Types Summer School
Gothenburg Sweden August 2005

Dependently Typed Programming

Benjamin Grégoire
INRIA Sophia Antipolis, France

Lecture 2:
Conversion test, compilation

Summary of the problem

Proof / type checkers based on dependent types work up to conversion:

$$\frac{\Gamma \vdash M : A \quad A \approx B}{\Gamma \vdash M : B}$$

It is very convenient: allows small proofs and automation (using reflexive proofs)

If we have a algorithm for testing convertibility, we get a type checker

Testing convertibility require strong β-reduction (under λ abstractions)

For most proofs, the amount of reduction is small (simple interpreter suffice)

But proofs based on reflection require large amounts of reductions. The speed of the reducer becomes the limiting factor

Convertibility is decidable

Testing convertibility of two terms is decidable if the reduction rules are

- Confluents $\Rightarrow$ Church-Rosser, uniqueness of normal forms
- Strongly normalizing

$$
\begin{tikzcd}
A \arrow[dr, "*"] & \approx & B \arrow[dl, "*"'] \\
& M \arrow[dl, "*"'] \arrow[dr, "*"] & \\
\mathsf{NF}(A) & = & \mathsf{NF}(B)
\end{tikzcd}
$$

λ-calculus

$$
\begin{array}{lrcl}
\text{terms} & t & ::= & x \mid \lambda x.t \mid t\ t \\
\text{values(WNF)} & v & ::= & \lambda x.t \mid x\ v_1 \ldots v_n
\end{array}
$$

Conversion algorithm:

$$
\frac{t_1 = t_2}{t_1 \approx t_2} \qquad \frac{\mathsf{WNF}(t_1) \approx \mathsf{WNF}(t_2)}{t_1 \approx t_2}
$$

$$
\frac{v_1 = v_2}{v_1 \approx v_2} \qquad \frac{x = y \quad v_i \approx w_i}{x\ v_1 \ldots v_n \approx y\ w_1 \ldots w_n}
$$

$$
\frac{\mathsf{WNF}(\lambda x.M\ \ z) \approx \mathsf{WNF}(\lambda y.M'\ \ z) \quad z \text{ fresh}}{\lambda x.M \approx \lambda y.M'}
$$

Computing the WNF

```
type term = Var of var | Abs of var*term | App of term*term

let rec wnf t =
   match t with
   | Var _ | Abs _ -> t

   | App(t1, t2) ->
      let v1 = wnf t1 in
      let v2 = wnf t2 in
      match v1 with
      | Abs(x,u) -> wnf (subst u x v2)
      | _ -> App(v1,v2)
```

WNF by compilation

WNF : execution of ML-like program

$$\lambda\text{-term} \xrightarrow{\text{Compilation}} \text{bytecode} \xrightarrow[\text{Abs. Machine}]{\text{Execution}} \text{value}$$

bytecode : sequence of instructions

Problem: usual compilation techniques work only for closed terms

$$\text{WNF}(\lambda x.M\,z)$$

ZINC abstract machine

ZINC : a stack based abstract machine in call by value

Instructions : Acc, Closure, Grab, Pushra, Apply, Return

Representation of values v (closures): $[c, e]$

Environment e : $[v_1; \ldots; v_n]$

Components of the machine:

c code pointer

e environment (values associate to variables)

s stack (arguments + intermediate results + return address)

n number of available arguments on the top of s

Compilation and execution of variables

Compilation scheme: $[\![t]\!]k \leadsto c$

The resulting code c compute the value corresponding to t, push it on top of the stack, then restart the execution of k

$$[\![x]\!]k = \mathsf{Acc}(i);\ \ k$$

where i = deBruijn index of x

Code	Env	Stack	#args
$\mathsf{Acc}(i); k$	e	s	n
k	e	$e(i).s$	n

Compilation and execution of applications

$$[\![f\ a_1\ \dots\ a_i]\!]k \ = \ \mathsf{Pushra}(k);\ [\![a_i]\!]\ \dots\ [\![a_1]\!]\ [\![f]\!]\ \mathsf{Apply}(i)$$

Code	Env	Stack	#args
Pushra(k); c	e	s	n
c	e	$\langle k, e, n\rangle.s$	n
Apply(i)	e	$[c, e'].v_1 \dots v_i.\langle k, e, n\rangle.s$	n
c	e'	$v_1 \dots v_i.\langle k, e, n\rangle.s$	i

Compilation and execution of functions

$$
\begin{array}{rcl}
[\![\lambda x_1 \ldots . \lambda x_n . t]\!] k & = & \mathsf{Closure}(c); k \\
c & = & \underbrace{\mathsf{Grab}; \ldots; \mathsf{Grab}}_{n \text{ times}}; [\![t]\!] \mathsf{Return}
\end{array}
$$

