A Prototype Dependency Calculus

Peter Thiemann

Universität Freiburg
thiemann@informatik.uni-freiburg.de

Abstract. Dependency has been identified as the main ingredient underlying many program analyses, in particular flow analysis, secrecy and integrity analysis, and binding-time analysis. Driven by that insight, Abadi, Banerjee, Heintze, and Riecke [1] have defined a dependency core calculus (DCC). DCC serves as a common target language for defining the above analyses by translation *to* DCC.
The present work considers the opposite direction. We define a Prototype Dependency Calculus (PDC) and define flow analysis, secrecy analysis, and region analysis by translation *from* PDC.

1 Introduction

Dependency plays a major role in program analysis. There are two broad kinds of dependency, value dependency and control dependency. A value dependency from v to v' states that value v' is constructed from v, so that there is a direct influence of v on v'. A typical value dependency arises from the use of a primitive operation, $v' = p(v)$, or from just passing the value as a parameter. A control dependency from v to v' describes an indirect influence where v controls the construction of v'. A typical example of a control dependency is "$v' =$ if v then $\ldots$". Although v does not contribute to v', it still controls the computation of v'.

Many program analysis questions are derived from dependency information. A flow analysis answers questions like: which program points may have contributed to the construction of a value? Hence, a flow analysis is only interested in value dependencies. A region analysis answers similar questions, but instead of yielding answers in terms of program points, the analysis computes regions, which we regard as abstractions of sets of program points. Other analyses, like secrecy and binding-time analyses, also need to consider control dependencies. Here, the typical question is: if a particular value changes, which other values may also change?

The dependency core calculus DCC [1] is an attempt to unify a number of calculi that rely on dependency information. DCC builds on Moggi's computational metalanguage [10]. It formalizes the notion of dependency using a set of monads T_ℓ, indexed by the elements of a lattice. The idea is that each element ℓ of the lattice stands for a certain level of dependency and that computations up to that level must occur in monad T_ℓ (an ℓ-computation). In particular, the bind-operator of DCC guarantees that the outcome of an ℓ-computation can be constructed only from the results of ℓ'-computations, where $\ell' \sqsubseteq \ell$ in the lattice.

D. Le Métayer (Ed.): ESOP 2002, LNCS 2305, pp. 228–242, 2002.

DCC has a denotational semantics and there are translations *from* various calculi — a two-level lambda calculus (binding-time analysis), SLam (secrecy analysis), a flow-type system (flow analysis) — into DCC. Each translation instantiates L to a suitable lattice — $\{static, dynamic\}$, $\{low\text{-}security, high\text{-}security\}$, powerset of the set of program points — and uses a variant of the monadic translation.

This beautiful approach has a number of drawbacks:

- There is no direct link from DCC to the region calculus [20]. The DCC authors [4] have made a separate effort to build a DCC-style denotational model for it.
  However, the region calculus is an important program analysis which relies on dependecy information to reason about memory allocation and memory reuse.
- DCC does not have a notion of polymorphism and it does not seem easy to extend it in this way. In particular, while a fixed number of regions can be tackled with a fixed lattice of dependency levels (viz. the flow-type system), an extension to region polymorphism does not seem to be possible.
  An extension to polymorphism is possible in the denotational model of the region calculus due to Banerjee et al [4]. But, as the authors point out in the conclusion, it is not clear how to unify this model with their work on DCC.

The present work approaches the problems from the other side. Instead of giving a target calculus that can be used as a common meta-language, we start from a prototypical calculus that collects all sensible dependency information. Instantiating this calculus to a particular analysis means to chop away part of the information that it provides. The appeal of this approach is that

- it can provide a common framework for a range of analyses: analyses can be factored through PDC and exploit PDC's minimal typing property for implementing the analyses;
- it scales easily to polymorphism;
- it can be mapped to all the problems that have been translated to DCC;
- it can be mapped to region calculus;
- a non-interference result for the PDC would give rise to corresponding properties for all analyses that are images of PDC;
- it uncovers an interesting connection between region calculus, dependency analysis, and flow analysis.

The main novelty of PDC is the modeling of dependency information as a graphical effect. That is, the effect of evaluating an expression e is a relation that approximates the data and control flow during the evaluation. The mappings from PDC to other calculi are abstraction mappings that cut away excess information.

There are also drawbacks in this approach. In particular, the evaluation strategy (call-by-value) is built into the calculus and some proofs need to be redone when changing the strategy. DCC avoids this by being based on the computational metalanguage. However, for a call-by-value language, the DCC authors

recommend a modified vDCC calculus that has a slightly different model for non-interference. It is hard to argue for the completeness of PDC. It might be that we missed an important kind of dependency in the construction. We cannot be certain until we encounter an analysis, which is not an image of PDC.

Related Work The present work draws heavily on the large body of work on type-based program analysis. We cannot aim for completeness here, for a good overview see the recent survey paper by Palsberg in the PASTE'01 workshop. Particularly influencing are the works on region calculus [20], on effect systems [17, 13, 14], on flow analysis [19], on secrecy and security analysis [8, 16, 22, 21], on binding-time analysis [3, 9, 12, 5, 6].

The only unifying efforts that we are aware of are the works of Abadi, Banerjee, Heintze, and Riecke [4, 1] that we discussed above.

Abadi et al [2] have defined a labeled lambda calculus which enables tracking of dependencies. From the labels present in a normalized expression they compute a pattern that matches lambda expressions with the same normal form. A cache maps these patterns to their respective normal forms. While their labeled expressions only represent the final results, our graphical approach can provide information about intermediate results, too.

Contributions We introduce the PDC and give its static and dynamic semantics. Next, we prove standard properties about its type system, culminating in a minimal typing result. We prove the soundness of the static semantics with respect to an instrumented big-step operational semantics. We define type preserving translations from PDC to the region calculus, to a particular flow analysis, and to SLam (a calculus of secrecy)[1]. Our translations shed some light on the relation between flow analysis and region analysis: a flow analysis is concerned with program points while a region analysis is concerned with portions of memory. In our setting, a region is an abstraction for a set of program points (which shares a common pool of memory to store its results).

