Type-Based Certifying Compilation

Zhong Shao

Department of Computer Science
Yale University

PLDI Tutorial, June 2001
Additional Info: `http://flint.cs.yale.edu`

Outline of this Talk

I. Introduction and motivation

II. A hacker's guide to type theory

III. Certifying low-level features

IV. Certifying high-level features

V. Engineering type-based compilers

VI. Conclusions

Disclaimer: this talk is intended strictly as a tutorial . Because of the time limit, I can only sample a very small set of the work done in the area.

Part I. Introduction and Motivation

Problem and Motivation

How to develop a "good" common mobile-code infrastructure for the Next Generation Internet ?

Why ?

- The world is getting more and more net-centric
  (Sun "the network is the computer"; Microsoft .NET)
- Mobile code will define the platform
  (e.g., Java VM, Microsoft Common Language Runtime)

Research Challenges

What makes a "good" common mobile-code infrastructure?

- **Expressiveness & efficiency**
  - *Should be capable of*
    - … proving safety & security properties
    - … reasoning about low-level machine code & data layout
    - … supporting multiple programming languages
- **Reliability & security**
  - formal semantics with rigor & smaller TCB
- **Support for distributed computing**
  - "platform independence"
  - small and extensible runtime system
  - dynamic services (e.g., linking & loading, marshalling)

Mobile Code Threats & Attacks

Mobile code can:

- Overwrite memory
- Read private memory
- Create machine code and jump to it
- Execute illegal instructions
- Obtain control
- Infinite-loop (in a system without interrupts)
- ……

Abuse of API calls:

- Read files on disk
- Write files on disk
- Obtain critical locks
- Denial of service
- Perform operations without holding required lock
- ……

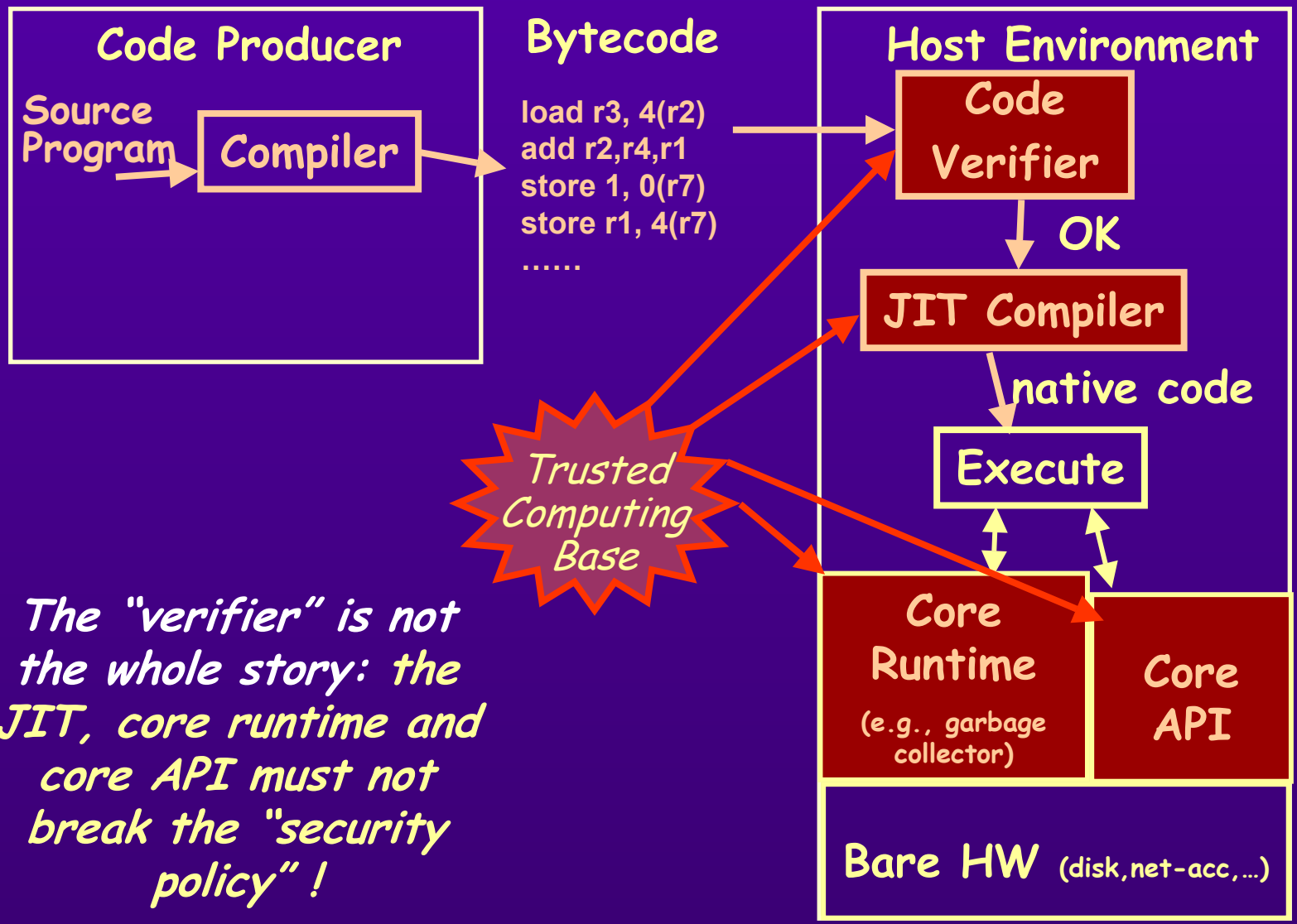
Examples of Advanced Security Policies

- Execute no more than N instructions between API calls
- Hold at most N locks at once
- Read/write only from readable/writable locations
- Sequence API operations according to a security automaton
- Any packet sent must be a copy of some packet received
- Any message sent must be logged to a file
- Domain specific ones: don't withdraw more than $100 from my bank account

Moral: need something more powerful than JVM, SFI, sandboxing, …

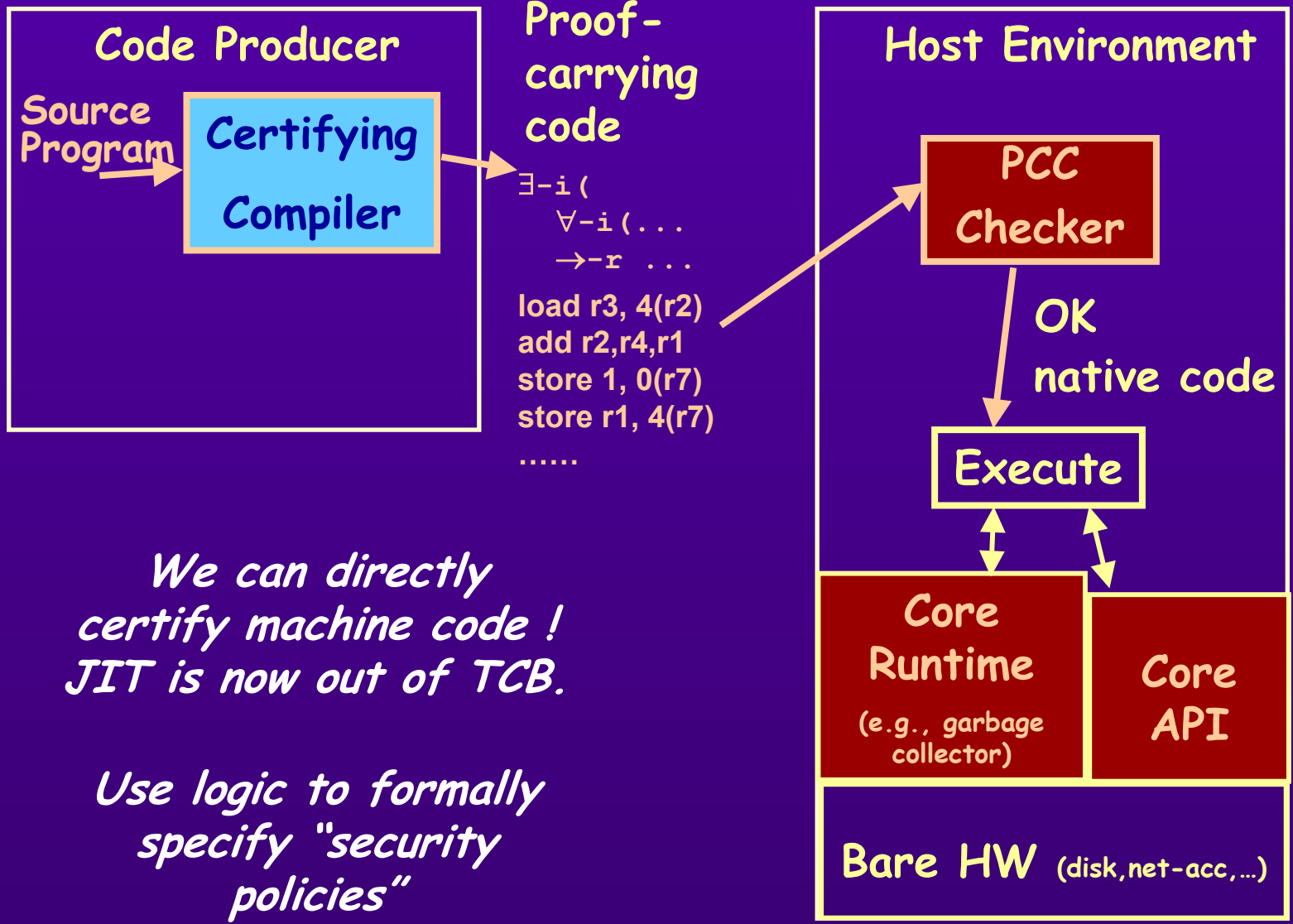
The traditional notion of "type safety" is not enough !

Mobile Code System: A Closer Look



The "verifier" is not the whole story: the JIT, core runtime and core API must not break the "security policy" !

