Tracking Data-Flow with Open Closure Types

Gabriel Scherer, Jan Hoffmann

Tracking Data-Flow with Open Closure Types

Gabriel Scherer[1] and Jan Hoffmann[2]

[1] INRIA Paris-Rocquencourt
[2] Yale University

Abstract. Type systems hide data that is captured by function closures in function types. In most cases this is a beneficial design that enables simplicity and compositionality. However, some applications require explicit information about the data that is captured in closures.
This paper introduces open closure types, that is, function types that are decorated with type contexts. They are used to track data-flow from the environment into the function closure. A simply-typed lambda calculus is used to study the properties of the type theory of open closure types. A distinctive feature of this type theory is that an open closure type of a function can vary in different type contexts. To present an application of the type theory, it is shown that a type derivation establishes a simple non-interference property in the sense of information-flow theory. A publicly available prototype implementation of the system can be used to experiment with type derivations for example programs.

Keywords: Type Systems, Closure Types, Information Flow

1 Introduction

Function types in traditional type systems only provide information about the arguments and return values of the functions but not about the data that is captured in function closures. Such function types naturally lead to simple and compositional type systems.

Recently, syntax-directed type systems have been increasingly used to statically verify strong program properties such as resource usage [8, 7, 6], information flow [5, 15], and termination [1, 3, 2]. In such type systems, it is sometimes necessary and natural to include information in the function types about the data that is captured by closures. To see why, assume that we want to design a type system to verify resource usage. Now consider for example the curried append function for integer lists which has the following type in OCaml.

$$\texttt{append} : \mathit{int\ list} \to \mathit{int\ list} \to \mathit{int\ list}$$

At first glance, we might say that the time complexity of `append` is $O(n)$ if n is the length of the first argument. But a closer inspection of the definition of `append` reveals that this is a gross simplification. In fact, the complexity of the partial function call `app_par = append` ℓ is constant. Moreover, the complexity of the function `app_par` is linear—not in the length of the argument but in the length of the list ℓ that is captured in the function closure.

In general, we have to describe the resource consumption of a curried function $f : A_1 \to \cdots \to A_n \to A$ with n expressions $c_i(a_1, \ldots, a_i)$ such that c_i describes the complexity of the computation that takes place after f is applied to i arguments $a_1, \ldots, a_i$. We are not aware of any existing type system that can verify a statement of this form.

To express the aforementioned statement in a type system, we have to decorate the function types with additional information about the data that is captured in a function closure. It is however not sufficient to directly describe the complexity of a closure in terms of its arguments and the data captured in the closure. Admittedly, this would work to accurately describe the resource usage in our example function `append` because the first argument is directly captured in the closure. But in general, the data captured in a closure $f a_1 \cdots a_i$ can be any data that is computed from the arguments $a_1, \ldots, a_i$ (and from the data in the environment). To reference this data in the types would not only be meaningless for a user, it would also hamper the compositionality of the type system. It is for instance unclear how to define subtyping for closures that capture different data (which is, e.g., needed in the two branches of a conditional.)

To preserve the compositionality of traditional type systems, we propose to describe the resource usage of a closure as a function of its argument and the data that is visible in the current environment. To this end we introduce *open closure types*, function types that refer to their arguments and to the data in the current environment.

More formally, consider a typing judgment of the form $\Gamma \vdash e : \sigma$, in a type system that tracks fine-grained intensional properties characterizing not only the shape of values, but the behavior of the reduction of e into a value (e.g., resource usage). A typing rule for open closure types, $\Gamma, \Delta \vdash e : [\Gamma'](x{:}\sigma) \to \tau$, captures the idea that, under a weak reduction semantics, the computation of the closure itself, and later the computation of the closure *application*, will have very different behaviors, captured by two different typing environments Γ and Γ' of the same domain, the free variables of e. To describe the complexity of `append`, we might for instance have a statement

$$\ell{:}\mathit{int\ list} \vdash \texttt{append}\ \ell : [\ell{:}\mathit{int\ list}](y{:}\mathit{int\ list}) \to \mathit{int\ list}\ .$$

This puts us in a position to use type annotations to describe the resource usage of `append` ℓ as a function of ℓ and the future argument y. For example, using type-based amortized analysis [6], we can express a bound on the number of created list notes in `append` with the following open closure type.

$$\texttt{append} : []({x{:}\mathit{int\ list}}^{0}) \to [x{:}\mathit{int\ list}^{1}](y{:}\mathit{int\ list}^{0}) \to \mathit{int\ list}^{0}\ .$$

The intuitive meaning of this type for `append` is as follows. To pay for the cons operations in the evaluation of `append` ℓ_1 we need $0{\cdot}|\ell_1|$ resource units and to pay for the cons operations in the evaluation of `append` $\ell_1\,\ell_2$ we need $0{\cdot}|\ell_1| + 1{\cdot}|\ell_2|$ resource units.

The development of a type system for open closure types entails some interesting technical challenges: term variables now appear in types, which requires mechanisms for scope management not unlike dependent type theories. If x ap-

pears in σ, the context $\Gamma, x{:}\tau, y{:}\sigma$ is not exchangeable with $\Gamma, y{:}\sigma, x{:}\tau$. Similarly, the judgment $\Gamma, x{:}\tau \vdash e_2 : \sigma$ will not entail $\Gamma \vdash \texttt{let}\, x = e_1\, \texttt{in}\, e_2 : \sigma$, as the return type σ may contain open closures scoping over x, so we need to substitute variables in types.

The main contribution of this paper is a type theory of open closure types and the proof of its main properties. We start from the simply-typed lambda calculus, and consider the simple intensional property of data-flow tracking, annotating each simply-typed lambda-calculus type with a single boolean variable. This allows us to study the metatheory of open closure types in clean and straightforward way. This is the first important step for using such types in more sophisticated type systems for resource usage and termination.

Our type system for data-flow tracking captures higher-order data-flow information. As a byproduct, we get our secondary contribution, a non-interference property in the sense of information flow theory: high-level inputs do not influence the (low-level) results of computations.

To experiment with of our type system, we implemented a software prototype in OCaml (see Section 5). A full version of this article, containing the full proofs and additional details and discussion, is available online.[3]

Related Work. In our type system we maintain the invariant that open closure types only refer to variables that are present in the current typing context. This is a feature that distinguishes open closure types from existing formalisms for closure types.

For example, while our function type $[\Gamma^{\Phi}](x{:}\sigma_1) \to \sigma_2$ superficially resembles a contextual arrow type $[\Psi](\sigma_1 \to \sigma_2)$ of contextual type theory[12, 14, 16], we are not aware of any actual connection in application or metatheory with these systems. In particular, the variable in our captured context Γ^{Φ} are *bound occurrences* of the ambient typing context, while the context Ψ of a contextual type $[\Psi]T$ *binds* metavariables to be used to construct inhabitants. As such a binding can make sense in any context, our substitution judgment has no counterpart in contextual type theory, or other modal type theories for multi-stage programming ([11, 17]).

