Type Safety

Robert Harper
Carnegie Mellon University
Spring 1999

What is a safe language?

- Programmer's view: nothing goes wrong.
  - No core dumps.
  - No ill-defined behavior.
  - No implementation-dependent behavior.
- Compiler-writer's view: well-behaved run-time environment.
  - No casts / confusions of values.
  - No stray or dangling pointers.
  - No freeing of active memory, no leaks of unused memory.

What is a Safe Language?

- Language designer's view: coherence of static and dynamic semantics.
  - No "stuck" states.
  - No "wild" transitions.
  - No reliance on a machine model.

Confounding Issues

- Domain errors / partiality
  - division by zero
  - subscript out of bounds
- Exhausting capacity
  - arithmetic overflow
  - out of memory
- Loopholes
  - downcasting in Java

Varieties of Errors

- Trapped errors are detected.
  - Cleanly abort computation.
  - Raise exceptions (may be handled).
- Untrapped errors are not detected.
  - Cause core dumps, unpredictable behavior.

Achieving Safety

- Safety = no untrapped errors.
  - Thus safe languages exhibit predictable, portable behavior.
- How is safety achieved?
  - Static checking: forbid all untrapped errors (and some trapped errors).
  - Dynamic checking: ensure that all remaining errors are trapped.

Dynamic Checking

- Dynamic checks are very costly!
  - Primitive operations must check validity each time they are applied.
  - Must pay an "up front" cost to make dynamic checks possible.
- It must be possible to check for ill-defined situations!
  - Often requires "tagging" values.

Dynamic Checking

- Some checks are easy and obvious.
  - Check for 0 divisor.
  - Array bounds checks.
- Others are not.
  - Is this integer a file descriptor?
  - Is this address a valid code pointer?
  - Is this double word a timestamp?

Dynamic Checking

- Dynamic checking usually relies on tagged values.
  - Every value is tagged with an identifier.
    - Eg, "I am a file descriptor."
  - Only trusted code can put tags on values.
    - No forgeries are possible.
  - Primitives all check for expected tag.
    - Raise exception if not.

Dynamic Checking

- Obvious costs:
  - Repeated run-time checks for validity.
  - Space for tags, time for creation.
- Non-obvious cost:
  - All objects must be tagged, even if they are not passed to unsafe operations.
- Violates "pay-as-you-go" principal.

Static Checking

- Static checking = conformance with a type system.
  - Context-sensitive constraints on the formation of programs.
- Type systems rule out all untrapped errors and some trapped errors.
  - Dynamic checks are usually required for the rest.

Static Checking

- Type system predicts the "shape" of objects at run-time.
  - If e:int, then e must evaluate to an integer (not to a boolean or a function).
  - If e:int->int, then e must evaluate to a function (not a string).

Static Checking

- Consequently, no tag bits or run-time checks are needed.
  - No need for "I am an integer" or "I am a function".
  - Native representations are possible, eg int's as machine words.
- Validates "pay-as-you-go" principal: no hidden costs for unused features.

Static Checking

- Goals of type system design:
  - Rule out as many errors as possible ...
  - ... while preserving safety.
- Fundamental tensions:
  - (feasible) decidability of type checking
  - expressiveness of the type system
  - flexibility of the language

Static Checking

- Some benefits of type safety:
  - Economy of execution: fewer run-time checks, fewer "up front" costs.
  - Economy of development: type checking catches many bugs; interfaces = module types mediate interaction.
  - Economy of features: prog lang = a collection of orthogonal types.

Design of Type Systems

- How do you know that a type system is sound?
  - Ie, does it forbid untrapped errors?
- As expressiveness increases, it gets harder to know if you're right.
  - There are many famous errors -- type systems that "look good" but are in fact unsound.

Type Safety for MinML

- Claim: MinML is type safe.
  - "Well-typed programs cannot go wrong."
  - That is, no unchecked errors can arise.
- How can we make this claim precise?
  - Formalize the notion of type safety.
  - Prove that the language is type safe.

Type Safety for MinML

- Preservation: evaluation steps preserve the type of the expression.
  - No "wild" transitions.
- Progress: every expression is either a value or can make a valid step of computation.
  - No "stuck" states.