2 The Calculus PDC

In this section, we fix the syntax of PDC, define its static semantics, and give an instrumented dynamic (big-step operational) semantics. Finally, we prove type preservation with respect to the static semantics.

2.1 Syntax

An expression, *Expr*, is a variable, a recursive function, a function application, a let-expression, a base-type constant, a primitive operation, or a conditional.

Similar to labeled expressions in flow analysis, all expressions carry a source label, $s \in \mathit{Source}$. Contrary to labels in flow analysis, these source labels need

[1] This is found in an extended version of the paper.

$$
\begin{array}{lll}
\mathit{Expr} \ni e & ::= & x^s \mid \mathtt{rec}^s\ f(x)\, e \mid e@^s e \mid \mathtt{let}^s\ x = e\ \mathtt{in}\ e \mid \\
 & & c^s \mid \mathtt{op}^s(e) \mid \mathtt{if}^s\ e\ e\ e \\
\quad s & \in & \mathit{Source} \\
\mathit{AType} \ni \phi & ::= & (\tau, s) \\
\mathit{Type} \ni \tau & ::= & \alpha \mid \mathbf{B} \mid \phi \xrightarrow{\ \epsilon\ } \phi \\
\quad \epsilon & \in & \mathit{Effect} = \mathbf{P}(\mathit{Source} \times \mathit{Source} \times \mathit{Indicator}) \\
\quad \iota & \in & \mathit{Indicator} = \{V, W, C\} \qquad \text{where} \qquad V \sqsubset W \text{ and } W \sqsubset C \\
\quad \mathit{TE} & \in & \mathit{Variable} \xrightarrow{\mathit{fin}} \mathit{AType}
\end{array}
$$

Fig. 1. Syntax of PDC

not be unique. In fact, they should be looked upon as variables that may be substituted later on.

We shall not define explicitly the underlying expressions of an applied lambda calculus. Rather, we define them intuitively through an erasure mapping $|\cdot|$, where $|e|$ is an expression with the same structure as e, but all source annotations removed.

An annotated type, ϕ, is a pair (τ, s) where τ is a type and s is a source annotation. A type can be a type variable, a base type, or a function type. The function arrow is decorated with a source annotation and an effect ϵ. An effect is a labeled graph where the edges are *atomic* dependencies.

An atomic dependency is either a value dependency (s, s', V) (written as $(s, s')_V$), a control dependency (s, s', C) (written as $(s, s')_C$), or a weak control dependency (s, s', W) (written as $(s, s')_W$). A control dependency is weak, if it does not lead to the construction or examination of a value. If we do not care about the kind of dependency, we write (s, s'). Dependency indicators are totally ordered by $V \sqsubset W$ and $W \sqsubset C$. An effect always stands for the least reflexive and transitive relation generated by its atomic dependencies, as formalized by the judgement $(s, s, \iota) \in \epsilon$:

$$
(\textit{e-atom})\frac{(s, s', \iota) \in \epsilon}{(s, s', \iota) \in \epsilon} \qquad (\textit{e-refl})\ (s, s, \iota) \in \epsilon
$$

$$
(\textit{e-trans})\frac{(s_1, s_2, \iota_1) \in \epsilon \quad (s_2, s_3, \iota_2) \in \epsilon}{(s_1, s_3, \iota_1 \sqcup \iota_2) \in \epsilon}
$$

That is, if there is a single control dependency on the path from s to s', then there is a control dependency $(s, s')_C$. If all atomic steps on the path from s to s' are value dependencies, then there is a value dependency $(s, s')_V$.

For comparison with the simply-typed lambda calculus, we define:

$$
\begin{array}{lll}
\mathit{BType} \ni \zeta & ::= & \alpha \mid \mathbf{B} \mid \zeta \to \zeta \\
 & & \mathit{BTE} \ \in \ \mathit{Variable} \xrightarrow{\mathit{fin}} \mathit{BType}
\end{array}
$$

$$(var)\frac{TE(x) = (\tau, s'')}{TE, s \vdash x^{s'} : (\tau, s') \,!\, \{(s, s')_W, (s'', s')_V\}}$$

$$(rec)\frac{TE[f \mapsto (\phi' \xrightarrow{\epsilon} \phi, s'), x \mapsto \phi'], s' \vdash e : \phi \,!\, \epsilon}{TE, s \vdash \mathtt{rec}^{s'}\ f(x)\, e : (\phi' \xrightarrow{\epsilon} \phi, s') \,!\, \{(s, s')_C\}}$$

$$(app)\frac{TE, s \vdash e_1 : (\phi' \xrightarrow{\epsilon} \phi, s'') \,!\, \epsilon_1 \qquad TE, s \vdash e_2 : \phi' \,!\, \epsilon_2}{TE, s \vdash e_1 @^{s'} e_2 : \phi \,!\, \epsilon_1 \cup \epsilon_2 \cup \epsilon \cup \{(s, s'')_C\}} \qquad \phi = (\tau, s')$$

$$(let)\frac{TE, s \vdash e_1 : \phi_1 \,!\, \epsilon_1 \qquad TE[x : \phi_1], s \vdash e_2 : \phi_2 \,!\, \epsilon_2}{TE, s \vdash \mathtt{let}^{s'}\ x = e_1\ \mathtt{in}\ e_2 : \phi_2 \,!\, \epsilon_1 \cup \epsilon_2} \qquad \phi_2 = (\tau, s')$$

$$(const)\frac{}{TE, s \vdash c^{s'} : (B, s') \,!\, \{(s, s')_C\}}$$

$$(op)\frac{TE, s \vdash e : (B, s'') \,!\, \epsilon}{TE, s \vdash \mathtt{op}^{s'}(e) : (B, s') \,!\, \epsilon \cup \{(s'', s')_C\}}$$

$$(if)\frac{TE, s \vdash e_1 : (B, s'') \,!\, \epsilon_1 \qquad TE, s'' \vdash e_2 : \phi \,!\, \epsilon_2 \qquad TE, s'' \vdash e_3 : \phi \,!\, \epsilon_3}{TE, s \vdash \mathtt{if}^{s'}\ e_1\ e_2\ e_3 : \phi \,!\, \epsilon_1 \cup \epsilon_2 \cup \epsilon_3} \qquad \phi = (\tau, s')$$

Fig. 2. Static Semantics of PDC

The set *BType* is exactly the set of types for a simply-typed lambda calculus. *BTE* ranges over type environments for this calculus, wheras *TE* ranges over annotated type environments for PDC.

