Generalizing Parametricity Using Information-flow

Geoffrey Washburn Stephanie Weirich
Department of Computer and Information Science
University of Pennsylvania
{geoffw, sweirich}@cis.upenn.edu

Abstract

Run-time type analysis allows programmers to easily and concisely define operations based upon type structure, such as serialization, iterators, and structural equality. However, when types can be inspected at run time, nothing is secret. A module writer cannot use type abstraction to hide implementation details from clients: clients can determine the structure of these supposedly "abstract" data types. Furthermore, access control mechanisms do not help isolate the implementation of abstract datatypes from their clients. Buggy or malicious authorized modules may leak type information to unauthorized clients, so module implementors cannot reliably tell which parts of a program rely on their type definitions.

Currently, module implementors rely on parametric polymorphism to provide integrity and confidentiality guarantees about their abstract datatypes. However, standard parametricity does not hold for languages with run-time type analysis; this paper shows how to generalize parametricity so that it does. The key is to augment the type system with annotations about information-flow. Implementors can then easily see which parts of a program depend on the chosen implementation by tracking the flow of dynamic type information.

1 Introduction

Type analysis is an important programming idiom. Traditional applications for type analysis include serialization, structural equality, cloning and iteration. Many systems use type analysis for more sophisticated purposes such as generating user interfaces, testing code, implementing debuggers and XML support. For this reason, it is important to support type analysis in modern programming languages.

A canonical example of run-time type analysis is the generic structural equality function.

```
fun eq['a] = typecase 'a of
    bool    => fn (x:bool, y:bool) =>
                 if x then y else false
  | 'b * 'c => fn (x:'b*'c, y:'b*'c) =>
                 eq ['b] (fst x, fst y) &&
                 eq ['c] (snd x, snd y)
  | ...
```

The `eq` function analyzes its type argument `'a` and returns an equality function for that type.

Authors of abstract datatypes can use such generic operations to quickly build implementations. For example, because equality for the following `Employee.t` datatype is structural, one may implement it via generic equality.

```
module Employee = struct
  (* name, SSN, address and salary *)
  type t = string * int * string * int
  (* An equality for this type. *)
  fun empEq (x : t) (y : t) =
    Generic.eq [t] (x,y)
end :> sig
  type t
  val empEq : t -> t -> bool
end
```

Although type analysis is very useful, it can also be dangerous. When types are analyzable, software developers cannot be sure that abstraction boundaries will be respected and that code will operate in a compositional fashion. Consequently, type analysis may destroy properties of *integrity* and *confidentiality* that the author of the `Employee` module expects. Using type analysis, anyone may create a value of type `Employee.t`. Although the type will be correct, other invariants not captured in the type system may be broken. For example, malicious code can create employees with negative salaries.

Proceedings of the 20th Annual Symposium on Logic in Computer Science (LICS'05)
0-7695-2266-1/05 $ 20.00 **IEEE**

Furthermore, even if the author of the `Employee` module tries to keep aspects of the employee representation hidden, another module can simply use generic operations to discover them. For example, if no accessor was provided to the salary component of an `Employee.t`, malicious code could still extract it.

One answer to these problems is to simply prohibit run-time type analysis. However, we believe the benefits of type analysis are too compelling to abandon altogether. Therefore, we propose a basis for a language that permits type analysis, yet allows module writers to define integrity and confidentiality policies for abstract datatypes. In particular, we want authors to know: How does changing the implementation of datatype affect the rest of a program? How does code she writes depend on the other abstract types they use?

In languages without type analysis, these questions are easy to answer. Authors rely on *parametric polymorphism* to provide guarantees. The author knows the rest of the program must treat her abstract datatypes as black boxes that may only be "pushed around", not inspected, modified or created. Dually, authors are restricted in the same fashion when using other abstract datatypes. In the presence of type analysis, the programmer cannot know what code may depend on the definition of an abstract datatype. Any part of the program can dynamically discover the underlying type and introduce dependencies on its definition.

In the past it has been suggested that type analysis could be tamed by distinguishing between analyzable and unanalyzable types [6]. Unfortunately, just controlling which parts of the program may analyze a type does not allow programmers to answer our questions. Imagine an extension, not unlike "friends" in C++, where an author can specify which modules may analyze a type. In the following code, modules `A` and `B` may analyze the type `A.t`, and modules `B` and `C` may analyze the type `B.u`.

```
module A = struct
 type t = int
 val x = 3
end :> sig
 type t permit A, B
 val x : t
end

module B = struct
 type u = A.t
 val y = A.x
end :> sig
 type u permit B, C
 val y : u
end

module C = struct
 val z = case (Generic.cast [B.u] [int])
          of SOME f => "It is an int"
           | NONE   => "It is not an int"
end :> sig
 val z : string
end
```

Module `C` is not parametric with respect to `A.t`, even though module `C` is not allowed to analyze `A.t`: If the implementation of `A.t` changes, so does the value of `C.z`. Despite restricting analysis of `A.t` to `A` and `B`, the implementation of the type has been leaked to a third-party. Furthermore, because the type `B.u` is abstract, the author of `A` cannot know of the dependency. Access control places undue trust in a client not to provide others with the capabilities and information it has been granted. Consequently, we must look beyond access-control for a method of answering the desired questions.

We propose that tracking the flow of type information through a program with *information-flow labels* allows a programmer to easily determine how their type definitions influence the rest of the program. Information-flow extends a standard type system with elements of a lattice that describes the information content for each computation. For example, we could use a simple lattice containing two points L (low-security) and H (high-security). A type $\texttt{bool}^{\mathsf{H}}$ then means the expression it describes could use "high-security" information to produce the resulting boolean, while an expression of type $\texttt{bool}^{\mathsf{L}}$ requires only "low-security" information to produce its result. The novelty of our approach compared to previous information-flow type systems is that we also label kinds to track the information content of types.

To reason about abstract types in the presence of type analysis, we label types with an information content that can be tracked. Computations depending on those types must also have that label.

```
module A = struct
 type t = int
 val x = 3
end :> sig
 type t^H
 val x : t^L
end

module B = struct
 type u = A.t
 val y = A.x
end :> sig
 type u^H
 val y : u^L
end

module C = struct
 val z = case (Generic.cast [B.u] [int])
          of SOME f => "It is an int"
           | NONE   => "It is not an int"
end :> sig
 val z : string^H
end
```

In the revised example, sealing module `A` with the signature $\texttt{sig type t}^{\mathsf{H}}\ \texttt{val x : t}^{\mathsf{L}}\ \texttt{end}$ indicates that the type definition `t` depends upon high-security information and the value `x` on only low-security information. The type `B.u` and value `C.z` must both be labeled as high security because they depend upon the high-security information in `A.t`. The presence of a label H alerts the author of `A` to a dependency.

