Contexts in *reFLect*: A Theorem Proving Meta-Language

Jim Grundy	Intel Corporation, Strategic CAD Labs
Tom Melham	Oxford University, Computing Laboratory
John O'Leary	Intel Corporation, Strategic CAD Labs
Sava Krstić	Intel Corporation, Strategic CAD Labs

October 2005

Language Overview

reFLect is

- 2nd version of *FL* with reflection
- a dialect of *ML* used at Intel for applications including
  - correctness preserving design transformations
  - interactive theorem proving of design properties

Language Overview

reFL^ect is

- 2nd version of *FL* with reflection
- a dialect of *ML* used at Intel for applications including
  - correctness preserving design transformations
  - interactive theorem proving of design properties
- *reFL^ect* is typed λ-calculus +
  - A quotation mechanism, like: ⟨|`this`|⟩
  - An anti-quotation mechanism, like: `^this`

intel

Language Overview

reFL^ect is

- 2nd version of *FL* with reflection
- a dialect of *ML* used at Intel for applications including
  - correctness preserving design transformations
  - interactive theorem proving of design properties
- *reFL^ect* is typed λ-calculus +
  - A quotation mechanism, like: $\langle\!|$`this`$|\!\rangle$
  - An anti-quotation mechanism, like: `^this`
- Quoted expressions denote values of type *term*

intel

Language Overview

reFLect is

- 2nd version of *FL* with reflection
- a dialect of *ML* used at Intel for applications including
  - correctness preserving design transformations
  - interactive theorem proving of design properties
- *reFLect* is typed λ-calculus +
  - A quotation mechanism, like: ⟨|`this`|⟩
  - An anti-quotation mechanism, like: `^this`
- Quoted expressions denote values of type *term*
  - Values of type *term* are ASTs of well-typed expressions

intel

Language Overview

reFLect is

- 2nd version of *FL* with reflection
- a dialect of *ML* used at Intel for applications including
  - correctness preserving design transformations
  - interactive theorem proving of design properties
- *reFLect* is typed λ-calculus +
  - A quotation mechanism, like: $\langle\!|$`this`$|\!\rangle$
  - An anti-quotation mechanism, like: `^this`
- Quoted expressions denote values of type *term*
  - Values of type *term* are ASTs of well-typed expressions
  - $\texttt{1} + \texttt{2}$ and $\texttt{2} + \texttt{1}$ are equal, they both describe the number 3

intel

Language Overview

reFL^ect is

- 2nd version of *FL* with reflection
- a dialect of *ML* used at Intel for applications including
  - correctness preserving design transformations
  - interactive theorem proving of design properties
- *reFL^ect* is typed λ-calculus +
  - A quotation mechanism, like: $\langle\!|\texttt{this}|\!\rangle$
  - An anti-quotation mechanism, like: $\hat{}\texttt{this}$
- Quoted expressions denote values of type *term*
  - Values of type *term* are ASTs of well-typed expressions
  - $1 + 2$ and $2 + 1$ are equal, they both describe the number 3
  - $\langle\!|1 + 2|\!\rangle$ and $\langle\!|2 + 1|\!\rangle$ are not equal, they are different ASTs

Language Overview

reFLect is

- 2nd version of *FL* with reflection
- a dialect of *ML* used at Intel for applications including
  - correctness preserving design transformations
  - interactive theorem proving of design properties
- *reFLect* is typed λ-calculus +
  - A quotation mechanism, like: $\langle\!|\texttt{this}|\!\rangle$
  - An anti-quotation mechanism, like: $\hat{}\texttt{this}$
- Quoted expressions denote values of type *term*
  - Values of type *term* are ASTs of well-typed expressions
  - $1 + 2$ and $2 + 1$ are equal, they both describe the number 3
  - $\langle\!|1 + 2|\!\rangle$ and $\langle\!|2 + 1|\!\rangle$ are not equal, they are different ASTs
  - $\langle\!|\hat{}\langle\!|1|\!\rangle + 2|\!\rangle$ and $\langle\!|1 + \hat{}\langle\!|2|\!\rangle|\!\rangle$ are equal, they describe $\langle\!|1 + 2|\!\rangle$

Example

```
– letrec
  comm ⟦^x + ^y⟧        = ⟦^(comm y) + ^(comm x)⟧
| comm ⟦^f ^x⟧          = ⟦^(comm f) ^(comm x)⟧
| comm ⟦λ^p. ^b⟧        = ⟦λ^p. ^(comm b)⟧
| ...
| comm x                = x;
```

Example

```
– letrec
    comm ⟪^x + ^y⟫          = ⟪^(comm y) + ^(comm x)⟫
  | comm ⟪^f ^x⟫            = ⟪^(comm f) ^(comm x)⟫
  | comm ⟪λ^p. ^b⟫          = ⟪λ^p. ^(comm b)⟫
  | ...
  | comm x                  = x;
  comm: term→term
–
```

Example

```
- letrec
  comm ⟨|^x + ^y|⟩          = ⟨|^(comm y) + ^(comm x)|⟩
| comm ⟨|^f ^x|⟩            = ⟨|^(comm f) ^(comm x)|⟩
| comm ⟨|λ^p. ^b|⟩          = ⟨|λ^p. ^(comm b)|⟩
| ...
| comm x                    = x;
comm: term→term
- comm ⟨|y = m*x + c|⟩;
```

Example

```
- letrec
  comm ⟦^x + ^y⟧      = ⟦^(comm y) + ^(comm x)⟧
| comm ⟦^f ^x⟧        = ⟦^(comm f) ^(comm x)⟧
| comm ⟦λ^p. ^b⟧      = ⟦λ^p. ^(comm b)⟧
| ...
| comm x              = x;
comm: term→term
- comm ⟦y = m*x + c⟧;
⟦y = c + m*x⟧: term
```

The Higher Order Logic of *reFLect*

The HOL Logic

$$\left\{\begin{array}{c} \lambda - \text{calculus} \\ + \\ \text{constants: =, true, false} \\ + \\ \text{axioms, inference rules} \\ + \\ \text{definitions} \end{array}\right.$$

The Higher Order Logic of *reFLect*

The HOL Logic

λ – calculus
\+
constants: =, true, false
\+
axioms, inference rules
\+
definitions

The *reFLect* Logic

reFLect
\+
constants: =, true, false
\+
axioms, inference rules
\+
definitions

The Higher Order Logic of *reFLect*

The HOL Logic

$$\left\{\begin{array}{c}\lambda-\text{calculus}\\+\\\text{constants: =, true, false}\\+\\\text{axioms, inference rules}\\+\\\text{definitions}\end{array}\right.$$

The *reFLect* Logic

$$\left\{\begin{array}{c}\textit{reFLect}\\+\\\text{constants: =, true, false}\\+\\\text{axioms, inference rules}\\+\\\text{definitions}\end{array}\right.$$

Common to Both

- Not everything that may be discussed may be executed
  - $\texttt{let}\ \forall\ \texttt{f} = \texttt{f} = (\lambda \texttt{x}.\texttt{true})$
- Reductions in the language are valid inferences in the logic
  - If $\Lambda \rightarrow \texttt{true}$, then $\vdash \Lambda$

Levels and Their Relationships

- A deep embedding of LTL in HOL:
  0: ML
  1: HOL logic, deeply embedded in ML
  2: LTL logic, deeply embedded in HOL

