Optimizing Compilation Techniques for Logic Programming

Jose F. Morales$^{1,2}$

$^1$School of Computer Science, Technical University of Madrid, Spain
$^2$IMDEA Software Institute, Spain

Prometidos Summer School
Facultad de Informática, UCM – September 19-21, 2011
Introduction and Motivation
Introduction

- **Prolog**: Turing complete, general purpose language, based on SLD resolution.
- Program: sets of Horn clauses (subset of formulas of first order predicate logic). Data: Herbrand terms.

**Example**

\[ P = \{ p(0), (p(s(X)) \leftarrow p(X)) \} \]

- **Execution**: given a query \( Q \) (e.g. \( p(s(s(0))) \)), disprove the negated query and the program \( (\bigwedge_{C \in P} C \land \neg Q) \).
- SLD is refutation complete. However, Prolog typically implements a depth-first strategy. It is potentially incomplete if the search space contains infinite branches.
Introduction

- Despite this, Prolog has some nice properties:
  - Very efficient and cheap search (backtracking).
  - Predictable execution order resembling procedural languages (left-to-right, recursive).
  - Extensions to interact with the resolution strategy and unification: cut, negation as failure, attributed variables, updatable program database, backtrackable/nonbacktrackable global variables ⇒ Prolog as basis of logic engines.

- Examples: lazy functional, SMT/SAT solver (Howe & King), CLP, CHR, ontologies and semantic web, etc.

- More advanced extensions (sharing considerable part of the machinery): tabling, paralellism.

- Many uses in the academic and industry (e.g. Prolog success story with IBM Watson’s win at Jeopardy!).

- In this talk: (some) techniques to achieve mature and fast implementations. Not in this talk: source-to-source transformations.
Roadmap

- Generic Implementation of Abstract Machines
- Defining and Transforming Instructions
- Fine-Grained Instructions and Native Code Generation
- A Case Study
- Open Research Problems
- Conclusions
Generic Implementation of Abstract Machines
Anatomy of A.M. Systems

At different language levels:

- $\mathcal{L}_P$: source program (functions, predicates, objects, constraints, etc.)
- $\mathcal{L}_A$: intermediate code
- $\mathcal{L}_B$: bytecode (a stream of bytes) $\leftarrow$ interpreted code
- $\mathcal{L}_C$: low level C/assembler $\leftarrow$ the interpreter
Introducing A.M. Specification ($M$)

- **EMUComp approach** uses a specification common to the interpreter and compiler ($M$, abstract machine definition):
  - Read by the compiler (to generate bytecode)
  - Used by the A.M. compiler (emucomp) to generate $int$

- Starting point for further transformations.
Example $\mathcal{L}_A$ Interpreter

Towards a Generic Interpreter

- Fetch-execute cycle (as tail-recursive procedure)
- We keep it simple in this part for presentation purposes.
- Good starting point: concise translation of $\mathcal{L}_A$ instructions into $\mathcal{L}_C$
- An example of a simple $\mathcal{L}_A$-level interpreter

\[
\text{emu}_A(p, \text{program}) \equiv \\
\langle \text{ins}, p' \rangle = \text{fetch}_A(p, \text{program}) \\
\text{case } \text{ins} \text{ of} \\
\langle \text{move}, [r(i), r(j)] \rangle : \text{reg}[j] := \text{reg}[i]; \ p'' := p' \\
\langle \text{call}, [\text{label}(l)] \rangle : \text{push}(p'); \ p'' := l \\
\langle \text{ret}, [] \rangle : p'' := \text{pop}() \\
\text{otherwise} : \text{error} \\
\text{emu}_A(p'', \text{program})
\]
Deconstruction of $L_A$ Instructions
Towards a Generic Interpreter

- Original definition:
  \[ \langle \text{move}, [r(i), r(j)] \rangle : \ reg[j] := reg[i]; p'' := p' \]

- Extract the arguments $M_{\text{args}}$:
  \[
  r(i) \rightarrow reg[i]
  \]

- And obtain $M_{\text{def}}$: ($cont$ abstracts continuations)
  \[ \langle \text{move}, [a, b] \rangle \rightarrow [b := a; \ cont(next)] \]

- Define operand types, to obtain the signature (or format) for each instruction:
  \[ M_{\text{absexp}}(r(\_)) = r \]

- Make the instruction set explicit (useful later):
  \[ \langle \text{move}, [r, r] \rangle \in M_{\text{ins}} \]
The Generic Interpreter

