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.
- Suprisingly 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):

$$\textit{Bit Strings} \qquad \text{bits} \ ::= \ \epsilon \mid \text{bits}\,\mathbf{1} \mid \text{bits}\,\mathbf{0}$$

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

$$\begin{array}{lrcl} \textit{Natural Numbers} & \text{nat} & ::= & \epsilon \mid \text{pos} \\ \textit{Positive Numbers} & \text{pos} & ::= & \text{pos}\,\mathbf{0} \mid \text{nat}\,\mathbf{1} \end{array}$$

The Need for Subtyping and Intersections

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

$$\begin{array}{rcl} \vdash \lambda x.\, x\,\mathbf{0} & : & \mathsf{bits} \to \mathsf{bits} \\ \vdash \lambda x.\, x\,\mathbf{0} & : & \mathsf{nat} \to \mathsf{bits} \\ \vdash \lambda x.\, x\,\mathbf{0} & : & \mathsf{pos} \to \mathsf{pos} \end{array}$$

- Intersections allow these to be expressed simultaneously.

$$\begin{array}{rcl} \vdash \lambda x.\, x\,\mathbf{0} & : & (\mathsf{bits} \to \mathsf{bits}) \\ & & \wedge\, (\mathsf{nat} \to \mathsf{bits}) \\ & & \wedge\, (\mathsf{pos} \to \mathsf{pos}) \\ \not\vdash \lambda x.\, x\,\mathbf{0} & : & \mathsf{nat} \to \mathsf{nat} \qquad (!) \end{array}$$

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

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

- Elimination rules

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

Subtyping and Greatest Lower Bounds

- Subsumption

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

- Intersection as a greatest lower bound

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

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

- Elimination rules now derivable

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

Intersections are Unsound with Effects

- Counterexample

$$
\begin{array}{rll}
\text{let} & x = \text{ref}\,(\epsilon\,\mathbf{1}) & : \text{nat ref} \wedge \text{pos ref} \\
\text{in} & & \\
 & x := \epsilon; & \%\ \textit{use}\ x : \text{nat ref} \\
 & !\,x & \%\ \textit{use}\ x : \text{pos ref} \\
\text{end} & : \text{pos} &
\end{array}
$$

  evaluates to ϵ which does not have type pos.

- Analogous counterexample with parametric polymorphism:

$$
\begin{array}{rll}
\text{let} & x = \text{ref}\,(\lambda y.\,y) & : \forall\alpha.\,(\alpha \to \alpha)\,\text{ref} \\
\text{in} & & \\
 & x := (\lambda y.\,\epsilon); & \%\ \textit{use}\ x : (\text{nat} \to \text{nat})\,\text{ref} \\
 & (!\,x)\,(\epsilon\,\mathbf{1}) & \%\ \textit{use}\ x : (\text{pos} \to \text{pos})\,\text{ref} \\
\text{end} & : \text{pos} &
\end{array}
$$

Subtyping

$$\textit{Types} \quad A \quad ::= \quad \mathsf{bits} \mid \mathsf{nat} \mid \mathsf{pos} \\ \mid A_1 \to A_2 \mid A\,\mathsf{ref} \mid \mathsf{unit} \\ \mid A_1 \wedge A_2$$

$$\frac{}{A \le A} \qquad \frac{A \le B \qquad B \le C}{A \le C}$$

$\le$: Reflexive and transitive

$$\frac{B_1 \le A_1 \qquad A_2 \le B_2}{A_1 \to A_2 \le B_1 \to B_2}$$

$\to$: Contra- and co-variant

$$\frac{A \le B \quad B \le A}{A\,\mathsf{ref} \le B\,\mathsf{ref}}$$

ref: Non-variant

Subtyping and Intersections

$$\frac{}{\mathsf{pos} \leq \mathsf{nat}} \qquad \frac{}{\mathsf{nat} \leq \mathsf{bits}}$$

Data types

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

$\wedge$: Lower bound

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

$\wedge$: Greatest lower bound

$$\left[\frac{}{(A \to B) \wedge (A \to C) \leq A \to (B \wedge C)}\right]$$

?? (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):

$$\frac{}{\Gamma \vdash \epsilon : \mathsf{nat}}$$

$$\frac{\Gamma \vdash M : \mathsf{pos}}{\Gamma \vdash M\,\mathbf{0} : \mathsf{pos}} \qquad \frac{\Gamma \vdash M : \mathsf{bits}}{\Gamma \vdash M\,\mathbf{0} : \mathsf{bits}}$$

$$\frac{\Gamma \vdash M : \mathsf{nat}}{\Gamma \vdash M\,\mathbf{1} : \mathsf{pos}} \qquad \frac{\Gamma \vdash M : \mathsf{bits}}{\Gamma \vdash M\,\mathbf{1} : \mathsf{bits}}$$

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

- Note: $\mathsf{case}\,(M{:}\mathsf{pos})$ 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.

