

Proof Pearl: Using Combinators to Manipulate `let`-Expressions

Michael Norrish¹ Konrad Slind²

¹National ICT Australia

²University of Utah

23 August 2005

What is a `let`?

Syntax:

$$\text{let } v = e_1 \text{ in } e_2$$

Semantics (β -reduction):

$$(\text{let } v = e_1 \text{ in } e_2) \rightarrow e_2[v := e_1]$$

Why `lets`?

In functional programming, `lets` allow

- ▶ abbreviation of sub-expressions;
- ▶ control of evaluation order

In logic (specification):

- ▶ abbreviation is again important;
- ▶ evaluation order seems irrelevant

Evaluating `lets` in Logic

Abbreviation: presentation only?

Order of evaluation: irrelevant?

No!

- ▶ Keeping goals abbreviated can mean the difference between comprehension and confusion
- ▶ Evaluation order is semantically irrelevant, but controlling it is still important in interactive proof

Evaluating `lets`: An Alternative

β -reduction is **Approach #1**.

Approach #2: Move abbreviations into a goal's assumptions.

Move from

Initial goal:

```
0 < (let x = 3 ** 100 in x + x)
```

to

```
0 < x + x
```

```
x = 3 ** 100
```

Achieving Approach #2

(Approach #2 = Lifting Abbreviations to Assumptions)

Outline

- Representing `let`-expressions

 - (Including twists caused by tuples)

- Two other techniques

- Doing it by rewriting (and combinators)

Representing `let`-expressions

Syntax

`let` $v = e_1$ `in` e_2

is sugar for

`LET` $(\lambda v. e_2)$ e_1

with `LET` defined to be

`LET` f $x = f$ x

The Pain of Paired Abstractions

It's easy to encode paired-abstractions, of the form:

$$\lambda(p, q). \dots p \dots q \dots$$

Paired notation above is sugar for

$$\text{UNCURRY } (\lambda p q. \dots p \dots q \dots)$$

and `UNCURRY` is defined to be

$$\text{UNCURRY } f \ p = f \ (\text{FST } p) \ (\text{SND } p)$$

(Rewriting with this definition is even less appealing than rewriting with `LETs`.)

Paired Abstractions Under LETs

A paired abstraction can be the first argument to a `LET`, giving

$$\text{let } (u, v) = e_1 \text{ in } e_2$$

In what follows,

- ▶ the “easy” answer for normal abstractions comes first;
- ▶ followed by the additional work needed for dealing with paired abstractions too

Paired Abstractions Under LETs

A paired abstraction can be the first argument to a `LET`, giving

$$\text{let } (u, v) = e_1 \text{ in } e_2$$

In what follows,

- ▶ the “easy” answer for normal abstractions comes first;
- ▶ followed by the additional work needed for dealing with paired abstractions too

Aside: HOL4 also supports “restricted abstractions”, with syntax $(\lambda x :: P. \dots)$. These can also appear under `LETs`, but dealing with these is future work!

The “Obvious” Answer—A Custom Tactic

Operation:

- ▶ Scan goal for `let`-expression
- ▶ Given `LET f x`, take f 's bound variables and assert new assumption: $\vec{v} = x$
 - ▶ Justified by $\vdash P = (\forall v. (v = e) \Rightarrow P)$ and $\vdash (\forall p. P(p)) = (\forall a b. P(a, b))$
- ▶ Then `LET f x = LET f \vec{v}` , followed by β -reduction to produce just body of f .

Provisoes:

- ▶ x must not include any bound variables.
- ▶ \vec{v} must be fresh wrt the rest of the goal

Embodying the Obvious Answer in Rewrites

Much of the “custom tactic” is an implementation of the rewrite

$$\vdash P (\text{LET } f M) = (\forall v. (v = M) \Rightarrow P (f v))$$

(Rest of tactic moves implication’s antecedent into assumptions.)

Problems:

- ▶ This (higher-order) rewrite can’t be applied by standard simplification technology, as it is not an instance of a higher-order pattern. (Finding the instantiation for P is non-trivial, but doable. See Isabelle’s “splitter” technology.)
- ▶ Rewriting/splitting with the above will also mess with the user’s choice of bound variable names.

Bound Variable Name Preservation

Users pick their bound names for a reason.

Gratuitously renaming them is Sure To Annoy.

If a rewrite includes *new* bound names, as v is in

$$\vdash P (\text{LET } f \ M) = (\forall v. (v = M) \Rightarrow P (f \ v))$$

and the a, b are in

$$\vdash (\forall p. P(p)) = (\forall a \ b. P(a, b))$$

then the system has to *invent* new names, and they're not likely to be "right".