- Using previous definitions we can write a generic interpreter:

\[
\text{int}_1(p, \text{program}, M) \equiv \\
\langle \langle \text{name}, \text{args} \rangle, p' \rangle = \text{fetch}_A(p, \text{program}) \\
\text{if } \neg \text{valid}_A(\langle \text{name}, \text{args} \rangle, M_{\text{ins}}, M_{\text{absexp}}) \text{ then } \text{error} \\
\text{cont} = \lambda a \rightarrow [p'' := a] \\
\llbracket M_{\text{def}}(p', \text{cont}, \text{name}, M_{\text{args}}(\text{args})) \rrbracket \\
\text{int}_1(p'', \text{program}, M)
\]

- Definition of an abstract machine \( M = (M_{\text{def}}, M_{\text{arg}}, M_{\text{ins}}, M_{\text{absexp}}) \)

- \( M_{\text{def}} \) (relates \( \mathcal{L}_A \) instructions and their \( \mathcal{L}_C \) code)

- \( M_{\text{args}} \) (relates \( \mathcal{L}_A \) args and their \( \mathcal{L}_C \) representation)

- \( M_{\text{ins}}, M_{\text{absexp}} \) validate that the instruction is correct

- \( \llbracket \ldots \rrbracket \) executes an expression representing \( \mathcal{L}_C \) code

- Common structure shared by a whole family of interpreters!

- But \( \mathcal{L}_A \) not intended to be executed — lower-level language needed
Example: A.M. Specification

\[ M_{\text{def}}(\text{next}, \text{cont}, \text{name}, \text{args}) = \]
\[
\text{case } \langle \text{name}, \text{args} \rangle \text{ of}
\]
\[
\langle \text{move}, [a, b] \rangle \rightarrow [a := b; \ \text{cont(next)}]
\]
\[
\langle \text{call}, [a] \rangle \rightarrow [\text{push(next)}; \ \text{cont(a)}]
\]
\[
\langle \text{ret}, [] \rangle \rightarrow [\text{cont(pop())}]
\]
\[
\langle \text{halt}, [] \rangle \rightarrow [\text{return}]
\]

\[ M_{\text{ins}} = \]
\[
\{ \langle \text{move}, [r, r] \rangle, \langle \text{call}, [\text{label}] \rangle, \langle \text{ret}, [] \rangle, \langle \text{halt}, [] \rangle \}
\]

\[ M_{\text{arg}}(\text{arg}) = \]
\[
\text{case } \text{arg} \text{ of}
\]
\[
r(i) \rightarrow \text{reg}[i]
\]
\[
\text{label}(l) \rightarrow l
\]

\[ M_{\text{absexp}}(\text{arg}) = \]
\[
\text{case } \text{arg} \text{ of}
\]
\[
r(_) \rightarrow r
\]
\[
\text{label}(_) \rightarrow \text{label}
\]
\[
\text{otherwise } \rightarrow \bot
\]
From Symbolic Code to Bytecode and Back

- $\mathcal{L}_A$: move r(0) r(2); move r(1) r(0); move r(2) r(1); ret
- $\mathcal{L}_B$: \[\begin{array}{cccccc}
0 & 0 & 2 & 0 & 1 & 0 \\
0 & 2 & 1 & 2 & & \\
\end{array}\]
- $\mathcal{L}_B$ is much more convenient than $\mathcal{L}_A$ (for $\mathcal{L}_C$)
Pass Separation

- Augmented $M = (M_{\text{def}}, M_{\text{arg}}, M_{\text{ins}'}, M_{\text{absexp}}, M_{\text{enc}}, M_{\text{dec}})$
- To divide concerns between compiler and high-level interpreter:
  - **Encode** to a lower level in the compiler
  - (Conceptually) **decode** in the interpreter
  - With the hope that a suitably defined **decode** and interpreter can be fused and $L_A$ disappears from the runtime section
A Generic Bytecode Interpreter

- Use `decode` to build an interpreter for bytecode:

  $\text{int}_2(p, prg, M) \equiv \text{int}_1(p, decode(prg), M)$

  More directly:

  $\text{int}_2(p, prg, M) \equiv$

  $\text{opcode} = prg[p]$

  $\langle \text{name}, \text{format} \rangle = M_{\text{ins}}'(\text{opcode})$

  $\langle \text{args}, p' \rangle = \text{decode}_{\text{ins}}(\text{format}, [p], [prg], M)$

  $\text{cont} = \lambda a \rightarrow [\text{int}_2(a, prg, M); \text{return}]$

  $[M_{\text{def}}(p', \text{cont}, \text{name}, M_{\text{args}}(\text{args})); \text{cont}(p')]$

- `int_2` is not meant to be directly executed (big overhead!)

- Efficiency comes from partial evaluation of `int_2` w.r.t. $M$.

