Computer Science and Artificial Intelligence Laboratory
Technical Report

MIT-CSAIL-TR-2014-020 October 1, 2014

Constraint Generation for the Jeeves Privacy Language

Eva Rose

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Constraint Generation for the Jeeves Privacy Language

Eva Rose
Courant Institute, New York University*
evarose@cs.nyu.edu

September 30, 2014

Abstract

Our goal is to present a completed, semantic formalization of the Jeeves privacy language evaluation engine, based on the original Jeeves constraint semantics defined by Yang et al at POPL12 [23], but sufficiently strong to support a first complete implementation thereof. Specifically, we present and implement a syntactically and semantically completed concrete syntax for Jeeves that meets the example criteria given in the paper. We also present and implement the associated translation to λ_J, but here formulated by a completed and decompositional operational semantic formulation. Finally, we present an enhanced and decompositional, non-substitutional operational semantic formulation and implementation of the λ_J evaluation engine (the dynamic semantics) with privacy constraints. In particular, we show how implementing the constraints can be defined as a monad, and evaluation can be defined as monadic operation on the constraint environment. The implementations are all completed in Haskell, utilizing its almost one-to-one capability to transparently reflect the underlying semantic reasoning when formalized this way. In practice, we have applied the "literate" program facility of Haskell to this report, a feature that enables the source LaTeX to also serve as the source code for the implementation (skipping the report-parts as comment regions). The implementation is published as a github project [17].

*This work was conducted whilst at CSAIL, Massachussets Institute of Technology, 2012.

Contents

1 Introduction

Jeeves was first introduced as an (impure) functional (constraint logic) programming language by Yang et al [23], which distinguish itself by allowing *explicit syntax for automatic privacy enforcement*. In other words, the syntax and semantics of the language is designed to support that a programmer composes privacy policies directly at the source level, by way of a special, designated privacy syntax over a not yet known context. It is worth noticing, that there is *no semantic specification for Jeeves at the source level*. Jeeves' semantics is entirely defined by a syntax translation to an intermediary constraint functional language, λ_J , together with a λ_J evaluation engine (defined over the same input-output function as source-level Jeeves). In order to run Jeeves with the argued privacy guarantees, it is therefore pivotal to have a correct and running implementation of λ_J evaluations as well as a correct Jeeves-to-λ_J syntax-translation, which is the main goal of this report. In Figure 1 we have illustrated how Jeeves' evaluation engine is logistically defined in terms of the λ_J language:

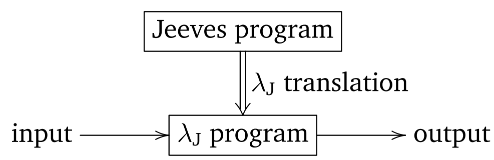

Figure 1: Running a Jeeves program

The explicit privacy constructs in Jeeves, and thus λ_J is in fact not just syntactic sugar for the underlying conventional semantics, but is interpreted independently in terms of logical constraints on the data access and writes. The runtime generated set of logical constraints that safeguards the policies, are defined as part of the usual dynamic and static semantics. As we show with our re-formalization of the dynamic semantics, the constraint part of the semantics can in fact be defined as a monoid, thus following an othogonal evaluation pattern with respect to the underlying traditional evaluation semantics. An observation which not only makes it straighforward to implement, but makes privacy leak arguments straight forward to express and proof.

In this report, we have re-stated the original formalizations of the abstract syntax for source-level Jeeves, as well as for λ_J , by way of algebraic and denotational (domain) specifications. As a new thing, we have added a concrete syntax for source-level Jeeves as an LL(1) grammar, along which we have re-adjusted the λ_J compilation to be specified as a syntax-directed translation. Furthermore, we are re-formulating the definition of the dynamic (evaluation) semantics by way of operational (natural) semantics. In the process, we have added a number of technical clarifying details and assumptions, as summarized in section A. Notably, we have imposed a formal (denotational) definition of a Jeeves aka λ_J "*program*", and semantically specified how programs should be evaluated at the top level . We should mention, that the treatment of types (and the associated static semantics) has been omitted, thus leaving it to the user not to evaluate ill-formed terms or recursively defined policies.

The implementation has been conducted in Haskell. Using that specific functional language, provides a particular elegant and one-to-one imlementation map of the denotational and operational specifications of Jeeves, aka λ_J . In fact, by having implemented the dynamic, operational semantics of λ_J , we have obtained a Jeeves/ λ_J interpreter. To implement the parser, we in fact used the Haskell monadic parser combinator library [10], which has been included in full in Appendix B.2. One limitation with the current implementation, however, is that we have not included a constraint solver, but merely outputs all constraints to be further analysed. It is, however, a minor technical detail to add an off-the-shelf constraint solver to the backend.

