

# Verifying Program Invariants with Refinement Types

Rowan Davies and Frank Pfenning  
Carnegie Mellon University

Logic Colloquium

Max-Planck-Institut für Informatik  
and Universität des Saarlandes  
October 2001

**Acknowledgments:** Robert Harper

# Overview

---

- Introduction
- Refinement Types
- A Value Restriction
- Progress and Type Preservation
- Bi-Directional Type Checking
- Parametric Polymorphism
- Conclusion

## Why Aren't Most Programs Verified?

---

- Difficulty of expressing a precise specification.
- Difficulty of proving correctness.
- Difficulty of co-evolving program, specification, and proof.
- Problems exacerbated by poorly designed languages.

## Why Are Most Programs Type-Checked?

---

- Ease of expressing types.
- Ease of checking types.
- Ease of co-evolving programs and types.
- Most useful in properly designed languages.

## A Continuum?

---

- Types as a *minimal* requirement for meaningful programs.
- Specifications as a *maximal* requirement for correct programs.
- Surprisingly few intermediate points have been investigated.
- Many errors are caught by simple type-checking.
- But many errors also escape simple type-checking.

# A Research Program

---

- Designing systems for statically verifying program properties.
- Evaluation along the following dimensions:
  - Elegance, generality, brevity (ease of expression)
  - Practicality of verification (ease of checking)
  - Explicitness (ease of understanding and evolution)
- Some of these involve trade-offs.

## Goals

---

- Catch more errors at compile-time.
- Increase confidence in correctness.
- Document crucial program invariants.
- Check consistency at module boundaries.
- Programmer guidance and involvement.
- **Not:** optimize compiled code.
- **Not:** extend type system to admit more programs.
- Instead: *refine* type systems to admit fewer programs.

# Traditional Static Program Analysis

---

- Many useful lessons and ideas (e.g. abstract interpretation)
- Emphasis on compiler optimization (here: error discovery).
- Emphasis on inference of properties (here: checking).
- Additional documentation?
- Additional errors discovered?
- Problems at module boundaries.

# Traditional Type Systems

---

- Many useful lessons and ideas (e.g. module interfaces)
- Emphasis on generality (e.g. polymorphism, record subtyping, intersection types).
- Emphasis on inference of types.
- Additional documentation?
- Additional errors discovered?

## The Basic Idea

---

- ML as host language.
- Data structures via datatypes.
- Invariants on data structures specified by regular tree grammars.
- Extend to full language via subtyping and intersections.
- Bi-directional type checking.

## Example: Bit Strings and Natural Numbers

---

- Datatype of bit strings (freely generated):

*Bit Strings*      bits ::=  $\epsilon$  | bits **1** | bits **0**

- $\epsilon$  represents empty string, **0** and **1** are postfix operators.
- For example:  $\lceil 0 \rceil = \epsilon$ ,  $\lceil 6 \rceil = \epsilon \mathbf{110}$ .
- Natural numbers have no leading **0**s.
- Refinements of type bits inductively defined:

*Natural Numbers*      nat ::=  $\epsilon$  | pos

*Positive Numbers*      pos ::= pos **0** | nat **1**

# The Need for Subtyping and Intersections

---

- Subtyping:  $\text{pos} \leq \text{nat} \leq \text{bits}$  (in general: lattice).
- Intersections: Consider  $\text{shif}t_l = \lambda x. x \mathbf{0}$ .

$\vdash \lambda x. x \mathbf{0} : \text{bits} \rightarrow \text{bits}$

$\vdash \lambda x. x \mathbf{0} : \text{nat} \rightarrow \text{bits}$

$\vdash \lambda x. x \mathbf{0} : \text{pos} \rightarrow \text{pos}$

- Intersections allow these to be expressed simultaneously.

