Programming with Delegation

Abstract

It is often difficult to implement and maintain supporting functionality that is intertwined with the core program. For instance, ensuring information flow security requires global reasoning about how seemingly unrelated interactions in the core program can leak information. There is rich literature on technologies that can check the correctness of these interactions, but the programmer is fully responsible for producing correct programs. Information flow type systems can check that the program does not leak values, but the programmer is responsible for designing safe code. To address this problem, we present *programming with delegation*, a model that allows the programmer to transfer responsibility for parts of the program. The programmer transfers authority to the runtime system by introducing nondeterminism via *deferred* values, which the system determines according to declarative constraints that can refer to contexts in which the values will be used. The programming model builds on constraint functional logic programming, extended with implicitly bound parameters. To efficiently support this model, we present an execution strategy that takes advantage of the assumption that the program is mostly deterministic. The interpreter performs symbolic execution only when necessary to handle deferred values, and invokes an external SMT solver to resolve instances of nondeterminism. In this paper, we describe the semantics for programming with delegation, describe the implementation of an efficient interpreter, and demonstrate how to use programming with delegation for information flow and other related cross-cutting concerns.

1. Introduction

In software development, innovation is often at odds with robustness and security; the rapid prototyping and experimentation involved in developing new features precludes the kind of careful design and reasoning that leads to stable and secure systems. Eddie Kohler—the creator of the HotCRP system that runs this conference—reflected on this conflict in relationship to a serious security bug in his system. In his words, "More security-minded programmers wouldn't have perpetrated this bug, but they might not have written the system in the first place: the mindset required to write new features seriously differs from the mindset required to enumerate how combinations of features can be abused"[6]. This basic tension often means that there is no room in the timeline for both innovation and robustness. The race to deliver features to the market faster has more than once resulted in the release of software with serious security vulnerabilities [cite].

One reason for this tension between innovation and robustness is a lack of modularity between a core program and its cross-cutting concerns. Specifically, security and robustness properties are cross-cutting concerns that require coordination across the entire system.

[Copyright notice will appear here once 'preprint' option is removed.]

In the case of information flow, determining whether an assignment from a variable x to y can leak information requires us to be aware of the confidentiality policies of every value that may have been involved in the computation of x and ensure that they are consistent with the use of every value that may eventually depend on y. This lack of modularity makes it difficult to retrofit information flow security into a system not initially designed to accommodate it.

In this paper, we present *programming with delegation*, a model that supports flexible implementation of cross-cutting concerns by allowing the programmer to selectively transfer responsibility to the system. Our model allows the programmer to write a core functional program and separately provide declarative constraints as a specification for how the system should handle cross-cutting functionality. In order for the system to guarantee that the behavior of the program will always satisfy the specification, it needs to be given some control over the program execution. The programmer provides this control by introducing nondeterminism via *delegated* values that the system can control. Deferred values and constraints allow the programmer to delegate cross-cutting concerns to the runtime with minimal changes to the code.

We have implemented this model in a language Jeeves that extends an ML-like functional language with 1) constructs **defer** and **concretize** and 2) support for implicit contexts. For example, to ensure that a password is only seen by authorized users, the programmer can introduce a delegated expression with the appropriate constraint.

```
defer passwd
  { if (Authorized context.viewer)
      then (passwd == some realpasswd)
      else (passwd == none) }
```

This constraint says that the variable passwd is an option type that is either equal to **some** of the real password or to **none** based on whether the viewer is authorized. The variable **context** is an implicit parameter that allows the constraint to refer to the context in which the delegated variable will flow. Deferred expressions of type τ can be used interchangeably with expressions of type τ: the system is responsible for propagating delegated expressions through computations and requiring the appropriate concretization context. The programmer can call **concretize** with different contexts, allowing the system to ensure that the correct version of the value is revealed when, for example, sending passwd to the owner of the password and to unauthorized users. Jeeves allows the programmer to implement the core functionality independently of this and other security policies. This provides the "flexible information flow layer" Kohler calls for when describing this exact bug in the HotCRP system [6].

Programming with delegation builds on functional logic programming [11] extended with implicit parameters [9]. To efficiently support this model, we present an execution strategy that takes advantage of the assumption that the program is mostly deterministic. The programming idiom we anticipate is that the programmer will write a functional core program and introduce delegated values to handle sensitive information. We describe an interpreter that performs symbolic execution only when necessary to handle delegated values and performs eager simplifications to reduce the size of the symbolic state. The execution engine invokes an external SMT solver to resolve instances of nondeterminism.

In this paper, we describe the semantics for programming with delegation, describe the implementation of an efficient interpreter, and demonstrate how to use programming with delegation for information flow and other related cross-cutting concerns. Our main contributions are as follows:

- We present *programming with delegation*, a programming model that provides modularity for cross-cutting concerns.
- We demonstrate how to use programming with delegation to handle access control and information flow.
- We present a formalization of this programming model in terms of Jeeves, an ML-style functional language extended with constraints and implicit contexts.
- We describe an efficient execution model and interpreter implementation. We report performance on using programming with delegation for information flow and data processing examples.

We begin by demonstrating how programming with delegation works in the context of access control and information flow. We then describe the semantics of underlying programming model: how we combine constraint functional programming and implicit parameters to support **defer** and **concretize**. We then discuss useful idioms for programming in Jeeves and the implementation of the Jeeves interpreter, which uses the SMT solver Yices in the back end. We then describe an additional case study in using Jeeves for the cross-cutting concern of handling missing values in a census data processing example.

2. Information Flow with Delegation

We show how to use programming with delegation and Jeeves to handle access and information control in the context of a simple social network example.

Consider a social network where we have users uploading photographs with tags. Users can upload photographs and tag their friends to associate them with the photos. We store users and photographs as follows:

```
data user { uid      : user_id
          ; photos   : [photo_id]
          ; friends  : [user_id] }
data photo { pid     : photo_id
           ; image   : jpg
           ; owner   : user_id
           ; tagged  : [user_id] }
```

To get the list of users tagged in a photograph, it is simple to write the following function.

```
let get_photo_tags (p : photo) = p.tagged
```

The goal is to be able to implement access control and information flow with minimal changes to this function.

