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 $((\bigwedge_{C \in P} C) \wedge \neg Q)$.
- SLD is refutation complete. However, Prolog typically implements a depth-first strategy. It is potentially incomplete if the search space contains infinite branches.

Introduction

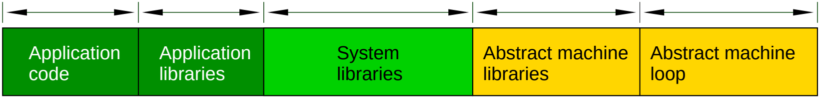
- Despite this, Prolog has some nice properties:
  - Very efficient and cheap search (backtracking).
  - Predictable execution order resembling procedural languages (left-to-right, recursive).
  - Extensions to interact with the resolution strategy and unification: cut, negation as failure, attributed 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

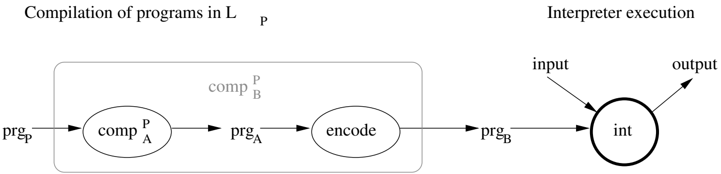
Generic Implementation of Abstract Machines

Anatomy of A.M. Systems

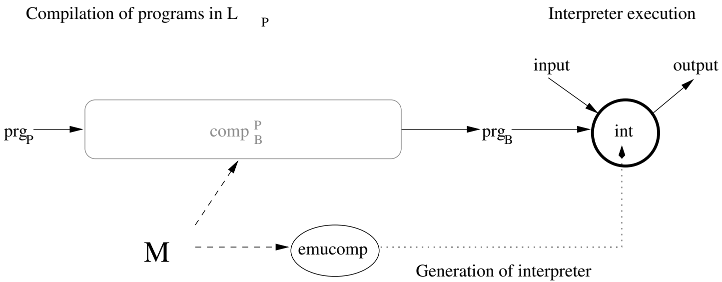


At different language levels:

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

$$
\begin{array}{l}
\mathbf{emu_A}(p, \mathit{program}) \equiv \\
\quad \langle \mathit{ins}, p' \rangle = \mathit{fetch}_A(p, \mathit{program}) \\
\quad \text{case } \mathit{ins} \text{ of} \\
\qquad \begin{array}{ll}
\langle \mathbf{move}, [\mathbf{r}(i), \mathbf{r}(j)] \rangle & : \mathit{reg}[j] := \mathit{reg}[i];\ \ p'' := p' \\
\langle \mathbf{call}, [\mathbf{label}(l)] \rangle & : \mathit{push}(p');\ \ p'' := l \\
\langle \mathbf{ret}, [] \rangle & : p'' := \mathit{pop}() \\
\text{otherwise} & : \mathbf{error}
\end{array} \\
\quad \mathbf{emu_A}(p'', \mathit{program})
\end{array}
$$

Deconstruction of $\mathcal{L}_A$ Instructions

Towards a Generic Interpreter

- Original definition:

$$\langle \mathbf{move}, [\mathbf{r}(i), \mathbf{r}(j)] \rangle : \quad reg[j] := reg[i]; p'' := p'$$

- Extract the arguments M_{args}:

$$\mathbf{r}(i) \rightarrow reg[i]$$

- And obtain M_{def}: (*cont* abstracts continuations)

$$\langle \mathbf{move}, [a, b] \rangle \rightarrow [b := a; \quad cont(next)]$$

- Define operand types, to obtain the signature (or *format*) for each instruction:

$$M_{absexp}(\mathbf{r}(\_)) = \mathbf{r}$$

- Make the instruction set explicit (useful later):

$$\langle \mathbf{move}, [\mathbf{r}, \mathbf{r}] \rangle \in M_{ins}$$

The Generic Interpreter

- Using previous definitions we can write a generic interpreter:

$$\begin{array}{l} \mathbf{int_1}(p, program, M) \equiv \\ \quad \langle\langle name, args\rangle, p'\rangle = fetch_A(p, program) \\ \quad \text{if } \neg valid_A(\langle name, args\rangle, M_{ins}, M_{absexp}) \text{ then } \mathbf{error} \\ \quad cont = \lambda a \rightarrow [p'' := a] \\ \quad [\![M_{def}(p', cont, name, M_{args}(args))]\!] \\ \quad \mathbf{int_1}(p'', program, M) \end{array}$$

