A Compiled Implementation of Normalization by Evaluation

Klaus Aehlig[1] Florian Haftmann[2] Tobias Nipkow[2]

[1]Department of Computer Science
Swansea University

[2]Institut für Informatik
Technische Universität München

Conference on Theorem Proving in Higher Order Logics 2008

Normalization

Compute normal form of term wrt list of equations (incl β)

Normalization

Compute normal form of term wrt list of equations (incl β)

Equations:

- Recursion equations with pattern matching:
  $S(x) + y = S(x + y)$

Normalization

Compute normal form of term wrt list of equations (incl β)

Equations:

- Recursion equations with pattern matching:
  $S(x) + y = S(x + y)$
- But also arbitrary term-rewriting rules:
  $(x + y) + z = x + (y + z)$

Normalization

Compute normal form of term wrt list of equations (incl β)

Equations:

- Recursion equations with pattern matching:
  $S(x) + y = S(x + y)$
- But also arbitrary term-rewriting rules:
  $(x + y) + z = x + (y + z)$

Terms:

- Ground terms: $S(0) + S(0) \ \rightarrow^* \ S(S(0))$

Normalization

Compute normal form of term wrt list of equations (incl β)

Equations:

- Recursion equations with pattern matching:
  $S(x) + y = S(x + y)$
- But also arbitrary term-rewriting rules:
  $(x + y) + z = x + (y + z)$

Terms:

- Ground terms: $S(0) + S(0) \ \rightarrow^* \ S(S(0))$
- But also free and bound variables:
  $\lambda a.S(a) + S(b) \ \rightarrow^* \ \lambda a.S(S(a + b))$

Normalization in Theorem Provers: Why and How

Why: Applications of *fast* evaluation/symbolic execution:

Normalization in Theorem Provers: Why and How

Why: Applications of *fast* evaluation/symbolic execution:

- Validation and testing

Normalization in Theorem Provers: Why and How

Why: Applications of *fast* evaluation/symbolic execution:

- Validation and testing
- Proofs involving complex computations
  (4CT, Kepler Conjecture)

Normalization in Theorem Provers: Why and How

Why: Applications of *fast* evaluation/symbolic execution:

- Validation and testing
- Proofs involving complex computations
  (4CT, Kepler Conjecture)

How:

Normalization in Theorem Provers: Why and How

Why: Applications of *fast* evaluation/symbolic execution:

- Validation and testing
- Proofs involving complex computations
  (4CT, Kepler Conjecture)

How:

1. Compile to ML-like language (with pattern-matching)

Normalization in Theorem Provers: Why and How

Why: Applications of *fast* evaluation/symbolic execution:

- Validation and testing
- Proofs involving complex computations (4CT, Kepler Conjecture)

How:

1. Compile to ML-like language (with pattern-matching)
2. Evaluate

Normalization in Theorem Provers: Why and How

Why: Applications of *fast* evaluation/symbolic execution:

- Validation and testing
- Proofs involving complex computations (4CT, Kepler Conjecture)

How:

1. Compile to ML-like language (with pattern-matching)
2. Evaluate
3. Read back

Normalization in Theorem Provers: Why and How

Why: Applications of *fast* evaluation/symbolic execution:

- Validation and testing
- Proofs involving complex computations
  (4CT, Kepler Conjecture)

How:

1. Compile to ML-like language (with pattern-matching)
2. Evaluate
3. Read back

Bypass inference kernel.

Normalization in Theorem Provers: Why and How

Why: Applications of *fast* evaluation/symbolic execution:

- Validation and testing
- Proofs involving complex computations
  (4CT, Kepler Conjecture)

How:

1. Compile to ML-like language (with pattern-matching)
2. Evaluate
3. Read back

Bypass inference kernel.

Model and verify implementation.

