Lightweight static capabilities

Oleg Kiselyov (FNMOC)
Chung-chieh Shan (Rutgers University)

PLPV 2006

Goals

- Safety
  - no buffer overflow
  - modular arithmetic
- Performance (minimal runtime checking)
- Static assurance
- Available now
  - languages (Haskell, OCaml)
  - tools (compilers, debuggers)
  - features (IO, general recursion, mutation)
  - algorithms (Knuth-Morris-Pratt string search)

Goals

- Safety
  - no buffer overflow
  - modular arithmetic
- Performance (minimal runtime checking)
- Static assurance
- Available now
  - languages (Haskell, OCaml)
  - tools (compilers, debuggers)
  - features (IO, general recursion, mutation)
  - algorithms (Knuth-Morris-Pratt string search)

```
bsearch cmp arr key = brand arr (\arr ->
  let rec loop i k = compare i k None (\i' k' ->
    let j = middle i' k' and x = get arr j in
    case cmp x key of LT -> loop (succ j) k
                      EQ -> Just (unbi j, x)
                      GT -> loop i (pred j))
  in loop)
```

Static capabilities

Continuum of correctness

- Assure safety properties, not full correctness
- Extend trust from small kernel to large sandbox

System requirements

- Higher-rank polymorphism
- Phantom types instead of dependent types

Static capabilities

Continuum of correctness

- Assure safety properties, not full correctness
- Extend trust from small kernel to large sandbox

System requirements

- Higher-rank polymorphism
- Phantom types instead of dependent types

A *capability* authorizes access to a protected object and certifies that a safety condition holds.

Outline

Trivial example: Empty-list checking

- List reverse
- Abstract data type to witness a runtime invariant
- Formalization: putting data constructors to work

Main example: Array-bounds checking

- Binary search
- Higher-rank polymorphism for an infinite family of invariants
- Formalization: lightweight dependent typing

List reverse

Starting point: ensure safety by *redundant* runtime checks.

```
rev l acc = if null l then acc
            else rev (tail l) (cons (head l) acc)
```

Idea: record the result of `null` check by wrapping the list type.

```
newtype List+ a = Nonempty (List a)
```

Runtime check supplies certifying witness to continuation.

```
indeed :: List a -> w -> (List+ a -> w)
head   :: List+ a -> a
tail   :: List+ a -> List a
```

Now `head` and `tail` need not check safety—

```
rev l acc = indeed l acc
            (\l -> rev (tail l) (cons (head l) acc))
```

—as long as `Nonempty` does not wrap an empty list.

Abstract data type to witness a runtime invariant

Use Milner's idea in LCF/ML:

- Divide the program into a small *security kernel* and a large *client sandbox*.

```
module Kernel
(
    List, List+,
    nil, cons,
    indeed, head, tail
)
where ...
```

- Using a module or namespace system, ensure that only the security kernel may apply the data constructor `Nonempty`.

Formalize security as an invariant of an abstract data type.

System F

Metavariables

Term variables	x, y, z
Terms	E
Type variables	s, t
Types	N, T, W
Natural numbers	m, n

System F

Metavariables

Term variables	x, y, z
Terms	E
Type variables	s, t
Types	N, T, W
Natural numbers	m, n

$$\frac{T : \star \quad T' : \star}{T \to T' : \star} \qquad \frac{\begin{array}{c}[t : \star] \\ \vdots \\ T' : \star\end{array}}{\forall t.\, T' : \star}$$

$$\frac{\begin{array}{c}[x : T] \\ \vdots \\ T : \star \quad E : T'\end{array}}{\lambda x.E : T \to T'} \qquad \frac{E_1 : T \to T' \quad E_2 : T}{E_1 E_2 : T'} \qquad \frac{\begin{array}{c}[t : \star] \\ \vdots \\ E : T'\end{array}}{\Lambda t.E : \forall t.\, T'} \qquad \frac{E : \forall t.\, T' \quad T : \star}{ET : T'\{t \mapsto T\}}$$

Putting data constructors to work

$$\frac{T : \star}{\mathrm{List}\ T : \star} \qquad \frac{T : \star}{\mathrm{List}^+\ T : \star} \qquad \frac{}{\mathrm{Int} : \star}$$

$$\frac{T : \star}{\mathrm{nil} : \mathrm{List}\ T} \qquad \frac{E_1 : T \quad E_2 : \mathrm{List}\ T}{E_1 :: E_2 : \mathrm{List}\ T} \qquad \frac{E_1 : T \quad E_2 : \mathrm{List}\ T}{\mathrm{nonempty}\ (E_1 :: E_2) : \mathrm{List}^+\ T} \qquad \frac{}{n : \mathrm{Int}}$$

$$\frac{E : \mathrm{List}\ T \quad E_1 : W \quad E_2 : \mathrm{List}^+\ T \to W}{\mathrm{indeed}\ E\ E_1\ E_2 : W} \qquad \frac{E : \mathrm{List}^+\ T}{\mathrm{head}\ E : T} \qquad \frac{E : \mathrm{List}^+\ T}{\mathrm{tail}\ E : \mathrm{List}\ T}$$