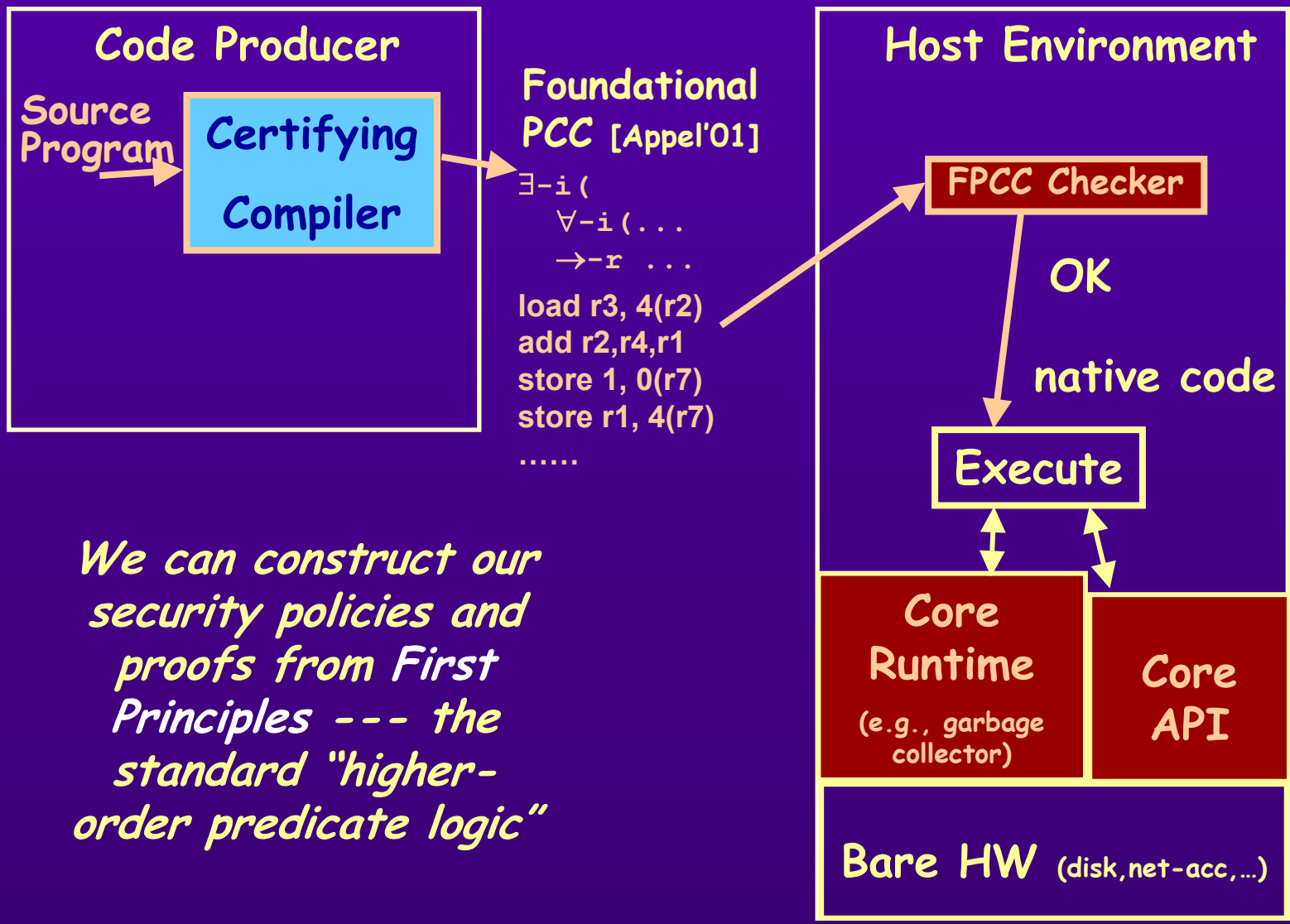
The PCC Approach [Necula&Lee 1996]



We can directly certify machine code ! JIT is now out of TCB.

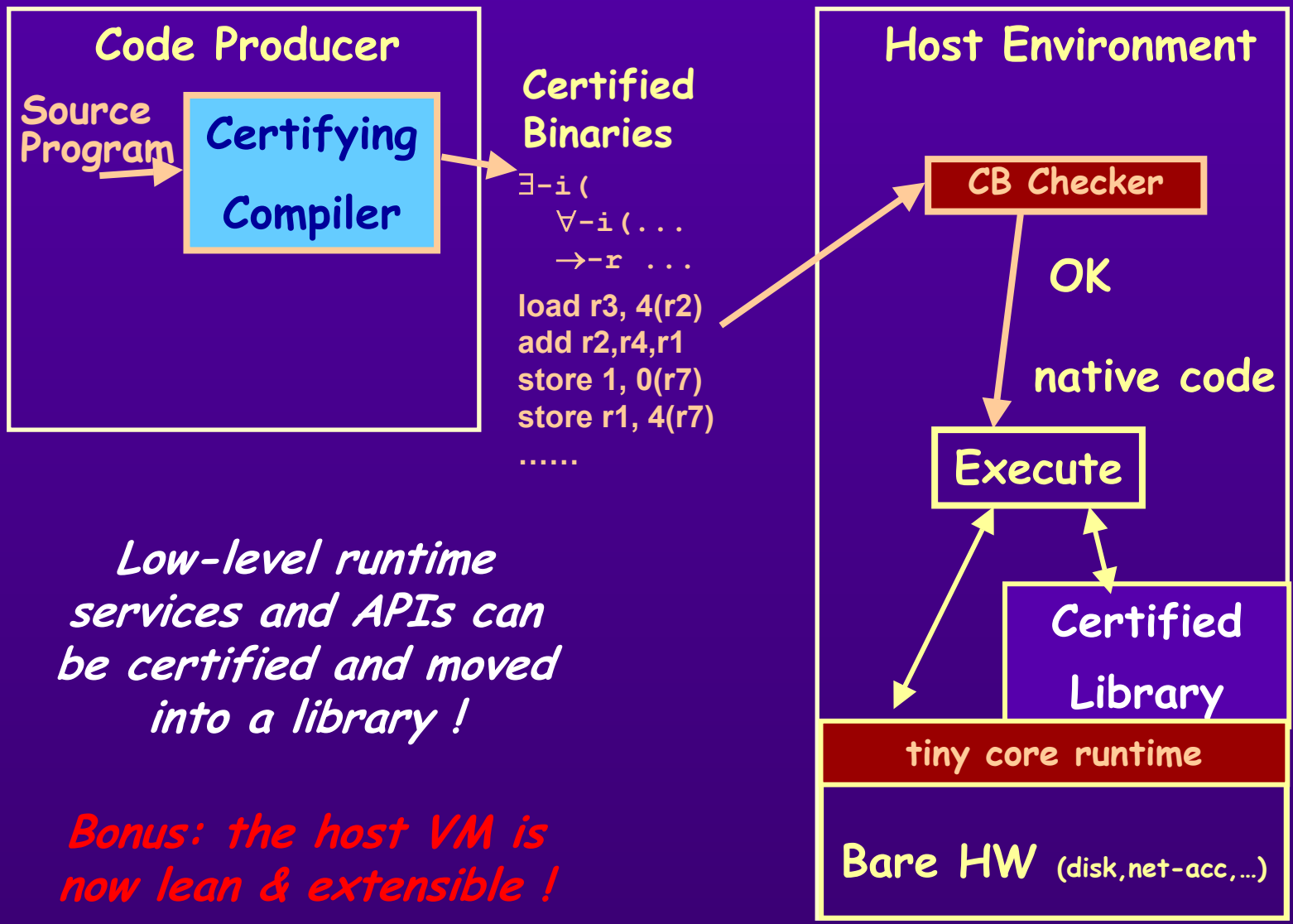
Use logic to formally specify "security policies"

The FLINT/FPCC Approach [AFS 1999]



We can construct our security policies and proofs from First Principles --- the standard "higher-order predicate logic"

