Three Applications of Strictness in the Efficient Implementaton of Higher-Order Terms

Frank Pfenning

Workshop on Implementation of Logic
Réunion Island, France
November 11, 2000

Joint work with Carsten Schürmann and Kevin Watkins

Outline

- Notational Definitions
- Strictness

1 Unification with Definitions

2 Matching and Rewriting

3 Syntactic Redundancy Elimination

- Conclusion

Notational Definitions

- Common in mathematical practice
- Some examples:

$$
\begin{array}{rcl}
\neg A & = & A \supset \bot \\
(A \equiv B) & = & (A \supset B) \wedge (B \supset A) \\
\exists ! x.\, A(x) & = & \exists x.\, A(x) \wedge \forall y.\, A(y) \supset x \doteq y \\
A \rightarrow B & = & \Pi x{:}A.\, B \qquad \text{where } x \text{ not free in } B \\
nat & = & \mu\alpha.\, 1 + \alpha \\
zero & = & fold\ (inl\ \star)
\end{array}
$$

- Based on higher-order abstract syntax, not concrete syntax [Griffin'89]

Derived Rules of Inference as Notational Definitions

- $andelr\ D = andel\ (ander\ D)$

$$
\dfrac{\begin{matrix} D \\ A \wedge (B \wedge C) \end{matrix}}{B}\, \wedge E_{LR} \qquad = \qquad \dfrac{\dfrac{\begin{matrix} D \\ A \wedge (B \wedge C) \end{matrix}}{B \wedge C}\, \wedge E_R}{B}\, \wedge E_L
$$

- $trans\ D\ E = impi\ (\lambda u.\, impe\ E\ (impe\ D\ u))$

$$
\dfrac{\begin{matrix} D \\ A \supset B \end{matrix} \qquad \begin{matrix} E \\ B \supset C \end{matrix}}{A \supset C}\, trans \qquad = \qquad \ldots
$$

Other Kinds of Definitions

- By cases $\quad pred(0) = 0, pred(n + 1) = n$
- Recursive $\quad double(0) = 0, double(n + 1) = double(n) + 2$
- Syntactic sugar $\quad [1, 2, 3] = 1{::}2{::}3{::}nil$
- Admissible rules of inference
- Sometimes, these may be thought of as notational definitions at a different semantic level.

$$pred = lam\ (\lambda x.\ case\ x\ 0\ (\lambda n.\ n))$$

Setting

- Logical framework or type theory (LF, Coq, Isabelle, Nuprl)
- All support notational definitions
- Framework tasks: (in Twelf system based on LF)
  - representation and checking (LF)
    (terms, formulas, proofs, ...)
  - search and meta-programming (Elf)
    (theorem proving, logic programming, proof transformation, ...)
  - meta-theorem proving (Twelf)
    (logical interpretations, soundness & completeness, type preservation, ...)

Core Operations

- Type checking $\Gamma \vdash M : A$
  requires convertibility $\Gamma \vdash M \doteq N : A$.
- Search $\Gamma \vdash ? : A$ and type inference
  require unification $\exists\theta. \Gamma \vdash \theta M \doteq \theta N : \theta A$
- Meta-theorem proving splits cases
  (also requiring unification).
- All of these require β-reduction
  (use deBruijn indices and explicit substitutition
  [Dowek,Hardin,Kirchner,Pf'96]).

Language

$$
\begin{array}{rrcll}
\text{Types} & A & ::= & a\,M_1 \ldots M_n \mid A \to A \mid \Pi x{:}A.\,A & \\
\text{Objects} & M & ::= & h\ M_1 \ldots M_n \mid \lambda x{:}A.\,M & \\
\text{Heads} & h & ::= & x & \textit{variables} \\
 & & \mid & c & \textit{constructors} \\
 & & \mid & d & \textit{defined constants} \\
\text{Signatures} & \Sigma & ::= & \cdot \mid \Sigma, c{:}A \mid \Sigma, d{:}A = M \mid \ldots &
\end{array}
$$

- Type-checking is easy except for convertibility.
- $\delta(d) = M$ if $d{:}A = M$ in Σ
- Will omit types.
- Consider only $\beta\eta$-long normal forms.

Semantics of Definitions

- Definitions must be semantically transparent.
- For human interface: preserve definitions!
- For efficiency: preserve definitions!
- How do we reconcile these?