Preservation Theorem

- Theorem: If $\vdash e : \tau$ and $e \mapsto e'$, then $\vdash e' : \tau$.
- Proof is by induction on the definition of the transition relation $e \mapsto e'$.
  - Consider each rule in turn.
  - Assume theorem for premises of the rule, derive it for conclusions of the rule.

Preservation: Instructions

- e = if true then e1 else e2 $\mapsto$ e1 = e'.
- Assume e:τ, check e':τ.
- By inversion, we have e1 : τ and e2 : τ.
- Hence, e' : τ, as required.
- (false case is similar)

Preservation: Instructions

- e = apply(v,v1), e' = [v,v1/f,x]e2, where v = fun f(x: τ1):τ is e2 end.
- Assume e: τ, show e' : τ.
- By inversion, v : τ1 -> τ and v1 : τ1 (for some type τ1).
- By inversion, f: τ1-> τ, x: τ1 $\vdash$ e2 : τ.
- By substitution, [v,v1/f,x] e2 : τ, as required.

Preservation: Instructions

- e = o(v1,…,vn) $\mapsto$ v = e', where v is the result of o on arguments v1, …, vn.
  - Eg, m+n $\mapsto$ p, where p is sum of m and n
- Assume e:τ, show e':τ.
- By inversion, vi : τi, where o maps τ1, … τn to τ.
- That v:τ is simply postulated!
  - Eg, + maps int's to int's

Preservation: Search

- e = apply(e1,e2) $\mapsto$ apply(e1',e2) = e', where e1 $\mapsto$ e1'.
- Assume e:τ, show e':τ.
- By inversion, e1 : τ2 -> τ and e2 : τ2 (for some type τ2).
- By induction, e1' : τ2 -> τ.
- Therefore, e' : τ, as required.

Preservation: Search

- e = apply(v1,e2) apply(v1, e2') = e', where e2 $\mapsto$ e2'.
- Assume e:τ, show e':τ.
- By inversion, v1 : τ2 -> τ and e2 : τ2.
- By induction e2' : τ2.
- Therefore e' : τ, as required.

Preservation Theorem

- What does this prove?
  - Typing is stable under evaluation.
  - No "wild" transitions are possible.
- Therefore if e:τ and e ↦* v, then v: τ.
  - If e has a value, then the value has the same type as e.
  - Vacuously true if e has no value!
  - This is called partial correctness.

Canonical Forms Lemma

- Lemma: Suppose that v : τ.
  - If τ =int, then v = n for some n.
  - If τ =bool, then v = true or v = false.
  - If τ = τ1 -> τ2, then for some e, v = fun f(x:τ1):τ2 is e end.
- Proof: By induction on the typing rules and the definition of values.

Canonical Forms Lemma

- The CF Lemma tells us that we can predict the "shapes" of values.
  - If v:int, then v is a number.
  - If v:bool, then v is a boolean constant.
  - If v: τ1 -> τ2, then v is a function.
- Consequently, no run-time checks are required for primop's, conditionals, or applications!

Canonical Forms Lemma

- If apply(v1,v2) is well-typed, then v1 must have function type.
- By the canonical forms lemma, v1 must be a function expression.
- Therefore we can proceed by applying it using the standard calling sequence (substitution for "self" and parameter).

Progress Theorem

- Theorem If e:τ, then either e is a value or there exists e' st e ↦ e'.
- If e is well-typed, then either
  - it is fully-evaluated, and hence "done", or
  - there is a well-defined "next step" to take.
- That is, well-typed programs cannot "get stuck".

Progress Theorem

- Examples of "stuck" expressions:
  - apply (1, 2)
  - true + 4
  - if 17 then 1 else 2
  - (fun f(x:int):int is x-1) + 7
- Each is ill-typed!
  - Contrapositive of progress theorem: if e is "stuck", then e is ill-typed!

Progress Theorem

- Proof of progress proceeds by induction on typing derivations.
- Consider each typing rule in turn.
- Argue that either the expression is a value or that progress can be made.
- Relies on two facts:
  - canonical forms lemma
  - there are "enough" search rules

Progress Theorem

- n : int.
  - Numbers are values.
- true : bool, false : bool.
  - Boolean constants are values.
- fun f(x: τ1):τ2 is e end : τ1 -> τ2, where f: τ1 -> τ2, x: τ1 $\vdash$ e : τ2.
  - Function expressions are values.