2.1 Information Flow without Delegation

Consider what happens without delegation if we want to add policies about who is allowed to view the photos and tags. For instance, user may want to restrict who can see the tags on a photograph. A general strategy for implementing permissions involves checking accesses to sensitive data:

```
(* Function that returns true if two users are friends. *)
let is_friends (u : user_id) (other : user_id) : bool =
  elem users[u].friends other

let get_photo_tags_w_viewer
  (p : photo) (viewer  : user_id) =
  filter (\(tag : user_id) -> is_friends viewer tag) p.tagged
```

This strategy for enforcing permissions requires tracking permissions and use context throughout the program. The first issue is that the policy on who can see the tags is associated with the function to retrieve the tags rather than the tags themselves. To add or modify policies about tags, the programmer is required to make sure all parts of the program enforce these policies. A second issue is that it is also now important to keep track of the user to whom the tags will be displayed.

2.2 Programming with Delegation

Programming by delegation allows the programmer to introduce *delegated values* to handle part of the program via declarative constraints. In this section we introduce the mechanisms for introducing nondeterminism through the **defer** construct and determinizing via **concretize**. We also describe the role of *implicit contexts* for allowing constraints to refer to the contexts in which the delegated values will be used.

2.2.1 Defer and Concretize

Programming with delegation is based on the idea of introducing nondeterministic delegated values that are associated with constraints. These constraints are used when the programmer later requests a concrete value for the delegated variable. We can introduce a delegated variable x as follows:

```
let x : delegated int = defer x' { x' > 0 }
```

This binds the variable x to a delegated variable that is greater than 0. We introduce the variable x' to be able to refer to it in the constraint. To get a concrete value for x, we can write

```
let result : option int = concretize x
```

This sets result to a value consistent with the constraints on x. The function **concretize** has type delegated $\tau \to$ **option** τ and returns a value consistent with the constraint environment if one exists. The result is an option type, returning **some** v if there is a satisfying assignment v and **none** otherwise.

As we saw above, Jeeves constraints can refer to not just the delegated variable,but to other variables in scope. This allows the programmer to add constraints to delegated variables:

```
let result : option int =
  let _ : unit = { x > 42 } in
    concretize x
```

This allows the programmer to refine the constraints on a delegated value after its creation.

A key advantage is that delegated values can be used interchangeably with core program values. We have the subtyping relationship delegated $\tau <: \tau$: any function that expect type τ can take a value of type delegated τ. For example, we can write:

```
let result : option int = concretize (x + 42)
```

The execution of the program can also further constrain the value of delegated variables. For example, the following use of x constrains it to be at least 58:

```
let g (a : int) (b : int) : int =
  defer z { (z == a + b) && (z > 100) }
let sum' : option int = concretize (f x 42)
```

If the constraints are not satisfiable, the call to **concretize** will return **none**.

2.2.2 Delegation Contexts

To allow constraints to refer to the context in which a a variable is concretized, the programming model supports *implicit contexts*. For example we can define a context that is a flag determining what "mode" the computation is in:

```
data mode = POS | NEG
data ctxt { m : mode }
```

The context is an implicit argument to the constraint and needs to be supplied when calling **concretize**. To allow constraints to refer to the context variable Jeeves provides the **context** keyword:

```
let x : delegated ?ctxt int =
  defer x
    { if (context.m == POS) then (x > 0) else (x < 0) }
```

The call to code concretize must supply the appropriate context:

```
let result : int =
  let c : ctxt = { m = NEG } in
    concretize c (x + 42)
```

This would give us the constraint x < 0, yielding a result less than 42. The system propagates the context types; programmer does not have to write the annotations explicitly. Note that separate concretizations are independent. We could have the following value that also depends on val:

```
let result ' : int = concretize { m = POS } val
```

These are independent calls to **concretize**: val does not have to be same value for these two calls.

2.2.3 Bounded Quantification

Jeeves supports bounded quantification to allow the programmer to easily express policies that apply to all elements of some data structure:

```
let lst : list int = [0, 1, defer x, 3]
let _ : unit = { forall elt in lst . (elt > 0) }
```

An example of something this facilitates is a mechanism for granting access to some piece of sensitive data to all users some list.

2.3 Information Flow with Delegation

We will now show how to delegate the cross-cutting concerns of access control and information flow by introducing delegated values with contexts.

2.3.1 Basic access control

It is straightforward to use delegation and contexts to handle access control. We can define a context containing the viewer of the protected piece of data.

```
data vctxt { viewer : user_id }
```

We can delegate the access control decision to the system. Below is code showing showing how we can enforce the policy that a viewer can only access tags if the viewer is friends with the user tagged.

```
let get_photo_tags_w_access (p : photo) =
  let can_see (tag : user_id) =
    defer acc { acc == is_friends tag context.viewer } in
     filter can_see p.tagged
```

The implicit context frees the programmer from having to propagate the context through the program. Next, we will show how the programmer can attach the policy to the tag when it is created rather than when it needs to be used. This allows for more separation between the core program and the security policies.

2.3.2 Information Flow

We can also use delegation to directly control the flow of sensitive values. Recall the definition of the photo data type:

```
data photo { pid    : photo_id
           ; image  : jpg
           ; owner  : user_id
           ; tagged : [user_id] }
```

Since a value of type delegated user_id can be used anywhere a photo tag is expected, we can store each tag as a delegated value and with a policy attached to what value is released. To create records with the policy attached, we can write the following function.

```
let add_photo_policy (p : photo) : photo =
  (* Function for adding a policy to a tag. *)
  let add_tag_policy (actual_tag : user_id) =
    defer t'
      { if ( is_friends actual_tag context.viewer)
          then (t' == actual_tag) else (t' == null) } in

  (* The new photo record. *)
  { photo_id = p.photo_id
  ; image = p.image
  ; owner = p.owner
  ; tagged = map add_tag_policy p.tagged }
```

This function takes a record of type photo and creates a new record where the tags have type delegated user_id and whose value depends on the tag policy. When concretized in a context with some viwer, this delegated value will reveal the actual tag. will reveal the actual tag if the viewer is friends with the owner and the **null** tag otherwise. Here, **null** is a keyword referring to the reserved ID belonging to no record.