Putting data constructors to work

$$\frac{T : \star}{\mathrm{List}\ T : \star} \qquad \frac{T : \star}{\mathrm{List}^+\ T : \star} \qquad \frac{}{\mathrm{Int} : \star}$$

$$\frac{T : \star}{\mathrm{nil} : \mathrm{List}\ T} \qquad \frac{E_1 : T \quad E_2 : \mathrm{List}\ T}{E_1 :: E_2 : \mathrm{List}\ T} \qquad \frac{E_1 : T \quad E_2 : \mathrm{List}\ T}{\mathrm{nonempty}\ (E_1 :: E_2) : \mathrm{List}^+\ T} \qquad \frac{}{n : \mathrm{Int}}$$

$$\frac{E : \mathrm{List}\ T \quad E_1 : W \quad E_2 : \mathrm{List}^+\ T \to W}{\mathrm{indeed}\ E\ E_1\ E_2 : W} \qquad \frac{E : \mathrm{List}^+\ T}{\mathrm{head}\ E : T} \qquad \frac{E : \mathrm{List}^+\ T}{\mathrm{tail}\ E : \mathrm{List}\ T}$$

Small-step operational semantics

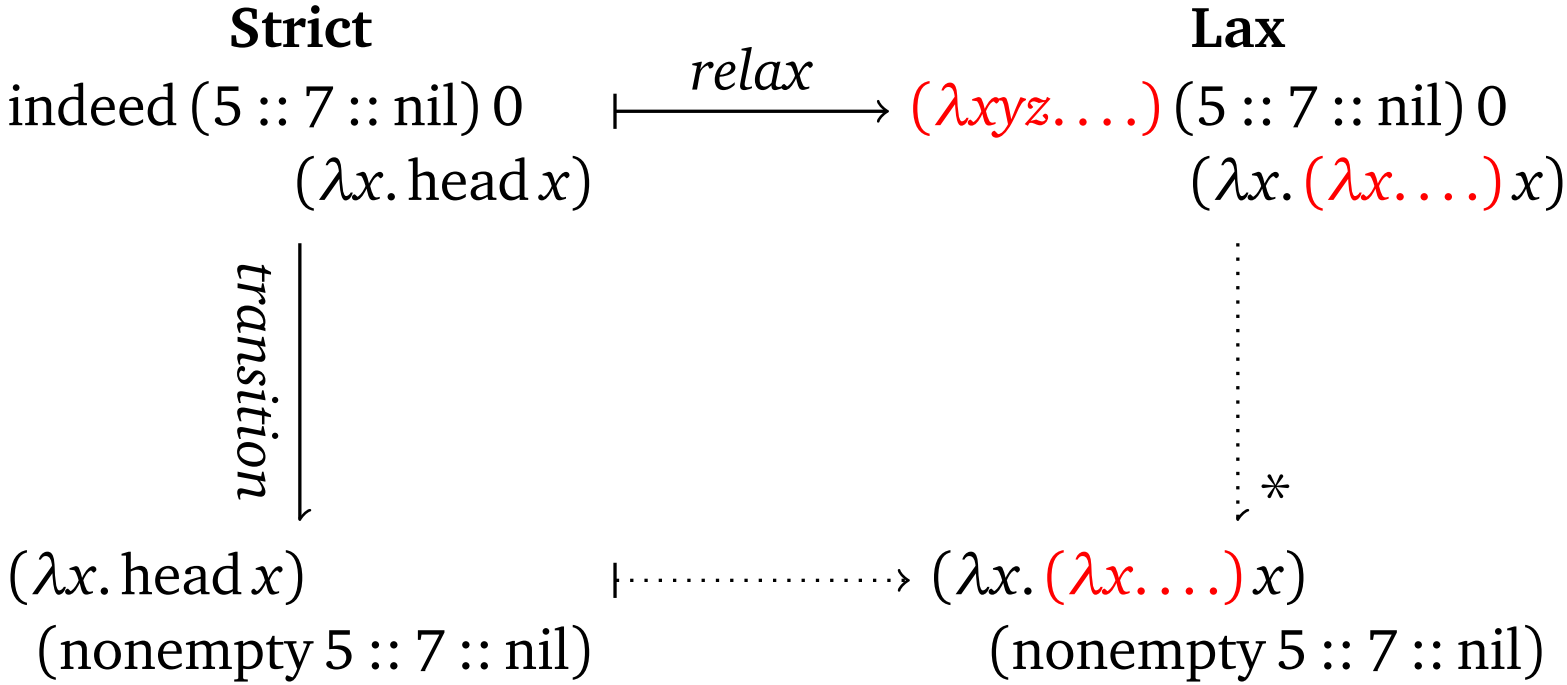
$$\dfrac{\vdots \quad \overline{} \quad \vdots}{\dfrac{5 :: 7 :: \mathrm{nil} : \mathrm{List\ Int} \quad 0 : \mathrm{Int} \quad \lambda x.\,\mathrm{head}\,x : \mathrm{List}^{+}\ \mathrm{Int} \to \mathrm{Int}}{\mathrm{indeed}\,(5 :: 7 :: \mathrm{nil})\,0\,(\lambda x.\,\mathrm{head}\,x) : \mathrm{Int}}}$$

$$\Big\downarrow \textit{transition}$$

$$\dfrac{\vdots \quad \dfrac{\overline{} \quad \vdots}{5 : \mathrm{Int} \quad 7 :: \mathrm{nil} : \mathrm{List\ Int}}}{\dfrac{\lambda x.\,\mathrm{head}\,x : \mathrm{List}^{+}\ \mathrm{Int} \to \mathrm{Int} \quad \mathrm{nonempty}\,(5 :: 7 :: \mathrm{nil}) : \mathrm{List}^{+}\ \mathrm{Int}}{(\lambda x.\,\mathrm{head}\,x)\big(\mathrm{nonempty}\,(5 :: 7 :: \mathrm{nil})\big) : \mathrm{Int}}}$$

