End-to-end Verification of Security Enforcement is Fine

Nikhil Swamy
Microsoft Research
nswamy@microsoft.com

Juan Chen
Microsoft Research
juanchen@microsoft.com

Ravi Chugh
University of California, San Diego
rchugh@cs.ucsd.edu

Abstract

Proving software free of security bugs is hard. Programming language support to ensure that programs correctly enforce their security policies would help, but, to date, no language has the ability to verify the enforcement of the kinds of policies used in practice—dynamic, stateful policies which address a broad range of concerns including forms of access control and information flow tracking.

This paper makes two main contributions. First, we present FINE, a new source-level security-typed language that, through the use of a simple module system and dependent, refinement, and affine types, can be used to check the enforcement of dynamic security policies applied to real software. Second, we define DCIL, a small extension to the type system of the .NET Common Intermediate Language, and show how to compile FINE in a type-preserving manner to DCIL. Our approach allows FINE programs to run on stock .NET virtual machines and to interface with .NET libraries. Additionally, our type-preserving compiler allows code consumers to download DCIL programs and check them for security while relying on a small trusted computing base. We have proved our source and target languages sound, our compilation type-preserving, and have made a prototype implementation of our compiler and several example programs available.

1. Introduction

Many modern software systems are assembled from components provided by multiple vendors. Whether due to malice or mistakes, third-party components can pose a security risk. To help a user protect her sensitive information from abuse by plugins, a software platform can aim to apply a security policy to control a plugin's behavior. However, the policies used in practice are often complex and simultaneously address various aspects of security including, for example, role-based access control and information flow tracking. Reliably enforcing such policies is difficult, and reports of security vulnerabilities due to incorrect enforcement are common.

Researchers have proposed a number of security-typed languages to ensure that programs correctly enforce their security policies. However, languages like FlowCaml (Simonet 2003), Jif (Chong et al. 2006), Fable (Swamy et al. 2008), Aura (Jia et al. 2008), and F7 (Bengtson et al. 2008) cannot handle many common policies. For example, all these languages assume that authorization policies are stateless, but prevalent security idioms like role- or history-based access control are inherently stateful. In role-based access control, for instance, the privileges granted to principals depend on an ever-changing set of activated roles. Another shortcoming of existing languages is their limited applicability to mobile-code settings and plugin-based applications. Ideally, we would like code consumers to download binaries and check them against their security policies before installing them. However, the compilers of these security-typed languages do not generate verifiable binaries.

1.1 FINE and DCIL

This paper makes two main contributions. The first is FINE, a new programming language that can be used to check that the stateful authorization and information-flow policies applied in practice are correctly enforced. Our main example is an implementation of a reference monitor and a plugin for LOOKOUT, a plugin-based office-utilities client that we have begun to build. LOOKOUT defines an interface that allows a plugin to read email from a user's inbox, send email, make appointments in the calendar and so on. A useful plugin could scan a user's inbox for messages that appear to be a meeting request, automatically make appointments in the calendar, and send email notification to the sender confirming the appointment. To ensure that such a plugin cannot steal sensitive emails from a user's inbox, a user can customize the behavior of LOOKOUT's reference monitor by defining a security policy that combines, say, aspects of information flow tracking with role- and history-based access control. The type system of FINE ensures that plugins always use a reference monitor's API correctly, and in doing so, comply with the user's security policy.

Our representation of stateful policies is based on a general framework for reasoning about dynamic policies due to Dougherty et al. (2006), and is applicable to checking the enforcement of the stateful, Datalog-based policies they explore. As such, our work is relevant beyond the space of plugin-based software. Indeed, one of our case studies includes a model of Continue, a widely-used conference management server, whose stateful security policy has been formalized by Dougherty et al.

The technical contribution of FINE is a new type system that includes dependent types and refinement types—these can be used to express authorization policies by including first-order logical formulas in the types of program expressions. We also include affine types, a facility that allows us to model changes to the state of an authorization policy. The combination of affine and dependent types is subtle and can require tracking uses of affine assumptions in both types and terms. Our formulation shows how to keep the metatheory simple by ensuring that affine variables never appear in types, while still allowing the state of a program to be refined by logical formulas. We also formalize a module system for FINE that provides a powerful information-hiding property. In combination with the other features, the module system allows FINE programs to properly track information flow.

Programming with these advanced typing constructs can place a significant burden on the programmer. For this reason, languages like Fable and Aura position themselves as intermediate languages because verification depends on intricate security proofs too cumbersome for programmers to write down. To alleviate this concern, FINE draws on the experience of languages like F7 and uses Z3 (de Moura and Bjorner 2008), an SMT solver, to automatically discharge proof obligations. However, unlike prior languages, we remove the solver from the TCB by extracting proofs from Z3 as typeable FINE values. Our interface with Z3 is simplified by our

```
1  module AC
2  type prin = U : string → prin | Admin : prin
3  private type cred :: prin → ⋆ = Auth: p:prin → cred p
4  (∗ Authenticating principals and obtaining credentials ∗)
5  val login : p:prin → pw:string → option (cred p)
6  let login p pw = if (check_pwd_db p pw) then Some (Auth p) else None
7  (∗ A proposition and axiom for defining file permissions ∗)
8  type CanWrite :: prin → file → ⋆
9  assume Ax1: forall f:file, CanWrite Admin f
10 (∗ A secure wrapper for a system call ∗)
11 val fwrite: p:prin→ cred p→ {f:file | CanWrite p f}→ string→ unit
12 let fwrite p c f s = Sys.fwrite f s
13 (∗ A utility function to test the policy ∗)
14 val check: p:prin→ cred p→ f:file→ {b:bool | b=true⇒ CanWrite p f}
15 let check p c f = match p with Admin → true | _ → false
16 end
17 open AC
18 let client (p:prin) (c:cred p) (f:file) =
19   if check p c f then fwrite p c f ‘‘Hello’’ else ()
```

Figure 1. Password authentication and access control in FINE

careful treatment of the combination of affine and dependent types. Refinement formulas only involve the standard logical connectives, avoiding the need for an embedding of linear logic in Z3.

Our second main contribution is a type-preserving translation of FINE to DCIL, a new extension to the type system of CIL, the bytecode language of the .NET runtime (ECMA 2006). DCIL augments CIL with type-level functions, classes parametrized by values, and affine types. DCIL programs can be checked purely syntactically, without reliance on an external solver, and all the additional typing constructs of DCIL can be accommodated within the existing specifications of CIL—value parameters can be encoded in fields, affine types using type modifiers, and type-level functions using custom attributes. Our approach makes it possible to run FINE programs on stock .NET virtual machines; to interface with the vast libraries and tool support (such as IDEs) for .NET; to validate the translation performed by our compiler with a small TCB; and, perhaps most importantly, allows us to build plugins for programs like LOOKOUT using FINE, taking advantage of state-of-the-art theorem provers to ease programming, while distributing plugins as DCIL programs that can be type checked for security by end users.

Outline. We begin in Section 2 by describing FINE using several examples. Section 3 formalizes FINE and proves it sound. Section 4 discusses our method of extracting and type-checking proof terms from Z3. Section 5 formalizes DCIL, proves it sound, and proves that our compilation strategy preserves types. Section 6 discusses a prototype implementation of our compiler and example programs, including a brief description of Continue. Section 7 compares related work, and Section 8 concludes. The appendix contains full formalizations and proofs of all theorems discussed in the main body of the paper.

2. FINE, by example

This section presents FINE using several examples. The first example shows how to enforce a simple authentication and access control policy. We build on this example to construct a reference monitor for LOOKOUT, the plugin-based office utilities application described in the Introduction. We show how the reference monitor can be customized by a user-specified policy, and how it can be used to enforce a stateful role- and history-based authorization policy while tracking information flow.

2.1 Simple authentication and access control

Our first example is intended as an introduction to the syntax and typing constructs used in FINE. The policy enforced by module AC in Figure 1 can also be expressed in languages like Fable and Aura. However, even with this simple policy, a key difference is that programmers in Fable and Aura must manually construct explicit security proofs, whereas in FINE, the type checker uses an SMT solver, Z3, to automatically discharge proof obligations. Using an external solver is also a feature of languages like F7 and Sage (Flanagan 2006). However, both these languages include the solver in the TCB. In contrast, our compiler extracts and type checks proof terms from Z3, and removes the solver from the TCB, and enabling a type-preserving translation—a key to our goal of providing lightweight checking of plugin binaries at the client.

In order to specify and enforce security policies, FINE programmers define modules that mediate access to sensitive resources. The module AC in Figure 1 is a reference monitor for a file system. The policy enforced by a FINE module has two components: the axioms introduced through the use of the **assume** construct (e.g., Ax1 at line 9), and the types given to values exposed in the module's interface (e.g., the type of fwrite at line 11). A security review of AC must confirm that the assumptions and the types ascribed to values adequately capture the intent of a high-level policy. Importantly, client code need not be examined at all—typing ensures that clients comply with the reference monitor's security policy.

The AC module implements a password-based authentication mechanism combined with a permission-based access control policy. AC defines prin (line 2), a standard variant type that represents principal names as either a string for the user's name, or the distinguished constant Admin. The type cred (line 3) is a dependent-type constructor that is given the *kind* prin → ⋆, e.g., (cred Admin) is a legal type of kind ⋆ (the kind of normal types, distinguished from the kind of affine types, introduced in the next section) and represents a credential for the Admin user. By declaring it **private**, AC indicates that its clients cannot directly use the data constructor Auth. Instead, the only way a client module can obtain a credential is through the use of the login function. The login function (lines 5-6) is given a dependent function type where the first argument p is the name of principal, the second argument pw is a password string, and, if the password check succeeds, login returns a credential for the user p. By indexing the cred with the name of the principal which it authenticates, we can statically detect common security errors, such as those that arise due to confused deputies, e.g., a client cannot use login to obtain a credential for U ‘‘Alice’’ and later pass it off as a credential for Admin.

Line 8 defines a dependent-type constructor CanWrite that is used to describe authorization permissions—AC interprets the type CanWrite p f as the proposition that the principal p can write to the file f. At line 9, AC defines a single policy assumption, Ax1, that states that the principal Admin can write to any file. A client program can use axioms like Ax1 to produce evidence of the propositions required to call functions like fwrite, which wraps a call to Sys.fwrite. Client programs are assumed to not have direct access to sensitive system calls like Sys.fwrite. The first two arguments of fwrite require the caller to present a credential for the principal p. The third argument is a file handle with a refined type—the type {f:file | CanWrite p f} is inhabited by any value f of type file for which the proposition CanWrite p f can be proved. The final argument to fwrite is the string to be written to the file.

AC also provides a utility function check that clients can use to query the policy. For this simple policy, the only principal that can write to a file is Admin. To type this function, our type checker utilizes information about runtime tests to refine types. For example, in the first branch, to prove that true can be given the type {b:bool | b=true⇒ CanWrite p f}, we pass the assumption p = Admin (derived from the result of the pattern match) and the axiom Ax1 to our solver, Z3, which decides that CanWrite p f is valid.

At lines 18-19 we show a client of the AC module, a function with three arguments: a principal name p, a credential c for p, and

```
1  module LookoutRM
2  open AC
3  private type email = {sender:prin; contents:string}
4  private type appt = {who:prin; starttime:date; endtime:date; note:string}
5  val mkEmail : prin → string → email
6  val sender : e:email → {p:prin | p=e.sender}
7  val mkAppt : prin → date → date → string → appt
8  (∗ Constructs for information flow tracking ∗)
9  type prov = E:email→ prov | A:appt→ prov | J:prov→ prov→ prov
10 private type tracked::⋆ → prov→ ⋆ = Tag:α → p:prov→ tracked α p
11 val fmap : (α → β ) → p:prov → tracked α p → tracked β p
12 val popt : p:prov → tracked (option α) p → option (tracked α p)
13 (∗ Constructs for a stateful authorization policy ∗)
14 type role = User : role | Friend : role | Plugin : role
15 type att = Role : prin → role → att
16          | HasRepliedTo : prin → email → att
17 type st = list att
18 type action = ReadEmail : email → action
19             | ReplyTo : email → p:prov → tracked email p → action
20             | MkAppt : p:prov → tracked appt p → action
21 type perm = Permit : prin → action → perm
22 type In :: att → st → ⋆
23 type Derivable :: st → perm → ⋆
24 type dst<p:perm> = {s:st | Derivable s p}
25 (∗ An affine type to assert the validity of the authorization state ∗)
26 private type StateIs::st → A = Sign:s:st → StateIs s
27 (∗ An API for plugins ∗)
28 val readEmail : p:prin → cred p → e:email →
29                 s:dst<Permit p (ReadEmail e)> → StateIs s →
30                 (tracked string (E e) ∗ StateIs s)
31 val mkAppt : p:prin → cred p → q:prov → a:tracked appt q →
32              s:dst<Permit p (MkAppt q a)> → StateIs s → StateIs s
33 val replyTo : p:prin → cred p →
34              orig:email → q:prov → reply:tracked email q →
35              s:dst<Permit p (ReplyTo orig q reply)> → StateIs s →
36              (s1:{x:st | In (HasRepliedTo p orig) x} ∗ StateIs s1)
37 val installPlugin: u:prin → cred u → p:prin →
38                   s:{x:st | In (Role u User) s} → StateIs s →
39                   (s1:{x:st | In (Role p Plugin) x} ∗ StateIs s1)
```

Figure 2. A fragment of a reference monitor for LOOKOUT

a file f. In order to call fwrite, client must prove the proposition CanWrite p f. It does so by calling check and calls fwrite if the check is successful. Once again, the type checker uses the result of the runtime test to conclude that CanWrite p f is true in the **then**-branch, and the call to fwrite type checks. A call to fwrite without the appropriate test (e.g., in the **else**-branch) results in a type error.

2.2 A reference monitor for LOOKOUT

In this section, we present a more substantial example of programming in FINE, where we model a fragment of a stateful authorization and information flow policy for use with LOOKOUT. We begin by showing a reference monitor LookoutRM, which exposes an API for plugins to read and reply to emails and make appointments in a calendar. In Section 2.3, we show how a user can specify policy rules to restrict the way in which a plugin can use LookoutRM's API. Section 2.3 also shows the code for a plugin. Our approach allows a user to download .NET assemblies for a plugin, type check it against a policy using a lightweight syntactic checker, and only install it if the check succeeds.

Security objectives. LookoutRM provides a way to track information flow. A user can use information flow tracking to ensure, for example, that emails sent by a plugin never disclose information not meant for the recipient. Additionally, LookoutRM models a stateful role and history-based authorization policy. This will allow a user to organize her contacts into roles, granting privileges to some principals but not others, and will allow the user to change role memberships dynamically. The state of the policy will also record events like the sending of emails. A user can define a policy over these events to, for example, ensure that plugins never spam a user's contacts by responding to emails repeatedly.

Technical aspects of our encoding. Expressing and enforcing these security objectives exercises various aspects of FINE's type system. Information flow policies are specified using dependent types, where the type of a secure object is indexed by a value indicating its provenance, i.e., its origin. Stateful policies are specified by refining the type of the authorization state using logical formulas. For instance, a refinement formula could require the authorization state to record that a principal is in a specific role before a permission is granted. Finally, changes to the state are modeled using *affine types* (Walker 2004). Affine types are closely related to linear types, with the distinction that affine variables can be used *at most once*. Our approach keeps affine types separate from refinement formulas, simplifying both programming and the extraction of proof terms from Z3. We discuss the code in Figure 2 sequentially, considering each of these three technical aspects in turn.

2.2.1 Information flow tracking

LookoutRM allows plugins to read emails and aims to track information flow for any data that is derived from the contents of an email. At lines 3-4 of Figure 2, LookoutRM defines the types email and appt, records that represent emails and appointments, respectively. To ensure that clients cannot directly inspect the contents of an email we make email a private type. LookoutRM includes a function mkEmail to allow plugins to construct emails, and sender, to allow a plugin to inspect the sender of an email. But, in order to read the contents of an email, a plugin will have to use readEmail (discussed in detail in Section 2.2.4).

Information flow tracking in LookoutRM is based on a pattern developed in the context of the Fable calculus. In this scheme, information flow policies are specified and enforced by tagging sensitive data with *security labels* that record provenance. At line 9, LookoutRM defines the type prov (values of this type will be used as security labels) and at line 10, the dependent-type constructor tracked provides a way of associating a label with some data. For example, tracked string (E x) will be the type of a string that originated from the email x. Importantly, tracked is defined as a private type. Client programs can only manipulate tracked values using functions that appear in the interface of LookoutRM, e.g., fmap is a functor that allows functions to be lifted into the tracked type. Several other functions can also be provided to allow client programs to work with the tracked datatype. Prior work on Fable showed that encodings of this style can be proved to correctly enforce security properties like noninterference.

2.2.2 Refined state for stateful authorization

The model of stateful authorization implemented by LookoutRM is based on a framework due to Dougherty et al. (2006) for reasoning about the correctness of Datalog-style dynamic policies. In this model, policies are specified as inference rules that derive permissions from a set of basic authorization attributes. For example, the attributes may include assertions about a principal's role membership, and the policy may include inference rules that grant permissions to principals in certain roles. Over time, whether due to a program's actions or due to external events, the set of authorization attributes can change. For example, in order to access a resource, a principal may alter the state of the authorization policy by activating a role. In this state, the policy may grant a specific privilege to the principal. A subsequent role deactivation causes the privilege to be revoked. Dougherty et al. show that many common policies can be captured by this model in a manner conducive to reasoning about policy correctness. Section 6 discusses an implementation of this model in the Continue conference management server.

The set of basic authorization attributes in LookoutRM is represented by the type st (lines 14-17). Attributes include values like Role (U ''Alice'') Friend to represent a role-activation for a principal, or values like HasRepliedTo p e to record an event that a principal p has sent an email in response to e. Permissions (the relations derived using policy rules from the basic authorization attributes) are represented using the type perm. For example, Permit (U ''Plugin'') (ReadEmail e), represents a permission that a user may grant to a plugin. Line 22 shows a type In, a proposition about list membership, e.g., In a s is a proposition that states that a is a member of the list s. We elide standard assumptions that axiomatize list membership. The proposition Derivable s p (line 23) is used to assert that a permission p is derivable from the collection of authorization attributes s. The type abbreviation dst<p> refines the state type st to those states in which the permission p is derivable.

2.2.3 Affine types for state evolution

The type constructor StateIs at line 26 addresses two concerns. A value of type StateIs s represents an assertion that s contains the current state of authorization facts. LookoutRM uses this assertion to ensure the integrity of its authorization facts—StateIs is declared private, so, untrusted clients cannot use the Sign constructor to forge StateIs assertions. Moreover, since the authorization state can change over time, FINE's type system provides a way to revoke StateIs assertions about stale states. For example, after a principal p has responded to an email e, we may add the fact HasRepliedTo p e to the set of authorization facts s. At that point, we would like to revoke the assertion StateIs s, and assert StateIs ((HasRepliedTo p e)::s) instead. o

Types in FINE are classified into two basic kinds, $\star$, the kind of normal types, and A, the kind of affine types. By declaring StateIs :: st $\rightarrow$ A we indicate that StateIs constructs an affine type from an argument of type st. When the state of the authorization policy changes from s to r, LookoutRM constructs a value Sign r to assert StateIs r, while destructing a StateIs s value to ensure that the assertion about the stale state s can never be used again.

2.2.4 A secure API for plugins

Lines 28-39 define the API that LookoutRM exposes to plugins. Each function requires the caller p to authenticate itself with a credential cred p. Using the refined state type dst<p>, the API ensures that each function is only called in states s where p has the necessary privilege. For example, in order to read the contents of an email e, the readEmail function requires ReadEmail p e to be derivable in the state s. To ensure that information flows are tracked on data derived from an email, readEmail returns the contents of e as a string tagged with its provenance, i.e., the label E e. To indicate that the authorization state s has not changed, readEmail also returns a value of type StateIs s. The mkAppt function allows p to make an appointment a only in states s where p has the MkAppt permission. The type of a indicates that its provenance is q, and, like readEmail, mkAppt leaves the authorization state unchanged. As we will see shortly, a user can grant a plugin permission to make an appointment a depending on a's provenance.

The function replyTo allows a plugin p to send a reply with provenance q to an email orig when the ReplyTo orig q reply has been granted to p. Unlike the other functions, replyTo modifies the authorization state to record a HasRepliedTo p orig event. The return type of replyTo, a dependent pair consisting of a new list of authorization attributes s1, and an assertion of type StateIs s1 to indicate that s1 is the current authorization state. Finally, we show a function installPlugin that allows a user u to register a plugin p.

2.3 A LOOKOUT user's policy and a plugin

Figure 3 shows a module UserPolicy that configures the behavior of the LookoutRM reference monitor with several user-provided

```
1  module UserPolicy : LookoutRM
2  let init = let a = [Role (U "Alice") Friend; ... ] in (a, Sign a)
3  assume U1: forall (p:prin) (e:email) (s:st).
4     In (Role p Plugin) s && In (Role e.sender Friend) s ⇒
5     Derivable s (Permit p (ReadEmail e))
6  assume U2: forall (p:prin) (e:email) (a:tracked appt (E e)) (s:st).
7     In (Role p Plugin) s && In (Role e.sender Friend) s ⇒
8     Derivable s (Permit p (MkAppt (E e) a))
9  assume U3: forall (p:prin) (e:email) (reply:tracked email (E e)) (s:st).
10    In (Role p Plugin) s && not (In (HasRepliedTo p e) s) ⇒
11    Derivable s (Permit p (ReplyTo e (E e) reply))
12 end
13 open LookoutRM
14 (* Utility functions for checking authorization attributes *)
15 val checkAtt: s:st → r:attr → {b:bool | (b=true ⇔ In r s)}
16 let rec checkAtt s r = match s with
17    | [] → false
18    | a::tl → if r=a then true else check tl r
19 (* Custom plugin logic *)
20 val detectAppt: prin → string → option appt
21 val mkNotification: appt → email
22 (* Type abbreviation for the current set of authorization facts s *)
23 type state = (s:st * StateIs s)
24 val processEmail: p:prin → cred p → email → state → state
25 let processEmail p c em (s, tok) =
26    let c1 = checkAtt s (Role p Plugin) in
27    let c2 = checkAtt s (Role (sender em) Friend) in
28    let c3 = checkAtt s (HasRepliedTo p em) in
29    if c1 && c2 && not c3 then
30      let (pstr, tok) = read_email p c em s tok in
31      let opt_appt = fmap (detectAppt (sender em)) (E em) pstr in
32       match popt (E em) opt_appt with
33         | None → (a, tok) (* no appointment extracted; do nothing *)
34         | Some appt →
35             let tok = mkAppt p c (E em) appt s tok in
36             let reply = fmap mkNotification (E em) appt in
37               replyTo p c em (E em) reply s tok
38    else (s,tok) (* can't read email, or already sent notification *)
```

Figure 3. A user's policy and fragment of plugin code

policy assumptions. At line 2, we show init, the initial collection of authorization attributes. The user includes facts like the roles of friends in a list a, and, using the data constructor Sign, attests that a is the authorization state. The Sign data constructor requires the privilege of the LookoutRM module—FINE's module system allows this privilege to be granted to UserPolicy using the notation **module** UserPolicy : LookoutRM.

The assumptions U1-U3 show how permissions can be derived from attributes. Assumption U1 allows a plugin to only read emails from friends. U2 allows a plugin to make an appointment a, only if the provenance of a is an email e that was sent by a friend. U3 allows a plugin to reply to an email e only if a reply has not already been sent. Moreover, the reply should only contain information derived from the original email, ensuring that plugins do not leak emails from one contact to another. Of course, more elaborate information flow constraints could also be specified. Section 6 briefly discusses a more traditional, lattice-based information flow policy that we have implemented.

The utility function checkAtt on lines 15-18 is a standard tail-recursive membership test on a list, and allows the authorization state to be queried. Type-checking checkAtt requires using standard axioms about In, e.g., **assume forall** (a:att). In a [a], which we have omitted for simplicity.

2.3.1 An example plugin

The rest of Figure 3 shows fragments from a plugin program. The processEmail function is meant to extract an appointment from an email, update the calendar with the appointment, and send an automated reply. It relies on two functions detectAppt and mkNotification, that implement some plugin-specific logic. The type of processEmail shows its arguments to be a credential c of type cred p, the email em that is to be processed, and the current authorization state (s, tok):state. This is a pair consisting of the set of authorization attributes s, and a token, tok:StateIs s, asserting the integrity and validity of s. Lines 26-28 show several checks on the authorization state to ensure that p has the privilege to read em and to send a response. If the authorization check fails, the plugin does nothing and returns the state unmodified. Otherwise, at line 17, it reads em and obtains pstr:tracked string (E em). It uses fmap and popt to try to extract an appointment from the email in a manner that tracks provenance. If an appointment was found, it makes an appointment and sends a reply. Several subtle points in processEmail reveal features of FINE—we discuss these next.

Non-affine state simplifies programming. Programming with affine types can sometimes be difficult, since affine variables can never be used more than once. Our approach of using an affine assertion StateIs s to track the current authorization state minimizes the difficulty. Importantly, the collection of authorization facts s is itself not affine and can be freely used several times, e.g., s is used in several calls to checkAtt. Non-affine state also enables writing functions like checkAtt, which, if s was affine, would destroy the state of the program. Only the affine token, tok:StateIs s, must be used with care, to ensure that it is not duplicated.

Non-affine refinement formulas simplify automated proofs. Even ignoring the inability of prior languages to handle stateful policies, the proof terms required for a program of this style in a language like Fable or Aura would be extremely unwieldy. The FINE type checker can use Z3 to synthesize proof terms for the proof obligations in this example. By ensuring that refinement formulas always apply to non-affine values, our proof system is kept tractable. A naïve combination of dependent and affine types would allow refinements to apply to affine values, necessitating an embedding of linear logic in Z3. Our approach avoids this complication, while retaining the ability to refine the changing state of a program with logical formulas.

Affine types ensure purity. When enforcing information flow policies, implicit flows due to side effects can be a concern. For example, fmap reveals the contents of an email as a string (rather than a tracked string p) to detectAppt. One may be worried that detectApp could subvert the information flow policy by sending the string in an email (a side effect). Our type system guarantees that detectAppt is a pure function which cannot cause side effects by calling functions like replyTo, or mkAppt. To see why, observe that in order to call replyTo, a caller must pass an affine StateIs s token as an argument. These tokens serve as capabilities (Walker et al. 2000) that permit the caller to cause side effects, such as sending emails. The types of detectAppt and mkNotification ensure that these values do not have access to any such capabilities—capabilities are affine and expressions that capture affine values must themselves be affine.

