Nikhil Swamy
Juan Chen
Ravi Chugh

FINE + DCIL

End-to-end Verification of Security Enforcement

Security Policies

- Many languages/logics to specify security policies
  - XACML, DKAL, SecPAL, DCC, SD3, Binder, Ponder, CDatalog, RT, SPKI/SDSI, …
- Policies address a variety of security concerns
  - Authentication, authorization, usage controls, information flow, security automata, …
- But, disconnect between specification and implementation

- SecPAL policy for EHR, an e-health database

```
P can read P's record r
D can read P's record r if D is treating P and r.Subject <> HIV
```

- C# function to enforce this policy

Okay to read record before security check?

```
public Record GetRecord (string pat, string recId) {
    Record rec = m_data.GetRecord (pat, recId);
    if (rec != null &&
          GetAuthContext().query("CanGetRecord", new List { pat, recId,
                                                            rec.Author,
                                                            rec.Subject }))
    {
        return result;
    }
    return null;
}
```

- SecPAL policy for EHR, an e-health database

```
P can read P's record r
D can read P's record r if D is treating P and r.Subject <> HIV
```

- C# function to enforce this policy

Okay to read record before security check?

```
public Record GetRecord (string pat, string recId) {
    Record rec = m_data.GetRecord (pat, recId);
    if (rec != null &&
          GetAuthContext().query("CanGetRecord", new List { pat, recId,
                                                            rec.Author,
                                                            rec.Subject }))
    {
        return result;
    }
    return null;
}
```

Can credentials be forged?

- SecPAL policy for EHR, an e-health database

```
P can read P's record r
D can read P's record r if D is treating P and r.Subject <> HIV
```

- C# function to enforce this policy

Okay to read record before security check?

```
public Record GetRecord (string pat, string recId) {
    Record rec = m_data.GetRecord (pat, recId);
    if (rec != null &&
          GetAuthContext() query("CanGetRecord", new List { pat, recId,
                                                            rec.Author,
                                                            rec.Subject }))
    {
        return result;
    }
    return null;
}
```

Can credentials be forged?

Is this the correct authorization query?

- SecPAL policy for EHR, an e-health database

```
P can read P's record r
D can read P's record r if D is treating P and r.Subject <> HIV
```

- C# function to enforce this policy

Okay to read record before security check?

```
public Record GetRecord (string pat, string recId) {
    Record rec = m_data.GetRecord (pat, recId);
    if (rec != null &&
          GetAuthContext().query("CanGetRecord", new List { pat, recId,
                                                            rec.Author,
                                                            rec.Subject }))
    {
        return result;
    }
    return null;
}
```

Can credentials be forged?

Is this the correct authorization query?

- Unclear if policy is enforced correctly

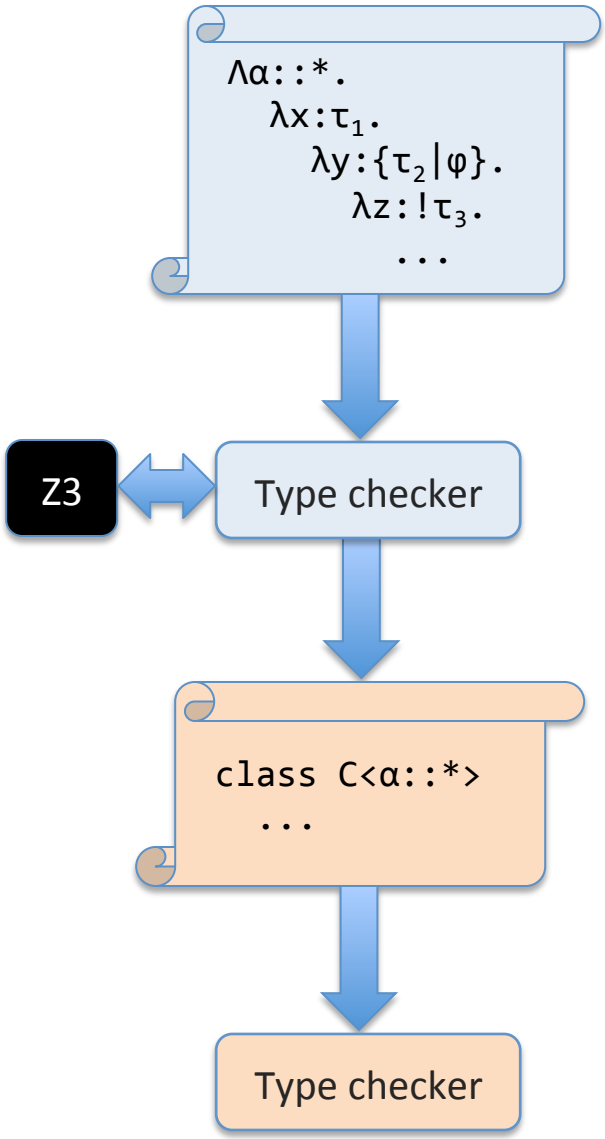
Verifying Security Enforcement by Typing