- **Hint**: $M$ static and finite:
  - `opcode` can be enumerated to generate a `case`
  - Partial evalution feasible: $\text{int}_3 \equiv \llbracket spec \rrbracket(\text{int}_2, M)$
  - Removes overhead (i.e. no metaprogramming, etc.)
Example: Generated Emulator

- Emulator in plain $\mathcal{L}_C$:

\[
\text{emu}_B(p, \text{program}) \equiv \\
\text{case program}[p] \text{ of} \\
0 : \text{reg}[\text{program}[p + 1]] := \text{reg}[\text{program}[p + 2]]; \\
\text{emu}_B(p + 3, \text{program}); \text{return} \\
1 : \text{push}(p + 2); \\
\text{emu}_B(\text{program}[p + 1], \text{program}); \text{return} \\
2 : \text{emu}_B(\text{pop}(), \text{program}); \text{return}
\]

- Tail recursion can be easily transformed into a loop
Emulator Compiler

- An optimized interpreter can be obtained by:
  - Partially evaluating the interpreter w.r.t. $M$
  - Applying an emulator compiler to $M$

Using second Futamura projection

\[
emucomp : M \rightarrow code_C
emucomp = [spec](spec, \text{int}_2)
\]

- We need a self-applicable partial evaluator for $\mathcal{L}_C$
- Quite feasible if restricting to a reasonable subset of $\mathcal{L}_C$
- Otherwise, the structure of the emulator compiler is simple
  - This allows writing the emulator compiler by hand
  - We do not need partial evaluator (but have correctness properties and a methodology)
Emulator Compiler (definition)

\[ \text{emucomp}(M) = \]
\[ [ \text{emu}_B(p, prg) \equiv \]
\[ \text{case get_opcode}(p, prg) \text{ of} \]
\[ \text{opcode}_1 : \text{inscomp}(\text{opcode}_1, M) \]
\[ \ldots \]
\[ \text{opcode}_n : \text{inscomp}(\text{opcode}_n, M) ] \]
\[ \text{where } \text{opcode}_i \in \text{domain}(M_{ins'}) \]

\[ \text{inscomp}(\text{opcode}, M) = \]
\[ [M_{\text{def}}(p', \text{cont}, \text{name}, M_{\text{args}}(\text{args})); \text{cont}(p')] \]
\[ \text{where} \]
\[ \langle \text{name}, \text{format} \rangle = M_{\text{ins'}}(\text{opcode}) \]
\[ \langle \text{args}, p' \rangle = \text{decode}_{\text{ins}}(\text{format}, [p], [prg], M) \]
\[ \text{cont} = \lambda a \rightarrow [\text{emu}_B(a, prg); \text{return}] \]
An Application: Reducing Real Life Emulators

Evaluation of Code Size

- Let $M'$ be a version of $M$ containing only instructions in $G$
- A reduced emulator $\text{emu}_G$ is automatically obtained as $\text{emu}_G = \text{emucomp}(M')$
- Size of bytecode program & emulator savings

<table>
<thead>
<tr>
<th>Emulator savings</th>
<th>Bytecode size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Full program+libs</td>
</tr>
<tr>
<td>hw</td>
<td>28%</td>
</tr>
<tr>
<td>boyer</td>
<td>26%</td>
</tr>
<tr>
<td>fib</td>
<td>28%</td>
</tr>
<tr>
<td>knights</td>
<td>27%</td>
</tr>
<tr>
<td>poly</td>
<td>26%</td>
</tr>
<tr>
<td>query</td>
<td>28%</td>
</tr>
<tr>
<td>stream</td>
<td>26%</td>
</tr>
<tr>
<td>tak</td>
<td>28%</td>
</tr>
<tr>
<td><strong>Range</strong></td>
<td><strong>≈27%</strong></td>
</tr>
</tbody>
</table>
Related Work

- Biernacki Dariusz, Olivier Danvy. **From Interpreter to Logic Engine by Defunctionalization**, LOPSTR (2003):
  - Continuations \( \xrightarrow{\text{Defunctionalization}} \) Constructors
  - Application of conts \( \xrightarrow{\text{apply\_cont}} \)

  Tail-recursive, continuation-passing (high order), Prolog interpreter transformed into a transition system. A.M. considered as models of computation (not high performance) and transformations as changes of representation (not optimizations).

  - Ad-hoc, mechanized description of instruction formats with C types, stack/register based.
  - Our solution lies between: explicit stack management, but (more) formal, abstract approach.
Defining and Transforming Instructions
Defining and Transforming Instructions

- Fast emulators (w/o. JIT): large instruction set (super-instructions, specialized cases, frequent built-ins, etc.).
- However, most of the complex instruction sets can be generated from a reduced subset by applying common program manipulation techniques.