Progress Theorem

- o(e_1,...,e_n):τ, where e_i: τ_i.
- By induction either e_i is a value vi or $e_i \mapsto e_i'$ for each $1 \le i \le n$.
- If all are values, then we assume there exists v such that $o(v_1,..., v_n) \mapsto v$.
- Otherwise, let e_i be the first non-value, so that e = $o(v_1,..., v_i\text{-}1, e_i,..., e_n)$.
- Then $e \mapsto e'$, where $e' = o(v_1,..., v_i\text{-}1, e_i',..., e_n)$.

Progress Theorem

- e = if e_1 then e_2 else e_3 : τ, where e_1 : bool, e_2 : τ, and e_3 : τ.
- By induction either e_1 is a value, or there exists e_1' such that $e_1 \mapsto e_1'$.
- If e_1 is a value, then by the CF Lemma, e_1=true or e_1=false, so either $e \mapsto e_2$ or $e \mapsto e_3$, respectively.
- Otherwise $e \mapsto$ if e_1' then e_2 else e_3.

Progress Theorem

- e = apply(e_1, e_2): τ, where e_1 : τ_2 -> τ and e_2 : τ_2.
- By induction, either e_1 is a value v_1, or else $e_1 \mapsto e_1'$ for some e_1'.
- If e_1 is not a value, then $e \mapsto$ apply(e_1', e_2).
- If e1 is a value v1, then by the CF Lemma, v1 = fun f(x: τ2): τ is e' end.
- By induction either e2 is a value v2 or there exists e2' such that $e2 \mapsto e2'$.
- In the latter case $e \mapsto$ apply(v1, e2').
- In the former, e = apply(v1, v2) $\mapsto$ [v1, v2/f,x]e'.

Summary

- MinML is type safe.
  - If a step can be taken, typing is preserved.
  - If an expression is well-typed and not a value, it cannot be "stuck".
- Proviso: we assume that
  - the primop's behave properly
  - the primop's are total functions.

A Failure of Soundness

- Type systems aren't always sound!
  - ML's treatment of polymorphic references.
  - Eiffel's treatment of inheritance.
  - Java's treatment of arrays.
- Appearances can be very deceiving!
  - Progress and Preservation Theorems are tools for finding subtle problems.

A Failure of Soundness

- Standard ML restricts polymorphism to values to avoid an infamous error.
- An example of what can go wrong:

```
let
  val r = ref nil
in
  r := [1]; not (hd (!r))
end
```


A Failure of Soundness

- What's wrong here?
  - Program is type correct.
  - Yet it "goes wrong" by applying not to an integer (rather than a boolean).
- The problem is the interaction between features.
  - Polymorphic let bindings.
  - References (more generally, effects).

A Failure of Soundness

- Formally, the progress theorem fails to hold.
  - Wind up with not(1), which is ill-defined.
- The implementation, which assumes soundness, dumps core when executed.
  - Ill-defined operation.

More on Type Safety

Robert Harper
Carnegie Mellon University
Spring 1999

Recap: Type Safety

- Preservation Theorem: the type of a program is preserved by evaluation.
  - No "wild" transitions are possible.
- Progress Theorem: well-typed programs are either values or can take a valid execution step.
  - No "stuck" programs.

Recap: Type Safety

- We relied on two key assumptions about the primitive operations!
- Suppose that o maps $\tau 1, \ldots, \tau n$ to τ, and that v1: $\tau 1$, …, vn: τn are values.
  - if o(v1,…, vn) evaluates to v, then v:t.
  - there exists v such that o(v1,…, vn) evaluates to v.

Recap: Type Safety

- The first is a natural, and necessary, assumption.
  - e.g., if m/n evaluates to p, then p is an integer
- The second is questionable, since it expresses totality of the primop's.
  - e.g., m/n is not defined if n=0

Dealing with Partiality

- What about partial primop's?
  - For some inputs, there is no value (e.g., zero divisor).
  - Progress fails in this case: n/0 is "stuck", yet it is well-typed!
- Two solutions:
  - strengthen the type system
  - modify the statement of soundness

Strengthening the Type System

- A simple-minded approach:
  - add a type nz_int of non-zero integers
  - add e1/e2 : int if e1:int and e2:nz_int
- But what are the rules for the type nz_int?
  - when is e1+e2:nz_int??
  - when is e1-e2:nz_int??