The presentation of the implementation in the report, has been done by using the *literate programming facility* of Haskell, as described in Notation 1.1. En bref, it permits us to use the source LaTeX of the report as the source code of the program. In the report, we have preceded each code fragments with the formalism it implements, so that the elegant, one-to-one correspondance between the formalism and the Haskell program serves as a convincing argument for the authenticity of the Jeeves implementation (and vice versa, in that the running program fragments support the formalizations). To ease readability we have furthermore been typesetting and color coding the Haskell implementation, also summarized in Notation 1.1.

1.1 Notation (The Haskell implementation). The Haskell program has been integrated with the report as specially designated **Haskell** sections by means of the literate programming facility for Haskell [6]. This facility (file extension .lhs), enables Haskell code and text to be intertwined, yet percieved either as program (like .hs extension) with text segments appearing as comments, or as a TeX report (like the .tex extension) where code fragments appear as text. All depending on which command is run upon the ensemble.

For convenience, the typesetting of the Haskell sections uses coloring for emphasis and prints the character sequences shown in the following table as special characters.

Symbol used in report	λ	$+\!\!+$	$\rightarrow$	$\leftarrow$	$\Rightarrow$	$\leq$	$\geq$	$\equiv$	$\circ$	$\gg$	$\gg\!=$
Haskell source form	`\`	`++`	`->`	`<-`	`=>`	`<=`	`>=`	`==`	`.`	`>>`	`>>=`

Before we proceed, we will introduce the literate Haskell programming head.

1.2 Haskell (main program and imports).

```haskell

------------------------------------------------------------------
-- Evaluates Jeeves programs and generates policy constraints
-- Eva Rose <evarose@mit.edu>
-- CSAIL August 2012.
------------------------------------------------------------------

-- Imported data types
import Data.Map (Map,(!),insert, delete ,empty,union,member,assocs)
import Char
```

The semantic and syntactic specification styles follow those of Plotkin [16], Kahn [13], Schmidt [20], Bachus and Naur [5], alongside the formal abbreviations, shorthands and stylistic elements which we have summarized in Notation 1.3.

1.3 Notation (Formal style summary). We have adopted the following conventions:

- the shorthand '$Sym\ \cdots\ Sym$' to denote a finite repetition of the pattern Sym, one or more times,
- the `teletype` font for keywords in source-level Jeeves, and sans serif for keywords in λ_J.

Before we describe how the report is structured we will recall, with two examples from the original paper, what programming with Jeeves looks like. The first being a simple naming policy example, and the second having to do with the tasks involved in accessing and managing papers for a scientific conference. Both will serve as our canonical examples throughout the report.

1.4 Example (Canonical examples). Figure 2 and Figure 3 consist of two Jeeves programming examples from Sec. 2.2 in [23, p.87], but as slightly altered versions. Among other things, we have fixed the format of a Jeeves program c.f. Definition 2.1. Furthermore, we have changed the examples in the following ways:

- tacitly omitted 'reviews' from the 'paper' record and from the policy definitions, as dealing with listings just introduce "noise" to the presentation without adding any significant insight,
- only to allow policies on the form "`policy` lx : e `then` lv `in` e"; we have thus moderated the original examples by adding "`in` p" to those policy definitions were the keyword "in" was missing,
- omitted types in accordance with our design decisions.

```
-- Jeeves example adapted from Yang etal. (POPL 2012).

let name =
  level a in
  policy a: !(context = "alice") then bottom in
    < "Anonymous" | "Alice" >(a)

let msg = "Author is " + name

print {"alice"} msg
print {"bob"} msg
```

Figure 2: Naming policy

The program in Figure 2 overall introduces a policy ('*policy...: !(context="alice")...*') which regulates what value the variable '*name*' is assigned: either to '`"Anonymous"`' or to '`"Alice"`'. Let us first hone in on the (first order) logical policy condition '*!(context="alice")*'. This is simply a boolean expression stating to be true if the value of the designated, built-in variable '*context*' is different from the string '`"alice"`', otherwise false. (The '`!`' stands for negation.) In the first case, '`bottom`' will select the first value of the pair '`<"Anonymous","Alice">`', whereas in the latter case, the second value will be chosen to be assigned to '`name`'. Now hone in on the print-statements at the bottom of the program. The semantics tells that the '*context*' variable first is automatically set to the string '`"alice"`' (by the '`print {"alice"}...`' statement); subsequently to the string '`"bob"`' (by the '`print {"bob"}...`' statement). These print-statements are also the ones responsible for the program output by printing the value of the variable '`msg`', which in turn is designated by the values of '`name`' (by the '`let msg = ...  name`' statement). In other words, the input-output functionality is given by the print statements. Thus, upon the input: '`alice`' '`bob`', the expected output of this program is: '`Author is Alice`' '`Author is Anonymous`'.

