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

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!

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
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!

How to build a compiler that can produce certified binaries?

Solution: use "typed" intermediate and "typed" assembly languages!
Phases of a Compiler

Traditional compiler

- **type-check**
- **Source Program**
- **Parse, Semantic**
  - High-level intermed. lang (IL)
- **Analysis, Optimization**
- **Medium-level IL**
- **Medium-level IL**
- **Code Generation**
- **Low-level IL**
- **Register Allocation**
- **Machine Language**

Certifying compiler

- **type-check**
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

Source Program
- ML
- Java
- Safe C
  - Parse & semantic
  - Parse & semantic
  - Parse & semantic

High-level FLINT
- type-check
  - Analysis & Optimization

Medium-level FLINT
- type-check
  - Code Gen.

Low-level FLINT
- type-check

Add new front-ends to build new certifying compilers!

Hints

Program for execution

Prover & Checker

Safety Theorem

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 \( \lambda \)-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”)

6/22/2001
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 Fundec₁, …, Fundecₙ IN Exp.
Fundec ::= FUN f(x₁, …, xₙ) = Exp;
Exp ::= RETURN (Val)
   | x = CALL (Val, Val₁, …, Valₙ); Exp
   | x = PrimOp(Val₁, …, Valₙ); Exp
   | IF CmpOp (Val₁, …, Valₙ) THEN Exp₁ ELSE Exp₂
   | SWITCH Val OF (Exp₁, …, Expₙ)
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 Fundec₁, ..., Fundecₙ IN Exp.
Fundec ::= FUN f(x₁ : Ty₁, ..., xₙ : Tyₙ) : Ty = Exp;
Exp ::= RETURN (Val)
| x = CALL (Val, Val₁, ..., Valₙ) ; Exp
| x = PrimOp(Val₁, ..., Valₙ); Exp
| IF CmpOp (Val₁, ..., Valₙ) THEN Exp₁ ELSE Exp₂
| SWITCH Val OF (Exp₁, ..., Expₙ)
Val ::= f | x | n
PrimOp ::= i32add | i32sub | ... | r64add | ... | alloc | upd | cast ...
CmpOp ::= ieq | ineq | ilt | igt | ...

Ty ::= Int₃₂ | Real₆₄ | ... | Code(Ty₁, ..., Tyₙ) : Ty

The Type Lang.: a set of representation specifications
To type-check: all primops must be given a type (what about "cast"?)
Some use Continuation-Passing Style (a variant of SSA):

| Program ::= | LETREC Fundec₁, ..., Fundecₙ IN Exp. |
| Fundec ::= | FUN f(x₁ : Ty₁, ..., xₙ : Tyₙ) : Ty = Exp; |
| Exp ::= | RETURN (Val) |
|         | x = CALL (Val, Val₁, ..., Valₙ) ; Exp |
|         | x = PrimOp(Val₁, ..., Valₙ); Exp |
|         | IF CmpOp (Val₁, ..., Valₙ) THEN Exp₁ ELSE Exp₂ |
|         | SWITCH Val OF (Exp₁, ..., Expₙ) |
| Val ::= | f | x | n |
| PrimOp ::= | i32add | i32sub | ... | r64add | ... | alloc | upd | cast |
| CmpOp ::= | ieq | ineq | ilt | igt | ... |
| Ty ::= | Int₃₂ | Real₆₄ | ... | Code(Ty₁, ..., Tyₙ) : 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
- \( \text{Int}_{32}, \text{Word}_{32}, \text{Real}_{32}, \text{Real}_{64}, \ldots \)
- Subrange type (enumerate) \( Rng(1,10) \) --- all numbers from 1 to 10
- Primops: basic arithmetics, comparisons, ...

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)

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

But what is “N_1” and “N_2”?
The Type Language (cont’d)

Types should not be dependent upon terms, so “N” here is different from those term-level constants.

