Short Paper: A Perspective on the Dependency Core Calculus

Maximilian Algehed
Chalmers
Göteborg, Sweden
algehed@chalmers.se

Abstract

This paper presents a simple but equally expressive variant on the terminating fragment of the Dependency Core Calculus (DCC) of Abadi et al. [2]. DCC is a concise and elegant calculus for tracking dependency. The calculus has applications in, among other areas, information flow control, slicing, and binding time analysis. However, in this paper we show that it is possible to replace a core technical device in DCC with an alternative, simpler, formulation. The calculus has a denotational semantics in the same domain as DCC, using which we prove that the two calculi are equivalent. As a proof of concept to show that our calculus provides a simple analysis of dependency we implement it in Haskell, obtaining a simpler implementation compared to previous work [4].

Keywords DCC; IFC; Noninterference; Lambda Caluclus; Haskell

1 Introduction

The Dependency Core Calculus (DCC) of Abadi et al. [2] provides a compact framework for unifying different programming language based dependency analyses, chief among which is information flow control. DCC has been extensively studied in different contexts, including in access control [1], logical relations [3, 7, 13], and encodings into Haskell [4, 13]. However, in this paper we show that the core construct of DCC, the notion of a type being protected with respect to a security level, written $\ell \preceq t$, can be replaced with two simpler constructs. As a consequence of this simplification, we obtain a simple encoding of DCC in Haskell compared to previous work by Algehed and Russo [4].

In this paper we make the following contributions:

- We present a new calculus, SDCC, which aims to provide a simple alternative to DCC, Section 3.
- We prove that SDCC is equivalent to DCC, Section 4.
- We show that SDCC gives rise to a simpler encoding of DCC in Haskell compared to previous work, Section 5.

PLAS'18, October 19, 2018, Toronto, ON, Canada
2018. ACM ISBN 978-1-4503-5993-1/18/10…$15.00
https://doi.org/10.1145/3264820.3264823

All our proofs have been mechanised in the Agda proof assistant [6] and can be found online alongside the Haskell code from Section 5 here: https://tinyurl.com/y9c8zgg4.

2 The Dependency Core Calculus

DCC is a simply typed lambda calculus with products, co-products, and a family of monads T indexed by elements of a lattice $\mathcal{L}$. The type $T_\ell(s)$ denotes data of type s which is accessible only to observers who have the privilege of being allowed to observe data at or above ℓ in the lattice.

Let us review the core concepts of DCC and its mechanisms for data isolation. Following previous work [3, 7, 13], we consider only the terminating fragment of the calculus. The types are generated by the simple grammar below:

$$t ::= \texttt{unit} \mid t \times t \mid t + t \mid t \rightarrow t \mid T_\ell(t)$$

Where $\ell \in \mathcal{L}$ is a security level drawn from the lattice $\mathcal{L}$. As noted previously, the language is simply typed and contains the one element type `unit`, products $t \times t$, co-products $t + t$, functions $t \rightarrow t$, and the T_ℓ family. The syntax of DCC is generated by an equally simple grammar:

$$e ::= x \mid e\ e \mid \lambda x.\ e \mid \texttt{<>} \mid (e, e) \mid \pi_i\ e \mid \iota_i\ e \mid$$
$$\texttt{case}\ e\ e\ e \mid \eta_\ell\ e \mid \texttt{bind}\ x = e\ \texttt{in}\ e$$

Where π_i and ι_i denote the standard projections and injections into and out of products and co-products respectively and i ranges over $\{0, 1\}$. The <> construct denotes the only element of the `unit` type. Apart from the standard constructs, the syntax contains two noteworthy operators, η_ℓ and `bind`. The η_ℓ construct makes an expression secret at level ℓ whereas `bind` is used to perform computation over secrets. Without getting too ahead of ourselves we can reveal that the x in $\texttt{bind}\ x = e\ \texttt{in}\ e'$ binds the secret value of e in the expression e'. Naturally, we wish to enforce some restrictions on how this `bind` primitive can be used in order to make sure that, even for programs using `bind`, secret things stay secret. The restrictions in question are enforced by the DCC type system, the natural place to look next.