Untyped Normalization by Evaluation

Formalisation in Isabelle

Related and Future Work

Untyped Normalization by Evaluation

Formalisation in Isabelle

Related and Future Work

Handling of Variables

- "compile, evaluate, read back" works fine...
  for *closed* term of *ground type*
  - But what about open terms?
  - Even closed functions can only be presented as $\lambda x.t$ with x free in t

Handling of Variables

- "compile, evaluate, read back" works fine...
  for *closed* term of *ground type*
  - But what about open terms?
  - Even closed functions can only be presented as $\lambda x.t$ with x free in t
- So we do have to handle free variables!
  - Need a data type containing both, its own function space and free variables
  - First attempt

```
datatype Univ =
| Var of string
| Clo of (Univ -> Univ)
```

Handling of Variables

- "compile, evaluate, read back" works fine...
  for *closed* term of *ground type*
  - But what about open terms?
  - Even closed functions can only be presented as $\lambda x.t$ with x free in t
- So we do have to handle free variables!
  - Need a data type containing both,
    its own function space and free variables
  - First attempt

    ```
    datatype Univ =
    | Var of string
    | Clo of (Univ -> Univ)
    ```

  - But need to implement application!
    What is `(Var "x")` v supposed to mean?

Handling of Variables (cont'd)

- Have to define what an application `(Var "x")` v means.

```
datatype Univ =
| Var of string
| Clo of (Univ -> Univ)
```

Handling of Variables (cont'd)

- Have to define what an application `(Var "x")` v means.
- An application $(x\vec{t})s$ *never* creates a new redex!

```
datatype Univ =
| Var of string
| Clo of (Univ -> Univ)
```

Handling of Variables (cont'd)

- Have to define what an application `(Var "x")` v means.
- An application $(x\vec{t})s$ *never* creates a new redex!

$\rightsquigarrow$ Can just collect the arguments

```
datatype Univ =
| Var of string * Univ list
| Clo of (Univ -> Univ)

apply (Var x vs) v = Var x (vs @ [v])
apply (Clo f)    v = f v
```

Handling of Variables (cont'd)

- Have to define what an application `(Var "x")` v means.
- An application $(x\vec{t})s$ *never* creates a new redex!

⇝ Can just collect the arguments

```
datatype Univ =
| Var of string * Univ list
| Clo of (Univ -> Univ)

apply (Var x vs) v = Var x (vs @ [v])
apply (Clo f)    v = f v
```

- As `Univ` denotes normal terms, we can go back easily

```
term (Var x vs) = foldl Tapply (V x) (map term vs)
term (Clo f)    = let x = new_var() in
                  Lam x (term (f x))
```

Constructors, Arity, ...

- Fine for the pure lambda-calculus.

```
datatype Univ =
| Var of string * Univ list
| Clo of (Univ -> Univ)

apply (Var x vs) v = Var x (vs @ [v])
apply (Clo  f )  v = f v
```

Constructors, Arity, ...

- Want lambda-calculus with data constructors (0, S, ...).

```
datatype Univ =
| Var of string * Univ list
| Clo of (Univ -> Univ)

apply (Var x vs) v = Var x (vs @ [v])
apply (Clo  f )  v = f v
```

Constructors, Arity, ...

- Want lambda-calculus with data constructors (0, S, ...).

$\rightsquigarrow$ Add constructors

```
datatype Univ =
| C of string
| Var of string * Univ list
| Clo of (Univ -> Univ)

apply (Var x vs) v = Var x (vs @ [v])
apply (Clo  f )  v = f v
```

Constructors, Arity, ...

- Want lambda-calculus with data constructors (0, S, ...).

⇝ Add constructors Application?

```
datatype Univ =
| C of string
| Var of string * Univ list
| Clo of (Univ -> Univ)

apply (Var x vs) v = Var x (vs @ [v])
apply (Clo  f )  v = f v
```

Constructors, Arity, ...

- Want lambda-calculus with data constructors (0, S, ...).

$\rightsquigarrow$ Add constructors Application won't cause a redex!

```
datatype Univ =
| C of string * Univ list
| Var of string * Univ list
| Clo of (Univ -> Univ)

apply (Var x vs) v = Var x (vs @ [v])
apply (C s args) v = C s (args @ [v])
apply (Clo  f )  v = f v
```

Constructors, Arity, ...

- Want lambda-calculus with data constructors (0, S, ...).
- Some functions have higher arity

$$\begin{array}{llllll} \texttt{min} & x & 0 & = & x \\ \texttt{min} & 0 & y & = & y \\ \texttt{min} & (Sx) & (Sy) & = & S(\texttt{min}\; x\; y) \end{array}$$

$\rightsquigarrow$ Add constructors

```
datatype Univ =
| C of string * Univ list
| Var of string * Univ list
| Clo of (Univ -> Univ)

apply (Var x vs) v = Var x (vs @ [v])
apply (C s args) v = C s (args @ [v])
apply (Clo  f )  v = f v
```

Constructors, Arity, ...