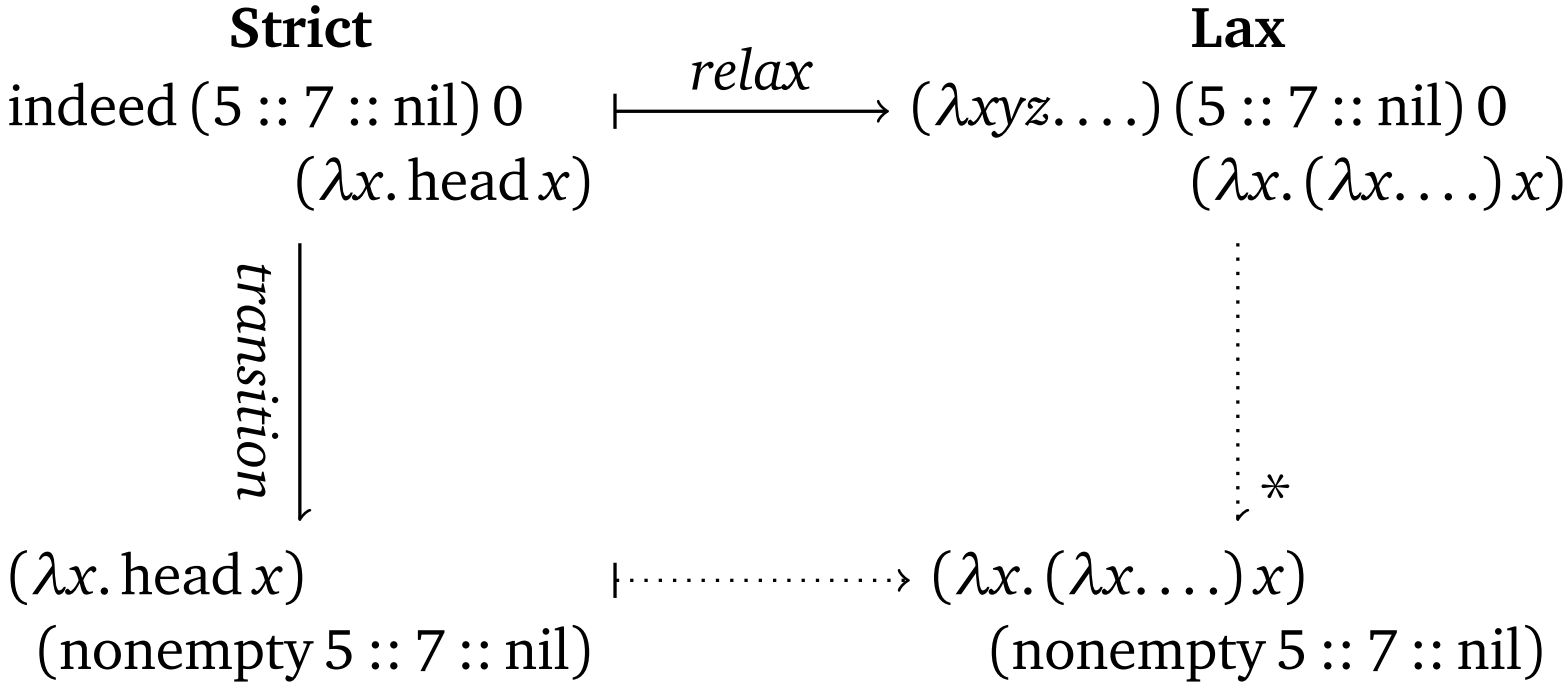
Formalization

Small-step semantics ($\downarrow$) with syntax-directed translation ($\mapsto$)



Formalization

Small-step semantics ($\downarrow$) with syntax-directed translation ($\mapsto$)

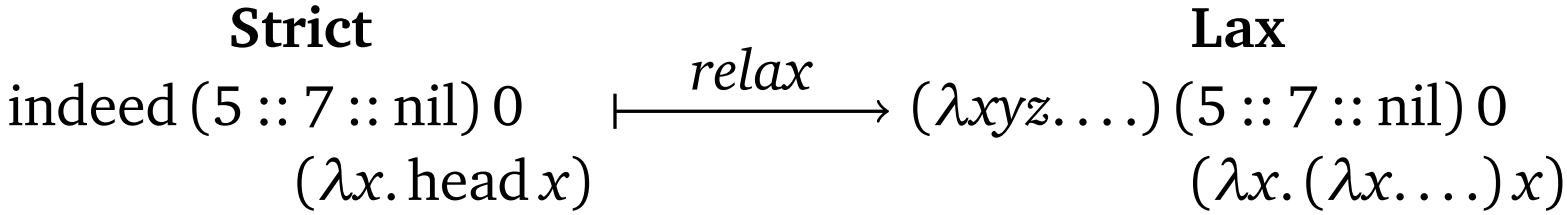


Relaxation preserves typing, valuehood, and transitions*. To prove:

- The kernel is implemented in Lax as specified in Strict.
- The sandbox constructs are identical in Lax and in Strict.

