table>
<thead>
<tr>
<th>Stack</th>
</tr>
</thead>
</table>

Heap

- a
  - a
    - twice
    - double
  - 2
Graph reduction (1)

- Finding the next redex by unwinding the spine
- Instantiating the supercombinator
- Updating

Example program

\[
\begin{align*}
\text{compose } f \; g \; x &= f \; (g \; x) \\
\text{twice } f \; x &= \text{compose } f \; f \; x \\
\text{double } x &= x + x
\end{align*}
\]

*twice double 2*
Graph reduction (1)

- Finding the next redex by unwinding the spine
- Instantiating the supercombinator
- Updating

Example program

\[
\begin{align*}
\text{compose } f \ g \ x &= f \ (g \ x) \\
\text{twice } f \ x &= \text{compose } f \ f \ x \\
\text{double } x &= x + x
\end{align*}
\]

An Abstract Machine for a Concurrent (and Parallel) Constraint Functional Programming Language
Graph reduction (1)

- Finding the next redex by unwinding the spine
- Instantiating the supercombinator
- Updating

Example program

\[
\begin{align*}
\text{compose } f \ g \ x &= f \ (g \ x) \\
\text{twice } f \ x &= \text{compose } f \ f \ x \\
\text{double } x &= x + x
\end{align*}
\]

\textit{twice double 2}
Graph reduction (1)

- Finding the next redex by unwinding the spine
- Instantiating the supercombinator
- Updating

Example program

\[
\begin{align*}
\text{compose } f \ g \ x &= f (g \ x) \\
\text{twice } f \ x &= \text{compose } f \ f \ x \\
\text{double } x &= x + x
\end{align*}
\]

\[
twice \ \text{double} \ 2
\]
Graph reduction (1)

- Finding the next redex by unwinding the spine
- Instantiating the supercombinator
- Updating

Example program

- \( \text{compose } f \ g \ x = f \ (g \ x) \)
- \( \text{twice } f \ x = \text{compose } f \ f \ x \)
- \( \text{double } x = x + x \)
Graph reduction (1)

- Finding the next redex by unwinding the spine
- Instantiating the supercombinator
- Updating

Example program

\[
\begin{align*}
\text{compose } f \ g \ x & = f (g \ x) \\
\text{twice } f \ x & = \text{compose } f \ f \ x \\
\text{double } x & = x + x
\end{align*}
\]
Graph reduction (1)

- Finding the next redex by unwinding the spine
- Instantiating the supercombinator
- Updating

**Example program**

\[
\begin{align*}
\text{compose } f & \quad g & \quad x & = & \quad f \ (g \ x) \\
\text{twice } f & \quad x & = & \quad \text{compose } f \quad f \quad x \\
\text{double } x & = & \quad x + & x
\end{align*}
\]

**twice double 2**

Diagram showing the evaluation process of the example function compositions.
Graph reduction (2)

### Built-in functions, let- and case-expressions

- Built-in functions (e.g. arithmetics) require arguments evaluated: reduction of the subgraph
- let-expressions are textual descriptions of a graph
- For a case analysis, the expression in question is reduced and the corresponding branch is instantiated

### G-machine

- Unwinding is done by the UNWIND instruction which pushes the pointers and jumps into the supercombinator code
- Supercombinator instantiation is done by a sequence of instructions that builds the graph
- Case analysis is implemented by a conditional jump
- Strict context is imposed by the EVAL instruction
Machine layout

General layout

| STORE | STACK₁ | … | STACKₙ | HEAP₁ | … | HEAPₙ | CODE |

Parallel machine instances

| STORE | CODE |

| process₀₀ | process₀₁ | … | processₙ₀ | processₙ₁ | … | process_noPE₋₁₀ | process_noPE₋₁₁ | … | process_noPE₋₁ₙ | process_noPE₋₁_noPE₋₁ |

CODE shared instruction segment
STORE shared constraint store with synchronisation
HEAP, STACK private segments for graph reduction
Machine layout

General layout

<table>
<thead>
<tr>
<th></th>
<th>(\text{STORE})</th>
<th>(\text{STACK}_1)</th>
<th>(\cdots)</th>
<th>(\text{STACK}_n)</th>
<th>(\text{HEAP}_1)</th>
<th>(\text{HEAP}_n)</th>
<th>(\text{CODE})</th>
</tr>
</thead>
</table>