- Security policies embedded in a program's types
  - Security checking amounts to type checking
    - Volpano and Smith '96
- Many proposals
  - FlowCaml, Jif, Fable, Aura, F7
- But, not yet widely applicable

Limitations of Security by Typing

- Cannot handle many constructs of real policies
  - Jif and FlowCaml: only information flow
  - Aura and Fable: only stateless policies
  - F7: targets stateless authentication
- Either not meant for source programming
  - Fable and Aura require explicit security proofs
- Or cannot be used to generate checkable binaries
  - Jif, FlowCaml, and F7 erase security types

Our Approach: FINE + DCIL



FINE: A source-level type system for F#

- Refinement types for authorization policies
- Dependent types for information flow
- Affine types for state-modifying policies

Expressive

Automated security proof construction

- Type checked with assistance from Z3
- Security proofs synthesized from Z3 proofs

Easier to program

Verifiable binaries by type preservation

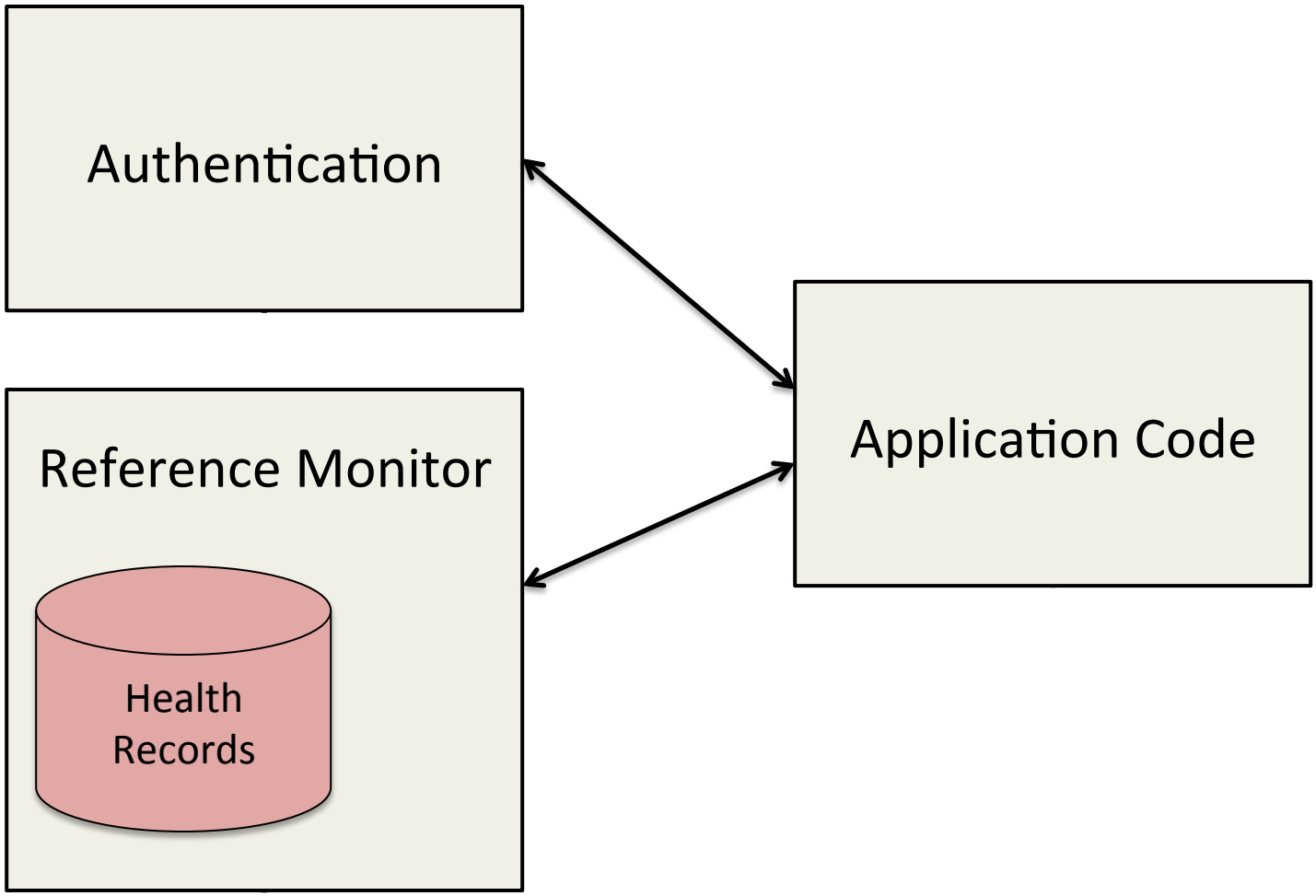
- DCIL: an extension of CIL's type system
- Security proofs carried from source level
- Checked syntactically (without Z3)