We can use the same get_photo_tags : photo −> [user_id] function we defined before. If we have photos created as above where the tags are delegated values, then the call to get_photo_tags will return a value of type delegated [user_id]. The code below shows how we can use **concretize** to get a concrete list of tags visible to a user.

```
let show_tagged (v : user_id) (p : photo) : unit =
  (* Get tags. *)
  let tags : [user_id] = get_photo_tags p in
  (* Create a viewer context. *)
  let ctxt : vctxt = { viewer_id = v } in
    print_list (concretize ctxt tags)
```

This code will print a list of tags that only show a user if that user is friends with the viewer. Delegation allows the programmer to write the program so that the security policy only needs to appear in the creation of the photo record and when a value needs to be used (in this case, displayed); intermediate functions such as get_photo_tags can remain oblivious to the security concerns.

2.4 Substituting Different Policies

Because the information flow policies are attached to the values themselves, it is easy to change policies or use photo records with different policies. For instance, we may want to implement a policy that the viewer can see photo tags if the viewer is friends with the photo owner.

```
let add_photo_policy' (p : photo) : photo =
  let add_tag_list_policy (tags : [user_id]) =
    defer lst
      { if ( is_friends p.owner context.viewer)
          then ( lst == tags) else ( lst == []) } in

  { photo_id = p.photo_id
  ; image = p.image
  ; owner = p.owner
  ; tagged = add_tag_list_policy p.tagged }
```

We can easily have photo records with different policies on information flow of the different fields while using the same code for the rest of the program.

We can also easily combine policies: applying the composition add_photo_policy ∘ add_photo_policy' produces a photo record where a given tag is only visible if the viewer is friends with both the owner and the tagged user. A policy may also be affected by other policies. For example, if the friends field of a user record is protected by a policy that friends are only revealed in the case of mutual friends, then a policy regarding a photo tag that depends on is_friends of two users also incorporates the friend-privacy policy.

2.5 Combining Policies

Having constraints specifying access control and information flow allows the system to collect constraints along the way. Suppose that instead of having is_friends as the sole basis for the policies of who can view the tags, tag viewing can be determined by various factors. For example, perhaps a user can set a permission that their photo can be visible to anybody. We could take this into account in the tag permissions by checking some property of each user, but we can take advantage of the declarative nature of the Jeeves policies by using a can_view_tags user_tag **context**.viewer for specifying the tag policy.

```
let add_tag_policy (actual_tag : user_id) =
  defer t'
    { if (can_view_tag tag context.viewer)
      then (t' == actual_tag) else (t' == null) } in
```

We can define can_view_tag as a delegated value with a default policy that says a user can view a tag if the viewer is friends with the tagged user:

```
let can_view_tag (tag : user_id) (viewer : user_id) : bool =
  defer cvt { ( is_friends tag viewer) implies cvt }
```

The programmer can refine the policy for can_view_tag elsewhere in the proram. For instance, we could write the following function to refine the policy for a given user:

```
let expose_all_tags (u : user) : unit =
  { forall viewer in users . (can_view_tag u.uid viewer.uid) }
```

Being able to refine constraints also allows us to add policies about photo tags when other data structures change. For instance, suppose the programmer extended the system so that users belong to networks and wanted to allow users to set tags to be visible to all users in the same network. The programmer could add the following policy without touching the photo records:

```
let expose_to_network (u : user) : unit =
  { forall viewer in users .
      ( if (viewer.network == u.network)
        then (can_view_tag u.uid viewer.uid) else true) }
```

Being able to create delegated values with constraint-based specifications for sensitive values allows the programmer to extend both the system and the security policies with small, local changes.

3. The Jeeves Language

In this section we describe the surface Jeeves language, the core Jeeves language, and the translation from the surface language to the core language.

3.1 Surface Language

Jeeves extends a standard Hindley-Milner polymorphic functional language with 1) the constructs **defer** and **concretize** associated with constraints and 2) implicit contexts.

The expression **defer** n { γ_d } introduces a delegated value with an associated constraint γ_d that can refer to the current value as n. The constraint can also refer to the implicit context variable **context**. Deferred values are associated with an implicit context; a delegated expression of type τ has type $?\tau_c\tau$, where τ_c is the type of the context. The system checks that a context of type τ_c is supplied when there is a call to **concretize**. The expression **concretize** e_c e produces a core program value from e based on the constraints that have been introduced and the context expression e_c.

Delegated expressions of type τ can be used interchangeably with expressions of type τ, allowing the programmer to write the core program independent of delegated cross-cutting behaviors. The system evaluates delegated expressions as symbolic expressions, collecting constraints in order to derive consistent expressions when evaluating **concretize**.

3.2 Core Jeeves

We describe the semantics of programming with delegation in terms of core Jeeves, which is System F extended with **defer** and implicit contexts. The system has polymorphism that allows delegated expressions of type τ to be used as expressions of type τ. In the core programming model there is explicit propagation of the constraints used in concretization. We show the core Jeeves syntax in Figure 1; we describe the semantics in the next section. [*Say why we have this core model and what properties we have from it.*]

3.3 Translation to Core

There is a trivial translation from the surface syntax to a monadic representation that captures effects and makes constraint propagation explicit. We can then translate to core... [*Say more explicitly how this happens.*]

4. Core Jeeves

In this section we describe the core semantics of Jeeves. We describe how constraints are introduced, propagated, and used for concretization. We also describe the semantics fo implicit contexts for concretization. [*Describe how everything works at a high level and what people should get out of reading this section.*]

[*Figure out where you want to say this.*] Semantics are nondeterministic ($\mathcal{P}(\cdot)$ indicates the power set), but not implemented using backtracking; rather, symbolic execution and SMT solver. We eta-expand some denotations for clarity.

4.1 Static Semantics

[*Talk about the motivations for presenting the static semantics: what are we ruling out, what people should be looking for while reading the rules, etc.*] [*Cite the paper(s) this is based on and say how we are similar and different (at a high level).*]

We show the static semantics in Figure 2. [*Go through the rules, highlighting the non-standard ones and telling people how they should understand them.*]

[*Prose your way through safety theorems.*]

4.2 Dynamic Semantics

[*Talk about the motivations for presenting the dynamic semantics. What is the main idea? What is the high-level idea? What should people get out of reading the rules? How does this correspond to the actual implementation?*] [*Cite references and how we differ.*]

We show denonational rules for the dynamic semantic rules in Figure 3. [*Talk about the rules in more detail, spending more time on the unusal ones.*]

4.3 Extensions

[*Talk about these in a more positive way.*] **What is not in the language that should be in real Jeeves:**