The program in Figure 3 overall introduces policies for managing access to conference papers, depending upon the formal role a person possesses. The policies to avoid leaking the name of a paper author at the wrong time in the review process, follows the basic principle of the naming policy in Figure 2, just in a more complex setting. The first let-statement of the program creates a paper record through the function '`mkpaper`' with information on '`title`', '`author`', and '`accepted`' status. By way of the level variables '`tp`', '`authp`', and '`accp`', three leak policies are being added as conditioned values, each of which is being defined by the subsequent let-statements. Take for example the first of these: '`addTitlePolicy p tp;`'. The policy states that if a viewer is not the author, and the viewer's role is neither that of a reviewer's or program chair, and finally, if not

```
-- Jeeves example adapted from Yang etal. (POPL 2012).

let mkPaper
   title author accepted =
   level tp, authp, accp in
   let p = { title = <""|title>(tp)
           ; author = <"Anonymized"|author>(authp)
           ; accepted = <"none"|accepted>(accp) } in
   addTitlePolicy p tp ; addAuthorPolicy p authp;
   addAcceptedPolicy p accp;
     p

let addTitlePolicy p a =
   policy a: ! (context.viewer.name = p.author
     || context.viewer.role = Reviewer
     || context.viewer.role = PC
     || context.stage = Public && isAccepted p) then bottom
   in p

let addAcceptedPolicy p a =
  policy a: ! (context.viewer.role = Reviewer
     || context.viewer.role = PC
     || context.stage = Public) then bottom
   in p

let addAuthorPolicy p n =
  policy n: ! (isAuthor p context.viewer
     || context.stage = Public && isAccepted p) then bottom
   in p

let alice = {name = "Alice"; role = PC}

let bob = {name = "Bob"; role = Reviewer}

let isAuthor p viewer = (p.author = viewer.name)

let isAccepted p = !(p.accepted = "none")

print {{viewer = alice; stage = Public}} mkPaper "MyPaper" "Alice" Accepted

print {{viewer = bob; stage = Public}} mkPaper "MyPaper" "Alice" Accepted
```

Figure 3: Conference management policies

the review process is over (the stage is then then '`public`') or the paper has been accepted, then the title can only be released as `""` (because the '`bottom`' value selects the first of the title pair values in '*mkpaper*', which is `""`). Similarly for the other policy specifications. The next set of let specifications set the variables '`alice`' and '`bob`' with concrete review records, and the two boolean functions '`isAuthor`' and '`isAccepted`' are similarly set with concrete boolean expressions. Also here, the print-statements are responsible for assigning the '`context`' variable with concrete viewer and stage information, and to output a record corresponding to a paper, through a call to "*mkpaper*", where the individual paper fields have been filtered by the specified policies.

We assume that the reader of this report is familiar with the core principles of the original Jeeves definition in Yang et al [23]. Furthermore, we assume an understanding of functional programming in Haskell [6, 12], as well as basic familiarity with algebraic specifications and semantics [5, 13, 16, 20].

Finally, we describe how the report is structured:

- In Sec. 2, (source-level) Jeeves is specified both by its abstract as well as a newly formulated concrete syntax. The concrete syntax is specified in terms of an LL(1) grammar along with the lexical tokens for Jeeves and their implementation in Haskell.
- In Sec. 3, (intermediary) λ_{J} is specified by its abstract syntax alongside its implementation in Haskell. Notably, the notion of a λ_{J} program has been added to the original syntax together with additional expression syntax (thunks). The ensemble is presented alongside its implementation in Haskell.
- In Sec. 4, we formally present the translation from Jeeves to λ_{J} as a derivation. The translation is given as a syntax directed compilation of the concrete Jeeves syntax to λ_{J}, together with its Haskell implementation. The implementation is in fact a set of Jeeves parsers, which builds abstract syntax trees in accordance with the abstract λ_{J} specification in Section 3.
- In Sec. 5, we formally present the symbolic normal forms with the addition of a static binding environment component. The implementation of those are presented together with operations on the environment, notably insertion and lookups.
- In Sec. 6, we specify the notion of a hard constraint algebra, and soft constraint algebra as well as the notion of a path condition algebra. We finally show how the set of hard and soft constraints can be implemented as a monad in Haskell, together with update and reset operations thereon.
- In Sec. 7, the λ_{J} evaluation engine is formally specified as a big step, compositional, non-substitution based operational semantics alongside our specification of a λ_{J} program evaluation. The Haskell implementation in terms of a λ_{J} interpreter is presented alongside the formalizations. The input-output functionality is equally specified, and a program outcome is defined in our setting as a series of "effects" written to output channels.
- In Sec. 8, we show how to load and run a jeeves program with our system, as well as how to use our system to translate a Jeeves program to λ_{J}.
- Finally, in section 9, we conclude our work, and discuss further directions in section 10.

We will describe in which way our formalizations deviates from the original formulations *c.f.* Yang et al [23] as we go along, and summarize the discrepancies in Appendix A.