- Want lambda-calculus with data constructors $(0, S, \ldots)$.
- Some functions have higher arity

$$\begin{array}{llllll} \mathtt{min} & x & 0 & = & x \\ \mathtt{min} & 0 & y & = & y \\ \mathtt{min} & (Sx) & (Sy) & = & S(\mathtt{min}\ x\ y) \end{array}$$

$\rightsquigarrow$ Add constructors, allow n-ary functions

```
datatype Univ =
| C of string * Univ list
| Var of string * Univ list
| Clo of int * (Univ list -> Univ)

apply (Var x vs) v = Var x (vs @ [v])
apply (C s args) v = C s (args @ [v])
apply (Clo  f )  v = f v
```

Constructors, Arity, ...

- Want lambda-calculus with data constructors (0, S, ...).
- Some functions have higher arity

$$\begin{array}{lllll} \texttt{min} & x & 0 & = & x \\ \texttt{min} & 0 & y & = & y \\ \texttt{min} & (Sx) & (Sy) & = & S(\texttt{min}\ x\ y) \end{array}$$

$\rightsquigarrow$ Add constructors, allow n-ary functions, partially applied

```
datatype Univ =
| C of string * Univ list
| Var of string * Univ list
| Clo of int * (Univ list -> Univ) * Univ list

apply (Var x vs) v = Var x (vs @ [v])
apply (C s args) v = C s (args @ [v])
apply (Clo 0 f vs)  v = f (vs @ [v])
apply (Clo n f vs)  v = Clo (n-1) f (vs @ [v])
```

Compiling Functions

- Still a little detail to solve: *How do we translate functions?*

Compiling Functions

- Still a little detail to solve: *How do we translate functions?*
- Example

$$
\begin{array}{llcll}
\texttt{apd} & \mathsf{Nil} & \mathit{bs} & = & \mathit{bs} \\
\texttt{apd} & (\mathsf{Cons}\ a\ \mathit{as}) & \mathit{bs} & = & \mathsf{Cons}\ a\ (\texttt{apd}\ \mathit{as}\ \mathit{bs})
\end{array}
$$

Compiling Functions

- Still a little detail to solve: *How do we translate functions?*
- Example

$$\begin{array}{llcll} \mathtt{apd} & \mathsf{Nil} & bs & = & bs \\ \mathtt{apd} & (\mathsf{Cons}\ a\ as) & bs & = & \mathsf{Cons}\ a\ (\mathtt{apd}\ as\ bs) \end{array}$$

- Just match against the constructors in `Univ`

```
fun apd [C "Nil" [],      bs] = bs
 |  apd [C "Cons" [a, as], bs] = C "Cons" [a, apd as bs]
```

Compiling Functions

- Still a little detail to solve: *How do we translate functions?*
- Example

$$\begin{array}{llcll} \mathtt{apd} & \mathsf{Nil} & bs & = & bs \\ \mathtt{apd} & (\mathsf{Cons}\ a\ as) & bs & = & \mathsf{Cons}\ a\ (\mathtt{apd}\ as\ bs) \end{array}$$

- Just match against the constructors in `Univ`
  Not exhaustive!! *E.g., we have* `Var "x"`.

```
fun apd [C "Nil" [],      bs] = bs
 |  apd [C "Cons" [a, as], bs] = C "Cons" [a, apd as bs]
```

Compiling Functions

- Still a little detail to solve: *How do we translate functions?*
- Example

$$
\begin{array}{llcll}
\mathrm{apd} & \mathrm{Nil} & bs & = & bs \\
\mathrm{apd} & (\mathrm{Cons}\ a\ as) & bs & = & \mathrm{Cons}\ a\ (\mathrm{apd}\ as\ bs)
\end{array}
$$

- Just match against the constructors in `Univ`
  and add a default clause

```
fun apd [C "Nil" [],       bs] = bs
 |  apd [C "Cons" [a, as], bs] = C "Cons" [a, apd as bs]
 |  apd [as,               bs] = C "apd" [as,bs]
```