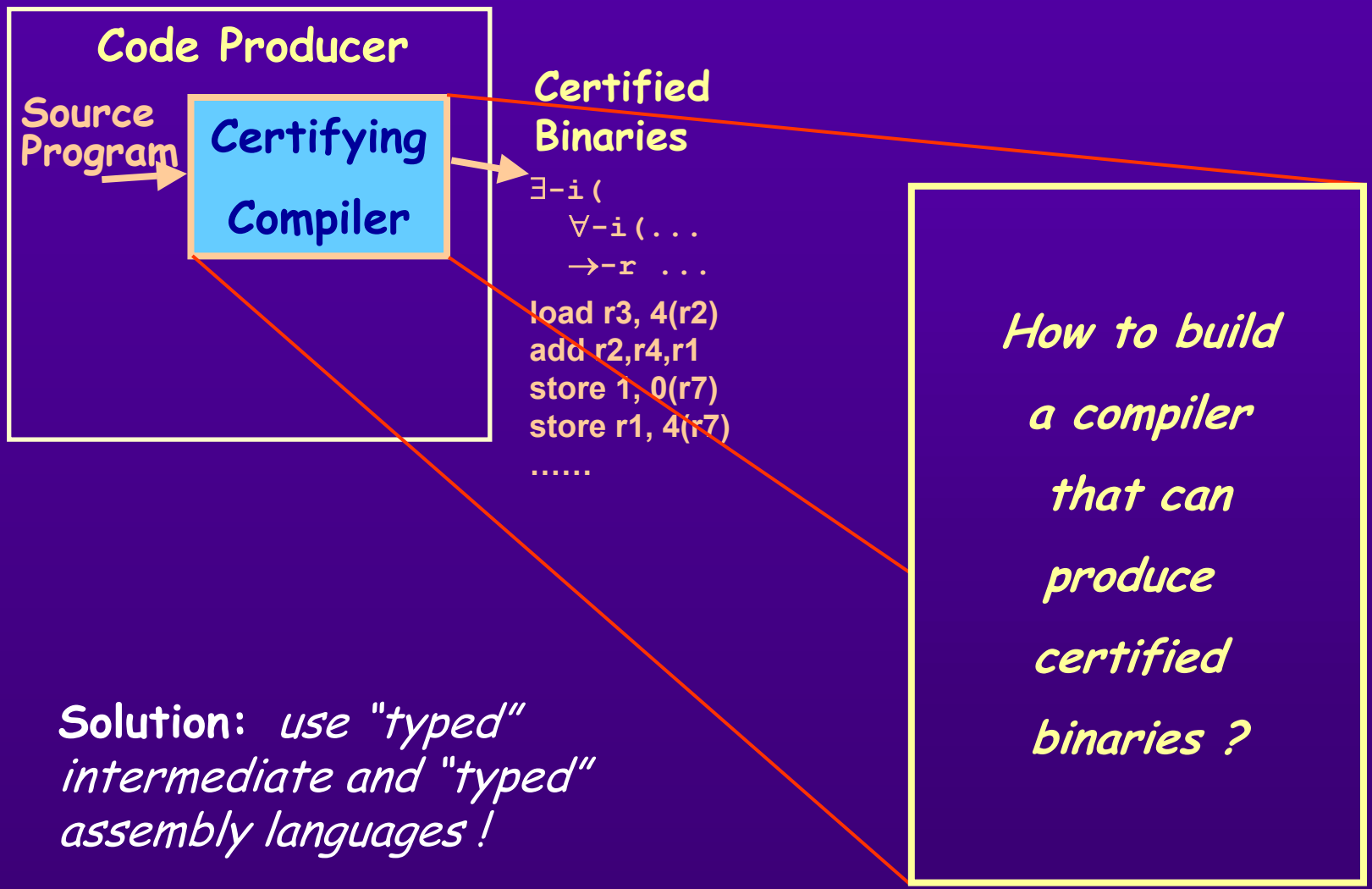
Evolving the FLINT/FPCC Approach



Low-level runtime services and APIs can be certified and moved into a library !

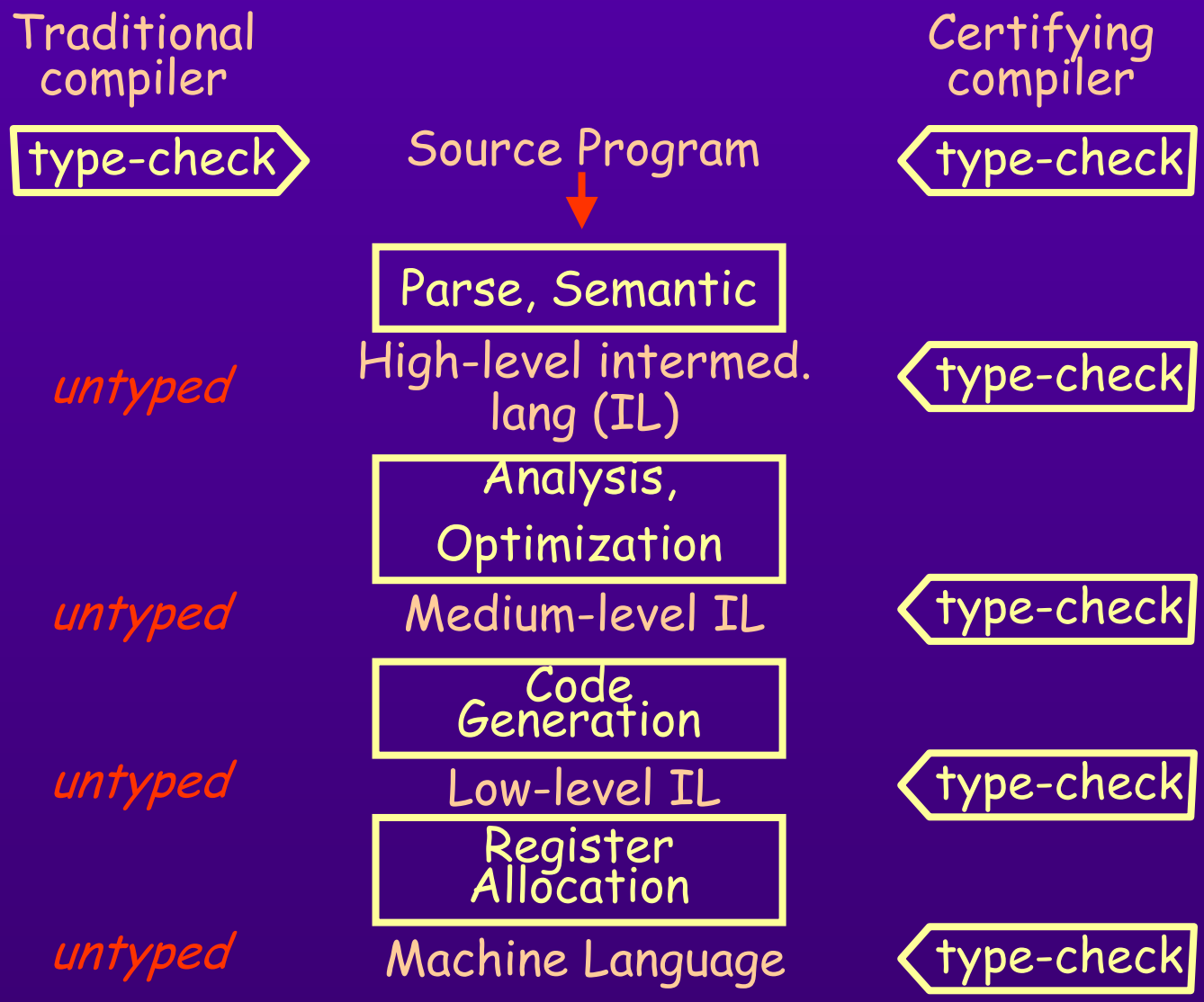
Bonus: the host VM is now lean & extensible !

Certifying Compiler is the Key !



Solution: *use "typed" intermediate and "typed" assembly languages !*

Phases of a Compiler



Why Typed Compilation ?

- Types scale well in expressiveness
  - as simple as representation specifications
  - as fancy as propositions in logic (any prog. invariants)
  - incredible amount of work on "type theory" already done
- Typed ILs can serve as secure interchange format
- Types are modular & user-friendly
  - IDL for common component libraries
- Types are useful for other purposes
  - type-based program analysis (see Palsberg's talk at PASTE'01)
  - new optimizations / debugging

Main Challenges in Typed Compilation

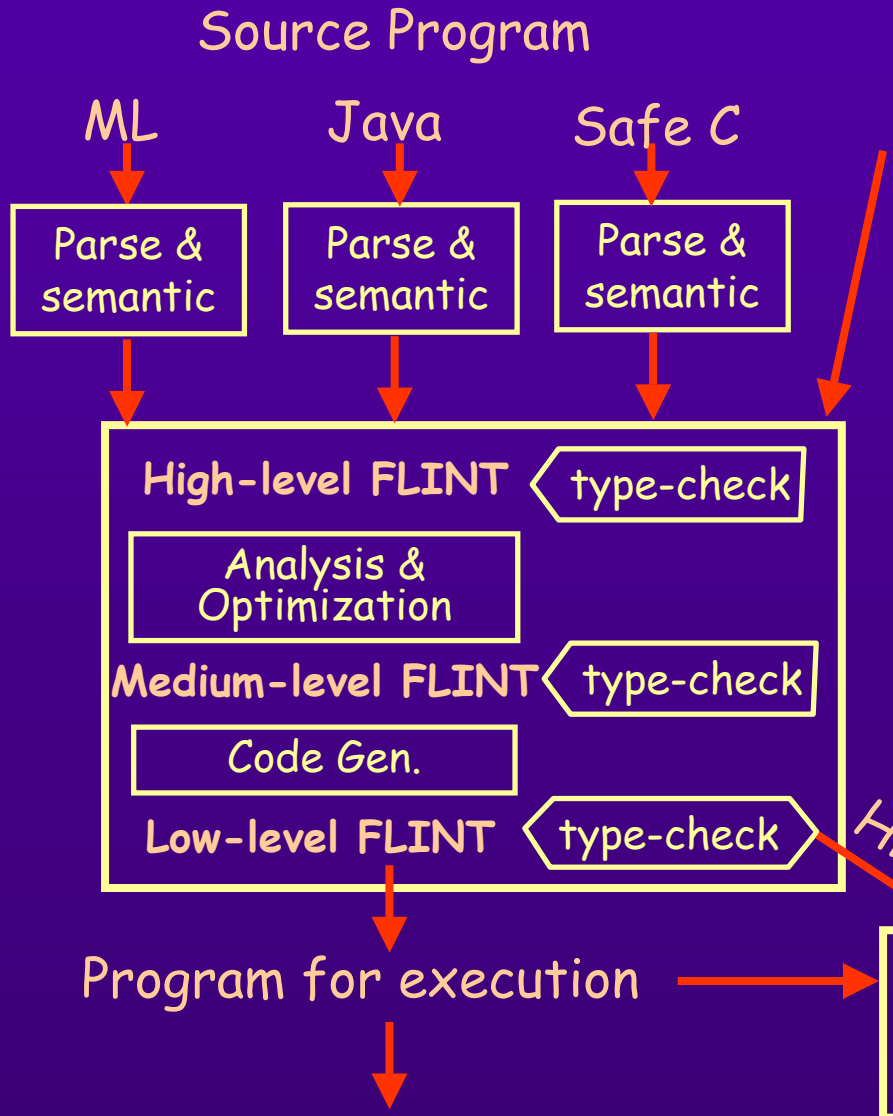
- How to design simple, general, yet expressive type systems ?

- How to support high-level features in existing programming languages (Java, ML) ?

