Full Reductions at Full Throttle

INRIA Research Group

July 21, 2014
Unification & Resolution

**Unification**
- Solving equations of symbolic expressions
- Search for constraints
- Deduce substitutions

**Resolution**
- Inference rule
- Satisfiability of propositional formula
- Unsatisfiability of first-order logic formula

Usage:
- Search local context, match goal with a local hypothesis
- If found, return a subgoal for each premise
Drawbacks

- Large search space
Drawbacks

- Large search space
- No computational power
Drawbacks

- Large search space
- No computational power
  - Even with sufficient information, can get stuck
Drawbacks

- Large search space
- No computational power
  - Even with sufficient information, can get stuck

\[ \forall \sigma : \text{State.} \forall v : \mathbb{Z}. (\text{lookup } \sigma \ v \ 1) \rightarrow \langle \text{while } 3 \leq v \ \text{do} \ skip, \sigma \rangle \rightsquigarrow \sigma \]

- Premises to left of arrow: \( v \mapsto 3 \)
- Goal requires that while loop does not change \( \sigma \)
- Have information to prove \( 3 \leq 1 \rightsquigarrow \text{false} \), but cannot create and compute proof
An alternative: Proofs as Function

- Represent proof objects as functions
- Step through with context, making deductions throughout
Conversion rule for dependently-typed proof assistants like Coq:

\[
\frac{
\Gamma \vdash M : A \quad A =_\beta B
}{
\Gamma \vdash M : B
}\]
Reflection gives us an implementation of $=_{\beta}$

Compute decision procedure once, use it to evaluate any $A$ or $B$
Example

From *Certified Programming with Dependent Types*, an example of where reflection becomes useful:

\[
\text{Inductive isEven : nat } \rightarrow \text{ Prop :=}
\]
\[
| \text{Even}_0 : \text{isEven 0}
| \text{Even_SS : forall n, isEven } n \rightarrow \text{isEven } (\text{S } (\text{S } n))
\]
Example

Inductive isEven : nat → Prop :=
  | Even_0 : isEven 0
  | Even_SS : forall n, isEven n → isEven (S (S n))

Theorem even_256 : isEven 256.
  repeat constructor.
Qed.
Example

Inductive isEven : nat -> Prop :=
| Even_0 : isEven 0
| Even_SS : forall n, isEven n -> isEven (S (S n))

Theorem even_256 : isEven 256.
  repeat constructor.
Qed.

print even_256.
  even_256 = Even_SS ( Even_SS ( Even_SS ( ... 

Size of proof term is super-linear with size of input
Second Example

- How to decide $x \leq y$?
Second Example

- How to decide $x \leq y$?
- Can use constructors to build derivation
  - Runs in time linear to the input size
Second Example

- How to decide $x \leq y$?
- Can use constructors to build derivation
  - Runs in time linear to the input size
- Can use decision procedure

$$f(x, y) \triangleq \begin{cases} \text{true} & \text{if } \max(x + 1 - y, 0) = 0 \\ \text{false} & \text{otherwise} \end{cases}$$
Second Example

- How to decide $x \leq y$?
- Can use constructors to build derivation
  - Runs in time linear to the input size
- Can use decision procedure
  $$f(x, y) \equiv \begin{cases} 
  \text{true} & \text{if } \max(x + 1 - y, 0) = 0 \\
  \text{false} & \text{otherwise}
  \end{cases}$$
  - Constant time
Reflection uses verified decision procedure to check proofs in linear space or better.
Reflection uses verified decision procedure to check proofs in linear space.

Need a verified way of normalizing terms
Reflection uses verified decision procedure to check proofs in at worst linear space.

Need a verified way of normalizing terms

Problem: Cannot normalize open terms in OCaml
Reflection

- Reflection uses verified decision procedure to check proofs in at worst linear space.

