Towards a Formal Language for Privacy Options

Stefan Berthold

HAL Id: hal-01559467

https://hal.inria.fr/hal-01559467

Submitted on 10 Jul 2017

Towards a Formal Language for Privacy Options

Stefan Berthold

Karlstad University
651 88 Karlstad, Sweden
`stefan.berthold@kau.se`

Abstract Describing complex ideas requires clear and concise languages. Many domains have developed their specific languages for describing problem instances independently from solutions and thus making a reference model of the domain available to solution developers. We contribute to the zoo of domain-specific languages within the privacy area with a language for describing data disclosure and usage contracts. Our Privacy Options Language is defined by a small number of primitives which can be composed to describe complex contracts. Our major contribution is the notion of contract rights which is based on the notion of obligations and therefore establishes both concepts as first-class language citizens in a new coherent model for privacy policy languages. Our model overcomes the traditional separation of the right and obligation notions known from access control based policy language approaches. We compare our language to the PrimeLife Policy Language and provide rules for the translation from our language to PrimeLife's language. Then, we present a canonical form of our contracts. It is used to ensure that contracts with equal semantics have the same syntax, thus eliminating the possibility of a covert channel in the syntax revealing information about the originator. Finally, we show different ways of how to extend our language.

1 Introduction

Informational self-determination means that individuals have control over their personal data. Control in this context particularly means that individuals can access, correct, and possibly delete their personal data stored in, e. g., customer databases of companies or institutions, and object to disclosure of their data to third parties. Control, however, can also mean the possibility to agree on data disclosure in exchange for a reasonable compensation.

An interesting idea has been proposed by Laudon [1]: in his scenario, a regulated "national information market" is the only place where personal information is traded between institutions, their customers, and third parties. The market thus becomes a single point of control where individuals and institutions can exercise control and claim their rights on an equal level. This approach is particularly interesting since it uses the strengths of three different fields in order to obtain privacy: regulation for creating a safe environment where crime can be sued after the fact, technology for authorisation of data usage, and a market

for determining fair prices for the data. Laudon assumes that customer data is stored by companies and institutions which may wish to use it for different purposes later. Taylor [2] names individual pricing and targeted advertising, among others, as reasons for letting stored customer (and consumer) data become highly valuable for companies and institutions. In order to keep in line with the legislation, institutions that intend to re-use data for another purpose have to return to the individual who will have to give consent and will claim compensation for the new usage of his data. Again, an ideal market would determine a fair price for the data.

These scenarios share the notion that personal data may be used long after its disclosure. Taking the view of Laudon [1], individuals should receive a compensation depending on the benefit an institution gains by using the individual's personal data. Determining the value of a fair compensation is, however, not necessarily easy [3]. In particular, if we cannot assume that individuals can *directly* control the use of their data after the disclosure, they have to anticipate the consequences of the data disclosure at the time of the disclosure. Part of this problem has been discussed by Berthold and Böhme in [4].

An important means for anticipating the consequences of data disclosure is an unambiguous language for describing all rights and obligations connected to the data disclosure. Such a language can be used by both, the individual that discloses data and the institution that receives the data. The individual will use the language for determining clear conditions of the data usage, e. g., limiting it to a specific purpose and possibly to a time frame. The institution can use the language as a management reference that determines under which conditions the data may be used for specific purposes and when data may not be used anymore. Given an appropriate legal framework (e. g., the one suggested in [1]), statements of this language would even form contracts with legal rights and obligations for the individual and the institution. This idea has been extensively explored by Laudon [1] and backed up by the results of Hann et al. [5]. In more recent work, Berthold and Böhme [4] elaborate on the similarities of contracts and data disclosure. The concrete specification of a suitable contract language for this purpose, however, is to the best of our knowledge still an open research question.

In order to fill this gap, we will turn our attention to specify a formal language for data disclosure contracts. The following requirements shall be met by our language (hence referred to as POL):

- expressive power to capture the notion of Privacy Options [4]
- expressive power to capture existing approaches such as "Sticky Policies" as used in the PrimeLife Policy Language (PPL) [6]
- easy extensibility and scalability (of syntax and semantics)

Our approach is driven by the privacy measurement perspective which is mainly presented in [4] while the notion of Sticky Policies in PrimeLife's PPL [6] has mainly developed with the (identity) management perspective in mind. Both approaches thus target similar goals. The major difference between both approaches is that PPL is an access control language, thus specifying which ac-

cesses are allowed and which not, and POL is a contract language, thus specifying commitments to data disclosure and usage. Moreover, the strict separation on a semantical as well as a syntactical level between rights and obligations known from PPL does not exist in our language. POL is rather building the notion of rights on the notion of obligations.

The rest of this paper is structured as follows. Section 2 defines the language primitives and demonstrates how these primitives can be combined to contracts. In Section 3, we define an operational semantics for POL which describes how the language can be translated into specific data management actions in order to satisfy the contract. In Section 4, we discuss how and to which extent it is possible to translate POL to PPL. A dialect of POL, POL$^-$, is defined and translation rules are given. In Section 5, we present a canonical form of POL contracts. An abstract rewriting system is defined which translates any POL contract to its canonical form. In Section 6, we show how to extend POL in different ways. Section 7 concludes this paper.