Parallel machine instances

<table>
<thead>
<tr>
<th>(\text{STORE})</th>
<th>(\text{CODE})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{process}_0)</td>
<td>(\cdots)</td>
</tr>
<tr>
<td>(\text{process}_0)</td>
<td>(\cdots)</td>
</tr>
<tr>
<td>(\text{process}_0)</td>
<td>(\cdots)</td>
</tr>
</tbody>
</table>

- CODE: shared instruction segment
- STORE: shared constraint store with synchronisation
- HEAP, STACK: private segments for graph reduction
Machine instance

Run\_i \quad \text{Currently running process}

Ready\_i \quad \text{Processes waiting for execution}

IC\_i \quad \text{Instruction counter for scheduling}

**Pointers**

Instruction pointer \quad IP\_j

Stack pointer \quad SP\_j

Heap pointer \quad HP\_j

Base registers \quad IB\_j, SB\_j, HB\_j
**Store**

- Access to the store is mutually exclusive
- The process granted access to the store runs uninterruptible

### Components

- **Registers:** TP, TB for memory management
- **Blocked** holds processes waiting for store access
- **Suspend[i]** hold processes waiting for a variable change

![Diagram](image.png)
Process management

Process creation and placement

- A new process is created on instance with fewest processes (simple load balancing)
- Process creation:
  1. Allocation of new heap and stack
  2. Initialising environment: copy required values from parent
  3. Inserting process into Ready queue and setting IP register ⇒ special kind of jump
- A process dies as soon as its stack is empty

Scheduling

- Simple round robin scheduler on each instance
- Processes may set themselves as *uninterruptible*
State transition system

Machine state

A machine state is a tuple

\[
\text{STATE} := \mathcal{M}^{\text{noPE}} \times \mathcal{I} \times \mathcal{S}
\]

with

- \(\mathcal{M}\) machine instance,
- \(\mathcal{I}\) code segment, and
- \(\mathcal{S}\) constraint store.

A machine instance \(\mathcal{M}\) consists of

- \(\text{IC}\) instruction counter,
- \(\text{Run}\) currently running process,
- \(\text{Ready}\) waiting processes, and
- \(\mathcal{P}\) set of stacks, heaps, pointer registers.

Operational semantics

The operational semantics of ATAF is specified by a state transition relation

\[
\iff \subseteq \text{STATE} \times \text{STATE}.
\]
Instruction set

Transition rules and pseudo-code

- Definition of machine instructions by rules for $\equiv$ is tedious
- Instructions are therefore specified by pseudo-code that is connected to transition relation

Important machine instructions

- LOCK, UNLOCK: Mutual exclusion for the store
- SPAWN: Create process
- SUSPEND: Wait for variables
- ISPACK, ISBOUND, . . .: Primitive constraints (conditional jumps)
- TELL: Augment a binding to the store
- CASEJUMP, SPLIT, JUMP: Control flow
- PUSHFUN, PACK, . . .: Build graph nodes
- UNWIND, EVAL: Unwind the spine and force evaluation
- ADD, SUB, MUL, . . .: Built-in primitives
Instruction \textbf{TELL}

\textbf{TELL } n \equiv \\
\langle \text{PTR} \mid a \rangle := \text{nth } n; \langle \text{PTR} \mid ptr \rangle := \text{top} \\
w := \text{lookup } ptr \\
\text{IF } \text{STORE}[a] = \langle \text{NIL} \rangle \text{ THEN} \\
\text{IF } w = \langle \text{NAT} \mid \cdot \rangle \lor w = \langle \text{FLT} \mid \cdot \rangle \text{ THEN} \\
\text{STORE}[a] := w \\
\text{ELSE} \\
\langle \text{PCK} \mid t \ as \rangle := w \\
vs := \text{allocateVars } \mid as \\
\text{STORE}[a] := \langle \text{PCK} \mid t \ vs \rangle \\
\text{FOREACH } (a', v') \in (as, vs) \text{ DO} \\
\quad \text{push } \langle \text{PTR} \mid a' \rangle \\
\quad \text{push } \langle \text{PTR} \mid v' \rangle \\
\text{END FOREACH} \\
\text{END IF} \\
\text{wakeup } a \\
\text{ELSE IF} \\
\text{STORE}[a] \neq w \text{ THEN fail} \\
\text{END IF} \\
\text{inc } IP