We extend the erasure function to types. Type erasure $|\cdot| : \mathit{AType} \to \mathit{BType}$ maps an annotated type to a bare type (*BType*) by $|(\alpha, s)| = \alpha$, $|(\mathbf{B}, s)| = \mathbf{B}$, and $|(\phi' \xrightarrow{\epsilon} \phi'', s')| = |\phi'| \to |\phi''|$. Technically, we should be using two different sets of type variables, one ranging over *Type* and the other ranging over *BType*, but context will ensure that no ambiguities arise.

Despite the presence of type variables in the type language, there is no polymorphism. The sole purpose of the type variables is to provide a principal typing property to the type system.

2.2 Static Semantics

The static semantics defines the typing judgement $TE, s \vdash e : \phi \,!\, \epsilon$ in Fig. 2 in the style of a type and effect system. The type environment *TE* binds variables to annotated types. The source label s on the left side of the turnstile denotes the source label of the program location that causes e to evaluate, in the sense of a control dependency. For example, the condition expression in a conditional causes one of the "then" or "else" expressions to evaluate. Hence (viz. rule (*if*)), its source label drives the evaluation of the branches. The annotated type $\phi = (\tau, s)$

consists of the real type τ and the source label of the last expression that either created the value of e or passed it on. As mentioned before, the effect ϵ is just a labeled relation, which stands for its least reflexive, transitive closure. It expresses the relation between the values in the environment, the sources labels in e, and the value computed by e.

The rule (*var*) gives rise to one weak control dependency and one value dependency. The value dependency arises due to "copying" the value from the environment (with source s'') to the result area of the expression (with source s'). The control dependency arises from the source s to the left of the turnstile. No parts of the value are examined or constructed, so it is only a weak control dependency.

The rule (*rec*) produces just one control dependency: the allocation of the closure is caused by s. But it also provides the explanation for the pair s'', ϵ on the function arrow. When applying the function, s'' will be the cause for applying the function, *i.e.*, the source of the application context. The effect ϵ is the dependency relation that the function promises to construct. Otherwise, the rule is just the usual rule for a recursive function.

In the function application rule (*app*), the source s is the cause for evaluating e_1, e_2, and also the body of the closure that is computed by e_1. The latter control dependency is created by the atomic dependency $(s, s'')_C$. There is an additional control dependency from the creation of the closure s''' and the result of the whole expression s'. As usual, the application of a function causes the release of its latent effect (the dependency relation ϵ). The relations resulting from the subexpressions are unioned together (and closed under reflexivity and transitivity).

The let expression has no surprises (*let*). However, it is important to see that the evaluation of e_1 can be independent of the evaluation of e_2 if the variable x does not appear in e_1.

The allocation of a constant (*const*) is caused by the context s, so there is a control dependency.

A primitive operation, shown in rule (*op*), is quite similar. The result of the operation is not part of the argument, but still it depends on the input value. Hence, it gives rise to a control dependency and collects the dependency relation from the subexpression.

A conditional expression (*if*) evaluates the condition in the outer context s. The result lives at source s'' and *it* causes the evaluation of either e_1 or e_2.

2.3 Subtyping

The most important bit to understand about this section is that the static semantics is not meant to be prescriptive, in the sense that it limits the applicability of a function. On the contrary, the idea is that every simply-typed program can be completed to a PDC expression (by adding source annotations), which is type correct. In order to obtain this descriptive property, we introduce a notion of subtyping which only works at the level of annotations. This is a typical step in program analysis [6].

First, we define the notion of an effect subset, $\epsilon_1 \sqsubseteq \epsilon_2 \,!\, \epsilon$. It means that in the presence of ϵ, ϵ_1 is a subeffect of ϵ_2.

$$\emptyset \sqsubseteq \epsilon_2 \,!\, \emptyset \qquad \frac{\epsilon_1' \sqsubseteq \epsilon_2 \,!\, \epsilon' \qquad \epsilon_1'' \sqsubseteq \epsilon_2 \,!\, \epsilon''}{(\epsilon_1' \cup \epsilon_2') \sqsubseteq \epsilon_2 \,!\, (\epsilon' \cup \epsilon'')}$$

$$\frac{(s, s', \iota) \in \epsilon_2}{\{(s, s', \iota)\} \sqsubseteq \epsilon_2 \,!\, \emptyset} \qquad \frac{(s, s', \iota) \notin \epsilon_2}{\{(s, s', \iota)\} \sqsubseteq \epsilon_2 \,!\, \{(s, s', \iota)\}}$$

The subtyping judgement is $\vdash \phi \leq \phi \,!\, \epsilon$ with the usual typing rule for subsumption, which includes subeffecting, too:

$$(sub)\frac{TE, s \vdash e : \phi \,!\, \epsilon \qquad \vdash \phi \leq \phi' \,!\, \epsilon' \qquad \epsilon \cup \epsilon' \subseteq \epsilon''}{TE, s \vdash e : \phi' \,!\, \epsilon''}$$

Subtyping is more complicated than usual because it adds to the dependency effect. Subtyping only introduces value dependencies because it only "connects" different source annotations.