Having closure types carry a set of captured variables has been done in the literature, as for example in Leroy [9], which use closure types to keep track of of *dangerous type variables* that can not be generalized without breaking type safety, or in the higher-order lifetime analysis of Hannan et al. [4], where variable sets denote variables that must be kept in memory. However, these works have no need to vary function types in different typing contexts and subtyping can be defined using set inclusion, which makes the metatheory significantly simpler. On the contrary, our scoping mechanism allows to study more complex properties, such as value dependencies and non-interference.

The classical way to understand value capture in closures in a typed way is through the *typed closure conversion* of Minamide et al. [10]. They use existential types to account for hidden data in function closures without losing

[3] `http://hal.inria.fr/INRIA-RRRT/hal-00851658`

compositionality, by abstracting over the difference between functions capturing from different environments. Our system retains this compositionality, albeit in a less apparent way: we get finer-grained information about the dependency of a closure on the ambient typing environment. Typed closure conversion is still possible, and could be typed in a more precise way, abstracting only over values that are outside the lexical context.

Petricek et al. [13] study *coeffects* systems with judgments of the form $C^r\,\Gamma \vdash e : \tau$ and function types $C^s\sigma \to \tau$, where r and s are coeffect annotations over an indexed comonad C. Their work is orthogonal to the present one. They study comonadic semantics and algebraic structure of effect indices. These indices are simply booleans in our work but we focus on the syntactic scoping rules that arise from tracking each variable of the context separately.

The non-interference property that we prove is different from the usual treatment in type systems for information flow like the SLam Calculus [5]. In SLam, the information flow into closure is accounted for at abstraction time. In contrast, we account for the information flow into the closure at application time.

2 A Type System for Open Closures

We define a type system for the simplest problem domain that exhibits a need for open closure types. Our goal is to determine statically, for an open term e, on which variables of the environment the value of e depends.

We are interested in weak reduction, and assume a call-by-value reduction strategy. In this context, an abstraction $\lambda x.e$ is already a value, so reducing it does not depend on the environment at all. More generally, for a term e evaluating to a function (closure), we make a distinction between the part of the environment the reduction of e depends on, and the part that will be used when the resulting closure will be applied. For example, the term $(y, \lambda x.z)$ depends on the variable y at evaluation time, but will not need the variable z until the closure in the right pair component is applied.

This is where we need open closure types. Our function types are of the form $[\Gamma^{\Phi}](x{:}\sigma^{\phi}) \to \tau$, where the mapping Φ from variables to Booleans indicates on which variables the evaluation depends at application time. The Boolean ϕ indicates whether the argument x is used in the function body. We call Φ the dependency annotation of Γ. Our previous example would for instance be typed as follows.

$$y{:}\sigma^1, z{:}\tau^0 \vdash (y, \lambda x.z) : \sigma * ([y{:}\sigma^0, z{:}\tau^1](x{:}\rho^0) \to \tau)$$

The typing expresses that the result of the computation depends on the variable y but not on the variable z. Moreover, result of the function in the second component of the pair depends on z but not on y.

In general, types are defined by the following grammar.

$$\begin{array}{rll} \texttt{Types} \ni \sigma, \tau, \rho ::= & & \text{types} \\ \mid & \alpha & \text{atoms} \\ \mid & \tau_1 * \tau_2 & \text{products} \\ \mid & [\Gamma^{\Phi}](x{:}\sigma^{\phi}) \to \tau & \text{closures} \end{array}$$

$$\textsc{Scope-Context-Nil}\quad \emptyset \vdash$$

$$\textsc{Scope-Context}\quad \frac{\Gamma \vdash \sigma}{\Gamma, x{:}\sigma \vdash}$$

$$\textsc{Scope-Atom}\quad \frac{\Gamma \vdash}{\Gamma \vdash \alpha}$$

$$\textsc{Scope-Product}\quad \frac{\Gamma \vdash \tau_1 \qquad \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 * \tau_2}$$

$$\textsc{Scope-Closure}\quad \frac{\Gamma_0, \Gamma_1 \vdash \qquad \Gamma_0 \vdash \sigma \qquad \Gamma_0, x{:}\sigma \vdash \tau}{\Gamma_0, \Gamma_1 \vdash [\Gamma_0^{\Phi}](x{:}\sigma^{\phi}) \to \tau}$$

Fig. 1. Well-scoping of types and contexts

The closure type $[\Gamma^{\Phi}](x{:}\sigma^{\phi}) \to \tau$ binds the new argument variable x, but not the variables occurring in Γ which are reference variables bound in the current typing context. Such a type is *well-scoped* only when all the variables it closes over are actually present in the current context. In particular, it has no meaning in an empty context, unless Γ is itself empty.

We define well-scoping judgments on contexts ($\Gamma \vdash$) and types ($\Gamma \vdash \sigma$). The judgments are defined simultaneously in Figure 1 and refer to each another. They use non-annotated contexts: the dependency annotations characterize data-flow information of *terms*, and are not needed to state the well-formedness of static types and contexts.

Notice that the closure contexts appearing in the return type of a closure, τ in our rule Scope-Closure, may capture the variable x corresponding to the function argument, which is why we chose the dependent-arrow–like notation $(x{:}\sigma) \to \tau$ rather than only $\sigma \to \tau$. There is no dependency of types on terms in this system, this is only used for scope tracking.

Note that $\Gamma \vdash \sigma$ implies $\Gamma \vdash$ (as proved by direct induction until an atom or a function closure is reached). Note also that a context type $[\Gamma_0](x{:}\sigma) \to \tau$ is well-scoped in any larger environment Γ_0, Γ_1: the context information may only mention variables existing in the typing context, but it need not mention all of them. As a result, well-scoping is preserved by context extension: if $\Gamma_0 \vdash \sigma$ and $\Gamma_0, \Gamma_1 \vdash$, then $\Gamma_0, \Gamma_1 \vdash \sigma$.

A Term Language, and a Naive Attempt at a Type System. Our term language, is the lambda calculus with pairs, let bindings and fixpoints. This language is sufficient to discuss the most interesting problems that arise in an application of closure types in a more realistic language.