3. Formalizing FINE

This section presents a core formalism for FINE based on a polymorphic lambda calculus with dependent, affine and refinement types. We also model the module system of FINE using syntactic type abstraction, a technique developed by Grossman et al. (2000). We prove our type system sound, and present a general-purpose security result for the source language, namely that the module system correcly establishes an information-hiding property. On a first reading, a reader comfortable with the typing constructs used in FINE may wish to skip ahead to Section 3.4, and beyond, for a description of our type-preserving compilation technique.

principals	p, q, r	$::=$	$\mathsf{p} \mid \top \mid \bot$
terms	e	$::=$	$x \mid D \mid \lambda x{:}\tau.e \mid \Lambda\alpha{::}\kappa.e$
		$\mid$	$\text{fix } f{:}\tau.e \mid e_1\ e_2 \mid e\ \tau \mid \langle e\rangle_p$
		$\mid$	$\text{match } v_p \text{ with } D\ \vec{\tau}\ \vec{x} \rightarrow e_1 \text{ else } e_2$
types	τ, ϕ	$::=$	$\alpha \mid x{:}\tau_1 \rightarrow \tau_2 \mid \forall\alpha{::}\kappa.\tau$
		$\mid$	$T \mid \tau_1\ \tau_2 \mid \tau\ e \mid \{x{:}\tau \mid \phi\} \mid\ !\tau$
kinds	κ	$::=$	$\star \mid \mathtt{A} \mid \star \rightarrow \kappa \mid \mathtt{A} \rightarrow \kappa \mid \tau \rightarrow \kappa$
signature	S	$::=$	$T{::}\kappa \mid D{:}(p, \tau) \mid p \sqsubseteq q \mid S, S \mid \cdot$
type env.	Γ	$::=$	$\alpha{::}\kappa \mid x{:}(p, \tau) \mid v_p \doteq v'_p \mid \Gamma, \Gamma' \mid \cdot$
pre p-values	u_p	$::=$	$x \mid D\ \vec{\tau}\ \vec{v}$
p-values	v_p	$::=$	$u_p \mid \lambda x{:}t.e \mid \Lambda\alpha{::}\kappa.e \mid \langle u_q\rangle_q$

Figure 4. Syntax of FINE

3.1 Syntax

Figure 4 defines the syntax of FINE. Source terms are annotated with the names of principals, ranged over by the metavariables p, q, r. Principals in our formalization correspond to module names, and expressions granted the privilege of p are allowed to view the types defined in module p concretely; other principals must view p's types abstractly. A principal constant is denoted p, and we include two distinguished principals: $\top$ includes the privileges of all other principals, and $\bot$ has no privileges.

The term language is standard for a polymorphic lambda calculus with data constructors D and a pattern matching construct. The form $\langle e\rangle_p$ represents an expression e that has been granted p-privilege. Types τ include dependent function types (pi types) $x{:}\tau \rightarrow \tau'$, where x names the formal parameter and is bound in τ'. Polymorphic types $\forall\alpha{::}\kappa.\tau$ decorate the abstracted type variable α with its kind κ. We include type constructors T, which can be applied to other types using $\tau_1\ \tau_2$ or terms using $\tau\ e$. Refinement types are written $\{x{:}\tau \mid \phi\}$, where ϕ is a type in which x is bound. An affine qualifier can be attached to any type using $!\tau$. Types are partitioned into normal kinds $\star$ and affine kinds $\mathtt{A}$. Type constructors can construct types of kind κ from normal types ($\star \rightarrow \kappa$), affine types ($\mathtt{A} \rightarrow \kappa$), or terms of type τ ($\tau \rightarrow \kappa$).

FINE programs are parameterized by a signature S, a finite map which, using $T{::}\kappa$, ascribes a kind to a type constructor T. The notation $D{:}(p, \tau)$ associates a principal name p and type τ with a data constructor. This gives D the type τ and limits its use to programs with p-privilege. The signature also records relations between principals $p \sqsubseteq q$, to indicate that q includes the privileges of p. For example, the Sign constuctor from Figure 2, is represented in this notation as Sign:(LookoutRM, a:st $\rightarrow$ StateIs a), and indicates that it is a data constructor which requires the privilege of the LookoutRM module. The notation **module** UserPolicy : LookoutRM from Figure 3, is represented as the relation LookoutRM $\sqsubseteq$ UserPolicy, which grants the UserPolicy module the privilege to use the Sign constructor. Axioms introduced via the **assume** construct are represented as data constructors (cf. Section 4).

The typing environment Γ records the kind of type variables. Just as with data constructors in the signature, variables x are associated with both their type τ and a principal name p. The assumption $v_p \doteq v'_p$ records the result of pattern matching tests and is used to refine types.

Values in FINE are partitioned into families corresponding to principals. A pre-value for code with p-privilege is either a variable, or a fully-applied data constructor. Values for p are either its pre-values, abstractions, or pre-values u_q for some other principal q, delimited within angle brackets to denote that u_q carries q-privilege. Following Grossman et al., we give a dynamic semantics that tracks the privilege associated with an expression using these bracket delimiters. This allows us to prove, in Section 3.4, that programs without p-privilege view p-values abstractly.

For simplicity, our formalization omits dependent pairs $(x{:}\tau * \tau')$ although we use these in our examples. Pairs can be derived using a standard higher-order encodings, e.g., using terms of type $\forall\alpha{::}\star.f{:}(x{:}\tau \rightarrow y{:}\tau' \rightarrow \alpha) \rightarrow \alpha$. Of course, our implementation provides primitive support for pairs (and record types), rather than requiring programmers to use this encoding. Our translation to DCIL is simplified by allowing only values to be discriminated by pattern matching (a source program can always be put in this form).

3.2 Static semantics

Two principles guide the static semantics FINE, defined in Figure 5. First, we aim for our target language DCIL to be a minimal extension of the type system of CIL. As such, we omit useful features like abstraction over types with higher-order kinds, because they require changes to the core CIL type system. Second, we limit the interaction between affine and dependent types to keep proof checking tractable. In particular, we forbid refinement formulas from using affine assumptions, thereby avoiding the need for an embedding of linear logic in our prover. The examples in the previous section show how this restriction can be turned to our advantage by always representing the state of a program by refining a non-affine value.

The first judgment $S \vdash_i \kappa$ defines a well-formedness relation on kinds. Intuitively, this judgment establishes two properties. First, types constructed from affine types must themselves be affine—this is standard (Walker 2004). Without this restriction, an affine value can be stored in non-affine value and be used more than once. To enforce this property, we index the judgment using $i ::= \cdot \mid 1$, and when checking a kind $\mathtt{A} \rightarrow \kappa$, we require κ to finally produce an A-kinded type. The second restriction, enforced by (WF-Dep), ensures that only non-affine values appear in a dependent type.

The judgment $S; \Gamma \vdash \tau :: \kappa$ states that τ can be given kind κ. Types that are inhabited by terms are always given either kind $\star$ or A, and in (K-Fun), we require that the type τ_1 of a function's parameter always have kind $\star$ or A. Additionally, we require functions which take affine arguments to produce affine results. These two constraints are captured using an auxiliary relation on kinds, $\kappa \leq \kappa'$. In (K-Uni) we allow abstraction only over $\star$ and A-kinded types. (K-Afn) rules out "doubly-affine" types ($!!\tau$). (K-Ref) requires refinement formulas ϕ to be non-affine.

The rule that checks the well-formedness of dependent types, (K-Dep), has two subtle elements. First, we restrict type-level terms to be values, e.g., Eq (+ 1 2) 3 is not a well-formed type, even with Eq::int$\rightarrow$ int$\rightarrow \star$. This simplifies the metatheory while limiting expressiveness—languages like Aura and F7 impose a similar restriction. In contrast, Fable permits types to be indexed by arbitrary expressions and, at the cost of decidability of type checking, can perform type-level computation to equate types. With this facility, Fable can statically check that certain kinds of information flow policies are properly enforced. Such policies in FINE would require dynamic checks. The second premise of (K-Dep) makes use of the typing judgment—we discuss it in detail below.

The typing judgment is written $S; \Gamma; X \vdash_p e : \tau$, and states that an expression e, when typed with the privilege of principal p in an environment Γ and signature S, can be given the type τ. The set X records a subset of the variable bindings in Γ, and each element of X represents a capability to use an assumption in Γ.

The rule (T-D) requires data constructors declared to be usable only by code with p-privilege to be used in a context with that privilege. In the second premise of (T-Match), we type check a pattern $D\,\vec{\tau}\,\vec{x}$ to ensure that data constructors are also destructed in a context with the appropriate privilege. To translate FINE to DCIL, we include a side-condition that requires data constructors to be fully applied—we elide this for brevity.

In (T-X) we type a non-affine variable x by looking up its type in the environment. (T-XA) allows an affine variable to be used only when a capability for its use appears in X. Unlike in linear typing, affine assumptions need not always be used. (T-Drop) allows an arbitrary number of assumptions X' to be forgotten, and for e to be checked with a privilege q that is not greater than privilege p that it has been granted. An expression is granted privilege by enclosing it in angle brackets, as shown in (T-Bracket).

Returning to the second premise of (K-Dep), we check a type-level term v_p with the privilege of $\top$. The intuition is that type-level terms have no operational significance and, as such, cannot violate information-hiding. We also check v_p in (K-Dep) with an empty set of capabilities X. According to (WF-Dep), no well-formed type constructor can be applied to an affine value, so a type-level term like v_p never uses an affine assumption.

In (T-Fun), we check that the type of the formal parameter is well-formed, and type check the body in an extended context. We record the privilege p of the program point at which the variable x was introduced to ensure that x is not destructed in unprivileged code in the function-body e. In the conclusion of (T-Fun), we use the auxiliary function $Q(X, \tau)$, which attaches an affine qualifier to τ if the function captures any affine assumptions from its environment. (T-Uni) is similar. In (T-Fix), we require fixed variables f to be given a non-affine types, and for the recursive expression to not capture any affine assumptions.

When typing an application $e_1\ e_2$ in (T-App), we allow e_1 to be a possibly affine function type—the shorthand $?\tau$ captures this, and we use the same notation in (T-TApp). In (T-App) we split the affine assumptions among the sub-terms, and, in the third premise, require the well-formedness of $\tau_2[e_2/x]$—this ensures that non-values never appear in types as the result of an application.

In (T-Match), we split the affine assumptions between v_p and the branches. In the second premise, we type check the pattern and derive bindings for each pattern-bound variable x_i. Constructed types in FINE are a form of generalized algebraic datatype (Xi et al. 2003). For simplicity, we do not induce equalities among types as a result of a pattern match. We do, however, record equality assumptions among values that appear in the type τ' of the discriminated expression (if any) and the pattern bound variables. These are shown as the $x_i \doteq v_i$ assumptions in the second premise. The true-branch e_1 is checked with an additional assumption that records the result of the successful pattern match. To illustrate using an example from Figure 2, if the discriminated expression v_p has type $\tau' =$ tracked string (E mail), and the pattern is Tag string $x\ y$, we include the assumptions $x{:}(p, \mathsf{string})$, $y{:}(p, \mathsf{prov})$, and $y \doteq$ (E mail) when typing the pattern in the second premise. When typing the true branch, we also record $v_p \doteq$ Tag string $x\ y$ in Γ.

We include a transitive subtyping relation $S; \Gamma \vdash \tau <: \tau'$, which does not include any structural rules, e.g., contra- and co-variant subtyping in function types. The type system of CIL uses nominal subtyping, and structural rules of this form are not easily translated. Coercions can be used to represent a richer subtyping relation, if necessary (Swamy et al. 2009). The rule (S-UnRef) treats a refined type $\{x{:}\tau \mid \phi\}$ as a subtype of the underlying type τ. (S-Ref) allows a type τ to be promoted to a refined type $\{x{:}\tau' \mid \phi(x)\}$ when τ is a subtype of τ', and when a proof of the formula $\phi(x)$ can be constructed in context Γ extended with a binding for x. (S-Ref) shows the proof term generated non-deterministically as a value v_p. Proof terms are typed with $\bot$-privilege and so can only use the public data constructors of every module in scope. For each variable y bound to a refined type $\{x{:}\tau_1 \mid \phi_1(x)\}$ in the environment, we let $\hat{y}$ denote a proof of the formula $\phi_1(x)$. The premise $\hat{y} \in FV(v_p)$ indicates that v_p makes use of other proof terms $\hat{y}$ from the context. In Section 4, we discuss how these proof terms are synthesized using an external prover (Z3) and type checked in FINE. Finally, subtyping includes an equivalence relation on types $S; \Gamma \vdash \tau \cong \tau'$. The

$S \vdash_i k \quad$ where $\quad i ::= \cdot \mid 1$ **Well-formedness of kinds**

$$\frac{}{S \vdash \star}\ \text{(WF-}\star\text{)} \qquad \frac{}{S \vdash_i \mathtt{A}}\ \text{(WF-A)} \qquad \frac{S \vdash_i \kappa}{S \vdash_i \star \to \kappa}\ \text{(WF-TFun)} \qquad \frac{S \vdash_1 \kappa}{S \vdash_i \mathtt{A} \to \kappa}\ \text{(WF-TFunA)} \qquad \frac{S; \cdot \vdash \tau :: \star \quad S \vdash_i \kappa}{S \vdash_i \tau \to \kappa}\ \text{(WF-Dep)}$$

$S;\Gamma \vdash \tau :: \kappa \quad$ where $\quad \star \leq \star,\ \mathtt{A} \leq \mathtt{A},\ \star \leq \mathtt{A}$ **Kinding of types**

$$\frac{}{S;\Gamma \vdash \alpha :: \Gamma(\alpha)}\ \text{(K-Var)} \qquad \frac{}{S;\Gamma \vdash T :: S(T)}\ \text{(K-TC)} \qquad \frac{S;\Gamma \vdash \tau :: \star}{S;\Gamma \vdash !\tau :: \mathtt{A}}\ \text{(K-Afn)} \qquad \frac{S;\Gamma, \alpha{:}\kappa \vdash \tau :: \kappa' \quad \kappa, \kappa' \in \{\star, \mathtt{A}\}}{S;\Gamma \vdash \forall \alpha{::}\kappa.\tau :: \star}\ \text{(K-Uni)}$$

$$\frac{\begin{array}{c} S;\Gamma \vdash \tau_1 :: \kappa \qquad \kappa \leq \kappa' \\ S;\Gamma, x{:}(p,\tau_1) \vdash \tau_2 :: \kappa' \end{array}}{S;\Gamma \vdash x{:}\tau_1 \to \tau_2 :: \star}\ \text{(K-Fun)} \qquad \frac{\begin{array}{c} S;\Gamma \vdash \tau_1 :: \kappa' \to \kappa \\ S;\Gamma \vdash \tau_2 :: \kappa' \end{array}}{S;\Gamma \vdash \tau_1\ \tau_2 :: \kappa}\ \text{(K-App)} \qquad \frac{\begin{array}{c} S;\Gamma \vdash \tau_1 :: \tau \to \kappa \\ S;\Gamma;\cdot \vdash_\top v_p : \tau \end{array}}{S;\Gamma \vdash \tau_1\ v_p :: \kappa}\ \text{(K-Dep)} \qquad \frac{\begin{array}{c} S;\Gamma \vdash \tau :: \star \\ S;\Gamma, x{:}(p,\tau) \vdash \phi :: \star \end{array}}{S;\Gamma \vdash \{x{:}\tau \mid \phi\} :: \star}\ \text{(K-Ref)}$$

$S;\Gamma;X \vdash_p e : \tau \quad$ where $X ::= \cdot \mid x, X; \quad Q(X,\tau) = !\tau, \quad Q(\cdot,\tau) = \tau; \quad$ and $?\tau$ denotes τ or $!\tau$ **Expression typing**

$$\frac{S(D) = (p,t)}{S;\Gamma;\cdot \vdash_p D : \tau}\ \text{(T-D)} \qquad \frac{\Gamma(x) = (p,\tau) \quad S;\Gamma \vdash \tau :: \star}{S;\Gamma;\cdot \vdash_p x : \tau}\ \text{(T-X)} \qquad \frac{\Gamma(x) = (p,\tau)}{S;\Gamma;x \vdash_p x : \tau}\ \text{(T-XA)} \qquad \frac{S;\Gamma;X \vdash_q e : \tau \quad q \sqsubseteq p \in S}{S;\Gamma;X,X' \vdash_p e : \tau}\ \text{(T-Drop)}$$

$$\frac{\begin{array}{c} S;\Gamma \vdash \tau_1 :: \kappa \qquad \kappa \in \{\star, \mathtt{A}\} \\ S;\Gamma, x{:}(p,\tau_1); X, x \vdash_p e : \tau_2 \end{array}}{S;\Gamma;X \vdash_p \lambda x{:}\tau_1.e : Q(X, x{:}\tau_1 \to \tau_2)}\ \text{(T-Fun)} \qquad \frac{\begin{array}{c} \kappa \in \{\star, \mathtt{A}\} \\ S;\Gamma, \alpha{::}\kappa; X \vdash_p e : \tau' \end{array}}{S;\Gamma;X \vdash_p \Lambda\alpha{::}\kappa.e : Q(X, \forall\alpha{::}\kappa.\tau')}\ \text{(T-Uni)} \qquad \frac{\begin{array}{c} S;\Gamma \vdash \tau :: \star \qquad \text{unrefined}(\tau) \\ S;\Gamma, f{:}(p,t);\cdot \vdash_p v_p : \tau \end{array}}{S;\Gamma;\cdot \vdash_p \text{fix } f{:}\tau.v_p : \tau}\ \text{(T-Fix)}$$

$$\frac{\begin{array}{c} S;\Gamma;X \vdash_p e_1 : ?x{:}\tau_1 \to \tau_2 \\ S;\Gamma;X' \vdash_p e_2 : \tau_1 \qquad S;\Gamma \vdash \tau_2[e_2/x] :: \kappa \end{array}}{S;\Gamma;X,X' \vdash_p e_1\ e_2 : \tau_2[e_2/x]}\ \text{(T-App)} \qquad \frac{\begin{array}{c} S;\Gamma;X \vdash_p e : ?\forall\alpha{::}\kappa.\tau \\ S;\Gamma \vdash \tau' :: \kappa \end{array}}{S;\Gamma;X \vdash_p e\ \tau' : \tau[\tau'/\alpha]}\ \text{(T-TApp)} \qquad \frac{S;\Gamma;X \vdash_q e : \tau}{S;\Gamma;X \vdash_p \langle e \rangle_q : \tau}\ \text{(T-Bracket)}$$

$$\frac{\begin{array}{c} S;\Gamma;X \vdash_p v_p : \tau' \qquad S;\Gamma, x_i{:}(p,\tau_i), x_i \doteq v_i; \vec{x} \vdash_p D\ \vec{\tau}\ \vec{x} : \tau' \\ S;\Gamma, x_i{:}(p,\tau_i), x_i \doteq v_i, v_p \doteq D\ \vec{\tau}\ \vec{x}; X', \vec{x} \vdash_p e_1 : \tau \qquad S;\Gamma;X' \vdash_p e_2 : \tau \end{array}}{S;\Gamma;X,X' \vdash_p \text{match } v_p \text{ with } D\ \vec{\tau}\ \vec{x} \to e_1 \text{ else } e_2 : \tau}\ \text{(T-Match)} \qquad \frac{\begin{array}{c} S;\Gamma;X \vdash_p e : \tau' \\ S;\Gamma \vdash \tau' <: \tau \end{array}}{S;\Gamma;X \vdash_p e : \tau}\ \text{(T-Sub)}$$

$S;\Gamma \vdash \tau <: \tau' \quad$ where $S;\Gamma;\cdot \vdash x : \{y{:}\tau \mid \phi\} \Rightarrow S;\Gamma, y{:}(p,\tau) \vdash \hat{x} : \phi$ **Subtyping**

$$\frac{S;\Gamma \vdash \tau_1 \cong \tau_2}{S;\Gamma \vdash \tau_1 <: \tau_2}\ \text{(S-Eq)} \qquad \frac{}{S;\Gamma \vdash \{x{:}\tau \mid \phi\} <: \tau}\ \text{(S-UnRef)} \qquad \frac{S;\Gamma \vdash \tau <: \tau' \quad \hat{y} \in FV(v_p) \quad S;\Gamma, x{:}(p,\tau) \vdash_\bot v_p : \phi}{S;\Gamma \vdash \tau <: \{x{:}\tau' \mid \phi\}}\ \text{(S-Ref)}$$

$S;\Gamma \vdash \tau \cong \tau' \quad S;\Gamma \vdash e \cong e'$ **Equivalence of types and type indices**

$$\frac{}{S;\Gamma \vdash \tau \cong \tau}\ \text{(TE-Id)} \qquad \frac{S;\Gamma \vdash \tau_1 \cong \tau_1' \quad S;\Gamma \vdash \tau_2 \cong \tau_2'}{S;\Gamma \vdash \tau_1\ \tau_2 \cong \tau_1'\ \tau_2'}\ \text{(TE-App)} \qquad \frac{S;\Gamma \vdash \tau_1 \cong \tau_1' \quad S;\Gamma \vdash v_p \cong v_p'}{S;\Gamma \vdash \tau_1\ v_p \cong \tau_1'\ v_p'}\ \text{(TE-Dep)}$$

$$\frac{}{S;\Gamma \vdash v_p \cong v_p}\ \text{(EE-Id)} \qquad \frac{v_p \doteq v_p' \in \Gamma \vee v_p' \doteq v_p \in \Gamma}{S;\Gamma \vdash v_p \cong v_p'}\ \text{(EE-Match)} \qquad \frac{\forall i,j \quad S;\Gamma \vdash \tau_i \cong \tau_i' \quad S;\Gamma \vdash v_j \cong v_j'}{S;\Gamma \vdash D\tau_1 \ldots \tau_m v_1 \ldots v_n \cong D\tau_1' \ldots \tau_m' v_1' \ldots v_n'}\ \text{(EE-Cons)}$$

Figure 5. Static semantics of FINE

key rule, (EE-Match), allows a type-level term v_p to be equated with v_p' when an assumption $v_p \doteq v_p'$ appears in the context.

3.3 Dynamic semantics

The operational semantics of FINE are instrumented to account for two program properties. First, our semantics places affinely typed values in a memory M. Reads from the memory are destructive, which allows us to prove that in well-typed programs, affine values are never used more than once. The semantics also tracks the privilege of expressions by propagating brackets through reductions. This allows us to prove an information-hiding property for our module system. The main judgment is written $(M,e) \overset{p}{\rightsquigarrow} (M',e')$, and states that given an initial memory M an expression e steps to e' and updates the memory to M'. The p-superscript indicates that e steps while using the privilege of the principal p.

Figure 6 shows the interesting rules from our small-step operational semantics for FINE. Evaluation contexts define a standard left-to-right, call-by-value semantics. As for values, evaluation contexts E_p are divided into families corresponding to principals. The omitted rules include congruences that allow reduction under a context, standard beta-reduction for type and term applications, unrolling of fixed points, and pattern matching.

Reduction rules that do not involve reading from memory are written $e \overset{p}{\rightsquigarrow} e'$. All the interesting rules that manage privileges and brackets fall into this fragment. Redundant brackets around p-values can be removed using (E-Strip). However, not all nested brackets can be removed, as (E-Nest) shows. In (E-Ext), a λ-binder is extruded from a function with q-privilege so that it can be applied to a p-value. We have to be careful to enclose occurrences of the the bound variable in e within p-brackets, to ensure that e treats

$$\text{p-Evaluation context} \quad E_p \quad ::= \quad \bullet \mid E_p\ e \mid v_p\ E_p \mid E_P\ \tau \mid \text{match } E_p \text{ with } D\ \vec{\tau}\ \vec{x} \rightarrow e_1 \text{ else } e_2 \qquad \text{Store} \quad M \quad ::= (x, v_p), M \mid \cdot$$

$$\langle v_p \rangle_p \overset{p}{\leadsto} v_p \text{ (E-Strip)} \quad \langle\langle v_q \rangle_q \rangle_r \overset{p}{\leadsto} \langle v_q \rangle_q \text{ (E-Nest)} \quad \langle \lambda x{:}t.e \rangle_q \overset{p}{\leadsto} \lambda y{:}t.\langle e[\langle y \rangle_p / x] \rangle_q \text{ (E-Ext)} \quad \langle \Lambda\alpha{::}\kappa.e \rangle_q \overset{p}{\leadsto} \Lambda\alpha{::}\kappa.\langle e \rangle_q \text{ (E-TExt)}$$

$$\frac{e \overset{q}{\leadsto} e'}{\langle e \rangle_q \overset{p}{\leadsto} \langle e' \rangle_q} \text{ (E-Br)} \quad \frac{S; \cdot; \cdot \vdash v_p : \tau \quad S; \cdot \vdash \tau :: \mathtt{A} \quad M' = M, (x, v_q) \quad x \text{ fresh}}{M, v_p \overset{p}{\leadsto} M', x} \text{ (E-Construct)} \quad \frac{M = M', (x, v_q)}{M, x \overset{p}{\leadsto} M', v_q} \text{ (E-Destruct)}$$

$$\frac{M, e \overset{p}{\leadsto} M', e'}{M, E_p[e] \overset{p}{\leadsto} M', E_p[e']} \text{ (E-Cong)} \quad \frac{e \overset{p}{\leadsto} e'}{M, E_p[e] \overset{p}{\leadsto} M, E_p[e']} \text{ (E-Pure)} \quad \frac{v_p \doteq \theta(D\ \vec{\tau}\ \vec{x}) \Rightarrow e = \theta(e_1) \quad e = e_2 \text{ otherwise}}{\text{match } v_p \text{ with } D\ \vec{\tau}\ \vec{x} \rightarrow e_1 \text{ else } e_2 \overset{p}{\leadsto} e} \text{ (E-Match)}$$

$$\lambda x{:}\tau.e\ v_p \overset{p}{\leadsto} e[v_p/x] \text{ (E-Beta)} \quad \Lambda\alpha{::}\kappa.e\ \tau \overset{p}{\leadsto} e[\tau/\alpha] \text{ (E-TBeta)} \quad \text{fix } f{:}t.v_p \overset{p}{\leadsto} v_p[(v_p[\text{fix } f{:}t.v_p/f])/f] \text{ (E-Fix)}$$

Figure 6. Dynamic semantics of FINE

its argument abstractly. (E-TExt) extrudes a Λ-binder. Since type-level terms are always checked with $\top$-privilege, we do not need to enclose α in p-brackets. Finally, (E-Br) allows evaluation to proceed under a bracket $\langle\cdot\rangle_q$ with q-privilege.

The only two rules in our semantics that manipulate the store are (E-Construct) and (E-Destruct). The former allocates a new location x for an affine value v_p into the store M, non-deterministically, and replaces v_p with x. When a location x is in destruct position, (E-Destruct) reads a value v_p from M and deletes x.

Theorem 1 proves the soundness of the FINE type system through the standard progress and preservation lemmas. In addition to showing that well-typed programs never get stuck, our soundness result guarantees that affine values are destructed at most once—a result that shows that state changes are modeled accurately. The appendix contains the full statement and proof.

Theorem 1 (Soundness)**.** *The* FINE *type system is sound.*

3.4 Security

FINE's module system provides two general purpose security properties—proofs appear in the appendix. The first, corresponding to a secrecy property, is value abstraction. Theorem 2, stated below, states that a program e without p-privilege cannot distinguish p-values. As a corollary, we can also derive an integrity property, namely that a program without p-privilege cannot manufacture a p-value to influence the behavior of code with p-privilege.