- Priority of nondeterminism (i.e., $e_1 \,[]\, e_2$ with e_1 preferred, or $\{\!| e_1 \mid e_2 |\!\}_{\text{maximizing } e_3}$).
- Effects (in real language, would be treated monadically, and would require closed deterministic values as operands).

$$
\begin{array}{rcl}
e & ::= & x \mid ?x \mid \lambda x.e \mid e_1\ e_2 \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \mid e_1\ \mathbf{with}\ ?x = e_2 \\
 & & \mid \mathbf{defer}\ \tau \mid \{\!|\, e_1 \,|\, e_2 \,|\!\}\ e_{\text{sat}}\ e_{\text{unsat}} \\
 & & \mid n \mid b \mid e_1 :: e_2 \mid \mathbf{nil} \\
 & & \mid e_1\ \text{(arith)}\ e_2 \mid e_1\ \text{(cmp)}\ e_2 \mid e_1\ \text{(b\_logic)}\ e_2 \mid \text{(u\_logic)}\ e \\
 & & \mid \mathbf{list\_rec}\ e_c\ e_n\ e \mid \mathbf{word\_rec}\ e_z\ e_s\ e \mid \mathbf{bool\_rec}\ e_t\ e_f\ e \\
n & ::= & 0 \mid 1 \mid \ldots \mid M \\
b & ::= & \mathbf{true} \mid \mathbf{false} \\
\text{(arith)} & ::= & + \mid - \mid \times \\
\text{(cmp)} & ::= & = \mid \neq \mid \leq \mid \geq \mid < \mid > \\
\text{(b\_logic)} & ::= & \wedge \mid \vee \\
\text{(u\_logic)} & ::= & \neg \\
 & & \\
\tau & ::= & \alpha \mid \tau_1 \to \tau_2 \mid \mathbf{list}\ \tau \mid \mathbf{word} \mid \mathbf{bool} \\
\sigma & ::= & \forall \vec{\alpha}.C \Rightarrow \tau \\
C, D & ::= & ?x_1 : \tau_1, \ldots, ?x_n : \tau_n \\
x & ::= & \text{lambda- and let-bound variable names} \\
?x & ::= & \text{implicit variable names} \\
\alpha & ::= & \text{type variable names}
\end{array}
$$

Figure 1. Abstract syntax of core Jeeves.

$$\frac{\Gamma(x) = \tau}{C; \Gamma; \Delta \vdash x : \tau}\ \textsc{Var}$$

$$\frac{\Gamma(x) = (\forall \vec{\alpha}.D \Rightarrow \tau_r) \quad D[\vec{\tau}/\vec{\alpha}] \subseteq C}{C; \Gamma; \Delta \vdash x : \tau_r[\vec{\tau}/\vec{\alpha}]}\ \textsc{PolyVar}$$

$$\frac{C(?x) = \tau}{C; \Gamma; \Delta \vdash ?x : \tau}\ \textsc{ImplicitVar}$$

$$\frac{C; \Gamma; \Delta \vdash e_1 : \tau_1 \to \tau_2 \quad C; \Gamma; \Delta \vdash e_2 : \tau_1}{C; \Gamma; \Delta \vdash e_1\ e_2 : \tau_2}\ \textsc{App}$$

$$\frac{C; \Gamma[x \mapsto \tau_1]; \Delta \vdash e : \tau_2}{C; \Gamma; \Delta \vdash \lambda x.e : \tau_1 \to \tau_2}\ \textsc{Abs}$$

$$\frac{D; \Gamma; \Delta \vdash e_1 : \tau_1 \quad \sigma = \text{gen}(D, \Gamma, \tau_1) \quad C; \Gamma[x \mapsto \sigma]; \Delta \vdash e_2 : \tau_2}{C; \Gamma; \Delta \vdash (\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2) : \tau_2}\ \textsc{Let}$$

$$\frac{C[?x \mapsto \tau_2]; \Gamma; \Delta \vdash e_1 : \tau_1 \quad C; \Gamma; \Delta \vdash e_2 : \tau_2}{C; \Gamma; \Delta \vdash (e_1\ \mathbf{with}\ ?x = e_2) : \tau_1}\ \textsc{With}$$

$$\frac{}{C; \Gamma; \Delta \vdash \mathbf{defer}\ \tau : \tau}\ \textsc{Defer}$$

$$\frac{C; \Gamma; \Delta \vdash e_1 : \tau_1 \quad C; \Gamma; \Delta \vdash e_2 : \mathbf{bool} \quad C; \Gamma; \Delta \vdash e_{\text{sat}} : \tau_1 \to \tau_2 \quad C; \Gamma; \Delta \vdash e_{\text{unsat}} : \tau_2}{C; \Gamma; \Delta \vdash \{\!|\, e_1 \,|\, e_2 \,|\!\}\ e_{\text{sat}}\ e_{\text{unsat}} : \tau_2}\ \textsc{Concretize}$$

$$\frac{C; \Gamma; \Delta \vdash e_c : \tau_1 \to \tau_2 \to \tau_2 \quad C; \Gamma; \Delta \vdash e_n : \tau_2 \quad C; \Gamma; \Delta \vdash e : \mathbf{list}\ \tau_2 \quad \Delta \sim e}{C; \Gamma; \Delta \vdash \mathbf{list\_rec}\ e_c\ e_n\ e : \tau_2}\ \textsc{List\_Rec}$$

Figure 2. Static semantics of core Jeeves.

$$\mathcal{T}[\![\cdot]\!] \in \text{Exp} \to \text{Set}$$

$$\mathcal{T}[\![\textbf{bool}]\!] = \{\text{true}, \text{false}\}$$
$$\mathcal{T}[\![\textbf{word}]\!] = \{0, 1, 2, \ldots, M\}$$
$$\mathcal{T}[\![\textbf{list}\ \tau]\!] = \bigcup_{i \in \mathbb{N}} L_i(\tau), L_0(\tau) = \{()\}, L_{i+1}(\tau) = \mathcal{T}[\![\tau]\!] \times L_i(\tau)$$
$$\mathcal{T}[\![\tau_1 \to \tau_2]\!] = \mathcal{T}[\![\tau_1]\!] \to \mathcal{T}[\![\tau_2]\!]$$

$$\mathcal{E}[\![e]\!] \in ((\text{Ctxt}_{\Gamma,C} \to \text{ICtxt}_C \to \mathcal{T}[\![\tau]\!]) \to \mathcal{P}(\mathcal{T}[\![\tau]\!])) \to \mathcal{P}(\mathcal{T}[\![\tau]\!])$$