$\vdash \lambda x. x \mathbf{0} : (\text{bits} \rightarrow \text{bits})$

$\wedge (\text{nat} \rightarrow \text{bits})$

$\wedge (\text{pos} \rightarrow \text{pos})$

$\not\vdash \lambda x. x \mathbf{0} : \text{nat} \rightarrow \text{nat} \quad (!)$

## Other Examples

---

- Even and odd length lists  
(but not lists of length  $n$ ).
- Empty and non-empty lists, single constructor types.
- Normal terms, head-normal terms, cps terms  
(but not closed terms).
- Color invariant on red/black trees  
(but not balance invariant).
- Valid stacks in operator precedence parsing.
- Intuition: recognizable by finite-state tree automaton.
- Generalization: restricted forms of dependent types.  
[Xi & Pf.'98,'99, Xi'99]

## What are Intersection Types?

---

- Introduction rule

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash M : B}{\Gamma \vdash M : A \wedge B}$$

- Elimination rules

$$\frac{\Gamma \vdash M : A \wedge B}{\Gamma \vdash M : A}$$

$$\frac{\Gamma \vdash M : A \wedge B}{\Gamma \vdash M : B}$$

# Subtyping and Greatest Lower Bounds

---

- Subsumption

$$\frac{\Gamma \vdash M : A \quad A \leq B}{\Gamma \vdash M : B}$$

- Intersection as a greatest lower bound

$$\overline{A \wedge B \leq A} \quad \overline{A \wedge B \leq B}$$

$$\frac{A \leq B \quad A \leq C}{A \leq B \wedge C}$$

- Elimination rules now derivable

$$\frac{\Gamma \vdash M : A \wedge B \quad \overline{A \wedge B \leq A}}{\Gamma \vdash M : A}$$

## Intersections are Unsound with Effects

---

- Counterexample

```
let  $x = \text{ref}(\epsilon \mathbf{1})$  :  $\text{nat ref} \wedge \text{pos ref}$ 
in
   $x := \epsilon$ ;           % use  $x : \text{nat ref}$ 
  ! $x$                  % use  $x : \text{pos ref}$ 
end : pos
```

evaluates to  $\epsilon$  which does not have type  $\text{pos}$ .

- Analogous counterexample with parametric polymorphism:

```
let  $x = \text{ref}(\lambda y. y)$  :  $\forall \alpha. (\alpha \rightarrow \alpha) \text{ref}$ 
in
   $x := (\lambda y. \epsilon)$ ; % use  $x : (\text{nat} \rightarrow \text{nat}) \text{ref}$ 
  (! $x$ ) ( $\epsilon \mathbf{1}$ )      % use  $x : (\text{pos} \rightarrow \text{pos}) \text{ref}$ 
end : pos
```

# Subtyping

---

Types  $A ::=$  bits | nat | pos  
|  $A_1 \rightarrow A_2$  |  $A$  ref | unit  
|  $A_1 \wedge A_2$

$$\frac{}{A \leq A} \quad \frac{A \leq B \quad B \leq C}{A \leq C}$$

$\leq$ : Reflexive and transitive

$$\frac{B_1 \leq A_1 \quad A_2 \leq B_2}{A_1 \rightarrow A_2 \leq B_1 \rightarrow B_2}$$

$\rightarrow$ : Contra- and co-variant

$$\frac{A \leq B \quad B \leq A}{A \text{ ref} \leq B \text{ ref}}$$

ref: Non-variant

# Subtyping and Intersections

---

$$\overline{\text{pos} \leq \text{nat}}$$

$$\overline{\text{nat} \leq \text{bits}}$$

Data types

$$\overline{A \wedge B \leq A}$$

$$\overline{A \wedge B \leq B}$$

$\wedge$ : Lower bound

$$\frac{A \leq B \quad A \leq C}{A \leq B \wedge C}$$

$\wedge$ : Greatest lower bound

$$\left[ \overline{(A \rightarrow B) \wedge (A \rightarrow C) \leq A \rightarrow (B \wedge C)} \right]$$

?? (Distributivity)

- Distributivity disturbs orthogonality of constructors.
- Distributivity is unsound with effects (see later).