2 Privacy Option Language

We are indeed not the first specifying a formal language. Tool support for language specification has been grown large in recent years and so has the number of domain-specific languages. In order to avoid redundancy with previous work, we build upon an existing language for describing financial contracts, proposed by Peyton Jones and Eber [7], and adapt it for our purposes. The domain of financial contracts turns out to be quite similar to the one of Privacy Options [4]. Thus, we refer to our language as Privacy Option Language (POL).

Like the language in [7], our language consists of a small number of primitives[1] with basic semantics. For instance, a contract c_1 that settles the immediate usage of personal data a_1 for purpose p_1 can be written as $c_1 = \texttt{data}\ a_1\ p_1$. This describes the rights and obligations of one contract party and let us assume that this is the institution that receives the data. The contract c_2 written by the individual who discloses the data is the 'negation' of c_1,[2] thus, we can write it as a function application with another primitive, $\texttt{give}$, $c_2 = \texttt{give}\ c_1$. We define $\texttt{give}$ such that c_2 itself is a contract as well as c_1, thus, we can understand $\texttt{give}$ as a primitive that transforms one contract to another one.

The other primitives of POL are syntactically equivalent to the language defined by Peyton Jones and Eber, i.e.,

- $(c_3 = c_1\ \texttt{`and`}\ c_2)$ and $(c_4 = c_1\ \texttt{`or`}\ c_2)$[3]
  each transforms two contracts into a single one, $\texttt{and}$ requires that both con-

---

[1] Like Peyton Jones and Eber, we use Haskell syntax [8] to specify our POL. This allows to translate the ideas of this paper directly into program code.

[2] 'Negation' here means the exchange of obligation and rights, i.e., the rights of the one contract party become the obligations of the other party and vice versa.

[3] The backticks around the functions $\texttt{and}$ and $\texttt{or}$ are Haskell syntax and let us use $\texttt{and}$ and $\texttt{or}$ in infix notation.

tracts are executed and `or` requires that, no matter which of the two contracts will be executed, the other one may not be executed,

- $c_5 = \text{if } b \ c_1$

  is equivalent to c_1 if the condition b is satisfied at the time of evaluation and equivalent to `zero` otherwise (the primitive `zero` is introduced in c_{10}),

- $c_6 = \text{ifnot } b \ c_1$

  is equivalent to c_1 if the condition b is not satisfied at the time of evaluation and equivalent to `zero` otherwise,

- $c_7 = \text{when } b \ c_1$

  postpones c_1 to the first time when the condition b becomes true,

- $c_8 = \text{anytime } b \ c_1$

  gives the holder of the contract the right to acquire c_1 once (but not necessarily the first time) when the condition b is satisfied,

- $c_9 = \text{until } b \ c_1$

  gives the holder the right to acquire c_1 once (but not necessarily the first time) until the condition b is satisfied the first time, and

- $c_{10} = \text{zero}$

  is a contract without rights and obligations.

All transformed contracts c_i with $i = 2, \ldots, 9$ are contracts like c_1 and c_{10} and can thus be part of new transformations. A simple Privacy Option in which an institution acquires the right (but not the obligation) to use the data a for purpose p at time t can thus be stated in POL as $c_{11} = \text{when } (\text{at } t) \ (\text{data } a \ p \ \text{'or' zero})$.[4]
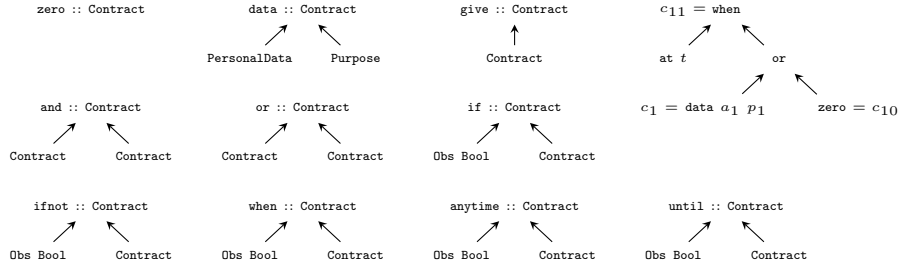
Figure 1. POL language primitives in type graph notation (left) and the contract c_{11} as function evaluation tree (right). The inductive definition of POL contracts is a major strength of the language.

A strict time constraint as in c_{11} is relevant when an institution that obtained data from an individual cannot use the data without interacting with the individual. Consider, for instance, a scenario where a shop can only use the purchase history of a customer when the customer returns to the shop [9,10].

---

[4] We assume that the function `at` transforms a time t into a condition suitable for the evaluation in contracts.

The right to choose the `zero` contract gives the shop the option not to use the data, e.g., if the costs connected to the usage are higher than the expected benefits. Other contract combinations are conceivable. For instance, $c_{12} =$ `until (at` t`) (data` a p `'or' zero)` models the obligation to delete data after a deadline t.[5]

3 POL Contract Management

Peyton Jones and Eber demonstrate how various semantics can be defined upon their language syntax. They elaborate on a valuation process $\mathcal{E}$, a denotational semantics which assigns a value to each contract. Valuation is better known as privacy measurement in the privacy domain. For many of the language primitives in POL, defining the valuation process would be a straight application of [7] and [4]. Due to that and space constraints, we focus here on an operational semantics for managing Privacy Options specified in POL. In this context, management means what is usually referred to as back-office management of contracts, i.e., timely execution, simplification, and possibly even deletion of outdated contracts.