Verify with small TCB

Outline

- Overview
- FINE, by example
- Proof terms
- Translation to DCIL
- Results and future work

EHR Application



module Authentication

```
val login : u:prin -> str -> option<cred<u>>
let login u pw =
  if (* password ok? *)
    then Some (MkCred u)
    else None
```

module Authentication

```
type prin =
  | User : str -> prin
  | Admin : prin
```

```
val login : u:prin -> str -> option<cred<u>>
let login u pw =
  if (* password ok? *)
    then Some (MkCred u)
    else None
```

module Authentication

```
type prin =
  | User : str -> prin
  | Admin : prin

type option<α> =
  | Some : α -> option<α>
  | None : option<α>
```

Polymorphic type constructor

- Type parametrized by another type

```
val login : u:prin -> str -> option<cred<u>>
let login u pw =
  if (* password ok? *)
    then Some (MkCred u)
    else None
```

module Authentication

```
type prin =
  | User : str -> prin
  | Admin : prin

type option<α> =
  | Some : α -> option<α>
  | None : option<α>

type cred<p:prin> =
  | MkCred : p:prin -> cred<p>

val login : u:prin -> str -> option<cred<u>>
let login u pw =
  if (* password ok? *)
    then Some (MkCred u)
    else None
```

Dependent type constructor

- Type parametrized by a **term**
- Correlates a particular principal with a credential

module Authentication

```
type prin =
  | User : str -> prin
  | Admin : prin

type option<α> =
  | Some : α -> option<α>
  | None : option<α>

type cred<p:prin> =
  | MkCred : p:prin -> cred<p>

val login : u:prin -> str -> option<cred<u>>
let login u pw =
  if (* password ok? *)
    then Some (MkCred u)
    else None
```

Dependent function type

- Formal parameter is bound in range type
- Way to create values of a dependent type

module Authentication

```
type prin =
  | User : str -> prin
  | Admin : prin

type option<α> =
  | Some : α -> option<α>
  | None : option<α>

type cred<p:prin> =
  | MkCred : p:prin -> cred<p>

val login : u:prin -> str -> option<cred<u>>
let login u pw =
  if (* password ok? *)
    then Some (MkCred u)
    else None
```

Want `login` to be the only way to obtain a credential for a user

module Authentication

```
type prin =
  | User : str -> prin
  | Admin : prin

type option<α> =
  | Some : α -> option<α>
  | None : option<α>

private type cred<p:prin> =
  | MkCred : p:prin -> cred<p>

val login : u:prin -> str -> option<cred<u>>
let login u pw =
  if (* password ok? *)
    then Some (MkCred u)
    else None
```

Want `login` to be the only way to obtain a credential for a user

module EHR_ReferenceMonitor

```
private type record = { patient:prin; subject:str; data:str }
```

```
let read_data p’ c r = r.data
```

module EHR_ReferenceMonitor

```
private type record = { patient:prin; subject:str; data:str }
```

read_data mediates access to patient records

- The user must present a login credential

```
val read_data : p':prin -> cred<p'> ->
                  {x:record | hasPerm p' (CanRead x)} -> str

let read_data p' c r = r.data
```

module EHR_ReferenceMonitor

```
private type record = { patient:prin; subject:str; data:str }
```

`read_data` mediates access to patient records

- The user must present a login credential
- The record to be read must be a record x such that `hasPerm p' (CanRead x)` is true