P analysis, specialization, transformations, verification...

- Handmade (easier?)
- Emulator generation
- Automatic!
- Program compilation

E (partially written in P) → program execution → LB emu → prg_B → prg_P
Language for Instructions

- Let introduce here a simplified dialect of Prolog (denoted as ImProlog). Two new **native** features (still expressible within standard Prolog):

  Native types and operations on them: opaque types used to reflect in ImProlog machine-level data representations and data types required by the A.M. (integers, floats, tagged words, etc.).

  **Mutable variables (mutvars):** associations between a mutable identifier and some arbitrary term

- **Aims:** express A.M. naturally, making automatic program manipulation possible, without sacrificing efficient mapping to $\mathcal{L}_C$
Language for Instructions

Operations on mutvars:

**Assignment:** \( MutVar \leftarrow Value \) associates \( Value \) with the identifier \( MutVar \).

**Access:** \( @MutVar \) stands for a function which returns the value previously associated to \( MutVar \).

**Typed assignment:** Type-related assertions can constrain which values can be associated to a mutvar; these are checked for each assignment.

Compile-time analysis tries to get rid of the dynamic components (allocation of mutvars, type checking, etc.)
Example: term dereference

Implementation note

Prolog variables are represented in the WAM by a self-referencing cell which may have to be reached following a reference chain.

- Dereferencing in ImProlog:

\[
\text{deref}(\text{Reg}) : - \\
( \text{tagof}(\text{Reg}, \text{ref}) \rightarrow \\
\text{tagval}(\text{Reg}, V), \\
T = @V, \\
( @\text{Reg} = T \rightarrow \text{true} \\
; \text{Reg} \leq T, \\
\text{deref}(\text{Reg}) \\
) \\
; \text{true} ).
\]

- Follow reference chain; stop when non-variable found or when cell points to itself
- Uses mutable variables
- Uses external operations on native types: \text{tagof}/2 and \text{tagval}/2
Defining WAM instructions in ImProlog

- Example: unify a term (in a mutvar) and a constant (tagged word):

\[
\text{:- pred u_cons(mutable, cons).}
\]
\[
\text{u_cons(A, Cons) :-}
\]
\[
\quad T \leq @A, \text{deref}(T),
\]
\[
\quad ( \text{tagof}(@T, \text{ref}) \rightarrow \text{bind}(@T, \text{Cons}) ; @T = \text{Cons} ).
\]

- Generate code for same (conceptually) instruction with first argument in \(x\) register:

\[
\text{:- ins_alias(ux_cons, u_cons(xreg_mutable, cons)).}
\]

- Force \(ux\_cons\) to have opcode 97:

\[
\text{:- ins_entry(97, ux_cons).}
\]

- A panoply of transformation operations available
Transformations on the Instruction Set
Generate new instructions out of existing ones

They change both emulator and compiler backend

- Instruction Merging [om]: Several instructions can be fused (fetch cycles saved, but size emulator increased)
- Instructions for Built-ins [ib]: Convert calls to built-ins into specialized emulator instructions (e.g. one instruction per arithmetic operation)
Transformations on Instruction Code

Optimize existing instructions

- Those do not create new instructions
- Code is manipulated or alternative translation schemes chosen
  - Unfolding Rules [ur]: user unfolding rules to control builtin expansion, code sharing between merged instructions, etc.
  - Different Tag Switching Schemes [ts]: C switch or predefined switch patterns based on tag encodings
  - Connected Continuations [jc]: Tests in conditionals have sometimes been just performed: reuse the result (e.g. in \(\text{deref}(T), (\text{ref}(T) \to A ; B), T\) is reference-checked just before exiting \text{deref}/1)
  - Read/Write mode specialization [rw]: generate two switches (for reading and writing) and two versions of each instruction by partial evaluation w.r.t. the \text{mode} flag
Example of generated code (all together)

... 
ux_cons: 
   tagged t;
   t=X(Op(short,P,2));
   deref(&t);
   if (TagOf(t)==REF) {
      bind(t,Op(tagged,P,4));
   } else {
      if (t!=Op(tagged,P,4))
         goto failure_ins;
   }
   P=Skip(P,8);
   goto loop;
... 

loop:
   switch(Op(short,P,0)) {
      ...
      case 97: goto ux_cons;
      ...
   }

void deref(tagged_t *a0) {
   deref:
      if (tagged_tag(*a0)==REF) {
         tagged_t t0;
         t0=*(tagged_val(*a0));
         if ((*a0)!=t0) {
            *a0=t0;
            goto deref; })
      }
      else {
         if (t!=Op(tagged,P,4))
            goto failure_ins;
      }
   P=Skip(P,8);
   goto loop;
...
Experimental Evaluation