2 The Jeeves syntax

In this section, we restate the Jeeves abstract syntax from the original paper [23, Figure 1], and a (new) formulation of a Jeeves concrete syntax. We also specify the basic algebraic sorts for literals that are assumed by the specifications, and present them as Jeeves lexical tokens for the λ_J translation in subsequent sections. The syntax specifications include some language restrictions and modifications compared to the original rendering in accordance with section A. Notably, restrictions on the shape of a Jeeves program, such that all `let`-statements (*i.e.*, `let` constructs without an `in`-part) must be trailed by `print`-statements, and both are only to appear at the top-level of the program.

The abstract syntax merely serves as a quick guide to the Jeeves language just as in the original form [23, Figure 1]. It is presented as a complete, algebraic specification which describes Jeeves programs, expressions, and tokens in a top-down fashion, following Notation 1.3. The concrete syntax for source-level Jeeves has been formulated as an (unambiguous) LL(1) grammar from scratch. Thereby making it straightforward to apply the Haskell monadic parser combinator library [10] when implementing the λ_J translation function in subsequent sections. The syntax precisely states the way operator precedence and scoping is being handled, if not by the original specification [23, Figure 1], then by the original Jeeves program examples [23, Section 2] (for more details on discrepancies and differences, visit section AS).

The only Haskell implementation in this section is that of the Jeeves lexical tokens in Haskell 2.6.

2.1 Definition (abstract Jeeves syntax).

$$
\begin{array}{rcl}
p \in Pgm & ::= & \texttt{let}\ x\ \ldots\ x \texttt{=} e \\
 & & \vdots \\
 & & \texttt{let}\ x\ \ldots\ x \texttt{=} e \\
 & & \texttt{output}\ \ \texttt{e}\ \ e \\
 & & \vdots \\
 & & \texttt{output}\ \ \texttt{e}\ \ e \\
e \in Exp & ::= & \ b \mid n \mid s \mid c \mid x \mid lx \mid \texttt{context} \\
 & & \mid\ e\ op\ e \mid uop\ e \\
 & & \mid\ \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
 & & \mid\ e\ \ldots\ e \\
 & & \mid\ \texttt{<} e\ \texttt{|} e \texttt{>} \texttt{(} lx \texttt{)} \\
 & & \mid\ \texttt{level}\ lx\texttt{,}\ \ldots\texttt{,}\ lx\ \texttt{in}\ e \\
 & & \mid\ \texttt{policy}\ lx : e\ \texttt{then}\ lv\ \texttt{in}\ e \\
 & & \mid\ \texttt{let}\ x\ \ldots x \texttt{=} e\ \texttt{in}\ e \\
 & & \mid\ \texttt{\{} x \texttt{=} e\texttt{;}\ \ldots\ \texttt{;} x \texttt{=} e\ \texttt{\}} \\
 & & \mid\ e \texttt{.} x \\
 & & \mid\ e\texttt{;}\ \ldots\ \texttt{;} e
\end{array}
$$

where $b \in Boolean,\ n \in Natural,\ s \in String,\ c \in Constant,$
and $lx, x \in Identifier,\ lv \in Level,\ op \in Op,\ uop \in UOp,\ \texttt{output} \in Outputkind$

The where-clause lists the basic value sorts of the language. They cover the same algebras in source-level Jeeves and the λ_J level, except for $Level$, which only exists in the source-level language.

For that reason, we will duplicate the formal (meta) variables between the abstract and concrete syntax and between source and target language specifications. In Definition 2.5, they are specified as concrete, lexical tokens.

2.2 Definition (basic algebraic sorts). The sorts are $Boolean$ for truth values, $Natural$ for natural numbers, $String$ for text strings, $Constant$ for constants, and $Identifier$ for variables. The $Level$ sort denotes public vs. private confidentiality levels (originally formalized by '$\top$' vs '$\bot$'), the Op sort denotes binary operations, and UOp denotes unary operations. The $Outputkind$ sort denotes the different channelings of output, here limited to `print` or `sendmail`.

2.3 Notation (Identifier naming conventions). We use x to denote a regular variable, and lx to denote a level variable.

The concrete syntax description is specified in (extended) Backus-Naur form, with regular expressions for the tokens [5]. In order to ease the implementation of the Jeeves parser, we have specifically formulated the concrete syntax as an LL(1) grammar,[1] because of the then direct applicability of the Haskell monadic parser combinator library [10].