Refinement type `{x:τ | φ}`

- φ is a FOL+equality formula over propositions

```
val read_data : p':prin -> cred<p'> ->
                {x:record | hasPerm p' (CanRead x)} -> str

let read_data p' c r = r.data
```

module EHR_ReferenceMonitor

```
private type record = { patient:prin; subject:str; data:str }
```

```
val read_data : p':prin -> cred<p'> ->
                 {x:record | hasPerm p' (CanRead x)} -> str

let read_data p' c r = r.data
```

module EHR_ReferenceMonitor

```
private type record = { patient:prin; subject:str; data:str }

type perm =
  | CanWrite : record -> perm
  | CanRead : record -> perm
```

```
val read_data : p':prin -> cred<p'> ->
                  {x:record | hasPerm p' (CanRead x)} -> str

let read_data p' c r = r.data
```

module EHR_ReferenceMonitor

```
private type record = { patient:prin; subject:str; data:str }

type perm =
  | CanWrite : record -> perm
  | CanRead : record -> perm

prop hasPerm :: prin -> perm -> *
```

Propositions

- dependent types that can be used in refinements

```
val read_data : p':prin -> cred<p'> ->
                  {x:record | hasPerm p' (CanRead x)} -> str

let read_data p' c r = r.data
```

module EHR_ReferenceMonitor

```
private type record = { patient:prin; subject:str; data:str }

type perm =
  | CanWrite : record -> perm
  | CanRead : record -> perm

prop hasPerm :: prin -> perm -> *

assume Ax1 :
  forall r:record, hasPerm Admin (CanRead r)

val read_data : p':prin -> cred<p'> ->
                  {x:record | hasPerm p' (CanRead x)} -> str

let read_data p' c r = r.data
```

Security assumptions

- establish ground facts and inference rules for policy

module EHR_ReferenceMonitor

Additional policy rule

```
assume Ax2 : forall r:record. hasPerm r.patient (CanRead r)
```

Can write functions with post-conditions in return values

```
val check : a:prin -> b:perm -> {z:bool | z=true => hasPerm a b}
let check a b = match a, b with
                  | Admin, CanRead _ -> true
                  | _, CanRead r     -> a = r.patient
                  | _                -> false
```

Type Checking

```
val check : a:prin -> b:perm -> {z:bool | z=true => hasPerm a b}
let check a b = match a, b with
                  | Admin, CanRead _ -> true
                  | _, CanRead r     -> a = r.patient
                  | _                -> false
```

Typing context Γ

```
Ax1 : forall r:record, hasPerm Admin (CanRead r)
Ax2 : forall r:record, hasPerm r.patient (CanRead r)
  a : prin
  b : perm
  a = Admin
  b = CanRead tmp
```

Well-typed?

```
true : {z:bool | z=true => hasPerm a b}
```

Valid?

```
Not ( true=true => hasPerm a b)
```

Z3

Type Checking

```
val check : a:prin -> b:perm -> {z:bool | z=true => hasPerm a b}
let check a b = match a, b with
                  | Admin, CanRead _ -> true
                  | _, CanRead r     -> a = r.patient
                  | _                -> false
```

Typing context Γ

```
Ax1 : forall r:record, hasPerm Admin (CanRead r)
Ax2 : forall r:record, hasPerm r.patient (CanRead r)
  a : prin
  b : perm
  a = Admin
  b = CanRead tmp
```

Well-typed?

```
true : {z:bool | z=true => hasPerm a b}
```

Sat?

```
not (true=true => hasPerm a b)
```

Z3

Type Checking

```
val check : a:prin -> b:perm -> {z:bool | z=true => hasPerm a b}
let check a b = match a, b with
                  | Admin, CanRead _ -> true
                  | _, CanRead r     -> a = r.patient
                  | _                -> false
```

Typing context Γ

```
Ax1 : forall r:record, hasPerm Admin (CanRead r)
Ax2 : forall r:record, hasPerm r.patient (CanRead r)
  a : prin
  b : perm
  a = Admin
  b = CanRead tmp
```

Well-typed?

```
true : {z:bool | z=true => hasPerm a b}
```

Sat?

```
not (true=true => hasPerm a b)
```

Z3

Unsat!

Type Checking

```
val check : a:prin -> b:perm -> {z:bool | z=true => hasPerm a b}
let check a b = match a, b with
                  | Admin, CanRead _ -> true
                  | _, CanRead r     -> a = r.patient
                  | _                -> false
```

Typing context Γ

```
Ax1 : forall r:record, hasPerm Admin (CanRead r)
Ax2 : forall r:record, hasPerm r.patient (CanRead r)
a : prin
b : perm
b = CanRead r
```

Well-typed?

```
(a = r.patient) : {z:bool | z=true => hasPerm a b}
```

Sat?

```
not ((a = r.patient)=true => hasPerm a b)
```

Z3

Unsat!