In the interest of brevity we will focus on typing the primitives unique to DCC and refer the reader to the original paper or one of many standard references on the subject, e.g. Pierce [12], for details on the rest. The typing rule for η_ℓ is not very surprising:

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash \eta_\ell\ e : T_\ell(t)}$$

All this construct is intended to do is to wrap e at security level ℓ. Typing `bind` is more interesting:

$$\frac{\Gamma \vdash e : T_\ell(s) \quad \Gamma, x : s \vdash e' : t \quad \ell \preceq t}{\Gamma \vdash \texttt{bind}\ x = e\ \texttt{in}\ e' : t}$$

The most noteworthy part of this rule is the $\ell \preceq t$ antecedent. To really understand this rule, we need to make sure we understand this requirement. The $(\preceq)$ relation is defined inductively over the type t using the rules below:

$$1.\ \texttt{protect}_T \frac{\ell \sqsubseteq \ell'}{\ell \preceq T_{\ell'}(s)} \qquad 2.\ \texttt{protect}_{T'} \frac{\ell \preceq s}{\ell \preceq T_{\ell'}(s)}$$

$$3.\ \texttt{protect}_\times \frac{\ell \preceq s \quad \ell \preceq t}{\ell \preceq s \times t} \qquad 4.\ \texttt{protect}_\to \frac{\ell \preceq t}{\ell \preceq s \to t}$$

The first rule in this four-part definition tells us that moving up the lattice preserves secrets, $\ell \preceq T_{\ell'}(s)$ as long as $\ell \sqsubseteq \ell'$. The second rule is saying that wrapping something in a $T_{\ell'}$ can not make it less secret. The third rule says that the structure of a product can not leak any information if the two types in the product do not leak. The intuition for why this makes sense is that given a value of type $T_\ell(s \times t)$ we know that it must contain precisely two secrets, one of type s and one of type t, likewise $T_\ell(s) \times T_\ell(t)$ also contains precisely two secrets. Finally, functions producing secret things are also secret-preserving. Most notable in this definition is that there is no rule for co-products. The reason for this is that the structure of the co-product is enough to leak information. Whether or not the value is an ι_0 or ι_1 is enough to leak one bit of information.

Having established the syntax and type system of DCC, we can turn to the issue of semantics. We adapt here the semantics described by Abadi et al. [2] in order to fit into the terminating setting. The terminating fragment of DCC has a denotational semantics in the category of sets with families of relations. We have a category $\mathcal{DC}$ where:

- An object A is a set $|A|$ and a family of relations $R_{A,\ell} \subseteq |A| \times |A|$, one for every $\ell \in \mathcal{L}$.
- A morphism $f : A \to B$ is a function from $|A|$ to $|B|$ such that if $(x, y) \in R_{A,\ell}$ then $(f(x), f(y)) \in R_{B,\ell}$.

The role of the family of relations $R_{A,\ell}$ is to establish a notion of observer-sensitive equivalence, two values in $|A|$ appear equal to an observer at level ℓ if they are related at the relation $R_{A,\ell}$. The category $\mathcal{DC}$ is bicartesian closed and forms a model of DCC.

The denotational semantics of DCC in $\mathcal{DC}$ are mostly standard, we use the notation $|t|$ and $|\Gamma|$ to refer to the translation of types and contexts, noting that the only special case in the translation is that of the type $T_\ell(s)$, which requires that we fix an observer level $\hat{\ell} \in \mathcal{L}$, corresponding to the level of the attacker. Having established this, the definition of $|T_\ell(s)|$ is straightforward:

$$|T_\ell(s)| = |s|$$

$$R_{T_\ell(s),\ell} = \begin{cases} R_{s,\ell} & \text{if } \ell \sqsubseteq \hat{\ell} \\ |s| \times |s| & \text{otherwise} \end{cases}$$

Where $\hat{\ell} \in \mathcal{L}$ is the observer level, information labeled above $\hat{\ell}$ is opaque to an attacker. A typing judgement $\Gamma \vdash e : t$ corresponds to a morphism (relation-preserving function) $[\![e]\!] : |\Gamma| \to |t|$ in $\mathcal{DC}$. The correspondence is standard for most of the primitives and can be found in one of the standard references, for example Pierce [11]. The parts that are non-standard are the translation of η_ℓ and `bind`, given here in the form of concrete functions on sets rather than as abstract morphisms in the category:

$$[\![\eta_\ell\ e]\!](\Gamma) = [\![e]\!](\Gamma)$$
$$[\![\texttt{bind}\ x = e\ \texttt{in}\ e']\!](\Gamma) = [\![e']\!](\Gamma, [\![e']\!](\Gamma))$$

Note that, because e' is typed in the environment $\Gamma, x : s$ we need to pair the context with $[\![e]\!](\Gamma)$ before passing to $[\![e']\!]$.