$$(sub\text{-}base)\frac{}{\vdash (\mathbf{B}, s) \leq (\mathbf{B}, s') \,!\, \{(s, s')_V\}}$$

$$(sub\text{-}fun)\frac{\vdash \phi_2 \leq \phi_1 \,!\, \epsilon \qquad \vdash \phi_1' \leq \phi_2' \,!\, \epsilon' \qquad \epsilon_1 \sqsubseteq \epsilon_2 \,!\, \epsilon''}{\vdash (\phi_1 \xrightarrow{\epsilon_1} \phi_1', s_1) \leq (\phi_2 \xrightarrow{\epsilon_2} \phi_2', s_2) \,!\, \epsilon \cup \epsilon' \cup \epsilon'' \cup \{(s_1, s_2)_V\}}$$

The best reading of a subtyping judgement $\vdash \phi \leq \phi' \,!\, \epsilon$ is in terms of the subsumption rule, *i.e.*, as a conversion of a value of type ϕ to the expected type ϕ'. The effect ϵ of the subtyping judgement registers the dependencies established by the conversion. In this reading, the rule (*sub-base*) is obvious. The rule (*sub-fun*) has the usual contravariant behavior in the argument part and covariant behavior in the result part and in all other components.

2.4 Basic Properties

The static semantics of PDC is closely tied to the simply-typed lambda calculus. Each expression of the simply-typed lambda calculus can be completed to a typable PDC expression and, vice versa, the erasure of a PDC expression yields a simply-typed lambda expression.

Lemma 1 (Erasure). *If $TE, s \vdash e : \phi \,!\, \epsilon$ then $|TE| \vdash_{st} |e| : |\phi|$ in the system of simple types.*

Lemma 2 (Type Extension). *If $|\phi| = |\phi'|$, then there exists a smallest ϵ so that $\vdash \phi \leq \phi' \,!\, \epsilon$.*

Proof. Case **B**: In this case, $\phi = (\mathbf{B}, s)$ and $\phi' = (\mathbf{B}, s')$. Clearly, $\vdash (\mathbf{B}, s) \leq (\mathbf{B}, s') \,!\, \{(s, s')_V\}$, by definition of $\leq$.

Case $\rightarrow$: In this case, $\phi = (\phi_{1a} \xrightarrow{\epsilon_f} \phi_{1r}, s)$ and $\phi' = (\phi'_{1a} \xrightarrow{\epsilon'_f} \phi'_{1r}, s')$. Furthermore, $|\phi_{1a}| = |\phi'_{1a}|$ as well as $|\phi_{1r}| = |\phi'_{1r}|$. By induction, there exist ϵ_a and ϵ_r so that $\vdash \phi'_{1a} \leq \phi_{1a} \,!\, \epsilon_a$ and $\vdash \phi_{1r} \leq \phi'_{1r} \,!\, \epsilon_r$. Furthermore, define ϵ' by $\epsilon_f \sqsubseteq \epsilon'_f \,!\, \epsilon'$. Setting $\epsilon = \epsilon_a \cup \epsilon_r \cup \epsilon' \cup \{(s, s')_V\}$, we obtain that $\vdash \phi \leq \phi' \,!\, \epsilon$, as claimed.

Lemma 3 (Completion). *If $BTE \vdash_{st} e_0 : \zeta$ then for all s and TE where $|TE| = BTE$ there exist e, ϕ, and ϵ such that $|e| = e_0$, $|\phi| = \zeta$, and $TE, s \vdash e : \phi \,!\, \epsilon$.*

Furthermore, for each ϕ' and ϵ' so that $TE, s \vdash e : \phi' \,!\, \epsilon'$ and $|\phi'| = \zeta$ it holds that $\vdash \phi \leq \phi' \,!\, \epsilon''$ and $\epsilon \cup \epsilon'' \subseteq \epsilon'$.

Lemma 4 (Minimal Type). *Suppose that $BTE \vdash_{st} e_0 : \zeta$ is a principal typing for e in the system of simple types. Then there exist TE, s, e, ϕ, and ϵ with $|TE| = BTE$, $|e| = e_0$, $|\phi| = \zeta$ and $TE, s \vdash e : \phi \,!\, \epsilon$, so that, for all TE', ϕ', e', and ϵ' with $|TE'| = BTE$, $|e'| = e_0$, $|\phi'| = \zeta$ and $TE', s \vdash e' : \phi' \,!\, \epsilon'$, it holds that $\vdash \phi \leq \phi' \,!\, \epsilon''$ and $\epsilon \subseteq \epsilon'$ and $\epsilon'' \subseteq \epsilon'$.*

Finally, a technical result that shows that the value of an expression depends on its control dependency. For this result, we need an assumption about the types in the environment, which is captured by the following definition:

An annotated type is *well-formed* if the judgement ϕ wft is derivable using the rules

$$(\alpha, s) \text{ wft} \qquad (\mathbf{B}, s) \text{ wft} \qquad \frac{\phi' \text{ wft} \quad \phi \text{ wft} \quad (s', s) \in \epsilon \quad \phi = (\tau, s)}{(\phi' \xrightarrow{\epsilon} \phi, s') \text{ wft}}$$

If $TE = \{x_1 : \phi_1, \ldots, x_n : \phi_n\}$ then TE wft holds if ϕ_i wft, for all $1 \leq i \leq n$.

Lemma 5 (Output Types). *Suppose that $TE, s \vdash e : (\tau, s') \,!\, \epsilon$ where TE* wft*. Then $(s, s') \in \epsilon$.*