Type Checking

```
val check : a:prin -> b:perm -> {z:bool | z=true => hasPerm a b}
let check a b = match a, b with
                  | Admin, CanRead _ -> true
                  | _, CanRead r     -> a = r.patient
                  | _                -> false
```

Typing context Γ

```
Ax1 : forall r:record, hasPerm Admin (CanRead r)
Ax2 : forall r:record, hasPerm r.patient (CanRead r)
  a : prin
  b : perm
```

Well-typed?

```
false : {z:bool | z=true => hasPerm a b}
```

Sat?

```
not (false=true => hasPerm a b)
```

Z3

Unsat!

Type Checking

```
val check : a:prin -> b:perm -> {z:bool | z=true => hasPerm a b}
let check a b = match a, b with
                  | Admin, CanRead _ -> true
                  | _, CanRead r     -> a = r.patient
                  | _                -> false
```

Type Checking

- Is application code using API correctly?

```
val ehr_app : p:prin -> cred<p> -> record -> option<str>

let ehr_app : p c rec =
  if check p (CanRead rec) then Some (read_data p c rec)
                           else None
```

Type Checking

- Is application code using API correctly?

```
val ehr_app : p:prin -> cred<p> -> record -> option<str>

let ehr_app : p c rec =
  if check p (CanRead rec) then Some (read_data p c rec)
                           else None
```

Well-typed?

```
rec : {x:record | hasPerm p (CanRead x)}
```

Type Checking

- Is application code using API correctly?

```
val ehr_app : p:prin -> cred<p> -> record -> option<str>

let ehr_app : p c rec =
  if check p (CanRead rec) then Some (read_data p c rec)
                           else None
```

Establishes the pre-condition

Well-typed? `rec : {x:record | hasPerm p (CanRead x)}` ✓

Compiling with Explicit Security Proofs

- Proofs can be communicated between systems to attest authorization rights
- Proofs can be logged for auditing access rights
- Mobile code/plugins with explicit proofs can be checked before execution
- Target programs can be checked with a small TCB
  - Translation validation to catch compiler bugs

Types for Proofs

- Proof terms will have type pf<α>
- Types for logical connectives

```
type not<α>
type and<α,β>
type imp<α,β>
...
```

- Universal quantifiers as dependent functions

```
forall r:record, hasPerm Admin (CanRead r)
                    && hasPerm Admin (CanWrite r)

r:record -> pf <and <hasPerm Admin (CanRead r),
                     hasPerm Admin (CanWrite r)>>
```

Embedding Proof Terms in Source

- Before translation to target, "derefine" program
- Remove refinement types (not in DCIL)
- Each expression paired with proof of refinement

```
{x:τ | φ}
```

```
(x:τ * pf<φ>)
```

Dependent pair type

- x is value of first component
- x is bound in type of second component

Proof Combinators

- General proof constructors

```
Tru      :        pf<True>
Contra   : ∀α.    pf<α> -> pf<not<α>> -> pf<False>
UseFalse : ∀α.    pf<False> -> pf<α>
Imp      : ∀α,β.  (pf<α> -> pf<β>) -> pf<imp<α,β>>
Lift     : ∀α.    α -> pf<α>
...
```

- Specialized proof constructors for each prop

```
Sub_1_hasPerm :  p1:prin -> p2:prin -> q:perm ->
                   pf<eq_prin p1 p2> ->
                     pf<hasPerm p1 q> ->
                       pf<hasPerm p2 q>
```

Proof Term for check

```
val check : a:prin -> b:perm ->
              {z:bool | z=true => hasPerm a b}
```

Proof Term for check

```
val check : a:prin -> b:perm ->
            (z:bool * pf<imp<eq_bool z true, hasPerm a b>>)
```

Derefined return type

Proof Term for check

```
val check : a:prin -> b:perm ->
              (z:bool * pf<imp<eq_bool z true, hasPerm a b>>)

let check a b = match a, b with
                  | Admin, CanRead _ -> (true, pf1)
                  | _, CanRead r     -> (a = r.patient, pf2)
                  | _                -> (false, pf3)
```