Proceedings of the 20th Annual Symposium on Logic in Computer Science (LICS'05)
0-7695-2266-1/05 $ 20.00 **IEEE**

Furthermore, only module A can create values of type A.t that are labeled with L. Using type analysis to create values of type A.t would taint the result with H. Therefore, if module A requires its inputs be of type $\mathtt{A.t}^{\mathrm{L}}$, then it is impossible to use its functions with forged values. The author now has a guarantee that module invariants will be maintained and the integrity of her abstraction will not be violated.

Information flow avoids the problems of access control because information must be propagated even when no access occurs. For example, the identity function can be assigned both the type $\mathtt{A.t}^{\mathrm{L}} \to^{\mathrm{L}} \mathtt{A.t}^{\mathrm{L}}$ and the type $\mathtt{A.t}^{\mathrm{H}} \to^{\mathrm{L}} \mathtt{A.t}^{\mathrm{H}}$ witnessing that it propagates the information content of the argument unchanged. Here the function type $\to$ is itself labeled to indicate the information content of creating the function—creating the identity function does not require any information.

In the next section, we describe a core calculus for combining information-flow and run-time type analysis. We then follow with our key contribution: By tracking the flow of type information, it is possible to generalize the standard parametricity theorem for languages with run-time type analysis. This generalized theorem can be used in the same manner as parametricity to establish integrity and confidentiality properties.

2 The λ_{SECi} language

λ_{SECi} is a core calculus combining information flow and type analysis. We designed λ_{SECi} to be as simple as possible while still retaining the flavor of the problem. It is derived from the type-analyzing language $\lambda_{\mathrm{i}}^{\mathrm{ML}}$ developed by Harper and Morrisett [6] and the information-flow security language λ_{SEC} of Zdancewic [21].

2.1 Run-time type analysis

The grammar for λ_{SECi} appears in Figure 1. The complete semantics for λ_{SECi} can be found in the extended version of this paper [19]. It is a predicative, call-by-value polymorphic λ-calculus with booleans, functions and recursion. Fix-points are separate from functions to make nontermination aspects of proofs modular.

As in $\lambda_{\mathrm{i}}^{\mathrm{ML}}$, type constructors, τ, which can be analyzed at run-time, are separated from types, σ, which describe terms. We conjecture our results extend to languages with impredicative and higher-order polymorphism, but for simplicity, we do not examine the problem in this paper.

The language of type constructors consists of the simply-typed λ-calculus and two primitive constructors that correspond to types: bool and $\tau_1 \to \tau_2$.

kinds

$\kappa ::= \star^{\ell} \mid \kappa_1 \xrightarrow{\ell} \kappa_2$	*types & operators*

type constructors

$\tau ::= \alpha \mid \lambda\alpha{:}\kappa.\tau \mid \tau_1\tau_2$	λ-*calculus*
$\mid \mathsf{bool} \mid \tau_1 \to \tau_2$	*booleans & functions*
$\mid \mathsf{Typerec}\ \tau\ \tau_{\mathsf{bool}}\ \tau_{\to}$	*analysis*

types

$\sigma ::= (\tau)^{\ell}$	*injection*
$\mid \sigma_1 \xrightarrow{\ell} \sigma_2$	*functions*
$\mid \forall^{\ell_1}\alpha{:}\star^{\ell_2}.\sigma$	*polymorphism*

terms

$e ::= \mathbf{true} \mid \mathbf{false}$	*booleans*
$\mid x \mid \lambda x{:}\sigma.e \mid e_1 e_2$	λ-*calculus*
$\mid \Lambda\alpha{:}\star^{\ell}.e \mid e[\tau]$	*polymorphism*
$\mid \mathbf{fix}\ x{:}\sigma.e$	*fix-point*
$\mid \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3$	*conditional*
$\mid \mathbf{typecase}[\gamma.\sigma]\ \tau\ e_{\mathsf{bool}}\ e_{\to}$	*analysis*

values

$v ::= \mathbf{true} \mid \mathbf{false} \mid \lambda x{:}\sigma.e \mid \Lambda\alpha{:}\star^{\ell}.e$

term substitutions	γ	$::=$	$\cdot \mid \gamma, [e/x]$
type substitutions	δ	$::=$	$\cdot \mid \delta, [\tau/\alpha]$
term variable contexts	Γ	$::=$	$\cdot \mid \Gamma, x{:}\sigma$
type variable contexts	Δ	$::=$	$\cdot \mid \Delta, \alpha{:}\kappa$

Figure 1. The λ_{SECi} language

The term form **typecase** can be used to define operations that depend on run-time type information. This term takes a constructor to scrutinize, τ, as well as two branches: e_{bool} and $e_{\to}$). During evaluation the constructor argument is reduced to its head form so that the appropriate branch can be chosen.

$$\frac{\tau \rightsquigarrow^* \mathsf{bool}}{\mathbf{typecase}\ [\gamma.\sigma]\ \tau\ e_{\mathsf{bool}}\ e_{\to}\ \rightsquigarrow e_{\mathsf{int}}}$$

$$\frac{\tau \rightsquigarrow^* \tau_1 \to \tau_2}{\mathbf{typecase}\ [\gamma.\sigma]\ \tau\ e_{\mathsf{bool}}\ e_{\to}\ \rightsquigarrow e_{\to}[\tau_1][\tau_2]}$$

We write $e \rightsquigarrow e'$ to mean that term e reduces in a single step to e' and $\tau \rightsquigarrow \tau'$ to mean that constructor τ makes a weak-head reduction step to τ'.

λ_{SECi} also includes a constructor, Typerec, allowing types to depend upon type information. Without Typerec, it is impossible to assign a type to some useful terms that perform type analysis [6]. Typerec implements a *paramorphism* (a type of fold) over the structure of the

Proceedings of the 20th Annual Symposium on Logic in Computer Science (LICS'05)
0-7695-2266-1/05 $ 20.00 **IEEE**

$$\mathcal{L}(\star^\ell) \triangleq \ell \qquad \mathcal{L}(\kappa_1 \xrightarrow{\ell} \kappa_2) \triangleq \ell$$

$$\mathcal{L}((\tau)^\ell) \triangleq \ell \qquad \mathcal{L}(\sigma_1 \xrightarrow{\ell} \sigma_2) \triangleq \ell$$
$$\mathcal{L}(\sigma_1 \times^\ell \sigma_2) \triangleq \ell \qquad \mathcal{L}(\forall^{\ell_1} \alpha{:}\star^{\ell_2}.\sigma) \triangleq \ell_1$$

Figure 2. Kind and type label operators

argument constructor. As in the following reduction rules, when the head of the argument is one of the two primitive constructors, Typerec will apply the appropriate branch to the constituent types and the recursive invocation of Typerec on the constituents.