- Recreate the Ciao emulator (more than 200 instructions) from a simpler instruction set (approx. 50 instructions)
- Automatically, by means of program transformations
- Initial code and performance (product of person-years effort) exactly recovered from simpler instruction set
- Plus: exercise all (compatible) transformation combinations and test them: 96 emulators $\times$ 13 benchmarks = 1248 performance figures
- Found slightly better configurations (around 20% speed-up for some cases, starting from an already optimized engine).
Fine-Grained Instructions and Native Code Generation
Using Types/Modes

- Define rich abstract domain for terms (including instantiation): nonvar, var, int, large, etc.

- Enriched from type annotations from analyzers at the source level.
Using Types/Modes: Optimizing Unification

- Finer granularity than WAM code to optimize **unification** and **control** (calls and backtracking).
- Simple data and control instructions, derived from splitting WAM instruction definitions: load, bind, read, move, deref, push-choice, etc.
- Explicit control instructions (conditional and unconditional jumps)

<table>
<thead>
<tr>
<th>Input program</th>
<th>Low-level code</th>
</tr>
</thead>
<tbody>
<tr>
<td>fact(0, 1).</td>
<td>...</td>
</tr>
<tr>
<td>fact(X, Y) :- X &gt; 0,</td>
<td></td>
</tr>
<tr>
<td>X0 is X - 1,</td>
<td></td>
</tr>
<tr>
<td>fact(X0, Y0),</td>
<td></td>
</tr>
<tr>
<td>Y is X * Y0.</td>
<td>deref(x(0),x(0))</td>
</tr>
<tr>
<td></td>
<td>cjump(not(test(var,x(0))),label(V3))</td>
</tr>
<tr>
<td></td>
<td>load(temp,int(0))</td>
</tr>
<tr>
<td></td>
<td>bind(var,x(0),nonvar,temp)</td>
</tr>
<tr>
<td></td>
<td>jump(label(V4))</td>
</tr>
<tr>
<td>label(V3):</td>
<td>label(V3):</td>
</tr>
<tr>
<td></td>
<td>cjump(not(test(int(0),x(0))),fail)</td>
</tr>
<tr>
<td>label(V4):</td>
<td>label(V4):</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
Using Types/Modes: Optimizing Unification (2)

- Assume that an assertion is introduced in code (e.g., by a source analysis).
- Conditions in jumps are reduced to true or false when enough information is available.
- Code in red is removed.

<table>
<thead>
<tr>
<th>Assertion with (instantiation) types</th>
</tr>
</thead>
<tbody>
<tr>
<td>:- true pred fact(X, Y): int<em>var=&gt;int</em>number.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input program</th>
<th>Low-level code</th>
</tr>
</thead>
<tbody>
<tr>
<td>fact(0, 1).</td>
<td>...</td>
</tr>
</tbody>
</table>
| fact(X, Y) :- X > 0,
X0 is X - 1,
fact(X0, Y0),
Y is X * Y0. | defer(x(0),x(0))
cjump(not(test(var,x(0))),label(V3))
load(temp,int(0))
bind(var,x(0),nonvar,temp)
jump(label(V4))
label(V3):
cjump(not(test(int(0),x(0))),fail)
label(V4):
... |
Mapping to C (Using Det/NF)

- Compile chunks of instructions (code blocks) as C functions.
- Loop driven execution of code blocks (that mimics emulator’s main loop):
  
  ```c
  while (code != NULL)
  code = ((Cont (*)(State *))code)(s);
  ```

- Determinism information can be used to specialize the control instructions (without passing through the loop).
- We implement different translation templates for each case of determinism (e.g. det, nondet, semidet) when generating C code.
  - 0-1 solutions: `bool foo()`
  - 1 solution: `void foo()`
  - 1 solution returning a value: `TAGGED foo()`

- We simplify choice point creation or fail instructions when possible (e.g. avoid untrailing).
Optimizing Control

Example:

\[
\begin{align*}
p(X) & :\text{ var}(X), \, !, \, q(X). \\
p(X) & :\text{ r}(X).
\end{align*}
\]

Introduction of explicit control instructions:

\[
\begin{align*}
p(X) : \\
&\text{metachoice Choice} \\
&\text{pushchoice p2}(X) \\
&\text{call var}(X) \\
&\text{cut Choice} \\
&\text{call q}(X) \\
&\text{proceed} \\
p2(X) : \\
&\text{call r}(X) \\
&\text{proceed}
\end{align*}
\]
Optimizing Control