Compiling Functions

- Still a little detail to solve: *How do we translate functions?*
- Example with rewrite rule

$$
\begin{array}{llcll}
\mathrm{apd} & \mathrm{Nil} & bs & = & bs \\
\mathrm{apd} & (\mathrm{Cons}\ a\ as) & bs & = & \mathrm{Cons}\ a\ (\mathrm{apd}\ as\ bs) \\
\\
\mathrm{apd} & (\mathrm{apd}\ as\ bs) & cs & = & \mathrm{apd}\ as\ (\mathrm{apd}\ bs\ cs)
\end{array}
$$

- Just match against the constructors in `Univ`
  and add a default clause

```
fun apd [C "Nil" [],      bs] = bs
 |  apd [C "Cons" [a, as], bs] = C "Cons" [a, apd as bs]
 |  apd [as,               bs] = C "apd" [as,bs]
```

Compiling Functions

- Still a little detail to solve: *How do we translate functions?*
- Example with rewrite rule

$$
\begin{array}{llcll}
\mathrm{apd} & \mathrm{Nil} & bs & = & bs \\
\mathrm{apd} & (\mathrm{Cons}\ a\ as) & bs & = & \mathrm{Cons}\ a\ (\mathrm{apd}\ as\ bs) \\
\\
\mathrm{apd} & (\mathrm{apd}\ as\ bs) & cs & = & \mathrm{apd}\ as\ (\mathrm{apd}\ bs\ cs)
\end{array}
$$

- Just match against the constructors in `Univ`
  and add a default clause
- *For rewrite rules, match against the function "constructors"*

```
fun apd [C "Nil" [],      bs] = bs
 |  apd [C "Cons" [a, as], bs] = C "Cons" [a, apd as bs]
 |  apd [C "apd" [as, bs], cs] =
        apd [as, apd [bs, cs]]
 |  apd [as,                bs] = C "apd" [as,bs]
```

Models of ML-Terms and λ-Terms

We use de Bruijn indices.

Models of ML-Terms and λ-Terms

We use de Bruijn indices.

ML-terms consist of ML's λ-calculus

$$
\begin{array}{lcl}
ml & = & C \; cname \\
 & | & V \; nat \\
 & | & A \; ml \; (ml \; list) \\
 & | & Lam \; ml
\end{array}
$$

Models of ML-Terms and λ-Terms

We use de Bruijn indices.

ML-terms consist of ML's λ-calculus + `constructors`

$$
\begin{array}{rcl}
ml & = & C\ cname \\
 & | & V\ nat \\
 & | & A\ ml\ (ml\ list) \\
 & | & Lam\ ml \\
 & | & \texttt{C}\ cname\ (ml\ list) \\
 & | & \texttt{Var}\ nat\ (ml\ list) \\
 & | & \texttt{Clo}\ ml\ (ml\ list)\ nat
\end{array}
$$

Models of ML-Terms and λ-Terms

We use de Bruijn indices.

ML-terms consist of ML's λ-calculus + `constructors` + functions

$$
\begin{array}{rcl}
ml & = & C\ cname \\
 & | & V\ nat \\
 & | & A\ ml\ (ml\ list) \\
 & | & Lam\ ml \\
 & | & \texttt{C}\ cname\ (ml\ list) \\
 & | & \texttt{Var}\ nat\ (ml\ list) \\
 & | & \texttt{Clo}\ ml\ (ml\ list)\ nat \\
 & | & apply\ ml\ ml
\end{array}
$$

Models of ML-Terms and λ-Terms

We use de Bruijn indices.

ML-terms consist of ML's λ-calculus + `constructors` + functions

$$
\begin{array}{rcl}
ml & = & C \ \mathit{cname} \\
 & | & V \ \mathit{nat} \\
 & | & A \ \mathit{ml} \ (\mathit{ml} \ \mathit{list}) \\
 & | & \mathit{Lam} \ \mathit{ml} \\
 & | & \texttt{C} \ \mathit{cname} \ (\mathit{ml} \ \mathit{list}) \\
 & | & \texttt{Var} \ \mathit{nat} \ (\mathit{ml} \ \mathit{list}) \\
 & | & \texttt{Clo} \ \mathit{ml} \ (\mathit{ml} \ \mathit{list}) \ \mathit{nat} \\
 & | & \mathit{apply} \ \mathit{ml} \ \mathit{ml}
\end{array}
$$

Abstract λ-terms:

$$tm \ = \ C \ \mathit{cname} \mid V \ \mathit{nat} \mid tm \bullet tm \mid \lambda tm$$

Models of ML-Terms and λ-Terms

We use de Bruijn indices.