$$\frac{}{\mathsf{Typerec}\ (\mathsf{bool})\ \tau_{\mathsf{bool}}\ \tau_{\rightarrow}\ \rightsquigarrow\ \tau_{\mathsf{bool}}}$$

$$\frac{}{\begin{array}{l}\mathsf{Typerec}\ (\tau_1 \rightarrow \tau_2)\ \tau_{\mathsf{bool}}\ \tau_{\rightarrow}\ \rightsquigarrow \\ \quad \tau_{\rightarrow}\ \tau_1\ \tau_2\ (\mathsf{Typerec}\ \tau_1\ \tau_{\mathsf{bool}}\ \tau_{\rightarrow})\ (\mathsf{Typerec}\ \tau_2\ \tau_{\mathsf{bool}}\ \tau_{\rightarrow})\end{array}}$$

2.2 The information content of constructors

Information-flow type systems track the flow of information by annotating types with labels that specify the information content of the terms they describe. Because our type constructors influence the evaluation of terms in λ_{SECi}, we also label kinds.

Labels, ℓ, are drawn from an unspecified join semilattice, with a least element ($\bot$), joins ($\sqcup$) for finite subsets of elements in the lattice, and a partial order ($\sqsubseteq$). The actual lattice used by the type system is determined by the desired confidentiality and integrity policies of the program. Intuitively, the higher a label is in the lattice, the more restricted the information content of a constructor or term should be. For most examples in this paper, we use a simple two point lattice ($\bot$ for low security, $\top$ for high security) that tracks the dynamic discovery of a single type definition. In practice, any lattice with the specified structure could be used. An example of a practical lattice with richer internal structure is the Decentralized Label Model (DLM) of Myers and Liskov [10].

The labels on kinds describe the information content of type constructors. The kind of a constructor (and therefore its information content) is described using the judgment $\Delta \vdash \tau : \kappa$, read as "constructor τ is well-formed having kind κ with respect to the type variable context Δ." The operator $\mathcal{L}(\kappa)$, defined in Figure 2, extracts the label of a kind.

Our calculus is conservative: If the label of κ is ℓ, then the information content of a constructor of kind κ is *at most* ℓ. The information level of a constructor can be raised via subsumption. As kinds are labeled, the ordering $\sqsubseteq$ on labels induces a sub-kinding relation, $\kappa_1 \leq \kappa_2$. A kind $\star^{\ell_1}$ is a sub-kind of $\star^{\ell_2}$ if $\ell_1 \sqsubseteq \ell_2$. Sub-kinding for function kinds is standard. The relation is reflexive and transitive by definition.

The label of a constructor τ of kind $\star^\ell$, also describes the information gained when the constructor is analyzed. Type variables (such as `Employee.t`) may be given a high security level so that their information content may be traced throughout the program. For example, the kind of a Typerec constructor must be labeled at least as high as the analyzed constructor τ. This requirement accounts for information gained by inspecting τ.

$$\frac{\Delta \vdash \tau : \star^\ell \qquad \ell \sqsubseteq \ell' \qquad \Delta \vdash \tau_{\mathsf{bool}} : \kappa \qquad \Delta \vdash \tau_{\rightarrow} : \star^\ell \xrightarrow{\ell'} \star^\ell \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \xrightarrow{\ell'} \kappa \quad \text{where } \ell' = \mathcal{L}(\kappa)}{\Delta \vdash \mathsf{Typerec}\ \tau\ \tau_{\mathsf{bool}}\ \tau_{\rightarrow} : \kappa}$$

By default the label on the bool constructor is set to $\bot$. The label of the kind for function constructors must be at least as high as the join of its two constituent constructors. This is because the label must reflect the information content of the entire constructor.

The kinds of type functions, $\kappa_1 \xrightarrow{\ell} \kappa_2$, have a label ℓ that represents the information propagated by invoking the function. As shown below, the information, ℓ, is propagated into the result of application as $\kappa_2 \sqcup \ell$. This is shorthand for relabeling κ_2 with $\mathcal{L}(\kappa_2) \sqcup \ell$.

$$\frac{\Delta \vdash \tau_1 : \kappa_1 \xrightarrow{\ell} \kappa_2 \qquad \Delta \vdash \tau_2 : \kappa_1}{\Delta \vdash \tau_1 \tau_2 : \kappa_2 \sqcup \ell}$$

2.3 Tracking information flow in terms

The labels on types describe the information content of terms. We use the judgment $\Delta^\star \mid \Gamma \vdash e : \sigma$ to mean that "term e is well-formed with type σ with respect to the term context Γ and the type context $\Delta^\star$." We use $\Delta^\star$ to denote type variable contexts restricted to variables of kind $\star^\ell$ for any label ℓ. As we did for kinds, we define (in Figure 2) the operator $\mathcal{L}(\sigma)$ to extract the label of a type. Like constructors, the information content specified by labels for terms is conservative. The lattice ordering induces a subtyping judgment $\Delta^\star \vdash \sigma_1 \leq \sigma_2$, and subsumption can raise the information level of a term.

The types of λ_{SECi} include the standard ones for functions $\sigma_1 \xrightarrow{\ell} \sigma_2$ and quantified types $\forall^{\ell_1} \alpha{:}\star^{\ell_2}.\sigma$, plus those that are computed by type constructors $(\tau)^\ell$. Note that in the well-formedness rule for types formed from type constructors, shown below, there is no need for a connection between the label ℓ on the kind and the label on the type.

$$\frac{\Delta^\star \vdash \tau : \star^\ell}{\Delta^\star \vdash (\tau)^\bot}$$

Proceedings of the 20th Annual Symposium on Logic in Computer Science (LICS'05)
0-7695-2266-1/05 $ 20.00 **IEEE**

That is because ℓ describes the information content of τ, while the label ℓ' on $(\tau)^{\ell'}$ describes the information content of a term with type $(\tau)^{\ell'}$. It is sound to discard ℓ because once a constructor has been coerced to a type it may only be used statically to describe terms and cannot be analyzed.

Information flow is tracked at the term level analogously to the type level. Term abstractions, $\sigma_1 \xrightarrow{\ell} \sigma_2$, like type functions propagate some information ℓ when the are applied. Similarly, type abstractions, $\forall^{\ell_1}\alpha{:}\star^{\ell_2}.\sigma$, propagate some information ℓ_1 when they are applied. The label ℓ_2 is part of the kind of α.