2.5 Dynamic Semantics

The dynamic semantics of PDC is defined using a big-step operational semantics. The judgement $VE, s \vdash e \Downarrow r$ states that with variable bindings VE and current "cause for the evaluation" s, the expression e evaluates to a return value r, where r is either a value paired with a dependency graph or an error ERR. Values, v, are either base-type constants or closures. Each value is tagged with a source annotation, s, that describes where the value has been created or passed through. The dependency graph tracks the actual dependencies that have occured during computation. Here is the formal definition of return values:

$$\begin{aligned} v &::= c \mid (VE, \lambda x.\, e) \\ w &::= v^s \\ r &::= w, \epsilon \mid ERR \end{aligned}$$

$$(\textit{ev-var})\frac{VE(x) = v^{s''}}{VE, s \vdash x^{s'} \Downarrow v^{s'}, \{(s, s')_W, (s'', s')_V\}}$$

$$(\textit{ev-rec})\frac{}{VE, s \vdash \mathtt{rec}^{s'}\ f(x)\, e \Downarrow (VE \downarrow fv(\mathtt{rec}^{s'}\ f(x)\, e), \mathtt{rec}\ f(x)\, e)^{s'}, \{(s, s')_C\}}$$

$$(\textit{ev-app})\frac{\begin{array}{c} VE, s \vdash e_1 \Downarrow (VE', \mathtt{rec}\ f(x)\, e)^{s''}, \epsilon_1 \\ VE, s \vdash e_2 \Downarrow w_2, \epsilon_2 \end{array} \quad VE'[f \mapsto (VE', \mathtt{rec}\ f(x)\, e)^{s''}, x \mapsto w_2], s'' \vdash e \Downarrow v^{s'''}, \epsilon}{VE, s \vdash e_1 @^{s'} e_2 \Downarrow v^{s'}, \epsilon_1 \cup \epsilon_2 \cup \epsilon \cup \{(s, s'')_C\}}$$

$$(\textit{ev-let})\frac{VE, s \vdash e_1 \Downarrow w_1, \epsilon_1 \qquad VE[x \mapsto w_1], s \vdash e_2 \Downarrow v^{s'''}, \epsilon_2}{VE, s \vdash \mathtt{let}^{s'}\ x = e_1\ \mathtt{in}\ e_2 \Downarrow v^{s'}, \epsilon_1 \cup \epsilon_2}$$

$$(\textit{ev-const})\frac{}{VE, s \vdash c^{s'} \Downarrow c^{s'}, \{(s, s')_C\}}$$

$$(\textit{ev-op})\frac{VE, s \vdash e \Downarrow c^{s''}, \epsilon}{VE, s \vdash \mathtt{op}^{s'}(e) \Downarrow (\mathtt{op}(c))^{s'}, \epsilon \cup \{(s'', s')_C\}}$$

$$(\textit{ev-if-true})\frac{VE, s \vdash e_1 \Downarrow c^{s''}, \epsilon_1 \quad c \neq \mathtt{false} \qquad VE, s'' \vdash e_2 \Downarrow v^{s'''}, \epsilon_2}{VE, s \vdash \mathtt{if}^{s'}\ e_1\ e_2\ e_3 \Downarrow v^{s'}, \epsilon_1 \cup \epsilon_2}$$

$$(\textit{ev-if-false})\frac{VE, s \vdash e_1 \Downarrow \mathtt{false}^{s''}, \epsilon_1 \qquad VE, s'' \vdash e_3 \Downarrow v^{s'''}, \epsilon_3}{VE, s \vdash \mathtt{if}^{s'}\ e_1\ e_2\ e_3 \Downarrow v^{s'}, \epsilon_1 \cup \epsilon_3}$$

Fig. 3. Dynamic Semantics

Figure 3 defines the inference rules for the judgement $VE, s \vdash e \Downarrow r$. Figure 4 shows an excerpt of the error transitions, namely those for the rule (*ev-app*). There are two more for (*ev-let*), two for (*ev-op*), and three for (*ev-if-true*) as well as for (*ev-if-false*). They are constructed in the usual way, so that evaluation propagates errors strictly.

2.6 Type Preservation

To establish a connection between the static semantics and the dynamic semantics, we define a typing relation for values $\vdash_v w : \phi$ and value environments as follows:

$$\vdash_v c^s : (\mathbf{B}, s) \qquad \frac{\vdash_v VE : TE \qquad TE[f : (\phi' \xrightarrow{\epsilon} \phi, s), x : \phi'], s \vdash e : \phi\, !\, \epsilon}{\vdash_v (VE, \mathtt{rec}\ f(x)\, e)^s : (\phi' \xrightarrow{\epsilon} \phi, s)}$$

$$\frac{\vdash_v w_i : \phi_i}{\vdash_v \{x_i : w_i\} : \{x_i : \phi_i\}}$$

$$(\textit{ev-app-err1})\frac{VE, s \vdash e_1 \Downarrow c^{s''}, \epsilon_1}{VE, s \vdash e_1 @^{s'} e_2 \Downarrow ERR}$$

$$(\textit{ev-app-err2})\frac{VE, s \vdash e_1 \Downarrow ERR}{VE, s \vdash e_1 @^{s'} e_2 \Downarrow ERR}$$

$$(\textit{ev-app-err3})\frac{\begin{array}{c} VE, s \vdash e_1 \Downarrow (VE', s'', \texttt{rec}\ f(x)\, e)^{s'''}, \epsilon_1 \\ VE, s \vdash e_2 \Downarrow ERR \end{array}}{VE, s \vdash e_1 @^{s'} e_2 \Downarrow ERR}$$

$$(\textit{ev-app-err4})\frac{\begin{array}{c} VE, s \vdash e_1 \Downarrow (VE', s'', \texttt{rec}\ f(x)\, e)^{s'''}, \epsilon_1 \\ VE, s \vdash e_2 \Downarrow w_2, \epsilon_2 \\ VE'[x \mapsto w_2], s'' \vdash e \Downarrow ERR \end{array}}{VE, s \vdash e_1 @^{s'} e_2 \Downarrow ERR}$$

Fig. 4. Error Transitions (Excerpt)

This enables us to prove the following by induction on the derivation of the evaluation judgement.

Lemma 6 (Type Preservation). *If* $TE, s \vdash e : \phi \,!\, \epsilon$ *and* $VE, s \vdash e \Downarrow w, \epsilon'$ *and* $\vdash_v VE : TE$ *then* $\vdash_v w : \phi$ *and* $\epsilon' \subseteq \epsilon$.

3 Translations

In this section, we demonstrate that the region calculus and a calculus for flow analysis are both images of PDC.

3.1 Region Calculus

We consider a simply-typed variant of the region calculus without region polymorphism and without the letregion construct. Polymorphism is out of the scope of the present work.