Formalization

Small-step semantics ($\downarrow$) with syntax-directed translation ($\mapsto$)



We call a Lax program *sandboxed* if it uses kernel constructs only by inlining the kernel implementation.

Extend trust from kernel to sandbox

- Relaxation preserves typing, valuehood, and transitions*.
- Every (well-typed) sandboxed Lax program is the relaxation of some (well-typed) Strict program.
- Strict enjoys progress and preservation: well-typed Strict code does not go wrong.

Hence, well-typed sandboxed Lax code does not go wrong.

Outline

Trivial example: Empty-list checking

- List reverse
- Abstract data type to witness a runtime invariant
- Formalization: putting data constructors to work

Main example: Array-bounds checking

- Binary search
- Higher-rank polymorphism for an infinite family of invariants
- Formalization: lightweight dependent typing

Lightweight dependent typing

$$\frac{}{\bar{n} : \star} \qquad \frac{N : \star \quad T : \star}{\mathrm{List}^N\, T : \star} \qquad \frac{N : \star}{\mathrm{Int}^N : \star} \qquad \frac{N : \star}{\mathrm{Int}_{\mathrm{L}}^N : \star} \qquad \frac{N : \star}{\mathrm{Int}_{\mathrm{H}}^N : \star}$$

$$\frac{E_1 : T \quad \ldots \quad E_n : T}{\mathrm{array}\, E_1 :: \ldots E_n :: \mathrm{nil} : \mathrm{List}^{\bar{n}}\, T} \qquad \frac{1 \le m \le n}{m_{\mathrm{I}} : \mathrm{Int}^{\bar{n}}} \qquad \frac{1 \le m}{m_{\mathrm{L}} : \mathrm{Int}_{\mathrm{L}}^{\bar{n}}} \qquad \frac{m \le n}{m_{\mathrm{H}} : \mathrm{Int}_{\mathrm{H}}^{\bar{n}}}$$

$$\frac{E : \mathrm{List}\, T \quad E' : \forall s.\, \mathrm{List}^s\, T \to \mathrm{Int}_{\mathrm{L}}^s \to \mathrm{Int}_{\mathrm{H}}^s \to W}{\mathrm{brand}\, E\, E' : W} \qquad \frac{E_1 : \mathrm{List}^N\, T \quad E_2 : \mathrm{Int}^N}{\mathrm{get}\, E_1\, E_2 : T}$$

$$\frac{E_{\mathrm{L}} : \mathrm{Int}_{\mathrm{L}}^N \quad E_{\mathrm{H}} : \mathrm{Int}_{\mathrm{H}}^N \quad E_1 : W \quad E_2 : \mathrm{Int}^N \to \mathrm{Int}^N \to W}{\mathrm{compare}\, E_{\mathrm{L}}\, E_{\mathrm{H}}\, E_1\, E_2 : W}$$

$$\frac{E_1 : \mathrm{Int}^N \quad E_2 : \mathrm{Int}^N}{\mathrm{middle}\, E_1\, E_2 : \mathrm{Int}^N} \qquad \frac{E : \mathrm{Int}^N}{\mathrm{succ}\, E : \mathrm{Int}_{\mathrm{L}}^N} \qquad \frac{E : \mathrm{Int}^N}{\mathrm{pred}\, E : \mathrm{Int}_{\mathrm{H}}^N} \qquad \frac{E : \mathrm{Int}^N}{\mathrm{unbi}\, E : \mathrm{Int}}$$

Lightweight dependent typing

$$\frac{}{\bar{n} : \star} \qquad \frac{N : \star \quad T : \star}{\mathrm{List}^N\, T : \star} \qquad \frac{N : \star}{\mathrm{Int}^N : \star} \qquad \frac{N : \star}{\mathrm{Int}^N_{\mathrm{L}} : \star} \qquad \frac{N : \star}{\mathrm{Int}^N_{\mathrm{H}} : \star}$$

$$\frac{E_1 : T \quad \ldots \quad E_n : T}{\mathrm{array}\, E_1 :: \ldots E_n :: \mathrm{nil} : \mathrm{List}^{\bar{n}}\, T} \qquad \frac{1 \le m \le n}{m_{\mathrm{I}} : \mathrm{Int}^{\bar{n}}} \qquad \frac{1 \le m}{m_{\mathrm{L}} : \mathrm{Int}^{\bar{n}}_{\mathrm{L}}} \qquad \frac{m \le n}{m_{\mathrm{H}} : \mathrm{Int}^{\bar{n}}_{\mathrm{H}}}$$

