A Type-Preserving Compiler Infrastructure

Christopher League
Yale University
17 May 2002

Advisor
Zhong Shao

Committee
Kim Bruce
Arvind Krishnamurthy
Carsten Schürmann

Acknowledgment
Valery Trifonov

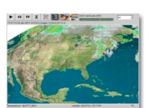
Thesis

- A strongly-typed compiler intermediate language can safely and efficiently accommodate very different programming languages

Mobile code, pervasive networks

- Wireless handheld computers

- Remotely programmable devices



- Browser applets

- Widely distributed computation

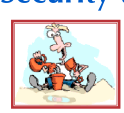
Security is critical

- We might not completely trust the programs we receive and run
- Must ensure they does not misbehave:
  - crash the device
  - exhaust resources
  - interfere with other programs/data
- Correctness is hard—focus on safety

Security toolbox: digital signature

- Confirms identity of producer, not safety of code
- I might not trust Microsoft, but would still run the code—assuming it is harmless

Security toolbox: reference monitor



- Code runs in a sandbox
  - interactions with outside world mediated by the monitor
- Hardware mechanisms
  - expensive context switches
  - not available on all devices
- Software rewriting
  - frequent dynamic checks

impractical for monitoring *fine-grained* properties

Security toolbox: language features

- Array bounds checking
- Garbage collection
- Exceptions
- Encapsulation, access control
- Type systems

Many vulnerabilities *are* type errors

Date	System	Kind
1/24/02	AOL ICQ	remotely exploitable buffer overflow
1/14/02	Solaris CDE	buffer overflow vulnerability
12/20/01	MS u-PNP	buffer overflow vulnerability
12/12/01	SysV 'login'	remotely exploitable buffer overflow
11/29/01	WU ftpd	format string vulnerability; `free()` on unallocated pointer
11/21/01	HP-UX lpd	remotely exploitable buffer overflow
10/25/01	Oracle9i AS	remotely exploitable buffer overflow
10/5/01	CDE ToolTalk	format string vulnerability

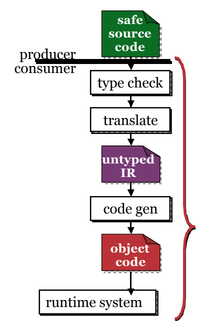
[CERT advisories]

The need for typed machine code

(ML, Haskell, Java, C#)

- Is it enough to program in type-safe languages?

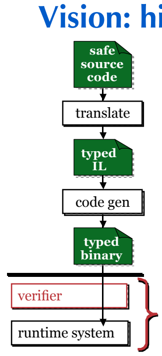
The need for typed machine code



- Is it enough to program in type-safe languages? **No.**
  1. Microsoft is unlikely to ship source code
  2. Still must trust compiler

  [Thompson 1984]
- Trusted Computing Base
  - system modules responsible for supporting security policy

Vision: high assurance systems
with minimal TCB

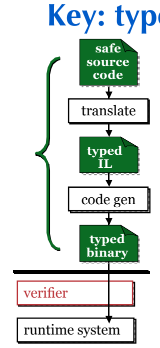


1,000 LOC *vs.* 30,000+

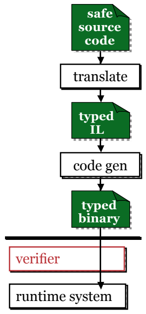
(We are not there yet!)

"Foundational proof-carrying code"
[Appel et al. 2001]

Key: type-preserving compiler



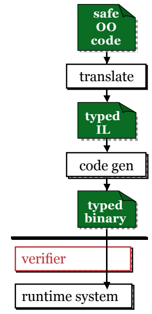
Until now: functional languages



or "safe C"

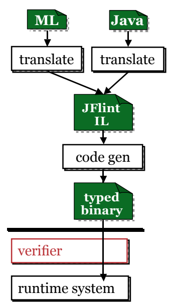
- TIL/ML
  [Tarditi, Morrisett et al. 1996]
- FLINT/ML
  [Shao 1997]
- Touchstone
  [Necula & Lee 1998]
- Popcorn
- Scheme––
  [Morrisett et al. 1999]

Real world: object-oriented



- Widely used (Java, C++)
- New, complex features
  - dynamic binding
  - dynamic casts
  - inheritance
  - interfaces
  - name-based types