Need to supply a proof for each branch

Proof Term for check

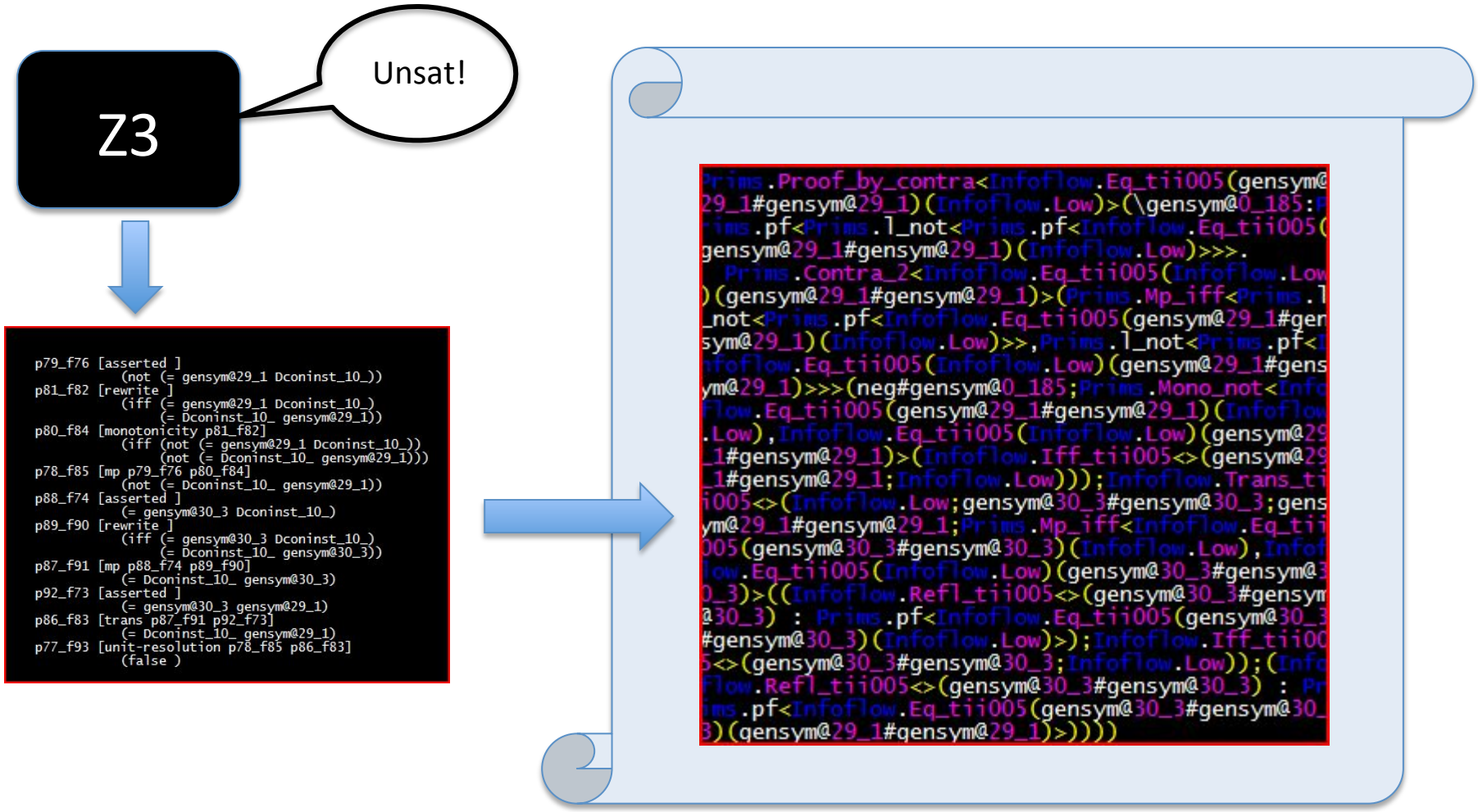
```
val check : a:prin -> b:perm ->
              (z:bool * pf<imp<eq_bool z true, hasPerm a b>>)

let check a b = match a, b with
                  | Admin, CanRead _ -> (true, pf1)
                  | _, CanRead r     -> (a = r.patient, pf2)
                  | _                -> (false, pf3)
```

Unreasonable to write these manually

```
pf1 =
  (Imp <eq_bool true true, hasPerm a b>
    (λfoo: pf<eq_bool true true>.
      (Sub_1_hasPerm Admin a b
        (Refl_prin a : (pf<eq_prin Admin a>))
        (Lift <hasPerm Admin b>
          (Ax1 tmp)))))
```