Stack layout

The pointers $v_i, e_i$ are used to evaluate sub-terms according to rule ($tellP$).
Instruction TELL

TELL \( n \equiv \)
\[
\langle \text{PTR} \mid a \rangle := \text{nth } n; \langle \text{PTR} \mid ptr \rangle := \text{top}
\]
\( w := \text{lookup } ptr \)

IF \( \text{STORE}[a] = \langle \text{NIL} \rangle \) THEN
\( \text{IF } w = \langle \text{NAT} \mid \cdot \rangle \lor w = \langle \text{FLT} \mid \cdot \rangle \) THEN
\( \text{STORE}[a] := w \)
ELSE
\( \langle \text{PCK} \mid t \ as \rangle := w \)
\( vs := \text{allocateVars } as \)
\( \text{STORE}[a] := \langle \text{PCK} \mid t \ vs \rangle \)
FOREACH \((a', v')\) IN \((as, vs)\) DO
\( \text{push } \langle \text{PTR} \mid a' \rangle \)
\( \text{push } \langle \text{PTR} \mid v' \rangle \)
END FOREACH
END IF

wakeup \( a \)
ELSE IF
\( \text{STORE}[a] \neq w \) THEN fail
END IF

inc IP

Stack layout

The pointers \( v_i, e_i \) are used to evaluate sub-terms according to rule (tellP).
**Instruction TELL**

**TELL** \( n \equiv \)

\[
\langle \text{PTR} \mid a \rangle := \text{nth} \ n; \langle \text{PTR} \mid ptr \rangle := \text{top}
\]

\( w := \text{lookup} \ ptr \)

IF \( \text{STORE}[a] = \langle \text{NIL} \rangle \) THEN

IF \( w = \langle \text{NAT} \mid \cdot \rangle \ \vee \ w = \langle \text{FLT} \mid \cdot \rangle \) THEN

\( \text{STORE}[a] := w \)

ELSE

\( \langle \text{PCK} \mid t \ as \rangle := w \)

\( vs := \text{allocateVars} \ | as| \)

\( \text{STORE}[a] := \langle \text{PCK} \mid t \ vs \rangle \)

FOREACH \((a', v')\) IN \((as, vs)\) DO

push \( \langle \text{PTR} \mid a' \rangle \)

push \( \langle \text{PTR} \mid v' \rangle \)

END FOREACH

END IF

wakeup \( a \)

ELSE IF

\( \text{STORE}[a] \neq w \) THEN fail

END IF

inc IP

---

**Stack layout**

The pointers \( v_i, e_i \) are used to evaluate sub-terms according to rule \((tellP)\).
Compilation function

Schemes $C$, $F^S$, and $F^L$

- Compilation function defined along the syntax of FATOM
- Translates FATOM expressions into list of ATAF machine instructions
- Constraint and functional abstractions have different stack handling
- For functional expressions, strict and lazy context is distinguished
  $\Rightarrow C$, $F^S$, $F^L$

Example: with-expression

$$C[\text{with } v_1 \ldots v_n \text{ in } e] \Gamma =$$

$$[\text{ALLOC } n] \ +\ C[e] \Gamma^{+n}[v_1 \mapsto 0, \ldots, v_n \mapsto n - 1] + [\text{POP } n]$$

$\Gamma$ maps variable names to their current stack frame number.
**main** supercombinator

- Evaluation begins with **main** combinator
- **main** is a constraint abstraction with argument for the result
- Example: \( \text{main } r = r =: = 5 + 4 \)

**Initialisation code**

Graph to be build and reduced:

```
main

\( a \)
```

Corresponding code:

```
0  ALLOC 1
1  PUSHFUN a
2  MKAP
3  UNWIND
```

\( a \) is the address of **main**.
We are here

1. Motivation
2. Core language FATOM
   - Syntax
   - Semantics
   - Examples
3. Abstract Machine ATAF
   - Graph reduction
   - Structure of ATAF
   - Machine semantics and instruction set
   - Compiling FATOM to ATAF
4. Implementation
5. Demonstration
6. Conclusion
Implementation (1)