- How to reason about low-level features (as in C or machine lang.) ? How to reason about arbitrary program invariants ?

- How to engineer type-based compilers --- do they scale ? are they practical ?

The FLINT Certifying Infrastructure



Add new front-ends to build new certifying compilers !

Key ideas:

- single typed IL capable of supporting very different source languages
- type info. allows efficient data reps. and safe language interoperation
- lang. independent type sys. facilitates lang. independent safety policy

The FLINT/FPCC Approach: A Summary

- **Efficiency**
  - **Go for low-level machine code & raw data layout**
- **Expressiveness**
  - Certify code w. propositions & proofs from higher order logic
  - Represent props & proofs in a typed λ-calculus
    (via Curry-Howard isomorphism)
  - Use common typed ILs to compile multiple prog. languages
- **Reliability & security**
  - Simple core language; JIT no longer in TCB
- **Support for distributed computing**
  - Safe dynamic services via runtime type analysis
    … moved to library for smaller & more extensible VM

Part I Recap

- Mobile code will define the platform (e.g., JVM, .NET)
- Enforcing "advanced" security policies requires building "new infrastructure":
  - Certifying compiler for mainstream languages (Java, C)
  - Efficient checker & prover
  - Tools & libraries that facilitate certified programming

  But the payoff is huge !
  - Highly extensible & secure common runtime (w. tiny kernel)
  - Language- & platform-indep. certified code & proof libraries
- Making it practical is challenging but very promising
  - Don't have to deploy the whole thing in one day
  - Technologies for compiler & theorem-proving are quite mature
  - Most security policies are simple (not for "full correctness")

Part II. A Compiler-Hacker's Guide to Type Theory

Theory or Practice ?

Type Theorist

- Types are sets, domains, invariants, or PERs
- Typed languages: lambda calculi, PCF, Fomega, ML
- Static semantics / typing rules/ decidable typechecking
- Dynamic semantics / interpreter
- Soundness / safety theorems
- LICS / POPL / ICFP / …

Compiler Hacker

- Types are representation specifications
- Intermediate languages, SSA, machine instructions
- Some typing rules / use "cast" if things break
- Efficiency, efficiency, efficiency …
- Who cares ? We'll debug !
- PLDI / CC / PACT / …

We Need Both !

- Rigorous formal semantics is mandatory if we want "real" safety guarantee .
- Certifying compilers should use the same efficient data representations as uncertified ones.
- Bridging this gap is not easy! Most of us were trained to do only one of the two …
- More and more people are now doing both:
  - CMU TILT, Cornell TAL, Yale FLINT, Princeton FPCC, Berkeley/CMU PCC, Microsoft Research, … …

The rest of this talk: to show you that the gap is not as bad as you may think

"Untyped" Intermediate Languages

- A fragment of a typical medium-level IR:

```
Program ::= LETREC Fundec_1, …, Fundec_n IN Exp.
Fundec  ::= FUN f(x_1 , …, x_n ) = Exp;
Exp     ::= RETURN (Val)
          | x = CALL (Val, Val_1, …, Val_n) ; Exp
          | x = PrimOp(Val_1, …, Val_n); Exp
          | IF CmpOp (Val_1, …, Val_n) THEN Exp_1 ELSE Exp_2
          | SWITCH Val OF (Exp_1, …, Exp_n)
Val     ::= f | x | n
PrimOp  ::= i32add | i32sub | … | r64add | … | alloc | upd | cast …
CmpOp   ::= ieq | ineq | ilt | igt | …
```

The Term Lang.: control structures + a bunch of primop-applications
(optional) use function call to simulate loop

"Typed" Intermediate Languages

- Making it "typecheck" is pretty easy:

Program ::= LETREC $Fundec_1$, …, $Fundec_n$ IN Exp.
Fundec ::= FUN $f(x_1 : Ty_1, \ldots, x_n : Ty_n) : Ty$ = Exp;
Exp ::= RETURN (Val)
| x = CALL (Val, Val_1, …, Val_n) ; Exp
| x = PrimOp(Val_1, …, Val_n); Exp
| IF CmpOp (Val_1, …, Val_n) THEN Exp_1 ELSE Exp_2
| SWITCH Val OF (Exp_1, …, Exp_n)
Val ::= f | x | n
PrimOp ::= i32add | i32sub | … | r64add | … | alloc | upd | cast …
CmpOp ::= ieq | ineq | ilt | igt | …

Ty ::= Int_{32} | $Real_{64}$ | … | $Code(Ty_1,\ldots,Ty_n) : Ty$

The Type Lang.: a set of representation specifications
To type-check: all primops must be given a type (what about "cast" ?)

"Typed" Intermediate Languages (cont'd)

- Some use Continuation-Passing Style (a variant of SSA):

Program ::= LETREC $Fundec_1$, …, $Fundec_n$ IN Exp.
Fundec ::= FUN $f(x_1 : Ty_1, \ldots, x_n : Ty_n)$ ~~: Ty~~ = Exp;
Exp ::= ~~RETURN (Val)~~
~~| x =~~ CALL (Val, Val_1, …, Val_n) ~~; Exp~~
| x = PrimOp(Val_1, …, Val_n); Exp
| IF CmpOp (Val_1, …, Val_n) THEN Exp_1 ELSE Exp_2
| SWITCH Val OF (Exp_1, …, Exp_n)
Val ::= f | x | n
PrimOp ::= i32add | i32sub | … | r64add | … | alloc | upd | cast …
CmpOp ::= ieq | ineq | ilt | igt | …

Ty ::= Int_{32} | $Real_{64}$ | … | $Code(Ty_1,\ldots,Ty_n)$ ~~: Ty~~

Type-checking can be done in the same way.

Several Observations

- "Types" and "terms" live in different worlds
  - Types are strictly compile-time entities
  - Types are not dependent on terms in any way
  - After type-checking is done, types can be erased
  - Types don't affect the runtime behavior
- Must reflect "types" as terms ("tags") if we want to use them at runtime.
- Often perform CPS- & closure-conversion if compiling functional languages (e.g., ML)
  - Must preserve "typing" during transformations; see [MMH'96, MWCG'98] for details

"Type theory" is to assign a precise type to various "cast" primops.

Types as Representation Specifications

- Primitive types
  - Int_{32}, $Word_{32}$, $Real_{32}$, $Real_{64}$, …
  - Subrange type (enumerate) Rng(1,10) --- all numbers from 1 to 10
  - Primops : basic arithmetics, comparisions, …
- Records / tuples / arrays / vectors
  - Mutability
  - Initialized?
  - Regular or packed
  - Primops: alloc, initialize, update, select
- Code pointers / functions
  - Multiple arguments (passed in registers or on stack)
  - Multiple return results ?
- Objects (and classes) --- see Part IV

The Type Language: A Closer Look

- To be more rigorous, we have to define the set of "valid" type expressions.
- For simplicity, I am listing a small set here only:
  (covering more of the previous rep. specs is easy)

T ::=	Int	32-bit integer
|	Real	64-bit real
|	Rng(N_1, N_2)	int btw N_1 & N_2
|	Code(T_1, …, T_n)	codeptr (w. CPS)
|	Tup(T_1, …, T_n)	n-element tuple

But what is "N_1" and "N_2" ?

The Type Language (cont'd)

- Types shoud not be dependent upon terms, so "N" here is different from those term-level constants

(type) T ::= Int
| Real
| Rng(N_1, N_2)
| Code(T_1, …, T_n)
| Tup(T_1, …, T_n)

(num) N ::= 0 | 1 | 2 | …

The Type Language (cont'd)

- Let's also formalize the type sequence $(T_1, \ldots, T_n)$:

(type) T ::= Int
| Real
| Rng(N_1, N_2)
| Code(R)
| Tup(R)

(num) N ::= 0 | 1 | 2 | …

(row) R ::= ∅ — empty sequence
| T ; R — add one more

The Type Language (cont'd)

- Having too many syntactic categories is a pain, let's merge them into a single one --- "type constructors":

  (tycon) T ::= Int | Real | Rng(T, T') | Code(T) | Tup(T)
  | 0 | 1 | 2 | … | ∅ | T ; T'

- Each tycon now must be "kinded" --- to prevent bad forms

  (kind) K ::= Tp | Num | Row | K -> K'

Examples:

Int	:	Tp	∅	:	Row
Real	:	Tp	;	:	Tp -> Row -> Row
Rng	:	Num -> Num -> Tp			
Code	:	Row -> Tp	0	:	Num
Tup	:	Row -> Tp	1	:	Num

The Type Language (cont'd)

- Let's make it more extensible:

(kind)	K	::= C | K -> K'
(primknd)	C	::= Tp | Num | Row
(tycon)	T	::= P | T(T')
(primtyc)	P	::= Int | Real | Rng | Code | Tup
		| 0 | 1 | ... | ∅ | ;

primtyc signature:

Int	:	Tp	∅	:	Row
Real	:	Tp	;	:	Tp -> Row -> Row
Rng	:	Num -> Num -> Tp			
Code	:	Row -> Tp	0	:	Num
Tup	:	Row -> Tp	1	:	Num

The Type Language: A Summary

- A type constructor "T" is well-formed (or valid) if it can be shown to have kind "K".
- The usual notion of "types" are tycons with kind "Tp".
- Extending our type language is easy:
  - Add your primitive kinds (the green stuff) if necessary
  - Add your primitive tycons (the red stuff), each with a properly assigned kind
- For every new "type" (of kind Tp) , we add a few term-level primops (which manipulate values of the new type).

Singleton Integer Type

- New primtyc:

  Sint : Num -> Tp

- Typing rules for the term-level integer constant:

  1 : Sint(1) 2 : Sint(2) ... n : Sint(n)

- If a value "v" has type Sint(n), it must be equal to "n".

- Term-level cast: no-op from Sint(n) to Int ; the reverse requires dynamic check.