  Use the prover (level 0 program) to reason about what HOL functions (level 1) do to LTL expressions (level 2)

Levels and Their Relationships

- A deep embedding of LTL in HOL:

  0: ML
  1: HOL logic, deeply embedded in ML
  2: LTL logic, deeply embedded in HOL

  Use the prover (level 0 program) to reason about what HOL functions (level 1) do to LTL expressions (level 2)

- A shallow embedding of LTL in *reFLect*

  0: *reFLect*
  1: quoted *reFLect* expressions
  2: twice quoted *reFLect* expressions

  Use the prover (level 0 program) to reason about what *reFLect* functions (level 1) do to *reFLect* expressions (level 2)

Levels are Separate

We want the same relationship between level n and $n+1$ *reFLect* expressions as between ML and HOL (or between HOL and LTL, the deeply embedded language)

- Level n expressions can manipulate level $n+1$ expressions

Levels are Separate

We want the same relationship between level n and $n+1$ *reFLect* expressions as between ML and HOL (or between HOL and LTL, the deeply embedded language)

- Level n expressions can manipulate level $n+1$ expressions
- Level n expressions don't interpret those above level $n+1$ (We don't implement LTL reasoning directly in ML.)

intel

Levels are Separate

We want the same relationship between level n and $n+1$ *reFLect* expressions as between ML and HOL (or between HOL and LTL, the deeply embedded language)

- Level n expressions can manipulate level $n+1$ expressions
- Level n expressions don't interpret those above level $n+1$
  (We don't implement LTL reasoning directly in ML.)
- They do not, usually, become level $n+1$ expressions
  (ML does not become HOL)

Levels are Separate

We want the same relationship between level n and $n+1$ *reFLect* expressions as between ML and HOL
(or between HOL and LTL, the deeply embedded language)

- Level n expressions can manipulate level $n+1$ expressions
- Level n expressions don't interpret those above level $n+1$
  (We don't implement LTL reasoning directly in ML.)
- They do not, usually, become level $n+1$ expressions
  (ML does not become HOL)
- Level $n+1$ expressions do not, usually, become level n expressions
  (HOL does not become ML)

Levels are Separate

We want the same relationship between level n and $n + 1$ *reFLect* expressions as between ML and HOL
(or between HOL and LTL, the deeply embedded language)

- Level n expressions can manipulate level $n + 1$ expressions
- Level n expressions don't interpret those above level $n + 1$
  (We don't implement LTL reasoning directly in ML.)
- They do not, usually, become level $n + 1$ expressions
  (ML does not become HOL)
- Level $n + 1$ expressions do not, usually, become level n expressions
  (HOL does not become ML)
- Variables are bound within a level, not across levels
  - Want $\langle\!| x |\!\rangle$ different to $\langle\!| 1 |\!\rangle$
  - Want usual quantifier rules
  - Do not want this ☛

$$\dfrac{\dfrac{\vdash \neg(\langle\!| x |\!\rangle = \langle\!| 1 |\!\rangle)}{\vdash \forall x.\, \neg(\langle\!| x |\!\rangle = \langle\!| 1 |\!\rangle)}\,[\forall I]}{\vdash \neg(\langle\!| 1 |\!\rangle = \langle\!| 1 |\!\rangle)}\,[\forall E]$$

intel

reFLect Abstract Syntax

$$
\begin{array}{lcll}
\Lambda, M, N & ::= & k & \text{– Constant} \\
& | & v & \text{– Variable} \\
& | & \lambda\Lambda.\, M & \text{– Abstraction} \\
& | & \lambda\Lambda.\, M \mid N & \text{– Alternation} \\
& | & \Lambda M & \text{– Application} \\
& | & \langle\!|\Lambda|\!\rangle & \text{– Quotation} \\
& | & \hat{}\Lambda & \text{– Anti-quotation}
\end{array}
$$

Note:

- Arbitrary expressions may be patterns
- Lambda abstractions may have match alternatives
- Omitting whole story about type annotations checking

intel

reFLect Abstract Syntax

Λ, M, N	$::=$	k	– Constant
	$\mid$	v	– Variable
	$\mid$	$\lambda\Lambda\,.\,M$	– Abstraction
	$\mid$	$\lambda\Lambda\,.\,M \mid N$	– Alternation
	$\mid$	ΛM	– Application
	$\mid$	$\langle\!\vert \Lambda \vert\!\rangle$	– Quotation
	$\mid$	$\hat{}\Lambda$	– Anti-quotation

On the path from the root of an AST to some subexpression:

- the level of the subexpression is the number of quotations on the path − the number of antiquotes
- an expression is well formed if no subexpression has negative level

intel

We Don't Do This

We *could* make values of *term* appear as if defined as follows:

```
lettype term = VAR string              // v
             | CONST val               // k
             | APPLY term term         // ΛM
             | ABS term term           // λΛ.M
             | ALT term term term      // λΛ.M | N
             | QUOTE term              // ⟨|Λ|⟩
             | ANTIQ term              // ^Λ
```

We Don’t Do This

We *could* make values of *term* appear as if defined as follows:

```
lettype term = VAR string             // v
             | CONST val              // k
             | APPLY term term        // ΛM
             | ABS term term          // λΛ.M
             | ALT term term term     // λΛ.M | N
             | QUOTE term             // ⦇Λ⦈
             | ANTIQ term             // ^Λ
```

Consider how to find the free variables in a term

We Don't Do This

We *could* make values of *term* appear as if defined as follows:

```
lettype term = VAR string              // v
             | CONST val               // k
             | APPLY term term         // ΛM
             | ABS term term           // λΛ.M
             | ALT term term term      // λΛ.M|N
             | QUOTE term              // ⦇Λ⦈
             | ANTIQ term              // ˆΛ
```

Consider how to find the free variables in a term

- just those at level 0
- variables at higher level are somebody else's problem

Example: What We Don't Do

```
let frees trm =
  letrec
    f 0     (VAR nam)             = {VAR nam}
  | f (n+1) (VAR nam)             = {}
  | f n     (CONST idn)           = {}
  | f n     (APP fun arg)         =
      f n fun U f n arg
  | f 0     (ABS pat bod)         =
      f 0 bod - f 0 pat
  | f (n+1) (ABS pat bod)         =
      f (n+1) pat U f (n+1) bod
  ...
  | f n     (QUOTE quo)           = f (n+1) quo
  | f (n+1) (ANTIQ ant)           = f n ant
  in
    f 0 trm;
```

Why Don't We Do It?

- The definition of `frees` was overly complex
  - It had to be careful to remember what to look at and what not to
  - It traversed regions it didn't need to look at

Why Don't We Do It?

- The definition of `frees` was overly complex
  - It had to be careful to remember what to look at and what not to
  - It traversed regions it didn't need to look at
- `QUOTE` and `ANTIQ` move expressions up and down levels without restriction

Why Don’t We Do It?