$\texttt{Terms} \ni t, u, e ::=$		terms
$\mid$	x	variables
$\mid$	(e_1, e_2)	pairs
$\mid$	$\pi_i(e)$	projections ($i \in \{1, 2\}$)
$\mid$	$\lambda x.e$	lambda abstractions
$\mid$	$t\ u$	applications
$\mid$	$\texttt{let}\, x = e_1\, \texttt{in}\, e_2$	let declarations

For didactic purposes, we start with an intuitive type system presented in Figure 2. The judgment $\Gamma^{\Phi} \vdash e : \sigma$ means that the expression e has type σ, in

$$\textsc{Var}\ \frac{}{\Gamma, x{:}\sigma, \Delta \vdash \Gamma^0, x{:}\sigma^1, \Delta^0 \vdash x : \sigma} \qquad \textsc{Product}\ \frac{\Gamma^{\Phi_1} \vdash e_1 : \tau_1 \qquad \Gamma^{\Phi_2} \vdash e_2 : \tau_2}{\Gamma^{\Phi_1+\Phi_2} \vdash (e_1, e_2) : \tau_1 * \tau_2} \qquad \textsc{Proj}\ \frac{\Gamma^{\Phi} \vdash e : \tau_1 * \tau_2}{\Gamma^{\Phi} \vdash \pi_i(e) : \tau_i}$$

$$\textsc{Lam}\ \frac{\Gamma^{\Phi}, x{:}\sigma^{\phi} \vdash t : \tau}{\Gamma^0 \vdash \lambda x.t : [\Gamma^{\Phi}](x{:}\sigma^{\phi}) \to \tau} \qquad \textsc{Let-Tmp}\ \frac{\Gamma^{\Phi_{\mathtt{def}}} \vdash e_1 : \sigma \qquad \Gamma^{\Phi_{\mathtt{body}}}, x{:}\sigma^{\phi} \vdash e_2 : \tau}{\Gamma^{\phi.\Phi_{\mathtt{def}}+\Phi_{\mathtt{body}}} \vdash \mathtt{let}\, x = e_1\, \mathtt{in}\, e_2 : \tau}$$

$$\textsc{App-Tmp}\ \frac{(\Gamma_0, \Gamma_1)^{\Phi_{\mathtt{fun}}} \vdash t : [\Gamma_0^{\Phi_{\mathtt{clos}}}](x{:}\sigma^{\phi}) \to \tau \qquad (\Gamma_0, \Gamma_1)^{\Phi_{\mathtt{arg}}} \vdash u : \sigma}{(\Gamma_0, \Gamma_1)^{\Phi_{\mathtt{fun}}+\Phi_{\mathtt{clos}}+\phi.\Phi_{\mathtt{arg}}} \vdash t\ u : \tau}$$

Fig. 2. Naive rules for the type system

the context Γ carrying the intensional information Φ. Context variable mapped to 0 in Φ are not used during the reduction of e to a value. We will show that the rules App-Tmp and Let-Tmp are not correct, and introduce a new judgment to develop correct versions of the rules.

In a judgment $\Gamma^0 \vdash \lambda x.t : [\Gamma^{\Phi}](x{:}\sigma^0) \to \tau$, Γ is bound only in one place (the context), and α-renaming any of its variable necessitates a mirroring change in its right-hand-side occurrences (Γ^{Φ} but also in σ and τ), while x is independently bound in the term and in the type, so the aforementioned type is equivalent to $[\Gamma^{\Phi}](y{:}\sigma) \to \tau[y/x]$. In particular, variables occurring in types do *not* reveal implementation details of the underlying term.

The syntax $\phi.\Phi$ used in the App-Tmp and Let-Tmp rules is a product, or conjunction, of the single boolean dependency annotation ϕ, and of the vector dependency annotation Φ. The sum $\Phi_1 + \Phi_2$ is the disjunction. In the Let-Tmp rule for example, if the typing of e_2 determines that the evaluation of e_2 does not depend on the definition $x = e_1$ (ϕ is 0), then $\phi.\Phi_{\mathtt{def}}$ will mark all the variables used by e_1 as not needed as well (all 0), and only the variables needed by e_2 will be marked in the result annotation $\phi.\Phi_{\mathtt{def}} + \Phi_{\mathtt{body}}$.

In the scoping judgment $\Gamma \vdash [\Gamma^{\Phi}](x{:}\sigma) \to \tau$, the repetition of the judgment Γ is redundant. We could simply write $[\Phi](x{:}\sigma) \to \tau$; – because in our simplified setting the intensional information Φ can be easily separated from the rest of the typing information, corresponding to simply-typed types. However, we found out that such a reformulation made technical developments harder to follow; the Γ^{Φ} form allows one to keep track precisely of the domain of the dependency annotation, and domain changes are precisely the difficult technical aspect of open closure types. For a more detailed discussion of this design point, see the full version of this article.

Maintaining Closure Contexts. As pointed out before, the rules App-Tmp and Let-Tmp of the system above are wrong (hence the "temporary" name): the left-hand-side of the rule App-Tmp assumes that the closure captures the same environment Γ that it is computed in. This property is initially true in the closure of the rule Lam, but is not preserved by Let-Tmp (for the body type) or

$$\textsc{Subst-Context-Nil}\quad \Gamma, y{:}\rho, \emptyset \overset{y\backslash\Psi}{\rightsquigarrow} \Gamma$$

$$\textsc{Subst-Context}\quad \frac{\Gamma, y{:}\rho, \Delta \vdash \sigma \overset{y\backslash\Psi}{\rightsquigarrow} \Gamma, \Delta' \vdash \tau}{\Gamma, y{:}\rho, \Delta, x{:}\sigma \overset{y\backslash\Psi}{\rightsquigarrow} \Gamma, \Delta', x{:}\tau}$$

$$\textsc{Subst-Atom}\quad \frac{\Gamma, y{:}\rho, \Delta \overset{y\backslash\Psi}{\rightsquigarrow} \Gamma, \Delta'}{\Gamma, y{:}\rho, \Delta \vdash \alpha \overset{y\backslash\Psi}{\rightsquigarrow} \Gamma, \Delta' \vdash \alpha}$$

$$\textsc{Subst-Product}\quad \frac{\Gamma, y{:}\rho, \Delta \vdash \sigma_1 \overset{y\backslash\Psi}{\rightsquigarrow} \Gamma, \Delta' \vdash \tau_1 \qquad \Gamma, y{:}\rho, \Delta \vdash \sigma_2 \overset{y\backslash\Psi}{\rightsquigarrow} \Gamma, \Delta' \vdash \tau_2}{\Gamma, y{:}\rho, \Delta \vdash \sigma_1 * \sigma_2 \overset{y\backslash\Psi}{\rightsquigarrow} \Gamma, \Delta' \vdash \tau_1 * \tau_2}$$

$$\textsc{Subst-Closure-Notin}\quad \frac{\Gamma_0, \Gamma_1, y{:}\rho, \Delta \overset{y\backslash\Psi}{\rightsquigarrow} \Gamma_0, \Gamma_1, \Delta'}{\Gamma_0, \Gamma_1, y{:}\rho, \Delta \vdash [\Gamma_0^{\Phi}](x{:}\sigma_1^{\phi}) \to \sigma_2 \overset{y\backslash\Psi}{\rightsquigarrow} \Gamma_0, \Gamma_1, \Delta' \vdash [\Gamma_0^{\Phi}](x{:}\sigma_1^{\phi}) \to \sigma_2}$$

$$\textsc{Subst-Closure}\quad \frac{\Gamma, y{:}\rho, \Delta, \Gamma_1 \overset{y\backslash\Psi}{\rightsquigarrow} \Gamma, \Delta', \Gamma_1' \qquad \Gamma, y{:}\rho, \Delta \vdash \sigma_1 \overset{y\backslash\Psi}{\rightsquigarrow} \Gamma, \Delta' \vdash \sigma_1 \qquad \Gamma, y{:}\rho, \Delta, x{:}\sigma_1 \vdash \sigma_2 \overset{y\backslash\Psi}{\rightsquigarrow} \Gamma, \Delta', x{:}\sigma_1 \vdash \tau_2}{\Gamma, y{:}\rho, \Delta, \Gamma_1 \vdash [\Gamma^{\Phi_1}, y{:}\rho^{\chi}, \Delta^{\Phi_2}](x{:}\sigma_1^{\phi}) \to \sigma_2 \overset{y\backslash\Psi}{\rightsquigarrow} \Gamma, \Delta', \Gamma_1' \vdash [\Gamma^{\Phi_1 + \chi.\Psi}, \Delta'^{\Phi_2}](x{:}\sigma_1^{\phi}) \to \tau_2}$$

Fig. 3. Type substitution

App-Tmp (for the return type). This means that the intensional information in a type may become stale, mentioning variables that have been removed from the context. We will now fix the type system to never mention unbound variables.