Principles of Bound Variable Name Preservation

1. Avoid rewrite theorems that introduce fresh variable names.
2. Make sure that (higher-order pattern) rewriting preserves existing variable names.

For example, rewriting with

$$\vdash ((\exists x. P(x)) \wedge Q) = (\exists x. P(x) \wedge Q)$$

shouldn't turn the actual bound variable in P into x .

Our Solution: Local, Bound-name Preserving Rewrites

Local Rewrites:

Our solution is to do `LET`-lifting with rewrites:

$$\begin{aligned} f \text{ (LET } g \text{ } M) &= \text{LET } (\lambda x. f(g(x))) \text{ } M \\ \text{(LET } f \text{ } M) \text{ } N &= \text{LET } (\lambda x. f \ x \ N) \text{ } M \end{aligned}$$

(Yes, these violate bound-name preservation for the moment.)

Features:

- ▶ Pending computation M stays pending, as `LET` heads upwards
- ▶ Normal rewriting suffices to apply these

Fixing Bound Names

Use **Bracket Abstraction**.

Instead of

$$f \text{ (LET } g \text{ } M) = \text{LET } (\lambda x. f(g(x))) \text{ } M$$

write

$$f \text{ (LET } g \text{ } M) = \text{LET } (\mathbf{B} \text{ } f \text{ } g) \text{ } M$$

Fixing Bound Names

Use **Bracket Abstraction**.

Instead of

$$f (\text{LET } g \ M) = \text{LET } (\lambda x. f(g(x))) \ M$$

write

$$f (\text{LET } g \ M) = \text{LET } (\text{B } f \ g) \ M$$

Similarly, instead of

$$(\text{LET } f \ M) \ N = \text{LET } (\lambda x. f \ x \ N) \ M$$

write

$$(\text{LET } f \ M) \ N = \text{LET } (\text{C } f \ N) \ M$$

One Combinator Theorem Needs Another

Start with

$$f \text{ (let } v = e_1 \text{ in } e_2)$$

One Combinator Theorem Needs Another

Start with

$$\begin{aligned} & f (\text{let } v = e_1 \text{ in } e_2) \\ = & f (\text{LET } (\lambda v. e_2) e_1) \end{aligned}$$

One Combinator Theorem Needs Another

Start with

$$\begin{aligned} & f \text{ (let } v = e_1 \text{ in } e_2) \\ = & f \text{ (LET } (\lambda v. e_2) e_1) \\ = & \text{LET (B } f \text{ (}\lambda v. e_2)) e_1 \\ = & ? \end{aligned}$$

One Combinator Theorem Needs Another

Start with

$$\begin{aligned} & f \text{ (let } v = e_1 \text{ in } e_2) \\ = & f \text{ (LET } (\lambda v. e_2) e_1) \\ = & \text{LET (B } f \text{ (}\lambda v. e_2\text{)) } e_1 \\ = & ? \end{aligned}$$

Use $\vdash \text{B } f \text{ (}\lambda x. g \text{ } x) = \lambda x. f \text{ (}g \text{ } x)$

One Combinator Theorem Needs Another

Start with

$$\begin{aligned} & f \text{ (let } v = e_1 \text{ in } e_2) \\ = & f \text{ (LET } (\lambda v. e_2) e_1) \\ = & \text{LET (B } f \text{ (}\lambda v. e_2)) e_1 \\ = & ? \end{aligned}$$

Use $\vdash \text{B } f \text{ (}\lambda x. g \text{ } x) = \lambda x. f \text{ (}g \text{ } x)$

$$= \text{LET } (\lambda v. f \text{ } e_2) e_1$$

One Combinator Theorem Needs Another

Start with

$$\begin{aligned} & f \text{ (let } v = e_1 \text{ in } e_2) \\ = & f \text{ (LET } (\lambda v. e_2) e_1) \\ = & \text{LET (B } f \text{ (}\lambda v. e_2)) e_1 \\ = & ? \end{aligned}$$

Use $\vdash \text{B } f \text{ (}\lambda x. g \text{ } x) = \lambda x. f \text{ (}g \text{ } x)$

$$\begin{aligned} = & \text{LET } (\lambda v. f \text{ } e_2) e_1 \\ = & \text{let } v = e_1 \text{ in } f \text{ } e_2 \end{aligned}$$

As desired!

One Combinator Theorem Needs Another

Start with

$$\begin{aligned}
 & f \text{ (let } v = e_1 \text{ in } e_2) \\
 = & f \text{ (LET } (\lambda v. e_2) e_1) \\
 = & \text{LET (B } f \text{ (} \lambda v. e_2)) e_1 \\
 = & ?
 \end{aligned}$$

$$\text{Use } \vdash \text{B } f \text{ (} \lambda x. g \text{ } x) = \lambda x. f \text{ (} g \text{ } x)$$

$$\begin{aligned}
 = & \text{LET } (\lambda v. f \text{ } e_2) e_1 \\
 = & \text{let } v = e_1 \text{ in } f \text{ } e_2
 \end{aligned}$$

As desired!