Figure 5 summarizes syntax and static semantics of the region calculus, where we take *Region* as a set of region variables. To simplify the translation, we have made subeffecting into a separate rule, rather than including it in the rule for functions (as in [20]).

Suppose now that we are given a derivation for the PDC judgement $TE, s \vdash e : \phi \,!\, \epsilon$. From this we construct a derivation for a corresponding judgement in the region calculus in two steps. In the first step, we extract an equivalence relation on source annotations from the PDC derivation. The equivalence classes of this relation serve as our region variables. In the second step, we translate a PDC expressions to an expression of the region calculus and map the type derivation accordingly. The main step here is the mapping from source annotations to their equivalence classes.

$$
\begin{array}{rcl}
\rho & \in & \mathit{Region} \\
e_r & ::= & x \mid \texttt{rec}\ f(x)\, e_r\ \texttt{at}\ \rho \mid e_r @ e_r \mid \texttt{let}\ x = e_r\ \texttt{in}\ e_r \mid \\
& & c\ \texttt{at}\ \rho \mid \texttt{op}(e_r)\ \texttt{at}\ \rho \mid \texttt{if}\ e_r\ e_r\ e_r
\end{array}
$$

$$
\begin{array}{rcl}
\phi & ::= & (\theta, \rho) \\
\theta & ::= & \alpha \mid \mathbf{B} \mid \phi \xrightarrow{\varepsilon} \phi \\
\varepsilon & \subseteq & \mathit{Region}
\end{array}
$$

$$(\textit{r-var})\ \frac{\mathit{RTE}(x) = \phi}{\mathit{RTE} \vdash_r x : \phi \,!\, \emptyset}$$

$$(\textit{r-rec})\ \frac{\mathit{RTE}[f \mapsto (\phi' \xrightarrow{\varepsilon} \phi, \rho'), x \mapsto \phi'] \vdash_r e : \phi \,!\, \varepsilon}{\mathit{RTE} \vdash_r \texttt{rec}\ f(x)\, e\ \texttt{at}\ \rho' : (\phi' \xrightarrow{\varepsilon} \phi, \rho') \,!\, \{\rho\}}$$

$$(\textit{r-app})\ \frac{\mathit{RTE} \vdash_r e_1 : (\phi' \xrightarrow{\varepsilon} \phi, \rho) \,!\, \varepsilon_1 \qquad \mathit{RTE} \vdash_r e_2 : \phi' \,!\, \varepsilon_2}{\mathit{RTE} \vdash_r e_1 @ e_2 : \phi \,!\, \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon \cup \{\rho\}}$$

$$(\textit{r-let})\ \frac{\mathit{RTE} \vdash_r e_1 : \phi_1 \,!\, \varepsilon_1 \qquad \mathit{RTE}[x : \phi_1] \vdash_r e_2 : \phi_2 \,!\, \varepsilon_2}{\mathit{RTE} \vdash_r \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \phi_2 \,!\, \varepsilon_1 \cup \varepsilon_2}$$

$$(\textit{r-const})\ \frac{}{\mathit{RTE} \vdash_r c\ \texttt{at}\ \rho' : (B, \rho') \,!\, \{\rho'\}}$$

$$(\textit{r-op})\ \frac{\mathit{RTE} \vdash_r e : (B, \rho'') \,!\, \varepsilon}{\mathit{RTE} \vdash_r \texttt{op}(e)\ \texttt{at}\ \rho' : (B, \rho') \,!\, \varepsilon \cup \{\rho'', \rho'\}}$$

$$(\textit{r-if})\ \frac{\mathit{RTE} \vdash_r e_1 : (B, \rho'') \,!\, \varepsilon_1 \qquad \mathit{RTE} \vdash_r e_2 : \phi \,!\, \varepsilon_2 \qquad \mathit{RTE} \vdash_r e_3 : \phi \,!\, \varepsilon_3}{\mathit{RTE} \vdash_r \texttt{if}\ e_1\ e_2\ e_3 : \phi \,!\, \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3}$$

$$(\textit{r-subeff})\ \frac{\mathit{RTE} \vdash_r e : \phi \,!\, \varepsilon \quad \varepsilon \subseteq \varepsilon'}{\mathit{RTE} \vdash_r e : \phi \,!\, \varepsilon'}$$

Fig. 5. Syntax and Static Semantics of the Region Calculus

For the first step, the computation of the equivalence relation, the function α extracts a set of pairs from effects, annotated types, and type environments. The most important part is that *control dependencies are ignored by* α.

$$
\begin{array}{ll}
\alpha(\emptyset) & = \emptyset \\
\alpha(\epsilon \cup \epsilon') & = \alpha(\epsilon) \cup \alpha(\epsilon') \\
\alpha((s, s')_V) & = \{(s, s')\} \\
\alpha((s, s')_W) & = \emptyset \\
\alpha((s, s')_C) & = \emptyset \\
\\
\alpha(\mathbf{B}, s) & = \emptyset \\
\alpha(\phi' \xrightarrow{\epsilon} \phi, s) & = \alpha(\phi') \cup \alpha(\phi) \cup \alpha(\epsilon) \\
\\
\alpha([x_1 : \phi_1, \ldots]) & = \alpha(\phi_1) \cup \ldots
\end{array}
$$

The *extract* of a type derivation is the union of the results of applying α to each occurrence of a type environment, an annotated type, and an effect in every step of the derivation (we sidestep the formal definition, which should be obvious). Then, we define the relation $\equiv \subseteq \mathit{Source} \times \mathit{Source}$ as the smallest equivalence relation containing the extract. We denote the equivalence classes of $\equiv$ by $[s]_\equiv$ or just $[s]$.