Contributions



- Single typed IL that supports Java and ML
  - Not partial to either language
- Novel techniques for encoding most features of Java
  - Same efficiency as untyped
  - Preserves safety guarantees
- New high-level IL for Java
  - JVM primitives with functional flow
  - Easier to verify and optimize

Outline

- Mobile code security
- The need for typed machine code
- Dynamic dispatch security hole
- How our approach is different
- Safe and efficient object encoding
- Functional Java byte code
- A prototype compiler for Java and ML

Dynamic binding: essential to OO

- Inheritance without polymorphism is possible, but certainly not very useful.
- One can declare derived types, but the actual operation being called is always known at compile time.

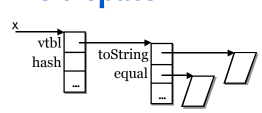
[Booch 1994]

Efficient dynamic dispatch

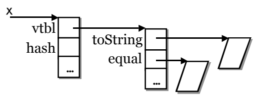
```
public static void
example ( Object x, Object y )
{  x.toString( );

   // compiles to:
   (null check)
   let r1 = x.vtbl ;        // method suite
   let r2 = r1.toString ;   // method pointer
   r2( x )                  // "self application"

}
```



Must ensure the call is safe




```
public static void
example ( Object x, Object y )
{  x.toString( );

   // compiles to:
   (null check)
   let r1 = x.vtbl ;
   let r2 = r1.toString ;
   r2( x ) ;              // this is sound
   r2( y )                // this is not!
}
```

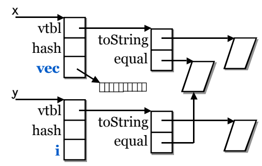
Any unsoundness can be exploited

Cast arbitrary integer to pointer

```
class Ref extends Object
{  public byte[ ]  vec ;
   public String toString( )
   {  vec[13] = 42 ;
      return "Ha!" ;
   }
}

class Int extends Object
{  public int  i ;
}

example( new Ref(...),
         new Int(...) );
```




```
public static void
example ( Object x, Object y )
{  let r1 = x.vtbl ;
   let r2 = r1.toString ;
   r2( y )  // wrong!
}
```

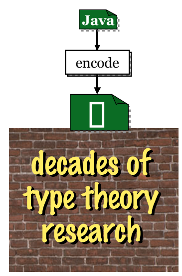

This is a major security hole

in Cedilla Systems' PCC [Colby et al. 2000]

Special J

- Java compiler; generates annotated *x86*
- Types retain Java abstractions
  - (object_ty C)
  - (method_ty C SIG)
- Faulty axiom; undetected for 3+ years

Our approach is different



- Start with a strong foundation
  - "off the shelf" type theory
  - not specific to Java (or ML)
  - simple soundness proof
- Design complex encodings of Java features that
  - preserve type safety
  - maintain efficiency

Others have encoded objects

in type theory

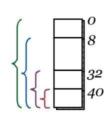
[Cardelli 84] [Cook et al. 89]
[Pierce & Turner 94] [Bruce 94]
[Hofmann & Pierce 95]
[Abadi, Cardelli, Viswanathan 96]

- These are models of OOP
- Rather inefficient
  - extra indirections
  - extra function calls
- Assumed subsumption
  - a Circle is also a Shape

Type theory toolbox: rows

[Rémy 1993]

- Intuition: the suffix or *tail* of a record type, starting at a given offset



Type theory toolbox: recursion

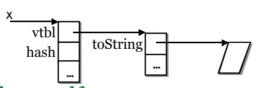
- Intuition: μ notation for recursive definitions
  - list ≡ { data: int, next: list }
- Replace recursive ref. with a type variable:
  - list ≡ μ□.{ data: int, next: □ }
  - let *x* = fold *y* as list
  - let *y* : { data: int, next: list } = unfold *x*

Type theory toolbox: existentials

[Mitchell & Plotkin 1988]

- Intuition: hide a type from outsiders
  - enforces abstract data types
  - let *x1* : □□. { z : □, f : □ → string }
    = □□=int, { z=42, f=int2string }
    : { z : □, f : □ → string } □
  - open *x1* as □□, *y* : { z : □, f : □ → string } □
    in *y*.f (*y*.z)

The type of Object



ObjTy[Object] ≡
  □***fs***::R⁸, ***ms***::R⁴.
  μ ***self***. ObjRcd[Object] *fs ms self*

ObjRcd[Object] ***fs ms self*** ≡
 { vtbl : { toString : *self* → string ;
   ms } ;
  hash : unsigned ;
  fs }

- Hide the differences between sub- and super-class.