3 SDCC, A Simpler Core

As can be seen in Section 2 the $\ell \preceq s$ relation is at the core of DCC. It is what makes the whole calculus tick and the key realisation which enables not just the calculus itself, but all results which build on it. However, we are going to remove it in favour of two different, equally expressive, operators.

While important, this relation causes some annoying complications when trying to work with DCC in other settings, both in theory and in practise. In the domain of theory, one example is Bowman and Ahmed [7] who translate DCC into System F_ω and are forced in their translation to introduce a non-trivial amount of technical machinery to deal with the `bind` construct and the $\ell \preceq s$ antecedent. Likewise, in the more practically minded domain, Algehed and Russo [4] are forced to use non-trivial features of the Haskell type system when encoding DCC in Haskell.

The language we propose as an alternative to DCC has the same plethora of types as the original calculus, but the syntax is somewhat different:

$$\begin{aligned} e ::= {} & x \mid e\ e \mid \lambda x.\ e \mid \texttt{<>} \mid (e, e) \mid \pi_i\ e \mid \iota_i\ e \mid \\ & \texttt{case}\ e\ e\ e \mid \eta_\ell\ e \mid \mu\ e \mid \texttt{map}\ e\ e \mid \texttt{up}\ e \mid \texttt{com}\ e \end{aligned}$$

Note some of the differences compared with DCC, the `bind` construct is gone and in its place are four new constructs, μ, `map`, `up`, and `com`. On the surface this seems like we have removed one primitive to add the complication of many more. However, this is not quite the case as we will see when we start to examine the typing rules of our new combinators. The μ and `map` constructs are standard forms to express monadic computations. The first, sometimes called "join", collapses multple T_ℓ wrappers into a single T_ℓ wrapper, one level at a time:

$$\frac{\Gamma \vdash e : T_\ell(T_\ell(s))}{\Gamma \vdash \mu\ e : T_\ell(s)}$$

The `map` primitive lifts computations on secret values to the level of the secret, staying at the same security level:

$$\frac{\Gamma \vdash f : s \to t \quad \Gamma \vdash e : T_\ell(s)}{\Gamma \vdash \mathtt{map}\ f\ e : T_\ell(t)}$$

It is important to note that the choice of μ and `map` is mostly stylistic, we could well have chosen the alternative `(>>=)` operator found in the functional programming literature as well.

Next we consider the non-standard constructs which we include in our language, `up` and `com`. These two primitives encode two fundamental insights behind the T monad family in DCC. Firstly, `up` says that we can always make something more secret by moving *up* the security lattice:

$$\frac{\Gamma \vdash e : T_\ell(s) \quad \ell \sqsubseteq \ell'}{\Gamma \vdash \mathtt{up}\ e : T_{\ell'}(s)}$$

Viewing the `up` primitive in terms of DCC, it is analogous to the $\mathtt{protect}_T$ rule in the definition of $\ell \preceq s$. The `com` primitive is slightly more subtle:

$$\frac{\Gamma \vdash e : T_\ell(T_{\ell'}(s))}{\Gamma \vdash \mathtt{com}\ e : T_{\ell'}(T_\ell(s))}$$

What this says is that we can *commute* any two T_ℓ and $T_{\ell'}$ wrappers. The intuition for why this is secure is that in order to observe data of type $T_\ell(T_{\ell'}(s))$ we need to be able to observe data above or at both ℓ and ℓ', commuting the wrappers does not change this. With the operators in SDCC established we turn to the issue semantics.

The semantic domain of SDCC is the same as for DCC, with the semantics of μ and `map` being precisely as expected:

$$[\![\mu\ e]\!](\Gamma) = [\![e]\!](\Gamma)$$
$$[\![\mathtt{map}\ f\ e]\!](\Gamma) = [\![f]\!](\Gamma)([\![e]\!](\Gamma))$$

The reason for this simplicity is precisely because of the meaning of T_ℓ, $|T_\ell(s)| = |s|$, no fancy definitions are necessary. The same goes for the translation of `up` and `com`:

$$[\![\mathtt{up}\ e]\!](\Gamma) = [\![e]\!](\Gamma)$$
$$[\![\mathtt{com}\ e]\!](\Gamma) = [\![e]\!](\Gamma)$$

Thinking about these operators computationally we see that they do not actually perform much computation other than "inspecting" η_ℓ wrappers around data. We will explore the computational perspective on SDCC in Section 5 when we give a Haskell encoding of the calculus.