# Typing Judgment

---

- Language is standard call-by-value language with functions, mutable references, unit, bit strings, let and recursion.
- Use pure type assignment for typeless operational semantics.
- Later: bi-directional type-checking.
- Pragmatically: refinement restriction.
- Typing rules are standard for functions, recursion, references.
- De-emphasize refinement restriction here.

## Typing Bit Strings

---

- Bit strings (two rules for case omitted):

$$\overline{\Gamma \vdash \epsilon : \text{nat}}$$

$$\frac{\Gamma \vdash M : \text{pos}}{\Gamma \vdash M \mathbf{0} : \text{pos}}$$

$$\frac{\Gamma \vdash M : \text{bits}}{\Gamma \vdash M \mathbf{0} : \text{bits}}$$

$$\frac{\Gamma \vdash M : \text{nat}}{\Gamma \vdash M \mathbf{1} : \text{pos}}$$

$$\frac{\Gamma \vdash M : \text{bits}}{\Gamma \vdash M \mathbf{1} : \text{bits}}$$

$$\frac{\Gamma \vdash M : \text{pos} \quad \Gamma, x:\text{pos} \vdash N_0 : A \quad \Gamma, y:\text{nat} \vdash N_1 : A}{\Gamma \vdash \text{case } M \text{ of } \epsilon \Rightarrow N_e \mid x \mathbf{0} \Rightarrow N_0 \mid y \mathbf{1} \Rightarrow N_1 : A}$$

- Note:  $\text{case}(M:\text{pos})$  does not need to check  $N_e$ .

## Datatype Refinement: The General Case

---

- First specify (ML) datatype.
- Then specify refinements of datatypes.
- Analysis of refinements generates:
  - Completing of lattice structure to include intersections (using algorithms from tree automata).
  - Determine most general types of constructors.
  - Determine inversion principles for constructors.
- Does not allow negative refinements.
- Polymorphic refinements must be parametric.

## Typing Judgment Continued

---

- Value restriction and subsumption.

$$\frac{\Gamma \vdash V : A \quad \Gamma \vdash V : B}{\Gamma \vdash V : A \wedge B} \qquad \frac{\Gamma \vdash M : A \quad A \leq B}{\Gamma \vdash M : B}$$

where

$$\text{Values } V ::= x \mid \lambda x. M \mid \epsilon \mid V \mathbf{0} \mid V \mathbf{1}$$

- Originally introduced for parametric polymorphism [Tofte'90] [Wright'95].
- Value restriction here not tied to let!

$$\frac{\Gamma \vdash M : A \quad \Gamma, x:A \vdash N : B}{\Gamma \vdash \text{let } x = M \text{ in } N \text{ end} : B}$$

## Counterexample Revisited

---

```
let  x = ref(ε 1) : nat ref ∧ pos ref
in
    x := ε;           % use x : nat ref
    ! x              % use x : pos ref
end : pos
```

- No longer well typed:

$\not\vdash \text{ref}(\epsilon \mathbf{1}) : \text{nat ref} \wedge \text{pos ref}$

since  $\text{ref}(\epsilon \mathbf{1})$  is not a value.

## Distributivity Revisited

---

- Distributivity is unsound with effects.

$$\left[ \overline{(A \rightarrow B) \wedge (A \rightarrow C) \leq A \rightarrow (B \wedge C)} \right]$$

- Counterexample:

$$\vdash \lambda u. \text{ref}(\epsilon \mathbf{1}) \quad : \quad (\text{unit} \rightarrow \text{nat ref}) \wedge (\text{unit} \rightarrow \text{pos ref})$$

by distributivity and subsumption:

$$\vdash \lambda u. \text{ref}(\epsilon \mathbf{1}) \quad : \quad \text{unit} \rightarrow (\text{nat ref} \wedge \text{pos ref})$$

$$\vdash (\lambda u. \text{ref}(\epsilon \mathbf{1})) \langle \rangle \quad : \quad \text{nat ref} \wedge \text{pos ref}$$

- In a program:

```
let x = (λu. ref(ε 1)) ⟨ ⟩ : nat ref ∧ pos ref
in ... end                % as on slide 5
```

## Results

---

- **Theorem:** Subtyping is structural.
- **Lemma:** (*Typing Inversion*) With a *store typing*  $\Delta$ :
  1. If  $\Delta; \cdot \vdash V : A$  and  $A \leq B \rightarrow C$   
then  $V = \lambda x. M$  and  $\Delta; x:B \vdash M : C$ .
  2. ... (*one for each type or type constructor*) ...