Definition 1 (Contract management). *A contract management function is a (total) function from time to a sequence of I/O operations and a possibly simplified contract, thus satisfying the (informal) type definition*

$$\mathcal{ACTION} = \texttt{Date} \rightarrow \mathcal{IO}\ \texttt{Contract} . \tag{1}$$

Note that in the type of contract management functions, $\mathcal{IO}$ `Contract`, is a monad [11] which allows returning a sequence of I/O operations as well as a contract as function results.

The management semantics defined in Figure 2 describes which operations take place depending on a given contract in POL and relies on the definition of a couple of auxiliary functions:

- return : `Contract` $\rightarrow \mathcal{ACTION}$
  The function return(c) propagates c without any further actions as a (possibly simplified) replacement of the original contract.
- use : `PersonalData` $\times$ `Purpose` $\rightarrow \mathcal{ACTION}$
  The function use(a, p) lets the data controller immediately use the data item a for purpose p and returns `zero`, i.e., nullifies the original contract.
- send : `Contract` $\rightarrow \mathcal{ACTION}$
  The function send(c) transmits attribute values contained in c and stores the contract c (on the individual's side) for later inspection, e.g., recording a Data Track as in PrimeLife.

---

[5] Here we assume that *not using* data after the deadline and *deleting* the data when the deadline occurs has the same consequences.

$$\mathcal{O}_{\mathcal{E}}[\![\,\cdot\,]\!] : \texttt{Contract} \rightarrow \mathcal{ACTION}$$

$$\mathcal{O}_{\mathcal{E}}[\![\texttt{zero}]\!] = \mathrm{return}(\texttt{zero}) \tag{O1}$$

$$\mathcal{O}_{\mathcal{E}}[\![\texttt{data}\ a\ p]\!] = \mathrm{use}(a, p) \tag{O2}$$

$$\mathcal{O}_{\mathcal{E}}[\![\texttt{give}\ c]\!] = \mathrm{send}(c) \tag{O3}$$

$$\mathcal{O}_{\mathcal{E}}[\![c_1\ \text{`and`}\ c_2]\!] = \mathcal{O}_{\mathcal{E}}[\![c_1]\!] \parallel \mathcal{O}_{\mathcal{E}}[\![c_2]\!] \tag{O4}$$

$$\mathcal{O}_{\mathcal{E}}[\![c_1\ \text{`or`}\ c_2]\!] = \mathrm{greedy}_{\mathcal{E}}(\mathcal{O}_{\mathcal{E}}[\![c_2]\!], \mathcal{O}_{\mathcal{E}}[\![c_1]\!]) \tag{O5}$$

$$\mathcal{O}_{\mathcal{E}}[\![\texttt{if}\ o\ c]\!] = \mathrm{ifthenelse}(\mathcal{V}[\![o]\!], \mathcal{O}_{\mathcal{E}}[\![c]\!], \mathrm{return}(\texttt{zero})) \tag{O6}$$

$$\mathcal{O}_{\mathcal{E}}[\![\texttt{ifnot}\ o\ c]\!] = \mathrm{ifthenelse}(\mathcal{V}[\![o]\!], \mathrm{return}(\texttt{zero}), \mathcal{O}_{\mathcal{E}}[\![c]\!]) \tag{O7}$$

$$\mathcal{O}_{\mathcal{E}}[\![\texttt{when}\ o\ c]\!] = \mathrm{when}'_{\mathcal{E}}(\mathcal{V}[\![o]\!], \mathcal{O}_{\mathcal{E}}[\![c]\!]) \tag{O8}$$

$$\mathcal{O}_{\mathcal{E}}[\![\texttt{anytime}\ o\ c]\!] = \mathrm{stopping}'_{\mathcal{E}}(\mathcal{V}[\![o]\!], \mathcal{O}_{\mathcal{E}}[\![c]\!]) \tag{O9}$$

$$\mathcal{O}_{\mathcal{E}}[\![\texttt{until}\ o\ c]\!] = \mathrm{absorb}'_{\mathcal{E}}(\mathcal{V}[\![o]\!], \mathcal{O}_{\mathcal{E}}[\![c]\!]) \tag{O10}$$

Figure 2. Management of Privacy Options as operational semantics of POL.