4 An Equally Expressive Calculus

The first thing to note about this calculus compared with DCC is that it is simpler, we have gotten rid of the $(\preceq)$ relation and boiled it down to two necessary primitives `up` and `com`. The obvious next question to ask is if this language is really as expressive as DCC? To see that this is indeed the case we are going to show that every SDCC term corresponds to a DCC term with the same semantics and vice-verse. We will use the notation $\Gamma \vdash e : t \hookrightarrow e'$ for the translation of a term e and its typing derivation in one language to the other. The direction of the translation will be clear from the context. The interesting part of the translation is going to be in the parts that differ between SDCC and DCC, the translation of all other constructs are homomorphic.

For the SDCC to DCC direction, we observe that the `bind` of DCC can be specialised to the monadic bind operation, sometimes written (>>=), which is known to be able to express the μ and `map` primitives. We therefore focus on the two interesting constructions, `up` and `com`. For `up` we take:

$$\frac{\Gamma \vdash e : T_\ell(s) \hookrightarrow e'}{\Gamma \vdash \mathtt{up}\ e : T_{\ell'}(s) \hookrightarrow \mathtt{bind}\ x = e'\ \mathtt{in}\ \eta_{\ell'}\ x}$$

All this definition says is that `up` moves up the security lattice without changing the value of the secret, following the denotational semantics of DCC and SDCC closely. The translation of `com` commutes the two monads without changing the underlying value:

$$\frac{\Gamma \vdash e : T_\ell(T_{\ell'}(s)) \hookrightarrow e'}{\Gamma \vdash \mathtt{com}\ e : T_{\ell'}(T_\ell(s)) \hookrightarrow \begin{array}{l}\mathtt{bind}\ v = e'\ \mathtt{in} \\ \mathtt{bind}\ x = v\ \mathtt{in} \\ \eta_{\ell'}\ (\eta_\ell\ x)\end{array}}$$

The first thing we have to prove about the translation is that it is statically correct. If e is of type t in SDCC and $e \hookrightarrow e'$ holds, then e' should be of the same type in DCC.

Theorem 4.1 (Static Correctness 1). *If* $\Gamma \vdash_{\mathsf{SDCC}} e : t \hookrightarrow e'$, *then* $\Gamma \vdash_{\mathsf{DCC}} e' : t$.

Proof. By induction on the derivation of $\Gamma \vdash_{\mathsf{SDCC}} e : t \hookrightarrow e'$. The interesting cases are the ones for `up` and `com`.

_ For `up` we need to construct the proof that $\ell \preceq T_{\ell'}(s)$ for some s, which we do using the $\ell \sqsubseteq \ell'$ antecedent to the typing derivation and the $\mathtt{protect}_T$ rule in the definition of $\preceq$.

_ For `com` the situation is similar, requiring two proofs, one of $\ell \preceq T_{\ell'}(T_\ell(s))$ and one of $\ell' \preceq T_{\ell'}(T_\ell(s))$ The first uses the $\mathtt{protect}_{T'}$ and $\mathtt{protect}_T$ rules along with the reflexivity of $\sqsubseteq$, the other uses just $\mathtt{protect}_T$ and reflexivity of $\sqsubseteq$.

All other cases follow by induction and simple first-order reasoning. □

Having established that we can take well-typed terms to well-typed terms, we also want to ensure that we have not altered the semantics of programs. Intuitively, this theorem should trivially hold, up, com, and η_ℓ do not do anything to their arguments and neither does bind other than applying the function which arises from the variable binding.

Theorem 4.2 (Semantic Correctness 1). *If* $\Gamma \vdash_{\mathsf{SDCC}} e : t \hookrightarrow e'$, *then* $[\![e]\!] \equiv [\![e']\!]$.