$\beta\eta\delta$-Convertibility

- Huet's algorithm for definitions in (early?) Coq (ignoring some issues of control):

$$
\begin{array}{rcl}
c\,\overline{M} \doteq c\,\overline{N} & \Rightarrow & \overline{M} \doteq \overline{N} \\
c\,\overline{M} \doteq c'\,\overline{N} & & \text{fails for } c \neq c' \\
d\,\overline{M} \doteq c\,\overline{N} & \Rightarrow & \delta(d)\,\overline{M} \doteq c\,\overline{N} \\
c\,\overline{M} \doteq d\,\overline{N} & \Rightarrow & c\,\overline{M} \doteq \delta(d)\,\overline{N} \\
d\,\overline{M} \doteq d\,\overline{N} & \Rightarrow & \overline{M} \doteq \overline{N} \qquad \text{or} \\
d\,\overline{M} \doteq d'\,\overline{N} & & \text{for } d \neq d' \text{ or prev. case fails} \\
 & \Rightarrow & \delta(d)\,\overline{M} \doteq d'\,\overline{N} \\
 & \text{else} & d\,\overline{M} \doteq \delta(d')\,\overline{N} \\
 & \text{else} & \delta(d)\,\overline{M} \doteq \delta(d')\,\overline{N}
\end{array}
$$

Example: Must Expand Definitions

- $k = \lambda x.\, \lambda y.\, x$

$$
\begin{aligned}
k\ 1\ 2 \doteq k\ 1\ 3 \quad &\Rightarrow \quad 1 \doteq 1 \wedge 2 \doteq 3 \\
& \quad \text{fails} \\
&\Rightarrow \quad \delta(k)\ 1\ 2 \doteq k\ 1\ 3 \\
&\Rightarrow \quad 1 \doteq k\ 1\ 3 \\
&\Rightarrow \quad 1 \doteq \delta(k)\ 1\ 3 \\
&\Rightarrow \quad 1 \doteq 1
\end{aligned}
$$

- Even identical defined constants need to be expanded.

Example: Need Not Expand Definitions

- $\neg A = A \supset \bot$ (literally: $\neg = \lambda A.\, A \supset \bot$)

$$
\begin{array}{rcl}
\neg A_0 \doteq \neg B_0 & & \text{for } A_0 \not\doteq B_0 \\
& \Rightarrow & A_0 \doteq B_0 \\
& & \text{fails} \\
& \Rightarrow & (A_0 \supset \bot) \doteq (B_0 \supset \bot) \\
& \Rightarrow & A_0 \doteq B_0 \\
& & \text{fails again}
\end{array}
$$

- Expanding identical defined constants if often redundant.

Analysis of Huet's Algorithm

- Preserves definitions as much as possible.
- Inefficient mostly during failure.
- Type-checking mostly succeeds.
- Tactic-based search often fails because of constructor clashes.
- No unification supported!

Injectivity

- d is **injective** if for all appropriate $\overline{M}$ and $\overline{N}$,

$$d\,\overline{M} \doteq d\,\overline{N} \quad \text{implies} \quad \overline{M} \doteq \overline{N}$$

- Injectivity of d implies:

$$\text{whenever} \quad \overline{M} \not\doteq \overline{N} \quad \text{then} \quad d\,\overline{M} \not\doteq d\,\overline{N}$$

- Allows early failure, avoids expansion of definitions.
- k is not injective (must expand).
- $\neg$ is injective (need not expand).

Strictness

- Strictness is a syntactic criterion on $\delta(d)$ that guarantees injectivity.
- A definition $d{:}A = \lambda\overline{x}.\, h\,\overline{M}$ is **strict** if every parameter x_i has *at least one* strict occurrence in $\overline{M}$.
- An occurence of x of the form $x\,y_1 \ldots y_n$ in M is **strict** if

  1. all heads on the path from the root to the occurrence of x are either constructors c, strict definitions d, or locally bound variables y, but not definition parameters x_i;
  2. $y_1, \ldots, y_n$ are distinct locally bound variables (x occurs as a pattern variable).

Examples Revisited

$$
\begin{array}{rcll}
\neg A & = & A \supset \bot & \text{strict in } A \\
\mathit{nimp}\ A\ B & = & \neg B \supset \neg A & \text{strict in } A \text{ and } B \\
(A \equiv B) & = & (A \supset B) \wedge (B \supset A) & \text{strict in } A \text{ and } B
\end{array}
$$

$$
\begin{array}{rcl}
\exists! x.\, A(x) & = & \exists x.\, A(x) \wedge \forall y.\, A(y) \supset x \doteq y \\
\exists! & = & \lambda A{:}i \to o.\, \exists x.\, A\ x \wedge \forall y.\, A\ y \supset x \doteq y \\
 & & \text{two strict occurrences of } A
\end{array}
$$

Counterexamples Revisited

$$
\begin{array}{rcl}
k\ x\ y & = & x \\
 & & \text{no strict occurrence of } x \text{ or } y \\
\pi_1\ C\ e_1\ e_2 & = & \mathit{rew}\ (C(\mathit{fst}\langle e_1, e_2\rangle))\ (C(e_1)) \\
 & & \text{no strict occurrences of } C,\ e_1 \text{ or } e_2 \\
\mathit{inst}\ t\ A & = & \mathit{infer}\ (\forall\, (\lambda x.\, A\ x))\ (A\ t) \\
 & & \text{strict in } A,\text{ not strict in } t
\end{array}
$$

- These are often declared as constructors, not defined.