- $\parallel : \mathcal{ACTION} \times \mathcal{ACTION} \rightarrow \mathcal{ACTION}$
  The function $a_1 \parallel a_2$ executes the actions a_1 and a_2 in parallel. After fetching the respective return values c_1 and c_2, the contract (c_1 `and` c_2) will be returned.
- $\mathrm{greedy}_{\mathcal{E}} : \mathcal{ACTION} \times \mathcal{ACTION} \rightarrow \mathcal{ACTION}$
  The function $\mathrm{greedy}_{\mathcal{E}}(a_1, a_2)$ executes either a_1 or a_2 depending on which action is most beneficial. The (prospective) benefit is determined by $\mathcal{E}$.
- $\mathrm{ifthenelse} : \mathcal{PR}\ \texttt{Bool} \times \mathcal{ACTION} \times \mathcal{ACTION} \rightarrow \mathcal{ACTION}$
  The function $\mathrm{ifthenelse}(o, a_1, a_2)$ executes a_1 if the observable o is true and a_2 otherwise.
- $\mathrm{when}'_{\mathcal{E}} : \mathcal{PR}\ \texttt{Bool} \times \mathcal{ACTION} \rightarrow \mathcal{ACTION}$
  The function $\mathrm{when}'_{\mathcal{E}}(o, a)$ executes a if the observable o is true, or returns the original contract otherwise.
- $\mathrm{stopping}'_{\mathcal{E}} : \mathcal{PR}\ \texttt{Bool} \times \mathcal{ACTION} \rightarrow \mathcal{ACTION}$
  The function $\mathrm{stopping}'_{\mathcal{E}}(o, a)$ solves the optimal stopping problem with regard to o, executes the action a if the optimal stopping point is reached, or returns the original contract otherwise.
- $\mathrm{absorb}'_{\mathcal{E}} : \mathcal{PR}\ \texttt{Bool} \times \mathcal{ACTION} \rightarrow \mathcal{ACTION}$
  The function $\mathrm{absorb}'_{\mathcal{E}}(o, a)$ returns $\texttt{zero}$ if the condition o is true, else it executes a if the optimal stopping point is reached, or otherwise returns the original contract.

4   Converting POL Contracts to PPL Sticky Policies

POL and PPL are both expressive languages, but of different nature, i.e., PPL is an access control language, whereas POL is a contract language, and none of them subsumes the semantics of the other one to its full extent. We therefore

refrain from translating the full-featured POL to PPL, but instead define a subset of our language, POL$^-$, and provide a mapping from this subset to PPL.

PPL Sticky Policies are an extension of XACML 3.0 [12], a powerful XML dialect for defining access control policies. While the complete XML Schema for PPL is yet not settled, the general language model is specified in [6]. Figure 3 displays the PPL elements which are relevant in this section.
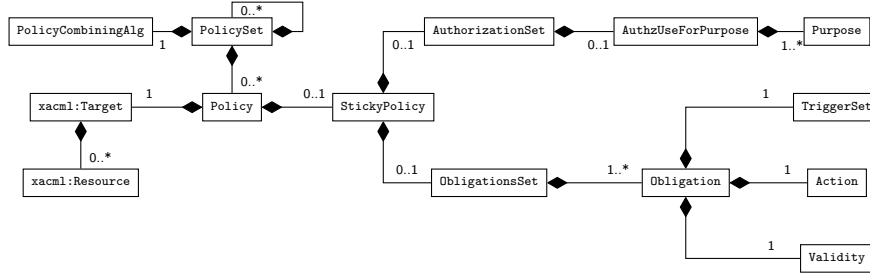

Figure 3. Model of the Sticky Policy and the obligation language as defined in PrimeLife [6, Figure 12]. Only elements being relevant for the conversion from POL to PPL are displayed.

In POL, we know three primitives that change the availability of data over time. PPL can simulate one of them, `until`, i.e., it is possible to delete data by means of triggering an obligation after a certain time. The other two time-dependent POL primitives, `when` and `anytime`, require to make previously un-accessible data available. In the current version of PPL [6], this can neither be achieved by triggering (a sequence of) obligations nor by defining Sticky Policies. The POL primitives `when` and `anytime` will therefore *not* be part of POL$^-$.

Three further assumptions are made for POL$^-$ and its translation to PPL:

A1. The conditions of the POL$^-$ primitives `until`, `if`, and `ifnot` are limited to points in time and can therefore be used as triggers of PPL `Obligations` (*Trigger at Time* [6]).
A2. We define a new `Action`, *immediate usage*, for PPL `Obligations` which causes the data to be used instantly.
A3. PPL `Actions` triggered by `Obligations` are always permitted.

Let us start with three simple examples:

- The POL contract $c_{10} = $ `zero` is a contract without rights or obligations. It therefore translates to an empty `PolicySet` in PPL.
- The POL contract $c_1 = $ `data` a_1 p_1 requires the immediate use of the data item a_1 for purpose p_1. After that usage, the data may not be used again. After the translation, the purpose p_1 will be enforced by a `StickyPolicy` (nested in a `Policy` within a `PolicySet`). The immediate usage of the data can only be enforced by an `Obligation` attached to the `StickyPolicy`. The

Obligation will be triggered *when data is stored*[6] and cause the Action *immediate usage.* Another Obligation will make sure that the data is used only once. It will be triggered *on data usage*[7] and will cause the data to be deleted instantly.

- For translating a complex POL contract, e. g., c_{13} = data a_1 p_1 'and' data a_2 p_2, we first translate the data statements to PolicySets, wrap them in a new PolicySet and add two Obligations, both triggering *on data usage* of either a_1 or a_2.[8] The idea is to cause the Action *immediate usage* of one data item as soon as the other one is used.

While the first two examples already provide general translation rules, the third is a special case of and in which the left and the right hand side are data statements. A general translation rule can, however, easily be derived by replacing the two Obligations in our example, which were tailored to the data primitive, with obligations that trigger other primitives accordingly.

The key idea outlined in these three examples is that every primitive of POL$^-$ can be translated to a PolicySet in PPL which may contain nested PolicySets. A translation of POL$^-$ to PPL will therefore form a tree of PolicySets which reflects the POL language structure as outlined in Figure 1, with one exception: the primitive give simply vanishes in the translation to PPL, since policies in PPL have the same appearance, no matter whether stored on the data subject's side or on the institution's side.