Proof. By induction on the derivation of $\Gamma \vdash_{\mathsf{SDCC}} e : t \hookrightarrow e'$, the cases for up and com are discharged using standard equational reasoning, applying the lemma $\mathsf{ap} \circ \langle \mathsf{cur}(\pi_1), e\rangle = e$ where necessary. □

With the SDCC to DCC translation done, we can move on to the more interesting exercise of translating DCC into SDCC. The interesting case in the translation is that of bind. In this case, we are going to make direct use of the $\ell \preceq s$ proof in the typing derivation. We want to construct $\Gamma \vdash \mathsf{bind}\ x = e\ \mathsf{in}\ e' : s \hookrightarrow e''$ such that $\Gamma \vdash e'' : s$. We will do this by using an auxiliary translation bind^*:

$$\frac{\Gamma \vdash e_0 : T_\ell(t) \hookrightarrow e_0' \quad \Gamma, x : t \vdash e_1 \hookrightarrow e_1' \qquad \Gamma \vdash \mathsf{bind}^*\ (\ell \preceq s)\ e_0'\ (\lambda x.\ e_1') \hookrightarrow e}{\Gamma \vdash \mathsf{bind}\ x = e_0\ \mathsf{in}\ e_1 : s \hookrightarrow e}$$

The idea of the $\mathsf{bind}^*\ (\ell \preceq s) \ldots \hookrightarrow e$ judgement is to "implement" bind in the SDCC calculus. It is defined inductively and follows the structure of the $\ell \preceq s$ proof. The four cases in bind^* are therefore the exact four cases in the definition of $(\preceq)$. We begin with the analogue of $\mathsf{protect}_T$:

$$\frac{\Gamma \vdash e : T_\ell(s) \quad \Gamma \vdash e' : s \to T_{\ell'}(t) \quad \ell \sqsubseteq \ell'}{\Gamma \vdash \mathsf{bind}^*\ (\ell \preceq T_{\ell'}(t))\ e\ e' \hookrightarrow \mu\ (\mathsf{up}\ (\mathsf{map}\ e'\ e))}$$

Here we see how up plays the role of replacing the $\mathsf{protect}_T$ rule, it takes an expression of type $T_\ell(T_{\ell'}(t))$ to $T_{\ell'}(T_{\ell'}(s))$ and μ does the rest. Next, we tackle the other interesting case, $\mathsf{protect}_{T'}$:

$$\frac{\Gamma \vdash e : T_\ell(s) \quad \Gamma \vdash e' : s \to T_{\ell'}(t) \qquad \Gamma, x : T_\ell(t) \vdash \mathsf{bind}^*\ (\ell \preceq t)\ x\ (\lambda y.\ y) \hookrightarrow e''}{\Gamma \vdash \mathsf{bind}^*\ (\ell \preceq T_{\ell'}(t))\ e\ e' \hookrightarrow \mathsf{map}\ (\lambda x.\ e'')\ (\mathsf{com}\ (\mathsf{map}\ e'\ e))}$$

Here we see how com helps us replace $\mathsf{protect}_{T'}$, com takes $\mathsf{map}\ e'\ e : T_\ell(T_{\ell'}(s))$ to something of type $T_{\ell'}(T_\ell(s))$, meaning we can make use of the proof that $\ell \preceq t$ to get rid of the inner T_ℓ wrapper. In translating the case for products, $\mathsf{protect}_\times$ we realise that we can construct an expression of type $T_\ell(s{\times}t)$ and map both π_0 and π_1 over it, then recursively applying bind^*:

$$\frac{\Gamma \vdash e : T_\ell(a) \quad \Gamma \vdash e' : a \to (s \times t) \qquad \Gamma \vdash \mathsf{bind}^*\ (\ell \preceq s)\ e\ (\lambda x. \pi_0\ (e'\ x)) \hookrightarrow l \qquad \Gamma \vdash \mathsf{bind}^*\ (\ell \preceq t)\ e\ (\lambda x. \pi_1\ (e'\ x)) \hookrightarrow r}{\Gamma \vdash \mathsf{bind}^*\ (\ell \preceq s \times t)\ e\ e' \hookrightarrow (l, r)}$$

Similarly, the idea behind the translation of functions is that we can move an abstraction to the top level:

$$\frac{\Gamma \vdash e : T_\ell(a) \quad \Gamma \vdash e' : a \to (s \to t) \qquad \Gamma, x : s \vdash \mathsf{bind}^*\ (\ell \preceq t)\ e\ (\lambda y.\ e'\ y\ x) \hookrightarrow e''}{\Gamma \vdash \mathsf{bind}^*\ (\ell \preceq s \to t)\ e\ e' \hookrightarrow \lambda x.\ e''}$$

Note that all this part of the translation does is that it introduces an abstraction and pushes the bind in under it.