- Two parts: compiler `fc` and machine interpreter `ataf`
- Written entirely in **HASKELL**
- Parallel communication via **MPI** library

**Machine interpreter**

Multi-threaded implementation:

The store and scheduler are operated by one process
- `serv` handles request like `wakeup` or `process creation`
- `comp` interprets the machine instructions
- `↔` denotes **MPI** communication
Monadic style programming

- Machine interpreter is a monad transformer embedding a state monad in the IO monad
- Implementation of instructions look like their specification:

  $$\text{iPushfun } a \text{ ar } =$$
  $$\text{do } a' \leftarrow \text{allocate}$$
  $$\text{(HEAP, a')} \%= \text{FUN } a \text{ ar}$$
  $$\text{push } (\text{PTR } a')$$
  $$\text{inc } IP$$

  $$\text{PUSHFUN } a \text{ ar } \equiv$$
  $$a' := \text{allocate}$$
  $$\text{HEAP}[a'] := \langle \text{FUN } | \text{ a ar} \rangle$$
  $$\text{push } \langle \text{PTR } | \text{ a'} \rangle$$
  $$\text{inc } IP$$

Garbage collection

- For heaps, a two-space copying collector is used
- The store is tidied by a mark-and-scan collector
  - On a full store, all processes send entry pointers to store thread
  - With mark-and-scan collection, entry pointers remain untouched
    $$\Rightarrow$$ easier to implement
Performance

Test case

- Measurements took place on TUSCI cluster of KBS group
- Test program: calculation of square roots of 1, . . . , 1000
- `pfarm` coordination: like `farm` but only one worker process per processing element

Result

![Graph showing performance measurement](image)

Test program

```python
def main r = pfarm sqrt (enumFromTo 1 1000) r
```
Motivation

Core language FATOM
- Syntax
- Semantics
- Examples

Abstract Machine ATAF
- Graph reduction
- Structure of ATAF
- Machine semantics and instruction set
- Compiling FATOM to ATAF

Implementation

Demonstration

Conclusion
Demonstration

$ ./fc --no-spawn -p -o PfarmSqrt.ataf PfarmSqrt.fatom \ Sqrt.fatom PreludeFun.fatom PreludeCoord.fatom

$ mpirun -np 2 -machinefile tusci ./ataf -S 8192 \ --fast-prop PfarmSqrt.ataf

** Scheduler statistics:
   # total processes: 14
   # processes on 000: 7
   # processes on 001: 7

** Store statistics:
   # GCs: 0, # store words: 256

** RESULT <...>
We are here

Motivation

Core language **FATOM**
- Syntax
- Semantics
- Examples

Abstract Machine **ATAF**
- Graph reduction
- Structure of **ATAF**
- Machine semantics and instruction set
- Compiling **FATOM** to **ATAF**

Implementation

Demonstration

Conclusion
Conclusion

We have seen...

- Concurrent constraint functional programming is possible with only a few language constructs and suitable to also express parallel programs.
- The abstract machine presented is a fair compromise between ease of compilation and ease of implementation.

Open questions and problems

- How to translate a high-level language into FATOM?
- How to cope with a varying number of processing elements?
- How to overcome the store bottleneck?
- How to implement ATAF natively, including proper storage management?
Conclusion

We have seen...

- Concurrent constraint functional programming is possible with only a few language constructs and suitable to also express parallel programs.
- The abstract machine presented is a fair compromise between ease of compilation and ease of implementation.

Open questions and problems

- How to translate a high-level language into FATOM?
- How to cope with a varying number of processing elements?
- How to overcome the store bottleneck?
- How to implement ATAF natively, including proper storage management?

Thank You!
References


P. Hofstedt and A. Wolf. 
Einführung in die Constraint-Programmierung.


D. R. Lester and S. L. Peyton Jones. 
Implementing Functional Languages: a tutorial.

H. R. Nielson and F. Nielson. 
Semantics With Applications.

P. Pepper. 
Programmiersprachen und -systeme.