Automatic Proof Construction

- Z3v2 produces a proof for an unsat formula



```
p79_f76 [asserted ]
            (not (= gensym@29_1 Dconinst_10_))
p81_f82 [rewrite ]
            (iff (= gensym@29_1 Dconinst_10_)
                 (= Dconinst_10_ gensym@29_1))
p80_f84 [monotonicity p81_f82]
            (iff (not (= gensym@29_1 Dconinst_10_))
                 (not (= Dconinst_10_ gensym@29_1)))
p78_f85 [mp p79_f76 p80_f84]
            (not (= Dconinst_10_ gensym@29_1))
p88_f74 [asserted ]
            (= gensym@30_3 Dconinst_10_)
p89_f90 [rewrite ]
            (iff (= gensym@30_3 Dconinst_10_)
                 (= Dconinst_10_ gensym@30_3))
p87_f91 [mp p88_f74 p89_f90]
            (= Dconinst_10_ gensym@30_3)
p92_f73 [asserted ]
            (= gensym@30_3 gensym@29_1)
p86_f83 [trans p87_f91 p92_f73]
            (= Dconinst_10_ gensym@29_1)
p77_f93 [unit-resolution p78_f85 p86_f83]
            (false )
```

```
let proof_fun (neg#gensym@0_185:Prims.pf<Prims.l_not<Prims.pf<Infoflow.Eq_tii005(gensym@29_1#gensym@29_1)(Infoflow.Low)>>>) =

  let p79_f76 = neg#gensym@0_185 in

  let p81_f82 = Infoflow.Iff_tii005<;gensym@29_1#gensym@29_1;Infoflow.Low> in

  let p80_f84 = Prims.Mono_not<Infoflow.Eq_tii005(gensym@29_1#gensym@29_1)(Infoflow.Low),
                               Infoflow.Eq_tii005(Infoflow.Low)(gensym@29_1#gensym@29_1);p81> in

  let p78_f85 = Prims.Mp_iff<Prims.l_not<Prims.pf<Infoflow.Eq_tii005(gensym@29_1#gensym@29_1)(Infoflow.Low)>>,
                             Prims.l_not<Prims.pf<Infoflow.Eq_tii005(Infoflow.Low)(gensym@29_1#gensym@29_1)>>;p79;p80> in

  let p88_f74 = (Infoflow.Refl_tii005<>(gensym@30_3#gensym@30_3)
                 : Prims.pf<Infoflow.Eq_tii005(gensym@30_3#gensym@30_3)(Infoflow.Low)>) in

  let p89_f90 = Infoflow.Iff_tii005<;gensym@30_3#gensym@30_3;Infoflow.Low> in

  let p87_f91 = Prims.Mp_iff<Infoflow.Eq_tii005(gensym@30_3#gensym@30_3)(Infoflow.Low),
                             Infoflow.Eq_tii005(Infoflow.Low)(gensym@30_3#gensym@30_3);p88;p89> in

  let p92_f73 = (Infoflow.Refl_tii005<>(gensym@30_3#gensym@30_3)
                 : Prims.pf<Infoflow.Eq_tii005(gensym@30_3#gensym@30_3)(gensym@29_1#gensym@29_1)>) in

  let p86_f83 = Infoflow.Trans_tii005<;Infoflow.Low;gensym@30_3#gensym@30_3;gensym@29_1#gensym@29_1;p87;p92> in

  let p77_f93 = Prims.Contra_2<Infoflow.Eq_tii005(Infoflow.Low)(gensym@29_1#gensym@29_1);p78;p86> in

  p77_f93

let proof = Proof_by_contra <Infoflow.Eq_tii005(gensym@29_1#gensym@29_1)(Infoflow.Low);proof_fun>
```

Automatic Proof Construction

application	LOC	# of proof obligations	# of Z3 proof rule applications	# of FINE proof rule applications
mini ehr	32	5	131	432
mini iflow	111	20	1047	2912
mini lookout	190	7	494	3942

- Z3 proof terms map to several FINE proof terms
  - n-ary logical connectives converted to binary
  - Our quantifier intro proofs uses many subterms
  - Our monotonicity proofs reason about all subterms
- Opportunities for improvement
  - Simplifying Z3 proofs before translation
  - Handling tautologies and rewrites more generally

DCIL

- Extends the .NET Common Intermediate Language

```
class C <T1::*, T2::A, p:prin, λx:T1.T>
```