Provably safe method invocation…

example (*x*, *y* : ObjTy[Object]) ≡
open *x* as □***fx, mx, x1*** : μ ***self*** . ObjRcd[Object] *fx mx self* □; (*rx*)
let *x2* : ObjRcd[Object] *fx mx rx* = unfold *x1* ;
let *r1* : { toString : *rx* → string ; *mx* } = *x2*.vtbl ;
let *r2* : *rx* → string = *r1*.toString ;
r2(*x1*)

ObjTy[Object] ≡
  □***fs***::R⁸, ***ms***::R⁴.
  μ ***self***. ObjRcd[Object] *fs ms self*

ObjRcd[Object] ***fs ms self*** ≡
{ vtbl : { toString : *self* → string ;
  ms } ;
 hash : unsigned ;
 fs }

…without sacrificing efficiency

example (*x*, *y* : ObjTy[Object]) ≡
open *x* as □***fx, mx, x1*** : μ ***self*** . ObjRcd[Object] *fx mx self* □; (*rx*)
let *x2* : ObjRcd[Object] *fx mx rx* = unfold *x1* ;
let *r1* : { toString : *rx* → string ; *mx* } = *x2*.vtbl ;
let *r2* : *rx* → string = *r1*.toString ;
r2(*x1*)

ObjTy[Object] ≡
  □***fs***::R⁸, ***ms***::R⁴.
  μ ***self***. ObjRcd[Object] *fs ms self*

ObjRcd[Object] ***fs ms self*** ≡
{ vtbl : { toString : *self* → string ;
  ms } ;
 hash : unsigned ;
 fs }

Techniques extend to most of Java

- classes
- inheritance
- dynamic dispatch
- dynamic cast
- mutual recursion
- interfaces
- constructors
- super calls
- subroutines
- exceptions
- privacy

- Chapters 3–5 contain:
  - Formal definition of source language (Featherweight Java)
  - Formal definition of intermediate language (JFlint)
  - Proofs that JFlint is sound and decidable
  - Type-directed translation
  - Proof that well-typed inputs yield well-typed outputs

Act III

- Mobile code security
- The need for typed machine code
- Dynamic dispatch security hole
- How our approach is different
- Safe and efficient object encoding
- Functional Java byte code
- A prototype compiler for Java and ML

System building

- A prototype compiler for Java and ML
- Many practical problems must be solved
  - Efficient implementation of IL
  - Large semantic gap between Java and JFlint

Where to start?

- Java
  x.println(y);
  ↓ javac
- JVML byte code

```
3  aload_0   # this
4  getfield PrintStream C::x
7  dload 2
9  invokevirtual
      void PrintStream::println(double)
```

- Many details are not explicit in Java source
- Java byte code has untyped local vars & implicit data flow

Two sets of concerns

1. data & control flow, type inference
2. expanding Java primitives

- JVML byte code ——→ JFlint

```
3  aload_0   # this
4  getfield PrintStream C::x
7  dload 2
9  invokevirtual
      void PrintStream::println(double)
```


A new IL to bridge the gap

- High-level Java primitives, types
- JVML ——→ □JVM ——→ JFlint
- Functional control and data flow

A better Java byte code

- JVML ——→ □JVM ——→ JFlint
- Fully explicit
  - Supports all of JVML, yet is
  - Easier to verify and optimize
- Nastiest parts of JVM become tractable
  - Object initialization [Freund & Mitchell 1999]
  - Subroutines [Stata & Abadi 1998]
- Verification is just simple type checking
  - < 260 lines of ML code

[Chapter 7]

Example: Factorial

```
public int fact(int n)
{ int x ;
  for ( x = 1;  n > 0;  n−−)
     x = x □ n ;
  return x ;
}
```

```
public int fact (int n)
{    iconst_1
     istore_1       # x=1
     goto T
L    iload_1
     iload_0
     imul
     istore_1       # x=x □ n
     iinc 0 −1      # n−−
T    iload_0
     ifgt L         # n > 0
     iload_1
     ireturn
}
```