- The definition of `frees` was overly complex
  - It had to be careful to remember what to look at and what not to
  - It traversed regions it didn’t need to look at
- `QUOTE` and `ANTIQ` move expressions up and down levels without restriction
- Programs can, and must, inspect arbitrarily higher levels

What We Do Instead: Contexts

$$
\begin{array}{lcll}
\Lambda, M, N & ::= & \ldots & \textit{– as in terms} \\
 & | & \sqcup & \textit{– hole}
\end{array}
$$

A context is well formed only if:

- all holes are at level 0
- no portion of the context has negative level

$\langle\!|\,\hat{}\,\sqcup + 1\,|\!\rangle \qquad (\sqcup + \sqcup) \qquad \langle\!|\,\sqcup + 1\,|\!\rangle \qquad (\hat{}\,\mathtt{x} + \sqcup)$

What We Do Instead: Contexts

$$
\begin{array}{lcll}
\Lambda, M, N & ::= & \ldots & \text{– \textit{as in terms}} \\
 & | & \text{\textvisiblespace} & \text{– \textit{hole}}
\end{array}
$$

A context is well formed only if:

- all holes are at level 0
- no portion of the context has negative level

✔ $\llparenthesis\,\hat{}\,\text{\textvisiblespace} + 1\,\rrparenthesis$ $\quad (\text{\textvisiblespace} + \text{\textvisiblespace})$ $\quad \llparenthesis\,\text{\textvisiblespace} + 1\,\rrparenthesis$ $\quad (\hat{}\,\mathtt{x} + \text{\textvisiblespace})$

What We Do Instead: Contexts

$$
\begin{array}{lcll}
\Lambda, M, N & ::= & \ldots & \textit{– as in terms} \\
 & | & \sqcup & \textit{– hole}
\end{array}
$$

A context is well formed only if:

- all holes are at level 0
- no portion of the context has negative level

✔ $\langle\!|\,\hat{}\sqcup + 1\,|\!\rangle$ ✔ $(\sqcup + \sqcup)$ $\langle\!|\,\sqcup + 1\,|\!\rangle$ $(\hat{}x + \sqcup)$

What We Do Instead: Contexts

$$\Lambda, M, N \quad ::= \quad \ldots \qquad \text{– as in terms}$$
$$\quad | \quad \llcorner\!\lrcorner \qquad \text{– hole}$$

A context is well formed only if:

- all holes are at level 0
- no portion of the context has negative level

✔ $⟅\hat{}\llcorner\!\lrcorner + 1⟆$ ✔ $(\llcorner\!\lrcorner + \llcorner\!\lrcorner)$ ✘ $⟅\llcorner\!\lrcorner + 1⟆$ $(\hat{}\mathrm{x} + \llcorner\!\lrcorner)$

What We Do Instead: Contexts

$$
\begin{array}{llll}
\Lambda, M, N & ::= & \ldots & \text{– \textit{as in terms}} \\
 & | & \text{␣} & \text{– \textit{hole}}
\end{array}
$$

A context is well formed only if:

- all holes are at level 0
- no portion of the context has negative level

✔ $\langle\!|\,\hat{}\,\text{␣} + 1\,|\!\rangle$ ✔ $(\text{␣} + \text{␣})$ ✘ $\langle\!|\,\text{␣} + 1\,|\!\rangle$ ✘ $(\hat{}\,\mathrm{x} + \text{␣})$

What We Do Instead: Contexts

$$
\begin{array}{rcll}
\Lambda, M, N & ::= & \ldots & \text{– \textit{as in terms}} \\
 & | & \text{␣} & \text{– \textit{hole}}
\end{array}
$$

A context is well formed only if:

- all holes are at level 0
- no portion of the context has negative level

✔ $\langle\!|\,\hat{}\,\text{␣} + 1\,|\!\rangle$ ✔ $(\text{␣} + \text{␣})$ ✘ $\langle\!|\,\text{␣} + 1\,|\!\rangle$ ✘ $(\hat{}\,x + \text{␣})$

We assume the usual hole filling operation on contexts

What We Do Instead: Contexts

$$\begin{array}{llll} \Lambda, M, N & ::= & \ldots & \text{– \textit{as in terms}} \\ & | & \llcorner\!\lrcorner & \text{– \textit{hole}} \end{array}$$

A context is well formed only if:

- all holes are at level 0
- no portion of the context has negative level

✔ $\langle\!|\, \hat{}\ \llcorner\!\lrcorner + 1 \,|\!\rangle$ ✔ $(\llcorner\!\lrcorner + \llcorner\!\lrcorner)$ ✘ $\langle\!|\, \llcorner\!\lrcorner + 1 \,|\!\rangle$ ✘ $(\hat{}\,x + \llcorner\!\lrcorner)$

We assume the usual hole filling operation on contexts

$(\llcorner\!\lrcorner + \llcorner\!\lrcorner)[2, 1]$ is

What We Do Instead: Contexts

$$
\begin{array}{llll}
\Lambda, M, N & ::= & \ldots & \textit{– as in terms} \\
& | & \textvisiblespace & \textit{– hole}
\end{array}
$$

A context is well formed only if:

- all holes are at level 0
- no portion of the context has negative level

✔ $\langle\!|\, \hat{}\,\textvisiblespace + 1 \,|\!\rangle$ ✔ $(\textvisiblespace + \textvisiblespace)$ ✘ $\langle\!|\, \textvisiblespace + 1 \,|\!\rangle$ ✘ $(\hat{}\,x + \textvisiblespace)$

We assume the usual hole filling operation on contexts

$(\textvisiblespace + \textvisiblespace)[2,1]$ is $2 + 1$

What We Do Instead: Contexts

$$
\begin{array}{rcll}
\Lambda, M, N & ::= & \ldots & \textit{– as in terms} \\
 & \mid & \sqcup & \textit{– hole}
\end{array}
$$

A context is well formed only if:

- all holes are at level 0
- no portion of the context has negative level

✔ $\langle\!| \hat{}\sqcup + 1 |\!\rangle$ ✔ $(\sqcup + \sqcup)$ ✘ $\langle\!| \sqcup + 1 |\!\rangle$ ✘ $(\hat{}x + \sqcup)$

We assume the usual hole filling operation on contexts

$(\sqcup + \sqcup)[2, 1]$ is $2 + 1$

$\langle\!| \hat{}\sqcup + 1 |\!\rangle [\langle\!| 2 |\!\rangle]$ is

What We Do Instead: Contexts

$$\Lambda, M, N \quad ::= \quad \ldots \qquad \text{– as in terms}$$
$$\qquad\qquad \mid \quad \llcorner\!\lrcorner \qquad \text{– hole}$$

A context is well formed only if:

- all holes are at level 0
- no portion of the context has negative level

✔ $\langle\!|\,\hat{}\,\llcorner\!\lrcorner + 1\,|\!\rangle$ ✔ $(\llcorner\!\lrcorner + \llcorner\!\lrcorner)$ ✘ $\langle\!|\,\llcorner\!\lrcorner + 1\,|\!\rangle$ ✘ $(\hat{}\,\mathtt{x} + \llcorner\!\lrcorner)$

We assume the usual hole filling operation on contexts

$$(\llcorner\!\lrcorner + \llcorner\!\lrcorner)[2,1] \quad \text{is} \quad 2 + 1$$
$$\langle\!|\,\hat{}\,\llcorner\!\lrcorner + 1\,|\!\rangle[\langle\!|\,2\,|\!\rangle] \quad \text{is} \quad \langle\!|\,\hat{}\langle\!|\,2\,|\!\rangle + 1\,|\!\rangle$$

The Context Property

All well-formed expressions of the form $\langle\!|\Lambda|\!\rangle$
have a *unique* factorization into:

- a well-formed context $\mathcal{C}$
- a list of well-formed expressions $M_1, \ldots M_n$

such that $\langle\!|\mathcal{C}[\hat{}M_1, \ldots \hat{}M_n]|\!\rangle$ is $\langle\!|\Lambda|\!\rangle$