We need a *closure substitution mechanism* to explain the closure type $\tau_f = [\Gamma^{\Phi}, y{:}\rho^{\chi}](x{:}\sigma^{\phi}) \to \tau^{\psi}$ of a closure f in the smaller environment Γ, given dependency information for y in Γ. Assume for example that y was bound in a let binding `let` $y = e \ldots$ and that the type τ_f leaves the scope of y. Then we have to adapt the type rules to express the following. "If f depends on y (at application time) then f depends on the variables of Γ that e depends on."

We define in Figure 3 the judgment $\Gamma, y{:}\rho, \Delta \vdash \sigma \overset{y\backslash\Psi}{\rightsquigarrow} \Gamma, \Delta' \vdash \tau$. Assuming that the variable y in the context $\Gamma, y{:}\rho, \Delta$ was let-bound to an definition with usage information Γ^{Ψ}, this judgment transforms any type σ in this context in a type τ in a context Γ, Δ' that does not mention y anymore. Note that Δ and Δ' have the same domain, only their intensional information changed: any mention of y in a closure type of Δ was removed in Δ'. Also note that $\Gamma, y{:}\rho, \Delta$ and Γ, Δ', or σ and τ, are not annotated with dependency annotations themselves: this is only a scoping transformation that depends on the dependency annotations of y *in the closures* of σ and Δ.

As for the scope-checking judgment, we simultaneously define the substitutions on contexts themselves $\Gamma, y{:}\rho, \Delta \overset{y\backslash\Psi}{\rightsquigarrow} \Gamma, \Delta'$. There are two rules for substituting a closure type. If the variable being substituted is not part of the closure type context (rule Subst-Closure-Notin), this closure type is unchanged. Otherwise (rule Subst-Closure) the substitution is performed in the closure type, and the neededness annotation for y is reported to its definition context Γ_0.

The following lemma verifies that this substitution preserves well-scoping of contexts and types.

Lemma 1 (Substitution and scoping). *If* $\Gamma, y{:}\rho, \Delta \vdash$ *and* $\Gamma, y{:}\rho, \Delta \overset{y\backslash\Psi}{\rightsquigarrow} \Gamma, \Delta'$ *then* $\Gamma, \Delta' \vdash$. *If* $\Gamma, y{:}\rho, \Delta \vdash \sigma$ *and* $\Gamma, y{:}\rho, \Delta \vdash \sigma \overset{y\backslash\Psi}{\rightsquigarrow} \Gamma, \Delta' \vdash \tau$ *then* $\Gamma, \Delta' \vdash \tau$.

We can now give the correct rules for binders:

LET

$$\frac{\Gamma^{\Phi_{\mathtt{def}}} \vdash e_1 : \sigma \qquad \Gamma^{\Phi_{\mathtt{body}}}, x{:}\sigma^{\phi} \vdash e_2 : \tau \qquad \Gamma, x{:}\sigma \vdash \tau \overset{x\backslash\Phi_{\mathtt{def}}}{\rightsquigarrow} \Gamma \vdash \tau'}{\Gamma^{\phi.\Phi_{\mathtt{def}}+\Phi_{\mathtt{body}}} \vdash \mathtt{let}\, x = e_1\, \mathtt{in}\, e_2 : \tau'}$$

APP

$$\frac{\begin{array}{c}(\Gamma_0, \Gamma_1)^{\Phi_{\mathtt{fun}}} \vdash t : [\Gamma_0^{\Phi_{\mathtt{clos}}}](x{:}\sigma^{\phi}) \to \tau \\ (\Gamma_0, \Gamma_1)^{\Phi_{\mathtt{arg}}} \vdash u : \sigma \qquad \Gamma_0, \Gamma_1, x{:}\sigma \vdash \tau \overset{x\backslash\Phi_{\mathtt{arg}}}{\rightsquigarrow} \Gamma_0, \Gamma_1 \vdash \tau'\end{array}}{(\Gamma_0, \Gamma_1)^{\Phi_{\mathtt{fun}}+\Phi_{\mathtt{clos}}+\phi.\Phi_{\mathtt{arg}}} \vdash t\ u : \tau'}$$

Lemma 2 (Typing respects scoping). *If* $\Gamma \vdash t : \sigma$ *holds, then* $\Gamma \vdash \sigma$ *holds.*

This lemma guarantees that we fixed the problem of stale intensional information: types appearing in the typing judgment are always well-scoped.

It is handy to introduce a convenient derived notation $\Gamma^{\Phi} \vdash \tau \overset{y\backslash\Psi}{\rightsquigarrow} \Gamma'^{\Phi'} \vdash \tau'$ that is defined below. This substitution relation does not only remove y from the open closure types in Γ, it also updates the dependency annotation on Γ to add the dependency Ψ, corresponding to all the variables that y depended on – if it is itself marked as needed.

$$\frac{\Gamma, y{:}\rho, \Delta \vdash \tau \overset{y\backslash\Psi}{\rightsquigarrow} \Gamma, \Delta' \vdash \tau'}{\Gamma^{\Phi_1}, y{:}\rho^{\chi}, \Delta^{\Phi_2} \vdash \tau \overset{y\backslash\Psi}{\rightsquigarrow} \Gamma^{\Phi_1+\chi.\Psi}, \Delta'^{\Phi_2} \vdash \tau'}$$

3 A Big-Step Operational Semantics

In this section, we will define an operational semantics for our term language, and use it to prove the soundness of the type system (Theorem 1). Our semantics is equivalent to the usual call-by-value big-step reduction semantics for the lambda-calculus in the sense that computation happens at the same time. There is however a notable difference.

Function closures are not built in the same way as they are in classical big-step semantics. Usually, we have a rule of the form $V \vdash \lambda x.t \implies (V, \lambda x.t)$ where the closure for $\lambda x.t$ is a pair of the value environment V (possibly restricted to its subset appearing in t) and the function code. In contrast, we capture no values at closure creation time in our semantics: $V \vdash \lambda x.t \implies (\emptyset, \lambda x.t)$. The captured values will be added to the closure incrementally, during the reduction of binding forms that introduced them in the context.