where $e \in \text{Exp}, C; \Gamma; \Delta \vdash e : \tau,$ and $\text{Ctxt}_{\Gamma,C}, \text{ICtxt}_C$ are contexts appropriate to Γ and Δ, resp.[1]

$$\mathcal{E}[\![x]\!]k = k(\lambda\rho.\lambda\gamma.\rho(x))$$
$$\mathcal{E}[\![?x]\!]k = k(\lambda\rho.\lambda\gamma.\gamma(?x))$$
$$\mathcal{E}[\![\lambda x.e]\!]k = \mathcal{E}[\![e]\!](\lambda r.k(\lambda\rho.\lambda\gamma.\lambda v.r(\rho[x \mapsto v])\gamma))$$
$$\mathcal{E}[\![e_1\ e_2]\!]k = \mathcal{E}[\![e_1]\!](\lambda r_1.\mathcal{E}[\![e_2]\!](\lambda r_2.k(\lambda\rho.\lambda\gamma.(r_1\ \rho\ \gamma)(r_2\ \rho\ \gamma))))$$
$$\mathcal{E}[\![\textbf{let}\ x = e_1\ \textbf{in}\ e_2]\!]k = \mathcal{E}[\![e_1]\!](\lambda r_1.\mathcal{E}[\![e_2]\!](\lambda r_2.k(\lambda\rho.\lambda\gamma.r_2(\rho[x \mapsto (\lambda\gamma'.r_1\ \rho\ \gamma')])\gamma)))$$
$$\mathcal{E}[\![e_1\ \textbf{with}\ ?x = e_2]\!]k = \mathcal{E}[\![e_2]\!](\lambda r_2.\mathcal{E}[\![e_1]\!](\lambda r_1.k(\lambda\rho.\lambda\gamma.r_1\rho(\gamma[?x \mapsto r_2\ \rho\ \gamma]))))$$
$$\mathcal{E}[\![\textbf{defer}\ \tau]\!]k = \bigcup_{v \in \mathcal{T}[\![\tau]\!]} k(\lambda\rho.\lambda\gamma.v)$$
$$\mathcal{E}[\![n]\!]k = k(\lambda\rho.\lambda\gamma.n)$$
$$\mathcal{E}[\![b]\!]k = k(\lambda\rho.\lambda\gamma.b)$$
$$\mathcal{E}[\![e_1 :: e_2]\!]k = \mathcal{E}[\![e_1]\!](\lambda r_1.\mathcal{E}[\![e_2]\!]\lambda r_2.k(\lambda\rho.\lambda\gamma.(r_1\ \rho\ \gamma, r_2\ \rho\ \gamma)))$$
$$\mathcal{E}[\![b]\!]k = k(\lambda\rho.\lambda\gamma.())$$

Figure 3. Dynamic semantics of core Jeeves.

[a] I.e., $\rho \in \text{Ctxt}_{\Gamma,C}$ if for all x free in Γ, $\hat{\gamma}(\rho(x)) \in \mathcal{T}[\![\Gamma(x)]\!]$ (where $\hat{}$ denotes extended substitution); and $\gamma \in \text{Ctxt}_C$ if for all $?x_i : \tau_i$ in C, $\gamma(?x_i) \in \mathcal{T}[\![\tau_i]\!]$.

- General recursion/nontermination (for illustrative purposes, we show that even with this kind of angelic nondeterminism, the language can be terminating, but real Jeeves should have general recursion).
- Arrays/finite maps.

5. Useful Idioms

In programming with Jeeves we have found certain idioms useful for associating constraints with values. We have built support for these idioms in the Jeeves interpreter.

5.1 Type-Level Constraint Functions

Many of introductory information flow examples involved functions that associate record types with constraints. A useful programming abstraction is *type-level constraint functions*, which allow the programmer to define constraints that apply to all delegated expressions of a given type.

Below is an example of how we can define a new type with an associated type-level constraint function:

```
data t = { f1 : int; f2 : int } with { this.f1 > this.f2 }
```

If either fields f1 or f2 of an expression e : t is a delegated expression, then this constraint is added to the constraint environment. The constraint can refer to the current value with the **this** keyword. The programmer can also define a type that attaches constraints to an existing type:

```
type t' = t with { this.f1 == this.f2 }
```

This defines a constraint function that incorporates the constraint already associated with type t. In this case, we have (redundantly) the constraints f1 > f2 and f1 == f2 for expressions of type t' when either field is a delegated expression. The type t' is a subtype of type t: expressions of type t' have the same fields as expressions of type t and satisfy at least as many constraints.

5.2 Hidden Values

In the information flow examples we described, we create deferred values that reveal the "actual" value if some constraint is satisfied. To be able to write type-level constraint functions that refer to the "actual" values, we introduce 1) the construct **hidden** for introducing a delegated expression that stores the "actual" value and 2) the operator ! for accessing the "actual" value from within a type-level constraint function. We can use this extension as follows:

```
data ctxt = { b : bool }
let x : int =
  hidden x 3 { if (context.b) then (x == !x) else (x == -1) }
```

Evaluation of **concretize** b = **true** x would yield 3; evaluation of **concretize** b = **false** x would yield −1.

With type-level constraint functions and hidden values, we can rewrite the policy-application functions from the introductory example by defining policy types like the following:

```
type photo_tagp = photo with
  { forall tag in this.tagged .
      (if (is_friends tag context.viewer)
          then (t' == actual_tag) else (t' == null)) }
```

Each of these policy-types can be used as code photo types.

5.3 Default Values

To set boundaries on nondeterminism it is useful to have *default* values, which we allow the programmer to supply when deferring a value:

```
let x : int = defer x' { x' > 0 } default 42
```

This binds x to a deferred value that is greater than 0. In the absence o f further constraints on x, concretization of x will yield 42. Def ault values are useful for ensuring that access controls are set to be more restr ictive by default.

Default values are useful in specifying access control and information flow policies. Recall the following function from the introductory example determining whether a user viewer has access to a given tag:

```
let can_view_tag (tag : user_id) (viewer : user_id) : bool =
  defer cvt { (is_friends tag viewer) ==> cvt }
```

The function does not constrain deferred value cvt to be false so as to allow other policies to set can_view_tag. It is useful, however, to be able to set the access to **false** in the absence of any constraints indicating otherwise. We can write this constraint as follows:

```
let can_view_tag (tag : user_id) (viewer : user_id) : bool =
  defer cvt { (is_friends tag viewer) ==> cvt } default false
```

If there are no constraints setting cvt to **true**, then the viewer ca nnot view the tag.