Example

Expression	Factors

The Context Property

All well-formed expressions of the form $\langle\!|\Lambda|\!\rangle$ have a *unique* factorization into:

- a well-formed context $\mathcal{C}$
- a list of well-formed expressions $M_1, \ldots M_n$

such that $\langle\!|\mathcal{C}[\hat{}M_1, \ldots \hat{}M_n]|\!\rangle$ is $\langle\!|\Lambda|\!\rangle$

Example

Expression	Factors
$\langle\!|\hat{}x + \hat{}y|\!\rangle$	

intel

The Context Property

All well-formed expressions of the form $\langle\!|\Lambda|\!\rangle$
have a *unique* factorization into:

- a well-formed context $\mathcal{C}$
- a list of well-formed expressions $M_1, \ldots M_n$

such that $\langle\!|\mathcal{C}[\hat{}M_1, \ldots \hat{}M_n]|\!\rangle$ is $\langle\!|\Lambda|\!\rangle$

Example

Expression	Factors	
$\langle\!|\hat{}x + \hat{}y|\!\rangle$	$(\sqcup + \sqcup)$	$[x, y]$

The Context Property

All well-formed expressions of the form $\langle\!|\Lambda|\!\rangle$
have a *unique* factorization into:

- a well-formed context $\mathcal{C}$
- a list of well-formed expressions $M_1, \ldots M_n$

such that $\langle\!|\mathcal{C}[\hat{}M_1, \ldots \hat{}M_n]|\!\rangle$ is $\langle\!|\Lambda|\!\rangle$

Example

Expression	Factors	
$\langle\!|\hat{}x + \hat{}y|\!\rangle$	$(\_ + \_)$	$[x, y]$
$\langle\!|x + y|\!\rangle$		

The Context Property

All well-formed expressions of the form $\langle\!|\Lambda|\!\rangle$ have a *unique* factorization into:

- a well-formed context $\mathcal{C}$
- a list of well-formed expressions $M_1, \ldots M_n$

such that $\langle\!|\mathcal{C}[\hat{}M_1, \ldots \hat{}M_n]|\!\rangle$ is $\langle\!|\Lambda|\!\rangle$

Example

Expression	Factors	
$\langle\!|\hat{}x + \hat{}y|\!\rangle$	$(\llcorner\!\lrcorner + \llcorner\!\lrcorner)$	$[x, y]$
$\langle\!|x + y|\!\rangle$	$(x + y)$	$[]$

intel

The Context Property

All well-formed expressions of the form $\langle\!|\Lambda|\!\rangle$ have a *unique* factorization into:

- a well-formed context $\mathcal{C}$
- a list of well-formed expressions $M_1, \ldots M_n$

such that $\langle\!|\mathcal{C}[\hat{}M_1, \ldots \hat{}M_n]|\!\rangle$ is $\langle\!|\Lambda|\!\rangle$

Example

Expression	Factors	
$\langle\!|\hat{}x + \hat{}y|\!\rangle$	$(\llcorner + \llcorner)$	$[x, y]$
$\langle\!|x + y|\!\rangle$	$(x + y)$	$[]$
$\langle\!|\hat{}x + \hat{}\langle\!|y|\!\rangle|\!\rangle$		

intel

The Context Property

All well-formed expressions of the form $\langle\!|\Lambda|\!\rangle$ have a *unique* factorization into:

- a well-formed context $\mathcal{C}$
- a list of well-formed expressions $M_1, \ldots M_n$

such that $\langle\!|\mathcal{C}[\hat{\ } M_1, \ldots \hat{\ } M_n]|\!\rangle$ is $\langle\!|\Lambda|\!\rangle$

Example

Expression	Factors	
$\langle\!|\hat{\ } x + \hat{\ } y|\!\rangle$	$(\llcorner + \llcorner)$	$[x, y]$
$\langle\!|x + y|\!\rangle$	$(x + y)$	$[]$
$\langle\!|\hat{\ } x + \hat{\ }\langle\!|y|\!\rangle|\!\rangle$	$(\llcorner + \llcorner)$	$[x, \langle\!|y|\!\rangle]$

intel

The Context Property

All well-formed expressions of the form $\llparenthesis \Lambda \rrparenthesis$ have a *unique* factorization into:

- a well-formed context $\mathcal{C}$
- a list of well-formed expressions $M_1, \ldots M_n$

such that $\llparenthesis \mathcal{C}[\hat{\ } M_1, \ldots \hat{\ } M_n] \rrparenthesis$ is $\llparenthesis \Lambda \rrparenthesis$

Example

Expression	Factors	
$\llparenthesis \hat{\ }x + \hat{\ }y \rrparenthesis$	$(\sqcup + \sqcup)$	$[x, y]$
$\llparenthesis x + y \rrparenthesis$	$(x + y)$	$[]$
$\llparenthesis \hat{\ }x + \hat{\ }\llparenthesis y \rrparenthesis \rrparenthesis$	$(\sqcup + \sqcup)$	$[x, \llparenthesis y \rrparenthesis]$
$\llparenthesis \hat{\ }f\ \llparenthesis \hat{\ }\hat{\ }x + \hat{\ }\hat{\ }y \rrparenthesis \rrparenthesis$		

The Context Property

All well-formed expressions of the form $\llparenthesis \Lambda \rrparenthesis$ have a *unique* factorization into:

- a well-formed context $\mathcal{C}$
- a list of well-formed expressions $M_1, \ldots M_n$

such that $\llparenthesis \mathcal{C}[\hat{\ } M_1, \ldots \hat{\ } M_n] \rrparenthesis$ is $\llparenthesis \Lambda \rrparenthesis$

Example

Expression	Factors	
$\llparenthesis \hat{\ } x + \hat{\ } y \rrparenthesis$	$(\llcorner\!\lrcorner + \llcorner\!\lrcorner)$	$[x, y]$
$\llparenthesis x + y \rrparenthesis$	$(x + y)$	$[]$
$\llparenthesis \hat{\ } x + \hat{\ } \llparenthesis y \rrparenthesis \rrparenthesis$	$(\llcorner\!\lrcorner + \llcorner\!\lrcorner)$	$[x, \llparenthesis y \rrparenthesis]$
$\llparenthesis \hat{\ } f \ \llparenthesis \hat{\ }\hat{\ } x + \hat{\ }\hat{\ } y \rrparenthesis \rrparenthesis$	$(\llcorner\!\lrcorner \ \llparenthesis \hat{\ } \llcorner\!\lrcorner + \hat{\ } \llcorner\!\lrcorner \rrparenthesis)$	$[f, x, y]$

A Context Centric Term View

```
lettype term
   = VAR string   |   CONST val     // v | k
   | APPLY term term                // Λ M
   | ABS term term                  // λΛ. M
   | ALT term term term             // λΛ. M
   | QUOTE context (term list)      // ⦇C[^Λ1,...^Λn]⦈
```

- No term ever changes level with these constructions

A Context Centric Term View

```
lettype term
  = VAR string  |  CONST val     // $v \mid k$
  | APPLY term term              // $\Lambda\ M$
  | ABS term term                // $\lambda\Lambda.\ M$
  | ALT term term term           // $\lambda\Lambda.\ M$
  | QUOTE context (term list)    // $\llparenthesis \mathcal{C}[\hat{}\Lambda_1, \ldots \hat{}\Lambda_n] \rrparenthesis$
```