Analogous theorem for `C` is: $\vdash \text{C } (\lambda x. f \text{ } x) y = \lambda x. f \text{ } x \text{ } y$

Dealing with Pairs

If the abstraction under a `LET` is paired, then

$$f (\text{LET } g \ M) = \text{LET } (\text{B } f \ g) \ M$$

applies, but

$$\vdash \text{B } f \ (\lambda x. g \ x) = \lambda x. f \ (g \ x)$$

does not.

Need a theorem

$$\vdash \text{B } f \ (\text{UNCURRY } g) = \ ?$$

Bracket Abstraction to the Rescue Once More

`B f (UNCURRY g)`

Bracket Abstraction to the Rescue Once More

$$\begin{aligned} & \mathbf{B} \ f \ (\mathbf{UNCURRY} \ g) \\ = & \ \lambda p. \ f \ (\mathbf{UNCURRY} \ g \ p) \end{aligned}$$

Bracket Abstraction to the Rescue Once More

$$\begin{aligned} & \mathbf{B} \ f \ (\mathbf{UNCURRY} \ g) \\ = & \ \lambda p. \ f \ (\mathbf{UNCURRY} \ g \ p) \\ = & \ \lambda(a, b). \ f \ (\mathbf{UNCURRY} \ g \ (a, b)) \end{aligned}$$

Bracket Abstraction to the Rescue Once More

$$\begin{aligned} & \mathbf{B} \ f \ (\mathbf{UNCURRY} \ g) \\ = & \ \lambda p. \ f \ (\mathbf{UNCURRY} \ g \ p) \\ = & \ \lambda(a, b). \ f \ (\mathbf{UNCURRY} \ g \ (a, b)) \\ = & \ \lambda(a, b). \ f \ (g \ a \ b) \end{aligned}$$

Bracket Abstraction to the Rescue Once More

$$\begin{aligned} & \mathbf{B} \ f \ (\mathbf{UNCURRY} \ g) \\ = & \ \lambda p. \ f \ (\mathbf{UNCURRY} \ g \ p) \\ = & \ \lambda(a, b). \ f \ (\mathbf{UNCURRY} \ g \ (a, b)) \\ = & \ \lambda(a, b). \ f \ (g \ a \ b) \\ = & \ \mathbf{UNCURRY} \ (\lambda a \ b. \ f \ (g \ a \ b)) \end{aligned}$$

Bracket Abstraction to the Rescue Once More

$$\begin{aligned} & \mathbf{B} \ f \ (\mathbf{UNCURRY} \ g) \\ = & \ \lambda p. \ f \ (\mathbf{UNCURRY} \ g \ p) \\ = & \ \lambda(a, b). \ f \ (\mathbf{UNCURRY} \ g \ (a, b)) \\ = & \ \lambda(a, b). \ f \ (g \ a \ b) \\ = & \ \mathbf{UNCURRY} \ (\lambda a \ b. \ f \ (g \ a \ b)) \\ = & \ \mathbf{UNCURRY} \ (\lambda a. \ \mathbf{B} \ f \ (g \ a)) \end{aligned}$$

Bracket Abstraction to the Rescue Once More

$$\begin{aligned} & \mathbf{B} \ f \ (\mathbf{UNCURRY} \ g) \\ = & \ \lambda p. \ f \ (\mathbf{UNCURRY} \ g \ p) \\ = & \ \lambda(a, b). \ f \ (\mathbf{UNCURRY} \ g \ (a, b)) \\ = & \ \lambda(a, b). \ f \ (g \ a \ b) \\ = & \ \mathbf{UNCURRY} \ (\lambda a \ b. \ f \ (g \ a \ b)) \\ = & \ \mathbf{UNCURRY} \ (\lambda a. \ \mathbf{B} \ f \ (g \ a)) \\ = & \ \mathbf{UNCURRY} \ (\mathbf{B} \ (\mathbf{B} \ f) \ g) \end{aligned}$$

Bracket Abstraction to the Rescue Once More

$$\begin{aligned} & \mathbf{B} f (\mathbf{UNCURRY} g) \\ = & \lambda p. f (\mathbf{UNCURRY} g p) \\ = & \lambda(a, b). f (\mathbf{UNCURRY} g (a, b)) \\ = & \lambda(a, b). f (g a b) \\ = & \mathbf{UNCURRY} (\lambda a b. f (g a b)) \\ = & \mathbf{UNCURRY} (\lambda a. \mathbf{B} f (g a)) \\ = & \mathbf{UNCURRY} (\mathbf{B} (\mathbf{B} f) g) \end{aligned}$$