Like Typerec, **typecase** examines the structure of the scrutinee and learns the information it carries, so the label ℓ' on the type of the term must be at least as high in the lattice as the label ℓ on the scrutinee.

$$\frac{\begin{array}{c}\Delta^\star \vdash \tau : \star^\ell \\ \Delta^\star, \gamma{:}\star^\ell \vdash \sigma \qquad \ell \sqsubseteq \ell' \qquad \Delta^\star \mid \Gamma \vdash e_{\mathsf{bool}} : \sigma[\mathsf{bool}/\gamma] \\ \Delta^\star \mid \Gamma \vdash e_\rightarrow : \forall^{\ell'}\alpha{:}\star^\ell.\forall^{\ell'}\beta{:}\star^\ell.\sigma[\alpha \to \beta/\gamma] \\ \text{where } \ell' = \mathcal{L}(\sigma[\tau/\gamma])\end{array}}{\Delta^\star \mid \Gamma \vdash \textbf{typecase } [\gamma.\sigma]\ \tau\ e_{\mathsf{bool}}\ e_\rightarrow\ : \sigma[\tau/\gamma]}$$

Because the type of a **typecase** term can depend upon the scrutinized constructor τ, an annotation, $[\gamma.\sigma]$, is required for type checking.

2.4 Soundness

λ_{SECi} has the basic property expected from a typed language, that well-typed programs will not go wrong.

Theorem 2.1 (Type Safety). *If $\vdash e : \sigma$ then e either evaluates to a value or diverges.*

The theorem is proven syntactically using the standard progress and preservation lemmas [20]. Details can be found in the extended version of this paper [19].

3 Generalizing parametricity

Reynold's parametricity theorem has long been used to reason about programs in languages with parametric polymorphism [13]. For example, the theorem can be used to show that different implementations of an abstract datatype do not influence the behavior of the program or to show that external modules cannot forge values of abstract types. These are only a few of the corollaries of the parametricity theorem. This sections starts with an overview of the standard parametricity theorem, and then examine how it can be generalized for λ_{SECi}. Proofs for the lemmas and theorems that follow be can found in the extended version of this paper [19].

$$\frac{\alpha \mapsto R \in \eta \qquad \nu_1 R \nu_2}{\eta \vdash \nu_1 \sim \nu_2 : \alpha}\ \mathsf{lr{:}var}$$

$$\frac{}{\eta \vdash \nu \sim \nu : \mathsf{bool}}\ \mathsf{lr{:}bool}$$

$$\frac{\forall(\eta \vdash e_1 \approx e_2 : \sigma_1).\eta \vdash \nu_1 e_1 \approx \nu_2 e_2 : \sigma_2}{\eta \vdash \nu_1 \sim \nu_2 : \sigma_1 \to \sigma_2}\ \mathsf{lr{:}arr}$$

$$\frac{\begin{array}{c}\forall\tau_1,\tau_2.\forall R \in \tau_1 \leftrightarrow \tau_2. \\ \eta, \alpha \mapsto R \vdash \nu_1[\tau_1] \approx \nu_2[\tau_2] : \sigma\end{array}}{\eta \vdash \nu_1 \sim \nu_2 : \forall\alpha{:}\star.\sigma}\ \mathsf{lr{:}all}$$

$$\frac{e_1 \rightsquigarrow^* \nu_1 \qquad e_2 \rightsquigarrow^* \nu_2 \qquad \eta \vdash \nu_1 \sim \nu_2 : \sigma}{\eta \vdash e_1 \approx e_2 : \sigma}\ \mathsf{lr{:}term}$$

$$\frac{e_1 \uparrow \qquad e_2 \uparrow}{\eta \vdash e_1 \approx e_2 : \sigma}\ \mathsf{lr{:}divr}$$

Figure 3. Logically related terms

3.1 Parametricity

For pedagogical purposes, this section and and the following section considers only the core of λ_{SECi} without type constructors, security labels, or type analysis. That is, a simple predicative polymorphic λ-calculus. None of the results presented in these sections are new. Informally, the parametricity theorem states that well-typed expressions, after applying related substitutions for their free type and term variables, are related to themselves. The power of the theorem comes from the fact that terms typed by universally quantified type variables can be related by any relation. Section 3.2 considers some important corollaries of this theorem for reasoning about data abstraction in programs.

The logical relation used by the parametricity theorem is defined in Figure 3. Terms are related with the judgment $\eta \vdash e_1 \approx e_2 : \sigma$, read as "terms e_1 and e_2 are related at type σ with respect to the relations in η." The relation between values is similarly defined as $\eta \vdash \nu_1 \sim \nu_2 : \sigma$. Because these relations are defined inductively over types which potentially contain free type variables, the relations are parameterized by a map, η, between type variables and binary relations on values. This map is used when σ is a type variable (see rule lr:var).

If σ is bool, the relation is identity. Typical for logical relations, values of function type are related only if when applied to related arguments, they produce related results. Terms are related if they evaluate to related values or both diverge. We write $e \uparrow$ to denote divergence.

Proceedings of the 20th Annual Symposium on Logic in Computer Science (LICS'05)
0-7695-2266-1/05 $ 20.00 **IEEE**

$$\frac{\forall \alpha{:}\star \in \Delta^\star.(\eta(\alpha) \in \delta_1(\alpha) \leftrightarrow \delta_2(\alpha))}{\eta \vdash \delta_1 \approx \delta_2 : \Delta^\star}\ \textsf{tslr:base}$$

$$\frac{\forall x{:}\sigma \in \Gamma.(\eta \vdash \gamma_1(x) \approx \gamma_2(x) : \sigma)}{\eta \vdash \gamma_1 \approx \gamma_2 : \Gamma}\ \textsf{slr:base}$$

Figure 4. Substitutions for parametricity

The most important rule defines the relationship between values of type $\forall\alpha{:}\star.\sigma$. Polymorphic values are related if their instantiations with *any* pair of types are related. Furthermore, we can use *any* relation R between values of those types as the relation on α. We use the notation $R \in \tau_1 \leftrightarrow \tau_2$ to mean that R is a binary relation on values of type τ_1 and of type τ_2. If quantification over types of higher kind were allowed, R must be a function on relations. This extension is orthogonal to our result, so we restrict ourselves to polymorphism over kind $\star$.

To state the parametricity theorem, we must define the notion of related substitutions for types and related terms. In Figure 4, the rule tslr:base states that a relation mapping η is well-formed with respect to two type substitutions δ_1 and δ_2 for the variables in the type context $\Delta^\star$. There are no restrictions on the range of the type substitutions. On the other hand, slr:base requires that a pair of term substitutions for the variables in Γ must map to related terms. Even though λ_{SECi} has call-by-value semantics, term substitutions must map to terms, not values. Otherwise, it would it be impossible to prove the case for fix-points which requires a term substitution.