- Sint can certify array-bounds checking or "tagged record"

  Tup(Sint(1); Int; Real)

1	40	3.14

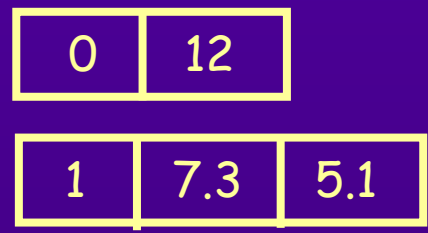
Sum Type (a.k.a. Tagged Union)

- New primtyc:

  Sum : Row -> Tp

- Example (ML) datatype t = A of int
  | B of real * real

Sum(Tup(Sint(0); Int) ;
Tup(Sint(1); Real; Real))



- Term-level primops:

  The "inj" primop casts tagged record into a sum type

  Must check the tag at runtime to cast back to tagged record.

- The C pointer type to T can be coded as Sum(Sint(0), Ref(T))

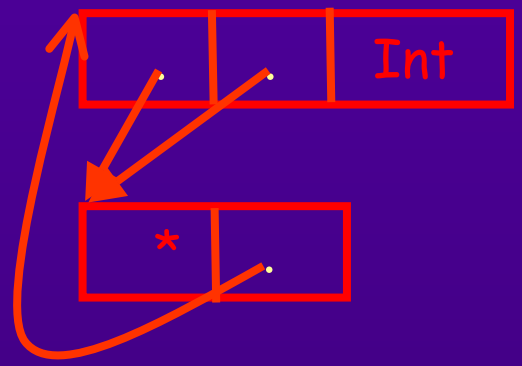
  (assuming "0" for NULL ptr, Ref for non-null ptr)

Recursive Type

- How to represent recursive types ?

```
struct tree { tree  *left;
              tree  *right;
              int    val;
            };
```

Most compilers use a "cyclic" data structure:



- But it is difficult to reason about cycles in our "type language".

Solution: use function to represent it !

Recursive Type (cont'd)

- New primtyc:

$$\mu \quad : \quad (Tp \rightarrow Tp) \rightarrow Tp$$

- Type language now has to support type variable & function:

$$\begin{array}{rl} \text{(kind)} & K ::= C \mid K \rightarrow K' \\ \text{(primknd)} & C ::= Tp \mid Num \mid Row \\ \text{(tycon)} & T ::= P \mid T(T') \mid t \mid \lambda t : K.\ T \\ \text{(primtyc)} & P ::= \ldots \mid \mu \end{array}$$

- An example "type function":

$$F = \lambda t : Tp\ .\ Sum\ (\ Sint(0)\ ;\ Tup(t, t, int)\ )$$

The tree type is now just $\mu(F)$

Recursive Type (cont'd)

- Need two new primops to manipulate recursive values:

  The "fold" primop casts a value of type **F(μ(F))** into **μ(F)**

  The "unfold" primop casts a value of type **μ(F)** into **F(μ(F))**

  For example, let

  Tree = μ (λt : Tp . Sum (Sint(0) ; Tup(t, t, int)))

  Unfolding a value of type Tree results in a value of type

  Sum (Sint(0) ; Tup (Tree, Tree, int))

  which we know how to manipulate.

Polymorphic Type

- Functions can take variable-typed arguments
- Polymorphic type is a safe alternative for " **void *** "

- We revise the kind of Code into:

$$\text{Code} \;:\; (\text{Tp} \rightarrow \text{Row}) \rightarrow \text{Tp}$$

- The term language has to be extended as well:

Fundec	::=	FUN f [t : Tp] ($x_1 : T_1, \ldots, x_n : T_n$) = Exp;
Exp	::=	.....
	|	CALL (Val, T, $Val_1, \ldots, Val_n$)

- Replace "Tp" above with "K", we get polymorphism over arbitrary kinds.

Polymorphic Type (cont'd)

- A polymorphic identify function (in CPS):

```
FUN f [t : Tp] (x : t, c : Code[](t))
  = CALL (c, [], x)
```

- Even monomorphc functions are polymorphic over "stack" and "callee-save registers" (at low level)

```
FUN f [ t : Tp ] (r1 : t, r3 : Code[](r1 : t)) = …
```

- Types still don't have any effect on "terms" --- all "red" stuff can be erased …

Existential Type

- Motivation: how to handle values of different types in a uniform way ?
  - Abstract datatypes

```
signature IntSet =
  sig   type t
        val isElem  :  int -> t -> bool
        val add     : int -> t -> t
  end
```

  - Closures: all ML functions of type "int -> int" have form "env * (env * int -> int)" where "env" is abstract
  - Type dynamic

Existential Type (cont'd)

- New primtyc:

  $\exists$: (Tp -> Tp) -> Tp

  Or if over arbitrary kinds:

  $\exists$: (K -> Tp) -> Tp

- Intset is $\exists$(F) where

  F = λt : Tp . Tup(Int->t->Bool; Int->t->t)

- Closure for "int->int" has type $\exists$(F) where

  F = λt : Tp . Tup(t; Tup(t; Int)->Int)

- Type-dynamic has type $\exists$(F) where

  F = λt : Tp . t

Existential Type (cont'd)

- Two new forms of "cast" for handling existentials:

```
Exp     ::=  ...
           |  x = PACK (t=T, Val : T') ;   Exp
           |  (t, x) = OPEN (Val) ;   Exp
```

PACK casts a value of type F(T) into ∃(F)

OPEN casts a value of type ∃(F) into F(t)

The old information about "t" is lost. You may get it back if you maintain runtime "type tag".

The Type Lang.: How Far Can We Go?

- Our current type lang. is itself a "simply typed lambda caluclus"

(kind)	K	::= C | K -> K'
(primknd)	C	::= Tp | Num | Row
(tycon)	T	::= P | t | λt:K.T | T(T')
(primtyc)	P	::= Int | Real | Rng | Code | Tup
		| 0 | 1 | ... | ∅ | ;
		| Sint | Sum | ∃ | μ

- It can be extended as much as you like as long as we make sure all computation in it is terminating (for decidable typechecking)

- The newest version of FLINT uses "Calculus of Inductive Construction" (can express all propositions in higher-order logic)

Part II Recap

- Types are terms live in different world. Types should never be dependent upon terms.
- Type language is usually some kind of "lambda calculus" --- but types are compile-time entities, so we don't need compile them.
- Typechecking involes testing if two types are equal (this may require symbolic reduction of type expressions)
- All reductions done inside the type language must terminate --- otherwise typechecking won't be decidable.
- The term language remains pretty much the same --- other than a few new safe "cast" primops.

Part III. Certifying Low-Level Features

Sample Low-Level Features

- Array-bounds checking
- Memory management
  - Malloc/free
  - Garbage collection / stack allocation
  - Pointer data structures
- Dynamic type dispatch
  - Tagging and heap-tracing during GC
  - Pickling, persistence, type dynamic
- Advanced program invariants
  - General resource management

Array-Bounds Checking [XP 1999]

- New primtyc (for integer array):

  Array : Num -> Tp

- A integer array of length "10" has type

  Array(10)

- Array creation is polymorphic over "length":

  mkArray : ∀n : Num. (Sint(n) -> Array(n))

- Need to reflect all integer arith and cmp operations into the type language. Also need dependent kinds !

  sub : ∀n : Num. ∀i : [0, n-1] . (Array(n) -> Sint(i) -> Int)

- Array update can be handled in the same way.

Memory Management

- This area is still wide open ……
- Most existing schemes (regions, capabilities) are based on "linearity principle" :
  - It is safe to free an object if we're holding the only pointer
  - But this is too strong for most heap data
- In practice, people also do reference counting and allow harmless "dangling pointers"
- GC and malloc-free don't interact well --- the connection is a black art, badly in need of formalism

Regions and Capabilities [TT'94 CWM'99]

- Region and capabilities are new primknd:

(kind) K ::= C | K -> K'
(primknd) C ::= Tp | Num | Row | Reg | Cap

- Every heap data lives in a region:

Tup : Reg -> Row -> Tp

- Use capabilities to track which region is accesible and which is safe to be freed
- Every function has a capability precondition
- Can build type-safe copying GC above the region calculus

Runtime Tag Dispatch

Use special tags to represent runtime type information

```
datatype rttag  =  RT_INT
                |  RT_PAIR of rttag * rttag
                |  ……

fun sub (x : rttag) =
   case x
     of RT_INT =>  intsub
      |  RT_PAIR (t1, t2)  =>
                    (fn ((x,y), i) => (sub t1 (x, i), sub t2 (y, i)))
      | _  =>  boxedsub
```

Problem: what is the type of "sub" ?

Intensional Type Analysis [HM 1995]

- Extension to the type language:
  - Need to perform case analysis on all types of kind Tp
  - Require recursive transformation (only ok if it is primitive recursive)
  - Tp should be defined as an inductive definition [TSS'00]

  Example: mutator view ➔ collector view (see our type-safe GC paper in this PLDI)

- Extension to the term language:
  - Need to reflect types as "runtime tags" [CWM'98]
  - Programmer can't forge wrong "tags"

- Applications: data representation selection; ad-hoc polymorphic services

Propositions as Types

a.k.a. "Curry-Howard" isomorphism

- Extend the type lang. w. full-blown dependent kinds;
  (the term language remains separate --- no dependent type)

- A proof of an implication "A=>B" is a function from proof of A to proof of B.
- A proof of conjuction "A and B" is a pair of (p, q) where p is proof of A and q is proof of B.
- A proof of disjuction "A or B" is a tagged union of "proof of A" and "proof of B"

- See [Barendregt & Geuvers 2000] for good intro.

Part IV. Certifying High-Level Features

Sample High-Level Features

Languages	Challenges	Approaches
ML	▪ module system (functor) ▪ closures & polymorphism ▪ recursive data type	prim/record/code ∃∀μ ref exn
Java & JVML	▪ classes; interfaces; objects ▪ access control & privacy ▪ name-based subtyping ▪ dynamic linking & loading ▪ reflections; concurrency	prim/record/code ∃∀μ row-kind ???
"Safe" C	▪ explicit memory management ▪ efficient array access	region & alias types singleton types

Why Compiling Them Away ?