(type) $T ::= \text{Int}$
  $| \text{Real}$
  $| \text{Rng}(N_1, N_2)$
  $| \text{Code}(T_1, \ldots, T_n)$
  $| \text{Tup}(T_1, \ldots, T_n)$

(num) $N ::= 0 | 1 | 2 | \ldots$
Let's also formalize the type sequence \((T_1, \ldots, T_n)\):

\[
\begin{align*}
\text{(type)} \quad T &::= \text{Int} \\
&\quad | \quad \text{Real} \\
&\quad | \quad \text{Rng}(N_1, N_2) \\
&\quad | \quad \text{Code}(R) \\
&\quad | \quad \text{Tup}(R)
\end{align*}
\]

\[
\begin{align*}
\text{(num)} \quad N &::= 0 \mid 1 \mid 2 \mid \ldots
\end{align*}
\]

\[
\begin{align*}
\text{(row)} \quad R &::= \emptyset \quad \text{empty sequence} \\
&\quad | \quad T ; R \quad \text{add one more}
\end{align*}
\]
The Type Language (cont’d)

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

\[
\text{(tycon)} \quad T ::= \text{Int} \mid \text{Real} \mid \text{Rng}(T, T') \mid \text{Code}(T) \mid \text{Tup}(T) \\
\quad \mid 0 \mid 1 \mid 2 \mid \ldots \mid \emptyset \mid T ; T'
\]

Each tycon now must be “kinded” --- to prevent bad forms

\[
\text{(kind)} \quad K ::= \text{Tp} \mid \text{Num} \mid \text{Row} \mid K \to K'
\]

Examples:

<table>
<thead>
<tr>
<th>Tycon</th>
<th>Kind</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int</td>
<td>Tp</td>
</tr>
<tr>
<td>Real</td>
<td>Tp</td>
</tr>
<tr>
<td>Rng</td>
<td>Num -&gt; Num -&gt; Tp</td>
</tr>
<tr>
<td>Code</td>
<td>Row -&gt; Tp</td>
</tr>
<tr>
<td>Tup</td>
<td>Row -&gt; Tp</td>
</tr>
</tbody>
</table>

\[\emptyset : \text{Row} \quad 0 : \text{Num} \quad 1 : \text{Num}\]
Let's make it more extensible:

(kind) \( K ::= C | K \rightarrow K' \)
(primknd) \( C ::= Tp | \text{Num} | \text{Row} \)
(tycon) \( T ::= P | T(T') \)
(primtyc) \( P ::= \text{Int} | \text{Real} | \text{Rng} | \text{Code} | \text{Tup} \)
\[ | 0 | 1 | ... | \emptyset | ; \]

primtyc signature:

- Int : Tp
- Real : Tp
- Rng : Num \text{->} Num \text{->} Tp
- Code : Row \text{->} Tp
- Tup : Row \text{->} Tp
- \emptyset : Row
- ; : Tp \text{->} Row \text{->} Row
- 0 : Num
- 1 : Num
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 primitive:

\[ \text{Sint} : \text{Num} \to \text{Tp} \]

Typing rules for the term-level integer constant:

\[ 1 : \text{Sint}(1) \quad 2 : \text{Sint}(2) \quad \ldots \quad n : \text{Sint}(n) \]

If a value “v” has type \( \text{Sint}(n) \), it must be equal to “n”.

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

\textbf{Sint} can certify array-bounds checking or “tagged record”

\[ \text{Tup}(\text{Sint}(1); \text{Int}; \text{Real}) \]

\[ 1 \quad 40 \quad 3.14 \]
Sum Type (a.k.a. Tagged Union)

New primtype:

\[ \text{Sum} : \text{Row} \rightarrow \text{Tp} \]

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

\[ \text{Sum} \left( \text{Tup}(\text{Sint}(0); \text{Int}) ; \right.
\left. \text{Tup}(\text{Sint}(1); \text{Real}; \text{Real}) \right) \]

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 \( \text{Sum}(\text{Sint}(0), \text{Ref}(T)) \)
(assuming “0” for NULL ptr, \( \text{Ref} \) for non-null ptr)
Recursive Type