6. Implementation

The Jeeves interpreter is designed to minimize the size of the symbolic state and constraint environment. The interpreter consists of an OCaml evaluation loop, a C++ implementation of the environment for storing possibly symbolic expression, and the Yices constraint solver for resolving constraints. To improve programmability, the Jeeves interpreter supports the idioms we have describe.

6.1 Interpreter loop

The frontend performs preprocessing, drives the evaluation loop, and performs type-level constraint function applications. The interpreter loop is responsible for minimizing constraint derivation time by short-circuiting application of type-level constraints, either when it encounters constraints on concrete values or when implication conditions are false. To avoid unnecessary traversal of concrete data structures and to avoid building constraints that are not used, the interpreter loop only constructs constraints for the symbolic components of data structures. When traversing data structures to collect constraints the interpreter takes advantage of primary keys: if equality of a primary key field is a condition, the interpreter will check that equality to determine whether to continue evaluating the constraint.

6.2 Environment

The environment keeps track of currently live expressions being evaluated and their corresponding constraints, storing expressions as directed acyclic graphs of nodes. Two key factors in Jeeves's performance feasibility are the environment's *eager garbage collection* and *eager expression simplification*. The backend performs reference counting garbage collection; this is important because many expressions quickly become garbage due to the optimizations. Whenever it binds an expression, the environment performs two main optimizations: constant propagation and common subexpression elimination. Constant propagation simplifies all concrete portions of live expressions, allowing the environment state to remain small when most of the computation is concrete. Common sub-expression greatly reduces the state growth: for instance, the state now grows linearly rather than exponentially in the number of variable appearances in substitution. To combine common-subexpressions we have implemented the structural hashing techniques performed by SMT solvers such as UCLID [8] and STP [3]. The environment also supports common optimizations for data processing such as storing tables as hash tables indexed by primary key. Without implementing common-subexpression sharing environment can grow exponentially from storing symbolic expressions, by a factor of the number of variable appearances for each variable substitution.

# users	File proc. (s)	Eval. (s)	Yices (s)
128	0.004	0.004	0.000
256	0.016	0.004	0.000
512	0.084	0.004	0.000
1024	0.348	0.012	0.000
2048	1.540	0.024	0.000
4096	6.220	0.056	0.000
8192	30.634	0.151	0.000

Table 1. Times for resolving is_friends when one user is symbolic and each of n users is friends with each other.

6.3 Solver

The environment makes calls to the Yices SMT solver via the Yices C interface. The implementation uses basic interface functions to declare variables and assert constraints; it also uses the push_context and pop_context functions for helping with calls to **concretize** with different contexts.

Because the system accumulates small linear constraints that SMT solvers are optimized for solving, it makes the most sense to use an external solver to resolve these constraints. Note, however, that the reason we have small linear constraints is because of optimizations in the environment; it would not be feasible to use the solver directly for performing symbolic evaluation.

7. Information Flow Measurements

We demonstrate the feasibility of this execution model by taking some measurements of the performance of our prototype interpreter. We show that evaluation time is reasonable and constraint solving time, is as expected, negligible.

First consider the example where photo tags are visible to friends of the photo o wner. We can express this with the following type-level constraint function, which says that the list of tagged users is to be displayed only if the viewer is friends with the photo owner:

```
type photo_tagp = photo with
  { if (is_friends this.owner context.viewer)
      (this.tagged == !this.tagged) else (this.tag == [])) }
```

We ran some tests accessing on the times it takes to access a specific tag list for a specific viewer. During evaluation with on a users table of size n, **context**.viewer is a symbolic variable that

# users	Constraint deriv. (s)	Total eval. (s)	Yices (s)
128	0.512	0.528	0.008
256	2.396	2.456	0.040
512	10.128	10.377	0.016
1024	41.703	42.738	0.700

Table 2. Times for applying constraints to delegated photo tags.

can equal any of the n users, so the is_friends function creates a symbolic expression comparing **context**.viewer to all elements of the owner's friends list. We show in Table 1 that the interpreter can resolve the symbolic comparisons in less than one second (far less than the time it takes to process the file) and that the Yices solving time is negligible.

We also examine the case when we resolve permissions for each individual photo tag based on whether the viewer is friends with the tagged user:

```
type photo_tagp = photo with
  { forall t in this.tagged .
      if ( is_friends tag context.viewer)
        then (t == !t) else (t == null) }
```

Deriving the constraint for each tag involves creating a symbolic value based on the is_friends function. We show the running times in Table 2. These times show that evaluating the constraints takes up the bulk of evaluation time. We expect that engineering optimizations such as deriving constraints lazily and memoization can reduce these times.

8. Application: Data Processing

We demonstrate how to use programming with delegation for processing data with missing values. To evaluate the Jeeves language and interpreter implementation we implemented benchmarks that process real census data according to documented strategies for handling missing values in a way that does not bias the results. In this section, we describe the following.

- We measured the performance of the Jeeves interpreter on documented data processing strategies to process real data from the U.S. Census Bureau. The Jeeves interpreter was able to process almost 400,000 records with negligible constraint solving time; the performance scales linearly.
- We compared the Jeeves implementation to implementations in SQL and Python. We found that Jeeves has expressiveness advantages to both languages and that Jeeves performance is comparable to that of Python on the census data benchmarks.

8.1 Current Population Survey and imputation

We use data from the U.S. Census Bureau's Current Population Survey's [18] Annual Social and Economic (ASEC) March supplement. The Census Bureau documents standard procedures for reconstructing answers from nonresponse from other available information [19]. The documentation refers to the following strategies: *relational imputation* involves inferring the missing value from other characteristics on the person's record or within the household; *longitudinal edits* involve looking at the previous month's data; *"Hot deck"* imputation infers the missing value from a different record with similar characteristics. The supplement data sets contain preprocessed data that is annotated with a set of fields containing flags indicating how certain values from certain fields have been filled in.