2.4 Definition (concrete Jeeves syntax).

$$
\begin{array}{rll}
p ::= & lst^* \ pst^* & \text{(Program)}\\
lst ::= & \texttt{let}\ x\ x^* \,\texttt{=}\, e & \text{(LetStatement)}\\
pst ::= & \texttt{output}\ \texttt{\{}e\texttt{\}}\ e & \text{(OutputStatement)}\\
e ::= & lie \mid lie\ \texttt{;}\ e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{let}\ x\ x^* \,\texttt{=}\, e\ \texttt{in}\ e & \text{(Expression)}\\
 & \mid \texttt{level}\ lx\,(\,\texttt{,}\ lx)^*\ \texttt{in}\ e \mid \texttt{policy}\ lx : e\ \texttt{then}\ lv\ \texttt{in}\ e & \\
lie ::= & loe \Rightarrow loe \mid loe & \text{(LogicalImplyExpression)}\\
loe ::= & loe\ \texttt{||}\ lae \mid lae & \text{(LogicalOrExpression)}\\
lae ::= & lae\ \texttt{\&\&}\ ce \mid ce & \text{(LogicalAndExpression)}\\
ce ::= & ae \,\texttt{=}\, ae \mid ae \,\texttt{>}\, ae \mid ae \,\texttt{<}\, ae \mid ae & \text{(ComparisonExpression)}\\
ae ::= & ae + fe \mid ae - fe \mid fe & \text{(AdditiveExpression)}\\
fe ::= & fe\ pe \mid pe & \text{(FunctionExpression)}\\
pe ::= & lit \mid x \mid \texttt{context} & \text{(PrimaryExpression)}\\
 & \mid \texttt{<}ae\texttt{|}ae\texttt{>}\,\texttt{(}lx\texttt{)} \mid rec \mid pe\,\texttt{.}\,x \mid \texttt{!}pe \mid \texttt{(}e\texttt{)} & \\
lit ::= & b \mid n \mid s \mid c & \text{(Literal)}\\
rec ::= & \texttt{\{}\,xe\,(\texttt{;}\ xe)^*\,\texttt{\}} \mid \texttt{\{\}} & \text{(Record)}\\
xe ::= & x \,\texttt{=}\, pe & \text{(Field)}
\end{array}
$$

$$
\begin{array}{l}
\text{where } b \in Boolean,\ n \in Natural,\ s \in String,\ c \in Constant,\\
\text{and } lx, x \in Identifier,\ lv \in Level,\ op \in Op,\ uop \in UOp,\ \texttt{output} \in Outputkind
\end{array}
$$

To simplify where potential privacy leaks may appear in a program, we restrict the Jeeves language semantics by imposing a number of simple restrictions. Notably, that statements are only allowed at the top-level of a program. There are two types of (source-level) Jeeves statements: simple `let` statements that define the global, recursively defined binding environment, and the

[1] LL(1) grammars are context-free and parsable by LL(1) parsers: input is parsed from left to right, constructing a leftmost derivation of the sentence, using 1 lookahead token to decide on which production rule to proceed with.

`output` statements, that induce (output) side effects. Because (output) side effects represent potential privacy leaks, we have simplified matters by only allowing output statements to be stated at the end of a program, thus textually after the global binding environment has been established. Even though this is simply a syntactic decision, it supports a programmer's intuition when to let the semantics apply in this way. By only allowing recursion to appear at the top-level of a Jeeves program, we hereby simplify how and where policy (constraint) side effects can appear, in accordance with a programmer's view.

We proceed by specifying the basic algebraic sorts from Definition 2.2, as concrete lexical tokens, together with their implementation in Haskell 2.6.

2.5 Definition (Jeeves lexical tokens).

$$\begin{aligned}
b &::= \texttt{true} \mid \texttt{false} && \text{(Boolean)}\\
n &::= \text{[0-9]}^{+} && \text{(Natural)}\\
s &::= \texttt{"}\ [\neg\texttt{"\textbackslash n}]^{*}\ \texttt{"} && \text{(String)}\\
c &::= \text{[A-Z] [A-Za-z0-9]}^{*} && \text{(Constant)}\\
lx,\, x &::= \text{[a-z] [A-Za-z0-9]}^{*} && \text{(Identifier)}\\
lv &::= \texttt{top} \mid \texttt{bottom} && \text{(Level)}\\
op &::= \texttt{+} \mid \texttt{-} \mid \texttt{<} \mid \texttt{>} \mid \texttt{=} \mid \texttt{\&\&} \mid \texttt{||} \mid \texttt{=>} && \text{(BinaryOp)}\\
uop &::= \texttt{!} && \text{(UnaryOp)}\\
\texttt{output} &::= \texttt{print} \mid \texttt{sendmail} && \text{(Outputkind)}
\end{aligned}$$

2.6 Haskell (Jeeves lexical tokens). Lexical tokens are straight forwardly implemented as Haskell literals. Boolean and String literals are predefined in Haskell. Other literals are mapped to Haskell's Integer and String types.

```haskell
type Natural    = Integer
type Constant   = String
type Identifier = String
type Level      = String
type BinaryOp   = String
type UnaryOp    = String
type Outputkind = String
```

2.7 Remark. The implementation of Constant, Identifier, Level, BinaryOp, UnaryOp, and Outputkind does not really reflect the restrictions imposed by the regular expression definition in Definition 2.5. For example, by allowing constants or identifiers to start with a digit. We will instead address these restrictions by the (error) semantics.