- Definition of an abstract machine $M = (M_{def}, M_{arg}, M_{ins}, M_{absexp})$
- M_{def} (relates $\mathcal{L}_A$ instructions and their $\mathcal{L}_C$ code)
- M_{args} (relates $\mathcal{L}_A$ args and their $\mathcal{L}_C$ representation)
- M_{ins}, M_{absexp} validate that the instruction is correct
- $[\![...]\!]$ 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

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

$$
\begin{array}{l}
M_{ins} = \\
\quad \{\ \langle \mathbf{move}, [\mathbf{r}, \mathbf{r}] \rangle \\
\quad\ \ \ \langle \mathbf{call}, [\mathbf{label}] \rangle \\
\quad\ \ \ \langle \mathbf{ret}, [] \rangle \\
\quad\ \ \ \langle \mathbf{halt}, [] \rangle\ \}
\end{array}
$$

$$
\begin{array}{l}
M_{arg}(arg) = \\
\quad \text{case } arg \text{ of} \\
\qquad \mathbf{r}(i) \rightarrow reg[i] \\
\qquad \mathbf{label}(l) \rightarrow l
\end{array}
$$

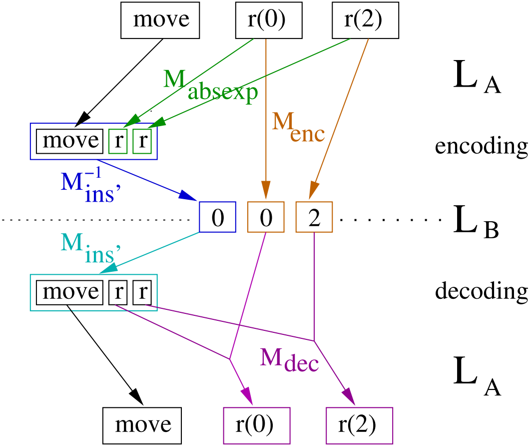
$$
\begin{array}{l}
M_{absexp}(arg) = \\
\quad \text{case } arg \text{ of} \\
\qquad \mathbf{r}(\_) \rightarrow \mathbf{r} \\
\qquad \mathbf{label}(\_) \rightarrow \mathbf{label} \\
\qquad \text{otherwise} \rightarrow \bot
\end{array}
$$

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**
- $\mathcal{L}_B$:

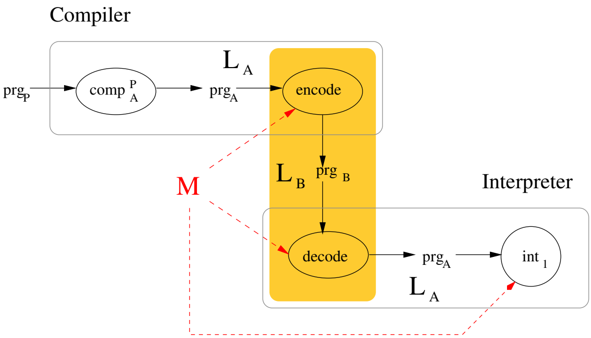
0	0	2	0	1	0	0	2	1	2

- $\mathcal{L}_B$ is much more convenient than $\mathcal{L}_A$ (for $\mathcal{L}_C$)



Pass Separation

- Augmented $M = (M_{def}, M_{arg}, M_{ins'}, M_{absexp}, M_{enc}, M_{dec})$
- 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 $\mathcal{L}_A$ disappears from the runtime section



A Generic Bytecode Interpreter

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

$$\mathbf{int_2}(p, prg, M) \equiv \mathbf{int_1}(p, decode(prg), M)$$

  More directly:

$$\begin{array}{l} \mathbf{int_2}(p, prg, M) \equiv \\ \quad opcode = prg[p] \\ \quad \langle name, format \rangle = M_{ins'}(opcode) \\ \quad \langle args, p' \rangle = decode_{ins}(format, [p], [prg], M) \\ \quad cont = \lambda a \rightarrow [\mathbf{int_2}(a, prg, M);\ \ return] \\ \quad [\![M_{def}(p', cont, name, M_{args}(args));\ \ cont(p')]\!] \end{array}$$

- $\mathbf{int_2}$ is not meant to be directly executed (big overhead!)
- Efficiency comes from partial evaluation of $\mathbf{int_2}$ w.r.t. M.
- Hint: M static and finite:
  - *opcode* can be enumerated to generate a **case**
  - Partial evalution feasible: $\mathbf{int_3} \equiv [\![spec]\!](\mathbf{int_2}, M)$
  - Removes overhead (i.e. no metaprogramming, etc.)