Theorem 2 (Value abstraction)**.**

$\forall S, x, p, q, \tau, \tau', v_p^1, v_p^2, e.$ *where e, a non-value free of p-privilege*
$(S; x{:}(p,\tau); x \vdash_q e : \tau' \ \wedge\ p \sqsubseteq q \notin S \ \wedge\ \forall i.S; \cdot; \cdot \vdash_p v_p^i : \tau)$
$\exists e'.\ S; x{:}(p,\tau) \vdash_q e' : \tau' \ \wedge \forall i.e[v_p^i/x] \overset{q}{\leadsto} e'[v_p^i/x]$

Type soundness and these general-purpose security theorems provide a useful set of primitives using which application-specific security properties can be proved. For example, applying our type soundness and security theorems to LookoutRM, it is straightforward to show (with suitable type-correct implementations of the functions in LookoutRM's API) that state updates are modeled accurately. Specifically, one can show that a reduction sequence of any program using LookoutRM will never use more than a single memory location of type StateIs s, for any s. Additionally, following prior work on Fable, we can show that our mechanism for information-flow tracking accounts for dependences accurately.

Ultimately, we would like to formalize and prove higher-level security theorems for applications. For example, we would like to prove that LookoutRM and UserPolicy correctly ensure that plugins never leak the contents of emails from one friend to another, and that no plugin ever replies to an email more than once. Formalizing and proving these properties for specific programs is beyond the scope of this work. However, we plan to investigate the integration of tools like Margrave (Fisler et al. 2005), specifically designed for the analysis of the style of state-modifying authorization policies investigated here, with the analysis of FINE programs.

4. Proof extraction

Our compiler extracts proofs of refinement formulas from Z3 as typeable FINE proof terms. This section discusses our representation of proofs, and an initial "derefinement" translation of source programs. The result of this translation is a FINE program in which all values v given a refinement type $\{x{:}\tau \mid \phi\}$ are replaced (to a first approximation) with pairs of the form $(x{:}\tau * \text{proof } \phi)$, i.e., dependent pairs containing the value v and a proof term that serves as evidence for the refinement formula ϕ. This approach removes the prover from our TCB, and enables a translation to our target language DCIL, described in detail in Section 5.

4.1 Representation of proof terms

At the source level, we interpret user-provided assumptions and the types ϕ that appears in refinements $\{x{:}\tau \mid \phi\}$ as formulas in a classical first-order logic. To give a value v a refined type, we present a theory with the user axioms, equality assumptions accumulated in the context, and the negated formula $\neg\phi[v/x]$ to Z3. If Z3 can refute the formula, it produces a proof trace. We use an LCF-style (Milner 1979) approach to translate the proof traces reported by Z3 into proof terms in FINE.

The proof system in FINE axiomatizes a classical first-order logic with equality by defining an abstract datatype proof::$\star \rightarrow \star$. Inference rules of the logic and user-provided axioms are represented using data constructors for the proof type. Logical connectives in formulas are represented using type constructors, e.g., And::$\star \rightarrow \star \rightarrow \star$, Not::$\star \rightarrow \star$, and quantified formulas are represented using the binding constructs provided by dependent function types. A selection of the constructors in the kernel of our proof system are shown below. These include inference rules and constructors that allow proof terms to be composed monadically. We also show the translation of the user axiom Ax1 from Figure 1.

T: proof True
Contra: proof (not α) $\rightarrow$ proof $\alpha \rightarrow$ proof False
Destruct_false: proof False $\rightarrow$ proof α
Bind_pf: proof $\alpha \rightarrow (\alpha \ \rightarrow$ proof $\beta) \rightarrow$ proof β
Ax1: proof (f:file $\rightarrow$ proof (CanWrite Admin f))

In addition to the core inference rules, we generate proof principles for a first-order treatment of equality. A more compact higher-order treatment of equality is not possible, since our target language does not support quantification over types with higher-order kinds. For example, for the prin type defined in Figure 1, we automatically generate a type Eq_prin corresponding to equality for prin values, and substitution principles relating Eq_prin to other propositions in the program. Some of the auto-generated types and axioms are shown below.

```
type Eq_prin:: prin → prin → ⋆
Refl_eq_prin: p:prin → proof (Eq_prin p p)
Mono_CanWrite: p1:prin → p2:prin → proof (Eq_prin p1 p2) → f:file →
                   proof (CanWrite p1 f) → proof (CanWrite p2 f)
```

For a flavor of the proof terms we generate, consider the check function from Figure 1. In order to type check its return value, we must prove the validity of CanWrite p f in a context that includes the assumption p $\doteq$ Admin and the Ax1 axiom. We currently translate Z3 proofs directly into corresponding FINE terms. For example, Z3 proofs often end with applications of the Contra and Destruct_false rules, even when these are not necessary. We omit these rules in the following proof term, for clarity.

```
(Mono_CanWrite Admin p (Refl_eq_prin p)
    (Bind_pf (x:file → proof (CanWrite Admin x))
             (CanWrite Admin f)
      Ax1 (λg: f0:file → proof (CanWrite Admin f0). g f))))
```

The sub-term Refl_eq_prin p can be given the type Eq_prin Admin p in a typing context that includes an assumption p $\doteq$ Admin (using the (TE-App) and (EE-Match) rules).

FINE includes recursion. So, we do not claim that this proof system is logically consistent. However, our type soundness and value abstraction theorems guarantee that proof terms are constructed using only the data constructors from our proof system and the user-supplied axioms, and that if a proof term has a normal form, then that normal form has the desired type. As a defense against obviously incorrect proofs, we implement a simple syntactic check to ensure that values of the proof α type are constructed in a recursion-free fragment of FINE (and also DCIL). In the future, we plan to investigate approaches such as Operational Type Theory (Stump et al. 2008) to recover logical consistency in the presence of recursion.

4.2 Derefinement of FINE

Our compiler normalizes the type structure of FINE programs so that every type is of the form $\{x{:}\tau \mid \phi\}$, where both τ and ϕ are unrefined types—a type can always be put in this form. After type checking and generating proof terms for all refinement formulas, we replace all refinement types with dependent pair types, i.e., the translation $[\![\cdot]\!]$ of a normalized type $\{x{:}\tau \mid \phi\}$ is the type $(x{:}[\![\tau]\!] * \text{proof } [\![\phi]\!])$. In other words, our translation "boxes" every τ-value with a proof term for a refinement formula. The uniform structure of a derefined program simplifies our translation and allows us to properly account for proof and non-proof values by distinguishing the kind of boxed types from unboxed types—the appendix contains the details. However, this representation is inefficient and requires inserting code to unbox a value by projecting out its non-proof component when a boxed value appears in destruct position. An optimization pass to remove redundantly boxed terms is straightforward and can be used to give fwrite the type:

```
fwrite: p:prin→ cred p→ f:file→ proof (CanWrite p f)→ string→ unit
```

We assume the optimized type for fwrite in Section 5, to keep our examples compact.

5. Translating FINE to DCIL

This section presents DCIL, an extension of a functional fragment of CIL. We use CIL generics to translate many basic FINE constructs (Kennedy and Syme 2004). DCIL extends CIL with affine types, type-level functions, and classes parametrized by values. We discuss how to represent all our extensions in standards-compliant .NET assemblies. Code consumers can choose to use a type checker for DCIL for security checking, but otherwise can run FINE programs on stock virtual machines.

module	mod.	::=	$\{\overrightarrow{\text{tdcl}}, \overrightarrow{\text{ddcl}} \text{ in } e\}$
vis. qual.	ψ	::=	public | internal
abs. class	tdcl	::=	$\psi\ T\langle\vec{\alpha}{::}\vec{\kappa}, \vec{x}{:}\vec{\tau}\rangle{::}\kappa\{\overrightarrow{\text{fdcl}}, \overrightarrow{\text{mdcl}}\}$
data class	ddcl	::=	$\psi\ D\langle\vec{\alpha}{::}\vec{\kappa}, \vec{x}{:}\vec{\tau}\rangle{:}T\langle\vec{\tau}, \vec{v}\rangle\{\overrightarrow{\text{fdcl}}, \overrightarrow{\text{mdcl}}\}$
fld. decl.	fdcl	::=	$f{:}\tau$
meth. decl.	mdcl	::=	$\tau\ m\langle\alpha{::}\kappa\rangle(x{:}\tau)\{e\}$
expr.	e	::=	$v \mid D\langle\vec{\tau}, \vec{v}\rangle \mid v.f \mid v.m\langle\tau\rangle(v)$
		|	v isinst $D\langle\vec{\tau}, \vec{v}\rangle$ then e_t else e_f
		|	let $x = e_1$ in $e_2 \mid \langle e\rangle_p$
value	v	::=	$x \mid D\langle\vec{\tau}, \vec{v}\rangle$
type	τ	::=	$\alpha \mid T\langle\vec{\tau}, \vec{v}\rangle \mid\ !\tau \mid \backslash x{:}\tau_1.\tau_2 \mid \tau\ v$
kind	κ	::=	$\star \mid \mathtt{A} \mid \tau \to \kappa$

Figure 7. Syntax of DCIL

5.1 Syntax

Figure 7 shows the syntax of DCIL. We re-use metavariables from FINE for syntactic categories in DCIL—the context will make the distinction clear. Modules in FINE are translated to modules in DCIL, and we use visibility qualifiers to model information-hiding in DCIL. All types in FINE are translated to abstract classes T. FINE values $v{:}\tau$ are translated to instances of *data classes* D, where D extends T, the class corresponding to τ. Classes are parametrized by a list of type parameters $\vec{\alpha}{::}\vec{\kappa}$ and also by a list of value parameters $\vec{x}{:}\vec{\tau}$. Classes include field and method declarations, as usual.

The syntax of expressions in DCIL is presented in a form that resembles ANF (Flanagan et al. 1993), which helps simplify our typing rules. For this reason, expressions include let-bindings. Expressions also include values v (variables or instances of data classes D), field projections, method calls, and a runtime type-test construct, (v isinst $D\langle\vec{\tau}, \vec{v}\rangle$ then e_t else e_f), used to translate pattern matching. As in FINE, $\langle e\rangle_p$ records the privilege of e to be module p. Let-bindings and type-tests are macro instructions in DCIL—each corresponds to several CIL instructions.

Types include type variables and fully-instantiated abstract classes $T\langle\vec{\tau}, \vec{v}\rangle$. Affine types are written $!\tau$, as in FINE. DCIL includes a restricted form of type-level function $\backslash x{:}\tau_1.\tau_2$ to represent dependent types. Type-level function application is denoted $\tau\ v$. The kind language in DCIL includes $\star$ and $\mathtt{A}$ to categorize normal and affine types, respectively, and $\tau \to \kappa$, the kind of type-level functions.

5.2 Static semantics of DCIL

The main innovation of DCIL is in the following three features. First, in addition to $\star$-kinded type parameters, classes in DCIL can include affine type parameters, type-function parameters and value parameters. We show how value parameters can be represented using standard field declarations. The appendix discusses how type functions can be encoded using custom attributes. Importantly, DCIL does not include type parameters of kind $\star \to \kappa$ or $\mathtt{A} \to \kappa$ even though these kinds appear in FINE. We show how to translate uses of these kinds in FINE using parametrized class declarations.

Second, we formalize affine types and use this to model stateful programming in FINE. The appendix shows how affine types can be represented in CIL using .NET type modifiers. Affine type modifiers are opaque to the .NET runtime, and only need to be interpreted by a DCIL-aware bytecode verifier.

Finally, we distinguish DCIL classes that represent source-level types (abstract classes $T\langle\vec{\tau}, \vec{v}\rangle$) from data classes ($D\langle\vec{\tau}, \vec{v}\rangle$). This allows us to account for affine assumptions in a manner that corresponds closely to the source language. A naïve formulation that does not include this distinction results in a more complex metatheory, in which affine typing has to account for uses of variables both within terms as well as in types.

$\Sigma; \Delta; \Gamma; X \vdash_p e : \tau$ — Expression typing

$$\frac{\Sigma_p(D) = \psi\ D\langle \vec{\alpha}{::}\vec{\kappa}, \vec{x}{:}\vec{\tau'} \rangle : T\langle \vec{\tau''}, \vec{v'} \rangle \quad \forall i. \Sigma; \Delta; \Gamma \vdash_p \tau_i :: \kappa_i \quad \forall j. \Sigma; \Delta; \Gamma, X_j \vdash_p v_j : \tau'_j[\vec{\tau}/\vec{\alpha}][v_1 \ldots v_{j-1}/x_1 \ldots x_{j-1}]}{\Sigma; \Delta; \Gamma; X_1 \ldots X_m \vdash_p D\langle \tau_1 \ldots \tau_n, v_1 \ldots v_m \rangle : T\langle \vec{\tau''}, \vec{v'} \rangle[\vec{\tau}/\vec{\alpha}][\vec{v}/\vec{x}]} \text{ (T-New)}$$

$$\frac{\Sigma; \Delta; \Gamma; X \vdash_p v : T\langle \vec{\tau_3}, \vec{v_3} \rangle \quad \Sigma_p(T\langle \vec{\tau_3}, \vec{v_3} \rangle) = \tau_2\ m\langle \alpha{::}\kappa \rangle(x{:}\tau_1) \quad \Sigma; \Delta; \Gamma \vdash \tau :: \kappa \quad \Sigma; \Delta; \Gamma; X' \vdash_p v' : \tau_1[\tau/\alpha]}{\Delta; \Gamma; X, X' \vdash_p v.m\langle \tau \rangle(v') : \tau_2[\tau/\alpha][v'/x]} \text{ (TT-App)}$$

$\Sigma; \Delta; \Gamma \vdash \tau :: \kappa$ — Kinding of types

$$\frac{\Sigma; \Delta; \Gamma \vdash \tau_1 :: \star \quad \Sigma; \Delta; \Gamma, x{:}\tau_1 \vdash \tau_2 :: \kappa}{\Sigma; \Delta; \Gamma \vdash \backslash x{:}\tau_1.\tau_2 :: \tau_1 \to \kappa} \text{ (TK-Fun)} \qquad \frac{\Sigma; \Delta; \Gamma \vdash \tau :: \tau_1 \to \kappa \quad \Sigma; \Delta; \Gamma; \cdot \vdash v : \tau_1}{\Sigma; \Delta; \Gamma \vdash \tau\ v :: \kappa} \text{ (TK-App)}$$

$$\frac{\Sigma(T) = T\langle \vec{\alpha}{::}\vec{\kappa}, \vec{x}{::}\vec{\tau'} \rangle{::}\kappa' \quad \Sigma; \Delta; \Gamma \vdash \tau_i :: \kappa_i \quad \Sigma; \Delta; \Gamma; \cdot \vdash_{\top} v_j : \tau'_j[\vec{\tau}/\vec{\alpha}][v_1 \ldots v_{j-1}/x_1 \ldots x_{j-1}]}{\Sigma; \Delta; \Gamma \vdash T\langle \vec{\tau}, \vec{v} \rangle :: \kappa'} \text{ (TK-T)}$$

Figure 8. Static semantics of DCIL (selected rules)

Figure 8 shows key elements from the static semantics of DCIL. The typing judgment uses Σ, a context that records the class declarations in scope, corresponding to the signature S in FINE; Δ, a context that records type variables and their kinds; Γ, a typing environment with variable bindings and equations resulting from runtime type-tests (as in FINE); and X, a subset of the variable bindings in Γ, corresponding to the set of affine capabilities in FINE. Expressions e are typed in the context of a module p, corresponding to source-level principals.

The rule (T-New) shows the typing rule for $D\langle \vec{\tau}, \vec{v} \rangle$, the constructor of a data class D with type parameters $\vec{\tau}$ and value arguments $\vec{v}$. In the first premise, we look up the declaration of class D in Σ_p, the restriction of the signature Σ to declarations visible in module p. In the second premise, we check that each type parameter has the kind expected by the declaration. The third premise checks each value argument v_j with a subset of the affine assumptions X_j. The expected type of each v_j is dependent on all the type parameters $\vec{\tau}$, and the prefix of arguments $v_1 \ldots v_{j-1}$. Importantly, in the conclusion, we give $D\langle \vec{\tau}, \vec{v} \rangle$ a type of the form $T\langle \vec{\tau}, \vec{v} \rangle$, where T is the abstract super-class of D. This allows us to ensure that affine variables never appear in DCIL types, simplifying the connection between the typing and kinding judgment (discussed shortly).

The type and term application constructs in FINE are collapsed into a single method invocation construct in DCIL. The first two premises of (TT-App) check that v, the object on which the method is invoked, has a declaration for method m. The third and fourth premises check that the arguments to m have appropriate kinds or types, and the conclusion substitutes the actual type and term arguments in the return type τ_2. DCIL's use of A-normal form ensures that non-values never escape into types.

A selection of the kinding rules are also shown in Figure 8. (TK-Fun) ensures that the argument of a type function is always non-affine, echoing a similar restriction on kinds in the source language. (TK-App) and (TK-T) check type-function application and instantiations of abstract classes, respectively. Both rules show that type-level values are checked using an empty set of affine capabilities X. Although affine variables can appear in data class instantiations, our separation of data classes D from abstract classes T ensures that affine variables never escape into types.

The semantics of DCIL also includes a type-equivalence judgment $\Sigma; \Delta; \Gamma \vdash \tau \cong \tau'$. This is similar to the corresponding judgment in FINE, with the addition of a single rule that equates types related by β-reduction of type-function applications. Since DCIL does not contain refinement types, its semantics does not contain an analog of FINE's semantic sub-typing relation $S; \Gamma \vdash \tau <: \tau'$.

We have proved DCIL sound, using the standard progress and preservation lemmas. Additionally, we have shown that DCIL programs respect their visibility qualifiers, a property analogous to the value abstraction for FINE programs. The appendix contains the complete semantics of DCIL and the proofs of these theorems.

Theorem 3 (Soundness)**.** *The* DCIL *type system is sound.*

Theorem 4 (Visibility qualifier)**.** *Well-typed* DCIL *programs respect visibility qualifiers.*

5.3 Translation of FINE to DCIL

This section illustrates our translation from FINE to DCIL using examples. The appendix formalizes the translation and proves that it preserves types (Theorem 5).

Translation of modules. Figure 9 shows a DCIL program corresponding to a fragment of the FINE program in Figure 1. The type and data constructor declarations in a FINE module are accumulated as class declarations in a DCIL assembly, with visibility qualifiers used to capture source-level private types. Modules which are granted the privilege of other modules are placed within the same assembly, e.g., UserPolicy and LookoutRM, from Section 2, are compiled to modules in a common assembly.

Translation of type constructors. Type constructors are translated to declarations of abstract classes T. The type and value parameters of a type constructor are carried over directly. For example, the prin type is shown in Figure 9 at line 2 as an abstract class with no parameters. The dependent-type constructor cred::prin $\to \star$ is translated (line 5) to an abstract class with a prin-typed value parameter.

Translation of data constructors. Data constructors in FINE are translated to declarations of data classes D. At line 6, we show the data class corresponding to the Auth constructor from Figure 1. The AC module in FINE required the Auth constructor to only be usable by modules with AC-privilege. So, in DCIL, we qualify the Auth data class using the **internal** visibility qualifier. Data classes always extend abstract classes that correspond to the result type of the source-level data constructor. The field declarations of a data class are always in one-to-one correspondence with its value parameters, e.g., the prin p field of the Auth class. We use this correspondence to encode all value parameters in fields and do not require changing CIL. Note that user-provided assumptions are translated just as ordinary data constructors, e.g., Ax1 at line 10 of Figure 9.

Translation of function types. Kennedy and Syme (2004) show how to translate (non-dependent) function types to a CIL-like language through the use of a polymorphic abstract class. We extend this idea using type-level functions in DCIL to capture dependent function types. Our translation uses the following declarations:

$$\mathsf{DepArrow}\langle \alpha_1{::}\star, \alpha_2{::}\alpha_1 \to \star \rangle :: \star\{ \quad (\alpha_2\ x)\ \mathsf{App}(x{:}\alpha_1)\{\} \quad \}$$

```
1 assembly AC {
2 public prin<>:: ⋆ {}
3 public U<s:string>:prin{string s;}
4 public Admin<>:prin{}
5 public cred<p:prin>::⋆ {}
6 internal Auth<p:prin>: cred<p> {prin p;}
7 public CanWrite<p:prin,f:file>::⋆ {}
8 public Ax1<f:file>::CanWrite<Admin,f> {file f;}
9 public fwrite<...>:DepArrow<...>}
10
11 assembly Client {
12 (∗ client:p:prin → cred p → file → unit ∗)
13 public client<>:
14   DepArrow<prin,\p:prin.Arrow<cred<p>,Arrow<file, unit>>>
15   {Arrow<cred<p>, Arrow<file, unit>> App(p:prin) { clientp<p>;}}
16 (∗ (client p): cred p → file → unit ∗)
17 public clientp<p:prin>:Arrow<cred<p>, Arrow<file, unit>>
18   {Arrow<file, unit> App(c:cred<p>) { clientc<p,c>; }
19 (∗ (client p c):file → unit ∗)
20 public clientpc<p:prin, c:cred<p>>:Arrow<file, unit> {
21   unit App(f:file) {
22     p isinst Admin then
23       let pf = ... in (∗ translated proof term ∗)
24       let fwrite = AC.fwrite<...> in
25         fwrite.App(p).App(c).App(f).App(pf).App(''hello'')
26     else ()}}
```

Figure 9. Translation of FINE to DCIL

Class DepArrow takes two type parameters: α_1 for the argument type and α_2 for a *type function*—the return type of App is the result of applying α_2 to the argument x. Source-level types such as p:prin → cred p are translated to instances of DepArrow; in this case, DepArrow⟨prin, \p:prin.cred⟨p⟩⟩. We also include the abstract classes shown below to represent non-dependent functions, and functions that take affine arguments or produce affine results—other combinations of kinds are unnecessary.

Arrow$\langle\alpha_1{::}\star, \alpha_2{::}\star\rangle :: \star\{\ \ (\alpha_2)\ \mathrm{App}(x{:}\alpha_1)\{\}\ \ \}$
Arrow_AA$\langle\alpha_1{::}\mathtt{A}, \alpha_2{::}\mathtt{A}\rangle :: \star\{\ \ (\alpha_2)\ \mathrm{App}(x{:}\alpha_1)\{\}\ \ \}$
DepArrow_A$\langle\alpha_1{::}\star, \alpha_2{::}\alpha_1 \to \mathtt{A}\rangle :: \star\{\ \ (\alpha_2\ x)\ \mathrm{App}(x{:}\alpha_1)\{\}\ \ \}$

The source-level function client from Figure 1 is a function with three arguments. Because client is curried, it is translated (lines 13-26 of Figure 9) as three data class declarations. Each of the client data classes extends an instantiated DepArrow or Arrow class. The body of client simply calls the clientp version (by constructing it), and clientp calls clientpc.

Translation of expressions. The body of clientpc illustrates the translation of FINE expressions. The use of pattern matching at line 16 of Figure 1 is translated to a type-test at line 22 of Figure 9. At line 23 of Figure 9 we show a placeholder for the translation pf of the proof term (from Section 4) of type proof<CanWrite<p,f>>. At line 24, we obtain a reference to the fwrite value exposed by AC. As discussed in Section 4.2, after derefinement, the type of fwrite is normalized to p:prin→ cred p→ f:file→ CanWrite p f→ string→ unit, which in DCIL corresponds to the type

```
DepArrow<prin,
 \p:DepArrow<cred p, \f:DepArrow<file,
      Arrow<CanWrite<p, f>, Arrow<string, unit>>>>>
```

At line 25 of Figure 9, we show the call to fwrite translated as successive calls to the App method. The proof term pf is passed as an extra argument, although this is not evident in the source program of Figure 1. Since the source language does not provide a facility to extract a (proof ϕ)-typed value from an object of type $\{x{:}\tau \mid \phi\}$, we can show that proof-terms in DCIL are computationally irrelevant (although we have yet to prove this formally). If necessary for efficiency, proof terms could be erased after the target code has been type checked. Alternatively, proof terms could be logged at runtime, if an application like evidence-based audit (Vaughan et al. 2008) is to be supported, or, if running in a distributed setting, proof terms could be communicated between principals for proof-carrying authorization (Appel and Felten 1999).

Polymorphic FINE types $\forall\alpha{::}\kappa.\tau$ are translated to DCIL classes and type application to applications of polymorphic methods. This translation follows an encoding proposed by Kennedy and Syme and adds no further novelty. The appendix includes a formalization of this translation, as well as a full statement and proof of the following theorem, the main result of this section.

Theorem 5 (Type-preserving translation)**.** *A well-typed* FINE *program is translated to a well-typed* DCIL *program.*

6. Implementation

We have implemented a prototype compiler, currently 13,581 lines of F# code, extending a front-end for the F# compiler. Our compiler is able to type check all source programs that appear in this paper and several other examples besides. The extraction of typeable proofs from Z3 and the translation to DCIL remains a work in progress. For the AC example (and others like it), we are able to extract proofs from Z3, type check them, and translate the result to DCIL, and type check the generated DCILprogram. Although we are able to type check the LOOKOUT source program, at the time of writing, a type-preserving compilation to DCIL was not complete.

Proof extraction. Inspecting Z3 proofs and translating them to FINE proof terms presented a number of engineering challenges. Z3 often uses opaque rewriting strategies in proofs. We have devised a translation from several of Z3's rewriting strategies to our kernel of proof rules. However, handling all of Z3's strategies will require more work. We are considering extending Z3's proof reporting facility to provide more information about the rewrites it applies to help with this task.

6.1 Example programs

In addition LOOKOUT, our type-checked example programs include a model of Continue (Dougherty et al. 2006), a conference management server. This model carries over naturally to FINE, using the same refined state idiom from LOOKOUT. The authors of Continue point out that almost all interesting bugs in Continue have been related to access control. FINE provides a way to ensure that software like Continue is secure by construction. We have also implemented a more elaborate version of AC, a reference monitor that implements an automaton-based policy for a file system API. This example is interesting because the state of the policy is partitioned into multiple pieces, each piece recording the state of a particular file handle. Our examples also show how to enforce an information flow policy, more elaborate than the policy we use in Figure 3. Our policy defines a CanFlow p q proposition and lattice axioms for this proposition to show when data labeled p is allowed to flow to a sink labeled q. This policy could easily be integrated with LOOKOUT.

In the remainder of this section we describe our model of Continue in further detail.

6.1.1 Modeling the Continue conference management server

In this section, we model the enforcement of a fragment of the stateful authorization policy used by the Continue conference management tool (Krishnamurthi 2003). This policy combines elemenets of role and attribute-based access control, with the simple authentication mechanism developed with AC in Section 2.1. Our enforcement model closely follows the model for enforcing stateful policies developed in Section 2.2.