- No term ever changes level with these constructions
- From level n I can construct any level $n+1$ expression I want

A Context Centric Term View

```
lettype term
   = VAR string   |   CONST val   // v | k
   | APPLY term term              // Λ M
   | ABS term term                // λΛ. M
   | ALT term term term           // λΛ. M
   | QUOTE context (term list)    // ⦇C[ˆΛ1,...ˆΛn]⦈
```

- No term ever changes level with these constructions
- From level n I can construct any level $n+1$ expression I want
- All I can do with expressions above $n+1$ is access the $n+1$ subexpressions

Free Variables Revisited

```
letrec
  frees (VAR nam)                = {VAR nam}
| frees (CONST idn)              = {}
| frees (APP fun arg)            =
    frees fun U frees arg
| frees (ABS pat bod)            =
    frees bod - frees pat
| frees (ALT pat bod alt)        =
    (frees bod - frees pat) U (frees alt)
| frees (QUOTE ctx tms)          =
    fold (U) {} (map frees tms);
```

Free Variables Revisited

```
letrec
   frees (VAR nam)                 = {VAR nam}
 | frees (CONST idn)               = {}
 | frees (APP fun arg)             =
      frees fun U frees arg
 | frees (ABS pat bod)             =
      frees bod - frees pat
 | frees (ALT pat bod alt)         =
      (frees bod - frees pat) U (frees alt)
 | frees (QUOTE ctx tms)           =
      fold (U) {} (map frees tms);
```

Contexts hide what you don’t to see behind an SEP field.

- no need to for the . . . to fit it now

Free Variables Revisited

```
letrec
  frees (VAR nam)                    = {VAR nam}
| frees (CONST idn)                  = {}
| frees ⟨|^fun ^arg|⟩               =
    frees fun ∪ frees arg
| frees ⟨|λ^abs. ^bod|⟩             =
    frees bod − frees pat
| frees ⟨|λ^pat. ^bod | ^alt|⟩ =
    (frees bod − frees pat) ∪ (frees alt)
| frees (QUOTE ctx tms)              =
    fold (∪) {} (map frees tms);
```

Contexts hide what you don't to see behind an SEP field.

- no need to for the . . . to fit it now

Implementing *reFLect*

Consider how to write an evaluator for terms in *reFLect*.

```
eval: term → term
```

- Regular language features 'easy', let's assume done
  - `eval ⟪(λ[x,y]. x + y) [1,2]⟫;`

Implementing *reFLect*

Consider how to write an evaluator for terms in *reFLect*.

```
eval: term → term
```

- Regular language features 'easy', let's assume done

```
- eval ⟪(λ[x,y]. x + y) [1,2]⟫;
⟪3⟫: term
```

Implementing *reFLect*

Consider how to write an evaluator for terms in *reFLect*.

```
eval: term → term
```

- Regular language features 'easy', let's assume done
  - `eval ⟪(λ[x,y]. x + y) [1,2]⟫;`

    `⟪3⟫: term`
- How do we do anti-quote based term construction?
  - `eval ⟪⟪^(fst (⟪1⟫,⟪2⟫)) + ^⟪3⟫⟫⟫;`

intel

Implementing *reFLect*

Consider how to write an evaluator for terms in *reFLect*.

```
eval: term → term
```

- Regular language features 'easy', let's assume done

```
- eval ⟪(λ[x,y]. x + y) [1,2]⟫;
⟪3⟫: term
```

- How do we do anti-quote based term construction?

```
- eval ⟪⟪ˆ(fst (⟪1⟫,⟪2⟫)) + ˆ⟪3⟫⟫⟫;
⟪⟪1 + 3⟫⟫: term
```

intel

Implementing *reFLect*

Consider how to write an evaluator for terms in *reFLect*.

```
eval: term → term
```

- Regular language features 'easy', let's assume done

```
- eval ⟨|(λ[x,y]. x + y) [1,2]|⟩;
⟨|3|⟩: term
```

- How do we do anti-quote based term construction?

```
- eval ⟨|⟨|^(fst (⟨|1|⟩,⟨|2|⟩)) + ^⟨|3|⟩|⟩|⟩;
⟨|⟨|1 + 3|⟩|⟩: term
```

intel

Implementing *reFLect*

Consider how to write an evaluator for terms in *reFLect*.

```
eval: term → term
```

- Regular language features ‘easy’, let’s assume done
  - `eval ⟪(λ[x,y]. x + y) [1,2]⟫;`

    `⟪3⟫: term`
- How do we do anti-quote based term construction?
  - `eval ⟪⟪^(fst (⟪1⟫,⟪2⟫)) + ^⟪3⟫⟫⟫;`

    `⟪⟪1 + 3⟫⟫: term`
- How do we do anti-quote based term destruction?
  - `eval ⟪(λ⟪^x + ^y⟫. x) ⟪1 + 2⟫⟫;`

Implementing *reFLect*

Consider how to write an evaluator for terms in *reFLect*.

```
eval: term → term
```

- Regular language features 'easy', let's assume done

```
- eval ⟪(λ[x,y]. x + y) [1,2]⟫;
⟪3⟫: term
```

- How do we do anti-quote based term construction?

```
- eval ⟪⟪^(fst (⟪1⟫,⟪2⟫)) + ^⟪3⟫⟫⟫;
⟪⟪1 + 3⟫⟫: term
```

- How do we do anti-quote based term destruction?

```
- eval ⟪(λ⟪^x + ^y⟫. x) ⟪1 + 2⟫⟫;
⟪⟪1⟫⟫: term
```

Filling Context Holes

We require the following primitive function, to implement eval:

```
fill: context → term list → term
```

This is a version of the primitive context hole filling operation

```
- c;
(␣ + ␣): context
- fill c [⟪⟪1⟫⟫, ⟪⟪2⟫⟫];
```

Filling Context Holes

We require the following primitive function, to implement eval:

```
fill:  context → term list → term
```

This is a version of the primitive context hole filling operation

```
- c;
(␣ + ␣): context
- fill c [⟪⟪1⟫⟫, ⟪⟪2⟫⟫];
⟪⟪1 + 2⟫⟫: term
```

Filling Context Holes

We require the following primitive function, to implement eval:

```
    fill:  context → term list → term
```

This is a version of the primitive context hole filling operation

```
- c;
(␣ + ␣): context
- fill c [⟪⟪1⟫⟫, ⟪⟪2⟫⟫];
⟪⟪1 + 2⟫⟫: term
```

`fill` is similar to `QUOTE`:
but removes quotes, doesn't add anti-quote to balance levels

```
- QUOTE c [⟪⟪1⟫⟫, ⟪⟪2⟫⟫];
```

Filling Context Holes

We require the following primitive function, to implement eval:

```
fill:  context → term list → term
```

This is a version of the primitive context hole filling operation

```
- c;
(␣ + ␣): context
- fill c [⟪⟪1⟫⟫, ⟪⟪2⟫⟫];
⟪⟪1 + 2⟫⟫: term
```

`fill` is similar to `QUOTE`:
but removes quotes, doesn't add anti-quote to balance levels

```
- QUOTE c [⟪⟪1⟫⟫, ⟪⟪2⟫⟫];
⟪⟪^⟪1⟫ + ^⟪2⟫⟫⟫:term
```