Consider for example the following two derivations; one in the classic big-step reduction, and the other in our alternative system.

$$\textsc{Classic-Red-Let}\quad \frac{x{:}v \vdash x \overset{c}{\Longrightarrow} v \qquad x{:}v, y{:}v \vdash \lambda z.y \overset{c}{\Longrightarrow} ((x \mapsto v, y \mapsto v), \lambda z.y)}{x{:}v \vdash \mathtt{let}\, y = x\, \mathtt{in}\, \lambda z.y \overset{c}{\Longrightarrow} ((x \mapsto v, y \mapsto v), \lambda z.y)}$$

$$\textsc{Our-Red-Let}\quad \frac{x{:}v \vdash x \implies v \qquad x{:}v, y{:}v \vdash \lambda z.y \implies ([x, y], \emptyset, \lambda z.y) \qquad (\emptyset, \lambda z.y) \overset{y\backslash v}{\rightsquigarrow} ([x], y \mapsto v, \lambda z.y)}{x{:}v \vdash \mathtt{let}\, y = x\, \mathtt{in}\, \lambda z.y \implies ([x], y \mapsto v, \lambda z.y)}$$

Rather than capturing the whole environment in a closure, we store none at all at the beginning (merely record their names), and add values incrementally, just before they get popped from the environment. This is done by the *value substitution* judgment $w \overset{x\backslash v}{\rightsquigarrow} w'$ that we will define in this section. The reason for this choice is that this closely corresponds to our typing rules, value substitution being a runtime counterpart to substitution in types $\Gamma \vdash \sigma \overset{x\backslash \Phi}{\rightsquigarrow} \Gamma' \vdash \sigma'$; this common structure is essential to prove of the type soundness (Theorem 1).

Note that derivations in this modified system and in the original one are in one-to-one mapping. It should not be considered a new dynamic semantics, rather a reformulation that is convenient for our proofs as it mirrors our static judgment structure.

Values and Value Substitution. Values are defined as follows.

$\mathtt{Val} \ni v, w$	$::=$		values
	$\mid$	$\mathtt{v}_\alpha$	value of atomic type
	$\mid$	(v, w)	value tuples
	$\mid$	$([x_j]_{j\in J}, (x_i \mapsto v_i)_{i\in I}, \lambda x.t)$	function closures

The set of variables bound in a closure is split into an ordered mapping $(x_i \mapsto v_i)_{i\in I}$ for variables that have been substituted to their value, and a simple list $[x_j]_{j\in J}$ of variables whose value has not yet been captured. They are both binding occurrences of variables bound in t; α-renaming them is correct as long as t is updated as well.

To formulate our type soundness result, we define a typing judgment on values $\Gamma \vdash v : \sigma$ in Figure 4. An intuition for the rule Value-Closure is the following. Internally, the term t has a dependency Γ^Φ on the ambient context, but also dependencies $(\tau_i^{\psi_i})$ on the captured variables. But externally, the type may not mention the captured variables, so it reports a different dependency $\Gamma^{\Phi'}$ that corresponds to the internal dependency Γ^Φ, combined with the dependencies (Ψ_i) of the captured values. Both families $(\psi_i)_{i\in I}$ and $(\Psi_i)_{i\in I}$ are existentially quantified in this rule.

In the judgment rule, the notation $(x_j : \tau_j)_{j<i}$ is meant to define the environment of each $(x_i : \tau_i)$ as Γ^Φ, plus all the $(x_j : \tau_j)$ that come before x_i in the

$$\textsc{Value-Atom}\ \frac{\Gamma \vdash}{\Gamma \vdash \mathtt{v}_\alpha : \alpha} \qquad \textsc{Value-Product}\ \frac{\Gamma \vdash v_1 : \tau_1 \qquad \Gamma \vdash v_2 : \tau_2}{\Gamma \vdash (v_1, v_2) : \tau_1 * \tau_2}$$

$$\textsc{Value-Closure}\ \frac{\Gamma, \Gamma_1 \vdash \qquad \forall i \in I,\ \Gamma, (x_j{:}\tau_j)_{j<i} \vdash v_i : \tau_i \qquad \Gamma^{\Phi}, (x_i{:}\tau_i^{\psi_i})_{i\in I}, x{:}\sigma^{\phi} \vdash t : \tau \qquad \Gamma^{\Phi}, (x_i{:}\tau_i^{\psi_i})_{i\in I}, x{:}\sigma^{\phi} \vdash \tau \overset{(x_i)\backslash(\Psi_i)}{\rightsquigarrow} \Gamma^{\Phi'}, x{:}\sigma^{\phi} \vdash \tau'}{\Gamma, \Gamma_1 \vdash (\operatorname{dom}\Gamma, (x_i \mapsto v_i)_{i\in I}, \lambda x.t) : [\Gamma^{\Phi'}](x{:}\sigma^{\phi}) \to \tau'}$$

Fig. 4. Value typing

$$\textsc{Subst-Value-Atom}\ \mathtt{v}_\alpha \overset{y\backslash v}{\rightsquigarrow} \mathtt{v}_\alpha \qquad \textsc{Subst-Value-Product}\ \frac{w_1 \overset{y\backslash v}{\rightsquigarrow} w_1' \qquad w_2 \overset{y\backslash v}{\rightsquigarrow} w_2'}{(w_1, w_2) \overset{y\backslash v}{\rightsquigarrow} (w_1', w_2')}$$

$$\textsc{Subst-Value-Closure}\ ([x_{j_1}, \ldots, x_{j_n}, y], (x_i \mapsto w_i)_{i\in I}, t) \overset{y\backslash v}{\rightsquigarrow} ([x_{j_1}, \ldots, x_{j_n}], (y \mapsto v)(x_i \mapsto w_i)_i, t)$$

$$\textsc{Subst-Value-Closure-Notin}\ \frac{y \notin (x_j)_{j\in J}}{([x_j]_{j\in J}, (x_i \mapsto w_i)_{i\in I}, t) \overset{y\backslash v}{\rightsquigarrow} ([x_j]_{j\in J}, (x_i \mapsto w_i)_{i\in I}, t)}$$

Fig. 5. Value substitution

typing judgment $\Gamma^{\Phi}, (x_i : \tau_i)_{i\in I}, x : \sigma^{\phi} \vdash t$. The notation $\ldots \overset{(x_i)\backslash(\Psi_i)}{\rightsquigarrow} \ldots$ denotes the sequence of substitutions for all (x_i, Ψ_i), with the rightmost variable (introduced last) substituted first: in our dynamic semantics, values are captured by the closure in the LIFO order in which their binding variables enter and leave the lexical scope.

Substituting Values. The value substitution judgment, define in Figure 5, is an operational counterpart to the substitution of variables in closures types.