Figure 10 defines two modules ConfRM, a reference monitor that mediates access to a database of paper submissions and reviews, and ConfWeb the main request-processing loop of a webserver that interacts with the database via ConfRM.

```
1 module ConfRM =
2 open AC
3 type role = Author | Reviewer | Chair
4 type action = Submit | Review | ReadScore
5 type phase = Submission | Reviewing | Meeting
6 type paper = {id:int; title:string; ...}
7 type attr = Role : prin → role → attr
8             | Assigned : prin → paper → attr
9             | Reviewed : prin → paper → attr
10            | Phase : phase → attr
11 type attrs = list attr
12 type In :: attrs → attr → ∗
13 assume Hd:forall (a:attr), (tl:attrs). In (a::tl) a
14 assume Tl:forall (a:attr), (b:attr), (tl:attrs). In tl a ⇒ In (b::tl) a
15
16 type perm = Permit : prin → action → paper → perm
17 type Valid :: attrs → perm → ∗
18
19 abstract type StateIs::attrs → A = Sign : s:attrs → StateIs s
20 type state::A = (s:attrs ∗ StateIs s)
21 let init:state = let a = [Role (U ‘‘Jens’’) Chair; ...] in (a, Sign a)
22
23 assume C1: forall (p:prin), (r:paper), (s:attrs).
24  (StateIs s) && In s (Phase Submission) && In s (Role p Author) ⇒
25   Valid s (Permit p Submit r)
26 assume C2: forall (p:prin), (r:paper), (s:state).
27   (StateIs s) && In s (Phase Reviewing) && In s (Assigned p r) ⇒
28   Valid s (Permit p Review r)
29 assume C3: forall (p:prin), (r:paper), (s:state).
30   (StateIs s) && In s (Phase Meeting) && In s (Reviewed p r) ⇒
31   Valid s (Permit p ReadScore r)
32
33 type st1<r> = (s:{a:attrs | StateIs a ⇒ In a r} ∗ StateIs s)
34 type st2<r,r’> = (s:{a:attrs | StateIs a⇒ In a r && In a r’} ∗ StateIs s)
35 type ok<p> = (s:{a:attrs | StateIs a ⇒ Valid a p} ∗ StateIs s)
36
37 val submit:p:prin → r:paper → ok<Permit p Submit r> → st
38 let submit p r s = let _ = write_to_db p r in s
39
40 val review: p:prin → r:paper → q:string →
41             ok<Permit p Review r> → st1<Reviewed p r>
42 let review p r q s =
43     let _ = write_review_to_db p r q in
44     let (attrs, tok) = s in
45     let nextstate = (Reviewed p r)::attrs in
46     (nextstate, Sign nextstate)
47
48 val close_sub: c:prin → cred c →
49                  st2<Role c Chair, Phase Submission> →
50                  st1<Phase Reviewing>
51 val assign: c:prin → cred c → r:prin → q:paper →
52               st2<Role c Chair, Role r Reviewer> →
53               st1<Assigned r q>
54 end
55
56 module ConfWeb
57 val check: l:attrs → a:attr → {b:bool | b=true ⇒ In l a}
58 let rec check l a = match l with
59   | [] → false
60   | hd::tl → if a=hd then true else check_attr tl a
61
62 let rec loop s = match get_request() with
63   | Submit_paper p paper →
64       let (a, tok) = s in
65        if check a (Phase Submission) && check a (Role p Author) then
66         let s1 = ConfRM.submit author paper (a,tok) in
67         let _ = resp ‘‘Thanks for your submission!’’ in loop s1
68        else
69         let _ = resp ‘‘Sorry, submissions are closed.’’ in loop (a, tok)
70  | Submit_review reviewer paper review → ...
71
72 let _ = loop ConfRM.initial_state
73 end
```

Figure 10. A secure conference management server

The high-level security policy enforced by the reference monitor is represented by the assumptions C1C3 at lines 21-29 of ConfRM.Each assumption is an inference rule that allows propositions of the form Permit p a r to be derived in the current state *s* of the authorization environment. Intuitively, Permit p a r grants the principal p the right to perform action a on resource r. For example, C1 allows p to Submit a paper p, if it can be shown that in the current state s, that p is in the role Author, and that the current phase of the conference is Submission. The assumption C2 is similar in structure, and allows p to Review a paper r if p has been Assigned r in the current state, and if the conference is in the Reviewing phase. C3 is similar, and allows p to view the scores of a paper r only after p has submitted a review for r.

These policy rules make use of the standard ML-style type and data constructors that appear on lines 2-10. The type attrs is a list of attributes that constitute the authorization state and the proposition In is used to assert that a specific attribute is present in the authorization state. The data constructors for the In type are the assumptions Hd and Tl and represent the standard axioms about list membership.

The policy rules allow permissions that are instances of the perm type to be derived from the authorization state. We use the Valid s p type to represent a proposition that a permissiom p is derivable from the state s, although p is not literally present in the attributes that constitute s. The constructors of the Valid type are the three policy rules C1C3.

The final, critical piece of our enforcement strategy is the StateIs proposition. As attributes are added to or removed from the authorization state, we need a mechanism to revoke propositions that were true in a prior state that may no longer be true. We employ the mechanism of *affine types* to achieve this (Walker 2004)—values given an affine type may be destructed *at most once* on all code paths. We classify types into two basic kinds, $*$, the kind of normal types, and A, the kind of affine types. By declaring StateIs :: attrs → A we state that StateIs constructs an affine type from an argument of type attrs, e.g., StateIs [Phase Submission] is a well-formed affine type, and any value of this type can be used at most once. The StateIs type has a single constructor Sign that can be used to assert that a particular list of attributes is the current state of the program. Since authorization decisions depend on the current state, we make StateIs an abstract type, thereby ensuring that only the ConfRM module, can make assertions about the current state of the environment. The current state of the program is represented using the (dependent pair) type state, which is a pair consisting of a list s of attributes, and a value of type StateIs s which attests that s is indeed the current state of the program. Note that since state is a pair that contains an affine component, it is itself affine. Line 20 constructs the initial state of the program by constucting a list a of attributes and using Sign a to assert that it is the current state of the reference monitor.

We now turn to lines 36-50 which defines the external interface exposed by ConfRM to security-sensitive operations. FINE provides parameterized type abbreviations of the form defined on lines 32-34 to simplify the syntax. Each of these abbreviations refines the type of the current state st with a formula that asserts that either a attribute is present in the state or that some permission is derivable from it.

At lines 36-37 we define the submit function. Its type states that in order for a principal p to submit a paper r the caller must be able to derive the Permit p Submit r permission from the current program state s. In the body of the function, we call an internal function (write_paper_to_db, whose definition is omitted) and return the unchanged state s back to the caller.

The review function on lines 39-45 is a little more interesting. This time, in order to use this function, the caller must be able to derive the permission Permit p Review r from the current state. In the body of the function we call another internal function to update a database, and then on lines 43-45 update the state of the program to record that the attribute Review p r, to record that a review has been submitted by p. A state update involves destructing the state tuple into its components, adding attributes to (or removing attributes from) the attribute list a, and then signing the new list of attributes to assert that it is the most current state of the program. The return type of review indicates only that the Reviewed p r attribute has been added to the current state. A more precise refinement could have been used to record that all other attributes in the initial state remain unchanged, although that would complicate our presentation significantly.

Our implementation exposes a number of other functions in the interface of ConfRM. Here, we simply show the types of two of these functions. The close_sub function provides the conference Chair to close the submission phase of the conference. The type of close_sub indicates that it changes the phase of the conference from Submission to Reviewing. The assign function allows the Chair to assign a paper to a Reviewer, and changes the state to record this fact. Note the use of the AC module from Section 2.1 to represent user credentials. Other mechanisms for authentication could just as easily have been slotted in.

We turn now to the ConfWeb module, a client of ConfRM. The function check searches through a list l of attributes to see if it contains an attribute a. The body of check is a standard tail-recursive scan of a list, however the type of check indicates that it return true only if the attribute was indeed in the list.

The main event loop of ConfWeb waits for a request (the type of requests is elided). In the case principal p requests to submit a paper, we first check that the conference is in the Submission phase, and that p is registered in the role of an Author. The type we give to the built-in boolean conjuction operator && is x:bool → y:bool → {z:bool | z=true ⇒ x=true && y=true}, where the && in the formula is logical conjunction. We can use this type, the type of check, and assumption C1, to refine the type of the current state (a,tok) in the **then**-branch to ok<Permit p Submit paper>.

7. Related and future work

Several programming languages and proof assistants use dependent types, including Agda (Norell 2007), Coq (Bertot and Castéran 2004), and Epigram (McBride and McKinna 2004). All of these systems can be used to verify full functional correctness of programs. However, to ensure logical consistency of the type system, these languages exclude arbitrary recursion, making them less applicable for general-purpose programming. Projects like YNot (Chlipala et al. 2009) and Guru (Stump et al. 2008) aim to mix effects like non-termination with dependently typed functional programming; YNot also supports programming with state in an imperative style. Restrictions in both languages ensure that proofs are pure, ensuring that logical consistency is preserved. All of these systems include automation and tactic languages, but programmers must usually construct proofs of correctness along with their code. In contrast, FINE targets weaker, security properties; forgoes logical consistency in favor of practical programming by including recursion; and automatically synthesizes proof terms using an SMT solver. FINE also provides affine types to allow the enforcement of state-modifying policies, which could be expressed in YNot, but not easily in the other languages. To recover logical consistency, FINE could follow Guru's operational type theory—an approach we plan to consider in the future.

Dependent types have also been used for security verification. Jif (Chong et al. 2006) uses a limited form of dependent typing to express dynamic information flow policies. Aura (Jia et al. 2008) is specialized for the enforcement of policies specified in a policy language based on an intuitionistic modal logic. This makes Aura less applicable to policies specified in other other logics, e.g., the Datalog-based policy language of (Dougherty et al. 2006), and Aura cannot model stateful policies. Aura provides logical consistency by excluding arbitrary recursion. Proof terms in Aura are always programmer-provided. As such, Aura is positioned as an intermediate language, rather than a source-level language. Fable (Swamy et al. 2008), is another intermediate language for security verification that uses dependent types. Security policies in Fable are enforced using a TCB delimited from untrusted code using a simple, two-principal module system. FINE's module system generalizes Fable's, with support for a lattice of multiple principals. FINE is also related to λAIR (Swamy and Hicks 2008), a calculus that targets the enforcement of declassification policies. λAIR is lower-level than FINE, and its heavyweight combination of affine and dependent types does not lend itself to integration with a solver.

Refinement types in FINE are related to a similar construct in RCF (Bengtson et al. 2008). Refinement formulas in RCF are drawn from an unsorted logic, rather than using dependent-type constructors, as we do. The lack of dependent type constructors in RCF makes it difficult to derive typeable proof terms, and F7, the implementation of RCF, uses Z3 as a trusted oracle. Without dependent type constructors, it appears impossible to enforce information flow policies in F7. RCF also lacks support for stateful authorization policies, although recent work shows how stateful policies can be modeled in F7 using a refined state monad (Borgstroem et al. 2009). However, the soundness of this encoding relies on a trusted compilation of the program in a linear, store-passing style. FINE's type system also allows the use of refined state monads, but, additionally, through the use of affine types, FINE can check that monadic programs never replay stale states.

Other hybrid-typed languages like Sage (Flanagan 2006) also use trusted external solvers to discharge proofs, but automatically insert runtime checks when the prover fails to discharge a proof obligation. Failed runtime checks can cause subtle leaks of information, and so automatic insertion of runtime checks is not yet a feature of FINE, where security is the primary concern. In the future, we plan to apply recent work on type coercions (Swamy et al. 2009) to FINE to automatically insert security enforcement code in a predictable and secure manner.

A key feature that distinguishes our work from all of the aforementioned projects is the type-preserving translation of FINE to DCIL. This allows us to do translation validation, as well as to apply our tools to the setting of verification of mobile code, e.g., with plugin-based software. DCIL is, to our knowledge, the first bytecode-level, object-oriented, dependently typed language. In the future, we plan to carry types to a lower-level assembly language, further reducing the TCB.

8. Conclusions

This paper has presented FINE, a programming language for enforcing rich, stateful authorization and information flow policies. We showed how to compile FINE to DCIL, a target language for use with the standards compliant .NET virtual machines. Our compiler makes it feasible to construct source programs using state-of-the-art provers, and to distribute low-level code that can be checked for security by code consumers using a small TCB. We plan to continue our development efforts, focusing primarily on applying our tools to the construction of provably secure application software.

A. Soundness of FINE

Definition 6 (Well-formed signature)**.** *Well-formedness of a signature S is defined inductively as*

```
1. S = S', T::k   =>
   S' is well-formed
   and S' |- k

2. S= S', D:(p,t) =>
   S' is well-formed
   and S';- |- t :: k
   and k in {*, A}
   and p in S'
   and D constructs a constructed type
       (i.e., exists T ti ei, final_typ(t) = T ti ei)

3. S=FOL,p1<p2,...,p<p' where FOL is the basic signature for first-order logic with equality
   specialized to a set of ground types TT={T1 .. Tn} and all the principal names p_i are
   distinct.

   FOL=
     T1::*, ..., Tn::*
     Eq_1:T1 -> T1 -> *
     ...
     Eq_n:Tn -> Tn -> *
     And::* -> * -> *
     Or::* -> * -> *
     Not::* -> *
     True::*
     proof::* -> *
     tt:True
```

Definition 7 (Well-formed environment)**.** An environment Env=S;G;X is well-formed iff S;G bind distinct names and all of the following are true

```
1. Env=S;G;X,x => x in dom(G) and
                  x not in X  and
                  and S;G;X is well-formed

2. Env=S;G,x:(p,t);- => FreeVariables(t) <= dom(G)      and
                        p in S                          and
                        S;G;- is well-formed

3. Env=S;G,e1=e2;- => FreeVariables(e1) <= dom(G)       and
                      FreeVariables(e2) <= dom(G)       and
                      and S;G;- is well-formed

4. Env=S;G,'a::k;- => k in {*, A} and
                      S;G;- is well-formed

5. Env=S;-;- => S is a well-formed signature
```

Definition 8 (Well-formed memory)**.**

```
Given a signature S, and a memory M, the environment corresponding to M is written S;G(M), where
G(M) is defined inductively as:

G(.) = .
G(M,(x,v_p)) = G(M),x:(p,t) where S;-;- |- v_p:t

A memory M is well-formed when G(M) exists
```

Lemma 9 (Canonical forms)**.** Forall S, G, X, p, t, v_p,

```
 (A1) S;G;X well-formed
 (A2) S;G;X |-_p v_p : ?(x:t1) -> t2  => exists e, v_p = \x:t.e
 (A3) S;G;X |-_p v_p : ?(\/'a::k.e) => exists e, v_p = /\'a::k.e
```

Proof. By induction on the structure of the typing derivation, appealing to fully-applied data constructors to exclude $(Dv1 : t1 \rightarrow t2)$ etc. □

Theorem 10 (Progress). For all S M e t p,

```
(A1)    S;G(M);dom(M) well-formed and
(A2)    S;G(M);dom(M) |-_p e : t
=> exists v_p, e=v_p  or  exists M' e', (M,e) ~p~> (M',e')
```

Proof. Proof: By induction on the structure of (A2).

```
Case (T-Var-A, T-Var):

  x is a p-value.

Case (T-AVal, T-Datacon, T-Abs, T-Univ):

  v_p, \x:t.e, /\'a::k.e are all values for p.

Case (T-Fix):

  fix f:t.e ~p~> e [fix f:t.e/f]   using [E-Fix]

Case (T-App):

  a. (e = e1 e2): In this case there exists an evaluation context E such that e=E[e1]. From the
     first antecedent of (T-App) we have S;-;- |-_p e1:t and from the induction hyptothesis, exists
     e1', e1 ~p~> e1'. For the conclusion, we use [E-Cong], and produce E[e1'] as the witness for
     the right-side of the goal.

  b. (e = v_p e2): Similar to sub-case a.

  c. (e = v_p v_p'): From the first antecedent of (T-App), we have S;-;- |-_p v_p : ?(x:t1) ->
     t2. From Lemma 3 (Canonical forms), we can conclude that exists e', v_p = \x:t1.e'.

     For the conclusion, we apply [E-Beta] producing e'[v_p/x] as the the witness for the right-side
     of the goal.

Case (T-TApp):

  a. (e= e' t): Similar to (T-App), sub-case a.

  b. (e = v_p t): From the first antecedent of (T-TApp) and Lemma 3 (Canonical forms), we can
     conclude that exists e', v_p = /\'a::k.

     For the conclude we apply [E-TBeta] produces e'[t/'a] as the witness on the right-side of the
     goal.

Case (T-Bracket):

  a. (e=<e'>_q): [E-Bracket] is applicable.

  b. (e=<v_q>_q), where p<>q: We enumerate sub-cases on v_q.

    i. (v_q=u_q):

       <u_q>_q is a p-value, satisfying the left-side of the goal.

   ii. (v_q=\x:t.e):

       <\x:t.e>_q ~p~> \y:t.<e[<y>_p/x]>_q using E-Extrude satisfying the right side of the goal.

  iii. (v_q=/\'a::k.e):

       </\'a::k.e>_q ~p~> /\'a::k.<e>_q using E-TExtrude satisfying the right side of the goal.


   iv. (v_q=<u_r>_r):

        <<u_r>_r>_q ~p~> <u_r>_r using E-Nest, satisying the right-side of the goal.
```

c. (e=<v_p>_p):

<v_p>_p ~p~> v_p using [E-Strip], satisfying the right-side of the goal.

Case (T-Match):

a. (e=match e' with...): Step using evaluation context rule E-Cong

b. (s=match v_p with ...): The antecedents of rules [E-Match1] and [E-Match2] form a tautology. We can satisfy the right-side of the goal using one of the two rules.

Cases (T-Sub, T-Drop-A):

The goal follows directly from the induction hypothesis.

(Note: the rules (E-Construct) and (E-Destruct) are unnecessary for progress)

□

Lemma 11 (Weakening)**.** Lemma (Weakening for typing judgment):

```
For all S G1 G2 G X1 X2 X e t p,
  (A1) S;G1,G2;X1,X2 well-formed
  (A2) S;G1,G2;X1,X2 |-_p e : t
  (A3) S;G1,G,G2;X1,X,X2 well-formed
  => S;G1,G,G2;X1,X,X2 |-_p e : t
```

Lemma (Weakening for kinding judgment):

```
For all S G1 G2 G t k p,
  (B1) S;G1,G2;- well-formed
  (B2) S;G1,G2 |-_p t :: k
  (B3) S;G1,G,G2;- well-formed
  => S;G1,G,G2;- |- t :: k
```

Proof. By mutual induction on the structure of (A2) and (B2), generalizing on G2.
The interesting cases of (A2)

Case (T-Var), (T-Var-A):

In both cases, we can establish the conclusion by using (T-Var/T-Var-A) in the premise of (T-Drop-A), to re-establish to drop the additional affine assumptions in X, and note that G1,G,G2 binds x iff G1,G2 binds x.

Case (T-Abs):

For the third premise of (A2), we use the mutual induction hypothesis to show that the kinding derivation can be weakened to

```
(A2.1')            S;G1,G,G2 |- t::k
```

The fifth premise of (A2) in this case is of the form

```
(A2.3)    S;G1,G2,x:t; X1,X2 |-_p e : t'
```

In this case, we can use the induction hypothesis since we have been careful to generalize on the extended environment G2, since the induction hypothesis specifically allows weakening by inserting assumptions in the "middle" of a context. Thus, we can then establish

```
(A2.3.broken)  S;G1,G,G2,x:t; X1,X,X2,X' |-_p e : t'
```

However, this derivation is not always of the right shape, since when X1,X2 is empty and X is not empty, then when constructing

```
    S;G1,G,G2;X1,X,X2 |-_p \x:t.e : ?(x:t) -> t'
```

with (A2.3.broken) in the premise, the qualifier on the introduced type ?(x:t) -> t' may differ from the qualifier on tf, the type introduced in (A2).

To remedy this, we establish the conclusion by first showing that S;G1,G,G2;X1,X2 is well-formed. Next, we use the induction hypothesis to construct

```
(A2.3')  S;G1,G,G2,x:t; X1,X2,X' |-_p e : t'
```

Finally, we use an application of (T-Drop-A) in the context (S;G1,G,G2; X1,X,X2) to discard the affine assumptions X. In the premises of (T-Drop-A), we use (T-Abs) with (A2.1') and (A2.3'). The other premises are unchanged.

Case (T-Fix): Similar to (T-Abs), always using (T-Drop-A) in the conclusion to discard the affine assumptions in X.

Case (T-Tabs): Similar to (T-Abs).

The interesting cases of (B2)

Case (K-Arrow): Similar to (T-Abs), where generalization of G2 and allowing weakening in the middle of the context is key.

Case (K-Univ): Similar to (K-Arrow).

Case (K-App): The mutual induction hypothesis allows us to establish a weakening for the typing derivation in the second premise.

□

Lemma 12 (Well-kinded typings)**.** Lemma (Well-kinded typings):

```
  For all S G X e t p,
    (A1) S;G;X |-_p e : t
    => exists k, S;G |- t :: k  and k in {*, A}
```

Lemma 6.2 (Well-formed kindings):

```
  For all S G t k,
    (B1) S;G |- t :: k
    =>  S |- k
```

Lemma 6.3 (Well-formed type conversions):

```
  For all S G t t' k,
    (C1) S;G |- t :: k
    (C2) S;G |- t <: t'
    =>  S;G |- t' :: k
```

Proof. Straightforward from mutual induction on the structure of (A1) and (B1).
The interesting cases in (A1)

Case (T-Match):

From the last premise, we ensure that the result type t does not contain any pattern-bound variables.

Case (T-App):

The last premise ensures the result by construction. This ensures that non-values do not escape into types.

Case (T-Abs):

The first two premises ensure that the ascribed type is well-formed.

The cases of (B1) and (C1) are all straightforward. □

Lemma 13 (Substitution)**.** Lemma (Substitution for typing judgment):

```
  For all S G1 G2 X X' X'' x tx e t v p q tx' Phi,
    (A0) X'=x or X'={}
    (A1) S;G1,x:(p,tx),G2;X,X',X'' well-formed
    (A2) S;G1,x:(p,tx),G2;X,X',X'' |-_q e : t
    (A3) S;G1;- |-_p v_p : tx
    (A4) substitution s=[v_p/x]
    (A5) tx={x:tx' | Phi} => exists v', S;G1, refinements(G1);- |-_p v' : Phi[v_p/x]
    => S;G1,s(G2);X,X'' |-_q s(e) : s(t)
```

Lemma (Substitution for kinding judgment):

```
  For all S G1 G2 x tx t k v p tx' Phi,
    (B1) S;G1,x:(p,tx),G2;- well-formed
    (B2) S;G1,x:(p,tx),G2 |- t :: k
    (B3) S;G1;- |-_p v_p : tx
    (B4) substitution s=[v_p/x]
    (B5) tx={x:tx' | Phi} => exists v', S;G1, refinements(G1);- |-_p v' : Phi[v_p/x]
    => S;G1,s(G2) |- s(t) :: k
```

Lemma (Substitution for type conversion):

```
  For all S G1 G2 x tx t v p tx' Phi,
    (C1) S;G1,x:(p,tx),G2;- well-formed
    (C2) S;G1,x:(p,tx),G2 |- t <: t'
    (C3) S;G1;- |-_p v_p : tx
    (C4) substitution s=[v_p/x]
    (C5) tx={x:tx' | Phi} => exists v', S;G1, refinements(G1);- |-_p v' : Phi[v_p/x]
    => S;G1,s(G2) |- s(t) <: s(t')
```

Lemma (Substitution for type equivalence):

```
  For all S G1 G2 x tx t v p tx' Phi,
    (D1) S;G1,x:(p,tx),G2;- well-formed
    (D2) S;G1,x:(p,tx),G2 |- t ~ t'
    (D3) S;G1;- |-_p v_p : tx
    (D4) substitution s=[v_p/x]
    (D5) tx={x:tx' | Phi} => exists v', S;G1, refinements(G1);- |-_p v' : Phi[v_p/x]
    => S;G1,s(G2) |- s(t) ~ s(t')
```

Proof. By mutual induction on the structure of (A2), (B2), (C2), (D2), generalizing on q, the tail of the environment G2, and the set of affine assumptions X,X',X".

Cases of (A2)

```
Case (T-Var):

        G=G1,x:(p,tx),G2                        (A2.1)
        G(y) = (q,ty)                           (A2.2)
        S; G |- ty :: *                         (A2.3)
     ------------------------------------------ [T-Var]
        S; G; - |-_q  y : ty

  We consider two sub-cases, depending on whether x=y.

  Sub-case (x<>y):

     In this case, we have s(y)=y. We have two further sub-cases, depending on whether y in dom(G1)
     or y in dom(G2).

     Sub-sub-case (y in dom(G1)):

        From the well-formedness of S;G1,x:(p,tx);- and G1(y)=(q,ty), we can conclude that x not in
        FV(ty). Thus, s(ty)=ty, and we have, as required:
```

```
S;G1,s(G2);- |-_q s(y) : s(ty)
```

Sub-sub-case (y in dom(G2)):

We have y in dom(G2) and

```
G2(y)=(q,ty) => s(G2(y))=(q, s(ty)).
```

The conclusion is immediate.

Sub-case (x=y):

In this case, G(y) = G(x) = (p, tx) = (q, ty), and s(y)=v_p.

For the conclusion, we apply Lemma 5 (weakening). In order to do this, we must first show that

```
(WF)         S;G1,s(G2);X is well-formed
```

This is easily accomplished by induction on the length of G2, noting that x not in FV(s(G2)), and x not in X.

Now, using Lemma 5, WF and assumption (A3), we conclude

```
(Goal.0)    S;G1,s(G2);X |-_p v_p : tx
```

Finally, for the goal, we note that p=q, we use Lemma 6 (Well-kinded typings) on assumption (A3) to establish that (S;G1;- |- tx :: k) and hence that x not in FV(tx), and finally that s(tx) = tx to get:

```
(Goal)    S;G1,s(G2);X |-_q v_p : s(tx)
```

Case (T-Var-A):

Identical to (T-Var)

Case (T-Abs):

```
   G=G1,x:(p,tx),G2                     (A2.1)
   k in {*, A}                          (A2.2)
   S; G |- t :: k                       (A2.3)
   S; G,y:(q,t); X,X',X'',y |-_q e : t' (A2.4)
   X={} => tf = (y:t) -> t'             (A2.5)
   X<>{} => tf = !((y:t) -> t')         (A2.6)
 -------------------------------------- [T-Abs]
    S; G; X,X',X'' |-_q \y:t.e : tf
```

We begin by applying the mutual induction hypothesis for substitution on the kinding judgment to (A2.3) to establish

```
(A2.3')      S;G1,s(G2) |- s(t) :: k
```

Next, we apply the induction hypothesis (havng generalized on the tail of the environment G2, and X,X'X'') to obtain

```
(A2.4')     S; G1,s(G2),y:(q,s(t)); X,X'',y |-_q s(e) : s(t')
```

We can now construct

```
(A2')       S;G1,s(G2);X,X'' |-_q s(\y:t.e) : ?(y:s(t)) -> s(t')
```

using (T-Abs) with (A2.3') and (A2.4') in the premises.