Example: Generated Emulator

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

$$
\begin{array}{l}
\textbf{emu}_{\textbf{B}}(p, \mathit{program}) \equiv \\
\quad \text{case } \mathit{program}[p] \text{ of} \\
\qquad 0 : \mathit{reg}[\mathit{program}[p+1]] := \mathit{reg}[\mathit{program}[p+2]]; \\
\qquad\quad \textbf{emu}_{\textbf{B}}(p+3, \mathit{program}); \mathit{return} \\
\qquad 1 : \mathit{push}(p+2); \\
\qquad\quad \textbf{emu}_{\textbf{B}}(\mathit{program}[p+1], \mathit{program}); \mathit{return} \\
\qquad 2 : \textbf{emu}_{\textbf{B}}(\mathit{pop}(), \mathit{program}); \mathit{return}
\end{array}
$$

- 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

$$emucomp : M \to code_C$$
$$emucomp = [\![spec]\!](spec, \mathbf{int_2})$$

- 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)

$$
\begin{array}{l}
\mathit{emucomp}(M) = \\
\;[\;\mathbf{emu_B}(p, prg) \equiv \\
\quad\quad \text{case } \mathit{get\_opcode}(p, prg) \text{ of} \\
\quad\quad\; \mathit{opcode}_1 \;:\; \mathit{inscomp}(\mathit{opcode}_1, M) \\
\quad\quad\; \ldots \\
\quad\quad\; \mathit{opcode}_n \;:\; \mathit{inscomp}(\mathit{opcode}_n, M)] \\
\quad\quad \text{where } \mathit{opcode}_i \in \mathit{domain}(M_{ins'})
\end{array}
$$

$$
\begin{array}{l}
\mathit{inscomp}(\mathit{opcode}, M) = \\
\;[M_{def}(p', \mathit{cont}, \mathit{name}, M_{args}(\mathit{args}));\;\; \mathit{cont}(p')] \\
\text{where} \\
\;\langle \mathit{name}, \mathit{format} \rangle = M_{ins'}(\mathit{opcode}) \\
\;\langle \mathit{args}, p' \rangle = \mathit{decode}_{ins}(\mathit{format}, [p], [prg], M) \\
\;\mathit{cont} = \lambda a \rightarrow [\mathbf{emu_B}(a, prg);\;\; \mathit{return}]
\end{array}
$$

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 $\mathbf{emu_G}$ is automatically obtained as $\mathbf{emu_G} = emucomp(M')$
- Size of bytecode program & emulator savings

	Emulator savings		Bytecode size	
	Full program+libs	Stripped-down	Full	Strip.
hw	28%	71%	33116	48
boyer	26%	46%	40198	8594
fib	28%	70%	31758	154
knights	27%	54%	32306	702
poly	26%	48%	34682	3078
query	28%	59%	32816	1212
stream	26%	46%	34496	1428
tak	28%	67%	31886	282
Range	≈27%	45%-75%	-	-

Related Work

- Biernacki Dariusz, Olivier Danvy. **From Interpreter to Logic Engine by Defunctionalization**, LOPSTR (2003):

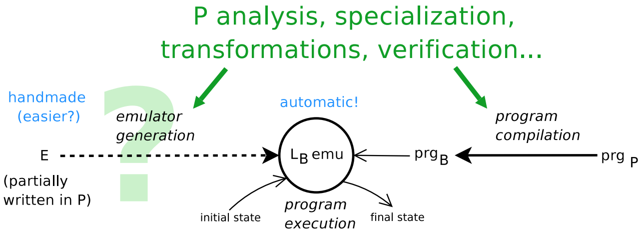
Continuations	*Defunctionalization* $\longrightarrow$	Constructors
Application of conts		apply_cont

  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).
- M. Anton Erlt, David Gregg, Andreas Krall, and Bernd Paysan. **vmgen - A Generator of Efficient Virtual Machine Interpreters**, Software: Practice and Experience (2002).
  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.