Implementing Anti-quote Based Term Construction

```
letrec eval (QUOTE ctx tms) =
    fill c (map eval tms)
     ...;

  eval ⟪⟪^(fst (⟪1⟫,⟪2⟫)) + ^⟪3⟫⟫⟫
=




  ⟪⟪1 + 3⟫⟫
```

Implementing Anti-quote Based Term Construction

```
letrec eval (QUOTE ctx tms) =
     fill c (map eval tms)
      ...;

   eval ⟪⟨^(fst (⟨1⟩,⟨2⟩)) + ^⟨3⟩⟩⟫
= eval (QUOTE (␣ + ␣) [⟪fst (⟨1⟩,⟨2⟩)⟫, ⟪⟨3⟩⟫])
= fill (␣ + ␣)
    (map eval [⟪fst (⟨1⟩,⟨2⟩)⟫, ⟪⟨3⟩⟫])
= fill (␣ + ␣)
    [eval ⟪fst (⟨1⟩,⟨2⟩)⟫, eval ⟪⟨3⟩⟫]
= fill (␣ + ␣) [⟪⟨1⟩⟫, ⟪⟨3⟩⟫]
= ⟪⟨1 + 3⟩⟫
```

Implementing Anti-quote Based Term Construction

```
letrec eval (QUOTE ctx tms) =
    fill c (map eval tms)
     ...;

  eval ⟪⟪^(fst (⟪1⟫,⟪2⟫)) + ^⟪3⟫⟫⟫
= eval (QUOTE (␣ + ␣) [⟪fst (⟪1⟫,⟪2⟫)⟫, ⟪⟪3⟫⟫])
= fill (␣ + ␣)
     (map eval [⟪fst (⟪1⟫,⟪2⟫)⟫, ⟪⟪3⟫⟫])
= fill (␣ + ␣)
     [eval ⟪fst (⟪1⟫,⟪2⟫)⟫, eval ⟪⟪3⟫⟫]
= fill (␣ + ␣) [⟪⟪1⟫⟫, ⟪⟪3⟫⟫]
= ⟪⟪1 + 3⟫⟫
```

Implementing Anti-quote Based Term Construction

```
letrec eval (QUOTE ctx tms) =
    fill c (map eval tms)
     ...;

  eval ⟪⟪^(fst (⟪1⟫,⟪2⟫)) + ^⟪3⟫⟫⟫
= eval (QUOTE (␣ + ␣) [⟪fst (⟪1⟫,⟪2⟫)⟫, ⟪⟪3⟫⟫])
= fill (␣ + ␣)
     (map eval [⟪fst (⟪1⟫,⟪2⟫)⟫, ⟪⟪3⟫⟫])
= fill (␣ + ␣)
     [eval ⟪fst (⟪1⟫,⟪2⟫)⟫, eval ⟪⟪3⟫⟫]
= fill (␣ + ␣) [⟪⟪1⟫⟫, ⟪⟪3⟫⟫]
= ⟪⟪1 + 3⟫⟫
```

Implementing Anti-quote Based Term Construction

```
letrec eval (QUOTE ctx tms) =
    fill c (map eval tms)
     ...;

  eval ⟪⟪^(fst (⟪1⟫,⟪2⟫)) + ^⟪3⟫⟫⟫
= eval (QUOTE (␣ + ␣) [⟪fst (⟪1⟫,⟪2⟫)⟫, ⟪⟪3⟫⟫])
= fill (␣ + ␣)
    (map eval [⟪fst (⟪1⟫,⟪2⟫)⟫, ⟪⟪3⟫⟫])
= fill (␣ + ␣)
    [eval ⟪fst (⟪1⟫,⟪2⟫)⟫, eval ⟪⟪3⟫⟫]
= fill (␣ + ␣) [⟪⟪1⟫⟫, ⟪⟪3⟫⟫]
= ⟪⟪1 + 3⟫⟫
```

Implementing Anti-quote Based Term Construction

```
letrec eval (QUOTE ctx tms) =
    fill c (map eval tms)
     ...;

  eval ⟪⟪^(fst (⟪1⟫,⟪2⟫)) + ^⟪3⟫⟫⟫
= eval (QUOTE (␣ + ␣) [⟪fst (⟪1⟫,⟪2⟫)⟫, ⟪⟪3⟫⟫])
= fill (␣ + ␣)
     (map eval [⟪fst (⟪1⟫,⟪2⟫)⟫, ⟪⟪3⟫⟫])
= fill (␣ + ␣)
     [eval ⟪fst (⟪1⟫,⟪2⟫)⟫, eval ⟪⟪3⟫⟫]
= fill (␣ + ␣) [⟪⟪1⟫⟫, ⟪⟪3⟫⟫]
= ⟪⟪1 + 3⟫⟫
```

Implementing Anti-quote Based Term Construction

```
letrec eval (QUOTE ctx tms) =
    fill c (map eval tms)
     ...;

  eval ⟪⟪^(fst (⟪1⟫,⟪2⟫)) + ^⟪3⟫⟫⟫
= eval (QUOTE (␣ + ␣) [⟪fst (⟪1⟫,⟪2⟫)⟫, ⟪⟪3⟫⟫])
= fill (␣ + ␣)
      (map eval [⟪fst (⟪1⟫,⟪2⟫)⟫, ⟪⟪3⟫⟫])
= fill (␣ + ␣)
      [eval ⟪fst (⟪1⟫,⟪2⟫)⟫, eval ⟪⟪3⟫⟫]
= fill (␣ + ␣) [⟪⟪1⟫⟫, ⟪⟪3⟫⟫]
= ⟪⟪1 + 3⟫⟫
```

Implementing Anti-quote Based Term Construction

```
letrec eval (QUOTE ctx tms) =
     fill c (map eval tms)
      ...;

   eval ⟨|⟪^(fst (⟪1⟫,⟪2⟫)) + ^⟪3⟫⟫|⟩
 = eval (QUOTE (␣ + ␣) [⟨|fst (⟪1⟫,⟪2⟫)|⟩, ⟨|⟪3⟫|⟩])
 = fill (␣ + ␣)
      (map eval [⟨|fst (⟪1⟫,⟪2⟫)|⟩, ⟨|⟪3⟫|⟩])
 = fill (␣ + ␣)
      [eval ⟨|fst (⟪1⟫,⟪2⟫)|⟩, eval ⟨|⟪3⟫|⟩]
 = fill (␣ + ␣) [⟨|⟪1⟫|⟩, ⟨|⟪3⟫|⟩]
 = ⟨|⟪1 + 3⟫|⟩
```

Pattern Matching Contexts

We require the following primitive function, to implement eval:

```
match: context → term → term list
```

For any context `c`, `match c` inverts `fill c`

```
- c;
(␣ + ␣): context
- match c ⟨⟨1 + 2⟩⟩;
```

Pattern Matching Contexts

We require the following primitive function, to implement eval:

```
match: context → term → term list
```

For any context `c`, `match c` inverts `fill c`

```
- c;
(␣ + ␣): context
- match c ⟪⟪1 + 2⟫⟫;
[⟪⟪1⟫⟫, ⟪⟪2⟫⟫]: term list
- match c ⟪⟪^x + ^y⟫⟫;
```

Pattern Matching Contexts

We require the following primitive function, to implement eval:

```
match: context → term → term list
```