- To have a simpler type system for our ILs.
- To show these high-level features are just "derived constructs"
- To get better interoperability.
- To have a language-independent IDL or runtime system.
- Examples: ML modules and Java classes are very complex constructs --- making them live together is just impossible (see ML2000)

Case Study: Java Classes

Compare with other class-based lang., Java has:

- Inheritance from a single superclass
  - Multiple interfaces are possible
  - No instance fields in interfaces
- "Pseudo" binary methods
  - Method parameters, results, and instance field of "twin" type have access to private fields
- Name-based subtyping hierarchy

Typed Compilation of Java [LST 1999]

- Efficient Java object encoding:
  - casts to superclass for free
  - low-cost invokevirtual and invokeinterface

  → self-application semantics of method invocation: recursive types

  → row polymorphism instead of subtyping

- Name-based class hierarchy
  - use existential types

Fω type system features used:

- polymorphism for code reuse (inheritance)
- existential types for privacy

Sample Java Program

private mutable fields
public methods

dynamic binding

interface methods

inheritance &
interface implementation

overriding
super calls

```
class Pt {
   private int x = 0;
   public void move (int dx)
      { this.x = this.x + dx;  }
   public void bump ()
      { this.move(1); } }

interface Zoomable {
   public void zoom (int s); }

class SPt extends Pt
implements Zoomable {
  private int scale = 1;
  public void move (int dx)
     {super.move (this.scale * dx); }
  public void zoom (int s)
    { this.scale = this.scale * s; } }
```

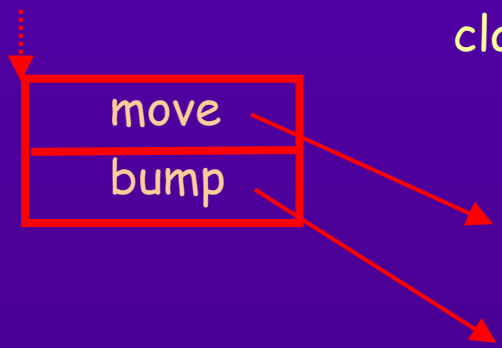
Sample Object Layout

```
class Pt {
    private int x = 0;
    public void move (int dx)
        { this.x = this.x + dx;  }
    public void bump ()
        { this.move(1); } }

interface Zoomable {
    public void zoom (int s);}

class SPt extends Pt
implements Zoomable {
   private int scale = 1;
   public void move (int dx)
       {super.move (this.scale * dx); }
   public void zoom (int s)
     { this.scale = this.scale * s; } }
```

Sample Object Layout: *Vtable*

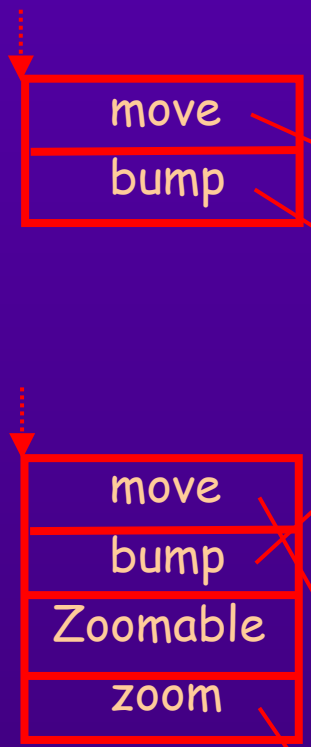



```
class Pt {
    private int x = 0;
    public void move (int dx)
        { this.x = this.x + dx;  }
    public void bump ()
        { this.move(1); } }

interface Zoomable {
    public void zoom (int s);}

class SPt extends Pt
implements Zoomable {
   private int scale = 1;
   public void move (int dx)
       {super.move (this.scale * dx); }
   public void zoom (int s)
     { this.scale = this.scale * s; } }
```

Sample Object Layout: *Vtable*

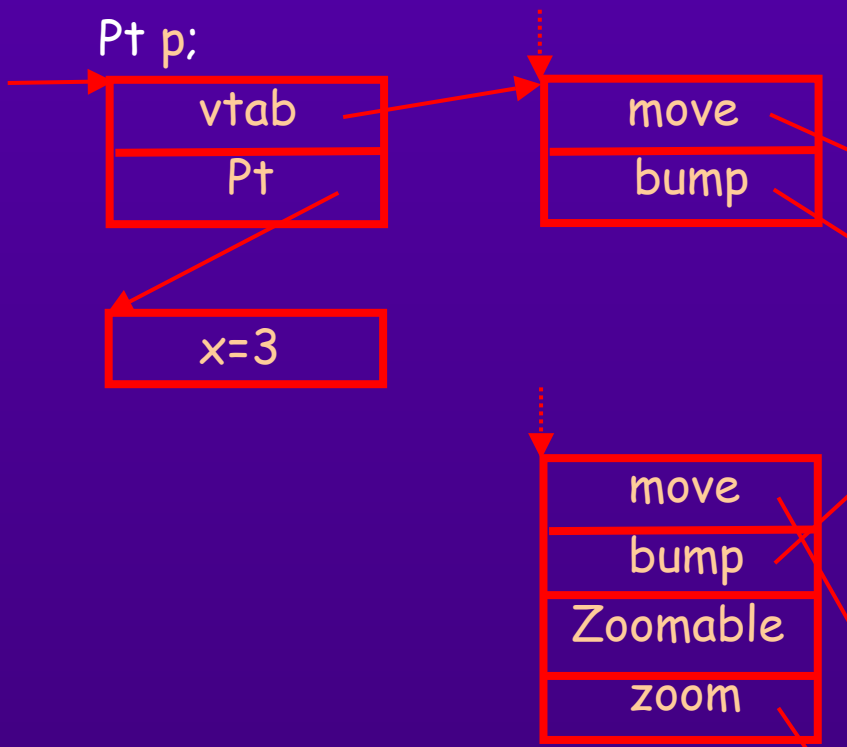



```
class Pt {
   private int x = 0;
   public void move (int dx)
      { this.x = this.x + dx;  }
   public void bump ()
      { this.move(1); } }

interface Zoomable {
   public void zoom (int s);}

class SPt extends Pt
implements Zoomable {
  private int scale = 1;
  public void move (int dx)
     {super.move (this.scale * dx); }
  public void zoom (int s)
    { this.scale = this.scale * s; } }
```

Sample Object Layout: *Pt p*

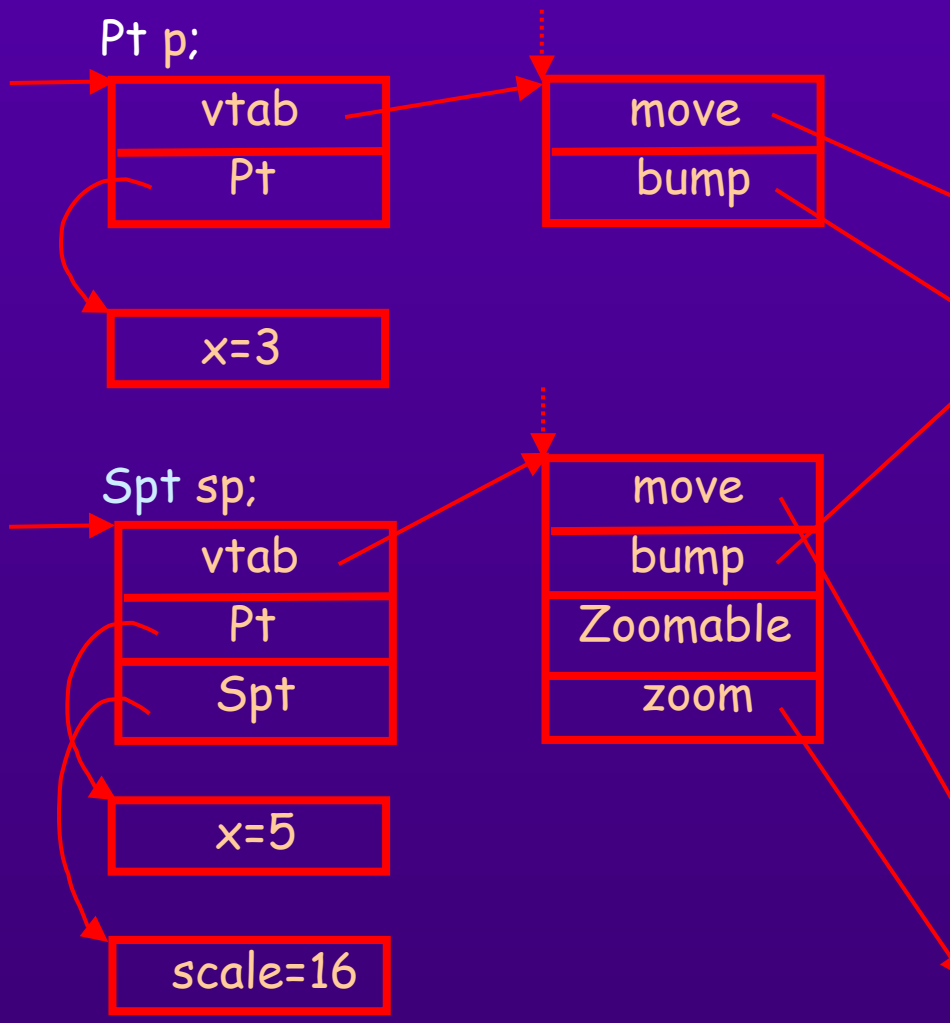



```
class Pt {
   private int x = 0;
   public void move (int dx)
      { this.x = this.x + dx;  }
   public void bump ()
      { this.move(1); } }

interface Zoomable {
   public void zoom (int s);}

class SPt extends Pt
implements Zoomable {
  private int scale = 1;
  public void move (int dx)
     {super.move (this.scale * dx); }
  public void zoom (int s)
    { this.scale = this.scale * s; } }
```