8.1.1 Data Processing with Jeeves

We use selected columns of the Current Population Survey data [18] to use the following record type:

```
data cps_data { household      : int
              ; line_no        : int
              ; age            : int
              ; marital_status : int
              ; spouse_no      : int
              ; num_hot_lunch  : int}
```

Each household has a unique ID and the data identifies individuals according to their line number with respect to a household. Each individual has an integer age. Marital status can be integer values $0 - 9$ indicating statuses including "single, never married" and "widowed." The number of children in a household who get hot lunch is an integer that should be the same for all children in a household.

We tested the performance of the LogLog implementation on the following kinds of strategies for filling in missing values:

1. *Relational*. We derive missing marital status and spouse fields for a record with household ID h line number ℓ by looking at whether there is another record with household h listing ℓ as the spouse line number.
2. *Longitudinal*. We derive missing ages by looking up the value in a reference table.
3. *"Hot deck."* We derive a missing hot lunch number for a record with household ID h by using a value from another record with household ID h.
4. *Combined*. We combine these three strategies, creating a type with all three constraints and performing a sum over the age field conditional on the value of the "hot lunch" field.

For each example, we created a data set with missing values by examining the relevant flags. For example, for the relational test we use the flags for the marital status and spouse line number to set the implicated items as missing while leaving the other items intact. For the longitudinal example, we used the original data set as the reference table. The input data for the "combined" example has missing items in all relevant columns.

Below is a type defined in terms of cps_data that adds the relational constraints.

```
type cps_infer_relational = relation cps_data
  with
    { forall r in this .
        forall r' in this.
          ((r.household == r'.household) and
           (r.spouse_no == r'.line_no)) implies
            ((r.line_no == r'.spouse_no) and
             (r.marital_status == r'.marital_status)) }
```

This constraint function says that for each record in the census relation, if there is another record with the same household identifier that lists the current individual as the spouse, then they should have matching marital status and spouse information. We write the other constraints analogously.

8.1.2 Running times

Our results suggest that blended execution scales well on real-world applications. We ran the tests on increasingly large subsets of the data with the following total percentages of records with missing items:

Relational	Longitudinal	"Hot deck"	Combined
0.433%	1.445%	0.588%	1.450%

We reconstruct the missing fields by looking at the flags in the original data set. These percentages correspond to the actual documented missing data and how the released data sets were processed.

Total records	# bad	Relational		Longitudinal		"Hot deck"		Combined		
		Cnsts.	Total	Cnsts.	Total	Cnsts.	Total	Cnsts.	Yices	Total
8,192	66	0.02	0.08	0.04	0.08	0.02	0.06	**0.10**	**0.000**	0.16
16,384	152	0.06	0.15	0.07	0.16	0.06	0.12	**0.20**	**0.000**	0.32
32,768	344	0.25	0.33	0.14	0.33	0.12	0.26	**0.42**	**0.000**	0.67
65,546	1007	0.28	0.64	0.30	0.69	0.25	0.54	**0.88**	**0.000**	1.40
131,072	1990	0.57	1.30	0.64	1.46	0.53	1.11	**1.82**	**0.000**	2.87
262,144	3745	1.16	2.62	1.32	3.02	1.09	2.30	**3.86**	**0.000**	6.04
392,550	5693	1.81	4.09	2.04	4.62	1.67	3.54	**5.82**	**0.000**	9.04

Table 3. Times (in seconds) for 1) deriving constraints on the input with incomplete values (Cnsts.) and 2) running the interpreter on the entire program (Total). We show the total number of records on the left, along with the total number of missing records in the data set combining all missing items.

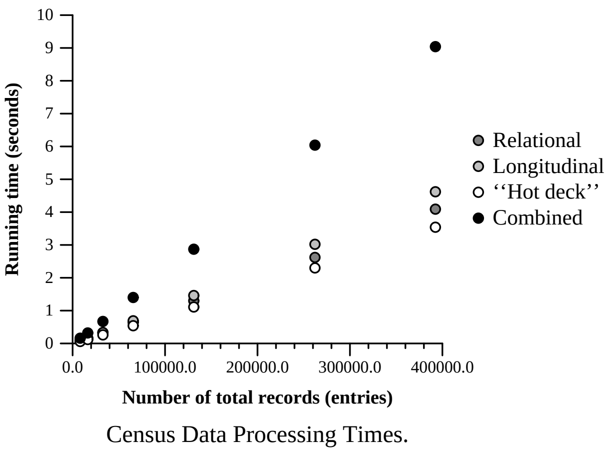

Figure 4. Jeeves data processing scales linearly with increasing input sizes, even in the presence of missing data.

We show the running times from a machine with an Intel Core 2 Quad Q9650 processor (3.0 GHz, 12M L2 cache, 1333MHz FSB) running 64-bit Linux in Figure 3. The results show that the constraint-solving time is negligible and the total execution time is low. As we show in graph in Figure 4, the running times scale linearly. Since the Jeeves runtime system performs aggressive simplifications on the constraint environment, Yices is good at solving unquantified constraints, and we derive constraints linear in the number of missing records, we expect constraint solving to be reasonable across Jeeves programs. Constraint derivation and symbolic evaluation scaled linearly on programs processing such large inputs, we do not expect performance to be an issue for reasonable uses of delegation.

8.2 Comparison with other languages

We found that SQL was less natural for implementing the imputation strategies because it requires the programmer to manually manage how and when each imputation occurs. SQL can perform fast queries (under 0.10 seconds) for each *individual* imputation strategy, but when many items can be missing it is up to the programmer to manage the order of the updates and other dependencies. In SQL, the programmer has to determine how to store the imputed values and is responsible for managing different versions of tables. The SQL programmer would need to manually consider all cases and manually execute the imputations by hand, in the right order. The programmer needs to propagate information for mutually dependent imputation strategies and chains of dependencies by hand and via intermediate tables.

Records	Rel.	Long.	"Hot deck"	Combined
8,192	0.02	0.02	0.02	0.11
16,384	0.03	0.03	0.03	0.15
32,768	0.06	0.06	0.07	0.25
65,546	0.13	0.14	0.13	0.43
131,072	0.33	0.31	0.30	0.9
262,144	0.72	0.72	0.68	2.22
392,550	1.29	1.28	1.17	3.68

Table 4. Python running times (in seconds) for benchmark examples.