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

  where

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

- Originally introduced for parametric polymorphism [Tofte'90] [Wright'95].

- Value restriction here not tied to let!

$$\frac{\Gamma \vdash M : A \qquad \Gamma, x{:}A \vdash N : B}{\Gamma \vdash \mathsf{let}\, x = M \;\mathsf{in}\; N \,\mathsf{end} : B}$$

Counterexample Revisited

$$
\begin{array}{rll}
\text{let} & x = \mathsf{ref}\,(\epsilon\,\mathbf{1}) & : \mathsf{nat\ ref} \wedge \mathsf{pos\ ref} \\
\text{in} & & \\
& x := \epsilon; & \%\ \textit{use}\ x : \mathsf{nat\ ref} \\
& !\,x & \%\ \textit{use}\ x : \mathsf{pos\ ref} \\
\text{end} & : \mathsf{pos} &
\end{array}
$$

- No longer well typed:

$$\nvdash \mathsf{ref}\,(\epsilon\,\mathbf{1}) : \mathsf{nat\ ref} \wedge \mathsf{pos\ ref}$$

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

Distributivity Revisited

- Distributivity is unsound with effects.

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

- Counterexample:

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

by distributivity and subsumption:

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

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

- In a program:

let $x = (\lambda u.\,\mathsf{ref}\,(\epsilon\,\mathbf{1}))\,\langle\,\rangle$: nat ref $\wedge$ pos ref

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

Results

- **Theorem:** Subtyping is structural.
- **Lemma:** *(Typing Inversion)* With a *store typing* Δ:
  1. If $\Delta; \cdot \vdash V : A$ and $A \leq B \to C$
     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:

$$\vdash \mathsf{ref}\,(\epsilon\,\mathbf{1}) \;\;:\;\; \mathsf{bits\ ref}$$

$$\vdash \mathsf{ref}\,(\epsilon\,\mathbf{1}) \;\;:\;\; \mathsf{nat\ ref}$$

$$\vdash \mathsf{ref}\,(\epsilon\,\mathbf{1}) \;\;:\;\; \mathsf{pos\ ref}$$

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

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

Bi-Directional Type-Checking

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

$$\begin{array}{lrcl} \text{Inferable} & I & ::= & x \mid I\,C \mid C{:}A \\ \text{Checkable} & C & ::= & I \mid \lambda x.\,C \end{array}$$

- 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

$$\frac{x{:}A \text{ in } \Gamma}{\Gamma \vdash x \uparrow A} \qquad \frac{\Gamma \vdash I \uparrow A \to B \qquad \Gamma \vdash C \downarrow A}{\Gamma \vdash I\,C \uparrow B}$$

$$\frac{\Gamma \vdash C \downarrow A}{\Gamma \vdash (C{:}A) \uparrow A} \qquad \frac{\Gamma \vdash I \uparrow A \wedge B}{\Gamma \vdash I \uparrow A} \qquad \frac{\Gamma \vdash I \uparrow A \wedge B}{\Gamma \vdash I \uparrow B}$$

- Checkable (C_v a checkable value)

$$\frac{\Gamma \vdash I \uparrow A \qquad A \leq B}{\Gamma \vdash I \downarrow B} \qquad \frac{\Gamma \vdash C_v \downarrow A \qquad \Gamma \vdash C_v \downarrow B}{\Gamma \vdash C_v \downarrow A \wedge B}$$

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

Pragmatics

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

parametric polymorphism	intersection polymorphism
type variable	boolean variable
unification	boolean constraint simplification

Another Example

- Converting a bit string to standard form.