Finally, we will re-visit the first of our canonical examples, the enforcement of a naming policy, from Example 1.4. The goal is to informally explain the overall syntactic structure of a simple Jeeves program, as a stepping stone to familiarize a programmer with the language.

2.8 Example (Jeeves name policy program).

```
1. let name =
2.  level a in
3.     policy a: !(context = alice) then bottom in < "Anonymous" | "Alice" >(a)
```

```
4. let msg = "Author is " + name

5. print {alice} msg
6. print {bob} msg
```

This program begins with a sequence of let-statements (‘`let name...`’, and ‘`let msg...`’), trailed by a sequence of print-statements (‘`print alice msg`’, and ‘`print bob msg`’). We expect the let-statements in line 1 and 4, by means of the underlying semantics, to set up a global (and recursively) defined binding environment (which we shall express as [‘`name`’ $\rightarrow \ldots$; ‘`msg`’ $\rightarrow \ldots$] in accordance with tradition). It is the print-statements, however, which are causing side effects in terms of printing the values of ‘`msg`’ in line 5 and 6. We notice that the build-up of constraints by the ‘`level a in policy a:...`’ expression in line 2 and 3, is tacitly expected to be resolved by the semantics. The program captures in many ways the essence of Jeeves' unique capability to "filter" a program outcome: a naming policy, associated with the level variable ‘`a`’, is explicitly defined in terms of a predicate ‘`n!(context = alice)`’ in line 3 (‘!’ stands for negation), where ‘`context`’ is a keyword for the implicit, designated input variable that gets set by the print statements in line 5 and 6. The value of the predicate will in turn decide how the sensitive value ‘`<"Anonymous"|"Alice">`’ evaluates to either ‘`"Anonymous"`’ or ‘`"Alice"`’. The final outcome results in ‘`msg`’ being assigned in line 4 to the result of the policy expression evaluation. To summarize, we have that the input-output function is uniquely given by the print-statements in line 5 and 6. *The input* is read from the expression, stated between the ‘{’ and the ‘}’, and assigned the designated ’`context`’ variable (here, ‘`alice`’ and ‘`bob`’). *The output* by the two print statements, however, is given by the expression trailing the curley braces (here, ‘`msg`’). For further details on the meaning of this example, we refer to Example 1.4.

In Sec. 8, we show how to run this program with the system developed in this report.

3 The λ_J syntax

In this section, we re-state the λ_J abstract syntax from the original paper [23, Figure 2], adding a (new) formulation of a λ_J program, along a (new) type of expression (thunks). We specify λ_J programs, statements, and expressions algebraically in a top-down manner, following the stylistic guidelines in Notation 1.3. We do, however, redefine the notion of a λ_J value to be a property over the expression sort, and the error primitive to be redefined from a syntactic value to a semantic entity. Finally, the error primitive is redefined from a syntactic value to a semantic entity, and the () (unit) primitive is removed completely as a value.[2] All which is necessary to maintain the role of λ_J as an intermediary language for Jeeves. The ensemble has been implemented in Haskell with code shown alongside the presentation of the concepts. The Haskell implementation of λ_J is designed as a one-to-one mapping from the λ_J syntax algebras to Haskell data types, where the basic algebraic sorts and the formal (meta) variables remain shared between the Jeeves and λ_J level, as specified in previous sections.

First, we define our notion of a λ_J program ‘p’. It is specified as a list of mutually recursive (function) bindings ‘$x = ve \ldots x = ve$’ that constitutes the static environment for evaluating the `output` statements ‘$s\ \ldots\ s$’. (It is the ‘letrec’, which semantically specifies the recursive nature of the bind-

[2]The unit primitive only appears in the E-ASSERT rule in [23, Figure 3], hiding the fact that the Jeeves translation only generates `assert` expressions which include an "in e" part [23, Figure 6]. Thus eliminating the need for a unit.

ings by its traditional meaning [9].) The $Statement$, Exp, and $ValExp$ algebraic sorts are all being defined later in this section.

3.1 Definition (abstract λ_J program syntax).

$$p \in Program ::= \ \mathsf{letrec} \quad \begin{array}{l} x{=}ve \ \ldots \ x{=}ve \end{array} \quad \mathsf{in} \quad \begin{array}{l} s \ \ldots \ s \end{array}$$

where $x \in Identifier,\ ve \in ValExp,\ s \in Statement,$ and $ValExp \subseteq Exp$

The list of bindings, $x = ve \ldots x = ve$, and statements, $s \ \ldots \ s$, are auxiliary algebraic sorts.