$$\frac{E : \mathrm{List}\, T \quad E' : \forall s.\, \mathrm{List}^s\, T \to \mathrm{Int}^s_{\mathrm{L}} \to \mathrm{Int}^s_{\mathrm{H}} \to W}{\mathrm{brand}\, E\, E' : W} \qquad \frac{E_1 : \mathrm{List}^N\, T \quad E_2 : \mathrm{Int}^N}{\mathrm{get}\, E_1\, E_2 : T}$$

$$\frac{E_{\mathrm{L}} : \mathrm{Int}^N_{\mathrm{L}} \quad E_{\mathrm{H}} : \mathrm{Int}^N_{\mathrm{H}} \quad E_1 : W \quad E_2 : \mathrm{Int}^N \to \mathrm{Int}^N \to W}{\mathrm{compare}\, E_{\mathrm{L}}\, E_{\mathrm{H}}\, E_1\, E_2 : W}$$

$$\frac{E_1 : \mathrm{Int}^N \quad E_2 : \mathrm{Int}^N}{\mathrm{middle}\, E_1\, E_2 : \mathrm{Int}^N} \qquad \frac{E : \mathrm{Int}^N}{\mathrm{succ}\, E : \mathrm{Int}^N_{\mathrm{L}}} \qquad \frac{E : \mathrm{Int}^N}{\mathrm{pred}\, E : \mathrm{Int}^N_{\mathrm{H}}} \qquad \frac{E : \mathrm{Int}^N}{\mathrm{unbi}\, E : \mathrm{Int}}$$

Small-step operational semantics

$$\frac{\dfrac{\vdots}{5 :: 7 :: \mathrm{nil} : \mathrm{List}\ \mathrm{Int}} \qquad \dfrac{\vdots}{\Lambda s.\, \lambda xyz.\, \mathrm{compare}\, y\, z\, 0\, \lambda yz.\, \mathrm{get}\, x\, (\mathrm{middle}\, y\, z) : \forall s.\, \mathrm{List}^{s}\ \mathrm{Int} \to \mathrm{Int}^{s}_{\mathrm{L}} \to \mathrm{Int}^{s}_{\mathrm{H}} \to \mathrm{Int}}}{\mathrm{brand}\ (5 :: 7 :: \mathrm{nil})\ \ \Lambda s.\, \lambda xyz.\, \mathrm{compare}\, y\, z\, 0\, \lambda yz.\, \mathrm{get}\, x\, (\mathrm{middle}\, y\, z) : \mathrm{Int}}$$

$$\downarrow$$

$$(\Lambda s.\, \lambda xyz.\, \mathrm{compare}\, y\, z\, 0\, \lambda yz.\, \mathrm{get}\, x\, (\mathrm{middle}\, y\, z))\ \bar{2}\ (\mathrm{array}\ 5 :: 7 :: \mathrm{nil})\ 1_{\mathrm{L}}\ 2_{\mathrm{H}}$$

$$\downarrow_{*}$$

$$\frac{\dfrac{\vdots}{\mathrm{array}\ 5 :: 7 :: \mathrm{nil} : \mathrm{List}^{\bar{2}}\ \mathrm{Int}} \qquad \dfrac{\dfrac{}{1 \le 1 \le 2}}{1_{\mathrm{I}} : \mathrm{Int}^{\bar{2}}}}{\mathrm{get}\ (\mathrm{array}\ 5 :: 7 :: \mathrm{nil})\ 1_{\mathrm{I}} : \mathrm{Int}}$$

Lightweight dependent typing

$$\frac{}{\bar{n} : \star} \qquad \frac{N : \star \quad T : \star}{\mathrm{List}^N\, T : \star} \qquad \frac{N : \star}{\mathrm{Int}^N : \star} \qquad \frac{N : \star}{\mathrm{Int}^N_{\mathrm{L}} : \star} \qquad \frac{N : \star}{\mathrm{Int}^N_{\mathrm{H}} : \star}$$

$$\frac{E_1 : T \quad \ldots \quad E_n : T}{\mathrm{array}\, E_1 :: \ldots E_n :: \mathrm{nil} : \mathrm{List}^{\bar{n}}\, T} \qquad \frac{1 \le m \le n}{m_{\mathrm{I}} : \mathrm{Int}^{\bar{n}}} \qquad \frac{1 \le m}{m_{\mathrm{L}} : \mathrm{Int}^{\bar{n}}_{\mathrm{L}}} \qquad \frac{m \le n}{m_{\mathrm{H}} : \mathrm{Int}^{\bar{n}}_{\mathrm{H}}}$$

$$\frac{E : \mathrm{List}\, T \quad E' : \forall s.\, \mathrm{List}^s\, T \to \mathrm{Int}^s_{\mathrm{L}} \to \mathrm{Int}^s_{\mathrm{H}} \to W}{\mathrm{brand}\, E\, E' : W} \qquad \frac{E_1 : \mathrm{List}^N\, T \quad E_2 : \mathrm{Int}^N}{\mathrm{get}\, E_1\, E_2 : T}$$

$$\frac{E_{\mathrm{L}} : \mathrm{Int}^N_{\mathrm{L}} \quad E_{\mathrm{H}} : \mathrm{Int}^N_{\mathrm{H}} \quad E_1 : W \quad E_2 : \mathrm{Int}^N \to \mathrm{Int}^N \to W}{\mathrm{compare}\, E_{\mathrm{L}}\, E_{\mathrm{H}}\, E_1\, E_2 : W}$$

$$\frac{E_1 : \mathrm{Int}^N \quad E_2 : \mathrm{Int}^N}{\mathrm{middle}\, E_1\, E_2 : \mathrm{Int}^N} \qquad \frac{E : \mathrm{Int}^N}{\mathrm{succ}\, E : \mathrm{Int}^N_{\mathrm{L}}} \qquad \frac{E : \mathrm{Int}^N}{\mathrm{pred}\, E : \mathrm{Int}^N_{\mathrm{H}}} \qquad \frac{E : \mathrm{Int}^N}{\mathrm{unbi}\, E : \mathrm{Int}}$$