With these definitions it is possible to state the parametricity theorem for our restricted language:

Theorem 3.1 (Parametricity). *If* $\Delta^\star \mid \Gamma \vdash e : \sigma$ *and* $\eta \vdash \delta_1 \approx \delta_2 : \Delta^\star$ *and* $\eta \vdash \gamma_1 \approx \gamma_2 : \Gamma$ *then* $\eta \vdash \delta_1(\gamma_1(e)) \approx \delta_2(\gamma_2(e)) : \sigma$.

The proof is by induction on the typing judgment.

One significant complication in the proof is circularity in relating fix-points. To escape this problem we apply a syntactic technique from Pitts [11]. We define a restricted fix-point that can only be unfolded a finite number of times before diverging. The term $\mathbf{fix}_{n+1}\ x{:}\sigma.e$ unwinds to $e[(\mathbf{fix}_n\ x{:}\sigma.e)/x]$. By definition $\mathbf{fix}_0\ x{:}\sigma.e$ always diverges. It is then straightforward to show that for any n, $\mathbf{fix}_n\ x{:}\sigma.e$ is related to itself. Then the following continuity lemma can be used to prove that unbounded fix-points are related to themselves.

Lemma 3.2 (Continuity). *If* $\eta \vdash \delta_1 \approx \delta_2 : \Delta^\star$ *and for all* n, $\eta \vdash \mathbf{fix}_n\ x{:}\sigma_1.e_1 \approx \mathbf{fix}_n\ x{:}\sigma_2.e_2 : \sigma$ *where* $\delta_1(\sigma) = \sigma_1, \delta_2(\sigma) = \sigma_2$ *then* $\eta \vdash \mathbf{fix}\ x{:}\sigma_1.e_1 \approx \mathbf{fix}\ x{:}\sigma_2.e_2 : \sigma$.

3.2 Applications of the parametricity theorem

The parametricity theorem has been used for many purposes, most famously for deriving *free theorems* about functions in the polymorphic λ-calculus, just by looking at their types [18]. Our purpose is more similar to that of Reynolds: reasoning about the properties of programs in the presence of type abstraction. While Reynolds separated parametric polymorphism from ad-hoc polymorphism, we show how to generalize his work to both sorts of polymorphism.

Corollaries of Theorem 3.1 provide important results for reasoning about abstract types in programs. Many specific properties can be proven as a consequence of parametricity, but we believe the following two are representative of what a programmer desires.

Corollary 3.3 (Confidentiality). *If* $\alpha{:}\star \mid x{:}\alpha \vdash e : \mathsf{bool}$ *and* $\vdash \nu_1 : \tau_1$ *and* $\vdash \nu_2 : \tau_2$ *then* $e[\tau_1/\alpha][\nu_1/x] \rightsquigarrow^* \nu$ *iff* $e[\tau_2/\alpha][\nu_2/x] \rightsquigarrow^* \nu$.

This first corollary says that a programmer is free to change the implementation of an abstract type without affecting the behavior of a program. It is the essence behind parametric polymorphism—type information is not allowed to influence program execution and values of abstract type must be treated parametrically.

Corollary 3.4 (Integrity). *If* $\alpha{:}\star \mid \cdot \vdash e : \alpha$ *then* e *must diverge.*

This second corollary states that there is no way for a program to invent values of an abstract type, violating the integrity of the abstraction.

3.3 Parametricity and type analysis

Consider the following λ_{SECi} term (eliding labels):

$$\mathbf{typecase}\ [\gamma.\mathsf{bool}]\ \alpha\ \mathbf{true}\ (\Lambda\alpha{:}\star.\Lambda\beta{:}\star.\mathbf{false})$$

This term violates Corollary 3.3, because we can substitute bool and $\mathsf{bool} \to \mathsf{bool}$ for α and it will produce different values: **true** versus **false**.

Still, we would like to state properties similar to Corollaries 3.3 and 3.4 for λ_{SECi}. It is not possible to directly extend the inductive proof for **typecase**. The proof would require that the two terms would produce related results, even when they may analyze different constructors. Furthermore, λ_{SECi} presents another complication: The weak-head normal forms of types include (for example) Typerec with its scrutinee a variable. Therefore, the logical relation must be extended to include these sorts of types.

Proceedings of the 20th Annual Symposium on Logic in Computer Science (LICS'05)
0-7695-2266-1/05 $ 20.00 **IEEE**

$$\frac{\ell_1 \not\sqsubseteq \ell_0}{\nu_1 \sim_{\ell_0} \nu_2 : \star^{\ell_1}}\ \text{tslr:type-opaq}$$

$$\frac{\ell_1 \sqsubseteq \ell_0}{\mathsf{bool} \sim_{\ell_0} \mathsf{bool} : \star^{\ell_1}}\ \text{tslr:type-bool}$$

$$\frac{\ell_1 \sqcup \ell_2 \sqsubseteq \ell_3 \qquad \ell_3 \sqsubseteq \ell_0 \qquad \tau_1 \approx_{\ell_0} \tau_3 : \star^{\ell_1} \qquad \tau_2 \approx_{\ell_0} \tau_4 : \star^{\ell_2}}{\tau_1 \to \tau_2 \sim_{\ell_0} \tau_3 \to \tau_4 : \star^{\ell_3}}\ \text{tslr:type-arr}$$

$$\frac{\forall(\tau_1 \approx_{\ell_0} \tau_2 : \kappa_1).\nu_1\tau_1 \approx_{\ell_0} \nu_2\tau_2 : \kappa_2 \sqcup \ell_1}{\nu_1 \sim_{\ell_0} \nu_2 : \kappa_1 \xrightarrow{\ell_1} \kappa_2}\ \text{tslr:arr}$$

$$\frac{\tau_1 \rightsquigarrow^* \nu_1 \qquad \tau_2 \rightsquigarrow^* \nu_2 \qquad \nu_1 \sim_{\ell_0} \nu_2 : \kappa}{\tau_1 \approx_{\ell_0} \tau_2 : \kappa}\ \text{tsclr:base}$$

Figure 5. Logically related constructors

To solve the problem with **typecase**, we require that the constructors used to instantiate polymorphic types be related to each other, as defined in the next subsection. Labeling kinds is the key to making this change practical, because it means the relation need not be the identity relation when types are used parametrically. The need for extra rules to handle additional weak-head normal form types is solved by generalizing the trick of quantifying over all relations between values of given types, to quantifying over families of relations on values of the correct types.

3.4 Related constructors

The first step towards a generalized parametricity theorem is formalizing what it means for type constructors to be related. We write $\tau_1 \approx_\ell \tau_2 : \kappa$ to mean closed constructors τ_1 and τ_2 are related at kind κ with respect to an observer at level ℓ in the label lattice. Similarly, the judgment $\nu_1 \sim_\ell \nu_2 : \kappa$ indicates that closed weak-head normal constructors ν_1 and ν_2 are related at kind κ with respect to an observer at level ℓ. The grammar of weak-head normal constructors and relations on constructors is defined in Figures 6 and 5, respectively.