We now consider three sub-cases

```
Sub-case (X'=x and X,X''={}):

   In this case, we have:
           s(tf) = !(y:s(t)) -> s(t')
   while
           ?(y:s(t)) -> s(t') = (y:s(t)) -> s(t')

   However, s(\y:t.e) is a syntactic q-value. So, to re-establish the appropriate affinity
   qualifier, we construct the goal by using (T-AVal) with (A2') in the premise.

Sub-cases (X'={} or X,X''<>{}):

   In both these cases,

          s(tf) = ?(y:s(t)) -> s(t')

   and (A2') directly satisfies the goal.
```

```
Case (T-Univ):

       G=G1,x:(p,tx),G2                (A2.1)
       k in {*, A}                     (A2.2)
       S; G, 'a::k; X |-_q e : t       (A2.3)
       X={} => tf = \/'a::k.t          (A2.4)
       X<>{} => tf = !(\/'a::k.t)      (A2.5)
     -------------------------------- [T-Univ]
       S; G; X |-_q /\'a::k.e : tf


  Similar to (T-Abs):

   We begin by applying the induction hypothesis (having generalized on the tail of the environment
   G2, and X,X',X'') to obtain

      (A2.3')     S; G1,s(G2),'a::k; X,X'' |-_q s(e) : s(t)

   We can now construct

      (A2')       S;G1,s(G2);X,X'' |-_q s(/\'a::k.e) : ?(\/'a::k.t)

   using (T-Univ) with (A2.3') in the premises.

   We now consider three sub-cases

   Sub-case (X'=x and X,X''={}):

      In this case, we have:
              s(tf) = !(\/'a::k.t)
      while
              ?(\/'a::k.t) = \/'a::k.t

      However, s(/\'a::k.e) is a syntactic q-value. So, to re-establish the appropriate affinity
      qualifier, we construct the goal by using (T-AVal) with (A2') in the premise.

   Sub-cases (X'={} or X,X''<>{}):

      In both these cases,

             s(tf) = ?(\/'a::k.t)

      and (A2') directly satisfies the goal.
```

Case (T-Fix):

```
      G=G1,x:(p,tx),G2                 (A2.1)
      S; G |- t :: *                   (A2.2)
      S; G, f:(q,t); - |-_q  e : t     (A2.3)
    --------------------------------- [T-Fix]
      S; G; - |-_q fix f:t.e : t
```

Similar to T-Abs.

Case (T-App):

```
      X1,X2 = X,X',X''
      G=G1,x:(p,tx),G2                       (A2.1)
      S; G; X1 |-_q e1 : ?(y:t1) -> t2       (A2.2)
      S; G; X2 |-_q  e2 : t1                 (A2.3)
      S; G |- t2 [e2/y] :: k                 (A2.4)
    -------------------------------------------- [T-App]
      S; G; X,X',X'' |-_q e1 e2 : t2[e2/y]
```

Induction hypothesis on (A2.2) and (A2.3) gives us

```
(A2.2')    S; G1,s(G2); X1 |-_q s(e1) : s(?(y:t1) -> t2)
(A2.3')    S; G; X2 |-_q  e2 : t1
```

Note that on splitting X,X',X'' into X1,X2, the assumption x in X' (if present) goes either in X1 or in X2, or in neither (if it is absent). In each case, we can use the induction hypothesis with either the left side of premise (A0) or the right side of (A0).

From the induction hypothesis on (A2.4) we get

```
(A2.4.1)    S; G1,s(G2) |-  s(t2[e2/y]) :: k
```

However, for the conclusion, we require

```
(A2.4')   S; G1,s(G2) |-  s(t2)[s(e2)/y] :: k
```

which requires showing

```
      s(t2[e2/y]) = s(t2)[s(e2)/y])
```

which following from the observation that x<>y (y is a bound variable and can be alpha renamed appropriately), and furthermore that y not in FV(range(s)).

For the conclusion, we apply (T-App) with (A2.2'), (A2.3') and (A2.4') in the premises.

Case (T-TApp):

```
      S; G; X,X',X'' |-_q e : ?(\/'a::k.t')
      S; G |- t :: k
    ------------------------------------ [T-TApp]
      S; G; X,X' |-_q e t : t' [t/'a]
```

Similar to (T-App), except there is no need for any special management of the affine assumptions.

Case (T-Bracket):

```
      G=G1,x:(p,tx),G2                 (A2.1)
      S; G; X,X',X'' |-_r e : t        (A2.2)
    ---------------------------------- [T-Bracket]
```

```
      S; G; X,X',X'' |-_q <e>_r : t
```

Having generalized on the index q on the turnstile of (A2), we can apply the induction hypothesis to (A2.2) to obtain

```
   (A2.2')    S; G1,s(G2); X,X'' |-_r s(e) : s(t)
```

The conclusion follows from an application of (T-Bracket) with (A2.2') in the premise.

Case (T-Match):

```
      G=G1,x:(p,tx),G2                                             (A2.1)
      X1,X2 = X,X',X''                                             (A2.2)
      X3 < x1..xn                                                  (A2.3)
      S; G; X1 |-_q e : t'                                         (A2.4)
      S; G, xi:(q,ti), xi=vi;X3 |-_q D t1..tn x1..xn : t'          (A2.5)
      S; G, xi:(q,ti), xi=vi, e=D t1..tn x1..xn; X2,X3 |-_q e1 : t   (A2.6)
      S; G; X2 |-_q e2 : t                                         (A2.7)
    ---------------------------------------------------------------- [T-Match]
      S;G;X1,X2 |-_q match e with
                    D t1..tn x1..xn -> e1        :  t
                    else e2
```

As in (T-App), the affine assumption in X', if present, either goes to X1 or X2. When using the induction hypothesis, we satisfy premise (A0) by using either side of the disjunct, depending on whether x in X1, X2 or neither.

We use the induction hypothesis to establish

```
  (A2.4')    S; G1,s(G2); X1 |-_q s(e) : s(t')
  (A2.5')    S; G1,s(G2),s(xi:(q,ti)),s(xi=vi); X3 |-_q s(D t1..tn x1..xn) : s(t')
  (A2.6')    S; G1,s(G2),s(xi:(q,ti)),s(xi=vi); s(e=D t1..tn x1..xn); X2,X3 |-_q s(e1) : s(t)
  (A2.7')    S; G1,s(G2); X2 |-_q s(e2) : s(t)
```

and use each of these in the premises of (T-Match) for the goal.

Case (T-Sub):

```
      G=G1,x:(p,tx),G2              (A2.1)
      S; G; X,X',X'' |-_q e : t'    (A2.2)
      S; G |- t' <: t               (A2.3)
    ------------------------------- [T-Sub]
      S; G; X,X',X'' |-_q  e : t
```

From the induction hypothesis we get

```
   (A2.2')     S; G1,s(G2); X,X'' |-_q s(e) : s(t')
```

From the mutual induction hypothesis with Lemma 7.3 (substition for type conversion), we get

```
   (A2.3')     S;G1,s(G2) |- s(t') <: s(t)
```

The conclusion follows from an application of (T-Sub) with (A2.2') and (A2.3') in the premises.

Cases (T-Val-A, T-Drop-A, T-Datacon):

Trivial, from the induction hypothesis.

Cases of (B2)

Case (K-Var):

Trivial, since G1,G2 binds 'a and kinds have no free variables.

Case (K-Constr):

Trivial, since type constructors are bound in S and S is unchanged.

Case (K-Bang):

Follows from the induction hypothesis.

Case (K-Arrow):

```
        G=G1,x:(p,tx),G2              (B2.1)
        k,k' in {*, A}                (B2.2)
        S; G |- t1 :: k               (B2.3)
        S;G,x:t1 |- t2 :: k'          (B2.4)
      ---------------------------- [K-Arrow]
         S; G |- (x:t1) -> t2
```

From the induction hypothesis, we have

```
(B2.3')   S;G1,s(G2) |- s(t1) :: k
```

From the induction hypothesis, having generalized on the tail of the environment G2, we have

```
(B2.4')   S;G1,s(G2),x:s(t1) |- s(t2) :: k'
```

For the conclusion, we can use (K-Arrow) with (B2.3') and (B2.4') in the premises.

Case (K-Univ):

Similar to (K-Arrow), relies on generalization over G2.

Case (K-App):

Straightforward, from induction hypothesis applied to each premise.

Case (K-Dep):

```
        G=G1,x:(p,tx),G2              (B2.1)
        S; G |- t :: t' -> k          (B2.2)
        S; G; . |-_q v_q : t'         (B2.3)
      ---------------------------- [K-Dep]
         S; G |- t v_q : k
```

From the induction hypothesis on (B2.2) we get

```
  (B2.2')    S; G1,s(G2) |- s(t) :: t' -> k
```

From Lemma 6.2, (Well-formed kindings), we have that

```
    S |- t' -> k
```

from which we can conclude that FreeVars(t' -> k) = {}, and hence s(t') = t'.

Next, we apply the mutual induction hypothesis on the typing judgment to (B2.3) to produce

```
  (B2.3.1)   S'; G1,s(G2); - |-_q s(v_q) : s(t')
```

Or,

```
(B2.3')   S'; G1,s(G2); - |-_q s(v_q) : t'
```

For the conclusion, we apply (K-Dep) with (B2.2') and (B2.3') in the premises.

Case (K-Refine):

Similar to (K-Arrow), since Phi is simply a type.

Cases of (C2)

Case (TC-Equiv):

By the mutual induction hypothesis on Lemma 7.4, substitution for the type-equivalence judgement.

Case (TC-Trans):

By the induction hypothesis on each premise.

Case (TC-A):

By the induction hypothesis on the premise.

Case (TC-Refine-1):

Immediate.

Case (TC-Refine-2):

Immediate.

Case (TC-Refine-3)

```
   G=G1,x:(p,tx),G2                          (C2.1)
   S; G |- {y:t | Phi'} :: *                 (C2.2)
   S; G |- t <: t'                           (C2.3)
   G'=refinements(G, y:{y:t | Phi})          (C2.4)
   S; G, y:{y:t | Phi},G';_ |-_q v_q : Phi'   (C2.5)
 --------------------------------------------- [TC-Refine-3]
   S; G |- {y:t | Phi} <: {y:t' | Phi'}
```

Our goal is to show that

```
S; G1,s(G2) |- {y:s{t} | s(Phi)} <: {y:s(t') | s(Phi')}
```

From the mutual induction hypothesis with Lemma 7.2, we have

```
(C2.2')  S; G1,s(G2) |- {y:s(t) | s(Phi')} :: *
```

From the induction hypothesis we have

```
(C2.3')  S; G1,s(G2) |- s(t) <: s(t')
```

From the definition of refinements, we have

```
refinements(G) = G',y':Phi,G''
```

We have C2.5

```
S;G1,x:(p,tx),G2,G',y':Phi,G''; - |- v_q : Phi'
```

First, we use a single application of the mutual induction hypothesis from Lemma 7.1 (substitution for typing), to construct:

```
(C2.3.1)   S;G1,s(G2,G',y':Phi',G''); - |- s(v_q) : s(Phi)
```

However, this is not sufficient for the conclusion, since we need

```
S;G1,s(G2,G',G''); - |- v'' : s(Phi)
```

To construct this, we first use assumption (C5) to produce a witness v

```
(C5)   S;G1;- |-_q v : s(Phi')
```

Next, Lemma 5 (weakening), to construct

```
(C5') S;G1,s(G2,G'); - |-_q v : s(Phi')
```

Finally, we use the mutual induction hypothesis from Lemma 7.1, to construct, substitution s'= [v/x'], and

```
(C2.3.2)    S;G1,s(G2,G'),s'(s(G'')); - |- s'(s(e)) : s'(s(Phi))
```

From, well-formedness of of type conversions, Lemma 6.3, and C2.2 we have that y' not in FV(s(Phi)).

Similarly, from the well-formedness of G and from the definition of refinement(G), we can conclude that y notin FV(G'') subseteq FV(s(G'')).

Thus, we have

```
(C2.3')    S;G1,s(G2,G'),s(G''); - |- s'(s(e)) : s(Phi)
```

Finally, for the conclusion, we apply TC-Refine-3 with (C2.1'), (C2.2') and (C2.3') in the premises.

Cases of (D2)

Case (EE-Id):

Trivial

Case (EE-Refine);

We consider two sub-cases depending on whether the used match assumption appears in G1 or G2.

Sub-case (match assumption in G1):

```
----------------------------------------- [EE-Refine]
  S; G1',e=e',x:(p,tx),G2 |- e ~ e'
```

From the well-formedness of the environment, the x not in FreeVars(e, e'). Thus s(e) = e, s(e')=e' and the goal follows.

```
(Goal) S;G1',e=e', s(G2) |- s(e) ~ s(e)'
```

Sub-case (match assumption in G2):

```
----------------------------------------- [EE-Refine]
  S; G1,x:(p,tx),e=e'G2 G' |- e ~ e'
```

The goal follows immediately.

```
(Goal) S;G1',s(e)=s(e'), s(G2) |- s(e) ~ s(e)'
```

□

Lemma 14 (Type substitution). Lemma (Type substitution for typing judgment):

```
  For all S G1 G2 X e t t1 k 'a p,
    (A1) S;G1,'a::k,G2;X  well-formed
    (A2) S;G1,'a::k,G2;X |-_p e : t
    (A3) S;G1 |- t' :: k
    (A4) substitution s=[t1/'a]
    => S;G1,s(G2);X |-_p s(e) : s(t)
```

```
Lemma (Type substitution for kinding judgment):

  For all S G1 G2 t1 k1 t2 k2 'a p,
    (B1) S;G1,'a::k2,G2;-  well-formed
    (B2) S;G1,'a::k2,G2 |- t :: k1
    (B3) S;G1 |- t2 :: k2
    (B4) substitution s=[t2/'a]
    => S;G1,s(G2) |-_p s(t) :: k1
```

```
Lemma (Type substitution for type conversion):

  For all S G1 G2 t t' t2 k2 'a p,
    (B1) S;G1,'a::k2,G2;-  well-formed
    (B2) S;G1,'a::k2,G2 |- t <: t'
    (B3) S;G1 |- t2 :: k2
    (B4) substitution s=[t2/'a]
    => S;G1,s(G2) |- s(t) <: s(t')
```

```
Lemma (Type substitution for type equivalence):

  For all S G1 G2 t t' t2 k2 'a p,
    (B1) S;G1,'a::k2,G2;-  well-formed
    (B2) S;G1,'a::k2,G2 |- t ~ t'
    (B3) S;G1 |- t2 :: k2
    (B4) substitution s=[t2/'a]
    => S;G1,s(G2) |- s(t) ~ s(t')
```

Proof. Straightforward mutual induction on the structure of (A2), (B2), (C2), (D2). □

Lemma 15 (Strengthening for inaccessible affine assumptions). For all S G G' X x p t e q t',

```
   (A1) S;G,x:(p,t),G'; X |-_q e : t'
   (A2) x not in X
   (A3) S;G;- |- t :: A
   =>   S;G,G';X |-_q e : t'
```

Proof.
```
Observation 1:  S;G,G';X is well-formed, since
                -- x not in FV(G), by well-formedness of S;G;-

                -- x not in FV(G'), since form (A2), x is affine
                   And, from well-formedness of kinds,
                   forall S,G,t,k. S;G |- t :: k => x \notin FV(t)

By induction on the structure of (A1), noting that (T-Var-A) requires x in X.
```
□

Corollary 16 (Strengthening for affine assumptions in kinding). For all S G G' t t' k,

```
   (A1) S;G,x:(p,t),G' |- t' :: k
   (A2) S;G |- t :: A
   =>   S;G,G';|- t' :: k
```

Lemma 17 (Destruction of affine assumption). For all S M p e t M' x,

```
   (A1) S;G(M);X |-_p e : t
   (A2) (M,e) ~p~-> (M',e')
   (A3) x in dom(M) /\ x notin dom(M')
  =>    x in X
```

Lemma 18 (Construction of affine assumption). For all S M p e t M' x,

```
   (A1) S;G(M);X |-_p e : t
```

```
  (A2) (M,e) ~p~-> (M',e')
  (A3) x in dom(M') /\ x notin dom(M)
 =>    FV(e') \subseteq X,x
```

Proof. Simple induction on the structure of (A2), noting from well-formed memory that values in the store are always closed. ☐

Lemma 19 (Redundant match assumptions)**.** `For all S M X p e t v,`

```
  (A1) S;G, v=v, G';X |-_p e : t
 =>    S;G,G';X |-_p e : t
```

Proof. Straightforward by noting that every application of (EE-Natch) can be replaced by (EE-Id). ☐

Lemma 20 (Proofs of refinement formulas)**.** `For all S G v t p Phi,`

```
   (A1) S;G;- |-_p v : {x:t | Phi}
    => exists v', S;G;- |-_bot v' : Phi[v/x]

where S;G;- |- x:{y:t' | Phi'} => S;G,y:t';- |- \hat{x} : Phi'
```

Proof.

```
By induction on the structure of (A1).

The point to note is that the only way to introduce a refined typpe {x:t | Phi} is with an
application of (T-Refine).

Importantly, from the well-formedness of S, we have that every data constructor application
introduces a non-refined type. Thus an application of (T-App) never produces a refined type.
```

☐

Theorem 21 (Subject reduction)**.** `For all S M M' X e t p,`

```
 (A0) S;G(M);X well-formed
 (A1) S;G(M);X |-_p e : t
 (A2) (M,e) ~p~> (M',e')
  =>  exists X'. S;G(M');X' |-_p e' : t /\
                 X' = X U (dom M' \ dom M) if dom M' >= dom M
                 X' = X \ (dom M \ dom M') otherwise
```

Proof. By induction on the structure of the typing derivation (A1).

```
Cases (T-Sub, T-Drop-A):

 Induction hypothesis.

Case (T-Var):

 Impossible, from the definition of well-formed memory, x is of kind A.

Case (T-Var-A):

 x steps using (E-Destruct) to v_q and, from the well-formedness of memory M, we have the result.


Case (T-AVal, T-Datacon, T-Abs, T-Univ):

 Irreducible.

Case (T-Fix):

      unrefined t
      S; G |- t :: *                    (A1.1)
      S; G, f:(p,t); - |-_p v_p : t     (A1.2)
    ----------------------------------- [T-Fix]
      S; G; - |-_p fix f:t.v_p : t


 Inversion of (A2) gives an application of (E-Fix):
```

```
  ------------------------------------------------- [E-Fix]
     fix f:t.v_p ~p~>  v_p[(v_p[fix f:t.v_p / f])/f]
```

From an application of the substitution lemma, Lemma 7, we get

```
     S;G;- |-_p v_p [fix f:t.v_p / f] : s(t)
```

From, well-kinding of typing (Lemma 6), we have that f not in dom(t). Thus, s(t) = t

A second application of the substitution lemma, Lemma 7, gives

```
     S;G;- |-_p v_p[(v_p[fix f:t.v_p /f])/x] : t
```

Case (T-App):

```
     S; G; X1 |-_p e1 : ?(x:t1) -> t2    (A1.1)
     S; G; X2 |-_p e2 : t1               (A1.2)
     S; G |- t2 [e2/x] :: k             (A1.3)
   ------------------------------------ [T-App]
     S; G; X1,X2 |-_p e1 e2 : t2[e2/x]
```

---Subcase: e1 is a non-value, reduction proceeds using (E-Cong) to

```
     (M, e1 e2) ~p~> (M', e1' e2)
```

From the induction hypothesis, we get

```
     S;G(M');X1' |-_p e1' : ?(x:t1) -> t2     (G1.1)
```

------Sub-subcase: if X1=X then use (T-App) with (G1.1), (A1.2) and (A1.3)

------Sub-Subcase: if X1' = X1,x, then dom(M') = X1,x,X2,X

For the conclusion, we use (T-App) with (G1.1) with (A1.2) and weakening on (A1.3).

------Sub-Subcase: if X1',x = X1, then, noting that x not in X2, we use (T-App) with (G1.1), and strengthening on inaccessible affine assumptions on (A1.2), and strengthening for affine assumptions in kinding for (A1.3).

---Subcase: e2 is a non-value, similar.

---Subcase: e1=v1 and e2=v2 are values.

We first use the canonical forms lemma to establish (v1 = \x:t1.e) and inversion of (A2) gives us an application of (E-Beta).

```
   ----------------------------- [E-Beta]
      \x:t1.e v2 ~p~> e[v2/x]
```

From an inversion of (A1.1), we get an application of (T-Abs) with

```
 (A1.1.1)       S;G,x:t1;X,x |-_p e : t2
```

Now, using (A1.1.1) and (A1.2), we apply the substitution lemma, to derive the goal

```
                S;G;X |-_p e[v2/x] : t2[v2/x]
```

Case (T-TApp):

Similar to (T-App), using the induction hypothesis in the first premise when e is reducible.

And, using canonical forms and the type-substitution lemma when reduction is via E-TBeta.

Case (T-Match):

```
    ----------------------------------------------------------------- (Defn. of pattern matching)
       D t1...tn v1...vn ~=~ D t1...tn x1..xn : (x1,v1)...(xn,vn)
```

If the discriminant steps (M,v_x) ~p~> (M',v) via (E-Cong), then, the context splitting rules, and strengthening of affine assumptions allows us to reason similarly to (E-App).

If a step is taken to the false branch via (E-Match), then, the conclusion follows using the last premise of (A1), with (T-Drop) to introduce unused affine assumptions, if any.

If a step is taken to the true branch via (E-Match), then from the definition of pattern matching above, and the repeated application of the substitution lemma, we get

```
S;G, v1=v1, ..., vn=vn, (D t1..tn v1..vn = D t1..tn v1..vn); X |- e1: sigma(t),
```

where dom(sigma) = x1..xn, the pattern variables.

From repeated use of the redundant match assumptions lemma, we arrive at

```
S;G; X |- e1: sigma(t)
```

Finally, from well-kinded typings lemma applied to the last premise, we get that sigma(t) = t, since FV(t) does not include any of the pattern variables.

Case (T-Bracket):

By inversion on (A2), we have one of several cases.

---Subcase (A2 is E-Br): Straightforward from induction hypothesis.

---Subcase (A2 is E-Strip): Use the premise of A1 for the conclusion.

---Subcase (A2 is E-Nest): Use T-Bracket with the nested premise of (A1) for the conclusion.

---Subcase (A2 is E-Extrude):

From inversion of (A1), we get:

```
S;G,x:(q,t);X,x |-_q e : t'      (A1.1)
```

By weakening (A1.1), we get

```
S;G,y:(p,t),x:(q,t);X,y,x |-_q e : t'      (A1.2)
```

We have that

```
S;G,y:(p,t);X,y |-_q <y>_p : t             (A1.3)
```

And <y>_p is a q-value.

So, from the substitution lemma, we get

```
S;G,y:(p,t);X,y |-_q e [<y>_p/x] : t'       (G1.1)
```

For the conclusion, we apply (T-Fun) with (G1.1) in the premise.

---Subcase (A2 is E-TExtrude):

We use (T-Tabs), with (T-Bracket) with the nested premise of (A1) for the conclusion.

□

B. Value abstraction for FINE

Theorem 22 (Value abstraction)**.**

```
forall S e t x p q tx v1_p v2_p.
  (A0) S;x:(p,tx);x well-formed
  (A1) S;x:(p,tx);x |-_q e : t
  (A2) q < p and e is a non-value free of <.>_r brackets, where r >= p, except <x>_p
  (A3) forall i. S;-;- |-_p vi_p : tx
  (A4) e[v1_p/x] ~q~> e1
=> exists e2. e2[v1_p/x] = e1 /\ e[v2_p/x] ~q~> e2[v2_p/x]
```

Proof. By induction on the structure of (A4). (Note the restriction to the pure fragment)

```
Case (E-Bracket):

                       e[v1_p/x] ~r~> e'
                ---------------------------------------- [E-Bracket]
                   (<e>_r)[v1_p/x] ~q~> <e'>_r

  e is free of <.>_p brackets, so, either

 Sub-case 1: (r < p): From the premise, and from the IH, we get

         e2[v1_p/x] = e1   and   e[v2_p/x] ~r~> e2[v2_p/x]

 Sub-case 2: (r=p and e=v1_p): Impossible, since v1_p is a p-value and is irreducible.


Case (E-Beta):

 From the definition of substitution, and alpha-converting the left-subterm to ensure that the bound
 var is distinct, we get:

        ----------------------------------------------------------- [E-Beta]
          (\y:t.e v_q)[v1_p/x] ~q~> (e[v1_p/x]) [v_q [v1_p/x]/y]

 For the conclusion, we construct:  e2 = e [v_q/y] and both

       (e[v1_p/x]) [v_q [v1_p/x]/y] = e2[v1_p/x]
 and
       (\y:t.e v_q)[v2_p/x] ~q~> e2[v2_p/x]

 are immediate.


Case (E-TBeta):

 Similar to the previous case, using the definition of substitution.


Case (E-Fix):

        --------------------------------------------------------- [E-Fix]
          (fix f:t.v_q)[v1_p/x] ~q~>
          (v[v1_p/x])[((v[v1_p/x])[fix f:t.v_q[v1_p/x] / f])/f]

 Again, similar to the previous two cases, following from the definition of substitution.


Case (E-Match1):

                   v_q[v1_p/x] ~=~ D t1...tn x1..xn : theta
      ------------------------------------------------------------------------------- [E-Match1]
          match (v_q)[v1_p/x] with D t1..tn x1..xn -> e1[v1_p/x] else e2[v1_p/x]
```

```
          ~p~> theta(e1[v1_p/x])
```

Sub-case 1: v_q = <x>_p

Impossible, since from the definition of (~=~), <v1_p>_p does not match any pattern

Sub-case 2: v_q = D t1 ... tn v1_q ... vn_q

Case (E-Match2): Similar

Case (E-Extrude):

By the definition of substitution, we have

<\y:t.e>_q [v1_p/x] = <\y:t. e[v1_p/x]>_q

```
    ------------------------------------------------------------- [E-Extrude]
         <\y:t. e[v1_p/x]>_q' ~q~> \z:t.<e[v1_p/x][<z>_q/y]>_q'
```

Pick e2 = \z:t<e [<z>_q/y]>_q'

We have e2[v1_p/x] = \z:t<e [<z>_q/y] [v1_p/x]>_q', which from z<>x gives the desired result.

Additionally

```
         <\y:t. e[v2_p/x]>_q' ~q~> e2[v2_p/x]
```

Case (E-TExtrude): Similar, but simpler since the type variable 'a is not wrapped.

Case (E-Strip):

```
   -------------------------------------- (E-Strip)
     <v_q>_q [v1_p/x] ~q~> v_q[v1_p/x]
```

Sub-case q=p.

From our assumption of p-bracket freedom, we have that v_q must be x. Thus, we have

<x>_p [v1_p/x] ~p~> v1_p

So, we pick e2=x, and the conclusion is immediate.

Sub-case q<>p.

Pick e2 = v_q.

Case (E-Nest): As in the previous case.