ML-terms consist of ML's λ-calculus + constructors + functions

$$
\begin{array}{rcl}
ml & = & C \ cname \\
 & | & V \ nat \\
 & | & A \ ml \ (ml \ list) \\
 & | & Lam \ ml \\
 & | & \mathtt{C} \ cname \ (ml \ list) \\
 & | & \mathtt{Var} \ nat \ (ml \ list) \\
 & | & \mathtt{Clo} \ ml \ (ml \ list) \ nat \\
 & | & apply \ ml \ ml
\end{array}
$$

Abstract λ-terms:

$$tm \ = \ C \ cname \mid V \ nat \mid tm \bullet tm \mid \lambda tm \mid term \ ml$$

Reduction $\rightarrow$ on pure λ-terms

Reduction $\longrightarrow$ on pure λ-terms

- β-reduction

Reduction $\to$ on pure λ-terms

- β-reduction
- η-expansion

Reduction $\longrightarrow$ on pure λ-terms

- β-reduction
- η-expansion
- rewriting wrt R :: $(\mathit{cname} \times \mathit{tm}\ \mathit{list} \times \mathit{tm})\mathit{set}$

Reduction $\to$ on pure λ-terms

- β-reduction
- η-expansion
- rewriting wrt $R :: (cname \times tm\,list \times tm)set$

$$\frac{(c, ts, t) \in R}{C\ c \bullet\bullet\ map\ (subst\ \sigma)\ ts\ \ \to\ \ subst\ \sigma\ t}$$

Reduction $\rightarrow$ on pure λ-terms

- β-reduction
- η-expansion
- rewriting wrt $R :: (cname \times tm\,list \times tm)set$

$$\frac{(c, ts, t) \in R}{C\ c \bullet\bullet\ map\ (subst\ \sigma)\ ts\ \rightarrow\ subst\ \sigma\ t}$$

where $t \bullet\bullet [t_1, \ldots, t_n]\ =\ t \bullet t_1 \bullet \cdots \bullet t_n$

Reduction $\Rightarrow$ on ML-terms

- β-reduction

Reduction $\Rightarrow$ on ML-terms

- β-reduction
- rewriting wrt *compR* $:: (cname \times ml\ list \times ml)set$

$$\frac{(c, vs, v) \in R \qquad \forall n.\ closed(\sigma\ n)}{A\ (C\ c)\ (map\ subst\ \sigma)\ vs\ \Rightarrow\ subst\ \sigma\ v}$$

Reduction $\Rightarrow$ on ML-terms

- β-reduction
- rewriting wrt *compR* :: $(cname \times ml\ list \times ml)set$

$$\frac{(c, vs, v) \in R \qquad \forall n.\ closed(\sigma\ n)}{A\ (C\ c)\ (map\ subst\ \sigma)\ vs\ \Rightarrow\ subst\ \sigma\ v}$$

- Reductions for *apply*, eg

$$apply\ (\mathtt{Clo}\ 0\ f\ \ vs)\ v\ \ \Rightarrow\ \ A\, f\ (vs@[v])$$

Reduction $\Rightarrow$ on ML-terms

- β-reduction
- rewriting wrt *compR* $:: (cname \times ml\ list \times ml)set$

$$\frac{(c, vs, v) \in R \qquad \forall n.\ closed(\sigma\ n)}{A\ (C\ c)\ (map\ subst\ \sigma)\ vs\ \Rightarrow\ subst\ \sigma\ v}$$

- Reductions for *apply*, eg

$$apply\ (\mathtt{Clo}\ 0\ f\ \ vs)\ v\ \Rightarrow\ A\, f\, (vs@[v])$$

- Reductions for *term*, eg

$$\begin{array}{l} term\ (\mathtt{Clo}\ f\ \ vs\ \ n)\ \Rightarrow \\ \lambda(term\ (apply\ (lift\ 0\ (\mathtt{Clo}\ f\ \ vs\ \ n))\ (\mathtt{Var}\, 0\, []))) \end{array}$$