Code	Env	Stack	#args
Closure(c); k	e	s	n
k	e	$[c, e].s$	n
Grab; k	e	$v.s$	$n + 1$
k	$v.e$	s	n
Return	e	$v.\langle k, e', n\rangle.s$	0
k	e'	$v.s$	n

Under or over application

Under application:

Code	Env	Stack	#args
Grab; c	e	$\langle k, e', n \rangle.s$	0
k	e'	$[(\text{Grab}; c), e].s$	n

Over application:

Code	Env	Stack	#args
Return	e	$[c, e'].s$	$n > 0$
c	e'	s	n

Compilation with free variables

Code	Env	Stack	#args
Acc(i); k	e	s	n
k	e	$e(i).s$	n

Free variables have no associated value in the environment

$\Rightarrow$ add values for free variables

What should be the value associated to a free variables?

What happens when this value is applied?

What is the computational behavior of a free variable?

Symbolic calculus:

$$\begin{array}{llcl} \text{Terms} & t & ::= & x \mid t\ t \mid v \\ \text{Values} & v & ::= & \lambda x.t \mid [\tilde{x}] \end{array}$$

Reduction rules:

$$\begin{array}{rcl} (\lambda x.t)\ v & \longrightarrow & t[x := v] \\ [\tilde{x}]\ v & \longrightarrow & [\tilde{x}\ v] \end{array}$$

Computational behavior of a free variable

Symbolic calculus:

$$
\begin{array}{lrcl}
\text{Terms} & t & ::= & x \mid t\ t \mid v \\
\text{Values} & v & ::= & \lambda x.t \mid [k] \\
\text{Accumulators} & k & ::= & \tilde{x} \mid k\ v
\end{array}
$$

Reduction rules:

$$
\begin{array}{rcl}
(\lambda x.t)\ v & \longrightarrow & t[x := v] \\
[k]\ v & \longrightarrow & [k\ v]
\end{array}
$$

The value associate to a free variable is a function that accumulate its arguments

Encoding accumulator

Code	Env	Stack	#args
Apply(n)	_	$[c, e].v_1 \ldots v_n.\langle c', e', n'\rangle.s$	_
c	e	$v_1 \ldots v_n.\langle c', e', n'\rangle.s$	n'

The top value can now be a accumulator, encoding of accumulator should be compatible with the one of closure

$$[\text{Accumulate}, \widehat{k}]$$

where $\widehat{k}$ is the machine-level encoding of k: $\widehat{k} = [\tilde{x}; v_1; \ldots; v_n]$

This suffices to trick function application:

Code	Env	Stack	#args
Apply(n)	e	$[\text{Accumulate}, \widehat{k}].v_1 \ldots v_n.\langle c', e', n'\rangle.s$	_
Accumulate	$\widehat{k}$	$v_1 \ldots v_n.\langle c', e', n'\rangle.s$	n
c'	e'	$[\text{Accumulate}, (\widehat{k}.v_1 \ldots v_n)].s$	n'

The move from $[\text{Accumulate}, \widehat{k}]$ to $[\text{Accumulate}, (\widehat{k}.v_1 \ldots v_n)]$ implements exactly the symbolic reduction

$[k]\ v_1\ \ldots\ v_n \longrightarrow [k\ v_1\ \ldots\ v_n]$

Distinguishing feature of this encoding

The representation of $[k]$ looks like a function

$\Rightarrow$ No need to test at application time whether the function is a closure or an accumulator

$\Rightarrow$ No overhead on evaluation of closed terms

Similarly, we arrange that the representation of $[k]$ looks like the representation of inductive constructors

$\Rightarrow$ No overhead for ι-reduction

Experimental results

4-colors theorem

Perimeter	Coq	Coq-vm	OCaml bytecode	OCaml natif
11	56.7s	1.68s	1.18s	0.30s
12	259s	6.50s	6.18s	1.92s
13	680s	14.8s	15.5s	4.11s

Prime numbers

		Size	time
1234567891 (10)			
Deductive	:	3099	13.26 s
Reflexive	:	58	0.59 s
20988936657440586486151264256610222593863921 (44)			
Deductive	:	18509	1862.52 s
Reflexive	:	95	21.30 s

Conclusion

Conversion is very convenient: allows small proofs and automation (using reflexive proofs)

The use of a compiler and an abstract machine for testing convertibility leads to an efficient algorithm

So reflexive proofs can be efficiently type checked