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* $<=$ *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:

```
deref(Reg) :-
  ( tagof(@Reg,ref) ->
      tagval(@Reg,V),
      T = @V,
      ( @Reg = T -> true
      ; Reg <= T,
        deref(Reg)
      )
  ; true ).
```

- Follow reference chain; stop when non-variable found or when cell points to itself
- Uses mutable variables
- Uses external operations on native types: `tagof/2` and `tagval/2`

Defining WAM instructions in ImProlog

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

```
:- pred u_cons(mutable, cons).
u_cons(A, Cons) :-
    T <= @A, deref(T),
    ( tagof(@T, ref) -> bind(@T,Cons) ; @T = Cons ).
```

- Generate code for same (conceptually) instruction with first argument in `X` register:

```
:- ins_alias(ux_cons, u_cons(xreg_mutable, cons)).
```

- Force `ux_cons` to have opcode 97:

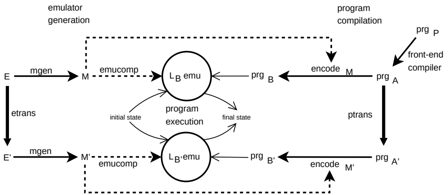
```
:- ins_entry(97, ux_cons).
```

- 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 `deref(T), (ref(T) -> A ; B)`, `T` is reference-checked just before exiting `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 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; }}}
```

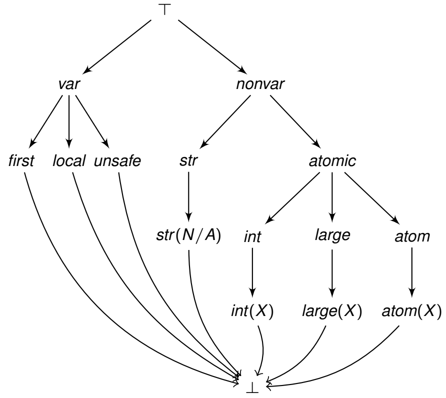
Experimental Evaluation

- 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>
<tr><th>Input program</th><th>Low-level code</th></tr>
<tr><td><pre>fact(0, 1).
fact(X, Y) :- X > 0,
    X0 is X - 1,
    fact(X0, Y0),
    Y is X * Y0.</pre></td><td><pre>   ...
   deref(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):
   ...</pre></td></tr>
</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.

Assertion with (instantiation) types

```
:- true pred fact(X, Y): int*var=>int*number.
```

Input program	Low-level code

`fact(0, 1).`
`fact(X, Y) :- X > 0,`
`    X0 is X - 1,`
`    fact(X0, Y0),`
`    Y is X * Y0.`
	`...`
	`deref(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):

```
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:

```
p(X) :- var(X), !, q(X).
p(X) :- r(X).
```

- Introduction of explicit control instructions:

```
p(X):
  metachoice Choice
  pushchoice p2(X)
  call var(X)
  cut Choice
  call q(X)
  proceed
p2(X):
  call r(X)
  proceed
```

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):

```
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):

```
det p(X):
  metachoice Choice
  pushchoice p2(X) [Choice2]
  if not var(X) then
    fail [Choice2]
  cut Choice
  call q(X)
  return
det p2(X):
  call r(X)
  return
```

Optimizing Control

- Fail specialization:

```
det p(X):
  metachoice Choice
  pushchoice p2(X) [Choice2]
  if not var(X) then
    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