Strengthening the Type System

- General solution: must prove that value is non-zero.
  - e : nz_int iff $\vdash$ e <> 0
  - Undecidable in general!
- Type checking requires full-scale mathematical theorem proving.
  - e.g. NuPRL system at Cornell, an interactive theorem prover

Modifying Soundness

- Another approach: add error as a possible final result of computation.
  - m / n $\mapsto$ error, if n = 0
  - m / n $\mapsto$ p, if n <> 0, p is m divided by n.
- All operations must propagate error.
  - apply(error, e) $\mapsto$ error
  - apply(v,error) $\mapsto$ error
  - etc

Modifying Soundness

- Define error : τ for any type τ.
  - Consistent with the partial correctness interpretation of typing!
- Observe that the Preservation Theorem continues to hold!
  - Relies on previous assumption.
- But the Progress Theorem fails!
  - error is not a value.

Modifying Soundness

- Revised Progress Theorem: If e : τ, then either e is a value, or e is error, or there exists e' st e $\mapsto$ e'.
- Proof is essentially as before, but two-way case splits become three-way case splits.
  - Need propagation rules to cover error cases.

Modifying Soundness

- For example, spse apply(e1,e2):τ.
- By inversion, e1: τ2->τ, e2:τ.
- By induction e1 is either a value, error, or e1 $\mapsto$ e1'.
- If e1=error, then apply(e1, e2) $\mapsto$ error.
- (Otherwise proceed as before.)

Modifying Soundness

- If e1 is a value v1, then by induction e2 is either a value, or error, or e2 $\mapsto$ e2'.
- If e2=error, then apply(v1, e2) $\mapsto$ error.
- (Otherwise proceed as before.)

A Hybrid Approach

- Modern compilers use a combination of the two methods.
  - Define "checked" and "unchecked" versions of the primop's.
    - checked version checks for domain violation and calls unchecked version
  - Try to prove that the unchecked version is safe to use; if not, use the checked version.

A Hybrid Approach

- For example,
  - unsafe_sub : 'a array * int -> 'a
  - fun safe_sub (A, i) =
    if 0 <= i andalso i <= length(A) then
    unsafe_sub(A,i)
    else
    raise Error
  - compiler tries to replace safe_sub by unsafe_sub where possible.

Dynamic vs Static Typing

- Dynamic typing
  - values are tagged with "type" information
  - primitives check tags at run-time
  - only trusted code can place tags on values
- Slogan: types are associated with values, rather than variables

Dynamic vs Static Typing

- Dynamic typing is often claimed to be an alternative to static typing.
  - Usually, to claim that it is superior to static typing.
- In reality dynamic typing is a special case of static typing.
  - Static type systems are strictly more expressive than dynamic systems!

Case Study: Scheme

- Overall structure:
  - language-based computation model
  - first-class higher-order functions
  - aggregate data structures such as lists and trees
  - dynamic typing
  - no exceptions, no modules, no patterns
  - mutable references

Case Study: Scheme

- Basic constants (plus many more)
  - integers: 43, 17, ...
  - symbols: 'abc, '*abc*
  - booleans: #t, #f
- Basic operations
  - (plus 43 23 17)
  - (zero? (minus 17 (plus 14 3)))

Case Study: Scheme

- Symbolic Expressions (S-Expr's)
  - "cons node": (cons a d)
  - 1st and 2nd components: (car l), (cdr l)
  - empty list: nil
  - empty test: (null? l)
- Lists are s-exprs
  - (cons 1 (cons 2 (cons 3 nil)))
  - (car ...) = 1, (cdr ...) = (cons 2 (cons 3 nil))

Case Study: Scheme

- "Heterogeneous lists"
  - (cons #t (cons 2 (cons '3 nil)))
  - often claimed to be an advantage of Scheme over ML

Case Study: Scheme

- Run-time "type" checks
  - (number? l), (boolean? l), (symbol? l), (null? l), (cons? l), ...
- Primitives perform run-time checks.
  - (plus 3 #f) ==> error
  - (cons 1) ==> error

Cast Study: Scheme

- First-class functions
  - All functions are multi-argument.
  - Functions are values, as in ML.
  - (lambda (x y)
    (plus (times x x) (times y y)))
  - (map (lambda (x) (1+ x))
    (cons 1 (cons 2 nil))) ==>
    (cons 2 (cons 3 nil))