Introduction of the calling interfaces (suppose that p/1 is det, var/1 is semidet, q/1 is det, r/1 is det):

```prolog
det p(X):
  metachoice Choice
  pushchoice p2(X)
  if not var(X) then fail
  cut Choice
  call q(X)
  return

det p2(X):
  call r(X)
  return
```
Optimizing Control

Choice point analysis (var/1 does not create choice points):

\[
\text{det } p(X): \\
\text{metachoice } \text{Choice} \\
\text{pushchoice } p2(X) \ [\text{Choice2}] \\
\text{if not var}(X) \text{ then} \\
\quad \text{fail } [\text{Choice2}] \\
\text{cut } \text{Choice} \\
\text{call } q(X) \\
\text{return} \\
\text{det } p2(X): \\
\text{call } r(X) \\
\text{return}
\]
Optimizing Control

**Fail specialization:**

```prolog
\[\text{det } p(X) :\]
\text{metachoice Choice}
\text{pushchoice } p2(X) \ [\text{Choice2}]
\text{if not } \text{var}(X) \text{ then}
\text{restorestate } [\text{Choice2}]
\text{cut Choice } [\text{prev Choice2 = Choice}]
\text{call } p2(X)
\text{return}
\text{cut Choice}
\text{call } q(X)
\text{return}
\text{det } p2(X) :\]
\text{call } r(X)
\text{return}
```

Morales (IMDEA, UPM)
Optimizing Control

Fail specialization (2) (var/1 does not modify the state):

```prolog
def p(X):
def p2(X):
```

```prolog
det p(X):
    metachoice Choice
    pushchoice p2(X) [Choice2]
    if not var(X) then
        {remove: restorestate [Choice2]}
    cut Choice [prev Choice2 = Choice]
    call p2(X)
    return
    cut Choice
    call q(X)
    return
    det p2(X):
        call r(X)
        return
```
Optimizing Control

- **Simplification (remove useless choice points and cuts):**

  ```
  det p(X):
  if not var(X) then
    call p2(X)
  return
  call q(X)
  return
  det p2(X):
  call r(X)
  return
  ```
Optimizing Control

Trivial to obtain:

```plaintext
det p(X):
    if not var(X) then
        call r(X)
        return
    call q(X)
    return
```
## Impact of Optimizations

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>queens11</td>
<td>691</td>
<td>391 (1.76)</td>
<td>208 (3.32)</td>
<td>166 (4.16)</td>
</tr>
<tr>
<td>crypt</td>
<td>1525</td>
<td>976 (1.56)</td>
<td>598 (2.55)</td>
<td>597 (2.55)</td>
</tr>
<tr>
<td>primes</td>
<td>896</td>
<td>697 (1.28)</td>
<td>403 (2.22)</td>
<td>402 (2.22)</td>
</tr>
<tr>
<td>tak</td>
<td>9836</td>
<td>5625 (1.74)</td>
<td>5285 (1.86)</td>
<td>771 (12.75)</td>
</tr>
<tr>
<td>deriv</td>
<td>125</td>
<td>83 (1.50)</td>
<td>82 (1.52)</td>
<td>72 (1.74)</td>
</tr>
<tr>
<td>poly</td>
<td>439</td>
<td>251 (1.74)</td>
<td>199 (2.20)</td>
<td>177 (2.48)</td>
</tr>
<tr>
<td>qsort</td>
<td>521</td>
<td>319 (1.63)</td>
<td>378 (1.37)</td>
<td>259 (2.01)</td>
</tr>
<tr>
<td>exp</td>
<td>494</td>
<td>508 (0.97)</td>
<td>469 (1.05)</td>
<td>459 (1.07)</td>
</tr>
<tr>
<td>fib</td>
<td>263</td>
<td>245 (1.07)</td>
<td>234 (1.12)</td>
<td>250 (1.05)</td>
</tr>
<tr>
<td>knights</td>
<td>621</td>
<td>441 (1.40)</td>
<td>390 (1.59)</td>
<td>356 (1.74)</td>
</tr>
<tr>
<td><strong>Average Speedup</strong></td>
<td>(1.47)</td>
<td>(1.88)</td>
<td>(3.18)</td>
<td></td>
</tr>
</tbody>
</table>
A Case Study
Case study: **Sound Spatializer**

- Monoaural (virtual) source
- Sound should arrive to each ear in a different phase (simulated by software)
- Hard real time
- Slow processor ($\approx \frac{1}{25}$ of regular laptop)