With the translation established, we can turn to the issue of correctness. We begin by establishing that the translation is indeed type-preserving:

Theorem 4.3 (Static Correctness 2). *If* $\Gamma \vdash_{\mathsf{DCC}} e : t \hookrightarrow e'$, *then* $\Gamma \vdash_{\mathsf{SDCC}} e' : t$.

Proof. Straightforward by induction on the derivation of $\Gamma \vdash_{\mathsf{DCC}} e : t \hookrightarrow e'$, the case for bind is discharged by an auxiliary static correctness lemma for the $\mathsf{bind}^* \ldots \hookrightarrow \ldots$ relation, which is proven by induction over the $\ell \preceq s$ derivation. □

Next we move on to establishing the semantic correctness of the translation. In order to establish this theorem we need a lemma for discharging the bind^* case.

Lemma 4.4 (Semantic Correctness of bind^*). *If*

$$\Gamma \vdash \mathsf{bind}^*\ (\ell \preceq s)\ e'\ (\lambda x.\ e_1') \hookrightarrow e$$

then $[\![e]\!] \equiv \mathsf{ap} \circ \langle \mathsf{cur}([\![e_1']\!]), [\![e']\!]\rangle$.

Proof. By induction on the derivation $\Gamma \vdash \mathsf{bind}^* \ldots \hookrightarrow e$, all cases have proofs by equational reasoning. The only crux is the implicit application of weakening to e' in the rules for products and functions, this is discharged by an appeal to the underlying set-theoretic semantics. □

Having established the necessary lemma we can move on to the semantic correctness theorem.

Theorem 4.5 (Semantic Correctness 2). *If* $\Gamma \vdash e : t \hookrightarrow e'$, *then* $[\![e]\!] \equiv [\![e']\!]$.

Proof. By induction on the derivation of $\Gamma \vdash e : t \hookrightarrow e'$, applying Lemma 4.4 in the case for bind. □

With the theorem above established, the translation is complete. We now know that DCC and SDCC are indeed equally expressive. We now know that we can get rid of the complicated $(\preceq)$ relation in favour of the simpler up and com primitives.

5 A Simple Haskell Implementation

Algehed and Russo [4] give an encoding of DCC in the Haskell programming language [10] which makes use of advanced features of the Haskell type system, including type-level functions and different universes of types. Here we show how to simplify this encoding by instead encoding SDCC in Haskell. In the interest of keeping the exposition as simple as possible we will consider only the two point lattice $\{H, L\}$ where $H \not\sqsubseteq L$ is the only disallowed flow.

We encapsulate everything in a module we call `SDCC`, which will provide the trusted computing base of the encoding:

```
module SDCC ( T, eta, mu, map, up
            , com, CanFlowTo, H, L) where
```

where `CanFlowTo` denotes ($\sqsubseteq$). Next we move on to the basic primitives:

```
data T l a = Hide a

eta :: a -> T l a
eta a = Hide a

map :: (a -> b) -> T l a -> T l b
map f (Hide a) = Hide (f a)

mu :: T l (T l a) -> T l a
mu (Hide a) = a
```

Following Algehed and Russo we simply implement `T` as an identity monad from which the other definitions follow. Note that while these definitions make heavy use of the `Hide` constructor. Access to this constructor is not exported from the module. This is crucial in order to disallow an attacker from accessing the contents of a secret at level `T H` in a low context. Had we exported `Hide` an attacker could simply pattern-match on the constructor and leak arbitrary information. The definitions of the other two primitives, `up` and `com`, are as simple as the ones above:

```
up :: CanFlowTo l l' => T l a -> T l' a
up (Hide a) = Hide a

com :: T l (T l' a) -> T l' (T l a)
com (Hide (Hide a)) = Hide (Hide a)
```

The only thing to note about these two primitives is that `up` has a constraint saying `CanFlowTo l l'`, this corresponds to the $\ell \sqsubseteq \ell'$ antecedent in the typing rule. Next we move on to the encoding of our lattice. The elements of the lattice are represented simply as empty types `data H` and `data L`. We give these types structure by defining the `CanFlowTo` relation as a type class. However, instead of encoding the relation directly in the `CanFlowTo` type class we are going to take a somewhat roundabout approach proposed by Buiras et al. [8], guarding `CanFlowTo` by another type class `Flows`:

```
class Flows l l' => CanFlowTo l l'
instance CanFlowTo L L
instance CanFlowTo L H
instance CanFlowTo H H
```

Where the instances of `Flows` are the same as for `CanFlowTo`. The reason this definition is necessary is that as soon as we export the name of a type class from a module, any client of that module can define new instances for it. This means that if we defined `CanFlowTo` without `Flows`, a client of the module could simply provide an instance of `CanFlowTo H L`. This would permit arbitrary information to flow from `H` to `L` through the `up` primitive. Using the construction above side-steps this issue by not exporting `Flows`, thereby making it impossible to add instances beyond what is already there.