Example: Factorial

```
public int fact(int n) =
letrec L = □( i : I, x : I ) .
          let y = x □ i ;
          let j = i − 1 ;
          T ( j, y )
and T = □( k : I, z : I ) .
          if n > 0 then L ( k, z )
          else return z
in T ( n, 1 )
```

```
public int fact (int n)
{    iconst_1
     istore_1       # x=1
     goto T
L    iload_1
     iload_0
     imul
     istore_1       # x=x □ n
     iinc 0 −1      # n−−
T    iload_0
     ifgt L         # n > 0
     iload_1
     ireturn
}
```


Subroutines are tricky

- **jsr** *offset* — push return address, jump
- **ret** *var* — return to address in *var*
- Used to implement 'finally' blocks

```
try { A }
catch (Error e) { B; throw e }
finally { C }
```

- Can achieve complicated control flow

Continuation-passing style

[Steele 1978] [Kranz et al. 1986]

- The higher-order answer to flexible control flow
  - Represent return address as a function

1. *ret* need not obey stack discipline

```
void main(String[] args)
{  try { }
   finally
   {  while(true)
      {  try { }
         finally { break; }
      }
   }
}
```

```
M   jsr A
    goto R
A   astore_1    # return addr
L   jsr B
    goto L
B   pop         # return addr
    ret 1
R   return
```


1. *ret* need not obey stack discipline

```
letrec M = □( ) . A(R)
and    A = □( k1 : ()→V ) .
           letrec L = □( ) . B(L)
           and B = □( k2 : ()→V ) .
                     k1( )
           in L( )
and    R = □( ) . return
in M( )
```

```
M   jsr A
    goto R
A   astore_1    # return addr
L   jsr B
    goto L
B   pop         # return addr
    ret 1
R   return
```

2. Subroutine might update local var

```
letrec M = □( ) . S(43, P)
and   S = □( i : I, k : (I)→V ) .
            let j = i – 1 ;
            k( j )
and   P = □( i : I ) .
            invoke printInt (i) ;
            return
in M( )
```

```
M   ldc 43
    istore_1      # i=43
    jsr S
    goto P
S   astore_2      # return addr
    iinc 0 –1     # i––
    ret 2
P   iload_1       # print i
    invoke printInt
    return
```


3. Polymorphic over untouched vars

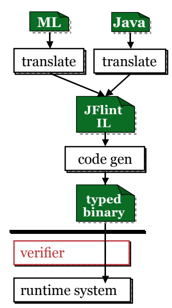
```
letrec S = □( k : ()→V ) . k() in
letrec M = □( ) .
   let x1 = 3.14 ;
   letrec I = □( ) .
      invoke printFloat (x1);
      let x2 = 42 ;
      letrec R = □( ) .
         invoke printInt (x2);
         return
      in S(R)
   in S(I)
in M( )
```

```
M   ldc 3.14
    fstore_1      # x=3.14
    jsr S
I   fload_1
    invoke printFloat
    ldc 42
    istore_1      # x=42
    jsr S
R   iload_1
    invoke printInt
    return
S   astore_2      # return addr
    ret 2
```


System overview

- Based on SML/NJ compiler 110.30
  - added a new type-preserving Java front end
  - interactively loads and runs Java classes
- Same back end and runtime system
- Front end □ 8k LOC; JFlint checker □ 1k LOC
- Runs CaffeineMark 3.0 (12 classes, □ 100k)
- Compile time just 60% longer than gcj
- Does not load native code!

Synergy with ML front end



- The type system is impartial

JFlint	Java	ML
□	inheritance	parametric poly.
□	object enc.	closures
µ	rec. classes	rec. datatypes
tags	dynamic cast	exceptions
rows	object enc.	—
records	vtable, objects	records, tuples
functions	methods	functions

Summary

- Mobile code security is critical
- High-assurance systems need minimal TCB
- Type-preserving compilers are the key
- With care, they scale to real languages
  - use type theory as the foundation;
  - focus on practical encodings
- A single typed IL can safely and efficiently support different kinds of source languages