Case Study: Scheme

- Definitions
  - (define l (cons 1 (cons 'a (cons #f nil))))
  - (define length
    (lambda (l)
    (if (null? l) 0
    (plus 1 (length (cdr l))))))
  - (length l) ==> 3

Dynamic Typing in SML

- Key intuition: untyped = uni-typed.
  - Scheme has one type for all values ...
  - ... and it's a recursive datatype!
- The type of Scheme values:
  - datatype SV = Int of int | Bool of bool |
  - Symbol of string | Nil | Cons of SV * SV |
  - Function of SV list -> SV

Dynamic Typing in SML

- That is, Scheme values are
  - integers, tagged with Int
  - booleans, tagged with Bool
  - strings, tagged with Symbol
  - the constant Nil
  - pairs, tagged with Cons
  - functions, tagged with Function

Primitive Operations

- local
- (* loop over arguments *)
- fun raw_plus nil = 0
- | raw_plus (Int m :: ms) = m + raw_plus ms
- | raw_plus _ = raise RunTimeError "plus"
- in
- val plus = Function (Int o raw_plus)
- end
- Note: run-time check for integer arguments!

Primitive Operations

- local
- fun raw_car [Cons (a, _)] = a
- | raw_car _ = raise RunTimeError "car"
- in
- val car = Function raw_car
- end

- Note: run-time check for one argument, and check that argument is a Cons node.

Primitive Operations

- local
- fun raw_cons [a, d] = Cons (a, d)
- | raw_cons _ = raise RunTimeError "cons"
- in
- val cons = Function raw_cons
- end

- Note: run-time check for 2 arguments.

Primitive Operations

- fun apply (Function f, args) = f args
- | apply _ = raise RunTimeError "apply"

- Run-time check for Function.

Examples

- val nill = Nil
- val zero = Int 0 and one = Int 1 and two = Int 2
- val tt = Bool true and ff = Bool false
- (* (cons 0 (cons 1 (cons 2 nil))) *)
- val l =
- apply (cons, [zero,
  apply (cons, [one,
  apply (cons, [two,
  nill])])])
- (* (car l), (car (cdr l)), (car (cdr (cdr l))) *)
- val a = apply (car, [l])
- val b = apply (car, [apply (cdr, [l])])
- val c = apply (car, [apply (cdr, [apply (cdr, [l])])])

Definitions

- In Scheme top-level definitions have type SV option ref.
  - Mutable, to allow re-definition of previous bindings without recompiling.
  - Optional, to allow implementation of recursion by backpatching.
- Penalty: unbound variables at run-time (if binding contains NONE).

Definitions in Scheme

- (* Retrieve the binding of a variable. *)
- fun binding (ref NONE) =
- raise RunTimeError "unbound variable"
- | binding (ref (SOME v)) = v

- Two parts:
  - Retrieve contents of cell.
  - Check for unbound variable.

Definitions in Scheme

```
val length : SV option ref = ref NONE
let
  fun raw_length [v] =
        case apply (null, [v])
          of Bool true => zero
           | Bool false =>
             apply (plus,
                    [one, apply (binding length,
                      [apply (cdr, [v])])])
  val length_function = Function raw_length
in
  length := SOME length_function
end
```


Why Dynamic Typing?

- Latest-possible detection of type errors.
  - if true then 1 else "1" is ill-typed in SML
  - if true then Int 1 else Symbol "1" : SV
  - defer decision on whether it's "really" an Int until you use it
- Flexible choice of data values.
  - e.g., can return an integer or a boolean

Why Static Typing?

- Earliest-possible detection of type errors!
  - if true then 1 else "1" is ill-typed
- Dynamic typing is a special case!
  - "dynamically-typed" really means "typed by a recursive datatype"
  - tags aren't really types, they're value constructors!

Why Static Typing?

- You can't recover static typing in a dynamic framework!
  - Committed to tagging values and checking tags, even though one may be able to prove they are not needed.
  - That is, violates "pay-as-you-go" principle of language design.

Why Static Typing?

- Static type disciplines support enforced data abstraction.
  - e.g., cannot see the real representation of a queue
- Dynamic type systems rely on exposure of the representation.
  - e.g., a queue is "really" a pair of lists