How to represent recursive types?

```c
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 primitive:

\[ \mu : (Tp \rightarrow Tp) \rightarrow Tp \]

Type language now has to support type variable & function:

(kind) \[ K ::= C \mid K \rightarrow K' \]

(primknd) \[ C ::= Tp \mid \text{Num} \mid \text{Row} \]

(tycon) \[ T ::= P \mid T(T') \mid t \mid \lambda t : K. T \]

(primtyc) \[ P ::= ... \mid \mu \]

An example “type function”:

\[ F = \lambda t : Tp . \text{Sum} ( \text{Sint}(0) ; \text{Tup}(t, t, \text{int}) ) \]

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 \( F(\mu(F)) \) into \( \mu(F) \)

The “unfold” primop casts a value of type \( \mu(F) \) into \( F(\mu(F)) \)

For example, let

\[
\text{Tree} = \mu ( \lambda t : Tp . \text{Sum ( Sint(0) ; Tup(t, t, int) ) } )
\]

Unfolding a value of type \( \text{Tree} \) results in a value of type

\[
\text{Sum (Sint(0) ; Tup (Tree, Tree, int) )}
\]

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:

Code : (Tp -> Row) -> Tp

The term language has to be extended as well:

Fundec ::= FUN f [t : Tp ] (x₁ : T₁, ..., xₙ : Tₙ) = Exp;
Exp ::= ..........
| CALL (Val, T, Val₁, ..., Valₙ)

Replace “Tp” above with “K”, we get polymorphism over arbitrary kinds.
Polymorphic Type (cont’d)

A polymorphic identify function (in CPS):

\[
\text{FUN } f \ [t \ : \ Tp] \ (x : t, \ c : \text{Code}[](t)) \\
= \text{CALL} (c, [], x)
\]

Even monomorphic functions are polymorphic over “stack” and “callee-save registers” (at low level)

\[
\text{FUN } f \ [t \ : \ Tp] \ (r1 : t, \ r3 : \text{Code}[](r1 : t)) = \ldots
\]

Types still don’t have any effect on “terms” --- all “red” stuff can be erased \ldots
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 primitive:

$$\exists : (T_p \rightarrow T_p) \rightarrow T_p$$

Or if over arbitrary kinds:

$$\exists : (K \rightarrow T_p) \rightarrow T_p$$

Intset is $$\exists(F)$$ where

$$F = \lambda t : T_p . \text{Tup}(\text{Int} \rightarrow t \rightarrow \text{Bool}; \text{Int} \rightarrow t \rightarrow t)$$

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

$$F = \lambda t : T_p . \text{Tup}(t; \text{Tup}(t; \text{Int} \rightarrow \text{Int})$$

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

$$F = \lambda t : T_p . t$$
Existential Type (cont’d)

Two new forms of “cast” for handling existentials:

\[
\text{Exp} ::= \ldots \\
\quad | \quad x = \text{PACK} (t=T, \text{Val : } T') ; \quad \text{Exp} \\
\quad | \quad (t, x) = \text{OPEN} (\text{Val}) ; \quad \text{Exp}
\]

PACK casts a value of type \( F(T) \) into \( \exists(F) \)

OPEN casts a value of type \( \exists(F) \) 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 calculus”

(kind) \[ K ::= C \mid K \rightarrow K' \]

(primknd) \[ C ::= Tp \mid \text{Num} \mid \text{Row} \]

(tycon) \[ T ::= P \mid t \mid \lambda t : K . T \mid T(T') \]

(primtyc) \[ P ::= \text{Int} \mid \text{Real} \mid \text{Rng} \mid \text{Code} \mid \text{Tup} \]
\[ \mid 0 \mid 1 \mid \ldots \mid \emptyset \mid ; \]
\[ \mid \text{Sint} \mid \text{Sum} \mid \exists \mid \mu \]

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 involves 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 primitive (for integer array):

\[
Array : \text{Num} \rightarrow \text{Tp}
\]

A integer array of length “10” has type

\[
Array(10)
\]

Array creation is polymorphic over “length”:

\[
\text{mkArray} : \forall n : \text{Num}. (\text{Sint}(n) \rightarrow \text{Array}(n))
\]

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

\[
\text{sub} : \forall n : \text{Num}. \forall i : [0, n-1]. (\text{Array}(n) \rightarrow \text{Sint}(i) \rightarrow \text{Int})
\]

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 \cite{TT94,CWM99}

Region and capabilities are new primknd:

\[
\begin{align*}
\text{(kind)} & \quad K ::= C \mid K \rightarrow K' \\
\text{(primknd)} & \quad C ::= \text{Tp} \mid \text{Num} \mid \text{Row} \mid \text{Reg} \mid \text{Cap}
\end{align*}
\]

Every heap data lives in a region:

\[
\text{Tup} \quad : \quad \text{Reg} \rightarrow \text{Row} \rightarrow \text{Tp}
\]

Use capabilities to track which region is accessible 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

```haskell
datatype rttag = RT_INT
               | RT_PAIR of rttag * rttag
               | ........
```

```haskell
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 $T_p$
- Require recursive transformation (only ok if it is primitive recursive)
- $T_p$ should be defined as an inductive definition [TSS'00]

Example: mutator view $\rightarrow$ 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 conjunction “A and B” is a pair of (p, q)
where p is proof of A and q is proof of B.