□

$S;\Gamma \hookrightarrow S;\Gamma'$	Translation of environments

$$\frac{\cdot \vdash S \hookrightarrow S' \quad S';\cdot \vdash \Gamma \hookrightarrow \Gamma'}{S;\Gamma \hookrightarrow S';\Gamma'}$$

$$\frac{S_0 \vdash \kappa \hookrightarrow \kappa' \quad S_0, \tau{::}\kappa' \vdash S \hookrightarrow S'}{S_0 \vdash \tau{::}\kappa', S \hookrightarrow \tau{::}\kappa', S'}$$

$$\frac{S_0;\cdot \vdash \tau \hookrightarrow \tau' :: K \quad S_0, D{:}\tau' \vdash S \hookrightarrow S'}{S_0 \vdash D{:}(p,\tau), S \hookrightarrow D{:}(p,\tau'), S'}$$

$$\frac{S_0, p \sqsubseteq q \vdash S \hookrightarrow S'}{S_0 \vdash p \sqsubseteq q, S \hookrightarrow p \sqsubseteq q, S'}$$

$$\frac{S_0, p \sqsubseteq q \vdash S \hookrightarrow S'}{S_0 \vdash p \sqsubseteq q, S \hookrightarrow p \sqsubseteq q, S'}$$

$$\frac{S \vdash \kappa \hookrightarrow \kappa' \quad S;\Gamma_0, \tau{::}\kappa' \vdash \Gamma \hookrightarrow \Gamma'}{S;\Gamma_0 \vdash \tau{::}\kappa', \Gamma \hookrightarrow \tau{::}\kappa', \Gamma'}$$

$$\frac{S;\Gamma_0 \vdash \tau \hookrightarrow \tau' :: k \quad S;\Gamma_0, x{:}(p,\tau') \vdash \Gamma \hookrightarrow \Gamma'}{S;\Gamma_0 \vdash x{:}(p,\tau), \Gamma \hookrightarrow x{:}(p,\tau'), \Gamma'}$$

$$\frac{\begin{array}{c} S;\Gamma_0 \vdash \tau \hookrightarrow (x{:}\tau' * \phi)_k :: K \\ S;\Gamma_0, x{:}(p,\tau'), x'{:}(p, \mathsf{proof}\ \phi) \vdash \Gamma \hookrightarrow \Gamma' \end{array}}{S;\Gamma_0 \vdash x{:}(p,\tau), \Gamma \hookrightarrow x{:}(p,\tau'), x'{:}(p, \mathsf{proof}\ \phi), \Gamma'}$$

$$\frac{\forall i. S;\Gamma_0 \vdash v_i \overset{k}{\hookrightarrow} v_i' \quad S;\Gamma_0, v_1' \doteq v_2' \vdash \Gamma \hookrightarrow \Gamma'}{S;\Gamma_0 \vdash v_1 \doteq v_2, \Gamma \hookrightarrow v_1' \doteq v_2', \Gamma'}$$

$S \vdash \kappa \hookrightarrow \kappa'$	Translation of kinds

$$S \vdash \star \hookrightarrow \star \text{ (XK-S)} \qquad S \vdash \mathtt{A} \hookrightarrow \mathtt{A} \text{ (XK-A)} \qquad \frac{S \vdash \kappa \hookrightarrow \kappa'}{S \vdash k \to \kappa \hookrightarrow k \to \kappa'} \text{ (XK-Tc)} \qquad \frac{S;\cdot \vdash \tau \overset{min}{\hookrightarrow} \tau' :: K \quad S \vdash \kappa \hookrightarrow \kappa'}{S \vdash \tau \to \kappa \hookrightarrow \tau' \to \kappa'} \text{ (XK-DTc)}$$

Figure 11. Translation of environments and kinds

C. Derefinement of FINE

Lemma 23 (Determinism of translation)**.**

$$\forall S, \kappa, \kappa_1, \kappa_2. S \vdash \kappa \hookrightarrow \kappa_1 \ \wedge \ S \vdash \kappa \hookrightarrow \kappa_2 \ \Rightarrow \ \kappa_1 = \kappa_2$$

$$\forall S, \Gamma, \tau, \tau_1, \tau_2, \kappa_1, \kappa_2. S;\Gamma \vdash \tau \hookrightarrow \tau_1 :: \kappa_1 \ \wedge \ S;\Gamma \vdash \tau \hookrightarrow \tau_2 :: \kappa_2 \ \Rightarrow \ \kappa_1 = \kappa_2$$

$$\forall S, \Gamma, \tau, \tau_1, \tau_2, K. \textit{well-formed}(S;G) \ \wedge \ S;\Gamma \vdash \tau \hookrightarrow \tau_1 :: K \ \wedge \ S;\Gamma \vdash \tau \hookrightarrow \tau_2 :: K \ \Rightarrow \ \tau_1 = \tau_2$$

$$\forall S, \Gamma, X, e, e_1, e_2, \tau, K. \textit{well-formed}(S;G;X) \ \wedge \ S;\Gamma;X \vdash e \overset{K}{\hookrightarrow} e_1 : \tau \ \wedge \ S;\Gamma;X \vdash e \overset{K}{\hookrightarrow} e_2 : \tau \ \Rightarrow \ e_1 = e_2$$

Lemma 24 (Derefinement of values)**.**

$$\forall S, \Gamma, S', \Gamma', X, p, v_p, \tau, \phi. S;\Gamma;X \vdash_p v_p : \tau \ \wedge \ S;\Gamma \hookrightarrow S';\Gamma' \Rightarrow \exists \tau', \phi', v_1, e_2. S';\Gamma';X \vdash_p v_p \overset{K}{\hookrightarrow} (x{:}(v_1{:}\tau'), (e_2{:}\phi'))_k : (x{:}\tau' * \phi')_k$$

Lemma 25 (Substitution lemma for translation)**.**

$$\begin{array}{l} \forall S, \Gamma_1, \Gamma_2, x, p, \tau, \tau_1, \kappa, \phi, \tau', v_p, v_p', e. S;\Gamma_1, x:(p,\tau), \Gamma_2 \hookrightarrow S';\Gamma_1', x:(p,\tau'), \_, \Gamma_2' \\ S;\Gamma_1, x:(p,\tau), \Gamma_2 \vdash \tau_1 :: \kappa \ \wedge \ S;\Gamma_1;\cdot \vdash_p v_p : \tau \ \wedge \\ S';\Gamma_1';\cdot \vdash_p v_p \overset{K}{\hookrightarrow} (x{:}(v_p'{:}\tau'), (e{:}\phi)) : (x{:}\tau' * \phi) \ \wedge \ S';\Gamma_1', x:\tau', y:\mathsf{proof}\phi, \Gamma_2' \vdash \tau_1 \hookrightarrow \tau_2 :: K \\ \Rightarrow \ S';\Gamma_1', (\Gamma_2')[v_p'/x] \vdash \tau_1[v_p'/x] \overset{K}{\hookrightarrow} \tau_1'[v_p'/x] :: K \end{array}$$

Lemma 26 (Equivalence of derefined types)**.**

$$\begin{array}{l} \forall S, \Gamma, S', \Gamma', \tau_1, \tau_2, \tau_1', \tau_2'. S;\Gamma \vdash \tau_1 \cong \tau_2 \ \wedge \ S;\Gamma \hookrightarrow S';\Gamma' \ \wedge \\ S';\Gamma' \vdash \tau_1 \overset{min}{\hookrightarrow} \tau_1' :: K \ \wedge \ S';\Gamma' \vdash \tau_2 \overset{min}{\hookrightarrow} \tau_2' :: K \\ \Rightarrow S';\Gamma' \vdash \tau_1' \cong \tau_2' \end{array}$$

Proof. A corollary of the determinism lemmas, by induction on the shape of the equivalence judgment. □

Lemma 27 (Translation of subtyping)**.**

$$\begin{array}{l} \forall S, \Gamma, S', \Gamma', \tau_1, \tau_2, e, e', \tau_1', \tau_2'. S;\Gamma \vdash \tau_1 <: \tau_2 \ \wedge \ S;\Gamma \hookrightarrow S';\Gamma' \ \wedge \\ S';\Gamma' \vdash \tau_1 \hookrightarrow \tau_1' :: K \ \wedge \ S';\Gamma' \vdash \tau_2 \hookrightarrow \tau_2' :: K \ \wedge \ S';\Gamma' \vdash e \hookrightarrow e' : \tau_1' \\ \Rightarrow \exists e''. S';\Gamma' \vdash e \hookrightarrow e'' : \tau_2' \end{array}$$

Proof. □

simple kinds $k ::= \star \mid \mathtt{A}$ kinds $\kappa ::= k \mid k \to \kappa \mid \tau \to \kappa$ pseudo kinds $K ::= \mathtt{box}(k) \mid \kappa$

$S;\Gamma \vdash \tau \hookrightarrow \tau' :: K$ where $(x{:}t_1 * t_2)_k = Q(k, \forall\alpha{::}k.\_{:}(x{:}t_1 \to y{:}\mathsf{proof}\ t_2 \to \alpha) \to \alpha)$, $S;\Gamma \vdash \tau \stackrel{min}{\hookrightarrow} \tau' :: K \wedge S;\Gamma \vdash \tau \stackrel{min}{\hookrightarrow} \tau' :: \kappa \Rightarrow K = \kappa$ **Translation of types**

$$\frac{}{S;\Gamma \vdash \alpha \hookrightarrow \alpha :: \Gamma(\alpha)}\ \text{(DK-Var)} \qquad \frac{}{S;\Gamma \vdash T \hookrightarrow T :: S(T)}\ \text{(DK-TC)} \qquad \frac{S;\Gamma \vdash \tau \hookrightarrow \tau' :: \star}{S;\Gamma \vdash !\tau \hookrightarrow !\tau' :: \mathtt{A}}\ \text{(DK-Afn)}$$

$$\frac{S;\Gamma,\alpha{::}k \vdash \tau \hookrightarrow \tau' :: k'}{S;\Gamma \vdash \forall\alpha{::}k.\tau \hookrightarrow \forall\alpha{::}k.\tau' :: \star}\ \text{(DK-Uni)} \qquad \frac{S;\Gamma \vdash \tau_1 \hookrightarrow (x{:}\tau_1' * \phi)_k :: \mathtt{box}(k) \quad S;\Gamma, x{:}(p,\tau_1'), y{:}(p,\phi) \vdash \tau_2 \stackrel{min}{\hookrightarrow} \tau_2' :: K}{S;\Gamma \vdash x{:}\tau_1 \to \tau_2 \hookrightarrow x{:}\tau_1' \to y{:}\phi \to \tau_2' :: \star}\ \text{(DK-Fun)}$$

$$\frac{S;\Gamma \vdash \tau_1 \hookrightarrow \tau_1' :: k \to \kappa \quad S;\Gamma \vdash \tau_2 \stackrel{min}{\hookrightarrow} \tau_2' :: K \quad \bar{K} = k}{S;\Gamma \vdash \tau_1\ \tau_2 \hookrightarrow \tau_1'\ \tau_2' :: \kappa}\ \text{(DK-App)} \qquad \frac{S;\Gamma \vdash \tau_1 \hookrightarrow \tau_1' :: \tau \to \kappa \quad S;\Gamma;\cdot \vdash_\top v_p \stackrel{K}{\hookrightarrow} v_p' : \tau}{S;\Gamma \vdash \tau_1\ v_p \hookrightarrow \tau_1'\ v_p' :: \kappa}\ \text{(DK-Dep)}$$

$$\frac{S;\Gamma \vdash \tau \hookrightarrow \tau' :: k \quad S;\Gamma, x{:}(p,\tau') \vdash \phi \hookrightarrow \phi' :: \star}{S;\Gamma \vdash \{x{:}\tau \mid \phi\} \hookrightarrow (x{:}\tau' * \phi')_k :: \mathtt{box}(k)}\ \text{(DK-DeRefine)} \qquad \frac{S;\Gamma \vdash \tau \hookrightarrow \tau' :: k}{S;\Gamma \vdash \tau \hookrightarrow (x{:}\tau' * \mathsf{True})_k :: \mathtt{box}(k)}\ \text{(DK-Box)}$$

$S;\Gamma;X \vdash e \stackrel{K}{\hookrightarrow} e' : \tau$ where $(x{:}(e{:}\tau), (e'{:}\phi))_k = \Lambda\alpha{::}k.\lambda f{:}(x{:}\tau \to \_{:}\mathsf{proof}\ \phi \to \alpha).(f\ e\ e')$ **Translation of terms**

$$\frac{S(D) = (p,\tau)}{S;\Gamma;\cdot \vdash_p D \stackrel{\star}{\hookrightarrow} D : \tau}\ \text{(DT-D)} \qquad \frac{S;\Gamma \vdash \tau' \hookrightarrow \tau :: K \quad \bar{K} = \mathtt{A} \Rightarrow x \in X}{S;\Gamma, x{:}(p,\tau), \Gamma'; X \vdash_p x \stackrel{K}{\hookrightarrow} x : \tau}\ \text{(DT-X)} \qquad \frac{q \sqsubseteq p \in S \quad S;\Gamma;X \vdash_q e \stackrel{K}{\hookrightarrow} e' : \tau}{S;\Gamma;X,X' \vdash_p e \stackrel{K}{\hookrightarrow} e' : \tau}\ \text{(DT-S)}$$

$$\frac{\begin{array}{c} S;\Gamma \vdash \tau_1 \hookrightarrow (x{:}\tau_1' * \phi)_k :: \mathtt{box}(k) \\ S;\Gamma, x{:}(p,\tau_1'), y{:}(p,\phi); X, x \vdash_p e \stackrel{K}{\hookrightarrow} e' : \tau_2 \end{array}}{S;\Gamma;X \vdash_p \lambda x{:}\tau_1.e \stackrel{k'}{\hookrightarrow} \lambda x{:}\tau_1'.\lambda y{:}\phi.e' : Q(X, x{:}\tau_1' \to y{:}\phi \to \tau_2')}\ \text{(DT-Fun)} \qquad \frac{\begin{array}{c} S;\Gamma \vdash \tau \hookrightarrow \tau' :: \star \\ S;\Gamma, f{:}(p,\tau'); \cdot \vdash_p v_p \stackrel{\star}{\hookrightarrow} v_p' : \tau' \end{array}}{S;\Gamma;\cdot \vdash_p \text{fix } f{:}\tau.v_p \stackrel{\star}{\hookrightarrow} \text{fix } f{:}\tau'.v_p' : \tau'}\ \text{(DT-Fix)}$$

$$\frac{S;\Gamma;X \vdash_q e \stackrel{K}{\hookrightarrow} e' : \tau}{S;\Gamma;X \vdash_p \langle e \rangle_q \hookrightarrow \langle e' \rangle_q : \tau}\ \text{(DT-Bracket)} \qquad \frac{S;\Gamma,\alpha{::}k; X \vdash_p e \stackrel{K}{\hookrightarrow} e' : \tau'}{S;\Gamma;X \vdash_p \Lambda\alpha{::}k.e \stackrel{k'}{\hookrightarrow} \Lambda\alpha{::}k.e' : Q(X, \forall\alpha{::}k.\tau')}\ \text{(DT-Uni)}$$

$$\frac{\begin{array}{c} S;\Gamma;X \vdash_p e_1 \stackrel{k}{\hookrightarrow} e_1' : ?x{:}\tau_1 \to ?y{:}\phi \to \tau_2 \\ S;\Gamma;X' \vdash_p e_2 \stackrel{\mathtt{box}(k)}{\hookrightarrow} e_2' : (x{:}\tau_1 * \phi)_k \end{array}}{S;\Gamma;X,X' \vdash_p e_1\ e_2 \stackrel{K}{\hookrightarrow} (e_2'\ \tau_2)\ (\lambda x{:}\tau_1.\lambda y{:}\phi.e_1'\ x\ y) : \tau_2}\ \text{(DT-AppE)} \qquad \frac{\begin{array}{c} S;\Gamma;X \vdash_p e \stackrel{k}{\hookrightarrow} e' : ?x{:}\tau_1 \to ?y{:}\phi \to \tau_2 \\ S;\Gamma;X' \vdash_p v \stackrel{\mathtt{box}(k)}{\hookrightarrow} (x{:}(v'{:}\tau_1), (\mathsf{pf}{:}\phi))_k : \tau' \end{array}}{S;\Gamma;X,X' \vdash_p e\ v \stackrel{K}{\hookrightarrow} e'\ v'\ \mathsf{pf} : \tau_2[v'/x]}\ \text{(DT-AppV)}$$

$$\frac{S;\Gamma;X \vdash_p e \stackrel{k'}{\hookrightarrow} e' : ?\forall\alpha{::}k.\tau' \quad S;\Gamma \vdash \tau_1 \hookrightarrow \tau_1' :: K \quad \bar{K} = k}{S;\Gamma;X \vdash_p e\ \tau_1 \stackrel{K}{\hookrightarrow} e'\ \tau_1' : \tau'[\tau_1'/\alpha]}\ \text{(DT-TApp)}$$

$$\frac{\begin{array}{c} S;\Gamma;X \vdash_p v_p \stackrel{k}{\hookrightarrow} v_p' : \tau' \quad S;\Gamma \vdash \tau_i \hookrightarrow \tau_i' :: K \quad S;\Gamma, x_i{:}(p,\tau_i'), x_i \doteq v_i; \vec{x} \vdash_p D\ \vec{\tau}\ \vec{x} \stackrel{k}{\hookrightarrow} (D\ \vec{\tau}\ \vec{x})' : \tau' \\ S;\Gamma, x_i{:}(p,\tau_i'), x_i \doteq v_i, v_p' \doteq (D\ \vec{\tau}\ \vec{x})'; X', \vec{x} \vdash_p e_1 \stackrel{K}{\hookrightarrow} e_1' : \tau \qquad S;\Gamma;X' \vdash_p e_2 \stackrel{K}{\hookrightarrow} e_2' : \tau \end{array}}{S;\Gamma;X,X' \vdash_p \text{match } v_p \text{ with } D\ \vec{\tau}\ \vec{x} \to e_1 \text{ else } e_2 \stackrel{K}{\hookrightarrow} \text{match } v_p' \text{ with } (D\ \vec{\tau}\ \vec{x})' \to e_1' \text{ else } e_2' : \tau}\ \text{(DT-Match)}$$

$$\frac{S;\Gamma;X \vdash_p e \stackrel{\mathtt{box}(k)}{\hookrightarrow} e' : (x{:}\tau * \phi)_k}{S;\Gamma;X \vdash_p e \stackrel{k}{\hookrightarrow} (e'\ \tau)\ (\lambda x{:}\tau.\lambda y{:}\mathsf{proof}\ \phi.x) : \tau}\ \text{(DT-UnBox)} \qquad \frac{S;\Gamma;X \vdash_p e \stackrel{k}{\hookrightarrow} e' : \tau \quad \boxed{S;\Gamma;\cdot \vdash_\bot v : \mathsf{proof}\ \phi} \quad v \textit{ unique}}{S;\Gamma;X \vdash_p e \stackrel{\mathtt{box}(k)}{\hookrightarrow} (x{:}(e'{:}\tau), (v{:}\phi))_k : (x{:}\tau * \phi)_k}\ \text{(DT-Box)}$$

$$\frac{\begin{array}{c} S;\Gamma;X \vdash_p e \stackrel{K}{\hookrightarrow} e' : \tau \\ S;\Gamma \vdash \tau \cong \tau' \end{array}}{S;\Gamma;X \vdash_p e \stackrel{K}{\hookrightarrow} e : \tau'}\ \text{(DT-Eq)} \qquad \frac{S;\Gamma;X \vdash_p e \stackrel{\mathtt{box}(k)}{\hookrightarrow} e' : (x{:}\tau * \phi)_k \quad \boxed{S;\Gamma, x{:}\tau, y{:}\mathsf{proof}\ \phi; \cdot \vdash_\bot v : \mathsf{proof}\ \phi'} \quad v \textit{ unique}}{S;\Gamma;X \vdash_p e \stackrel{\mathtt{box}(k)}{\hookrightarrow} (e'\ (x{:}\tau * \phi')_k)\ (\lambda x{:}\tau.\lambda y{:}\mathsf{proof}\ \phi.(x{:}(x{:}\tau), (v{:}\phi'))_k) : (x{:}\tau * \phi')_k}\ \text{(DT-RefE)}$$

$$\frac{S;\Gamma;X \vdash_p v \stackrel{\mathtt{box}(k)}{\hookrightarrow} (x{:}(v_1{:}\tau), (e_2{:}\phi))_k : (x{:}\tau * \phi)_k \quad \boxed{S;\Gamma, x{:}\tau, y{:}\mathsf{proof}\ \phi; \cdot \vdash_\bot v' : \mathsf{proof}\ \phi'} \quad v' \textit{ unique}}{S;\Gamma;X \vdash_p e \stackrel{\mathtt{box}(k)}{\hookrightarrow} (x{:}(v_1{:}\tau), (((\lambda x{:}\tau.\lambda y{:}\mathsf{proof}\ \phi.v')\ v_1\ e_2){:}\phi'))_k : (x{:}\tau * \phi')_k}\ \text{(DT-RefV)}$$

Figure 12. Derefinement of FINE.

Theorem 28 (Derefinement correctness)**.**

$I.\quad \forall S, S', \Gamma, \Gamma', X, e, \tau, p.$
well-formed$(S;\Gamma;X)\ \wedge$ *(A1)*
$S;\Gamma;X \vdash_p e : \tau\ \wedge$ *(A2)*
$S;\Gamma \hookrightarrow S';\Gamma'\ \Rightarrow$ *(A3)*
$\exists e', \tau'. S';\Gamma';X \vdash_p e \hookrightarrow e' : \tau'\ \wedge$ *(G1)*
$\exists K. S';\Gamma' \vdash \tau \hookrightarrow \tau' :: K\ \wedge$ *(G2)*
$S';\Gamma';X \vdash e' : \tau'$ *(G3)*

$II.\quad \forall S, S', \Gamma, \Gamma', \tau, \kappa.$
well-formed$(S;\Gamma)\ \wedge$ *(B1)*
$S;\Gamma \vdash_p \tau :: \kappa\ \wedge$ *(B2)*
$S;\Gamma \hookrightarrow S';\Gamma'\ \Rightarrow$ *(B3)*
$\exists \tau', K'. S';\Gamma' \vdash_p \tau \hookrightarrow \tau' :: K\ \wedge$ *(F1)*
$S';\Gamma' \vdash \tau' :: \kappa$ *(F2)*

Proof. By mutual induction on the structure of (A2) and (B2):

```
Cases of (B2): The main cases are (DK-Fun) and (DK-Dep). The former relies on weakening of the
 kinding judgment in the second premise. The latter relies on the mutual induction hypothesis for
 the second premise.


Cases of (A2): The main interesting cases are (T-Fun) and (T-App).

--(T-D), (T-X), (T-XA):

 Straightforward from the definition of environment translation and the definitions
 of (DT-D) and (DT-X).

--(T-Drop), (T-Bracket): Induction hypothesis.

--(T-Fun):

   S;G |- t1 :: k                              (A1.1)
   S;G,x:(p,t1); X,x |- e : t2                 (A1.2)
 --------------------------------------------- (A1)
   S;G;X |-_p \x:t1.e : Q(X, x:t1 -> t2)

  From the mutual induction hypothesis applied to (A1.1) we get

  S';G' |- t1  --> t1' :: K                                        (F1.1)

  We consider two subcases:

----Subcase (K=box(k)):

    We have t1' = (x:t1'' * phi)

    From the definition of environment translation, we have that

    S;G,x:(p,t1) -->  S';G',x:(p,t1''), y:proof phi

    So, from the induction hypothesis applied to (A1.2), we get

    S';G',x:(p,t1''), y:proof phi; X,x |- e -K-> e' : t2'     (G1.1)
    S';G',x:(p,t1''), y:proof phi      |- t2 --> t2' :: K     (G2.1)
    S';G',x:(p,t1''), y:proof phi; X,x  |- e' : t2'           (G3.1)

    For the goal (G1), we use (DT-Fun) with (F1.1) and (G1.1) in the premises.

    For the goal (G2), we have to show that


       S';G' |- x:{x:t1 | phi} -> t2   -->  x:t1' -> y:proof phi -> t2' :: k     (G2.1)

    Which is immediate from the definition of (DK-Fun).

    For the goal (G3), we use (T-Fun) twice, with (G3.1) in the premise.

----Subcase (K=k):
```

```
    First, we use (DT-Box) with (F1.1) in the premise to construct

      S';G' |- t1 --> (x:t1' * True)                           (F1.1')

    Next, we use weakening on (A1.2) to construct

      S;G,x:(p,t1),y:proof True; X x |- e : t2                 (A1.2')

    From the definition of environment translation, we have

    S;G,x:(p,t1),y:proof True --> S';G',x:(p,t1'), y:proof True

    So, from the induction hypothesis applied to (A1.2'), we get

    S';G',x:(p,t1'), y:proof phi; X,x |- e -K-> e' : t2'      (G1.1)
    S';G',x:(p,t1'), y:proof phi      |- t2 --> t2' :: K      (G2.1)
    S';G',x:(p,t1'), y:proof phi; X,x  |- e' : t2'            (G3.1)

    The rest of the proof proceeds as in the previous sub-case.

--(T-Uni):


        S;G,'a::k;X |-_p e : t                     (A1.1)
   --------------------------------------------- (A1)
        S;G;X |-_p /\'a::k.e : Q(X, \/'a::k.t)


   From the induction hypothesis applied to (A1.1), and (XK-S) and (XK-A), and the definition of
   environment translation, we get:

    S';G','a::k; X |-_p e -K-> e' : t'             (G1.1)
    S';G','a::k |- t --> t' :: k                   (G2.1)
    S';G','a::k; X |-_p e' : t'                    (G3.1)


   For the goal (G1), we use (DT-Uni) with (G1.1) in the premise.
   For the goal (G2), we use (DK-Uni) with (G2.1) in the premise.
   For the goal (G3), we use (T-Uni) with (G3.1) in the premise.


--(T-Fix):

      S;G |- t :: *                        (A1.1)
      unrefined(t)                         (A1.2)
      S;G,f:(p,t);. |-_p v : t             (A1.3)
   ------------------------------------- (A1)
      S;G;. |-_p fix f:t.v : t

   From the mutual induction hypothesis applied to (A1.1), and (A1.2) we get

      S';G' |- t --> t' :: *               (F1.1)
      S';G' |- t' :: *                     (F2.1)

   From (F1.1), observing that t' is not boxed, and the definition of environment translation, and
   the induction hypothesis we get

      S';G',f:(p,t');. |-_p v -*-> v' : t''   (G1.1')
      S';G',f:(p,t') |-_p t -*-> t'' :: *     (G1.2')
      S';G,f:(p,t');. |- v' :: t''            (G1.3')

   From weakening applied to the translation of types (F1.1), we get

         S';G',f:(p,t') |- t --> t' :: *             (F1.1')
```

```
   Finally, from determinism of the type translation (DK-*) applied to G1.2' and F1.1', we get
   t'=t'', and so:

      S';G',f:(p,t');. |-_p v -*-> v' : t'   (G1.1)
      S';G',f:(p,t') |-_p t -*-> t' :: *     (G1.2)
      S';G,f:(p,t');. |- v' :: t'            (G1.3)


   For the goal (G1), we use (Dt-Fix) with (G1.2) and (G1.1) in the premises.
   For the goal (G2), we use (F1.1).
   For the goal (G3), we use (T-Fix), with (F2.1) and (G1.3).


--(T-App):


     S;G;X |-_p e1 : ? x:t1 -> t2      (A1.1)
     S;G;X' |-_p e2 : t1               (A1.2)
     S;G |- t2 [e2/x] :: k             (A1.3)
   ---------------------------------- (A1)
     S;G;X,X' |-_p e1 e2 : t2[e2/x]


    From the induction hypothesis applied to (A1.1), we get:

    S';G';X |-_p e1 -k-> e1' : t'        (G1.1.1)
    S';G' |- ?x:t1 -> t2  --> t' :: k    (G2.1.1)
    S';G';X |-_p e1' : t'                (G3.1.1)


    By inversion of (G2.1.1), we get an application of (DK-Fun), with

    t' = ? x:t1' -> y:proof phi -> t2'                      (Eq_t')

    with

    S';G' |- t1 --> (x:t1' * proof phi) :: box(k)           (Inv1)
    S';G',x:(p,t1'),y:(p,proof phi) |- t2 --> t2' :: K      (Inv2)


    From the induction hypothesis applied to (A1.2), we get:

    S';G';X |-_p e2 -box(k)-> e2' : t1''      (G1.1.2)
    S';G' |- t1 --> t1'' :: box(k)            (G2.1.2)
    S';G';X |-_p e2' : t1''                   (G3.1.2)

    From determinism applied to (G2.1.2) and (Inv1), we get

    t1''= (x:t1' * proof phi).                               (Eq_t1'')

    For the goal, we consider two separate sub-cases, depending on whether e2 is a value.

----Subcase (e2 is not a value):

    In this case, from the definition of (K-Dep), we can conclude that in (A1.3), t2[e2/x] = t2.

    For the goal (G1), we apply (DT-AppE), with
        (G1.1.1) in the first premise with (Eq_t'),
        (G1.1.2) in the second premise with (Eq_t1''), to derive

     S';G';X,X' |-_p e1 e2 -K-> (e2' t2') (\x:t1'.\y:proof phi. e1' x y) : t2'  (G1)

    For the goal (G2), we use Inv2, noting that t2[e2/x] = t2.

    For the goal (G3), we have to show that
```

```
     S';G';X,X' |-_p (e2' t2') (\x:t1'.\y:proof phi. e1' x y) : t2'

    which follows from the definition of (x:t * phi), (G3.1.2), and (G3.1.1).


----Subcase (e2=v2):

    From the derefinement of values, we have

    S';G';X |-_p v2 -box(k)-> (x:(v2':t1') * e2':proof phi) : (x:t1' * proof phi)  (G1.1.2)

    For the goal (G1), we apply (DT-AppV), with
         (G1.1.1) in the first premise
         (G1.1.2) in the second premise, to derive

     S';G';X,X' |-_p e1 v2 -K-> (e1' v2' pf) : t2'[v2'/x]   (G1)


    For the goal (G2), we use the subsitution lemma for translation, applied to Inv2.

    For the goal (G3), we use (T-App) twice, noting that y not in FV(t2'), and using (G3.1.1) and
    (G3.1.2).

--(T-TApp, T-Match):

    Both straightforward.

--(T-Sub):

    The definitions of (T-Sub) are inlined in the (DT-*) judgment. The conclusion follows from the
    lemmas for translation of type equivalence and subtyping.
```

□

D. Semantics of DCIL

Static semantics is shown in Figure 13. Dynamic semantics is shown in Figure 14.