Optimization of Convertibility

- Mark definition as strict if it is strict in all arguments.
- Avoid retry in Huet's algorithm: commit to

$$d\,\overline{M} \doteq d\,\overline{N} \Rightarrow \overline{M} \doteq \overline{N}$$

- Practical value depends on percentage of strict definitions.
- In applications, most definitions are strict.
- Non-strictness mostly for derived rules of inference.
- Find appropriate level for definitions:

$$\begin{array}{lcll} k & : & exp \to exp \to exp & \\ & = & \lambda x{:}exp.\, \lambda y{:}exp.\, x & \text{not strict} \\ k & : & exp & \\ & = & lam(\lambda x{:}exp.\, lam(\lambda y{:}exp.\, x)) & \text{strict} \end{array}$$

Application 1: Unification with Definitions

- Unification derived from convertibility.
- Definitions much harder: strictness indispensible.
- Huet's algorithm for convertibility incomplete for unification.
- Example: occurs-check with definitions

  $$Y \doteq k\,1\,Y$$

  occurs-check fails, but unification should succeed

Counterexample: Keeping Definitions

- Consider

$$
\begin{aligned}
& k\ 1\ 2 \doteq k\ 1\ Y \wedge Y \doteq 3 \\
& \Rightarrow Y \doteq 2 \wedge Y \doteq 3 \\
& \textit{fails}
\end{aligned}
$$

  but $Y = 3$ is the most general unifier!

- Problem: unification of

$$k\ 1\ 2 \doteq k\ 1\ Y$$

  succeeds, but unifier is not most general.

- No natural backtrack points as for convertibility.

Strict Unification

- Let d be a strict definition, $\mathcal{U}(\overline{E})$ the set of unifiers of E. Then

$$\mathcal{U}(d\,\overline{M} \doteq d\,\overline{N}) = \mathcal{U}(\overline{M} \doteq \overline{N})$$

  from properties of convertibility and substitution.

- Generalized occurs-check: $X \doteq h\,\overline{M}$ fails the occurs-check if there is a strict occurrence of X in $\overline{M}$.

- Note: $id = \lambda x.\, x$ is not strict because otherwise

$$X = id(id(id\ X))$$

  would fail the generalized occurs-check.

Implementation of Strict Unification

- Definitions are classified as strict or non-strict (“abbreviations”).
- Strict definitions are preserved as much as possible.
- Non-strict definitions are expanded during parsing.
- Critical cases of algorithm:

$$
\begin{array}{rcl}
d\,\overline{M} \doteq c\,\overline{N} & \Rightarrow & \delta(d)\,\overline{M} \doteq c\,\overline{N} \\
c\,\overline{M} \doteq d\,\overline{N} & \Rightarrow & c\,\overline{M} \doteq \delta(d)\,\overline{N} \\
d\,\overline{M} \doteq d\,\overline{N} & \Rightarrow & \overline{M} \doteq \overline{N} \\
d\,\overline{M} \doteq d'\,\overline{N} & \Rightarrow & \delta(d)\,\overline{M} \doteq \delta(d')\,\overline{N} \\
 & & \text{for } d \neq d'
\end{array}
$$

Assessment

- Unification is central in logical framework
  (type reconstruction, logic programming and search)
- Strict unification seems to work well in practice.
- Exploits interactions with other features of implementation
  (de Bruijn indices, explicit substitutions).

Refinements of Strictness

- Refinement (a): strictness per parameter

$$
\begin{aligned}
\mathit{inst}\ t\ A &= \mathit{infer}\ (\forall\,(\lambda x.\,A\ x))\ (A\ t) \\
&\quad \text{strict in } A\text{, not strict in } t \\
\mathit{inst}_t\ A &= \mathit{infer}\ (\forall\,(\lambda x.\,A\ x))\ (A\ t) \\
&\quad \text{strict in remaining argument } A
\end{aligned}
$$

- Refinement (b): hereditary analysis

$$\mathit{inst}\ t\ A \;=\; \mathit{infer}\ (\forall\,(\lambda x.\,A\ x))\ (A\ t)$$

  strict in t is $A = \lambda x.\,A\ x$ is strict in x.

- Context-dependent.
- Practical value questionable: too few non-strict definitions.

Application 2: Matching and Rewriting

- Higher-order matching used in several contexts (functional logic programming, higher-order rewriting, meta-programming)
- Theorem:

$$\exists \theta.\, \Gamma \vdash \theta M \doteq N : A$$

  has unique solutions if every free variable X has one strict occurrence in M.