We found that the Jeeves benchmarks had comparable performance to Python and that Jeeves made it easier to handle missing values. In Table 4 we show the Python running times: the prototype Jeeves interpreter is only a constant factor of three times slower than the Python 2.5.2 interpreter. We also found that it took significantly less time to write a bug-free version of the example in Jeeves than in Python. This code was more difficult to write in Python because the Python code needs to consider the different ways in which missing items could manifest. Also, the longer length made the code more difficult to debug: we had a performance bug that we did not catch until we graphed the results.

9. Related work

Programming with delegation builds on the constraint functional programming semantics that Mück *et al.* present [11]. This constraint functional model has also been implemented by languages such as Mercury [17] and Curry [4]. The implicit contexts in our language is based on the implicit parameter semantics described by Lewis *et al.* [9].

Current language support for implementing cross-cutting concerns does not ease the process of writing secure code. Information flow type systems such as Jif [12] and Fine [1] can check that the program does not leak values, but the programmer is responsible designing a system that performs the appropriate propagation and checks to adhere the specifications. System-level data flow frameworks such as Resin [20] provide abstractions for the programmer to insert checking code but do not help the programmer with producing and maintaining this code.

Aspect-oriented programming [5] mitigates propagation problem by providing support to hook in explicit annotations at join points, but changes in the core program require reasoning about the implications in the aspect code. Douglas Smith proposes viewing cross-cutting concerns as logical invariants [16] and describes a method for generating aspect code for behaviors such as error logging automatically [15]. The method Smith proposes involves reconstructing extra-computational values such a history and the runtime call stack. The execution model we propose allows for more direct recording and propagation of program data and metadata.

Our work is also related to previous work in executing declarative specifications alongside prescriptive code. The difference with our approach is that rather than executing constraints as isolated subprocedures, our execution model propagates constraints alongside the core program. Carroll Morgan's specification statements, which specify parts of the program that are "yet to be developed," are intended to execute as isolated procedures [10]. Squander, a mixed interpreter for declarative and imperative statements, executes declarative specification statements as subroutines [13]. Demsky's data structure repair uses goal-oriented programming to produce a concrete repaired structure when a data consistency property has been violated [2]. Plan B, a system for dynamic contract checking, uses a constraint solver to produce a value when it discovers a contract violation [14]. Kuncak *et al.*'s approach for complete functional synthesis inlines partially applied decision procedures to dynamically produce program expressions in a localized way [7].

10. Conclusions

In this paper, we make the following contributes:

- We present *programming with delegation*, a programming model that provides modularity for cross-cutting concerns.
- We demonstrate how to use programming with delegation to handle access control and information flow.
- We present a formalization of this programming model in terms of Jeeves, an ML-style functional language extended with constraints and implicit contexts.
- We describe an efficient execution model and interpreter implementation. We report performance on using programming with delegation for information flow and data processing examples.

References

[1] J. Chen, R. Chugh, and N. Swamy. Type-preserving compilation of end-to-end verification of security enforcement. *SIGPLAN Not.*, 45 (6):412–423, 2010. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1809028.1806643.

[2] B. Demsky and M. Rinard. Data structure repair using goal-directed reasoning. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 176–185, New York, NY, USA, 2005. ACM. ISBN 1-59593-963-2. doi: http://doi.acm.org/10.1145/1062455.1062499.

[3] V. Ganesh and D. L. Dill. A Decision Procedure for Bit-Vectors and Arrays. In *Computer Aided Verification (CAV '07)*, Berlin, Germany, July 2007. Springer-Verlag.

[4] M. Hanus, H. Kuchen, J. J. Moreno-Navarro, R. Aachen, and I. Ii. Curry: A truly functional logic language, 1995.

[5] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *ECOOP*, pages 220–242, 1997.

[6] E. Kohler. Hot Crap! In *WOWCS'08: Proceedings of the conference on Organizing Workshops, Conferences, and Symposia for Computer Systems*, pages 1–6, Berkeley, CA, USA, 2008. USENIX Association.

[7] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete functional synthesis. In *PLDI*, pages 316–329, 2010.

[8] S. K. Lahiri and S. A. Seshia. The UCLID Decision Procedure. In *CAV*, pages 475–478, 2004.

[9] J. R. Lewis, J. Launchbury, E. Meijer, and M. B. Shields. Implicit parameters: dynamic scoping with static types. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '00, pages 108–118, New York, NY, USA, 2000. ACM. ISBN 1-58113-125-9. doi: http://doi.acm.org/10.1145/325694.325708. URL `http://doi.acm.org/10.1145/325694.325708`.

[10] C. Morgan. The specification statement. *ACM Trans. Program. Lang. Syst.*, 10(3):403–419, 1988. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/44501.44503.

[11] A. Mück and T. Streicher. A tiny constraint functional logic language and its continuation semantics. In *ESOP '94: Proceedings of the 5th European Symposium on Programming*, pages 439–453, London, UK, 1994. Springer-Verlag. ISBN 3-540-57880-3.

[12] A. C. Myers. JFlow: Practical mostly-static information flow control. In *In Proc. 26th ACM Symp. on Principles of Programming Languages (POPL*, pages 228–241, 1999.

[13] D. Rayside, A. Milicevic, K. Yessenov, G. Dennis, and D. Jackson. Agile specifications. In *OOPSLA Companion*, pages 999–1006, 2009.

[14] H. Samimi, E. D. Aung, and T. D. Millstein. Falling back on executable specifications. In *ECOOP*, pages 552–576, 2010.

[15] D. R. Smith. A generative approach to aspect-oriented programming. In G. Karsai and E. Visser, editors, *GPCE*, volume 3286 of *Lecture Notes in Computer Science*, pages 39–54. Springer, 2004. ISBN 3-540-23580-9.

[16] D. R. Smith. Aspects as invariants. In O. Danvy, H. Mairson, F. Henglein, and A. Pettorossi, editors, *Automatic Program Development: A Tribute to Robert Paige*, pages 270–286, 2008.

[17] Z. Somogyi, F. J. Henderson, and T. C. Conway. Mercury, an efficient purely declarative logic programming language. In *In Proceedings of the Australian Computer Science Conference*, pages 499–512, 1995.

[18] The Bureau of Labor Statistics and the Census Bureau. Current Population Survey. `http://www.census.gov/cps/`, 2009.

[19] U.S. Census Bureau. *Current Population Survey, 2009 Annual Social and Economic (ASEc) Supplement*, 2009.

[20] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *Proceedings of the 22th ACM Symposium on Operating Systems Principles (SOSP '09)*, Big Sky, Montana, October 2009.