With this definition of the lattice constructs our encoding is complete. The only extension to the Haskell base language we require in order to make this compile is *MultiParamTypeClasses*, which is what lets us define the `CanFlowTo` and `Flows` type classes. To conclude this section, what we have obtained is a simple encoding of SDCC in base Haskell with the addition of one minor language extension.

6 Related and Future Work

Many papers have been written on the topic of translating DCC into other calculi. These include Bowman and Ahmed [7] who translate DCC into System F_ω, and Aguirre et al. [3] who translate DCC into a higher-order relational logic. None of these prior translations have translated DCC into a simpler language, rather they focus on showing the power of other languages.

Algehed and Russo [4] also give an embedding of DCC in Haskell. While their translation is direct, it makes use of some advanced features of the Haskell type system necessary to conveniently work with DCC in Haskell. In contrast to this approach this paper shows that starting from a simpler language means giving a simpler encoding which does not make use of any advanced features of the type system. We believe this is evidence for the simplicity of our calculus.

We have yet to fully explore the consequences of our simplification, we believe that translations into other languages, like those of Bowman and Ahmed [7] and Aguirre et al. [3] can either be simplified or their soundness theorems stated more directly using this language instead. Furthermore the SDCC calculus makes explicit the operations on the index of T which are important in the information-flow control setting. This, we believe, could be used to unify DCC with other effect-calculi based on the idea of graded monads and monads indexed by arbitrary algebraic structures like a monoid or a group [5, 9, 14].

7 Conclusions

In this short paper we have proposed a new calculus, SDCC, which aims to provide an alternative to the Dependency Core Calculus of Abadi et al. [2]. We have shown that the terminating fragments of the two calculi are equivalent by translating programs between the two calculi while preserving their denotational semantics. As a corollary of our simplification we also get a simple encoding of a language equivalent to DCC in our calculus, another contribution.

References

[1] Martín Abadi. 2007. Access control in a core calculus of dependency. *Electronic Notes in Theoretical Computer Science* 172 (2007), 5–31.

[2] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G Riecke. 1999. A core calculus of dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 147–160.

[3] Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Pierre-Yves Strub. 2017. A relational logic for higher-order programs. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 21.

[4] Maximilian Algehed and Alejandro Russo. 2017. Encoding DCC in Haskell. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*. ACM, 77–89.

[5] Robert Atkey. 2009. Parameterised notions of computation. *Journal of functional programming* 19, 3-4 (2009), 335–376.

[6] Ana Bove, Peter Dybjer, and Ulf Norell. 2009. A brief overview of Agda–a functional language with dependent types. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 73–78.

[7] William J Bowman and Amal Ahmed. 2015. Noninterference for free. *ACM SIGPLAN Notices* 50, 9 (2015), 101–113.

[8] Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo. 2015. HLIO: Mixing static and dynamic typing for information-flow control in Haskell. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 289–301.

[9] Marco Gaboardi, Shin-ya Katsumata, Dominic Orchard, Flavien Breuvart, and Tarmo Uustalu. 2016. Combining effects and coeffects via grading. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 476–489.

[10] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, et al. 1992. Report on the programming language Haskell: a non-strict, purely functional language version 1.2. *ACM SigPlan notices* 27, 5 (1992), 1–164.

[11] Benjamin C Pierce. 1991. *Basic category theory for computer scientists*. MIT press.

[12] Benjamin C Pierce. 2002. *Types and programming languages*. MIT press.

[13] Stephen Tse and Steve Zdancewic. 2004. Translating dependency into parametricity. In *ACM SIGPLAN Notices*, Vol. 39. ACM, 115–125.

[14] Philip Wadler. 1998. The marriage of effects and monads. In *ACM SIGPLAN Notices*, Vol. 34. ACM, 63–74.