In the second step, we translate the parts of an RC type derivation using the function $\beta_\equiv$. For expressions it is defined as follows.

$$\begin{array}{ll}
\beta_\equiv(x^s) & = x \\
\beta_\equiv(\mathtt{rec}^s\ f(x)\,e) & = \mathtt{rec}\ f(x)\,\beta_\equiv(e)\ \mathtt{at}\ [s]_\equiv \\
\beta_\equiv(e_1 @^s e_2) & = \beta_\equiv(e_1) @ \beta_\equiv(e_2) \\
\beta_\equiv(\mathtt{let}^s\ x = e_1\ \mathtt{in}\ e_2) & = \mathtt{let}\ x = \beta_\equiv(e_1)\ \mathtt{in}\ \beta_\equiv(e_2) \\
\beta_\equiv(c^s) & = c\ \mathtt{at}\ [s]_\equiv \\
\beta_\equiv(\mathtt{op}^s(e)) & = \mathtt{op}(e)\ \mathtt{at}\ [s]_\equiv \\
\beta_\equiv(\mathtt{if}^s\ e_1\ e_2\ e_3) & = \mathtt{if}\ \beta_\equiv(e_1)\ \beta_\equiv(e_2)\ \beta_\equiv(e_3)
\end{array}$$

For types and effects, it is defined by

$$\begin{array}{ll}
\beta_\equiv(\mathbf{B}, s) & = (\mathbf{B}, [s]_\equiv) \\
\beta_\equiv(\phi' \xrightarrow{\epsilon} \phi, s) & = (\beta_\equiv(\phi') \xrightarrow{\beta_\equiv(\epsilon)} \beta_\equiv(\phi), [s]_\equiv) \\
\\
\beta_\equiv(\emptyset) & = \emptyset \\
\beta_\equiv(\epsilon \cup \epsilon') & = \beta_\equiv(\epsilon) \cup \beta_\equiv(\epsilon') \\
\beta_\equiv((s, s')_V) & = \emptyset \\
\beta_\equiv((s, s')_W) & = \emptyset \\
\beta_\equiv((s, s')_C) & = \{[s']_\equiv\}
\end{array}$$

With these definition, we can show the following correspondence.

Lemma 7. *Let* $TE, s \vdash e : \phi\ !\ \epsilon$ *and* $\equiv$ *be defined as described above. Then* $\beta_\equiv(TE) \vdash_r \beta_\equiv(e) : \beta_\equiv(\phi)\ !\ \beta_\equiv(\epsilon)$.

Proof. By induction on the derivation of $TE, s \vdash e : \phi\ !\ \epsilon$. Note that uses of the subsumption rule can be mapped to the subeffecting rule (*r-subeff*) because subtyping only gives rise to value dependencies, which are equated by $\equiv$.

3.2 Flow Calculus

We consider a simply-typed flow calculus comparable to 0CFA [7, 15, 18, 19]. Figure 5 summarizes syntax and static semantics of the flow calculus. In this calculus, each subexpression is labeled by a location ℓ. The calculus uses subtyping in the usual way. The type judgement $FTE \vdash_f e_f : (\theta, L)$ means that the value of e_f is constructed and passed through (at most) the locations mentioned in L.

The mapping to PDC is straightforward for the flow calculus. Firstly, it ignores all control dependencies. Next, it maps source annotations to locations

$$
\begin{array}{rcl}
\ell & \in & \mathit{Location} \\
e_f & ::= & x^\ell \mid \mathtt{rec}^\ell\ f(x)\, e_f \mid e_f @^\ell e_f \mid \mathtt{let}^\ell\ x = e_f\ \mathtt{in}\ e_f \mid \\
& & c^\ell \mid \mathtt{op}^\ell(e_f) \mid \mathtt{if}^\ell\ e_f\ e_f\ e_f
\end{array}
$$

$$
\begin{array}{rcl}
L & \subseteq & \mathit{Location} \\
\phi & ::= & (\theta, L) \\
\theta & ::= & \alpha \mid \mathbf{B} \mid \phi \to \phi
\end{array}
$$

$$(\textit{f-var})\ \frac{FTE(x) = (\theta, L)}{FTE \vdash_f x^\ell : (\theta, \{\ell\} \cup L)}$$

$$(\textit{f-rec})\ \frac{FTE[f \mapsto (\phi' \to \phi, L'), x \mapsto \phi'] \vdash_f e : \phi}{FTE \vdash_f \mathtt{rec}^\ell\ f(x)\, e : (\phi' \to \phi, \{\ell\})}$$

$$(\textit{f-app})\ \frac{FTE \vdash_f e_1 : (\phi' \to \phi, L) \qquad FTE \vdash_f e_2 : \phi' \qquad \ell \in L'}{FTE \vdash_f e_1 @^\ell e_2 : \phi}\ \phi = (\theta, L')$$

$$(\textit{f-let})\ \frac{FTE \vdash_f e_1 : \phi_1 \qquad FTE[x : \phi_1] \vdash_f e_2 : \phi_2}{FTE \vdash_f \mathtt{let}^\ell\ x = e_1\ \mathtt{in}\ e_2 : \phi_2}$$

$$(\textit{f-const})\ \frac{}{FTE \vdash_f c^\ell : (B, \{\ell\})}$$

$$(\textit{f-op})\ \frac{FTE \vdash_f e : (B, L)}{FTE \vdash_f \mathtt{op}^\ell(e) : (B, \{\ell\})}$$

$$(\textit{f-if})\ \frac{FTE \vdash_f e_1 : (B, L) \qquad FTE \vdash_f e_2 : \phi \qquad FTE \vdash_f e_3 : \phi \qquad \ell \in L'}{FTE \vdash_f \mathtt{if}^\ell\ e_1\ e_2\ e_3 : \phi}\ \phi = (\theta, L')$$

$$(\textit{f-sub})\ \frac{FTE \vdash_f e : \phi \quad \vdash_f \phi \le \phi'}{FTE \vdash_f e : \phi'}$$

$$\frac{L \subseteq L'}{\vdash_f (\mathbf{B}, L) \le (\mathbf{B}, L')} \qquad \frac{\vdash_f \phi_1' \le \phi_1 \qquad \vdash_f \phi_2 \le \phi_2' \qquad L \subseteq L'}{\vdash_f (\phi_1 \to \phi_2, L) \le (\phi_1' \to \phi_2', L')}$$

Fig. 6. Syntax and Static Semantics of the Flow Calculus

(without lack of generality, we assume the identify mapping because there is always a PDC derivation where all source annotations are distinct). Then, to compute the set L for a type, we take its source annotation and close it under the current dependency graph.