- Obtain higher-order patterns if *all* occurrences must be strict [Miller'91] [Nipkow'91].
- More general case arises in practice [Virga'99].
- Of interest with and without dependent types.

Application 3: Syntactic Redundancy Elimination

- LF representations with dependent types carry significant redundant information for simplicity of type-checking.
- Inflates proof terms, slows down checking.
- Syntactic redundancy elimination: drop redundant information, retain decidability of type-checking.
- Important for proof compression (proof-carrying code [Necula'98], non-linear compression).
- Important for efficient checking and unification with dependencies.

Example

$$\frac{A \qquad B}{A \wedge B} \wedge I$$

$$andi : \Pi A.\, \Pi B.\, pf(A) \to pf(B) \to pf(A \wedge B)$$

- Consider

$$\underset{}{andi} \quad \underset{}{A} \quad \underset{}{B} \quad \underset{:pf(A)}{D} \quad \underset{:pf(B)}{E} \quad : \quad pf(A \wedge B)$$

- A is redundant if D can synthesize its type.
- B is redundant if E and synthesize its type.
- A and B are redundant we know the type of the whole because A and B occur strict in $pf(A \wedge B)$!

Another Example

$$\frac{\forall x.\, A(x)}{A(t)}\,\forall E$$

$$\mathit{foralle} : \Pi t{:}i.\, \Pi A{:}i \to o.\, \mathit{pf}\,(\forall(\lambda x.\, A\; x)) \to \mathit{pf}\,(A\; t)$$

- Consider

$$\mathit{foralle}\quad t\quad A\quad \underbrace{\qquad D \qquad}_{:\mathit{pf}\,(\forall(\lambda x.\, A\; x))} \quad : \quad \mathit{pf}\,(A\; t)$$

- A is redundant if D can synthesize its type.
- A is not redundant if we only **inherit** a type (A is not strict in $\mathit{pf}\,(A\; t)$).
- t is not redundant either way.

Bi-Directional Type Checking

- Annotate each constant occurrence as **synthesizing** $\Uparrow$ or **inheriting** $\downarrow$ a type.
- Annotations are by no means unique.
- Result of strictness analysis.
- Redundant quantifiers are bracketed.

$$
\begin{array}{rcl}
andi^{\downarrow} & : & [\Pi A.\,][\Pi B.\,]\ pf^{\downarrow}(A) \to pf^{\downarrow}(B) \to pf^{\downarrow}(A \wedge B) \\
andi^{\Uparrow} & : & [\Pi A.\,][\Pi B.\,]\ pf^{\Uparrow}(A) \to pf^{\Uparrow}(B) \to pf^{\Uparrow}(A \wedge B) \\
andel^{\downarrow} & : & [\Pi A.\,]\ \Pi B.\, pf^{\downarrow}(A \wedge B) \to pf^{\downarrow}(A) \\
andel^{\Uparrow} & : & [\Pi A.\,][\Pi B.\,]\ pf^{\Uparrow}(A \wedge B) \to pf^{\Uparrow}(A) \\
foralle^{\downarrow} & : & \Pi t.\, \Pi A.\, pf^{\downarrow}(\forall(\lambda x.\, A\ x)) \to pf^{\downarrow}(A\ t) \\
foralle^{\Uparrow} & : & \Pi t.\, [\Pi A.\,]\ pf^{\Uparrow}(\forall(\lambda x.\, A\ x)) \to pf^{\Uparrow}(A\ t)
\end{array}
$$

Complex Annotations

$$
\begin{aligned}
ore^{\downarrow} \quad : \quad & [\Pi A.\,][\Pi B.\,][\Pi C.\,] \\
& pf^{\Uparrow}(A \vee B) \\
& \rightarrow (pf^{\Uparrow}(A) \rightarrow pf^{\downarrow}(C)) \\
& \rightarrow (pf^{\Uparrow}(B) \rightarrow pf^{\downarrow}(C)) \rightarrow pf^{\downarrow}(C)
\end{aligned}
$$

- Generally, no best annotation (depends on usage).
- Strictness is critical.
- Hand-crafted annotations based on knowledge about object theory may be best.
- Automatic annotations for second-order case practical (non-linear proof size compression in LF_i [Necula'98]).

Avoiding Redundant Unification

- Maintain invariant that in
$$\exists\theta.\, \Gamma \vdash \theta M \doteq \theta N : \theta A$$
we have
$$\Gamma \vdash M : A \quad \text{and} \quad \Gamma \vdash N : A$$
- Then we don't need to unify inherited arguments.
- Empirical study: 50% improvement in logic programming efficiency of 13 case studies [Michaylov & Pf'93].
- No gain for simple types.

Summary

- Strictness, motivated from notational definitions

1 unification with definitions

2 matching and rewriting (with or without definitions)

3 proof compression (with or without definitions)

- Further empirical evaluation?
- Refined analysis?