Compilation from λ-Terms to ML-terms

Two variants:

Compilation from λ-Terms to ML-terms

Two variants:

- *comp-fixed* for compiling a term to be reduced

Compilation from λ-Terms to ML-terms

Two variants:

- *comp-fixed* for compiling a term to be reduced
  Treats variables as fixed: $V \mapsto$ `Var`

Compilation from λ-Terms to ML-terms

Two variants:

- *comp-fixed* for compiling a term to be reduced
  Treats variables as fixed: $V \mapsto$ `Var`
- *comp-open* for compiling rewrite rules

Compilation from λ-Terms to ML-terms

Two variants:

- *comp-fixed* for compiling a term to be reduced
  Treats variables as fixed: $V \mapsto$ `Var`
- *comp-open* for compiling rewrite rules
  Treats variables as open: $V \mapsto V$

Compilation from λ-Terms to ML-terms

Two variants:

- *comp-fixed* for compiling a term to be reduced
  Treats variables as fixed: $V \mapsto \texttt{Var}$
- *comp-open* for compiling rewrite rules
  Treats variables as open: $V \mapsto V$

Rule compilation:

$$compR \;=\; \ldots comp\text{-}open \ldots R \ldots$$

Main Correctness Theorem

Main Correctness Theorem

If t and t' are pure λ-terms (no *term*)

Main Correctness Theorem

If t and t' are pure λ-terms (no *term*)

and $\mathit{term}(\mathit{comp\text{-}fixed}\ t) \Rightarrow^* t'$

Main Correctness Theorem

If t and t' are pure λ-terms (no *term*)

and $\mathit{term}(\mathit{comp\text{-}fixed}\ t) \Rightarrow^* t'$

then $t \longrightarrow^* t'$

Statistics

Statistics

Size of theory:	1100 loc

Statistics

Size of theory:	1100 loc
Definitions:	30%

Statistics

Size of theory:	1100 loc
Definitions:	30%
Proofs about substitutions:	30%

Statistics

Size of theory:	1100 loc
Definitions:	30%
Proofs about substitutions:	30%
Main proof:	40%

Implementation

Implementation

- Builds on Isabelle's code generation infrastructure

Implementation

- Builds on Isabelle's code generation infrastructure
- 475 loc

Implementation

- Builds on Isabelle's code generation infrastructure
- 475 loc
- Does not perform proofs,

Implementation

- Builds on Isabelle's code generation infrastructure
- 475 loc
- Does not perform proofs, hence verification

Implementation

- Builds on Isabelle's code generation infrastructure
- 475 loc
- Does not perform proofs, hence verification
- Typical performance figures:

Implementation

- Builds on Isabelle's code generation infrastructure
- 475 loc
- Does not perform proofs, hence verification
- Typical performance figures:
  100 $\times$ faster than simplifier

Implementation

- Builds on Isabelle's code generation infrastructure
- 475 loc
- Does not perform proofs, hence verification
- Typical performance figures:

  100 × faster than simplifier

  10 × slower than direct compilation to ML

Untyped Normalization by Evaluation

Formalisation in Isabelle

Related and Future Work

Related Work

Related Work

Berger, Eberl & Schwichtenberg [98/03]
Compiled NbE in Scheme/MINLOG
Kernel extension

Related Work

Berger, Eberl & Schwichtenberg [98/03]

Compiled NbE in Scheme/MINLOG

Kernel extension

Barras [TPHOLs 00]

Abstract machine for fast rewriting by inference in HOL

Related Work

Berger, Eberl & Schwichtenberg [98/03]

Compiled NbE in Scheme/MINLOG

Kernel extension

Barras [TPHOLs 00]

Abstract machine for fast rewriting by inference in HOL

Grégoire & Leroy [ICFP 02]

Abstract machine for fast normalization in Coq

Kernel extension

Verified

Future Work

Generalize:

Future Work

Generalize:

- Repeated variables on lhs

Future Work

Generalize:

- Repeated variables on lhs
- Ordered rewriting for permutative rules

Future Work

Generalize:

- Repeated variables on lhs
- Ordered rewriting for permutative rules
- Conditional rewriting?

Future Work

Generalize:

- Repeated variables on lhs
- Ordered rewriting for permutative rules
- Conditional rewriting?
- ...