- Need a verified way of normalizing terms

- Problem: Cannot normalize open terms in OCaml
  - Open terms represent dependent types or assumptions within proof object
  - Proof checker needs to resolve these, but OCaml cannot reduce them
Symbolic Reduction

Syntax for expressing and evaluating potentially open terms. Treat free variables $\tilde{x}$ as *accumulators* which collect arguments.

**Syntax**

$\textbf{Term} \ni t ::= x \mid t_1 \ t_2 \mid v$

$\textbf{Val} \ni v ::= \lambda x. t \mid [\tilde{x} \ v_1 \ldots v_n]$

**Reduction Rules**

$(\lambda x. t) v \rightarrow t\{x \leftarrow v\}$ \quad \text{(}$\beta_v$\text{)}

$[\tilde{x} \ v_1\ldots v_n] v \rightarrow [\tilde{x} \ v_1 \ldots v_n \ v]$ \quad \text{(}$\beta_s$\text{)}

$\Gamma(t) \rightarrow \Gamma(t')$ if $t \rightarrow t'$ (with $\Gamma ::= t[] | []v$) \quad \text{context}$
Symbolic Reduction

- The Symbolic Reduction rules treat functions and open terms similarly.

- But we cannot just represent open terms as functions
  - Open terms can take any number of arguments
  - OCaml can only compare values at base type. Functions are not comparable.

- Need to be able to manipulate and compare open terms
- Main challenge is finding an efficient representation

- First, we give an interface for our values
module type Values = sig
  type t
  val app : t -> t -> t
  type atom = Var of var
  type head =
    | Lam of t -> t
    | Accu of atom * t list
  val head : t -> head
  val mkLam : (t -> t) -> t
  val mkAccu : atom -> t
end
Tagged Normalization

- Natural idea: use type head directly
- Can discern Accu from Lam by explicit pattern matching.
- Fold and unfold at each application

```ocaml
type t = head
let head v = v
let app t v = match t with
  | Lam f -> f v
  | Accu(a, args) -> Accu(a, v::args)
let mkLam f = Lam f
let mkAccu a = Accu(a, [])
```
Tagged Implementation

- Grègiore & Leroy, 2002
- Extension of the ZAM, which underlies the bytecode interpreter of OCaml
- Small modifications to existing abstract machine
Issues with Tags

Tags accomplish normalization, allowing proof checker to use reflection, but come with significant overhead.

- Additional memory allocation
  - Need to allocate (and immediately drop) $n - 1$ closures during the application of a function to $n$ arguments.

- Poorer locality

- Compiler has difficulty adding optimizations
OCaml has a powerful compiler — we want to use it for reductions. Much faster than proof search.

Limitations
- Cannot compare functions
- Programs are always closed terms
Tagging met our needs by explicitly converting open terms into type constructors. Arguments could then be added to the term, and we had a clear evaluation scheme.

We can do even better by treating accumulators as functions.
- Build open term by adding arguments to a function
- Treat these arguments as fields on an object
OCaml Internals

How? By taking advantage of the OCaml internals

- All objects in OCaml represented by 31 bits and one tag.
- Integers have tag ‘1’ as their LSB.
  \[
  \begin{array}{c|c|c|c|c|c|c|c}
  & & & & & & & \\
  & & & & & & & \\
  1 & 0 & \ldots & 1 \\
  & & & & & & & \\
  & & & & & & & \\
  \end{array}
  \]
- Aids in garbage collection. The tag distinguishes ints from pointers.
Functions are given a unique tag, $T_\lambda$.

| $T_\lambda$ | $C$ | $v_1$ | $\ldots$ | $v_n$ |

- $C$ is a code pointer
- $v_i$ are arguments. The free variables of $C$.