Type parameters

DCIL

- Extends the .NET Common Intermediate Language

```
class C <T1::*, T2::A, p:prin, λx:T1.T>
```

Type parameters

Affine type parameters

DCIL

- Extends the .NET Common Intermediate Language

```
class C <T1::*, T2::A, p:prin, λx:T1.T>
```

Type parameters

Affine type parameters

Value parameters (dependent classes)

DCIL

- Extends the .NET Common Intermediate Language

```
class C <T1::*, T2::A, p:prin, λx:T1.T>
```

Type parameters

Affine type parameters

Value parameters (dependent classes)

Type-level function parameters

Translating FINE to DCIL

- Type and data constructors

```
type prin
type cred =
  MkCred :
    p:prin -> cred p
```

```
abstract class prin {}
abstract class Cred<p:prin>{}
class MkCred<p:prin>
      : Cred<p> { prin p; }
```

- Dependent functions

```
login :
  p:prin -> cred p
```

```
abstract class
  DepArr<arg::*, ret::arg => *>
{    (ret x) App (x:arg) {}
}
class login
  : DepArr<prin, λx:prin.cred x>
{    (cred p) App (p:prin) { … }
}
```

Conclusions

- FINE+DCIL tries to bridge the gap between policy specification and enforcement
- Provides a high-level source programming model, while retaining a small TCB for verification of proof-carrying target code
- But, lots left to do …
  - Build more applications and integrate with .NET
  - Optimize proof extraction

Thanks!

Extra Slides

Policy for Patient Records

- Recall the example SecPAL policy:

```
P can read P's record r
D can read P's record r if D is treating P and r.Subject <> HIV
```

- In FINE:

```
assume forall r:record. hasPerm r.patient (CanRead r)
```

Policy for Patient Records

- Recall the example SecPAL policy:

```
P can read P's record r
D can read P's record r if D is treating P and r.Subject <> HIV
```

- In FINE:

```
assume forall r:record. hasPerm r.patient (CanRead r)

prop isTreating :: prin -> prin -> *

assume forall d:prin, r:record.
    (isTreating d r.patient && r.subject <> "HIV") =>
    hasPerm d (CanRead r)
```

Stateful Policy for Patient Records

- `isTreating d p` can change over time
- In FINE, model state by passing around a store

```
assume forall d:prin, r:record, st:state.
    (isTreating d r.patient st && r.subject <> "HIV") =>
    hasPerm d (CanRead r)
```

Policy can quantify over states

Stateful Policy for Patient Records

- `isTreating d p` can change over time
- In FINE, model state by passing around a store

```
assume forall d:prin, r:record, st:state.
    (isTreating d r.patient st && r.subject <> "HIV") =>
    hasPerm d (CanRead r)
```

Propositions can refer to current state

Policy can quantify over states

Stateful Policy for Patient Records

- `isTreating d p` can change over time
- In FINE, model state by passing around a store

```
assume forall d:prin, r:record, st:state.
    (isTreating d r.patient st && r.subject <> "HIV") =>
    hasPerm d (CanRead r)
```

Propositions can refer to current state

Policy can quantify over states

- Program operations can change state of world

```
val visit_doctor:
   p:prin -> cred<p> -> doc:prin ->
     (st:state) ->
       ({st':state | isTreating doc p st'})
```

Stateful Policy for Patient Records

- `isTreating d p` can change over time
- In FINE, model state by passing around a store

```
assume forall d:prin, r:record, st:state.
    (isTreating d r.patient st && r.subject <> “HIV”) =>
    hasPerm d (CanRead r)
```

Propositions can refer to current state

Policy can quantify over states

- Program operations can change state of world

```
val visit_doctor:
   p:prin -> cred<p> -> doc:prin ->
    (st:state * !stateToken<st>) ->
      ({st’:state | isTreating doc p st’} * !stateToken<st’>)
```

Affine types to ensure stale states are not re-used

Equality Propositions

- Need a proposition for equality of terms

```
prop eq :: ∀α. α -> α -> *
```

  - Requires second-order quantificiation over types
  - Would require bigger change to CIL

- Instead, first-order treatment of equality
- Specialized proof constructors for each type `t`

```
type eq_t :: t -> t -> * =
  | Refl_t : a:t -> pf<eq_t a a>
  ...
```

```
% date
Thu Aug  6 22:21:39 PDT 2009
%
% ./fine.exe test/<file>.f9
    --skip_translation
    --extract_proofs
    --proof_stats
```