The translation of the remaining POL$^-$ primitives is straight-forward. Just like and, c_4 = c_1 'or' c_2 can be translated by first translating c_1 and c_2 to PolicySets, placing them into a new PolicySet, and complementing them with an Obligation which in case of or will delete the data in c_1 and c_2 as soon as an *on data usage* event occurs with regard to data of either c_1 or c_2.

The translation rule for a contract c_9 = until b c_1 similarly first translates c_1 to a PolicySet, wraps it in a new PolicySet, and complements it with an Obligation which triggers when the condition b becomes true. Due to our assumption A1, this can be realised by the PPL *Trigger at Time.* As soon as the Obligation is triggered, it will delete the data.

Finally, translating c_5 = if b c_1 or c_6 = ifnot b c_1 respectively requires to translate c_1 to its corresponding PolicySet, placing it into a new one, and complementing it with Obligations such that the data items in c_1 are deleted if b is false (if) or true (ifnot), respectively.

We used data deletion in order to enforce the semantics of POL primitives in their PPL translation. This works well as long as the data items are not equal in different branches of the PolicySet tree. If this is not the case, PPL needs a method to (de)activate Policies depending on a (global) state. Note that if we use anonymous credentials [13,14] for representing data item, we do not need any

---

[6] This trigger is can be realised by *Trigger at Time* or *Trigger Personal Data Sent.* [6]

[7] In [6], this trigger is called *Trigger Personal Data Accessed for Purpose.*

[8] To be precise, each of these Obligations is embedded in a StickyPolicy which needs to be targeted to data a_1 and purpose p_1 or a_2 and p_2, respectively.

of these assumptions. Then even credentials representing the same data would not be linkable and deletion would only affect one occurrence of the data item.

5   Canonical Form Contracts in POL

POL is not only an effort in language design, but primarily one in the privacy research domain, and it is important that we do not create new problems in one domain when solving problems in the other. A problem that therefore deserves our attention is that of (unintentionally) creating *covert channels* by using freedom in the POL syntax for expressing semantically equivalent contracts. Such a covert channel may be used to transport any information, but even if not used intentionally it will most likely convey information about the originator of a contract. In this section, we show that POL has freedom in its syntax and how to eliminate it. We show that each POL contract can be transformed to a canonical form by using the abstract rewriting system we define.

An example illustrates the syntactical freedom which exists in POL. The contracts on the left and right sides of the Equations (2) and (3) are semantically equivalent (we use the symbol $\equiv_{\mathrm{POL}}$), however the syntax is different,

$$\texttt{give zero} \equiv_{\mathrm{POL}} \texttt{zero} , \tag{2}$$

$$\texttt{give (give (data } a\ p)) \equiv_{\mathrm{POL}} \texttt{data } a\ p . \tag{3}$$

The degree of freedom increases when contracts become more advanced. We see that transformations to equivalent contracts follow generic transformation rules, e. g.,

$$(\texttt{if } o_1\ (\texttt{ifnot } o_1\ (\texttt{data } a_1\ p_1)))\ \texttt{`or`}\ (\texttt{until } o_2\ (\texttt{when } o_2\ (\texttt{data } a_2\ p_2)))$$

$\overset{\mathrm{R20}}{\equiv_{\mathrm{POL}}}\ \texttt{zero `or` (until } o_2\ (\texttt{when } o_2\ (\texttt{data } a_2\ p_2)))$

$\overset{\mathrm{R45}}{\equiv_{\mathrm{POL}}}\ \texttt{zero `or` zero}$

$\overset{\mathrm{R7}}{\equiv_{\mathrm{POL}}}\ \texttt{zero}$

$\overset{\mathrm{R1}}{\equiv_{\mathrm{POL}}}\ \texttt{give zero} \tag{4}$

$\overset{\mathrm{R3}}{\equiv_{\mathrm{POL}}}\ \texttt{(give zero) `and` zero}$

$\overset{\mathrm{R1}}{\equiv_{\mathrm{POL}}}\ \texttt{(give zero) `and` (give zero)}$

$\overset{\mathrm{R5}}{\equiv_{\mathrm{POL}}}\ \texttt{give (zero `and` zero)}$

$\overset{\mathrm{R37}}{\equiv_{\mathrm{POL}}}\ \texttt{give (zero `and` (anytime } o_3\ (\texttt{ifnot } o_3\ (\texttt{data } a_3\ p_3)))) .$

In fact, the contract $c_{10} = \texttt{zero}$ has an infinite number of incarnations, thus giving each contract originator the option to use a unique version of c_{10}. For our canonical form it would be best, if all incarnations of c_{10} could be reduced to the simplest version, $\texttt{zero}$.

A standard method for reducing a language term to its canonical form is applying an abstract rewriting system (ARS). An ARS consists of a number of reduction rules which can be applied in any order to a given language term. Terms that cannot be reduced any further are in canonical form.

For POL, a suitable ARS is defined by the reduction rules in Figure 4. In Equation (4), we use the same set of rules to transform the initial contract to its equivalents. Two properties of ARS' are interesting for our canonical form of POL and we will look at them in the rest of this section:

- Termination: Given a POL contract, all applicable reduction rule sequences are *finite*, i.e., finish after a finite number of reduction steps with a canonical form contract which cannot be reduced any further.
- Confluence: Given a POL contract, all applicable reduction rule sequences produce the *same* canonical form contract.