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

```
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:

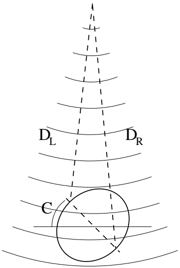
```
det p(X):
  if not var(X) then
    call r(X)
    return
  call q(X)
  return
```

Impact of Optimizations

Program	Bytecode (Std. Ciao)	Non opt. C	Opt1. C	Opt2. C
queens11 (1)	691	391 (1.76)	208 (3.32)	166 (4.16)
crypt (1000)	1525	976 (1.56)	598 (2.55)	597 (2.55)
primes (10000)	896	697 (1.28)	403 (2.22)	402 (2.22)
tak (1000)	9836	5625 (1.74)	5285 (1.86)	771 (12.75)
deriv (10000)	125	83 (1.50)	82 (1.52)	72 (1.74)
poly (100)	439	251 (1.74)	199 (2.20)	177 (2.48)
qsort (10000)	521	319 (1.63)	378 (1.37)	259 (2.01)
exp (10)	494	508 (0.97)	469 (1.05)	459 (1.07)
fib (1000)	263	245 (1.07)	234 (1.12)	250 (1.05)
knights (1)	621	441 (1.40)	390 (1.59)	356 (1.74)
Average Speedup		(1.47)	(1.88)	(3.18)

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

Compilation grade	Non-Specialized			Specialized		
	i686	Gumstix		i686	Gumstix	
	secs.	secs.	Util.	secs.	secs.	Util.
Bytecode	4.70	115.95	96.6%	3.91	103.49	86.2%
N.C. via C	3.87	98.08	81.7%	3.36	88.27	73.6%
" + det	3.28	92.42	77.0%	2.85	83.74	69.8%
" + modes/types	3.00	88.38	73.6%	2.57	79.42	66.2%
" + unboxing	2.90	85.70	71.4%	2.47	78.01	65.0%

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

Compilation grade	Non-Specialized			Specialized		
	i686	Gumstix		i686	Gumstix	
	secs.	secs.	Util.	secs.	secs.	Util.
Bytecode	4.70	115.95	96.6%	3.91	103.49	86.2%
N.C. via C	3.87	98.08	81.7%	3.36	88.27	73.6%
” + det	3.28	92.42	77.0%	2.85	83.74	69.8%
” + modes/types	3.00	88.38	73.6%	2.57	79.42	66.2%
” + unboxing	2.90	85.70	71.4%	2.47	78.01	65.0%

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

Compilation grade	Non-Specialized			Specialized		
	i686	Gumstix		i686	Gumstix	
	secs.	secs.	Util.	secs.	secs.	Util.
Bytecode	4.70	115.95	96.6%	3.91	103.49	86.2%
N.C. via C	3.87	98.08	81.7%	3.36	88.27	73.6%
" + det	3.28	92.42	77.0%	2.85	83.74	69.8%
" + modes/types	3.00	88.38	73.6%	2.57	79.42	66.2%
" + unboxing	2.90	85.70	71.4%	2.47	78.01	65.0%

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

Compilation grade	Non-Specialized			Specialized		
	i686	Gumstix		i686	Gumstix	
	secs.	secs.	Util.	secs.	secs.	Util.
Bytecode	4.70	115.95	96.6%	3.91	103.49	86.2%
N.C. via C	3.87	98.08	81.7%	3.36	88.27	73.6%
" + det	3.28	92.42	77.0%	2.85	83.74	69.8%
" + modes/types	3.00	88.38	73.6%	2.57	79.42	66.2%
" + unboxing	2.90	85.70	71.4%	2.47	78.01	65.0%

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

Compilation grade	Non-Specialized			Specialized		
	i686	Gumstix		i686	Gumstix	
	secs.	secs.	Util.	secs.	secs.	Util.
Bytecode	4.70	115.95	96.6%	3.91	103.49	86.2%
N.C. via C	3.87	98.08	81.7%	3.36	88.27	73.6%
" + det	3.28	92.42	77.0%	2.85	83.74	69.8%
" + modes/types	3.00	88.38	73.6%	2.57	79.42	66.2%
" + unboxing	2.90	85.70	71.4%	2.47	78.01	65.0%

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

Compilation grade	**Non-Specialized**			**Specialized**		
	i686	**Gumstix**		**i686**	**Gumstix**	
	secs.	secs.	Util.	secs.	secs.	Util.
Bytecode	4.70	115.95	96.6%	3.91	103.49	86.2%
N.C. via C	3.87	98.08	81.7%	3.36	88.27	73.6%
" + det	3.28	92.42	77.0%	2.85	83.74	69.8%
" + modes/types	3.00	88.38	73.6%	2.57	79.42	66.2%
" + unboxing	2.90	85.70	71.4%	2.47	78.01	65.0%

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 importante role
- Results: when arithmetic dominates, optimizing compilation achieves a **7-fold speedup** w.r.t. original byecode

Compilation mode	**Non-Specialized**	**Specialized**
Bytecode	25.64	14.00
N.C. via C	21.59	11.99
Id. + semidet	19.59	11.53
Id. + mode/type	19.19	11.08
Id. + unboxing	6.97	**3.62**

(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).