Sample Object Layout: *Spt sp*

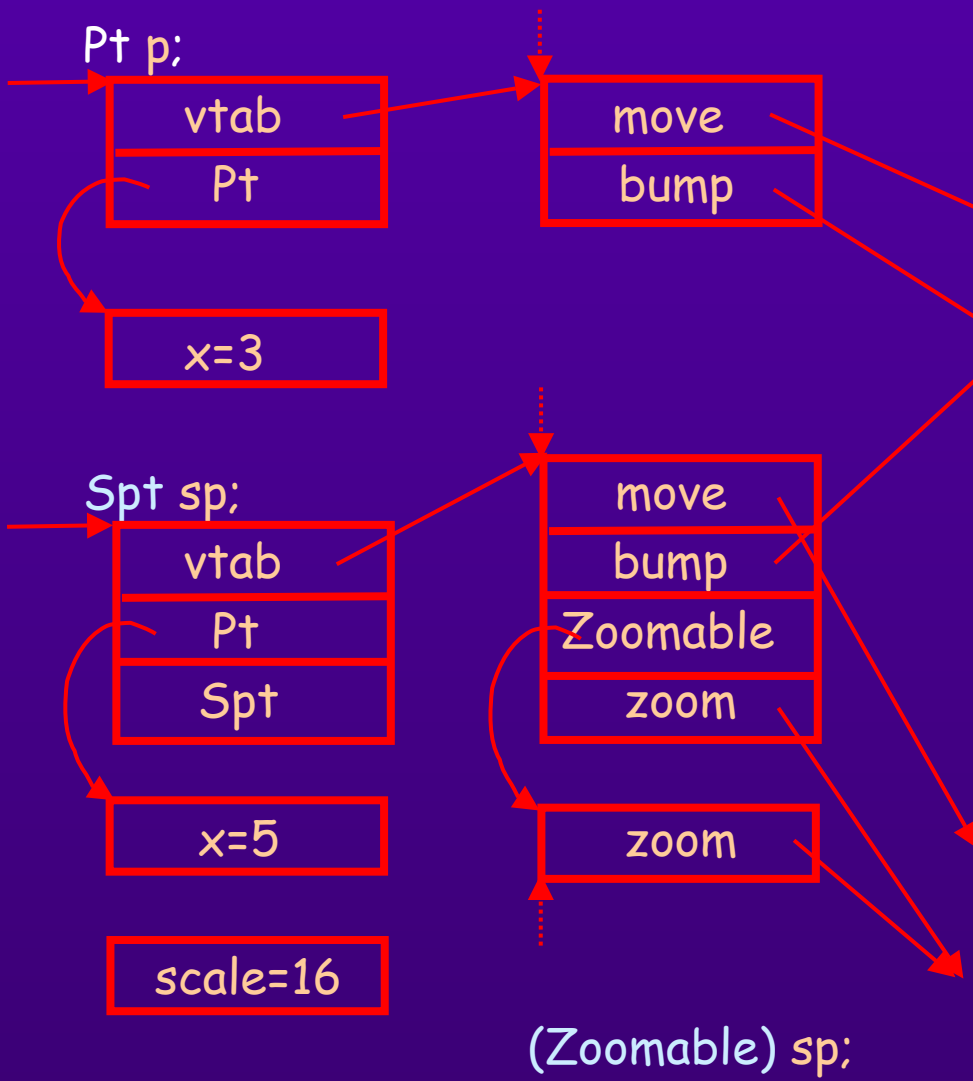



```
class Pt {
   private int x = 0;
   public void move (int dx)
      { this.x = this.x + dx;  }
   public void bump ()
      { this.move(1); } }

interface Zoomable {
   public void zoom (int s);}

class SPt extends Pt
implements Zoomable {
  private int scale = 1;
  public void move (int dx)
     {super.move (this.scale * dx); }
  public void zoom (int s)
    { this.scale = this.scale * s; } }
```

Sample Object Layout: *Itable*

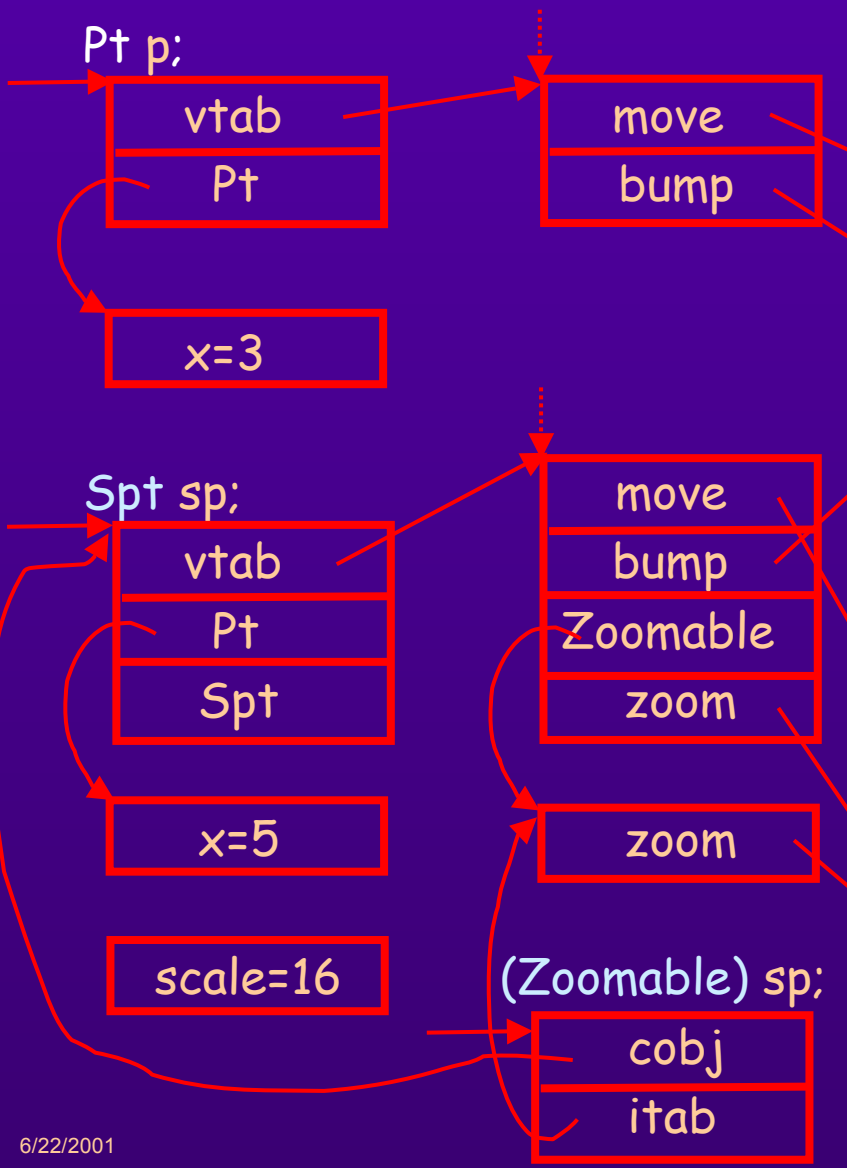



```
class Pt {
   private int x = 0;
   public void move (int dx)
      { this.x = this.x + dx;  }
   public void bump ()
      { this.move(1); } }

interface Zoomable {
   public void zoom (int s);}

class SPt extends Pt
implements Zoomable {
  private int scale = 1;
  public void move (int dx)
     {super.move (this.scale * dx); }
  public void zoom (int s)
    { this.scale = this.scale * s; } }
```

Sample Object Layout: *(Zoomable)* sp




```
class Pt {
   private int x = 0;
   public void move (int dx)
      { this.x = this.x + dx;  }
   public void bump ()
      { this.move(1); } }

interface Zoomable {
   public void zoom (int s);}

class SPt extends Pt
implements Zoomable {
  private int scale = 1;
  public void move (int dx)
     {super.move (this.scale * dx); }
  public void zoom (int s)
    { this.scale = this.scale * s; } }
```

The Mini-FLINT Calculus (1999)

Kinds $K ::= \Omega \mid K_1 \to K_2 \mid R^L$

Tp

Types $T ::= t \mid T_1 \to T_2 \mid \lambda t{:}K.T \mid T_1 T_2 \mid \forall t{:}K.T$
$\mid \varnothing^L \mid l{:}T;T' \mid \{T\}$
$\mid$ **Ref** $T \mid \mu t.T \mid \exists t{:}K.T \mid$ **Typ**(x)

Terms $e ::= x \mid \lambda x{:}T.e \mid x_1 x_2 \mid$ let $x = e_1$ in e_2
$\mid \Lambda t{:}K.e \mid x[T] \mid \{l_1 = e_1 \ldots l_n = e_n\} \mid e.l$
$\mid$ **ref** $e \mid !e \mid e_1 := e_2 \mid$ **fold**$(e,T) \mid$ **unfold** e
$\mid \langle t{:}K = T, e{:}T' \rangle \mid$ **Val**(x)

Extensible Records in Mini-FLINT

Type context $A ::= \varepsilon \mid A, x : T \mid A, t : K$

Type formation

$$\frac{A \vdash \mathbf{ok}}{A \vdash \varnothing^{L} : R^{L}} \qquad \frac{A \vdash T : \Omega \quad A \vdash T' : R^{L \cup \{l\}}}{A \vdash l{:}T;T' : R^{L}}$$

Term formation

$$\frac{A \vdash T : R^{\varnothing}}{A \vdash \{ T \} : \Omega}$$

$$\frac{A \vdash e_i : T_i \quad \text{where } i=1,\ldots,n}{A \vdash \{l_1=e_1 \ \ldots \ l_n=e_n\} : \{l_1:T_1 \ \ldots \ l_n:T_n;\ \varnothing^{\{l1,\ldots,ln\}}\}}$$