It is easy to see that for termination, an ARS may not contain reduction rules supporting commutativity. Let us assume that we have a rule c_1 'and' $c_2 \to c_2$ 'and' c_1 and c matches the rule's left hand side. Then, c can be reduced to the right hand side c', thus $c \to c'$, but then we see that c' matches the left hand side of the same rule again and we can create an infinite sequence of reductions $c \to c' \to c \to c' \to \cdots$. This would violate the termination property.

The non-commutativity of `and` and `or`, however, may interfere with the application of the Rules (R6) and (R7). We learn, for instance, from Rule (R7) that a contract $(c_1$ 'or' c_1 'or' $c_2)$ can be reduced to $(c_1$ 'or' $c_2)$, whereas the contract $(c_1$ 'or' c_2 'or' $c_1)$ is not reducible in the same way, though being equivalent except for the commutation of the contracts c_1 and c_2. This dilemma can be solved by assuming an appropriate order of contracts in the function arguments of `and` and `or` such that Rule (R6) and (R7) matches if matching is possible with any permutation of the function arguments.

Termination can be proved by showing that a well-founded order $\succ_{\text{POL}}$ exists such that $\ell \succ_{\text{POL}} r$ holds for all reductions where ℓ denotes the left hand side of the rule and r denotes the right hand side. For POL, this order relation can be defined by means of the order $\geq$ on the natural numbers,

$$c \succ_{\text{POL}} c' \iff m(c) \geq m(c') \qquad (\forall c, c' \in \text{POL}), \tag{5}$$

with the function $m : \text{POL} \to \mathbb{N}$ defined as in Figure 5. It is easy to see that for any contract c it holds that $m(c) > 1$ and for any instantiation of a reduction rule in Figure 4 it holds that $m(\ell) > m(r)$ with ℓ being the left hand side of the reduction and r being the right hand side. Thus a reduction sequence for a POL contract c finishes with a canonical form at latest after $m(c) - 2$ reduction steps which proves that our ARS has the termination property.

Confluence of a terminating ARS can be proved by showing local confluence for all contracts, i.e., showing that all two different (one-step) reductions of one contract always terminate in the same canonical form. We will not explicitly prove confluence in this paper and postpone this task to future work. In careful checks, however, we were not able to produce a contract which violates the conditions for confluence.

$$\text{give zero} \to \text{zero} \tag{R1}$$
$$\text{give (give } c) \to c \tag{R2}$$

$$c \text{ `and` zero} \to c \tag{R3}$$
$$\text{zero `and` } c \to c \tag{R4}$$
$$(\text{give } c_1) \text{ `and` (give } c_2) \to \text{give } (c_1 \text{ `and` } c_2) \tag{R5}$$
$$(\text{if } o\ c) \text{ `and` (ifnot } o\ c) \to c \tag{R6}$$

$$c \text{ `or` } c \to c \tag{R7}$$
$$(\text{give } c_1) \text{ `or` (give } c_2) \to \text{give } (c_1 \text{ `or` } c_2) \tag{R8}$$
$$(c_1 \text{ `and` } c_2) \text{ `or` } (c_1 \text{ `and` } c_3) \to c_1 \text{ `and` } (c_2 \text{ `or` } c_3) \tag{R9}$$
$$(c_1 \text{ `and` } c_2) \text{ `or` } (c_3 \text{ `and` } c_1) \to c_1 \text{ `and` } (c_2 \text{ `or` } c_3) \tag{R10}$$
$$(c_1 \text{ `and` } c_2) \text{ `or` } (c_3 \text{ `and` } c_2) \to (c_1 \text{ `or` } c_3) \text{ `and` } c_2 \tag{R11}$$
$$(c_1 \text{ `and` } c_2) \text{ `or` } (c_2 \text{ `and` } c_3) \to (c_1 \text{ `or` } c_3) \text{ `and` } c_2 \tag{R12}$$

$$\text{if } o \text{ (give } c) \to \text{give (if } o\ c) \tag{R13}$$
$$\text{ifnot } o \text{ (give } c) \to \text{give (ifnot } o\ c) \tag{R14}$$
$$\text{if } o\ (c_1 \text{ `and` } c_2) \to (\text{if } o\ c_1) \text{ `and` (if } o\ c_2) \tag{R15}$$
$$\text{ifnot } o\ (c_1 \text{ `and` } c_2) \to (\text{ifnot } o\ c_1) \text{ `and` (ifnot } o\ c_2) \tag{R16}$$
$$\text{if } o\ (c_1 \text{ `or` } c_2) \to (\text{if } o\ c_1) \text{ `or` (if } o\ c_2) \tag{R17}$$
$$\text{ifnot } o\ (c_1 \text{ `or` } c_2) \to (\text{ifnot } o\ c_1) \text{ `or` (ifnot } o\ c_2) \tag{R18}$$
$$\text{if } o \text{ (if } o\ c) \to \text{if } o\ c \tag{R19}$$
$$\text{if } o \text{ (ifnot } o\ c) \to \text{zero} \tag{R20}$$
$$\text{ifnot } o \text{ (if } o\ c) \to \text{zero} \tag{R21}$$
$$\text{ifnot } o \text{ (ifnot } o\ c) \to \text{ifnot } o\ c \tag{R22}$$
$$\text{if } o \text{ (when } o\ c) \to \text{if } o\ c \tag{R23}$$
$$\text{if } o \text{ (until } o\ c) \to \text{zero} \tag{R24}$$
$$\text{ifnot } o \text{ (until } o\ c) \to \text{until } o\ c \tag{R25}$$