The rule for type functions, tslr:arr, is standard for logical relations. There are three rules for kind $\star$. The first rule, tslr:type-opaq, codifies that if the label of the constructors is higher than the observer, then the constructors are indistinguishable. The remaining two rules state that if the label of a primitive constructor is less than the observer, their components must appear related to the observer. Constructors not in normal form are related by tsclr:base if and only if their weak-head normal forms are related.

constructor contexts
$\rho ::= \bullet \mid \mathsf{Typerec}\ \rho\ \tau_{\mathsf{bool}}\ \tau_{\to}\ \mid \rho\,\tau$

weak-head normal-form constructors
$\nu ::= \mathsf{bool} \mid \tau_1 \to \tau_2 \mid \lambda\alpha{:}\kappa.\tau$

weak-head normal-form types
$\zeta ::= (\mathsf{bool})^\ell \mid (\rho\{\alpha\})^\ell \mid \sigma_1 \xrightarrow{\ell} \sigma_2 \mid \forall^{\ell_1}\alpha{:}\star^{\ell_2}.\sigma$

Figure 6. Additional syntactic forms

$$\frac{\tau \rightsquigarrow \tau'}{(\tau)^\ell \rightsquigarrow (\tau')^\ell} \qquad \frac{}{(\tau_1 \to \tau_2)^\ell \rightsquigarrow (\tau_1)^\ell \xrightarrow{\ell} (\tau_2)^\ell}$$

Figure 7. Type reduction

As suggested by tslr:type-opaq, if two constructors carry information more restrictive than the level of the observer, the observer shouldn't be able to tell them apart. For example, $\mathsf{bool} : \star^\top$ and $\mathsf{bool} \to \mathsf{bool} : \star^\top$, which carry "high-security" information $\top$, will be indistinguishable to an observer at a "low-security" level $\bot$. This is formalized in the following lemma.

Lemma 3.5 (Obliviousness for constructors). *If* $\vdash \tau_1, \tau_2 : \kappa$ *and* $\mathcal{L}(\kappa) \not\sqsubseteq \ell_0$ *then* $\tau_1 \approx_{\ell_0} \tau_2 : \kappa$.

Finally, we can state a substitution theorem for constructors that is a simpler version of parametricity:

Lemma 3.6 (Substitution for constructors). *If* $\Delta \vdash \tau : \kappa$ *and* $\delta_1 \approx_{\ell_0} \delta_2 : \Delta$ *then* $\delta_1(\tau) \approx_{\ell_0} \delta_2(\tau) : \kappa$.

In this lemma, related type substitutions map type variables to related constructors, as defined in the following rule

$$\frac{\forall\alpha{:}\kappa \in \Delta.(\delta_1(\alpha) \approx_{\ell_0} \delta_2(\alpha) : \kappa)}{\delta_1 \approx_{\ell_0} \delta_2 : \Delta}$$

3.5 Related terms

As with constructors, we parameterize the logical relation on terms by an observer ℓ. We write $\eta \vdash e_1 \approx_\ell e_2 : \sigma$ to indicate that terms e_1 and e_2 are related to an observer at level ℓ at type σ, with the relation mapping η. As with constructors we distinguish between related terms and related normal forms, writing the judgment $\eta \vdash \nu_1 \sim_\ell \nu_2 : \zeta$ to indicate that values ν_1 and ν_2 are related to an observer at level ℓ at the weak-head normal type ζ, with the relation mapping η. These relations, as defined in Figure 8, are similar to the ones in

Proceedings of the 20th Annual Symposium on Logic in Computer Science (LICS'05)
0-7695-2266-1/05 $ 20.00 **IEEE**

$$\frac{\alpha \mapsto R \in \eta \qquad \ell_1 \sqsubseteq \ell_0 \implies v_1 R_\rho^{\ell_1} v_2}{\eta \vdash v_1 \sim_{\ell_0} v_2 : (\rho\{\alpha\})^{\ell_1}} \text{ slr:var}$$

$$\frac{\ell_1 \sqsubseteq \ell_0 \implies v_1 = v_2}{\eta \vdash v_1 \sim_{\ell_0} v_2 : (\mathsf{bool})^{\ell_1}} \text{ slr:bool}$$

$$\frac{\begin{array}{l}\forall(\eta \vdash e_1 \approx_{\ell_0} e_2 : \sigma_1).\\ \quad \eta \vdash v_1 e_1 \approx_{\ell_0} v_2 e_2 : \sigma_2 \sqcup \ell_1\end{array}}{\eta \vdash v_1 \sim_{\ell_0} v_2 : \sigma_1 \xrightarrow{\ell_1} \sigma_2} \text{ slr:arr}$$

$$\frac{\begin{array}{l}\forall(\tau_1 \approx_{\ell_0} \tau_2 : \star^{\ell_2}).\\ \quad \forall(R_\rho^{\ell_2} \in \delta_1((\rho\{\tau_1\})^{\ell_2}) \leftrightarrow \delta_2((\rho\{\tau_2\})^{\ell_2})).\\ \quad\quad \eta, \alpha \mapsto R \vdash v_1[\tau_1] \approx_{\ell_0} v_2[\tau_2] : \sigma \sqcup \ell_1\\ \quad\quad\quad R \text{ consistent}\end{array}}{\eta \vdash v_1 \sim_{\ell_0} v_2 : \forall^{\ell_1}\alpha{:}\star^{\ell_2}.\sigma} \text{ slr:all}$$

$$\frac{\begin{array}{c} e_1 \rightsquigarrow^* v_1\\ e_2 \rightsquigarrow^* v_2 \qquad \sigma \rightsquigarrow^* \zeta \qquad \eta \vdash v_1 \sim_{\ell_0} v_2 : \zeta\end{array}}{\eta \vdash e_1 \approx_{\ell_0} e_2 : \sigma} \text{ sclr:term}$$

$$\frac{(e_1 \uparrow) \vee (e_2 \uparrow)}{\eta \vdash e_1 \approx_{\ell_0} e_2 : \sigma} \text{ sclr:divr}$$

Figure 8. Logically related terms

Figure 3. One difference is that we relate only values at weak-head normal types ζ, defined in Figure 6. Restricting the value relation to weak-head normal types makes the logical relation much easier to state and understand.

Like constructors, the relation over terms is defined so that terms typed at a level greater than the observer will be indistinguishable. This is enforced by the precondition $\ell_1 \sqsubseteq \ell_0$ found in slr:var and slr:bool. The antecedent relations in slr:arr and slr:all have their types joined with ℓ_1; this accounts for information gained by destructing the value. The following lemma verifies our intuition about indistinguishability:

Lemma 3.7 (Obliviousness for terms). *If* $\Delta^\star \mid \cdot \vdash e_1, e_2 : \sigma$ *and* $\delta_1 \approx_{\ell_0} \delta_2 : \Delta^\star$ *and* $\mathcal{L}(\sigma) \not\sqsubseteq \ell_0$ *then* $\eta \vdash \delta_1(e_1) \approx_{\ell_0} \delta_2(e_2) : \sigma$.