Lemma 3 (Value substitution preserves typing). *If* $(\Gamma \vdash v : \rho)$, $(\Gamma, y{:}\rho \vdash w : \sigma)$, $(\Gamma, y{:}\rho \vdash \sigma \overset{y\backslash\Psi}{\rightsquigarrow} \Gamma \vdash \tau)$ *and* $(w \overset{y\backslash v}{\rightsquigarrow} w')$ *hold, then* $(\Gamma \vdash w' : \tau)$ *holds.*

The Big-Step Reduction Relation. We are now equipped to define in Figure 6 the big-step reduction relation on well-typed terms $V \vdash e \implies v$, where V is a mapping from the variables to values that is assumed to contain at least all the free variables of e. The notation $w \overset{V_2}{\rightsquigarrow} w'$ denotes the sequence of substitutions for each (variable, value) pair in V_2, from the last one introduced in the context to the first; the intermediate values are unnamed and existentially quantified.

$$\textsc{Red-Var}\quad V \vdash x \implies V(x) \qquad \textsc{Red-Lam}\quad V \vdash \lambda x.t \implies (\operatorname{dom} V, \emptyset, \lambda x.t)$$

$$\textsc{Red-Pair}\quad \frac{V \vdash e_1 \implies v_1 \qquad V \vdash e_2 \implies v_2}{V \vdash (e_1, e_2) \implies (v_1, v_2)}$$

$$\textsc{Red-Proj}\quad \frac{V \vdash e \implies (v_1, v_2)}{V \vdash \pi_i(e) \implies v_i} \qquad \textsc{Red-Let}\quad \frac{V \vdash e_1 \implies v_1 \qquad V, (x \mapsto v_1) \vdash e_2 \implies v_2 \qquad v_2 \overset{x \setminus v_1}{\rightsquigarrow} v_2'}{V \vdash \mathtt{let}\, x = e_1\, \mathtt{in}\, e_2 \implies v_2'}$$

$$\textsc{Red-App}\quad \frac{\begin{array}{c} V, V_1 \vdash t \implies (\operatorname{dom} V, V_2, \lambda y.t') \\ V, V_1 \vdash u \implies v_{\mathtt{arg}} \qquad V, V_1, V_2, y \mapsto v_{\mathtt{arg}} \vdash t' \implies w \qquad w \overset{y \setminus v_{\mathtt{arg}}}{\rightsquigarrow} w' \overset{V_2}{\rightsquigarrow} w'' \end{array}}{V, V_1 \vdash t\ u \implies w''}$$

Fig. 6. Big-step reduction rules

$$\textsc{Classic-Red-Lam}\quad W \vdash \lambda x.t \overset{c}{\implies} (W, \lambda x.t)$$

$$\textsc{Classic-Red-Let}\quad \frac{W \vdash e_1 \overset{c}{\implies} w_1 \qquad W, x \mapsto w_1 \vdash e_2 \overset{c}{\implies} w_2}{W \vdash \mathtt{let}\, x = e_1\, \mathtt{in}\, e_2 \overset{c}{\implies} w_2}$$

$$\textsc{Classic-Red-App}\quad \frac{W \vdash t \overset{c}{\implies} (W', \lambda y.t') \qquad W \vdash u \overset{c}{\implies} w_{\mathtt{arg}} \qquad W', y \mapsto w_{\mathtt{arg}} \vdash t' \overset{c}{\implies} w}{W \vdash t\ u \overset{c}{\implies} w}$$

Fig. 7. Classic big-step reduction rules

We write $V : \Gamma \vdash$ if the context valuation V, mapping free variables to values, is well-typed according to the context Γ. The definition of this judgment is given in the full version.

Theorem 1 (Type soundness). *If $\Gamma^{\Phi} \vdash t : \sigma$, $V : \Gamma \vdash$ and $V \vdash t \implies v$ then $\Gamma \vdash v : \sigma$.*

Finally, we recall the usual big-step semantics for the call-by-value calculus with environments, in Figure 7, and state its equivalence with our utilitarian semantics. Due to space restriction we will only mention the rules that differ, and elide the equivalence proof, but the long version contains all the details.

There is a close correspondence between judgments of both semantics, but as the value differ slightly, in the general cases the value bindings of the environment will also differ. We state the theorem only for closed terms, but the proof will proceed by induction on a stronger induction hypothesis using an equivalence between non-empty contexts.

Theorem 2 (Semantic equivalence). *Our reduction relation is equivalent with the classic one on closed terms: $\emptyset \vdash t \implies v$ holds if and only if $\emptyset \vdash t \overset{c}{\implies} v$ also holds.*

To formulate our induction hypothesis, we define the equivalence judgment $V \vdash v = W \overset{c}{\vdash} w$; on each side of the equal sign there is a context and a value, the right-hand side being considered in the classical semantics.

$$\emptyset \vdash \ = \emptyset \overset{c}{\vdash} \qquad \frac{V \vdash \ = W \overset{c}{\vdash} \qquad V \vdash v = W \overset{c}{\vdash} w}{V, x \mapsto v \vdash \ = W, x \mapsto w \overset{c}{\vdash}} \qquad \frac{V \vdash \ = W \overset{c}{\vdash}}{V \vdash \mathtt{v}_\alpha = W \overset{c}{\vdash} \mathtt{v}_\alpha}$$

$$\frac{V \vdash v_1 = W \overset{c}{\vdash} w_1 \qquad V \vdash v_2 = W \overset{c}{\vdash} w_2}{V \vdash (v_1, v_2) = W \overset{c}{\vdash} (w_1, w_2)} \qquad \frac{V \vdash \ = W \overset{c}{\vdash} \qquad V, x_i \mapsto v_i \vdash \ = W' \overset{c}{\vdash}}{V \vdash ((x_i \mapsto v_i)_{i \in I}, \lambda x.t) = W \overset{c}{\vdash} (W', \lambda x.t)}$$

Fig. 8. Equivalence of semantic judgements.

The stronger version of the theorem becomes the following: if $V \vdash \ = W \overset{c}{\vdash}$ and $V \vdash t \implies v$ and $W \vdash t \overset{c}{\implies} w$, then $V \vdash v = W \overset{c}{\vdash} w$.

4 Dependency Information as Non-Interference

We can formulate our dependency information as a *non-interference* property. Two valuations V and V' are Φ-equivalent, noted $V =_\Phi V'$, if they agree on all variables on which they depend according to Φ. We say that e respects non-interference for Φ when, whenever $V \vdash e \implies v$ holds, then for any V' such that $V =_\Phi V'$ we have that $V' \vdash e \implies v$ also holds. This corresponds to the information-flow security idea that variables marked 1 are low-security, while variables marked 0 are high-security and should not influence the output result.

This non-interference statement requires that the two evaluations of e return the same value v. This raises the question of what is the right notion of equality on values. Values of atomic types have a well-defined equality, but picking the right notion of equality for function types is more difficult. While we can state a non-interference result on atomic values only, the inductive subcases would need to handle higher-order cases as well.