For any context `c`, `match c` inverts `fill c`

```
- c;
(␣ + ␣): context
- match c ⟨|⟨|1 + 2|⟩|⟩;
[⟨|⟨|1|⟩|⟩, ⟨|⟨|2|⟩|⟩]: term list
- match c ⟨|⟨|^x + ^y|⟩|⟩;
[⟨|⟨|^x|⟩|⟩, ⟨|⟨|^y|⟩|⟩]: term list
- match c ⟨|⟨|1 - 2|⟩|⟩;
```

Pattern Matching Contexts

We require the following primitive function, to implement eval:

```
match: context → term → term list
```

For any context `c`, `match c` inverts `fill c`

```
- c;
(␣ + ␣): context
- match c ⟨⟨1 + 2⟩⟩;
[⟨⟨1⟩⟩, ⟨⟨2⟩⟩]: term list
- match c ⟨⟨^x + ^y⟩⟩;
[⟨⟨^x⟩⟩, ⟨⟨^y⟩⟩]: term list
- match c ⟨⟨1 - 2⟩⟩;
error: no match
```

Auxiliary Function for Term Destruction

We need an auxiliary function to transform a list of quotes to a quoted list

```
- pull [⟪1⟫, ⟪2⟫,⟪3⟫];
⟪[1,2,3]⟫: term
```

intel

Auxiliary Function for Term Destruction

We need an auxiliary function to transform a list of quotes to a quoted list

```
letrec pull []     = ⟪[]⟫
     | pull (h:t)  = ⟪^h:^(pull t)⟫;
```

```
- pull [⟪1⟫, ⟪2⟫,⟪3⟫];
⟪[1,2,3]⟫: term
```

Implementing Anti-Quote based Term Destruction

```
letrec eval ⟪(λ^(QUOTE ctx pts). ^bdy) ^val⟫ =
           eval ⟪(λ^(pull pts). ^bdy)
                  ^(pull (match ctx val))⟫
     ...;
  eval ⟪(λ⟪^x + ^y⟫. x) ⟪1 + 2⟫⟫
=
⟪⟪1⟫⟫
```

Implementing Anti-Quote based Term Destruction

```
letrec eval ⟦(λ^(QUOTE ctx pts). ^bdy) ^val⟧ =
            eval ⟦(λ^(pull pts). ^bdy)
                  ^(pull (match ctx val))⟧
       ...;
  eval ⟦(λ⟦^x + ^y⟧. x) ⟦1 + 2⟧⟧
= eval
    ⟦(λ^(QUOTE (␣ + ␣) [⟦x⟧, ⟦y⟧]). x) ⟦1 + 2⟧⟧
= eval ⟦(λ^(pull [⟦x⟧, ⟦y⟧]). x)
          ^(pull (match (␣ + ␣) ⟦⟦1 + 2⟧⟧))⟧
= eval ⟦(λ^⟦[x, y]⟧. x)
          ^(pull (match (␣ + ␣) ⟦⟦1 + 2⟧⟧))⟧
= eval
    ⟦(λ^⟦[x, y]⟧. x ^(pull [⟦⟦1⟧⟧, ⟦⟦2⟧⟧]))⟧
= eval ⟦(λ^⟦[x, y]⟧. x) ^⟦[⟦1⟧, ⟦2⟧]⟧⟧
= eval ⟦(λ[x, y]. x) [⟦1⟧, ⟦2⟧]⟧
= ⟦⟦1⟧⟧
```

Implementing Anti-Quote based Term Destruction

```
letrec eval ⟪(λ^(QUOTE ctx pts). ^bdy) ^val⟫ =
           eval ⟪(λ^(pull pts). ^bdy)
                ^(pull (match ctx val))⟫
       ...;
  eval ⟪(λ⟪^x + ^y⟫. x) ⟪1 + 2⟫⟫
= eval
   ⟪(λ^(QUOTE (␣ + ␣) [⟪x⟫, ⟪y⟫]). x) ⟪1 + 2⟫⟫
= eval ⟪(λ^(pull [⟪x⟫, ⟪y⟫]). x)
          ^(pull (match (␣ + ␣) ⟪⟪1 + 2⟫⟫))⟫
= eval ⟪(λ^⟪[x, y]⟫. x)
          ^(pull (match (␣ + ␣) ⟪⟪1 + 2⟫⟫))⟫
= eval
   ⟪(λ^⟪[x, y]⟫. x ^(pull [⟪⟪1⟫⟫, ⟪⟪2⟫⟫]))⟫
= eval ⟪(λ^⟪[x, y]⟫. x) ^⟪[⟪1⟫, ⟪2⟫]⟫⟫
= eval ⟪(λ[x, y]. x) [⟪1⟫, ⟪2⟫]⟫
= ⟪⟪1⟫⟫
```

Implementing Anti-Quote based Term Destruction

```
letrec eval ⟪(λ^(QUOTE ctx pts). ^bdy) ^val⟫ =
            eval ⟪(λ^(pull pts). ^bdy)
                  ^(pull (match ctx val))⟫
      ...;
   eval ⟪(λ⟪^x + ^y⟫. x) ⟪1 + 2⟫⟫
 = eval
     ⟪(λ^(QUOTE (␣ + ␣) [⟪x⟫, ⟪y⟫]). x) ⟪1 + 2⟫⟫
 = eval ⟪(λ^(pull [⟪x⟫, ⟪y⟫]). x)
          ^(pull (match (␣ + ␣) ⟪⟪1 + 2⟫⟫))⟫
 = eval ⟪(λ^⟪[x, y]⟫. x)
          ^(pull (match (␣ + ␣) ⟪⟪1 + 2⟫⟫))⟫
 = eval
     ⟪(λ^⟪[x, y]⟫. x ^(pull [⟪⟪1⟫⟫, ⟪⟪2⟫⟫]))⟫
 = eval ⟪(λ^⟪[x, y]⟫. x) ^⟪[⟪1⟫, ⟪2⟫]⟫⟫
 = eval ⟪(λ[x, y]. x) [⟪1⟫, ⟪2⟫]⟫
 = ⟪⟪1⟫⟫
```

Implementing Anti-Quote based Term Destruction

```
letrec eval ⟦(λ^(QUOTE ctx pts). ^bdy) ^val⟧ =
           eval ⟦(λ^(pull pts). ^bdy)
                ^(pull (match ctx val))⟧
    ...;
  eval ⟦(λ⟦^x + ^y⟧. x) ⟦1 + 2⟧⟧
= eval
    ⟦(λ^(QUOTE (␣ + ␣) [⟦x⟧, ⟦y⟧]). x) ⟦1 + 2⟧⟧
= eval ⟦(λ^(pull [⟦x⟧, ⟦y⟧]). x)
         ^(pull (match (␣ + ␣) ⟦⟦1 + 2⟧⟧))⟧
= eval ⟦(λ^⟦[x, y]⟧. x)
         ^(pull (match (␣ + ␣) ⟦⟦1 + 2⟧⟧))⟧
= eval
    ⟦(λ^⟦[x, y]⟧. x ^(pull [⟦⟦1⟧⟧, ⟦⟦2⟧⟧]))⟧
= eval ⟦(λ^⟦[x, y]⟧. x) ^⟦[⟦1⟧, ⟦2⟧]⟧⟧
= eval ⟦(λ[x, y]. x) [⟦1⟧, ⟦2⟧]⟧
= ⟦⟦1⟧⟧
```