$$
\begin{array}{lcl}
\mathit{stdize} & : & \mathsf{bits} \to \mathsf{nat} \\
 & = & \mathsf{fix}\ \mathit{stdize}.\ \lambda b.\, \mathsf{case}\ b \\
 & & \qquad\qquad\qquad\qquad \mathsf{of}\ \epsilon \Rightarrow \epsilon \\
 & & \qquad\qquad\qquad\qquad\ \mid x\,\mathbf{0} \Rightarrow\ \mathsf{case}\ \mathit{stdize}\ x \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{of}\ \epsilon \Rightarrow \epsilon \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\quad\ \mid y\,\mathbf{0} \Rightarrow y\,\mathbf{0}\,\mathbf{0} \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\quad\ \mid y\,\mathbf{1} \Rightarrow y\,\mathbf{1}\,\mathbf{0} \\
 & & \qquad\qquad\qquad\qquad\ \mid x\,\mathbf{1} \Rightarrow (\mathit{stdize}\ x)\,\mathbf{1}
\end{array}
$$

- Possible sequential pattern matching in second case.

Preliminary Assessment

+ Elegance

+? Generality (some rewriting, e.g. tests $x = \textbf{nil}$)

+ 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

$$\textit{Types} \quad A \quad ::= \quad \ldots \mid \alpha \mid \forall\alpha.\,A$$

- Subtyping

$$\frac{}{\forall\alpha.\,A \leq [B/\alpha]A} \qquad \frac{}{A_1 \wedge A_2 \leq A_1} \qquad \frac{}{A_1 \wedge A_2 \leq A_2}$$

$$\frac{A \leq B}{A \leq \forall\alpha.\,B}\,\alpha \notin \mathsf{FV}(A) \qquad \frac{A \leq B_1 \qquad A \leq B_2}{A \leq B_1 \wedge B_2}$$

- Distributivity is unsound.

$$\left[\frac{}{\forall\alpha.\,(A \to B) \leq A \to \forall\alpha.\,B}\,\alpha \notin \mathsf{FV}(A)\right]$$

Structural Subtyping (Sound & Complete)

$$\frac{}{A \trianglelefteq A}$$

$$\frac{}{\textsf{pos} \trianglelefteq \textsf{nat}} \qquad \frac{}{\textsf{pos} \trianglelefteq \textsf{bits}} \qquad \frac{}{\textsf{nat} \trianglelefteq \textsf{bits}}$$

$$\frac{B_1 \trianglelefteq A_1 \qquad A_2 \trianglelefteq B_2}{A_1 \to A_2 \trianglelefteq B_1 \to B_2} \qquad \frac{A \trianglelefteq B \qquad B \trianglelefteq A}{A\ \textsf{ref} \trianglelefteq B\ \textsf{ref}}$$

$$\frac{A_1 \trianglelefteq B^o}{A_1 \wedge A_2 \trianglelefteq B^o} \qquad \frac{A_2 \trianglelefteq B^o}{A_1 \wedge A_2 \trianglelefteq B^o} \qquad \frac{A \trianglelefteq B_1 \qquad A \trianglelefteq B_1}{A \trianglelefteq B_1 \wedge B_2}$$

$$\frac{[A'/\alpha]A \trianglelefteq B^o}{\forall\alpha.\, A \trianglelefteq B^o} \qquad \frac{A \trianglelefteq B}{A \trianglelefteq \forall\alpha.\, B}\,(\alpha \notin \textsf{FV}A)$$

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

Properties of Subtyping

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

Value Restriction

- Introduction rule

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

- Elimination via subtyping (unchanged)

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

Unsoundness of Distributivity

- Counterexample:

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

by distributivity and subsumption:

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

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

- In a program:

$$\begin{array}{ll} \mathsf{let} & x = (\lambda u.\, \mathsf{ref}\,(\lambda y.\, y))\,\langle\,\rangle \quad : \forall \alpha.\, (\alpha \rightarrow \alpha)\ \mathsf{ref} \\ \mathsf{in} & \ldots\ \mathsf{end} \qquad\qquad\qquad\qquad \textit{\% as on slide 5} \end{array}$$