Syntactic equality (even modulo α-equivalence) is not the right notion of equality for closure values. Consider the following example: $x{:}\tau^0 \vdash \mathtt{let}\, y = x\ \mathtt{in}\ \lambda z.z : [x{:}\tau^0](z : \sigma^1) \to \sigma$. This term contains an occurrence of the variable x, but its result does not depend on it. However, evaluating it under two different contexts $x{:}v$ and $x{:}v'$, with $v \neq v'$, returns distinct closures: $(x \mapsto v, \lambda z.z)$ on one hand, and $(x \mapsto v', \lambda z.z)$ on the other. These closures are not structurally equal, but their difference is not essential since they are indistinguishable in any context. Logical relations are the common technique to ignore those internal differences and get a more observational equality on functional values. They involve, however, a fair amount of metatheoretical effort that we would like to avoid.

Consider a different example: $x{:}\tau^0 \vdash \lambda y.x : [x{:}\tau^1](y{:}\sigma^0) \to \tau$. Again, we could use two contexts $x{:}v$ and $x{:}v'$ with $v \neq v'$, and we would get as a result two closures: $x{:}v \vdash \lambda y.x \implies (x \mapsto v, \lambda y.x)$ and $x{:}v' \vdash \lambda y.x \implies (x \mapsto v', \lambda y.x)$. Interestingly, these two closures are *not* equivalent under all contexts: any context applying the function will be able to observe the different results. However, our notion of interference requires that they can be considered equal. This is

$$\textsc{Equiv-Atom}\quad \Gamma \vdash \mathtt{v}_\alpha =_{\Phi_0} \mathtt{v}_\alpha : \alpha \qquad \textsc{Equiv-Pair}\quad \frac{\Gamma \vdash v_1 =_{\Phi_0} v_1' : \sigma_1 \qquad \Gamma \vdash v_2 =_{\Phi_0} v_2' : \sigma_2}{\Gamma \vdash (v_1, v_2) =_{\Phi_0} (v_1', v_2') : \sigma_1 * \sigma_2}$$

$$\textsc{Equiv-Closure}\quad \frac{\begin{array}{c}\forall i \in I,\ \Gamma, (x_j{:}\tau_j)_{j<i} \vdash v_i : \tau_i \qquad \Gamma^{\Phi}, (x_i{:}\tau_i^{\psi_i})_{i\in I}, x{:}\sigma^{\phi} \vdash t : \tau \\ \Gamma^{\Phi}, (x_i{:}\tau_i^{\psi_i})_{i\in I}, x{:}\sigma^{\phi} \vdash \tau \overset{(x_i)\setminus(\Psi_i)}{\rightsquigarrow} \Gamma^{\Phi'}, x{:}\sigma^{\phi} \vdash \tau' \qquad \forall i \in I, \Psi_i \subseteq \Phi_0 \implies v_i =_{\Phi_0} v_i'\end{array}}{\Gamma \vdash ((x_i \mapsto v_i)_{i\in I}, \lambda y.t) =_{\Phi_0} ((x_i \mapsto v_i')_{i\in I}, \lambda y.t) : [\Gamma^{\Phi'}](x{:}\sigma) \to \tau'}$$

Fig. 9. Value equivalence

motivated by real-world programming languages that only output a pointer to a closure in a program that returns a function.

While the aforementioned closures are not equal in any context, they are in fact equivalent from the point of view of the particular dependency annotation for which we study non-interference, namely $x{:}\tau^0$. To observe the difference between those closures, we would need to apply the closure of type $[x{:}\tau^1](y : \sigma) \to \tau$, so would be in the different context $x{:}\tau^1$.

This insight leads us to our formulation of value equivalence in Figure 9. Instead of being as modular and general as a logical-relation definition, we fix a *global dependency* Φ_0 that restricts which terms can be used to differentiate values.

Our notion of value equivalence, $\Gamma \vdash v =_{\Phi_0} v' : \sigma$ is typed and includes structural equality. In the rule Equiv-Closure, we check that the two closures values are well-typed, and only compare captured values whose dependencies are included in those of the global context Φ_0, as we know that the others will not be used. This equality is tailored to the need of the non-interference result, which only compares values resulting from the evaluation of the same subterm – in distinct contexts.

Theorem 3 (Non-interference). *If $\Gamma^{\Phi_0} \vdash e : \sigma$ holds, then for any contexts V, V' such that $V =_{\Phi_0} V'$ and values v, v' such that $V \vdash e \implies v$ and $V' \vdash e \implies v'$, we have $\Gamma \vdash v =_{\Phi_0} v' : \sigma$. In particular, if σ is an atomic type, then $v = v'$ holds.*

5 Prototype Implementation

To experiment with our type system, we implemented a software prototype in OCaml. At around one thousand lines, the implementation mainly contains two parts.

1. For each judgement in this paper, a definition of corresponding set of inference rules along with functions for building and checking derivations.
2. A (rudimentary) command-line interface that is based on a lexer, a parser, and a pretty-printer for the expressions, types, judgments and derivations of our system.

For the scope checking judgments for context and types, the implementation *checks* well-scoping of the given contexts and types. It either builds a derivation using the well-scoping rules or fails to do so because of ill-scoped input.

For the typing judgment, the implementation performs some *inference*. Given a type context Γ and an expression e, it returns Φ, σ, and a derivation $\Gamma^{\Phi} \vdash e : \sigma$ if such a derivation exists. Otherwise it fails. The substitution and reduction judgments are deterministic and computational in nature. Our implementation takes the left-hand side a judgement (with additional parameters) and *computes* the right-hand-side of the judgment along with a derivation.

Below is an example of interaction with the prototype interface:

```
% make
% ./closures.byte -str "let y = (y1, y2) in (y, \(x:\sigma) z)"
Parsed expression: let y = (y1, y2) in (y, λ(x:σ) z)

The variables (y1, y2, z) were unbound; we add them to the default
environment with dummy types (ty_y1, ty_y2, ty_z) and values
(val_y1, val_y2, val_z).

Inferred typing:
  y1:ty_y1[1],y2:ty_y2[1],z:ty_z[0] ⊢
    let y = (y1, y2) in (y, λ(x:σ) z)
    : ((ty_y1 * ty_y2) * [y1:ty_y1[0],y2:ty_y2[0],z:ty_z[1]](x:σ[0]) → ty_z)

Result value:
   ((val_y1, val_y2), ([y1,y2,z], ((y ↦ (val_y1, val_y2))), λ(x) z))
```

In this example, adapted from the starting example of the article, $y{:}\sigma^1, z{:}\tau^0 \vdash (y, \lambda x.z)$, one can observe that the value z is marked as non-needed by the global value judgment, but needed in the type of the closure $\lambda x.z$. Besides, the computed value closure has captured the local variable y, but still references the variables $y1, y2$, and z of the outer context.

The prototype can also produce ASCII rendering of the typing and reduction derivations, when passed `--typing-derivation` or `--reduction-derivation`. This can be useful in particular in the case of typing or reduction errors, as a way to locate the erroneous sub-derivation.