Lightweight dependent typing

$$\frac{}{\bar{n} : \star} \qquad \frac{N : \star \quad T : \star}{\mathrm{List}^N\, T : \star} \qquad \frac{N : \star}{\mathrm{Int}^N : \star} \qquad \frac{N : \star}{\mathrm{Int}_{\mathrm{L}}^N : \star} \qquad \frac{N : \star}{\mathrm{Int}_{\mathrm{H}}^N : \star}$$

dependent types (Martin-Löf, Dybjer, . . .)
singleton types (Hayashi, Xi, Stone, . . .)
phantom types (. . . , Fluet & Pucella, . . .)
reflecting values through types (Thurston, Kiselyov & Shan, . . .)

$$\mathrm{brand}\, E\, E' : W \qquad \mathrm{get}\, E_1\, E_2 : T$$

$$\frac{E_{\mathrm{L}} : \mathrm{Int}_{\mathrm{L}}^N \quad E_{\mathrm{H}} : \mathrm{Int}_{\mathrm{H}}^N \quad E_1 : W \quad E_2 : \mathrm{Int}^N \to \mathrm{Int}^N \to W}{\mathrm{compare}\, E_{\mathrm{L}}\, E_{\mathrm{H}}\, E_1\, E_2 : W}$$

$$\frac{E_1 : \mathrm{Int}^N \quad E_2 : \mathrm{Int}^N}{\mathrm{middle}\, E_1\, E_2 : \mathrm{Int}^N} \qquad \frac{E : \mathrm{Int}^N}{\mathrm{succ}\, E : \mathrm{Int}_{\mathrm{L}}^N} \qquad \frac{E : \mathrm{Int}^N}{\mathrm{pred}\, E : \mathrm{Int}_{\mathrm{H}}^N} \qquad \frac{E : \mathrm{Int}^N}{\mathrm{unbi}\, E : \mathrm{Int}}$$

Putting more data constructors to work

$$\frac{}{\bar{n} : \star} \qquad \frac{N : \star \quad T : \star}{\mathrm{List}^N\, T : \star} \qquad \frac{N : \star}{\mathrm{Int}^N : \star} \qquad \frac{N : \star}{\mathrm{Int}^N_{\mathrm{L}} : \star} \qquad \frac{N : \star}{\mathrm{Int}^N_{\mathrm{H}} : \star}$$

$$\frac{E_1 : T \quad \ldots \quad E_n : T}{\mathrm{array}\, E_1 :: \ldots E_n :: \mathrm{nil} : \mathrm{List}^{\bar{n}}\, T} \qquad \frac{1 \le m \le n}{m_{\mathrm{I}} : \mathrm{Int}^{\bar{n}}} \qquad \frac{1 \le m}{m_{\mathrm{L}} : \mathrm{Int}^{\bar{n}}_{\mathrm{L}}} \qquad \frac{m \le n}{m_{\mathrm{H}} : \mathrm{Int}^{\bar{n}}_{\mathrm{H}}}$$

Memory locations as term constants (Morrisett et al., Moggi & Sabry)

$$\mathrm{brand}\, E\, E' : W \qquad\qquad \mathrm{get}\, E_1\, E_2 : T$$

$$\frac{E_{\mathrm{L}} : \mathrm{Int}^N_{\mathrm{L}} \quad E_{\mathrm{H}} : \mathrm{Int}^N_{\mathrm{H}} \quad E_1 : W \quad E_2 : \mathrm{Int}^N \to \mathrm{Int}^N \to W}{\mathrm{compare}\, E_{\mathrm{L}}\, E_{\mathrm{H}}\, E_1\, E_2 : W}$$

$$\frac{E_1 : \mathrm{Int}^N \quad E_2 : \mathrm{Int}^N}{\mathrm{middle}\, E_1\, E_2 : \mathrm{Int}^N} \qquad \frac{E : \mathrm{Int}^N}{\mathrm{succ}\, E : \mathrm{Int}^N_{\mathrm{L}}} \qquad \frac{E : \mathrm{Int}^N}{\mathrm{pred}\, E : \mathrm{Int}^N_{\mathrm{H}}} \qquad \frac{E : \mathrm{Int}^N}{\mathrm{unbi}\, E : \mathrm{Int}}$$

Nonces in security protocols

$$\frac{E : \mathrm{List}\, T \quad E' : \forall s.\, \mathrm{List}^s\, T \to \mathrm{Int}^s_{\mathrm{L}} \to \mathrm{Int}^s_{\mathrm{H}} \to W}{\mathrm{brand}\, E\, E' : W}$$