E. Soundness of DCIL

Theorem 29 (Subject reduction)**.**

```
(A1) S; .; G(M); X |-_p e: t
(A2)(M, e) ~p~> (M', e')
 => S; .; G(M'); X' |-_p e': t /\
          X' = X U (dom M' \ dom M) if dom M' >= dom M
          X' = X \ (dom M \ dom M') otherwise
```

Proof. By induction on the structure of (A2).
Case (TE-Pure): by Lemma (Pure evaluation)
Other cases can be proved similarly to those in the source language. □

Lemma 30 (Pure evaluation)**.**

```
(A1) S; .; G; X |-_p e: t
(A2) e ~p~> e'
=> S; .; G; X |-_p e': t
```

Proof. By induction on the structure of (A1).

```
Case (TT-X, TT-XA): Irreducible.

Case (TT-Drop): induction hypothesis

Case (TT-Ldfld):
    S; .; G; X |-_p v:? T<ts,vs> (B1)
    Sp(T<ts, vs>) = fi:ti        (B2)
```

```
    --------------------------------[TT-Ldfld]
    S; .; G; X |-_p v.fi : ti

By (TE-Fld), v = D<ts', vs'> and e' = vi

By (TT-New) X = X1..Xh
Sp(D) = D<'a1::k1..'ag::kg, x1:t1'..xh:th'> : T<ts'', vs''>
T<ts'', vs''>[ts'/'a1..'ag, vs'/x1..xn] = T<ts,vs>

By (WF-Ddecl), Sp(D<ts', vs'>) = fi : ti

By (TT-New) S; .; G; X |- vi: ti

Case (TT-Bracket):

    S; .; G; X |-_q e1: t (B1)
    ------------------------[TT-Bracket]
    S; .; G; X |-_p <e1>q : t

By (TE-Br), e1 ~q~ e1' and e' = <e1'>q

By induction hypothesis on (B1), S; .; G; X |-_q e1' : t

By [TT-Bracket], S; .; G; X |-_p <e1'>q : t

Case (TT-Let):
      S; .; G; X |-_p v: t1        (B1)
       S; .; G,x:t1; X,x |-_p e2: t (B2)
    -----------------------------------[TT-Let]
     S; .; G; X |-_p let x = v in e2: t

By (TE-Let) e' = e2[v/x].

By Lemma (Substitution) and (B1) and (B2),  S; .; G; X |-_p e': t

Case (TT-Eq):
      S; .; G; X |-_p e: t1 (B1)
      S; .; G |- t1 = t    (B2)
     ------------------------[TT-Eq]
      S; .; G; X |-_p e: t

By induction hypothesis on (B1),  S; .; G; X |-_p e': t1

By [TT-Eq],  S; .; G; X |-_p e': t

Case (TT-New): Irreducible.

Case (TT-App):
    S; .; G; X1 |-_p v: ?T<ts,vs>    (B1)
    Sp(T<ts,vs>) = t2 m<'a::k>(x:t1) (B2)
    S; .; G |- t0:: k                (B3)
    S; .; G; X2 |-_p v': t1[t0/'a]   (B4)
    ------------------------------------------[TT-App]
    S; .; G; X1, X2 |-_p v.m<t0>(v'): t2[t0/'a][v'/x]

By (TE-App), v = D<ts', vs'>, Sp(D<ts', vs'>) = t2 m<'a::k>(x:t1) {eb}, e' = eb[t0/'a, v'/x].

By (WF-Method), S; 'a::k; G, x:t1; X1, X2 |-_p eb: t2

By (B3), (B4), and Lemma (Substitution), S; .; G; X1, X2 |-_p e': t

Case (TT-Isinst-obj):

    S; .; G; X1 |-_p D'<ts, vs>: ?tx                                    (B1)
    Sp(D) = D<'a1::k1..'am:km, x1':t1'..xn':tn'>: t_D                    (B2)
```

```
    S; .; G, x1:t1'[t1..tm/'a1..'am]..xn:tn'[t1..tm/'a1..'am, x1..x_(n-1)/x1'..x_(n-1)'];
       X1, x1..xn |-_p D<t1..tm, x1..xn> : tx                                    (B3)
    S; .; G, x1:t1'[t1..tm/'a1..'am]..xn:tn'[t1..tm/'a1..'am, x1..x_(n-1)/x1'..x_(n-1)'];
       X2, x1..xn |-_p et: t                                                     (B4)
    S; .; G; X2 |-_p ef: t                                                      (B5)
    ------------------------------------------------------------------------[TT-Isinst-x]
    S; .; G; X1, X2 |-_p D'<ts, vs> isinst D<t1..tm, x1..xn> then et else ef

There are two cases:
  case 1: D' = D, e' = et[vs/x1..xn]

  By (B1) and (TT-New), X1 = X11, ..., X1n,
  forall i=1..n, S; .; G; X1i |-_p vi: ti'[t1..tm/'a1..'am, v1..v_(i-1)/x_(i-1)]

  By Lemma (Substitution) and (B4), S; .; G; X2 |- et[vs/xi..xn] : t

  By (TT-Drop), S; .; G; X1, X2 |-_p e': t

  case 2: D' <> D, e' = ef
    By (B5) and (TT-Drop), S; .; G; X1, X2 |-_p e': t
```

□

Theorem 31 (Progress). (A1) S; .; G; dom(M) |-_p e: t,
=> either exists v_p s.t. e = v_p or exists M', e' s.t. (M, e) ~p~> (M', e')

Proof. By induction on the structure of (A1).

```
Case TT-X, TT-XA, TT-New: already values

Case TT-Drop: induction hypothesis

Case TT-Ldfld: e = v.fi.

    S; .; G; X |-_p v:? T<ts,vs> (B1)
    Sp(T<ts, vs>) = fi:ti        (B2)
    --------------------------------[TT-Ldfld]
    S; .; G; X |-_p v.fi : ti

By Lemma (Canonical forms) and (B1), v = D<ts', vs'>.

By (WF-Ddecl) and (B2), Sp(D<ts',vs'>) = fi:ti.
Therefore D<ts', vs'> has a field fi and (TE-fld) applies.

Case TT-Bracket: similar to the cases in Fine.

Case TT-Let: e = let x = e1 in e2.

If e1 is a value, apply (TE-Cong). Otherwise (TE-Let)

Case TT-Eq: induction hypothesis

Case TT-App:
    S; .; G; X1 |-_p v: ?T<ts,vs>    (B1)
    Sp(T<ts,vs>) = t2 m<'a::k>(x:t1) (B2)
    S; .; G |- t0:: k                (B3)
    S; .; G; X2 |-_p v': t1[t0/'a]   (B4)
    ------------------------------------------[TT-App]
    S; .; G; X1, X2 |-_p v.m<t0>(v'): t2[t0/'a][v'/x]

By Lemma (Canonical form) and (B1), v = D<ts',vs'>

By (WF-Ddecl) and (B2), Sp(D<ts',vs'>) = t2 m<'a::k>(x:t1). (TE-App) applies.

Case TT-Isinst: apply (TE-Isinst)
```

□

Lemma 32 (Canonical forms)**.**

```
S; .; G; X |- v: ?T<ts, vs>
=> v = D<ts', vs'>
```

Proof. By induction on the expression typing rules. □

Lemma 33 (Substitution)**.**

```
S; 'a::k; G1, G2; X |-_p e:t and S; .; G1 |- t0::k
=> S; .; G1, G2[t0/'a]; X |-_p e[t0/'a]: t[t0/'a]

S; .; G, x:t0; X,x |-_p e:t and S; .; G; . |- v: t0
=> S; .; G; X |-_p e[v/x] : t[v/x]
```

Proof. By induction on expression typing rules. □

F. Value Abstraction of DCIL

Theorem 34 (DCIL value abstraction)**.**

```
(A1) S;.;x:(p, tx); x |-_q e:t
(A2) q and p are not in the same assembly,
and e is a non-value free of r-brackets where r and p are in the same assembly, except for <x>_p
(A3) forall i, S; .; .; . |-_p vi_p : tx
(A4) e[v1_p/x] ~q~> e1
=> exists e2, s.t. e2[v1_p/x] = e1 and e[v2_p/x] ~q~> e2[v2_p/x]
```

Proof. by induction on expression evaluation rules.

```
Case (TE-Br), (TE-Strip), (TE-Nest):
similar to those in the source language value abstraction proof.

Case (TE-Fld):
    ------------------------------------[TE-Fld]
     (D<ts, vs>.fi)[v1_p/x] ~q~> v_i[v1_p/x]

pick e2 = vi

Case (TE-App):

     Sp(D<ts,vs>) = t m<'a::k>(y:t') { e }
    -----------------------------------------------------[TE-App]
    (D<ts,vs>.m<t>(v))[v1_p/x] ~q~> e[v1_p/x][t/'a, v[v1_p/x]/y]

pick e2 = e[t/'a, v/y]

Case (TE-Isinst):

    v = D<ts, vs> => e' = et[vs/xs]
    v = D'<ts', vs>, D' != D => e' = ef
    --------------------------------------------------[TE-Isinst]
    (v isinst D<ts, xs> the et else ef)[v1_p/x] ~q~> e'[v1_p/x]

pick e2 = e'

Case (TE-Let):

    -----------------------------[TE-Let]
    (let y = v in e)[v1_p/x] ~q~> e[v1_p/x][v[v1_p/x]/y]

pick e2 = e[v/y]
```

□

G. Translation from FINE to DCIL

Translation of types and expressions is shown in Figure 15.

H. Proofs of Type-preserving Translation

Suppose S0 collects all class declarations generated by the translation.

Definition 35 (Environment translation)**.** `||.|| = .`

```
||T::k|| =||T, k||
||D:(p, t)|| = ||D, t||
||S, S'|| = ||S||, ||S'||

||.|| = (., .)
||'a::k|| = ('a::||k||, .)
||x:(p, t)|| = (., x:(p, ||t||))
||\epmatch{x}{v}|| = (., x \cnv ||v||)
||G, G'|| = ((G1, G1'), (G2, G2')) where ||G|| = (G1, G2) and ||G'|| = (G1', G2')

If ||G|| = (G1, G2) and  ||S; G; F|| |- ||t_f|| ~> t_f', then
||S; G; (f:t_f, F)|| is defined as (||S||, S0; G1; G2, this:t_f') and
||S; G; X; (f:t_f, F)|| is defined as (||S||, S0; G1; G2, this:t_f'; X, this)
```

Lemma 36. *Type translation*

```
S; G |- t :: k in Fine,
S; G; F |- ||t|| ~> t'
=> ||S; G; F|| |- t' :: ||k|| in DCIL
```

Proof. By induction on translation of types ☐

Lemma 37. *Variable translation*

```
S; G; X |-_p x: t in Fine
S; G; F |- ||t|| ~> t'
=> ||S; G; X; F|| |-_p x: t' in DCIL
```

Proof. There are two cases:

```
case 1: X = ., G(x) = (p, t), and S; G |- t :: *

By Lemma (Type translation), ||S; G; F|| |- t' :: *

By Definition (Environment translation), ||G||(x) = (p, t')

By TT-X, ||S; G; X; F|| |-_p x:t' in DCIL

case 2: G(x) = (p, t). Similiar to case 1.
```

☐

Lemma 38. *Type translation commutes with substitution*

```
S; G; F |- ||t|| ~> s
forall i=1..m, S; G; F |- ||ti|| ~> si
forall j = 1..n, S; G; X; F |-_p ||vj|| ~> vj'
S; G; F |- ||t[t1..tm/'a1..'am, v1..vn/x1..xn]|| ~> s'
=>  s' = s[s1..sm/'a1..'am, v1'..vn'/x1..xn]
```

Proof. By induction on translation of types ☐

Lemma 39 (Value translation)**.** `1. Values in Fine are translated to values in DCIL.`

```
2. S; G; X |-_p v : t in Fine
   S; G; X; F |-_p ||v:t|| ~> v'
   S; G; F |- ||t|| ~> t'
  => ||S; G; X; F|| |-_p v' : t' in DCIL
```

Proof. By induction on value translation rules. ☐

Lemma 40 (Type-preserving translation)**.** `(A1) S; G; X |-_p e: t in Fine`

```
 (A2) S; G; X; F |-_p ||e: t|| ~> e'
 (A3) S; G; F |- ||t|| ~> t'
 => ||S; G; X; F|| |-_p e' : t' in DCIL.
```

Proof. By induction on A2.

```
Case (Tr-Obj):

     forall i = 1..m, S; G; F |- ||t_i|| ~> s_i       (B1)
     forall j = 1..n, S; G; X; F |-_p ||v_j|| ~> v_j' (B2)
     -----------------------------------------------------------------------[Tr-Obj]
     S; G; X; F |-_p ||D t1..tm v1..vn  : t|| ~> D<s1..sm, v1'..vn'>

By (A1), T-D, T-TApp, and T-App,
  (C1). S(D) = (p, \/'a1::k1..'am:km (x1:t1')->...->(xn:tn') -> t_D)
  (C2).  S; G |- ti :: ki forall i=1..m
  (C3).  X = X1, ..., Xn
  (C4). S; G; Xj |- vj: tj'[t1..tm/'a1..'am, v1..v(j-1)/x1..x(j-1)] forall j=1..n
  (C5). t = t_D[t1..tm/'a1..'am, v1..vn/x1..xn]

Let S; 'a1::k1..'am::km, x1:t1'..x_(j-1):t_(j-1)'; . |- ||tj'|| ~> sj' forall j = 1..n

Let S; 'a1::k1..'am::km, x1:t1'..xn:tn'; . |- ||t_D|| ~> s_D

By Tr-D and (C1), S0(D) = internal D<'a1::||k1||..'am:||km||, x1:s1'..xn:sn'> : s_D {}.

By Lemma (Type translation) and (C2), ||S; G; F|| |- si :: ||ki|| forall i=1..m

Let S; G; F |- ||tj'[t1..tm/'a1..'am, v1..v(j-1)/x1..x(j-1)]|| ~> ssj forall j=1..n

By Lemma (Value translation) and (C4), ||S; G; Xj; F|| |-_p vj': ssj forall j=1..n

By Lemma (Type translation commutes with substitution),
||S; G; Xj; F|| |-_p vj': sj'[s1..sm/'a1..'am, v1'..v(j-1)'/x1..x(j-1)] forall j=1..n

By TT-New, ||S; G; X; F|| |-_p D<s1..sm, v1'..vn'>: s_D[s1..sm/'a1..'am, v1'..vn'/x1..xn]

By Lemma (Type translation commutes with substitution) and (C5), ||S; G; X; F|| |-_p D<s1..sm, v1'..vn'>: t'

Case (Tr-X):

    -------------------------[Tr-X]
    S; G; X |-_p ||x : t|| ~> x

By Lemma (Variable translation), ||S; G; X; F|| |-_p x: t' in DCIL

Case (Tr-F):

    ----------------------------------[Tr-F]
    S; G; .; (f:t, F) |-_p ||f : t|| ~> this

By (T-x) and (T-XA), G(f) = (p, t)

By Definition (Environment translation), ||S; G; .; (f:t, F)||(this) = t'

By TD-XA, ||S; G; .; (f:t, F)|| |-_p this : t' in DCIL

Case (Tr-Fun): t = Q(X, x:t1->t2)

     S; G, x:(p, t1); X, x; F |-_p ||e:t2|| ~> eb'            (B1)
     C is a fresh class name                                  (B2)
     S; G; F |- ||t1|| ~> s1                                  (B3)
     S; G, x:(p, t1); F |- ||t2|| ~> s2                       (B4)
     internal C<tvars(G), vvars(G)> : t' { s2 App(x:s1){eb'}} (B5)
    --------------------------------------------------------------------[Tr-Fun]
    S; G; X; F |-_p ||\x:t1.e: t|| ~> C<tvars(G), vvars(G)>

By Lemmas (Type translation) and (Value translation) and (TT-New)

Case (Tr-Uni): t = Q(X, \/'a::k.t'')
```

```
     S; G, 'a::k; X; F |-_p ||e:t''|| ~> eb'                        (B1)
     C is a fresh class name                                        (B2)
     S; G, 'a::k; F |- ||t''|| ~> s'                                (B3)
     internal C<tvars(G), vvars(G)> : t' { s' TyApp<'a::k>(){eb'}} (B4)
    ---------------------------------------------------------[Tr-Uni]
    S; G; X; F |-_p ||/\'a::k.e : t|| ~> C<tvars(G), vvars(G)>
```

By Lemmas (Type translation) and (Value translation) and (TT-New)

Case (Tr-Fix):

```
     S; G, f:(p,t); .; (f:t, F) |-_p ||v_p: t|| ~> e' (B1)
    ------------------------------------------------[Tr-Fix]
    S; G; .; F |-_p ||fix f:t.v_p: t|| ~> e'
```

By induction hypothesis on B1, ||S; G, f:(p,t); .; (f:t, F)|| |-_p e': t'

e' has no free occurrence of f by (Tr-F), therefore, ||S; G; .; F|| |-_p e': t'

Case (Tr-App): t = t2[e2/x], e' = let x1 = e1' in let x2 = e2' in x1.App(x2)

```
     S; G; X; F |-_p ||e1: ?x:t1->t2|| ~> e1'  (B1)
     S; G; X'; F |-_p ||e2:t1|| ~> e2'         (B2)
    ------------------------------------------------[Tr-App]
    S; G; X, X'; F |-_p ||e1 e2: t|| ~>
       let x1 = e1' in let x2 = e2' in x1.App(x2)
```

Let S; G; F |- ||x:t1->t2|| ~> s_f
and S; G; F |- ||t1|| ~> s1
and S; G,x:t1; F |- ||t2|| ~> s2

By (Tr-tdep) s_f = C<s1, \x:s1.s2> where C = DepArrow, DepArrowSA, or DepArrowSS

By induction hypothesis on B1, ||S; G; X; F|| |-_p e1': ?s_f

By induction hypothesis on B2, ||S; G; X'; F|| |-_p e2': s1

By (TT-Let) and (TT-App), ||S; G; X, X'; F|| |-_p e': (\x:s1.s2) x2

(\x:s1.s2) x2 = s2[x2/x] = s2[e2'/x]

By Lemma (Type translation commutes with substitution), ||S; G; X, X'; F|| |-_p e': t'

Case (Tr-TApp): t = t1[t2/\'a], e' = let x = e1' in x.TyApp<s2>()

```
     S; G; X; F |-_p ||e1: ?\/'a::k.t1|| ~> e1'   (B1)
     S; G; F |-_p ||t2|| ~> s2'                   (B2)
    --------------------------------------------------------[Tr-TApp]
    S; G; X; F |-_p ||e1 t2: t|| ~ let x = e1' in x.TyApp<s2>()
```

Let ||S; G, 'a::k; F || |- ||t1|| = s1'. a-lifting of s1' = ('a1..'am)(s1..sm, s').

Then S; G; F |-_p ||\/'a::k.t1|| ~> All_'a1..'am<s1..sm>
and S0(All_'a1..'am) = public All_'a1..'am<'a1..'am>::*{s' TyApp<'a::k>()}

By (TT-Let) and (TT-App), ||S; G; X; F|| |-_p e': s1'[s2/'a].

By Lemma (Type translation commutes with substitution), ||S; G; X, X'; F|| |-_p e': t'

Case (Tr-Match): e' = let x = ev in x isinst tD then e1' else e2'

```
     S; G; X1; F |-_p ||vp: tt|| ~> ev                                (B1)
     S(D) = \/ 'a1::k1..'am::km (y1:t1') -> ... -> (yn:tn') -> t_D     (B2)
     S; G,xi:(p, ti'[t1..tm/'a1..'am]); X1, x1..xm |-_p D t1..tm x1..xn : tt  (B3)
```

```
     S; G, xi:(p, ti'[t1..tm/'a1..'am]), vp = D t1..tm x1..xn; X2, x1..xm; F  (B4)
         |-_p ||e1:t|| ~> e1'
     S; G; X2; F |-_p ||e2:t|| ~> e2'                                        (B5)
    ---------------------------------------------------------------------[Tr-match]
    S; G; X1, X2; F |-_p ||match vp with D t1..tm x1..xn -> e1 else e2: t|| ~>
      let x = ev in x isinst tD then e1' else e2'

Let S; G, 'a1::k1..'am::km, y1:t1'..y_(i-1):t_(i-1)'; F |- ||ti'|| ~> si' forall i=1..n,
S; G, 'a1::k1..'am::km, y1:t1'..yn:tn'; F |- ||t_D|| ~> s_D,
S; G; F |- ||tt|| ~> ss,
S; G; F |- ||ti|| ~> si forall i=1..m.

By (Tr-D), SO(D) = D<'a1::k1..'am::km, y1:s1'..yn:sn'>:s_D {}

By induction hypothesis on (B1), ||S; G; X1; F|| |-_p ev: ss

By induction hypothesis on (B3),
||S; G,xi:(p, ti'[t1..tm/'a1..'am]); X1, x1..xm|| |-_p D<s1..sm, x1..xn> : ss

By induction hypothesis on (B4),
||S; G, xi:(p, ti'[t1..tm/'a1..'am]), vp = D t1..tm x1..xn; X2, x1..xm; F ||
|-_p e1': t'

By induction hypothesis on (B5),  ||S; G; X2; F|| |-_p e2': t'

By Definition (Environment translation) and Lemma (Type translation commutes with substitution),
||S; G, xi:(p, ti'[t1..tm/'a1..'am]), vp = D t1..tm x1..xn; X2, x1..xm; F || =
||S; G; X; F||, xi:(p, si'[s1..sm/'a1..'am]), ev=D<s1..sm, x1..xn>]

By (TT-Let) and (TT-Isinst), ||S; G; X1, X2; F|| |-_p e': t'
```

□

References

A. W. Appel and E. W. Felten. Proof-carrying authentication. In *CCS*. ACM Press, 1999.

J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. In *CSF*, 2008.

Y. Bertot and P. Castéran. *Coq'Art: Interactive Theorem Proving and Program Development.* Springer Verlag, 2004.

J. Borgstroem, A. Gordon, and R. Pucella. Roles, stacks, histories: A triple for hoare. Unpublished manuscript, 2009.

A. Chlipala, G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Effective interactive proofs for higher-order imperative programs. In *ICFP*, 2009.

S. Chong, A. C. Myers, N. Nystrom, L. Zheng, and S. Zdancewic. Jif: Java + information flow, July 2006. Software release.

L. de Moura and N. Bjorner. Z3: An efficient smt solver. In *TACAS*, LNCS v. 4963, 2008.

D. J. Dougherty, K. Fisler, and S. Krishnamurthi. Specifying and reasoning about dynamic access-control policies. In *LNCS*, 2006.

ECMA. Standard ecma-335: Common language infrastructure (cli), 2006.

K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz. Verification and change-impact analysis of access-control policies. In *ICSE*, 2005.

C. Flanagan. Hybrid type checking. In *POPL*, 2006.

C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *PLDI*. 1993.

D. Grossman, G. Morrisett, and S. Zdancewic. Syntactic type abstraction. *ACM TOPLAS*, 22(6):1037–1080, 2000. ISSN 0164-0925.

L. Jia, J. Vaughan, K. Mazurak, J. Zhao, L. Zarko, J. Schorr, and S. Zdancewic. Aura: A programming language for authorization and audit. In *ICFP*, 2008.

A. Kennedy and D. Syme. Transposing f to c#: Expressivity of polymorphism in an object-oriented language. *Concurrency and Computation: Practice and Experience*, 16(7), 2004.

S. Krishnamurthi. The continue server. In *Practical aspects of declarative languages*, volume 2562 of *Lecture Notes in Computer Science*. Springer, 2003. ISBN 3-540-00389-4.

C. McBride and J. McKinna. The view from the left. *JFP*, 14(1), 2004.

R. Milner. LCF: A way of doing proofs with a machine. In *MFCS*, 1979.

U. Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Chalmers Institute of Technology, 2007.

V. Simonet. FlowCaml in a nutshell. In G. Hutton, editor, *APPSEM-II*, pages 152–165, Mar. 2003.

A. Stump, M. Deters, A. Petcher, T. Schiller, and T. Simpson. Verified programming in guru. In *PLPV*, 2008.

N. Swamy and M. Hicks. Verified enforcement of stateful information release policies. In *PLAS*, 2008.

N. Swamy, B. J. Corcoran, and M. Hicks. Fable: A language for enforcing user-defined security policies. In *S&P*, 2008.

N. Swamy, M. Hicks, and G. Bierman. A theory of typed coercions and its applications. In *ICFP*, 2009.

J. A. Vaughan, L. Jia, K. Mazurak, and S. Zdancewic. Evidence-based audit. In *CSF*, 2008.

D. Walker. *Advanced Topics in Types and Programming Languages*, chapter Substructural Type Systems. MIT Press, 2004.

D. Walker, K. Crary, and G. Morrisett. Typed memory management via static capabilities. *ACM TOPLAS*, 22(4), 2000.

H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *POPL*, 2003.