Using Extensible Records: An Example

Pt's vtable : { move : t -> int -> void ;
bump : t -> unit -> void ;
$\varnothing$\{move,bump\} : $R^{\{move,bump\}}$ }

SPt's vtable : { move : t -> int -> void ;
bump : t -> unit -> void ;
subVtab : $R^{\{move,bump\}}$ }

subVtab : $R^{\{move,bump\}}$
= Zoomable : ... {zoom : ...} ... ;
zoom : t -> int -> void ;
$\varnothing$\{move,bump,Zoomable,zoom\}

Recursive Types in Mini-FLINT

Type context $A ::= \varepsilon \mid A, x : T \mid A, t : K$

Type formation

$$\frac{A, t{:}\Omega \vdash T : \Omega}{A \vdash \mu t.T : \Omega}$$

Term formation

$$\frac{A \vdash e : T[\mu t.T/t]}{A \vdash \mathrm{fold}(e, \mu t.T) : \mu t.T}$$

$$\frac{A \vdash e : \mu t.T}{A \vdash \mathrm{unfold}\, e : T[\mu t.T/t]}$$

Using Recursive Types: An Example

$$\text{Object Pt} : \mu t.\{\text{vtab} : \{\text{move} : t \to \text{int} \to \text{void};\ \text{bump} : t \to \text{unit} \to \text{void};\ \varnothing^{\{\text{move,bump}\}} : R^{\{\text{move,bump}\}}\};\ \text{PtFld} : \{x : \text{Ref int};\ \varnothing^{\{x\}}\};\ \varnothing^{\{\text{vtab,PtFld}\}} : R^{\{\text{vtab,PtFld}\}}\}$$

"fold" and "unfold" operators are no-ops.

Must "unfold" before invoking a method.

Must "fold" when building an object.

Using Recursive Types: An Example

Object Pt : $\mu t.$ { vtab : { move : t -> int -> void ;
bump : t -> unit -> void ;
$\varnothing^{\{move,bump\}}$: $R^{\{move,bump\}}$ };
PtFld : { x : Ref int ; $\varnothing^{\{x\}}$ };
$\varnothing^{\{vtab,PtFld\}}$: $R^{\{vtab,PtFld\}}$ }

Object SPt : $\mu t.$ { vtab : { move : t -> int -> void ;
bump : t -> unit -> void ;
subVtab[t] : $R^{\{move,bump\}}$ };
PtFld : { x : Ref int ; $\varnothing^{\{x\}}$ };
subFlds : $R^{\{vtab,PtFld\}}$ };

subVtab : $\Omega \rightarrow R^{\{move,bump\}}$
= $\lambda t{:}\ \Omega.$ Zoomable : ... {zoom : ...} ... ;
zoom : t -> int -> void ;
$\varnothing^{\{move,bump,Zoomable,zoom\}}$

Existential Types in Mini-FLINT

Type context $A ::= \varepsilon \mid A, x : T \mid A, t : K$

Type formation

$$\frac{A, t{:}K \vdash T : \Omega}{A \vdash \exists t{:}K.T : \Omega} \qquad \frac{A \vdash x : \exists t{:}K.T}{A \vdash \mathbf{Typ}(x) : K}$$

Term formation (both are "no-ops")

$$\frac{A \vdash T' : K \quad A \vdash e : T[T'/t]}{A \vdash \langle t{:}K{=}T', e{:}T\rangle : \exists t{:}K.T}$$

$$\frac{A \vdash x : \exists t{:}K.T}{A \vdash \mathbf{Val}(x) : T[\mathbf{Typ}(x)/t]}$$

Using Existential Types: An Example

Object Pt : $\exists$ subFlds : $R^{\{vtab,PtFld\}}$.
$\exists$ subVtab : $\Omega \rightarrow R^{\{move,bump\}}$.
$\mu t.$ { vtab : { move : t -> int -> void ;
bump : t -> unit -> void ;
subVtab[t] : $R^{\{move,bump\}}$ }
PtFld : { x : Ref int ; $\varnothing^{\{x\}}$ };
subFlds : $R^{\{vtab,PtFld\}}$ };

Pt subVtab = $\lambda t{:}\ \Omega.$ $\varnothing^{\{move,bump\}}$

Spt subVtab = $\lambda t{:}\ \Omega.$ Zoomable : ... {zoom : ...} ... ;
zoom : t -> int -> void ;
$\varnothing^{\{move,bump,Zoomable,zoom\}}$

Encoding Java Objects in FLINT

```
Class List {
    private List next;
    public List tail()
    …
}
```

"Twin" object:

μ twin.
$\exists$ subFlds.
$\exists$ subVtab.
μ self.
{vtab: {tail: self-> twin; …;
subVtab self};
ListFld: {next: Ref twin};
subFlds}

Encoding Java Objects in FLINT

```
Class List {
    private List next;
    public List tail()
    …
}
```

Internal view:

μ twin.
$\exists$ subFlds.
$\exists$ subVtab.
μ self.
{vtab: {tail: self-> twin; …;
subVtab self};
ListFld: {next: Ref twin};
subFlds}

Encoding Java Objects in FLINT

```
Class List {
    private List next;
    public List tail()
    …
}
```

Class code: ListC = <private = …, …>

External view:

μ twin.
∃ subFlds.
∃ subVtab.
μ self.
{vtab: {tail: self-> twin; …;
subVtab self};
ListFld: Typ(ListC);
subFlds}

Encoding Java Classes and Methods

```
Class Pt { …
    public void bump() { this.move(1); }
    … }
```

Class code (untyped version)

```
Pt = <private = {x : Ref Int},
       let dict = ΛsubFlds . ΛsubVtab . let …
                  in  { bump = λself . λ_.
                                 self.vtab.move self 1
                                 …
                       }
           new = …
       in …
```

Encoding Java Classes and Methods

```
Class Pt { …
    public void bump() { this.move(1); }
    … }
```

```
Class code (typed version)
   Pt = <private = …
          let dict = ΛsubFlds . ΛsubVtab . let …
                     in  { bump = λself . λ_.
                                  let this = PackObj (self)
                                      o = Val(Val(unfold this)))
                                  in (unfold o).vtab.move self 1 } …
                in …
```

```
μt. { vtab  :  { move :  t -> int -> void ;
                 bump :  t -> unit -> void ;
                 subVtab t }
      PtFld :  { x : Ref int ; ∅{x} };
      subFlds};
```

```
μ twin. ∃ subFlds . ∃ subVtab . μ t . ……
```

Part IV Recap

- Complex high-level constructs can be certified using rather simple type system
- Going through a language-independent type system dramatically simplifies the certifying compiler
- Language interoperation through a common type system is possible.
- There are still open problems:
  - Fine-grain access control and security
  - Reflection, binary compatibility, dynamic linking, …

Part V. Engineering Type-Based Compilers

Implementation Issues

- How to represent and manipulate "types" efficiently ?
  - It could easily blow up your compilation time
  - Run time is also an issue if "types" are reflected as "tags".
- Each type-based compiler has its own bag of tricks:
  - FLINT
  - TILT
  - Popcorn
- Other certifying compilers have similar issues:
  - Touchstone
  - SpecialJ

Exponential Blowup

- Source of "sharing": module interfaces, polymorphic functions, …

- Example:

$$T_1 = \text{int} \rightarrow t$$
$$T_2 = T_1 \rightarrow T_1$$
$$\ldots\ldots$$
$$T_n = T_{n-1} \rightarrow T_{n-1}$$

without sharing: $O(2^n)$ with sharing: $O(n)$

Even with sharing, how to traverse the dag and perform substitution in linear time ?

Implementation Criteria [SLM 1998]

- Compact space usage
  - Good news: large types are highly redundant
- Linear-time traversal of types
  - Key: avoid traversing isomorphic subgraphs multiple times
- Fast type-equality operation
- Simple "user" interface
  - How easy to program with these "new" representations?
- Compact runtime "type tag" --- see work on optimal type lifting [SS 1998]

The FLINT Approach [SLM 1998]

- Use deBruijn indices
  - All bound variables are represented as integers
  - No need for alpha-renaming
- Use suspension-based representation [Nadathur 1994]
  - Lazy reduction (will do one only if really necessary)
- Use hash-consing
  - Alpha-equivalent tycons have unique representations
- Aggressive memoization
  - "free tyvars" for faster substitution
  - "result of each reduction" so don't have to do it again and again for future ones.

The FLINT Approach (cont'd)

- Typechecking also requires memoization
  - To check if "T->T" is well kinded, don't check "T" twice
  - What about "T -> ∃ t. (... T ...) " ?

    Solution: must take consideration of the current environment.

- Each little trick pays off --- otherwise the compilation time goes order-of-maganitude worse on certain programs

- The technique scales well and is also applicable to new FLINT (which uses calculus of construction).

The bottom line: each compilation stage must preserve the asymptotic time & space usage in representing and manipulating types.

Part VI. Conclusions

Conclusions

- Mobile code will define the platform
- But how to build a high-quality system that can certify advanced security policies?
  - Open question: do we really need to enforce these policies? what are the killer apps?
- What would be the APIs for these libraries of certified binaries ?
- Can the same also serve as a "multi-language and runtime infrastructure" ?

Conclusions --- What We Believe

- Any property (e.g., safety, security, resource usage) that Joe Hacker can understand can be specified, formalized, and reasoned about in a formal logic.
- A variant of low-level IL with a rich type system can form the world's most compact & expressive mobile-code language.
  - Propositions & proofs are represented using a typed λ-calculus
- Types & proofs can be used to certify various runtime services which can then be moved into a certified library---making the VM smaller & more extensible.
- Common typed ILs allow multiple prog. languages to share the same runtime system. Types serve as a glue to achieve principled language interoperation.