Accumulators (Objects) have tag 0

| 0 | $C$ | $k$ |

- $C$ is code pointer to a single instruction
- $k$ is memory representation of accumulator
Redefine accumulators as:

```
type t = t -> t
let rec accu atom args = fun v -> accu atom (v::args)
let mkAccu atom = accu atom []
```

- `mkAccu` gives function expecting one argument, stored in the list of `args`. 
Redefine accumulators as:

```ocaml
type t = t -> t
let rec accu atom args = fun v -> accu atom (v::args)
let mkAccu atom = accu atom []
```

- `mkAccu` gives function expecting one argument, stored in the list of args.
- **Issue**: Tag is not zero!
Use Obj library to explicitly set tag.

```ocaml
let rec accu atom args =  
  let res = fun v -> accu atom (v::args) in
  Obj.set_tag (Obj.repr res) 0;
(res : t)
```
We integrate this definition into a new head function:

```ocaml
type t = t -> t
let app f v = f v
let mkLam f = f
let getAtom o = (Obj.magic (Obj.field o 3)) : atom
let getArgs o = (Obj.magic (Obj.field o 4)) : t list
let rec head (v:t) =
  let o = Obj.repr v in
  if Obj.tag o = 0 then Accu(getAtom o, getArgs o)
  else Lam(v)
```
Sections 2 and 3 of *Full Reductions* give extensions for the full symbolic CIC and for Coinductive types

**CIC**
- Sorts, dependent products, inductive types, constructors, pattern matching & fixpoints
- Map inductive types and constructors of CIC to constructors in OCaml. New tags for each.

**CCIC**
- Infinite data, streams
- Matching forces evaluation
- Cache forced expression
Evaluation

- Compared performance against a lazy, syntactic representation manipulator and an eager implementation of tagged normalization

- Four test proofs:
  - **BDD**: Binary decision diagram for pigeonhole principle
  - **Four colour**: Gonthier & Werner’s proof (Microsoft Research, 2005)
  - **Lucas-Lehmer**: Check if a Mersenne number is prime
  - **Mini-Rubik**: Checks that any position of 2x2x2 Rubik’s cube is solvable in at most 11 moves
  - **Cooper**: Cooper’s quantification elimination on a formula with 5 variables
  - **RecNoAlloc**: $2^{27}$ recursive calls without memory allocation to store result.
Evaluation

**Standard Reduction:** Abstract machine, manipulates syntactic representations lazily

**Bytecode Interpreter:** Tagged normalization, call-by-value

**Native Compilation:** Tagless normalization

<table>
<thead>
<tr>
<th></th>
<th>Standard Reduction</th>
<th>Bytecode Interpreter</th>
<th>Native Compilation</th>
</tr>
</thead>
<tbody>
<tr>
<td>BDD</td>
<td>4min 53s (100%)</td>
<td>21.98s (7.5%)</td>
<td>11.36s (3.9%)</td>
</tr>
<tr>
<td>Four color</td>
<td>not tested</td>
<td>3h 7m (100%)</td>
<td>34m 47s (18.6%)</td>
</tr>
<tr>
<td>Lucas-Lehmen</td>
<td>10min 10s (100%)</td>
<td>29.80s (4.9%)</td>
<td>8.47s (1.4%)</td>
</tr>
<tr>
<td>Mini-Rubik</td>
<td>Out of memory</td>
<td>15.62s (100%)</td>
<td>4.48s (28.7%)</td>
</tr>
<tr>
<td>Cooper</td>
<td>Not tested</td>
<td>48.20s (100%)</td>
<td>9.38s (19.5%)</td>
</tr>
<tr>
<td>RecNoAlloc</td>
<td>2m 27s (100%)</td>
<td>14.32s (9.7%)</td>
<td>1.05s (1.05%)</td>
</tr>
</tbody>
</table>

- Run on 64-bit architecture
- Greater speedup with less garbage collection
Summary

- Used reflection to leverage computational power

- Saw trick to utilize source language for efficiently normalizing open terms

- Built off existing, trusted, powerful compiler instead of developing new techniques. Maintained separation between proof assistant and compiler.
The End