$\Sigma;\Delta;\Gamma \vdash \kappa$ **Well-formedness of kinds**

$$\frac{}{\Sigma;\Delta;\Gamma \vdash \star}\ \text{(TWF-*)} \qquad \frac{}{\Sigma;\Delta;\Gamma \vdash \mathtt{A}}\ \text{(TWF-A)} \qquad \frac{\Sigma;\Delta;\Gamma \vdash \tau :: \star \quad \Sigma;\Delta;\Gamma \vdash \kappa}{\Sigma;\Delta;\Gamma \vdash \tau \to \kappa}\ \text{(TWF-Dep)}$$

$\Sigma;\Delta;\Gamma \vdash \tau :: \kappa$ **Kinding of types**

$$\frac{\alpha :: \kappa \in \Delta}{\Sigma;\Delta;\Gamma \vdash \alpha :: \kappa}\ \text{(TK-Var)} \qquad \frac{\Sigma;\Delta;\Gamma \vdash \tau :: \star}{\Sigma;\Delta;\Gamma \vdash !\tau :: \mathtt{A}}\ \text{(TK-Affine)}$$

$$\frac{\begin{array}{c}\Sigma;\Delta;\Gamma \vdash \tau_1 :: \star \\ \Sigma;\Delta;\Gamma, x{:}\tau_1 \vdash \tau_2 :: \kappa\end{array}}{\Sigma;\Delta;\Gamma \vdash \backslash x{:}\tau_1.\tau_2 :: \tau_1 \to \kappa}\ \text{(TK-Fun)} \qquad \frac{\begin{array}{c}\Sigma;\Delta;\Gamma \vdash \tau :: \tau_1 \to \kappa \\ \Sigma;\Delta;\Gamma;\cdot \vdash v : \tau_1\end{array}}{\Sigma;\Delta;\Gamma \vdash \tau\ v :: \kappa}\ \text{(TK-App)} \qquad \frac{\begin{array}{c}\Sigma(T) = T\langle\vec{\alpha}{::}\vec{\kappa}, \vec{x}{::}\vec{\tau'}\rangle{::}\kappa' \quad \Sigma;\Delta;\Gamma \vdash \tau_i :: \kappa_i \\ \Sigma;\Delta;\Gamma;\cdot \vdash v_j : \tau'_j[\vec{\tau}/\vec{\alpha}][v_1 \dots v_{j-1}/x_1 \dots x_{j-1}]\end{array}}{\Sigma;\Delta;\Gamma \vdash T\langle\vec{\tau},\vec{v}\rangle :: \kappa'}\ \text{(TK-T)}$$

$\Sigma;\Delta;\Gamma;X \vdash_p e : \tau$ **Expression typing**

$$\frac{\Gamma(x) = (p,\tau) \quad \Sigma;\Delta;\Gamma \vdash \tau :: \star}{\Sigma;\Delta;\Gamma;\cdot \vdash_p x : \tau}\ \text{(TT-X)} \qquad \frac{\Gamma(x) = (p,\tau)}{\Sigma;\Delta;\Gamma;x \vdash_p x : \tau}\ \text{(TT-XA)} \qquad \frac{\Sigma;\Delta;\Gamma;X \vdash_p e : \tau}{\Sigma;\Delta;\Gamma;X,X' \vdash_p e : \tau}\ \text{(TT-Drop)}$$

$$\frac{\Sigma;\Delta;\Gamma;X \vdash_p v : ?T\langle\vec{\tau},\vec{v}\rangle \quad \Sigma_p(T\langle\vec{\tau},\vec{v}\rangle) = f_i : \tau_i}{\Sigma;\Delta;\Gamma;X \vdash_p v.f_i : \tau_i}\ \text{(TT-Ldfld)} \qquad \frac{\Sigma;\Delta;\Gamma;X \vdash_q e : \tau}{\Sigma;\Delta;\Gamma;X \vdash_p \langle e\rangle_q : \tau}\ \text{(TT-Bracket)}$$

$$\frac{\Sigma;\Delta;\Gamma;X_1 \vdash e_1 : \tau_1 \quad \Sigma;\Delta;\Gamma, x:\tau_1;X_2, x \vdash e_2 : \tau_2 \quad x \text{ not free in } \tau_2}{\Sigma;\Delta;\Gamma;X_1,X_2 \vdash_p \text{let } x = e_1 \text{ in } e_2 : \tau_2}\ \text{(TT-Let)} \qquad \frac{\Sigma;\Delta;\Gamma;X \vdash_p e : \tau_1 \quad \Sigma;\Delta;\Gamma \vdash \tau_1 \cong \tau_2}{\Sigma;\Delta;\Gamma;X \vdash_p e : \tau_2}\ \text{(TT-Eq)}$$

$$\frac{\Sigma_p(D) = \psi\ D\langle\vec{\alpha}{::}\vec{\kappa}, \vec{x}{:}\vec{\tau'}\rangle : T\langle\vec{\tau''},\vec{v'}\rangle \quad \forall i.\Sigma;\Delta;\Gamma \vdash \tau_i :: \kappa_i \quad \forall j.\Sigma;\Delta;\Gamma, X_j \vdash_p v_j : \tau'_j[\vec{\tau}/\vec{\alpha}][v_1 \dots v_{j-1}/x_1 \dots x_{j-1}]}{\Sigma;\Delta;\Gamma;X_1 \dots X_m \vdash_p D\langle\tau_1 \dots \tau_n, v_1 \dots v_m\rangle : T\langle\vec{\tau''},\vec{v'}\rangle[\vec{\tau}/\vec{\alpha}][\vec{v}/\vec{x}]}\ \text{(TT-New)}$$

$$\frac{\Sigma;\Delta;\Gamma;X \vdash_p v : ?T\langle\vec{\tau_3},\vec{v_3}\rangle \quad \Sigma_p(T\langle\vec{\tau_3},\vec{v_3}\rangle) = \tau_2\ m\langle\alpha{::}\kappa\rangle(x{:}\tau_1) \quad \Sigma;\Delta;\Gamma \vdash \tau :: \kappa \quad \Sigma;\Delta;\Gamma;X' \vdash_p v' : \tau_1[\tau/\alpha]}{\Sigma;\Delta;\Gamma;X,X' \vdash_p v.m\langle\tau\rangle(v') : \tau_2[\tau/\alpha][v'/x]}\ \text{(TT-App)}$$

$$\frac{\begin{array}{c}\Sigma;\Delta;\Gamma;X_1 \vdash_p x : ?\tau \quad \Sigma_p(D) = \psi\ D\langle\vec{\alpha}{::}\vec{\kappa}, \vec{x'}{:}\vec{\tau'}\rangle : T\langle\vec{\tau''},\vec{v'}\rangle \quad \Sigma;\Delta;\Gamma, \vec{x}{:}\vec{\tau'}[\vec{\tau}/\vec{\alpha}, x_1..x_{i-1}/x'_1..x'_{i-1}];\vec{x} \vdash_p D\langle\vec{\tau},\vec{x}\rangle : \tau \\ \Delta;\Gamma, \vec{x} : \vec{\tau'}[\vec{\tau}/\vec{\alpha}, x_1..x_{i-1}/x'_1..x'_{i-1}], x \cong D\langle\vec{\tau};\vec{x}\rangle; X_2, \vec{x} \vdash_p e_t : \tau' \quad \Sigma;\Delta;\Gamma;X_2 \vdash_p e_f : \tau'\end{array}}{\Sigma;\Delta;\Gamma;X_1,X_2 \vdash_p x \text{ isinst } D\langle\vec{\tau},\vec{x}\rangle \text{ then } e_t \text{ else } e_f : \tau'}\ \text{(TT-Isinst-x)}$$

$$\frac{\begin{array}{c}\Sigma;\Delta;\Gamma;X_1 \vdash_p D' < ts, vs >: \tau \quad \Sigma_p(D) = \psi\ D\langle\vec{\alpha}{::}\vec{\kappa}, \vec{x'}{:}\vec{\tau'}\rangle : T\langle\vec{\tau''},\vec{v'}\rangle \quad \Sigma;\Delta;\Gamma;X_2 \vdash_p e_f : \tau' \\ \Sigma;\Delta;\Gamma, \vec{x}{:}\vec{\tau'}[\vec{\tau}/\vec{\alpha}, x_1..x_{i-1}/x'_1..x'_{i-1}];\vec{x} \vdash_p D\langle\vec{\tau},\vec{x}\rangle : \tau \quad \Delta;\Gamma, \vec{x} : \vec{\tau'}[\vec{\tau}/\vec{\alpha}, x_1..x_{i-1}/x'_1..x'_{i-1}]; X_2, \vec{x} \vdash_p e_t : \tau'\end{array}}{\Sigma;\Delta;\Gamma;X_1,X_2 \vdash_p D' < ts, vs > \text{ isinst } D\langle\vec{\tau},\vec{x}\rangle \text{ then } e_t \text{ else } e_f : \tau'}\ \text{(TT-Isinst-obj)}$$

$\Sigma;\Delta;\Gamma \vdash t \cong t' \quad \Sigma;\Delta;\Gamma \vdash v \cong v'$ **Type and value equivalence**

$$\frac{}{\Sigma;\Delta;\Gamma \vdash \tau \cong \tau}\ \text{(TE-Id)} \qquad \frac{}{\Sigma;\Delta;\Gamma \vdash \backslash x{:}\tau.\tau'\ v \cong \tau'[v/x]}\ \text{(TE-Beta)} \qquad \frac{\Sigma;\Delta;\Gamma \vdash \tau \cong \tau' \quad \Sigma;\Delta;\Gamma \vdash v \cong v'}{\Sigma;\Delta;\Gamma \vdash \tau\ v \cong \tau'\ v'}\ \text{TE-Dep}$$

$$\frac{}{\Sigma;\Delta;\Gamma \vdash v \cong v}\ \text{(EE-Id)} \qquad \frac{v \cong v' \in \Gamma \text{ or } v' \cong v \in \Gamma}{\Sigma;\Delta;\Gamma \vdash v \cong v'}\ \text{(EE-Match)} \qquad \frac{\forall i,j\ \Sigma;\Delta;\Gamma \vdash \tau_i \cong \tau'_i \quad \Sigma;\Delta;\Gamma \vdash v_j \cong v'_j}{\Sigma;\Delta;\Gamma \vdash D\langle\tau_1 \dots \tau_n, v_1 \dots v_m\rangle \cong D\langle\tau'_1 \dots \tau'_n, v'_1 \dots v'_m\rangle}\ \text{(EE-Obj)}$$

$\Sigma;\Delta;\Gamma;X \vdash_p \tau\ m\langle\alpha{::}\kappa\rangle(x{:}\tau)\{e\}$ **Well-formedness of methods**

$$\frac{\begin{array}{c}\Sigma;\Delta;\Gamma \vdash \kappa \quad \Sigma;\Delta,\alpha :: \kappa;\Gamma \vdash \tau : \star \text{ or } \mathtt{A} \\ \Sigma;\Delta,\alpha :: \kappa;\Gamma, x : \tau; X, x \vdash_p e : \tau\end{array}}{\Sigma;\Delta;\Gamma;X \vdash_p \tau\ m\langle\alpha{::}\kappa\rangle(x{:}\tau)\{e\}}\ \text{(WF-Method)}$$

$\Sigma \vdash \psi\ D\langle\vec{\alpha}{::}\vec{\kappa}, \vec{x}{:}\vec{\tau}\rangle{:}T\langle\vec{\tau},\vec{v}\rangle\{\overrightarrow{\text{fdcl}}, \overrightarrow{\text{mdcl}}\}$ **Well-formedness of data class declarations**

$$\frac{\begin{array}{c}\Sigma;\vec{x} :: \vec{\kappa};\vec{x} : \vec{\tau} \vdash T\langle\vec{\tau},\vec{v}\rangle \quad \overrightarrow{\text{fdcl}} \subseteq fdecls(T\langle\vec{\tau},\vec{v}\rangle) \quad \overrightarrow{\text{mdcl}} \subseteq mdecls(T\langle\vec{\tau},\vec{v}\rangle) \\ \forall fi : ti \in \overrightarrow{\text{fdcl}}, \Sigma;\vec{x} :: \vec{\kappa};\vec{x} : \vec{\tau};\vec{x} \vdash_p \tau_i \quad \forall \text{mdcl} \in \overrightarrow{\text{mdcl}}, \Sigma;\vec{x} :: \vec{\kappa};\vec{x} : \vec{\tau};\vec{x} \vdash_p \text{mdcl}\end{array}}{\Sigma \vdash_p \psi\ D\langle\vec{\alpha}{::}\vec{\kappa}, \vec{x}{:}\vec{\tau}\rangle{:}T\langle\vec{\tau},\vec{v}\rangle\{\overrightarrow{\text{fdcl}}, \overrightarrow{\text{mdcl}}\}}\ \text{(WF-Ddecl)}$$

Figure 13. Static semantics of DCIL (complete)

p-Evaluation context E_p ::= $\bullet$ | let $x = E_p$ in e_2 Store M ::= $(x, v_p), M \mid \cdot$

$$\langle v_p\rangle_p \overset{p}{\rightsquigarrow} v_p \text{ (TE-Strip)} \qquad \langle\langle v_q\rangle_q\rangle_r \overset{p}{\rightsquigarrow} \langle v_q\rangle_q \text{ (TE-Nest)}$$

$$\frac{e \overset{q}{\rightsquigarrow} e'}{\langle e\rangle_q \overset{p}{\rightsquigarrow} \langle e'\rangle_q} \text{ (TE-Br)} \qquad \frac{\Sigma; \cdot; \cdot; \cdot \vdash v_p : \tau \qquad \Sigma; \cdot; \cdot \vdash \tau :: \mathtt{A} \qquad M' = M, (x, v_q) \qquad x \text{ fresh}}{M, v_p \overset{p}{\rightsquigarrow} M', x} \text{ (TE-Construct)}$$

$$\frac{M = M', (x, v_q)}{M, x \overset{p}{\rightsquigarrow} M', v_q} \text{ (TE-Destruct)} \qquad \frac{M, e \overset{p}{\rightsquigarrow} M', e'}{M, E_p[e] \overset{p}{\rightsquigarrow} M', E_p[e']} \text{ (TE-Cong)} \qquad \frac{e \overset{p}{\rightsquigarrow} e'}{M, E_p[e] \overset{p}{\rightsquigarrow} M, E_p[e']} \text{ (TE-Pure)}$$

$$\frac{}{D\langle\vec{\tau}, \vec{v}\rangle.f_i \overset{p}{\rightsquigarrow} v_i} \text{ (TE-Fld)} \qquad \frac{\Sigma_p(D\langle\vec{\tau}, \vec{v}\rangle) = \tau\ m\langle\alpha{::}\kappa\rangle(x{:}\tau')\{e\}}{D\langle\vec{\tau}, \vec{v}\rangle.m\langle\tau\rangle(v) \overset{p}{\rightsquigarrow} e[\tau/\alpha, v/x]} \text{ (TE-App)}$$

$$\frac{v = D\langle\vec{\tau}, \vec{v}\rangle \Rightarrow e = e_t[\vec{v}/\vec{x}] \qquad v = D'\langle\vec{\tau}', \vec{v}\rangle, D \neq D' \Rightarrow e = e_f}{v \text{ isinst } D\langle\vec{\tau}, \vec{x}\rangle \text{ then } e_t \text{ else } e_f \overset{p}{\rightsquigarrow} e} \text{ (TE-Isinst)} \qquad \frac{}{\text{let } x = v \text{ in } e \overset{p}{\rightsquigarrow} e[v/x]} \text{ (TE-Let)}$$

Figure 14. Dynamic semantics of DCIL

$\|\kappa\| = \kappa'$ **Translation of kinds**

$$\|\star\| = \star \quad \|\mathtt{A}\| = \mathtt{A}$$

$S;\Gamma;F \vdash \|T\| \leadsto \text{cdcl} \quad S;\Gamma;F \vdash \|T,\kappa,$ **Translation of type constructors**

$$\frac{}{S;\Gamma;F \vdash \|T\| \leadsto \|T, S(T), [\,]\|}\ \text{(Tr-Tyc)} \qquad \frac{\kappa = \star \text{ or } \mathtt{A} \quad \alpha \text{ fresh} \quad S;\Gamma;F \vdash \|T,\kappa',(tvs,\alpha :: \|\kappa\|)\| \leadsto \text{cdcl}}{S;\Gamma;F \vdash \|T, \kappa \to \kappa', tvs\| \leadsto \text{cdcl}}$$

$$\frac{x \text{ fresh} \quad S;\Gamma;\cdot;F \vdash \|\tau\| \leadsto s \quad S;\Gamma;F \vdash \|T,\kappa,(tvs, x : s)\| \leadsto \text{cdcl}}{S;\Gamma;F \vdash \|T, \tau \to \kappa, tvs\| \leadsto \text{cdcl}} \qquad \frac{\kappa = \star \text{ or } \mathtt{A}}{S;\Gamma;F \vdash \|T,\kappa,tvs\| \leadsto \text{public } T\langle tvs\rangle :: \kappa\{\}}$$

$S;\Gamma;F \vdash \|D\| \leadsto \text{ddcl} \quad S;\Gamma;F \vdash \|D, ($ **Translation of data constructors**

$$\frac{}{S;\Gamma;F \vdash \|D\| \leadsto \|D, S(D), [\,]\|}\ \text{(Tr-D)} \qquad \frac{S;\Gamma;F \vdash \|D,(p,\tau),(tvs,\alpha :: \|\kappa\|)\| \leadsto \text{ddcl}}{S;\Gamma;F \vdash \|D,(p,\forall\alpha{::}\kappa.\tau), tvs\| \leadsto \text{ddcl}}$$

$$\frac{S;\Gamma;F \vdash \|\tau\| \leadsto s \quad S;\Gamma;F \vdash \|D,(p,\tau'),(tvs, x : s)\| \leadsto \text{ddcl}}{S;\Gamma;F \vdash \|D,(p, x{:}\tau \to \tau'), tvs\| \leadsto \text{ddcl}} \qquad \frac{\tau = T\,\vec{\tau}\,\vec{x} \quad S;\Gamma;F \vdash \|\tau\| \leadsto s \quad \psi = \text{public if } p = \bot \text{ and internal otherwise}}{S;\Gamma;F \vdash \|D,(p,\tau),tvs\| \leadsto \psi\ D\langle tvs\rangle : s\{\}}$$

$S;\Gamma;F \vdash \|\tau\| \leadsto \tau'$ **Translation of types**

$$\frac{}{S;\Gamma;F \vdash \|\alpha\| \leadsto \alpha}\ \text{(Tr-tvar)} \qquad \frac{S;\Gamma;F \vdash \|\tau\| \leadsto \tau'}{S;\Gamma;F \vdash \|!\tau\| = !\tau'}\ \text{(Tr-afn)}$$

$$\frac{\begin{array}{c} C = \mathsf{DepArrow} \text{ if } S;\Gamma \vdash \tau : \star \text{ and } S;\Gamma \vdash \tau' : \star \\ C = \mathsf{DepArrowSA} \text{ if } S;\Gamma \vdash \tau : \star \text{ and } S;\Gamma \vdash \tau' : \mathtt{A} \\ C = \mathsf{DepArrowAA} \text{ if } S;\Gamma \vdash \tau : \mathtt{A} \text{ and } S;\Gamma \vdash \tau' : \mathtt{A} \\ S;\Gamma;F \vdash \|\tau\| \leadsto s \quad S;\Gamma, x:\tau;F \vdash \|\tau'\| \leadsto s' \end{array}}{S;\Gamma;F \vdash \|x{:}\tau \to \tau'\| \leadsto C\langle s, \backslash x{:}s.s'\rangle}\ \text{(Tr-tdep)}$$

$$\frac{\begin{array}{c} S;\Gamma;F \vdash \|\tau\| \leadsto s \quad \alpha - \text{lifting of } s = \vec{\alpha}(\vec{\tau},\tau') \\ \text{public } \mathsf{All}_{\vec{\alpha}}\langle\vec{\alpha}\rangle :: \star\{\tau' \ \mathsf{TyApp}\langle\alpha :: \|\kappa\|\rangle()\{\}\} \end{array}}{S;\Gamma;F \vdash \|\forall\alpha{::}\kappa.\tau\| \leadsto \mathsf{All}_{\vec{\alpha}}\langle\vec{\tau}\rangle}\ \text{(Tr-tforall)} \qquad \frac{\begin{array}{c} \forall i = 1..m, S;\Gamma;F \vdash \|\tau_i\| \leadsto \tau_i' \\ \forall j = 1..n, S;\Gamma;\cdot;F \vdash \|v_j\| \leadsto v_j' \end{array}}{S;\Gamma;F \vdash \|T\,\vec{\tau}\,\vec{v}\| \leadsto T\langle\vec{\tau'},\vec{v'}\rangle}\ \text{(Tr-tapp)}$$

$S;\Gamma;X;F \vdash_p \|e : \tau\| \leadsto e'$ **Translation of expressions**

$$\frac{\begin{array}{c} \forall i = 1..m, S;\Gamma;F \vdash \|\tau_i\| \leadsto s_i \\ \forall j = 1..n, S;\Gamma;X;F \vdash_p \|v_j\| \leadsto v_j' \end{array}}{S;\Gamma;X;F \vdash_p \|D\,\vec{\tau}\,\vec{v} : \tau\| \leadsto D\langle\|\vec{s}\|, \|\vec{v'}\|\rangle}\ \text{(Tr-Obj)} \qquad \frac{}{S;\Gamma;X;F \vdash_p \|x : \tau\| \leadsto x}\ \text{(Tr-X)} \qquad \frac{}{S;\Gamma;\cdot;(f : \tau_f, F) \vdash_p \|f : \tau_f\| \leadsto \mathsf{this}}\ \text{(Tr-F)}$$

$$\frac{\begin{array}{c} S;\Gamma, x{:}(p,\tau_1);X,x;F \vdash_p \|e : \tau_2\| \leadsto e' \quad \mathsf{C} \text{ is a fresh class name} \quad S;\Gamma;F \vdash \|\tau_1\| \leadsto s_1 \quad S;\Gamma, x{:}(p,\tau_1);X,x;F \vdash \|\tau_2\| \leadsto s_2 \\ \text{internal } \mathsf{C}\langle tvars(\Gamma), vvars(\Gamma)\rangle : \|Q(X, x{:}\tau_1 \to \tau_2)\|\{s_2 \ \mathsf{App}(x : s_1)\{e'\}\} \end{array}}{S;\Gamma;X;F \vdash_p \|\lambda x{:}\tau_1.e : Q(X, x{:}\tau_1 \to \tau_2)\| \leadsto \mathsf{C}\langle tvars(\Gamma), vvars(\Gamma)\rangle}\ \text{(Tr-Fun)}$$

$$\frac{\begin{array}{c} S;\Gamma,\alpha{::}\kappa;X;F \vdash_p \|e : \tau'\| \leadsto e' \quad \mathsf{C} \text{ is a fresh class name} \quad S;\Gamma,\alpha :: \kappa;F \vdash \|\tau'\| \leadsto s' \\ \text{internal } \mathsf{C}\langle tvars(\Gamma), vvars(\Gamma)\rangle : \|Q(X, \forall\alpha{::}\kappa.\tau')\|\{s' \ \mathsf{TyApp}\langle\alpha :: \|\kappa\|\rangle()\{e'\}\} \end{array}}{S;\Gamma;X;F \vdash_p \|\Lambda\alpha{::}\kappa.e : Q(X, \forall\alpha{::}\kappa.\tau')\| \leadsto \mathsf{C}\langle tvars(\Gamma), vvars(\Gamma)\rangle}\ \text{(Tr-Uni)}$$

$$\frac{S;\Gamma;X;F \vdash_p \|e_1 :?x{:}\tau_1 \to \tau_2\| \leadsto e_1' \quad S;\Gamma;X';F \vdash_p \|e_2 : \tau_1\| \leadsto e_2'}{S;\Gamma;X,X';F \vdash_p \|e_1\ e_2 : \tau_2[e_2/x]\| \leadsto \text{let } x_1 = e_1' \text{ in let } x_2 = e_2' \text{ in } x_1.\mathsf{App}(x_2)}\ \text{(Tr-App)}$$

$$\frac{S;\Gamma, f{:}(p,\tau);\cdot;(f : \tau, F) \vdash_p \|v_p : \tau\| \leadsto e'}{S;\Gamma;\cdot;F \vdash_p \|\text{fix } f{:}\tau.v_p : \tau\| \leadsto e'}\ \text{(Tr-Fix)} \qquad \frac{S;\Gamma;X;F \vdash_p \|e :?\forall\alpha{::}\kappa.\tau\| \leadsto e' \quad S;\Gamma;F \vdash \|\tau'\| \leadsto s'}{S;\Gamma;X;F \vdash_p \|e\ \tau' : \tau[\tau'/\alpha]\| \leadsto \text{let } x = e' \text{ in } x.\mathsf{TyApp}\langle s'\rangle()}\ \text{(Tr-TApp)}$$

$$\frac{\begin{array}{c} S;\Gamma;X;F \vdash_p \|v_p : \tau'\| \leadsto e' \quad S(D) = \forall\vec{\alpha} : \vec{\kappa}, (y1 : \tau_1') \to \ldots \to (yn : \tau_n') \to \tau_D \quad S;\Gamma, x_i{:}(p,\tau_i'[\vec{\tau}/\vec{\alpha}]);\vec{x} \vdash_p D\,\vec{\tau}\,\vec{x} : \tau' \\ S;\Gamma, x_i{:}(p,\tau_i'[\vec{\tau}/\vec{\alpha}]), v_p \doteq D\,\vec{\tau}\,\vec{x};X',\vec{x};F \vdash_p \|e_1 : \tau\| \leadsto e_1' \quad S;\Gamma;X';F \vdash_p \|e_2 : \tau\| \leadsto e_2' \end{array}}{S;\Gamma;X,X';F \vdash_p \|\text{match } v_p \text{ with } D\,\vec{\tau}\,\vec{x} \to e_1 \text{ else } e_2 : \tau\| \leadsto \text{let } x = e' \text{ in } x \text{ isinst } \|D\,\vec{\tau}\,\vec{x}\| \text{ then } e_1' \text{ else } e_2'}\ \text{(Tr-Match)}$$

Class declarations used in the translation

$$\text{public } \mathsf{DepArrow}\langle\alpha_1 : \star, \alpha_2 : \alpha_1 \to \star\rangle :: \star\{(\alpha_2\ x)\ \mathsf{App}(x : \alpha_1)\{\}\}$$

$$\text{public } \mathsf{DepArrowSA}\langle\alpha_1 : \star, \alpha_2 : \alpha_1 \to \mathtt{A}\rangle :: \star\{(\alpha_2\ x)\ \mathsf{App}(x : \alpha_1)\{\}\}$$

$$\text{public } \mathsf{DepArrowAA}\langle\alpha_1 : \mathtt{A}, \alpha_2 : \alpha_1 \to \mathtt{A}\rangle :: \star\{(\alpha_2\ x)\ \mathsf{App}(x : \alpha_1)\{\}\}$$

Figure 15. Translation from FINE to DCIL