Implementing Anti-Quote based Term Destruction

```
letrec eval ⟪(λ^(QUOTE ctx pts). ^bdy) ^val⟫ =
            eval ⟪(λ^(pull pts). ^bdy)
                  ^(pull (match ctx val))⟫
    ...;
  eval ⟪(λ⟪^x + ^y⟫. x) ⟪1 + 2⟫⟫
= eval
    ⟪(λ^(QUOTE (␣ + ␣) [⟪x⟫, ⟪y⟫]). x) ⟪1 + 2⟫⟫
= eval ⟪(λ^(pull [⟪x⟫, ⟪y⟫]). x)
        ^(pull (match (␣ + ␣) ⟪⟪1 + 2⟫⟫))⟫
= eval ⟪(λ^⟪[x, y]⟫. x)
        ^(pull (match (␣ + ␣) ⟪⟪1 + 2⟫⟫))⟫
= eval
    ⟪(λ^⟪[x, y]⟫. x ^(pull [⟪⟪1⟫⟫, ⟪⟪2⟫⟫]))⟫
= eval ⟪(λ^⟪[x, y]⟫. x) ^⟪[⟪1⟫, ⟪2⟫]⟫⟫
= eval ⟪(λ[x, y]. x) [⟪1⟫, ⟪2⟫]⟫
= ⟪⟪1⟫⟫
```

Implementing Anti-Quote based Term Destruction

```
letrec eval ⟪(λ^(QUOTE ctx pts). ^bdy) ^val⟫ =
          eval ⟪(λ^(pull pts). ^bdy)
                ^(pull (match ctx val))⟫
    ...;
  eval ⟪(λ⟪^x + ^y⟫. x) ⟪1 + 2⟫⟫
= eval
    ⟪(λ^(QUOTE (␣ + ␣) [⟪x⟫, ⟪y⟫]). x) ⟪1 + 2⟫⟫
= eval ⟪(λ^(pull [⟪x⟫, ⟪y⟫]). x)
         ^(pull (match (␣ + ␣) ⟪⟪1 + 2⟫⟫))⟫
= eval ⟪(λ^⟪[x, y]⟫. x)
         ^(pull (match (␣ + ␣) ⟪⟪1 + 2⟫⟫))⟫
= eval
    ⟪(λ^⟪[x, y]⟫. x ^(pull [⟪⟪1⟫⟫, ⟪⟪2⟫⟫]))⟫
= eval ⟪(λ^⟪[x, y]⟫. x) ^⟪[⟪1⟫, ⟪2⟫]⟫⟫
= eval ⟪(λ[x, y]. x) [⟪1⟫, ⟪2⟫]⟫
= ⟪⟪1⟫⟫
```

Implementing Anti-Quote based Term Destruction

```
letrec eval ⟪(λ^(QUOTE ctx pts). ^bdy) ^val⟫ =
           eval ⟪(λ^(pull pts). ^bdy)
                 ^(pull (match ctx val))⟫
    ...;
  eval ⟪(λ⟪^x + ^y⟫. x) ⟪1 + 2⟫⟫
= eval
    ⟪(λ^(QUOTE (␣ + ␣) [⟪x⟫, ⟪y⟫]). x) ⟪1 + 2⟫⟫
= eval ⟪(λ^(pull [⟪x⟫, ⟪y⟫]). x)
         ^(pull (match (␣ + ␣) ⟪⟪1 + 2⟫⟫))⟫
= eval ⟪(λ^⟪[x, y]⟫. x)
         ^(pull (match (␣ + ␣) ⟪⟪1 + 2⟫⟫))⟫
= eval
    ⟪(λ^⟪[x, y]⟫. x ^(pull [⟪⟪1⟫⟫, ⟪⟪2⟫⟫]))⟫
= eval ⟪(λ^⟪[x, y]⟫. x) ^⟪[⟪1⟫, ⟪2⟫]⟫⟫
= eval ⟪(λ[x, y]. x) [⟪1⟫, ⟪2⟫]⟫
= ⟪⟪1⟫⟫
```

Implementing Anti-Quote based Term Destruction

```
letrec eval ⟪(λ^(QUOTE ctx pts). ^bdy) ^val⟫ =
           eval ⟪(λ^(pull pts). ^bdy)
                ^(pull (match ctx val))⟫
    ...;
  eval ⟪(λ⟪^x + ^y⟫. x) ⟪1 + 2⟫⟫
= eval
    ⟪(λ^(QUOTE (␣ + ␣) [⟪x⟫, ⟪y⟫]). x) ⟪1 + 2⟫⟫
= eval ⟪(λ^(pull [⟪x⟫, ⟪y⟫]). x)
      ^(pull (match (␣ + ␣) ⟪⟪1 + 2⟫⟫))⟫
= eval ⟪(λ^⟪[x, y]⟫. x)
      ^(pull (match (␣ + ␣) ⟪⟪1 + 2⟫⟫))⟫
= eval
    ⟪(λ^⟪[x, y]⟫. x ^(pull [⟪⟪1⟫⟫, ⟪⟪2⟫⟫]))⟫
= eval ⟪(λ^⟪[x, y]⟫. x) ^⟪[⟪1⟫, ⟪2⟫]⟫⟫
= eval ⟪(λ[x, y]. x) [⟪1⟫, ⟪2⟫]⟫
= ⟪⟪1⟫⟫
```

Implementing Anti-Quote based Term Destruction

```
letrec eval ⟪(λ^(QUOTE ctx pts). ^bdy) ^val⟫ =
            eval ⟪(λ^(pull pts). ^bdy)
                  ^(pull (match ctx val))⟫
      ...;
  eval ⟪(λ⟪^x + ^y⟫. x) ⟪1 + 2⟫⟫
= eval
    ⟪(λ^(QUOTE (␣ + ␣) [⟪x⟫, ⟪y⟫]). x) ⟪1 + 2⟫⟫
= eval ⟪(λ^(pull [⟪x⟫, ⟪y⟫]). x)
         ^(pull (match (␣ + ␣) ⟪⟪1 + 2⟫⟫))⟫
= eval ⟪(λ^⟪[x, y]⟫. x)
         ^(pull (match (␣ + ␣) ⟪⟪1 + 2⟫⟫))⟫
= eval
    ⟪(λ^⟪[x, y]⟫. x ^(pull [⟪⟪1⟫⟫, ⟪⟪2⟫⟫]))⟫
= eval ⟪(λ^⟪[x, y]⟫. x) ^⟪[⟪1⟫, ⟪2⟫]⟫⟫
= eval ⟪(λ[x, y]. x) [⟪1⟫, ⟪2⟫]⟫
= ⟪⟪1⟫⟫
```

The End

Conclusions

- quote/anti-quote are a convenient way to manipulate terms
- most common manipulations preserve the level of a term
- context term view makes level preserving manipulation easy
- implementation is straightforward

The End

Conclusions

- quote/anti-quote are a convenient way to manipulate terms
- most common manipulations preserve the level of a term
- context term view makes level preserving manipulation easy
- implementation is straightforward

Interesting Things I Didn't Mention

- The type system and run-time type checking
- Manipulations that don't preserve level: true reflection

The End

Conclusions

- quote/anti-quote are a convenient way to manipulate terms
- most common manipulations preserve the level of a term
- context term view makes level preserving manipulation easy
- implementation is straightforward

Interesting Things I Didn't Mention

- The type system and run-time type checking
- Manipulations that don't preserve level: true reflection

Ideas About The Future

- More advanced types to eliminate run-time type checking
- Restrictions on reflection to ensure soundness