This definition has a straight forward implementation is Haskell:

3.2 Haskell (abstract λ_J program syntax). A program is implemented in terms of a combinator Bindings, and Statements data type. The letrec-defined environment is specifically implemented by the Binding list data type.

```haskell
data Program = P_LETREC Bindings Statements deriving (Ord,Eq)

type Bindings     = [Binding]
data Binding      = BIND Var Exp deriving (Ord,Eq)
```

The $Statement$ sort is defined as specified in the original paper [23, Figure 2], followed by is straight forward implementation:

3.3 Definition (abstract λ_J statement syntax).

$$s \in Statement ::= \ \mathsf{output}\ (\mathsf{concretize}\, e\, \mathsf{with}\, e)$$

where $e \in Exp,\ \mathsf{output} \in Outputkind$

3.4 Haskell (abstract λ_J statement syntax). The list of statements is straight forwardly implemented by the Statements list data type.

```haskell
type Statements  = [Statement]
data Statement   = CONCRETIZE_WITH Outputkind Exp Exp deriving (Ord,Eq)
```

We wish to address the issue of our introduction of thunks, and thereby our need for introducing the sub-sort $ValExp$ of Exp in Definition 3.11. Let us for a moment side-step the fact that the letrec-bindings in Definition 3.1 only are allowed to happen to value expressions ('$x = ve$') when the static binding environment is established, and instead assume that bindings are allowed to happen over all expressions ('$x = e$') as defined in Definiton 3.5. Because Jeeves, and whence λ_J, is defined to be an eager language, parsing of an expression 'e', however, may cause significant, unintended behaviour at binding time, as illustrated by the following λ_J program:

$$\begin{array}{l} \mathsf{letrec}\ \ x = (ack\ 100)\ 100 \\ \qquad \mathsf{in\ print}\ (\mathsf{concretize}\ 5\ \mathsf{with}\ 5) \end{array}$$

This program binds 'x' to an instance of the Ackermann function, even though it clearly outputs the number 5, regardless of the value of $(ack\ 100)\ 100$! The problem is that Ackermann with those arguments is a number of magnitude 10^{20000} digits![3] An eager language will cause this enormous number to be calculated at binding time, leading to a halt before any print statement has been evaluated.

The established manner to handle scope is to introduce 'thunks' as a way of "wrapping up" undesired expressions with a syntactic containment annotation. Thereby allowing binding resolution to be delayed until the correct scope is established. Precisely as prohibiting "evaluation under lamba" is a common way of "wrapping up" function evaluation. Technically, to put it on *weak head normal form*.

Because the original λ_J syntax does not allow this, we have extended the expression sort with '$thunk\ e$', and created a special subsort $ValExp$ which contains expressions on *weak head normal form*. These features will in particular show up as useful features when specifying and implementing the λ_J translation. A correct version of the above program hereafter is:

$$\begin{gathered}\mathsf{letrec}\ \ x = \mathsf{thunk}\,((ack\ 100)\ 100)\\ \mathsf{in\ print}\,(\mathsf{concretize}\,5\,\mathsf{with}\,5)\end{gathered}$$

We proceed by restating the abstract syntax according to the discussed considerations.

3.5 Definition (abstract λ_J expression syntax).

$$\begin{aligned} e \in Exp ::=\ & b \mid n \mid s \mid c \mid x \mid lx \mid \mathsf{context}\\ & \mid \lambda x.e \mid \mathsf{thunk}\,e\\ & \mid e\ op\ e \mid uop\ e\\ & \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e\\ & \mid e\ e\\ & \mid \mathsf{defer}\ lx\ \mathsf{in}\ e\\ & \mid \mathsf{assert}\ e\ \mathsf{in}\ e\\ & \mid \mathsf{let}\ x = e\ \mathsf{in}\ e\\ & \mid \mathsf{record}\ fi{:}e\ \cdots\ fi{:}e\\ & \mid e.fi \end{aligned}$$

$$\begin{gathered}\text{where } b \in Boolean,\ n \in Natural,\ s \in String,\ c \in Constant,\\ \text{and } op \in Op,\ up \in UOp,\ lx, x \in Var,\ fi \in FieldName\end{gathered}$$

Here, we have tacitly assume that the $Identifier$ sort has been partitioned into two separate namespaces: $lx,\ x \in Var$, and $fi \in FieldName$, with the obvious meaning.

3.6 Remark (empty expression). The empty record is represented by the keyword record.

3.7 Remark (defer expression). The original defer expression syntax come in two forms (with types omitted): '$\mathsf{defer}\,lx\,\{e\}\,\mathsf{default}\,v$' and '$let\,l = \mathsf{defer}\,lx\,\mathsf{default\,true}\,v\,\mathsf{in}\,e$' in Yang et al [23, Figure 2,E-DEFER] and [23, Figure 6,(TR-LEVEL)] respectively. The version we have chosen to formalize, is a modification in a couple of ways yet preserving the intended translation semantics. First, the 'default true' part is omitted from the syntax, because this contribution from the Jeeves translation is so trivial that it can be dealt with by the evaluation semantics instead *c.f.* Definition 7.36. Second, the contribution from '$\{e\}$' is none according to Yang et al [23, Figure 6,(TR-LEVEL)]. Thus, we have allowed a modified version '$\mathsf{defer}\,lx\,\mathsf{in}\,e$' as an expression and ajusted the semantics accordingly to still be in line with the intent of Yang et al [23].