Ill-typed term: $\mathrm{brand}\,(5 :: 7 :: \mathrm{nil})\, \Lambda s.\, \lambda xyz.\, \mathrm{get}\, x\, 1_{\mathrm{I}}$

(Can't open branded lock with unbranded key)

Rights amplification

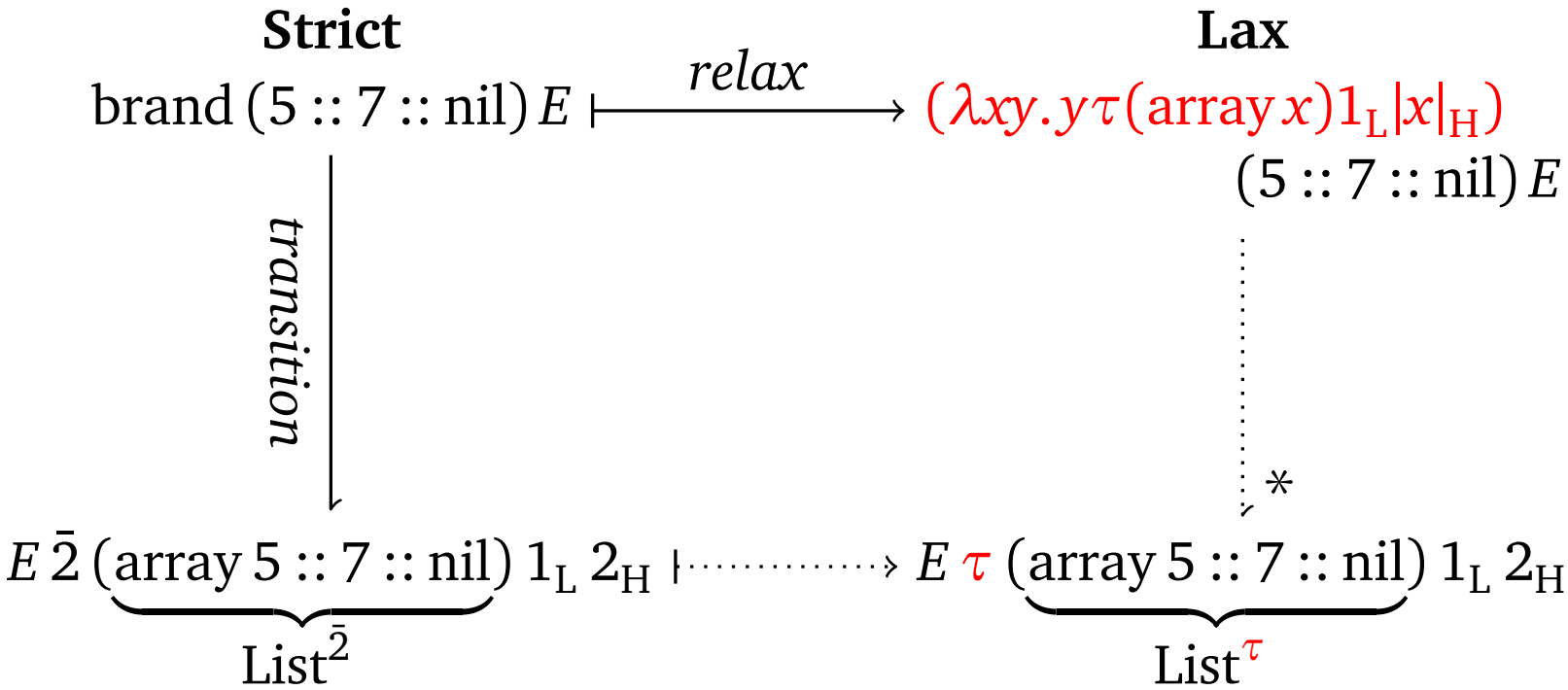
With rights amplification, the authority accessible from bringing two references together can exceed the sum of authorities provided by each individually. The classic example is the can and the can-opener—only by bringing the two together do we obtain the food in the can.

—Miller et al.

$$\frac{E_1 : \mathrm{List}^{N}\ T \quad E_2 : \mathrm{Int}^{N}}{\mathrm{get}\, E_1\, E_2 : T}$$