There are two other significant differences between Figures 3 and 8: additional preconditions in slr:all and generalizing lr:var to slr:var. The rule slr:var solves the problem with Typerec appearing in the weak-head normal form of types. It generalizes lr:var to terms related at a constructor that cannot be normalized further because of an undetermined type variable. We characterize these constructors with constructor contexts, ρ, defined in Figure 6. Contexts are holes $\bullet$, Typerecs of a context, or a context applied to an arbitrary constructor. We write $\rho\{\tau\}$ for filling a context's hole with τ.

Previously, values were related at a type variable only if they were in the relation mapped to that variable by η. Here η maps to families of relations. We write R_ρ^ℓ for the application of R to a label ℓ and a context ρ, yielding a relation. Therefore, when we write $R_\rho^\ell \in \delta_1((\rho\{\tau_1\})^\ell) \leftrightarrow \delta_2((\rho\{\tau_2\})^\ell)$ we mean that R is a dependent function of ℓ and ρ yielding a relation on values of type $\delta_1((\rho\{\tau_1\})^\ell)$ and $\delta_2((\rho\{\tau_2\})^\ell)$.

Quantification over R is required to be consistent. In this context, that means if $v_1 R_\rho^{\ell_1} v_2$ and $\ell_1 \sqsubseteq \ell_2$ then $v_1 R_\rho^{\ell_2} v_2$. This is adequate for call-by-value because quantification is over families of value relations. Therefore requiring that R yield relations that are strict or preserve least-upper bounds is unnecessary, as values are always terminating. It is important that the logical relation itself is consistent, that is, closed under subsumption.

Lemma 3.8 (Term relation consistent). *If* $\delta_1 \approx_{\ell_0} \delta_2 : \Delta^\star$ *and* $\eta \vdash \Delta^\star$ *and* $\Delta^\star \vdash \sigma_1 \le \sigma_2$ *and* $\eta \vdash e_1 \approx_{\ell_0} e_2 : \sigma_1$ *then* $\eta \vdash e_1 \approx_{\ell_0} e_2 : \sigma_2$.

We write $\eta \vdash \Delta^\star$ to mean that the mapping η is well-formed with respect to a pair of type substitutions, δ_1 and δ_2, as defined in the rule:

$$\frac{\begin{array}{c}\eta(\alpha) \text{ consistent}\\ \forall \alpha{:}\star^{\ell_1} \in \Delta^\star.(\eta(\alpha)_\rho^{\ell_1} \in \delta_1((\rho\{\alpha\})^{\ell_1}) \leftrightarrow \delta_2((\rho\{\alpha\})^{\ell_1}))\end{array}}{\eta \vdash \Delta^\star}$$

The last significant difference in Figure 3 is in slr:divr. Terms are related if either diverges, as opposed to our earlier definition where divergent terms were related only to other divergent terms. This is a significant theoretical weakening. In particular, the logical relation is no longer transitive. However, this definition is standard for information-flow logical relations proofs with recursion [1, 21]. Furthermore, we believe that this weakening is acceptable in practice. We discuss in more detail in Section 3.6 how this requirement is merely an artifact of call-by-value information-flow.

3.6 Generalized parametricity

Before stating the generalized parametricity theorem, we must define a notation of related term substitutions. Given related type substitutions, $\delta_1 \approx_{\ell_0} \delta_2 : \Delta^\star$, and a well-formed mapping, $\eta \vdash \Delta^\star$, term substitutions are related if they map variables to related terms.

$$\frac{\forall x{:}\sigma \in \Gamma.(\eta \vdash \gamma_1(x) \approx_{\ell_0} \gamma_2(x) : \sigma)}{\eta \vdash \gamma_1 \approx_{\ell_0} \gamma_2 : \Gamma}$$

The only change from slr:base is the additional of a label ℓ_0 for the observer.

Proceedings of the 20th Annual Symposium on Logic in Computer Science (LICS'05)
0-7695-2266-1/05 $ 20.00 **IEEE**

Theorem 3.9 (Generalized parametricity). *If* $\Delta^\star \mid \Gamma \vdash e : \sigma$ *and* $\delta_1 \approx_{\ell_0} \delta_2 : \Delta^\star$ *and* $\eta \vdash \Delta^\star$ *and* $\eta \vdash \gamma_1 \approx_{\ell_0} \gamma_2 : \Gamma$ *then* $\eta \vdash \delta_1(\gamma_1(e)) \approx_{\ell_0} \delta_2(\gamma_2(e)) : \sigma$.

As with standard parametricity, the proof is by induction over $\Delta^\star \mid \Gamma \vdash e : \sigma$. In addition to the lemmas mentioned in Sections 3.4 and 3.5, Lemma 3.2 must be extended in the straightforward manner.

We call Theorem 3.9 generalized parametricity because Theorem 3.1 can be (almost) recovered by a series of restrictions:

- Use a two element lattice, $\bot$ and $\top$, where $\bot \sqsubseteq \top$.
- For every κ in $\Delta^\star$, Γ, e, and σ require $\mathcal{L}(\kappa) = \top$.
- For every σ' in Γ, e, and σ require $\mathcal{L}(\sigma') = \bot$.
- Require that the observer be $\bot$.

Even with these restrictions, because of the difference in sclr:divr, Theorem 3.9 makes a weaker claim about the termination behavior of related terms than Theorem 3.1. Consider the generalized version of Corollary 3.3.

Corollary 3.10 (Confidentiality). *If* $\alpha{:}\star^\top \mid x{:}\alpha \vdash e : (\mathsf{bool})^\bot$ *then for any* $\vdash \nu_1 : \tau_1$ *and* $\vdash \nu_2 : \tau_2$ *if* $e[\tau_1/\alpha][\nu_2/x]$ *and* $e[\tau_2/\alpha][\nu_2/x]$ *both terminate, they will produce the same value.*

This corollary states that what we substitute for α and x will not affect the value computed by e. However, it is possible that our choice of α and x could cause e to diverge.

Unlike standard parametricity, Theorem 3.9 has an explicit observer. Standard parametricity has an implicit observer that can observe all computation. What makes information-flow techniques work is that some computations are opaque to the observer. Furthermore, the results of these computations are also inaccessible to the observer, making them effectively dead code. However, because the operational semantics is call-by-value, dead code must be executed even though the result is never used. Therefore, we conjecture that using a call-by-need operational semantics an exact correspondence could be recovered; the only part of the proof that would need to change is obliviousness for terms, Lemma 3.7.