Hence, define

$$
\begin{array}{ll}
F(s, \epsilon) & = \{s' \mid (s', s)_V \in \epsilon\} \\
F((\mathbf{B}, s), \epsilon) & = (\mathbf{B}, F(s, \epsilon)) \\
F((\phi \xrightarrow{\epsilon'} \phi', s), \epsilon) & = (F(\phi, \epsilon) \to F(\phi', \epsilon \cup \epsilon'), F(s, \epsilon)) \\
F([x_1 : \phi_1, \ldots], \epsilon) & = [x_1 : F(\phi_1, \epsilon), \ldots]
\end{array}
$$

and then we can prove that

Lemma 8. *If* $TE, s \vdash e : \phi \,!\, \epsilon$ *then* $F(TE, \emptyset) \vdash_f e : F(\phi, \epsilon)$.

4 Conclusion

We have defined a prototype dependency calculus, PDC, which subsumes important dependency-based program analyses. It is the first calculus that subsumes both the region calculus and other calculi like flow analysis and the SLam calculus for secrecy analysis. Other analyses, in particular binding-time analyses, would also be easy to derive.

We are presenting a number of typed translations into the above calculi. Taken together with the soundness proofs of these calculi, these results give some confidence in the construction of PDC, but ultimately we aim at proving a noninterference result directly for PDC.

On the positive side, the extension to a polymorphic base language and to polymorphic properties seems straightforward. However, it must be expected that such an extension loses the principal typing property enjoyed by PDC.

Another interesting extension would be to cover further effect-based analyses, like side-effects or communication.

References

1. M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In A. Aiken, editor, *Proc. 26th Annual ACM Symposium on Principles of Programming Languages*, pages 147–160, San Antonio, Texas, USA, Jan. 1999. ACM Press. 228, 228, 230
2. M. Abadi, B. Lampson, and J.-J. Lévy. Analysis and caching of dependencies. In R. K. Dybvig, editor, *Proc. International Conference on Functional Programming 1996*, pages 83–91, Philadelphia, PA, May 1996. ACM Press, New York. 230
3. K. Asai. Binding-time analysis for both static and dynamic expressions. In *Static Analysis*, pages 117–133, 1999. 230
4. A. Banerjee, N. Heintze, and J. G. Riecke. Region analysis and the polymorphic lambda calculus. In *Proc. of the 14th Annual IEEE Symposium on Logic in Computer Science*, pages 88–97, Trento, Italy, July 1999. IEEE Computer Society Press. 229, 229, 230
5. L. Birkedal and M. Welinder. Binding-time analysis for Standard-ML. In P. Sestoft and H. Søndergaard, editors, *Proc. ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation PEPM '94*, pages 61–71, Orlando, Fla., June 1994. University of Melbourne, Australia. Technical Report 94/9, Department of Computer Science. 230
6. D. Dussart, F. Henglein, and C. Mossin. Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time. In Mycroft [11], pages 118–136. 230, 233
7. N. Heintze. Control-flow analysis and type systems. In Mycroft [11], pages 189–206. 239
8. N. Heintze and J. G. Riecke. The SLam calculus: Programming with security and integrity. In L. Cardelli, editor, *Proc. 25th Annual ACM Symposium on Principles of Programming Languages*, pages 365–377, San Diego, CA, USA, Jan. 1998. ACM Press. 230
9. F. Henglein and C. Mossin. Polymorphic binding-time analysis. In D. Sannella, editor, *Proceedings of European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 287–301. Springer-Verlag, Apr. 1994. 230

10. E. Moggi. Notions of computations and monads. *Information and Computation*, 93:55–92, 1991. 228
11. A. Mycroft, editor. *Proc. International Static Analysis Symposium, SAS'95*, number 983 in Lecture Notes in Computer Science, Glasgow, Scotland, Sept. 1995. Springer-Verlag. 241, 241
12. F. Nielson and H. R. Nielson. Automatic binding-time analysis for a typed lambda calculus. *Science of Computer Programming*, 10:139–176, 1988. 230
13. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer Verlag, 1999. 230
14. H. R. Nielson, F. Nielson, and T. Amtoft. Polymorphic subtyping for effect analysis: The static semantics. In M. Dam, editor, *Proceedings of the Fifth LOMAPS Workshop*, number 1192 in Lecture Notes in Computer Science. Springer-Verlag, 1997. 230
15. J. Palsberg and P. O'Keefe. A type system equivalent to flow analysis. In *Proc. 22nd Annual ACM Symposium on Principles of Programming Languages*, pages 367–378, San Francisco, CA, Jan. 1995. ACM Press. 239
16. P. Ørbæk and J. Palsberg. Trust in the λ-calculus. *Journal of Functional Programming*, 7(6):557–591, Nov. 1997. 230
17. J.-P. Talpin and P. Jouvelot. The type and effect discipline. *Information and Computation*, 111(2):245–296, 1994. 230
18. Y. M. Tang and P. Jouvelot. Control-flow effects for closure analysis. In *Proceedings of the 2nd Workshop on Static Analysis*, number 81-82 in Bigre Journal, pages 313–321, Bordeaux, France, Oct. 1992. 239
19. Y. M. Tang and P. Jouvelot. Effect systems with subtyping. In W. Scherlis, editor, *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '95*, pages 45–53, La Jolla, CA, June 1995. ACM Press. 230, 239
20. M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997. 229, 230, 237
21. D. Volpano and G. Smith. A type-based approach to program security. In M. Bidoit and M. Dauchet, editors, *TAPSOFT'97: Theory and Practice of Software Development*, number 1214 in Lecture Notes in Computer Science, pages 607–621, Lille, France, Apr. 1997. Springer-Verlag. 230
22. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):1–21, 1996. 230