Fails in the presence of distributivity!

- **Theorem:** Call-by-value reduction semantics satisfies *progress* and *type preservation*.
- **Proof:** Follows [Wright & Felleisen '94] [Harper'94], using above inductive inversion properties.  
Fails in the presence of unrestricted intersection!

## Consequences

---

- Language has no principal types:

$\vdash \text{ref}(\epsilon \mathbf{1}) : \text{bits ref}$

$\vdash \text{ref}(\epsilon \mathbf{1}) : \text{nat ref}$

$\vdash \text{ref}(\epsilon \mathbf{1}) : \text{pos ref}$

but bits ref, nat ref and pos ref are unrelated and

$\not\vdash \text{ref}(\epsilon \mathbf{1}) : \text{bits ref} \wedge \text{nat ref} \wedge \text{pos ref}$

## Bi-Directional Type-Checking

---

- Simplified subtyping allows simplified bi-directional type-checking.
- Functional fragment

Inferable  $I ::= x \mid IC \mid C:A$

Checkable  $C ::= I \mid \lambda x. C$

- Normal forms require no type annotations.
- Two mutually recursive judgments:

$\Gamma \vdash I \uparrow A$        $I$  synthesizes  $A$  (non-deterministically)

$\Gamma \vdash C \downarrow A$        $C$  checks against  $A$

## Bi-Directional Typing Rules

---

- Inferable

$$\frac{x:A \text{ in } \Gamma}{\Gamma \vdash x \uparrow A}$$

$$\frac{\Gamma \vdash I \uparrow A \rightarrow B \quad \Gamma \vdash C \downarrow A}{\Gamma \vdash IC \uparrow B}$$

$$\frac{\Gamma \vdash C \downarrow A}{\Gamma \vdash (C:A) \uparrow A}$$

$$\frac{\Gamma \vdash I \uparrow A \wedge B}{\Gamma \vdash I \uparrow A}$$

$$\frac{\Gamma \vdash I \uparrow A \wedge B}{\Gamma \vdash I \uparrow B}$$

- Checkable ( $C_v$  a checkable value)

$$\frac{\Gamma \vdash I \uparrow A \quad A \leq B}{\Gamma \vdash I \downarrow B}$$

$$\frac{\Gamma \vdash C_v \downarrow A \quad \Gamma \vdash C_v \downarrow B}{\Gamma \vdash C_v \downarrow A \wedge B}$$

$$\frac{\Gamma, x:A \vdash M \downarrow B}{\Gamma \vdash \lambda x. M \downarrow A \rightarrow B}$$

# Pragmatics

---

- No distributivity: sometimes more explicit types.
- Bi-directionality: sometimes lift local functions.
- Boolean constraints for efficient implementation (speculative)

parametric polymorphism

type variable

unification

intersection polymorphism

boolean variable

boolean constraint simplification

## Another Example

---

- Converting a bit string to standard form.

$$\begin{aligned} \mathit{stdize} & : \text{bits} \rightarrow \text{nat} \\ & = \text{fix } \mathit{stdize}. \lambda b. \text{case } b \\ & \quad \text{of } \epsilon \Rightarrow \epsilon \\ & \quad | x \mathbf{0} \Rightarrow \text{case } \mathit{stdize} \ x \\ & \quad \quad \text{of } \epsilon \Rightarrow \epsilon \\ & \quad \quad | y \mathbf{0} \Rightarrow y \mathbf{00} \\ & \quad \quad | y \mathbf{1} \Rightarrow y \mathbf{10} \\ & \quad | x \mathbf{1} \Rightarrow (\mathit{stdize} \ x) \mathbf{1} \end{aligned}$$

- Possible sequential pattern matching in second case.