3.7 Applications of generalized parametricity

A typical corollary of Theorem 3.9 is normally called noninterference; that it is possible to substitute values indistinguishable to the observer and get indistinguishable results.

Corollary 3.11 (Noninterference). *If* $\cdot, x{:}\sigma_1 \vdash e : \sigma_2$ *where* $\mathcal{L}(\sigma_1) \not\sqsubseteq \mathcal{L}(\sigma_2)$ *then for any* $\vdash \nu_1 : \sigma_1$ *and* $\vdash \nu_2 : \sigma_1$ *it is the case that if both* $e[\nu_1/x]$ *and* $e[\nu_2/x]$ *terminate, they will both produce the same value*

More importantly, it is also possible to restate the corollaries of standard parametricity. The previous subsection stated the revised corollary for confidentiality. The same can be done for integrity:

Corollary 3.12 (Integrity). *If* $\alpha{:}\star^\top \mid \cdot \vdash e : (\alpha)^\bot$ *then e must diverge.*

Furthermore, it is also possible to make much richer and refined claims because the label lattice expands upon the implicit two level lattice used by parametricity.

4 Related work

λ_{SECi} draws heavily upon previous work on type analysis, parametricity, and information flow.

Most information flow systems use a lattice model originating from work by Bell and LaPadula [3] and Denning [4]. Volpano et al. [16] showed that Denning's work could be formulated as type system and proved its soundness with respect to noninterference. Heintze and Riecke's formalized information-flow and integrity in a typed λ-calculus with references, the SLam calculus [7], and proved a number of soundness and noninterference results. Pottier and Simonet have developed an extension to ML, called FlowCaml, and have shown noninterference using an alternative syntactic technique [12].

Prior to our research, FlowCaml was the only language with polymorphism and a noninterference proof. FlowCaml does not consider run-time type analysis and can rely on standard parametricity for types. The noninterference result for λ_{SECi} directly builds upon the methods of Zdancewic [21] and Pitts [11].

Other researchers have noticed the connection between parametricity and noninterference. The work of Tse and Zdancewic [15] compliments our research by showing how parametricity can be used to prove noninterference. Tse and Zdancewic do so by encoding Abadi, et al.'s [1] dependency core calculus into System $\mathbb{F}$.

The fact that run-time type analysis (and other forms of ad-hoc polymorphism) breaks parametricity has been long understood, but little has been done to reconcile the two. Leifer et al. [8] design a system that preserves type abstraction in the presence of (un)marshalling. This is a weaker result because marshalling is merely a single instance of an operation using run-time type analysis. Rossberg [14] and Vytiniotis et al. [17] use generative

Proceedings of the 20th Annual Symposium on Logic in Computer Science (LICS'05)
0-7695-2266-1/05 $ 20.00 **IEEE**

types to hide type information in the presence of run-time analysis, relying on colored-brackets [5] to provide easy access. However, none of this work has formalized the abstraction properties that their systems provide.

5 Conclusion

With λ_{SECi}, we address the conflict between run-time type analysis and enforceable data abstractions. By labeling their type abstractions, software developers can easily observe dependencies.

However, this refinement comes at with the penalty of having to write many annotations for a program to type check. We have not investigated how pervasive the necessary annotations will prove in practice. Existing large scale languages, such as Jif [9] and FlowCaml [12], implement some form of information-flow inference, but they can be difficult to use. Languages based on λ_{SECi} have the advantage that if the only goal is to secure type abstractions and no type analysis is performed, then no information-flow annotations are necessary. Regardless, it will be imperative to study the cost of maintaining the necessary annotations in practical languages based upon λ_{SECi}.

Acknowledgements

This paper benefitted greatly from conversations with Steve Zdancewic and Stephen Tse. We also appreciate the insightful comments by anonymous reviewers on earlier revisions of this work. This work was supported by NSF grant 0347289, CAREER: Type-Directed Programming in Object-Oriented Languages.

References

[1] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *Proc. 26th ACM Symp. on Principles of Programming Languages*, pages 147–160, San Antonio, TX, Jan. 1999.

[2] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. *Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

[3] D. E. Bell and L. J. LaPadula. Secure computer system: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, MITRE Corp. MTR-2997, Bedford, MA, 1975.

[4] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[5] D. Grossman, G. Morrisett, and S. Zdancewic. Syntactic type abstraction. *Transactions on Programming Languages and Systems*, 22(6):1037–1080, Nov. 2000.

[6] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *22nd ACM Symp. on Principles of Programming Languages*, pages 130–141, San Francisco, Jan. 1995.

[7] N. Heintze and J. G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proc. 25th ACM Symp. on Principles of Programming Languages*, San Diego, CA, 1998.

[8] J. J. Leifer, G. Peskine, P. Sewell, and K. Wansbrough. Global abstraction-safe marshalling with hash types. In *Proc. 8th ICFP*, pages 87–98, Uppsala, Sweden, 2003.

[9] A. C. Myers, S. Chong, N. Nystrom, L. Zheng, and S. Zdancewic. Jif: Java information flow. Software release. Located at http://www.cs.cornell.edu/jif.

[10] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.

[11] A. M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10:1–39, 2000.

[12] F. Pottier and V. Simonet. Information flow inference for ML. In *Proc. 29th ACM Symp. on Principles of Programming Languages*, Portland, OR, Jan. 2002.

[13] J. C. Reynolds. Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523, Amsterdam, 1983. Elsevier Science Publishers B. V.

[14] A. Rossberg. Generativity and dynamic opacity for abstract types. In *Proc. of the 5th International ACM Conference on Principles and Practice of Declarative Programming*, Uppsala, Sweden, Aug. 2003.

[15] S. Tse and S. Zdancewic. Translating dependency into parametricity. In *Proc. of the 9th ACM International Conference on Functional Programming*, Snowbird, Utah, September 2004.

[16] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[17] D. Vytiniotis, G. Washburn, and S. Weirich. An open and shut typecase. In *Proc. of the 2nd ACM Workshop on Types in Language Design and Implementation*, Longbeach, California, January 2005.

[18] P. Wadler. Theorems for free! In *FPCA89: Conference on Functional Programming Languages and Computer Architecture*, London, Sept. 1989.

[19] G. Washburn and S. Weirich. Generalizing parametricity using information flow (extended version). Technical Report MS-CIS-05-04, University of Pennsylvania, Philadelphia, PA, to appear 2005.

[20] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994.

[21] S. Zdancewic. *Programming Languages for Information Security*. PhD thesis, Cornell University, 2002.

Proceedings of the 20th Annual Symposium on Logic in Computer Science (LICS'05)
0-7695-2266-1/05 $ 20.00 **IEEE**