$$\text{when } o \text{ zero} \to \text{zero} \tag{R26}$$
$$\text{when } o \text{ (give } c) \to \text{give (when } o\ c) \tag{R27}$$
$$\text{when } o\ (c_1 \text{ `and` } c_2) \to (\text{when } o\ c_1) \text{ `and` (when } o\ c_2) \tag{R28}$$
$$\text{when } o \text{ (if } o\ c) \to \text{when } o\ c \tag{R29}$$
$$\text{when } o \text{ (ifnot } o\ c) \to \text{zero} \tag{R30}$$
$$\text{when } o \text{ (when } o\ c) \to \text{when } o\ c \tag{R31}$$
$$\text{when } o \text{ (anytime } o\ c) \to \text{when } o\ c \tag{R32}$$
$$\text{when } o \text{ (until } o\ c) \to \text{zero} \tag{R33}$$

$\text{anytime } o \text{ zero} \to \text{zero}$	(R34)		$\text{until } o \text{ zero} \to \text{zero}$	(R41)
$\text{anytime } o \text{ (give } c) \to \text{give (anytime } o\ c)$	(R35)		$\text{until } o \text{ (give } c) \to \text{give (until } o\ c)$	(R42)
$\text{anytime } o \text{ (if } o\ c) \to \text{anytime } o\ c$	(R36)		$\text{until } o \text{ (if } o\ c) \to \text{zero}$	(R43)
$\text{anytime } o \text{ (ifnot } o\ c) \to \text{zero}$	(R37)		$\text{until } o \text{ (ifnot } o\ c) \to \text{until } o\ c$	(R44)
$\text{anytime } o \text{ (when } o\ c) \to \text{anytime } o\ c$	(R38)		$\text{until } o \text{ (when } o\ c) \to \text{zero}$	(R45)
$\text{anytime } o \text{ (anytime } o\ c) \to \text{anytime } o\ c$	(R39)		$\text{until } o \text{ (anytime } o\ c) \to \text{zero}$	(R46)
$\text{anytime } o \text{ (until } o\ c) \to \text{zero}$	(R40)		$\text{until } o \text{ (until } o\ c) \to \text{until } o\ c$	(R47)

Figure 4. Abstract term rewriting system for POL. The application of the rewriting rules terminates in a canonical form of the language.

$$
m : \begin{cases}
\texttt{zero} & \mapsto 2 \\
\texttt{data } a\ p & \mapsto 2 \\
\texttt{give } c & \mapsto m(c) + 1 \\
c \text{ 'and' } c' & \mapsto m(c) \cdot m(c') \\
c \text{ 'or' } c' & \mapsto m(c) \cdot m(c') \\
\texttt{if } o\ c & \mapsto \big(m(c)\big)^3 \\
\texttt{ifnot } o\ c & \mapsto \big(m(c)\big)^3 \\
\texttt{when } o\ c & \mapsto \big(m(c)\big)^3 \\
\texttt{anytime } o\ c & \mapsto \big(m(c)\big)^3 \\
\texttt{until } o\ c & \mapsto \big(m(c)\big)^3
\end{cases}
$$

Figure 5. The 'mass' function $m : \text{POL} \to \mathbb{N}$. It holds that for any reduction $c_1 \to c_2$ with regard to a rule in Figure 4, $m(c_1) > m(c_2)$ $(c_1, c_2 \in \text{POL})$.

6 Extensions

There are two possible ways of extending the language: the first option is adding new primitives to the language and defining their semantics. While this is an obvious way of extending a language, it should be reserved to experts only. Adding primitives represents the risk of creating redundancy in the semantics which can lead to ambiguous contracts. The other way of extending the language is to define combinators. A combinator creates a contract by *applying* primitives rather than defining new primitives. In particular, if POL is implemented as *embedded* domain-specific language within a more expressive host language, creating a combinator library becomes easy even for end-users. Consider, for instance, a combinator that allows to execute a given contract twice. We can define this combinator as follows:

$$\texttt{twice } f\ c = (f\ c) \text{ 'and' } (f\ c) \, . \tag{C1}$$

This definition allows us to note a new contract, $c_{14} = \texttt{until } (\texttt{at } t) \text{ 'twice' } c_1$, which appears like native English and provides intuitive semantics without further definitions.

Assuming the full expressiveness of a multi-purpose language, here Haskell, we can even generalise `twice` to `times`,

$$\texttt{times } n\ c = \texttt{foldr1 (and) (replicate } n\ c) \, , \tag{C2}$$

which immediately allows us to note $c_{15} = 5 \text{ 'times' } c_{14}$ as a new contract and again with intuitive meaning.