The complete source code of the prototype is available at the following URL: `http://gallium.inria.fr/~scherer/research/open_closure_types`

6 Discussion

Before we conclude, we highlight three technical points that deserve a more in-depth discussion and that are helpful link our work to existing and future work.

Typed Closure Conversion. It is interesting to relate our open closure types and typed closure conversion of Minamide et al. [10]. In the classical semantics,

a λ-term $\Gamma \vdash \lambda x.e : \sigma_1 \to \sigma_2$ evaluates under the value binding W to a pair $(W, \lambda x.e)$, which can be given the type $(\Gamma * (\Gamma \to \sigma_1 \to \sigma_2))$ (writing Γ for the product of all types in the context). To combine closures of the same observable type that capture different environments, one needs to abstract away the environment type by using the existential type $\exists\rho.(\rho * (\rho \to \sigma \to \tau))$.

In our specific semantics, a closure that was originally defined in the environment Γ, Δ but is then seen in the environment Γ, only captures the values of variables in Δ. Typed closure conversion is still possible, but we would need to give it the less abstract type $\forall\Gamma.\exists\rho(\rho * (\Gamma \to \rho \to \sigma_1 \to \sigma_2))$. This reflects how our open closure types allow closure types to contain static information about variables of the current lexical context, while still allowing free composition of closures that were initially defined in distinct environments. Our closure types evolve from a very open type, at the closure construction point, into the usual "closure conversion" type that is completely abstract in captured values, in the empty environment.

Subtyping and Conservativity. As mentioned, our type system is *not* conservative over the simply-typed lambda-calculus because of the restriction on substitution of function types (domain types must be preserved by substitution). This is not a surprise as our types provide more fine-grained information without giving a way to forget some of this more precise information. Regaining conservativity is very simple. One needs a notion of subtyping allowing to hide variables present in closure types (eg., $[\Gamma, \Delta](x : \sigma_1) \to \sigma_2 \leq [\Gamma](x : \sigma_1) \to \sigma_2$ whenever σ_1, σ_2 are well-scoped under Γ alone). Systematically coercing all functions into closures capturing the empty environment then gives us exactly the simply-typed lambda-calculus.

Polymorphism. We feel the two previous points could easily be formally integrated in our work. A more important difference between our prototypical system and a realistic framework for program analysis is the lack of polymorphism. This could require significantly more work and is left for future work. We conjecture that adding abstraction on type variables (and their annotation ϕ) is direct, but a more interesting question is the abstraction over annotated contexts Γ^{Φ}. For example, we could want to write the following, where κ is a formal context variable:

$$\vdash \lambda f.\lambda x.\lambda y.fyx : \forall\kappa\alpha\beta\gamma.([\kappa](x{:}\alpha) \to [\kappa](y{:}\beta) \to \gamma) \to ([\kappa](y{:}\beta) \to [\kappa](x{:}\alpha) \to \gamma)$$

Polymorphism seems to allow greater flexibility in the analysis of functions taking functions as parameters. This use of polymorphism is related to the "resource polymorphism" of [7], which serves the same purpose of leaving freedom to input functions. Open closure types on the other hand, are motivated by expressions that *return* function closures; the flip side of the higher-order coin.

7 Conclusion

We have introduced open closure types and their type theory. The technical novelty of the type system is the ability to track intensional properties of function application in function closures types. To maintain this information,we have to update function types when they escape to a smaller context. This update is performed by a novel non-trivial substitution operation. We have proved the soundness of this substitution and the type theory for a simply-typed lambda calculus with pairs and let bindings.

To demonstrate how our open closure types can be used in program verification we have applied this technique to track data-flow information and to ensure non-interference in the sense of information-flow theory. We envision open closure types to be applied in the context of type systems for strong intensional properties of higher-order programs, and this simple system to serve as a guideline for more advanced applications.

We already have preliminary results from an application of open closure types in amortized resource analysis [7, 6]. Using them, we were for the first time able to express a linear resource bound for the curried append function (see Section 1).

Acknowledgments. This research is based on work supported in part by DARPA CRASH grant FA8750-10-2-0254 and NSF grant CCF-1319671. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

References

1. Abel, A.: Semi-continuous Sized Types and Termination. Log. Methods Comput. Sci. 4(2) (2008)
2. Barthe, G., Grégoire, B., Riba, C.: Type-Based Termination with Sized Products. In: Computer Science Logic, 17th Ann. Conf. (CSL'08). pp. 493–507 (2008)
3. Chin, W.N., Khoo, S.C.: Calculating Sized Types. High.-Ord. and Symb. Comp. 14(2-3), 261–300 (2001)
4. Hannan, J., Hicks, P., Liben-Nowell, D.: A Lifetime Analysis for Higher-Order Languages. Tech. rep., The Pennsylvania State University (1997), `http://www.cse.psu.edu/~hannan/papers/live.ps.gz`
5. Heintze, N., Riecke, J.G.: The SLam Calculus: Programming with Secrecy and Integrity. In: 25th Symp. on Principles of Programming Languages (POPL'98). pp. 365–377 (1998)
6. Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate Amortized Resource Analysis. ACM Trans. Program. Lang. Syst. (2012)
7. Jost, S., Hammond, K., Loidl, H.W., Hofmann, M.: Static Determination of Quantitative Resource Usage for Higher-Order Programs. In: 37th Symp. on Principles of Programming Languages (POPL'10). pp. 223–236 (2010)
8. Lago, U.D., Petit, B.: The Geometry of Types. In: 40th Symp. on Principles of Programming Languages (POPL'13). pp. 167–178 (2013)
9. Leroy, X.: Polymorphic Typing of an Algorithmic Language. Research report 1778, INRIA (1992)

10. Minamide, Y., Morrisett, J.G., Harper, R.: Typed Closure Conversion. In: 23rd Symp. on Principles of Programming Languages (POPL'96). pp. 271–283 (1996)
11. Moggi, E., Taha, W., Zine El-abidine Benaissa, Sheard, T.: An idealized metaml: Simpler, and more expressive. In: 8th Europ. Symp. on Programming (ESOP'99). pp. 193–207 (1999)
12. Nanevski, A., Pfenning, F., Pientka, B.: Contextual Modal Type Theory. ACM Trans. Comput. Log. 9(3) (2008)
13. Petricek, T., Orchard, D., Mycroft, A.: Coeffects: Unified static analysis of context-dependence. In: Automata, Languages, and Programming - 40th Int. Colloq. (ICALP'13). pp. 385–397 (2013)
14. Pientka, B., Dunfield, J.: Programming with Proofs and Explicit Contexts. In: 10th International Conference on Principles and Practice of Declarative Programming (PPDP'08). pp. 163–173 (2008)
15. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications 21(1), 5–19 (2003)
16. Stampoulis, A., Shao, Z.: Static and User-Extensible Proof Checking. In: 39th Symp. on Principles of Programming Languages (POPL'12). pp. 273–284 (2012)
17. Tsukada, T., Igarashi, A.: A Logical Foundation for Environment Classifiers. Logical Methods in Computer Science 6(4) (2010)