## Preliminary Assessment

---

- + Elegance
- +? Generality (some rewriting, e.g. tests  $x = \mathbf{nil}$ )
  - + Brevity (proportional to complexity of invariant)
- +? Practicality of verification (interaction with polymorphism?)
  - ! Full inference is decidable via abstract interpretation [Freeman'94], but captures too many accidental properties.
- + Explicitness (clean at module boundary)

# Adding Parametric Polymorphism

---

Types  $A ::= \dots \mid \alpha \mid \forall\alpha. A$

- Subtyping

$$\begin{array}{c} \overline{\forall\alpha. A \leq [B/\alpha]A} \\ \overline{A_1 \wedge A_2 \leq A_1} \quad \overline{A_1 \wedge A_2 \leq A_2} \\ \frac{A \leq B}{A \leq \forall\alpha. B} \alpha \notin \text{FV}(A) \quad \frac{A \leq B_1 \quad A \leq B_2}{A \leq B_1 \wedge B_2} \end{array}$$

- Distributivity is unsound.

$$\left[ \overline{\forall\alpha. (A \rightarrow B) \leq A \rightarrow \forall\alpha. B} \alpha \notin \text{FV}(A) \right]$$

# Structural Subtyping (Sound & Complete)

---

$$\overline{A \sqsubseteq A}$$

$$\overline{\text{pos} \sqsubseteq \text{nat}} \quad \overline{\text{pos} \sqsubseteq \text{bits}} \quad \overline{\text{nat} \sqsubseteq \text{bits}}$$

$$\frac{B_1 \sqsubseteq A_1 \quad A_2 \sqsubseteq B_2}{A_1 \rightarrow A_2 \sqsubseteq B_1 \rightarrow B_2} \quad \frac{A \sqsubseteq B \quad B \sqsubseteq A}{A \text{ ref} \sqsubseteq B \text{ ref}}$$

$$\frac{A_1 \sqsubseteq B^0}{A_1 \wedge A_2 \sqsubseteq B^0} \quad \frac{A_2 \sqsubseteq B^0}{A_1 \wedge A_2 \sqsubseteq B^0} \quad \frac{A \sqsubseteq B_1 \quad A \sqsubseteq B_1}{A \sqsubseteq B_1 \wedge B_2}$$

$$\frac{[A'/\alpha]A \sqsubseteq B^0}{\forall \alpha. A \sqsubseteq B^0} \quad \frac{A \sqsubseteq B}{A \sqsubseteq \forall \alpha. B} \quad (\alpha \notin \text{FVA})$$

$B^0 \neq \forall x. B_1$  and  $B^0 \neq B_1 \wedge B_2$

## Properties of Subtyping

---

- With distributivity have [Mitchell'88].
- Subtyping then undecidable [Tiuryn & Urzyczyn'96] [Wells'95].
- Without distributivity have structural subtyping.
- Undecidable [Chrzęszcz'98].
- Orthogonal to other type constructors.

## Value Restriction

---

- Introduction rule

$$\frac{\Gamma \vdash V : A}{\Gamma \vdash V : \forall \alpha. A} \alpha \notin \text{FV}(\Gamma)$$

- Elimination via subtyping (unchanged)

$$\frac{\Gamma \vdash M : A \quad A \leq B}{\Gamma \vdash M : B}$$

# Unsoundness of Distributivity

---

- Counterexample:

$\vdash \lambda u. \text{ref}(\lambda y. y) \quad : \quad \forall \alpha. \text{unit} \rightarrow (\alpha \rightarrow \alpha) \text{ ref}$

by distributivity and subsumption:

$\vdash \lambda u. \text{ref}(\lambda y. y) \quad : \quad \text{unit} \rightarrow \forall \alpha. (\alpha \rightarrow \alpha) \text{ ref}$

$\vdash (\lambda u. \text{ref}(\lambda y. y)) \langle \rangle \quad : \quad \forall \alpha. (\alpha \rightarrow \alpha) \text{ ref}$

- In a program:

let  $x = (\lambda u. \text{ref}(\lambda y. y)) \langle \rangle \quad : \quad \forall \alpha. (\alpha \rightarrow \alpha) \text{ ref}$

in ... end *% as on slide 5*

# Results

---

- **Lemma:** Typing inversion extends (without distributivity).
- **Theorem:** Progress and type preservation extend (with value restriction).
- New(?) view of value restriction and polymorphism.