**Highlights on actual code: (written in Ciao)**

- Compass polled concurrently
- Trigonometric functions
- Sound samples read lazily
- Right / left ear samples share memory area
- Samples automatically buffered / deallocated
- No need to declare data structure sizes
- $\approx 80$ lines of code!
Compilation Grades and their Impact
(Or, the power of static analysis)

- Test: Process a 120 sec. sample as fast as possible
- Utilization = \( \frac{\text{processing time}}{\text{sample time}} \) (how much processor time used?)
- Plus careful listening at right speed
- Also, results in a regular laptop (for reference) + in a Gumstix
- In green: the combination that is fast enough

<table>
<thead>
<tr>
<th>Compilation grade</th>
<th>Non-Specialized</th>
<th>Specialized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bytecode</td>
<td>4.70</td>
<td>115.95</td>
</tr>
<tr>
<td>N.C. via C</td>
<td>3.87</td>
<td>98.08</td>
</tr>
<tr>
<td>” + det</td>
<td>3.28</td>
<td>92.42</td>
</tr>
<tr>
<td>” + modes/types</td>
<td>3.00</td>
<td>88.38</td>
</tr>
<tr>
<td>” + unboxing</td>
<td>2.90</td>
<td>85.70</td>
</tr>
</tbody>
</table>
A Case Study

Compilation Grades and their Impact
(Or, the power of static analysis)

- Compilation to bytecode + execution in V.M.
- No analysis
- Fast enough on laptop – too slow in Gumstix
- O.S., etc. overhead produces artifacts & delays
- However: memory consumption stable and adequate

<table>
<thead>
<tr>
<th>Compilation grade</th>
<th>Non-Specialized</th>
<th>Specialized</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bytecode</td>
<td>4.70</td>
<td>115.95</td>
</tr>
<tr>
<td>N.C. via C</td>
<td>3.87</td>
<td>98.08</td>
</tr>
<tr>
<td>” + det</td>
<td>3.28</td>
<td>92.42</td>
</tr>
<tr>
<td>” + modes/types</td>
<td>3.00</td>
<td>88.38</td>
</tr>
<tr>
<td>” + unboxing</td>
<td>2.90</td>
<td>85.70</td>
</tr>
</tbody>
</table>

Morales (IMDEA, UPM)  
Optim. Compilation for Logic Programming  
Prometidos SC
Compilation Grades and their Impact
(Or, the power of static analysis)

- Automatic specialization applied first
- Source-to-source program transformation
- Improved results, but not fast enough yet
- Specialized versions consistently better

<table>
<thead>
<tr>
<th>Compilation grade</th>
<th>Non-Specialized</th>
<th></th>
<th>Specialized</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Bytecode</td>
<td>4.70</td>
<td>115.95</td>
<td>96.6%</td>
<td>3.91</td>
</tr>
<tr>
<td>N.C. via C</td>
<td>3.87</td>
<td>98.08</td>
<td>81.7%</td>
<td>3.36</td>
</tr>
<tr>
<td>” + det</td>
<td>3.28</td>
<td>92.42</td>
<td>77.0%</td>
<td>2.85</td>
</tr>
<tr>
<td>” + modes/types</td>
<td>3.00</td>
<td>88.38</td>
<td>73.6%</td>
<td>2.57</td>
</tr>
<tr>
<td>” + unboxing</td>
<td>2.90</td>
<td>85.70</td>
<td>71.4%</td>
<td>2.47</td>
</tr>
</tbody>
</table>

Morales (IMDEA, UPM) Optim. Compilation for Logic Programming Prometidos SC
A Case Study

Compilation Grades and their Impact
(Or, the power of static analysis)

- Compile control to native code
- C as intermediate language
- Data structures still represented as in V.M.
- First success on Gumstix with specialized program
- But still brittle

<table>
<thead>
<tr>
<th>Compilation grade</th>
<th>Non-Specialized</th>
<th></th>
<th>Specialized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bytecode</td>
<td>4.70</td>
<td>115.95</td>
<td>96.6%</td>
</tr>
<tr>
<td>N.C. via C</td>
<td>3.87</td>
<td>98.08</td>
<td>81.7%</td>
</tr>
<tr>
<td>” + det</td>
<td>3.28</td>
<td>92.42</td>
<td>77.0%</td>
</tr>
<tr>
<td>” + modes/types</td>
<td>3.00</td>
<td>88.38</td>
<td>73.6%</td>
</tr>
<tr>
<td>” + unboxing</td>
<td>2.90</td>
<td>85.70</td>
<td>71.4%</td>
</tr>
</tbody>
</table>

Morales (IMDEA, UPM) Optim. Compilation for Logic Programming Prometidos SC
A Case Study

Compilation Grades and their Impact
(Or, the power of static analysis)