Another easy extension is the `cond` combinator which was a primitive in an earlier version of POL. It combines the functionality of the new primitives `if` and `ifnot`, but was discarded as a first class citizen of the language in order to keep the abstract rewriting system (Figure 4) small,

$$\texttt{cond } o\ c_1\ c_2 = (\texttt{if } o\ c_1) \text{ 'and' } (\texttt{ifnot } o\ c_2) \, . \tag{C3}$$

While these three combinators make the language handy, they do not reflect the specific vocabulary used in the privacy research domain. One of these concepts which is easily translated to POL is the definition of a retention period for data usage. We assume that the host language provides a function `now` which returns the current time and define,

$$\texttt{retain}\ t\ c = \texttt{until}\ (\texttt{at}\ (\texttt{now} + t))\ (c\ \texttt{`or`}\ \texttt{zero})\,, \tag{C4}$$

a contract combinator which allows the holder of the contract to use the data in c (once) within the time frame t, e.g., $c_{16} = \texttt{retain}\ "6\ \texttt{months}"\ c_1$.

Even applications of our language may look like extensions at a first glance. While POL provides, for instance, syntax for rights and obligations of contract partners, we have intentionally excluded the specification of contract partners from the scope of POL. This allowed us to focus on rights and obligations as one subject and postpone the specification of contract partners as a subject for another language which could wrap round POL. Approaching the specification of contract partners in an independent language is particularly interesting when accounting for complex contract partner relations, e.g., when two parties write a contract about the data of a third party.

7 Conclusions

We have specified a formal language POL for Privacy Options and added a semantics for managing Privacy Options. The management semantics allows to execute and simplify contracts. A simple modification of this semantics would switch off the simplification and retain the original contracts, e.g., for PrimeLife's Data Track.

We compare POL with PPL, a language with similar purpose, by elaborating on translation rules from POL to PPL. The translation shows that POL has its strengths in defining contracts that evolve over time. While in PPL data, once it became inaccessible, stays inaccessible forever (data deletion), we can define contracts in POL which flexibly allow data usage depending on time or events. POL is, however, not meant to be a drop-in replacement for PPL and rather focuses on a fraction of the functionality provided by PPL, i.e., Sticky Policies.

In contrast to most other approaches in the privacy policy domain, POL contracts can be transformed to a canonical form. This allows to eliminate information about the contract originator hidden in the freedom of the POL syntax. We deem that this is a particularly interesting feature in privacy negotiations when the contract proposals are evaluated by negotiation partners and none of them wants to reveal more information than the terms and conditions under which they could accept a contract.

Moreover, we have outlined how POL can be extended. Experts may add new language primitives and benefit from the inductive structure of the language, i.e., in most cases it will be sufficient to define the semantics of the added primitive in order to provide an extended language with all semantics. End-users can extract standard contract patterns and make them available to a larger audience by

defining combinators, i. e., functions that combine primitives in a standardised way. Combinators may be defined in a high-level programming language or even in a visual environment, depending on the capabilities of the end-user. Neither the choice of a specific programming language, nor the choice of a suitable visual environment is determined by the work in this paper.

Acknowledgements

References

1. Laudon, K.C.: Markets and privacy. Commun. ACM **39**(9) (1996) 92–104
2. Taylor, C.R.: Consumer privacy and the market for customer information. The RAND Journal of Economics **35**(4) (2004) 631–650
3. Acquisti, A.: Protecting privacy with economics: Economic incentives for preventive technologies in ubiquitous computing environments. In: Workshop on Socially-informed Design of Privacy-enhancing Solutions in Ubiquitous Computing. (2002) 1–7
4. Berthold, S., Böhme, R.: Valuating privacy with option pricing theory. In: Workshop on the Economics of Information Security (WEIS), University College London, UK (2009)
5. Hann, I.H., Hui, K.L., Lee, T.S., Png, I.P.L.: Online information privacy: Measuring the cost-benefit trade-off. In: Proceedings of the 23rd International Conference on Information Systems (ICIS 2002). (2002)
6. Raggett, D.: Draft 2nd design for policy languages and protocols. Technical Report H 5.3.2, PrimeLife project (2009)
7. Peyton Jones, S., Eber, J.M.: How to write a financial contract. In Gibbons, J., de Moor, O., eds.: The Fun of Programming, Palgrave Macmillan (2003)
8. Peyton Jones, S., ed.: Haskell 98 Language and Libraries – The Revised Report. Cambridge University Press (2003)
9. Acquisti, A., Varian, H.R.: Conditioning prices on purchase history. Marketing Science **24**(3) (2005) 1–15
10. Böhme, R., Koble, S.: Pricing strategies in electronic marketplaces with privacy-enhancing technologies. Wirtschaftsinformatik **49** (2007) 16–25
11. Wadler, P.: The essence of functional programming. In: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. POPL '92, New York, NY, USA, ACM (1992) 1–14
12. Rissanen, E.: OASIS extensible access control language (XACML) version 3.0. OASIS working draft 10, OASIS (March 2009)
13. Chaum, D.: Security without identification: Transaction systems to make big brother obsolete. Communications of the ACM **28**(10) (1985) 1030–1044
14. Camenisch, J., Lysyanskaya, A.: An efficient system for non-transferable anonymous credentials with optional anonymity revocation. In: Proceedings of the Advances in Cryptology – EUROCRYPT 2001. Volume 2045/2001 of LNCS., Springer Berlin/Heidelberg (May 2001) 93–118