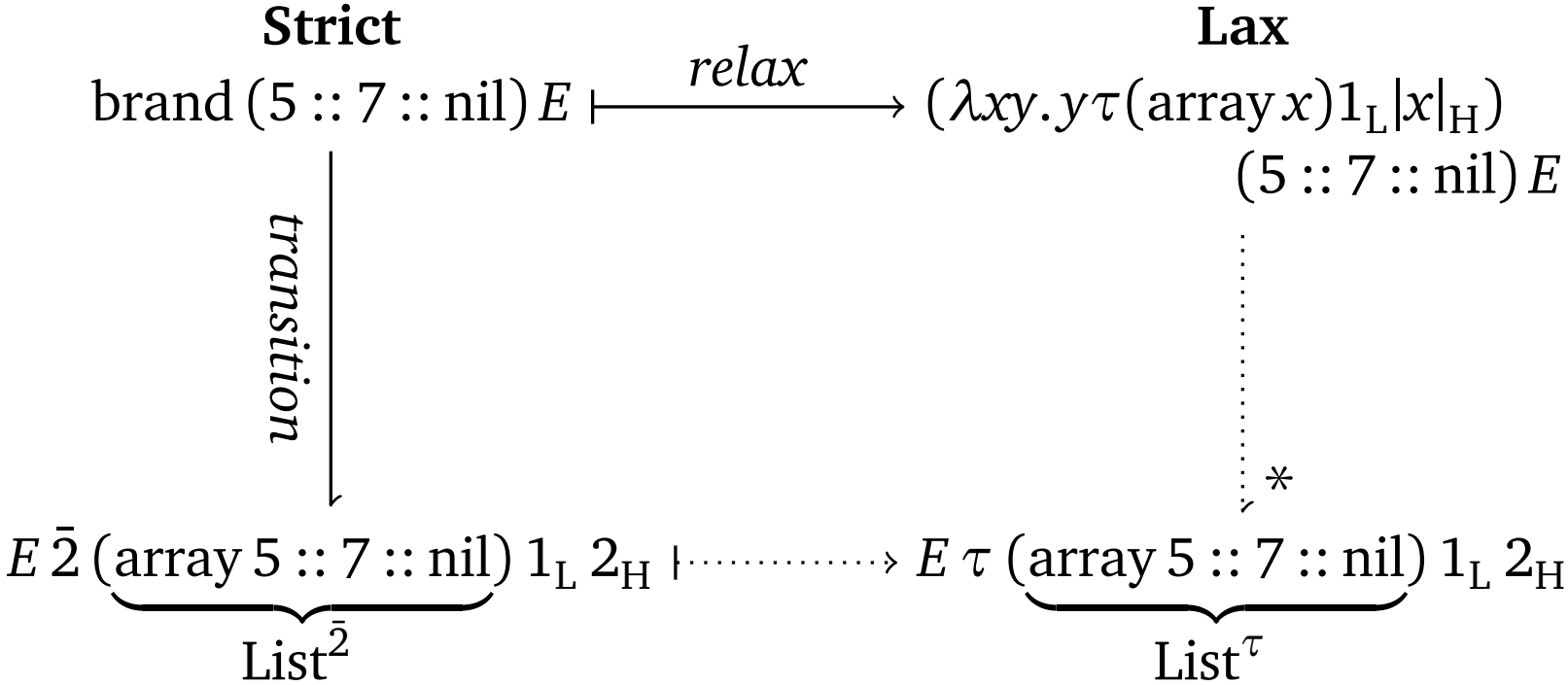
Formalization

Small-step semantics ($\lfloor\rfloor$) with syntax-directed translation ($\mapsto$)



Formalization

Small-step semantics ($\lfloor\rfloor$) with syntax-directed translation ($\mapsto$)



Relaxation preserves typing, valuehood, and transitions*. To prove:

- The kernel is implemented in Lax as specified in Strict.
- The sandbox constructs are identical in Lax and in Strict.

Formalization

Small-step semantics ($\downarrow$) with syntax-directed translation ($\mapsto$)

Strict $\qquad\qquad$ **Lax**

$$\mathrm{brand}\,(5 :: 7 :: \mathrm{nil})\,E \xmapsto{\;relax\;} (\lambda xy.\,y\tau(\mathrm{array}\,x)1_{\mathrm{L}}|x|_{\mathrm{H}})\,(5 :: 7 :: \mathrm{nil})\,E$$

We call a Lax program *sandboxed* if it uses kernel constructs only by inlining the kernel implementation.

Extend trust from kernel to sandbox

- Relaxation preserves typing, valuehood, and transitions*.
- Every (well-typed) sandboxed Lax program is the relaxation of some (well-typed) Strict program.
- Strict enjoys progress and preservation: well-typed Strict code does not go wrong.

Hence, well-typed sandboxed Lax code does not go wrong.

Twelf mechanization

Theorem (Progress)

Every well-typed term either is a value or transitions to another term.

Proof.

By induction on evaluation contexts. □

Twelf mechanization

Lemma

A value never has any type of the form $\mathrm{Int}^{T \to T'}$.

Proof.

There is only one case, the value nil. □

Theorem (Progress)

Every well-typed term either is a value or transitions to another term.

Proof.

By induction on evaluation contexts. □

Twelf mechanization

Lemma

The expression nil *never has any type of the form* Int^T.

Proof.

There are no cases. □

Lemma

A value never has any type of the form $\mathrm{Int}^{T \to T'}$.

Proof.

There is only one case, the value nil. □

Theorem (Progress)

Every well-typed term either is a value or transitions to another term.

Proof.

By induction on evaluation contexts. □

Summary

A concrete, rigorous, practical framework for extending trust from a small kernel to a large program

Available now

- Type proxies for values, instead of dependent types
- Download all code online, with more substantial examples
  - Folding over multiple arrays of various sizes
  - Knuth-Morris-Pratt string search

Easy to verify small kernel

- Prove progress and preservation for specification
- Prove implementation corresponds to specification

Ongoing work

- More examples (any suggestions?)
- Characterize lightweight dependent typing