A proof of disjunction “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

<table>
<thead>
<tr>
<th>Languages</th>
<th>Challenges</th>
<th>Approaches</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ML</strong></td>
<td>module system (functor) closures &amp; polymorphism recursive data type</td>
<td>prim/record/code $\exists \forall \mu \text{ ref exn}$</td>
</tr>
<tr>
<td><strong>Java &amp; JVML</strong></td>
<td>classes; interfaces; objects access control &amp; privacy name-based subtyping dynamic linking &amp; loading reflections; concurrency</td>
<td>prim/record/code $\exists \forall \mu \text{ row-kind}$ $???$</td>
</tr>
<tr>
<td><strong>“Safe” C</strong></td>
<td>explicit memory management efficient array access</td>
<td>region &amp; alias types singleton types</td>
</tr>
</tbody>
</table>
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

\( \omega \) type system features used:
- polymorphism for code reuse (inheritance)
- existential types for privacy
Sample Java Program

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

```java
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: \( Pt \ p \)

```java
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: \textit{Spt sp}

\begin{verbatim}
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; }
}
\end{verbatim}
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; } }
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
\[ K ::= \bigcirc \mid K_1 \rightarrow K_2 \mid R^L \]

Types
\[ T ::= \ddagger \mid T_1 \rightarrow T_2 \mid \lambda t:K.T \mid T_1 T_2 \mid \forall t:K.T \mid \emptyset \mid \{ T \} \mid \text{Ref } T \mid \mu \ddagger.T \mid \exists t:K.T \mid \text{Typ}(x) \]

Terms
\[ e ::= x \mid \lambda x:T.e \mid x_1x_2 \mid \text{let } x = e_1 \text{ in } e_2 \mid \Delta t:K.e \mid x[T] \mid \{ l_1=e_1 \ldots l_n=e_n \} \mid e.l \mid \text{ref } e \mid \!e \mid e_1:=e_2 \mid \text{fold}(e,T) \mid \text{unfold } e \mid <t:K=T, e:T'> \mid \text{Val}(x) \]
Extensible Records in Mini-FLINT

Type context

\[ A ::= \varepsilon \mid A, x : T \mid A, t : K \]

Type formation

\[
\begin{align*}
A &\vdash \text{ok} \quad & A &\vdash T : \Omega \\
A &\vdash \emptyset^L : R^L & A &\vdash l : T; T' : R^{\Omega(1)} \\
A &\vdash T : R^\emptyset & A &\vdash \{ T \} : \Omega \\
\end{align*}
\]

Term formation

\[
\begin{align*}
A &\vdash e_i : T_i \quad \text{where } i=1,\ldots,n \\
A &\vdash \{ l_1 = e_1 \ldots l_n = e_n \} : \{ l_1 : T_1 \ldots l_n : T_n ; \emptyset^{\{l_1,\ldots,l_n\}} \} \\
\end{align*}
\]
Using Extensible Records: An Example

Pt's vtable : { move : $t$ -> int -> void ;
              bump : $t$ -> unit -> void ;
∅{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 ;
∅{move,bump,Zoomable,zoom}
Recursive Types in Mini-FLINT

Type context

\[ A ::= \emptyset \mid A, x : T \mid A, t : K \]

Type formation

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

Term formation

\[ A \vdash e : T[\mu t. T/t] \]
\[ \frac{}{A \vdash \text{fold}(e, \mu t. T) : \mu t. T} \]
\[ A \vdash e : \mu t. T \]
\[ \frac{}{A \vdash \text{unfold} e : T[\mu t. T/t]} \]
Using Recursive Types: An Example

Object Pt : \( \mu t. \{ \text{vtab} : \{ \text{move} : t \to \text{int} \to \text{void} ; \text{bump} : t \to \text{unit} \to \text{void} ; \emptyset \{ \text{move}, \text{bump} \} ; \text{R}\{ \text{move}, \text{bump} \} \} ; \} \)

PtFld : \( \{ x : \text{Ref int} ; \emptyset \{ x \} ; \} \)

\( \emptyset \{ \text{vtab}, \text{PtFld} \} ; \text{R}\{ \text{vtab}, \text{PtFld} \} \} \)

“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 : \( \mu t. \{ \text{vtab} : \{ \text{move} : t \to \text{int} \to \text{void} ; \text{bump} : t \to \text{unit} \to \text{void} ; \emptyset\{\text{move,bump}\} : R\{\text{move,bump}\} \}; \text{PtFld} : \{ x : \text{Ref int} ; \emptyset\{x\} \}; \emptyset\{\text{vtab,PtFld}\} : R\{\text{vtab,PtFld}\} \} \)