If g is $(\lambda a b. \dots)$, then

$$\mathbf{UNCURRY} (\mathbf{B} (\mathbf{B} f) (\lambda a b. \dots))$$

Bracket Abstraction to the Rescue Once More

$$\begin{aligned}
& \mathbf{B} \ f \ (\mathbf{UNCURRY} \ g) \\
= & \ \lambda p. \ f \ (\mathbf{UNCURRY} \ g \ p) \\
= & \ \lambda(a, b). \ f \ (\mathbf{UNCURRY} \ g \ (a, b)) \\
= & \ \lambda(a, b). \ f \ (g \ a \ b) \\
= & \ \mathbf{UNCURRY} \ (\lambda a \ b. \ f \ (g \ a \ b)) \\
= & \ \mathbf{UNCURRY} \ (\lambda a. \ \mathbf{B} \ f \ (g \ a)) \\
= & \ \mathbf{UNCURRY} \ (\mathbf{B} \ (\mathbf{B} \ f) \ g)
\end{aligned}$$

If g is $(\lambda a \ b. \dots)$, then

$$\begin{aligned}
& \mathbf{UNCURRY} \ (\mathbf{B} \ (\mathbf{B} \ f) \ (\lambda a \ b. \dots)) \\
= & \ \mathbf{UNCURRY} \ (\lambda a. \ \mathbf{B} \ f \ (\lambda b. \dots))
\end{aligned}$$

Bracket Abstraction to the Rescue Once More

$$\begin{aligned}
& \mathbf{B} \ f \ (\mathbf{UNCURRY} \ g) \\
= & \ \lambda p. \ f \ (\mathbf{UNCURRY} \ g \ p) \\
= & \ \lambda(a, b). \ f \ (\mathbf{UNCURRY} \ g \ (a, b)) \\
= & \ \lambda(a, b). \ f \ (g \ a \ b) \\
= & \ \mathbf{UNCURRY} \ (\lambda a \ b. \ f \ (g \ a \ b)) \\
= & \ \mathbf{UNCURRY} \ (\lambda a. \ \mathbf{B} \ f \ (g \ a)) \\
= & \ \mathbf{UNCURRY} \ (\mathbf{B} \ (\mathbf{B} \ f) \ g)
\end{aligned}$$

If g is $(\lambda a \ b. \dots)$, then

$$\begin{aligned}
& \mathbf{UNCURRY} \ (\mathbf{B} \ (\mathbf{B} \ f) \ (\lambda a \ b. \dots)) \\
= & \ \mathbf{UNCURRY} \ (\lambda a. \ \mathbf{B} \ f \ (\lambda b. \dots)) \\
= & \ \mathbf{UNCURRY} \ (\lambda a \ b. \ f \ (\dots))
\end{aligned}$$

Bracket Abstraction to the Rescue Once More

$$\begin{aligned}
 & \mathbf{B} \ f \ (\mathbf{UNCURRY} \ g) \\
 = & \ \lambda p. \ f \ (\mathbf{UNCURRY} \ g \ p) \\
 = & \ \lambda(a, b). \ f \ (\mathbf{UNCURRY} \ g \ (a, b)) \\
 = & \ \lambda(a, b). \ f \ (g \ a \ b) \\
 = & \ \mathbf{UNCURRY} \ (\lambda a \ b. \ f \ (g \ a \ b)) \\
 = & \ \mathbf{UNCURRY} \ (\lambda a. \ \mathbf{B} \ f \ (g \ a)) \\
 = & \ \mathbf{UNCURRY} \ (\mathbf{B} \ (\mathbf{B} \ f) \ g)
 \end{aligned}$$

If g is $(\lambda a \ b. \dots)$, then

$$\begin{aligned}
 & \mathbf{UNCURRY} \ (\mathbf{B} \ (\mathbf{B} \ f) \ (\lambda a \ b. \dots)) \\
 = & \ \mathbf{UNCURRY} \ (\lambda a. \ \mathbf{B} \ f \ (\lambda b. \dots)) \\
 = & \ \mathbf{UNCURRY} \ (\lambda a \ b. \ f \ (\dots)) \\
 = & \ \lambda(a, b). \ f \ (\dots)
 \end{aligned}$$

Wrapping Up

Once a `LET` has been pulled to the top of a term, another combinatory rewrite can turn it into an equation:

$$\text{BracketAbstract} \vdash (\text{LET } f \ M : \text{bool}) = \forall v. (v = M) \Rightarrow f \ v$$

And we're done!

See paper for

- ▶ More combinatory machine-code
- ▶ Discussions of confluence, and termination