## Example with Mutable References

---

$\text{val } \textit{count}'$  : (nat ref  $\rightarrow$  (unit  $\rightarrow$  nat))  $\wedge$   
(pos ref  $\rightarrow$  (unit  $\rightarrow$  pos))  
 $= \lambda c. \lambda x.$   
    let  $y = !c$   
    in  $c := \textit{inc } y; y$  end

$\text{val } \textit{count}$  : (nat  $\rightarrow$  (unit  $\rightarrow$  nat))  $\wedge$   
(pos  $\rightarrow$  (unit  $\rightarrow$  pos))  
 $= \lambda n. \textit{count}'$  (ref  $n$ )

## Other Examples

---

- More programs

$\text{val } plus \quad : \quad (\text{nat} \rightarrow \text{nat} \rightarrow \text{nat}) \wedge$   
 $\quad \quad \quad (\text{pos} \rightarrow \text{nat} \rightarrow \text{pos}) \wedge$   
 $\quad \quad \quad (\text{nat} \rightarrow \text{pos} \rightarrow \text{nat})$

$\text{val } double \quad : \quad (\text{nat} \rightarrow \text{nat}) \wedge (\text{pos} \rightarrow \text{pos})$

$\text{val } stdize \quad : \quad \text{bits} \rightarrow \text{nat}$

$\text{val } \omega \quad : \quad \forall \alpha. \forall \beta. ((\alpha \rightarrow \beta) \wedge \alpha) \rightarrow \beta$   
 $\quad = \quad \lambda x. x x \quad (\text{without refinement restriction})$

- More refinements

$\text{zero} ::= \epsilon$

$\text{even} ::= \epsilon \mid \text{pos } \mathbf{0}$

$\text{odd} ::= \text{nat } \mathbf{1}$

## Host Language Dependence

---

- Interesting differences: call-by-value vs. call-by-name

*Lists*       $\alpha \text{ list} ::= \mathbf{nil} \mid \mathbf{cons}(\alpha, \alpha \text{ list})$

*Even*       $\alpha \text{ even} ::= \mathbf{nil} \mid \mathbf{cons}(\alpha, \alpha \text{ odd})$

*Odd*       $\alpha \text{ odd} ::= \mathbf{cons}(\alpha, \alpha \text{ even})$

- In call-by-value:  $\alpha \text{ even} \wedge \alpha \text{ odd} = \perp$
- In call-by-name:  $\vdash \text{fix } \omega. \mathbf{cons}(\langle \rangle, \omega) : \text{unit even} \wedge \text{unit odd}$
- Combined with dependent types in logical framework LF [Pf.'93] [Pf. & Kohlase'93]

## Related Work

---

- Intersection types (many)
- Forsythe [Reynolds'88] [Reynolds'96]
- Intersections and explicit polymorphism [Pierce'91] [Pierce'97]
- Refinement types [Freeman & Pf'91] [Freeman'94] [Davies'97]
- Intersection types and program analysis (many)
- Soft types (many)
- Local type inference [Pierce & Turner'97]
- Shape analysis and software model checking.

## Future Work

---

- Sequential pattern matching.
- Complete implementation under refinement restriction.
- Local type inference with intersections and parametric polymorphism?
- Valuability instead of values? [Harper & Stone'00]
- Pure and impure function spaces?

## Summary

---

### **Refinement types to statically verify program invariants.**

- Between simple types and full specifications.
- Subtyping and intersections required.
- Simplified type system for soundness with effects.
- Progress theorem holds.
- Effective bi-directional type checking.
- Applied techniques to parametric polymorphism.