Object SPt : \( \mu t. \{ \text{vtab} : \{ \text{move} : t \to \text{int} \to \text{void} ; \text{bump} : t \to \text{unit} \to \text{void} ; \text{subVtab}[t] : R\{\text{move,bump}\} \}; \text{PtFld} : \{ x : \text{Ref int} ; \emptyset\{x\} \}; \text{subFlds} : R\{\text{vtab,PtFld}\} \}; \text{subVtab} : \Omega \to R\{\text{move,bump}\} \)

\( = \lambda t : \Omega. \text{Zoomable} : \ldots \{\text{zoom} : \ldots\} \ldots ; \)
\( \text{zoom} : t \to \text{int} \to \text{void} ; \emptyset\{\text{move,bump,Zoomable,zoom}\} \)
Existential Types in Mini-FLINT

Type context

\[ A ::= \varepsilon \mid A, x : T \mid A, t : K \]

Type formation

\[
\begin{align*}
A, t : K &\quad \frac{T : \Omega}{A \vdash \exists t : K. T : \Omega} \\
A &\quad \frac{\text{Typ}(x) : K}{A \vdash x : \exists t : K. T}
\end{align*}
\]

Term formation (both are “no-ops”)

\[
\begin{align*}
A &\quad \frac{T' : K}{A \vdash T'[T'/t]} \\
A &\quad \frac{\text{Val}(x) : T[\text{Typ}(x)/t]}{A \vdash x : \exists t : K. T}
\end{align*}
\]
Using Existential Types: An Example

Object Pt : \( \exists \) subFlds : \( R\{vtab,PtFld\} \).
\( \exists \) subVtab : \( \Omega \rightarrow R\{move,bump\} \).
\( \mu t. \{ vtab : \{ move : t \rightarrow int \rightarrow void ; \\
    bump : t \rightarrow unit \rightarrow void ; \\
    subVtab[t] : R\{move,bump\} \} \)
            PtFld : \{ x : Ref int ; \( \emptyset \{x\} \} ; \\
            subFlds : R\{vtab,PtFld\} \};

Pt subVtab = \( \lambda t: \Omega. \ \emptyset\{move,bump\} \)

Spt subVtab = \( \lambda t: \Omega. \ \text{Zoomable} : \ldots \{\text{zoom} : \ldots\} \ldots ; \\
    \text{zoom} : t \rightarrow int \rightarrow void ; \\
    \emptyset\{move,bump,Zoomable,\text{zoom}\} \)
Encoding Java Objects in FLINT

Class List {
    private List next;
    public List tail()
    ...
}

"Twin" object:

\[ \mu \text{twin}. \]
\[ \exists \text{subFlds}. \]
\[ \exists \text{subVtab}. \]
\[ \mu \text{self}. \]
\[ \{ \text{vtab: } \{ \text{tail: self-> twin; } \ldots; \}
   \text{subVtab self}\}; \]
\[ \text{ListFld: } \{ \text{next: Ref twin}\}; \]
\[ \text{subFlds} \]
Class List {
    private List next;
    public List tail()
    ...
}

Internal view:
\[\mu \text{twin.} \exists \text{subFlds.} \exists \text{subVtab.} \mu \text{self.} \{vtab: \{tail: self-> twin; ...; subVtab self\}; ListFld: \{next: Ref twin\}; subFlds\} \]
Encoding Java Objects in FLINT

Class code: ListC = <private = ..., ...>

External view:

\( \mu \) twin.
\( \exists \) subFlds.
\( \exists \) subVtab.
\( \mu \) 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 = \subFlds . \subVtab . let ...
      in { bump = \self . \_.
           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 = \subFlds . \subVtab . let ...

  in { bump = \self . \_.
      let this = PackObj (self)
      o = Val(Val(unfold this)))

  in (unfold o).vtab.move self 1 } ...

\mu t. { vtab : { move : t -> int -> void ;
                bump : t -> unit -> void ;
                subVtab t }

PtFld : { x : Ref int ; \emptyset{x} };

subFlds};
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:

\[
\begin{align*}
T_1 &= \text{int} \rightarrow \top \\
T_2 &= T_1 \rightarrow T_1 \\
& \vdots \\
T_n &= T_{n-1} \rightarrow T_{n-1}
\end{align*}
\]

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-magnitude 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.