Results

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

Example: External vs Internal Invariants

$$\begin{array}{rcl}
\text{val } inc & : & (\text{bits} \to \text{bits}) \wedge (\text{nat} \to \text{pos}) \\
& = & \text{fix } inc.\, \lambda n.\ \text{case } n \\
& & \qquad\qquad\quad \text{of } \epsilon \Rightarrow \epsilon\,\mathbf{1} \\
& & \qquad\qquad\quad\ \ \mid x\,\mathbf{0} \Rightarrow x\,\mathbf{1} \\
& & \qquad\qquad\quad\ \ \mid x\,\mathbf{1} \Rightarrow (inc\ x)\,\mathbf{0} \\
\vdash inc & : & \text{nat} \to \text{nat} \qquad \%\ \textit{by subtyping} \\
\vdash inc & : & \text{pos} \to \text{pos} \qquad \%\ \textit{by subtyping}
\end{array}$$

$$\begin{array}{rcl}
\text{val } inc & \not: & \text{nat} \to \text{nat} \\
& = & \text{fix } inc.\, \lambda n.\ \text{case } n \\
& & \qquad\qquad\quad \text{of } \epsilon \Rightarrow \epsilon\,\mathbf{1} \\
& & \qquad\qquad\quad\ \ \mid x\,\mathbf{0} \Rightarrow x\,\mathbf{1} \\
& & \qquad\qquad\quad\ \ \mid x\,\mathbf{1} \Rightarrow (inc\ x)\,\mathbf{0} \qquad \%\ inc\,x : \text{pos?}
\end{array}$$

Example with Mutable References

$$
\begin{aligned}
\text{val } count' \quad &: \quad (\mathsf{nat\ ref} \to (\mathsf{unit} \to \mathsf{nat})) \wedge \\
&\quad\;\; (\mathsf{pos\ ref} \to (\mathsf{unit} \to \mathsf{pos})) \\
&= \quad \lambda c.\, \lambda x. \\
&\qquad\quad \mathsf{let}\ y = !\, c \\
&\qquad\quad \mathsf{in}\ c := inc\ y;\ \ y\ \mathsf{end}
\end{aligned}
$$

$$
\begin{aligned}
\text{val } count \quad &: \quad (\mathsf{nat} \to (\mathsf{unit} \to \mathsf{nat})) \wedge \\
&\quad\;\; (\mathsf{pos} \to (\mathsf{unit} \to \mathsf{pos})) \\
&= \quad \lambda n.\, count'\ (\mathsf{ref}\ n)
\end{aligned}
$$

Other Examples

- More programs

$$
\begin{array}{rcl}
\text{val } plus & : & (\text{nat} \to \text{nat} \to \text{nat}) \wedge \\
 & & (\text{pos} \to \text{nat} \to \text{pos}) \wedge \\
 & & (\text{nat} \to \text{pos} \to \text{nat}) \\
\text{val } double & : & (\text{nat} \to \text{nat}) \wedge (\text{pos} \to \text{pos}) \\
\text{val } stdize & : & \text{bits} \to \text{nat} \\
\text{val } \omega & : & \forall\alpha.\, \forall\beta.\, ((\alpha \to \beta) \wedge \alpha) \to \beta \\
 & = & \lambda x.\, x\, x \qquad \textit{(without refinement restriction)}
\end{array}
$$

- More refinements

$$
\begin{array}{rcl}
\text{zero} & ::= & \epsilon \\
\text{even} & ::= & \epsilon \mid \text{pos}\,\mathbf{0} \\
\text{odd} & ::= & \text{nat}\,\mathbf{1}
\end{array}
$$

Host Language Dependence

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

$$
\begin{array}{lrcl}
\textit{Lists} & \alpha\,\mathsf{list} & ::= & \mathbf{nil} \mid \mathbf{cons}(\alpha, \alpha\,\mathsf{list}) \\
\textit{Even} & \alpha\,\mathsf{even} & ::= & \mathbf{nil} \mid \mathbf{cons}(\alpha, \alpha\,\mathsf{odd}) \\
\textit{Odd} & \alpha\,\mathsf{odd} & ::= & \mathbf{cons}(\alpha, \alpha\,\mathsf{even})
\end{array}
$$

- In call-by-value: $\alpha\,\mathsf{even} \wedge \alpha\,\mathsf{odd} = \bot$
- In call-by-name: $\vdash \mathsf{fix}\,\omega.\,\mathbf{cons}(\langle\,\rangle, \omega) : \mathsf{unit}\,\mathsf{even} \wedge \mathsf{unit}\,\mathsf{odd}$
- 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.