- Use determinism plus mode (directionality) and type (shapes) information
- Used to optimize data compilation to native code
- More robust

<table>
<thead>
<tr>
<th>Compilation grade</th>
<th>Non-Specialized</th>
<th></th>
<th>Specialized</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>i686</td>
<td>Gumstix</td>
<td>i686</td>
<td>Gumstix</td>
</tr>
<tr>
<td>Bytecode</td>
<td>4.70</td>
<td>115.95</td>
<td>96.6%</td>
<td>3.91</td>
</tr>
<tr>
<td>N.C. via C</td>
<td>3.87</td>
<td>98.08</td>
<td>81.7%</td>
<td>3.36</td>
</tr>
<tr>
<td>&quot; + det</td>
<td>3.28</td>
<td>92.42</td>
<td>77.0%</td>
<td>2.85</td>
</tr>
<tr>
<td>&quot; + modes/types</td>
<td>3.00</td>
<td>88.38</td>
<td>73.6%</td>
<td>2.57</td>
</tr>
<tr>
<td>&quot; + unboxing</td>
<td>2.90</td>
<td>85.70</td>
<td>71.4%</td>
<td>2.47</td>
</tr>
</tbody>
</table>

Morales (IMDEA, UPM) Optim. Compilation for Logic Programming Prometidos SC
A Case Study

Compilation Grades and their Impact
(Or, the power of static analysis)

- Compile F.P. numbers to native data when possible
- Bypassing tagged V.M. representation
- But still well-behaved w.r.t. G.C.!
- Advantage: use of F.P. trigonometric functions

<table>
<thead>
<tr>
<th>Compilation grade</th>
<th>Non-Specialized</th>
<th>Specialized</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>i686</td>
<td>Gumstix</td>
</tr>
<tr>
<td></td>
<td>secs.</td>
<td>secs.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Util.</td>
</tr>
<tr>
<td>Bytecode</td>
<td>4.70</td>
<td>115.95</td>
</tr>
<tr>
<td>N.C. via C</td>
<td>3.87</td>
<td>98.08</td>
</tr>
<tr>
<td>” + det</td>
<td>3.28</td>
<td>92.42</td>
</tr>
<tr>
<td>” + modes/types</td>
<td>3.00</td>
<td>88.38</td>
</tr>
<tr>
<td>” + unboxing</td>
<td><strong>2.90</strong></td>
<td><strong>85.70</strong></td>
</tr>
</tbody>
</table>

Morales (IMDEA, UPM)  Optim. Compilation for Logic Programming  Prometidos SC
A More Demanding Scenario

- What would happen should compass rate be much more higher?
  - A compass data per audio sample
- Arithmetic (e.g., unboxing) becomes more relevant
- Also, specialization plays an important role
- Results: when arithmetic dominates, optimizing compilation achieves a 7-fold speedup w.r.t. original bytecode

<table>
<thead>
<tr>
<th>Compilation mode</th>
<th>Non-Specialized</th>
<th>Specialized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bytecode</td>
<td>25.64</td>
<td>14.00</td>
</tr>
<tr>
<td>N.C. via C</td>
<td>21.59</td>
<td>11.99</td>
</tr>
<tr>
<td>ld. + semidet</td>
<td>19.59</td>
<td>11.53</td>
</tr>
<tr>
<td>ld. + mode/type</td>
<td>19.19</td>
<td>11.08</td>
</tr>
<tr>
<td>ld. + unboxing</td>
<td>6.97</td>
<td>3.62</td>
</tr>
</tbody>
</table>

(i686 data)

- Speeds within 20% speed of a comparable program hand-written in C, achieved automatically from very high level language.
Open Research Problems
Open Research Problems

- Apply ImProlog-like language to write portable and specializable built-ins.
- Extend the emulator generation framework (e.g. stack operands).
- Exploit abstraction level to attempt engine/compiler (formal) verification.
- Just-In-Time, profile-driven, compilation.
- Analysis and optimizations for memory reuse.
- Unboxing and term representation.
- Optimizations for other domains, search strategies, and extensions (e.g., CLP, tabling).
Conclusions
Conclusions

- Technique & language uncommon for embedded applications
- However: success in case study
  - Efficiency comparable to C in demanding scenario
  - For a task traditionally perceived as unfit
- Complex cases can also benefit from declarative technology
- State-of-the-art analysis & compilation techniques applied
- Experimental framework implemented in the development (SVN) version of Ciao:
  http://www.ciaohome.org

Main message

Advanced compilation techniques can expand the application field of declarative languages to new generation of pervasive / embedded systems (including forthcoming multicore small processors).