[3] In comparison, the estimated age of the earth is approximately 10^{17} seconds.

*3.8 Remark (*assert *expression).* The original syntax, 'assert e', has been modified in accordance with the original translation scheme in Yang et al [23, Figure 6] to include an 'in e' part. (A fact that equally eliminates the need for the unit primitive () as originally stated in Yang et al [23, Figure 3].) These decisions render an assert expression on the form: 'assert $(e \Rightarrow (lx = b))$ in e'.

3.9 Definition (λ_J lexical tokens). Lexical tokens are the same as for Jeeves *c.f.* Definition 2.5. $Level$ ('lx') tokens are by default logical variables at the λ_J level.

3.10 Haskell (abstract λ_J expression syntax). The algebraic constructors for the Exp sort are implemented as a one-to-one map to Haskell constructors for the Exp datatype. The Op sort is implemented by the datatype Op, and UOp is implemented by UOp. The individual operations are implemented with (self-explanatory) Haskell constructors.

```haskell
data Exp = E_BOOL Bool | E_NAT Int | E_STR String | E_CONST String
         | E_VAR Var | E_CONTEXT
         | E_LAMBDA Var Exp | E_THUNK Exp
         | E_OP Op Exp Exp | E_UOP UOp Exp
         | E_IF Exp Exp Exp | E_APP Exp Exp
         | E_DEFER Var Exp | E_ASSERT Exp Exp
         | E_LET Var Exp Exp
         | E_RECORD [(FieldName,Exp)]
         | E_FIELD Exp FieldName
         deriving (Ord,Eq)

data Op = OP_PLUS | OP_MINUS | OP_LESS | OP_GREATER
        | OP_EQ | OP_AND | OP_OR | OP_IMPLY
        deriving (Ord,Eq)

data UOp = OP_NOT deriving (Ord,Eq)

data FieldName = FIELD_NAME String deriving(Ord,Eq)
data Var = VAR String deriving (Ord,Eq)
```

Finally, we need to characterize the notion of a *value expression*, among which is the notion of a thunk-expression as discussed above. As illustrated by the Ackermann program example, the problem is that "problematic" expressions might get unintentionally evaluated at compile-time instead of in a run-time scope, because the language is eager. To make sure that only expressions that are "safe" to bind in Definition 3.1 are in fact those allowed in the static binding environment, we introduce the notion of a value expression ('ve') as an expression on weak head normal form. To summarize, such expressions in λ_J may, as expected, take one of three forms:

- constant expressions (literals or records of values),
- non-constant functions ('$\lambda x.e$'), or
- constant functions ('thunk e').

To be precise, we specify an auxiliary value sort $ValExp \subseteq Exp$ with the purpose of syntactically capturing those sets of expressions, followed by its Haskell implementation:

3.11 Definition (value expressions).

$$ve \in ValExp \ ::= \ b \mid n \mid s \mid c \mid \lambda x.e \mid \mathsf{thunk}\, e \mid \mathsf{record}\, fi_1 : ve_1 \ldots fi_m : ve_m$$
$$\text{where } m \geq 1$$

3.12 Haskell (value expressions). The λ_J value property is straight forwardly implemented as a Haskell predicate isValue over the Exp datatype.

```haskell
isValue (E_BOOL _)      = True
isValue (E_NAT _)       = True
isValue (E_STR _)       = True
isValue (E_CONST _)     = True
isValue (E_LAMBDA _ _) = True
isValue (E_THUNK _)     = True
isValue (E_RECORD xes) = and [isValue e | (_,e)←xes]
isValue _               = False
```

4 The λ_J translation

In this section, we formally present a syntax directed translation of the concrete Jeeves syntax to λ_J , alongside its Haskell implementation. The translation follows the original outline in Yang et al [23, Fig. 6] on critical syntax parts, but has been extended to accomodate modifications as accounted for in Section A, 2, and 3. Specifically, we have added a translation from a Jeeves program to our notion of a λ_J program.

The translation is formalized as a *derivation*, marked by $[\![\_]\!]$, over the program, expression, and token sorts. A derivation is a particular simple form of compositional translations that is characterized by the fact that syntax cannot be re-used, and side-conditions cannot be stated, which makes them particularly easy to reason about termination, and straightforward to implement.

The Haskell implementation is given as a set of *Jeeves parsers*, which builds abstract λ_J syntax trees in accordance with the abstract syntax outlined in Section 3. The parsers are implemented using the Haskell monadic parser combinator library [10], which is also included in Appendix B.2.

4.1 Definition (translation of Jeeves program).

$$\left[\!\!\left[\begin{array}{l} \texttt{let } f_1\; x_{11} \ldots x_{1n_1} = e_1 \\ \vdots \\ \texttt{let } f_m\; x_{m1} \ldots x_{mn_m} = e_m \\ \texttt{output}_1\ \{e'_1\}\ e''_1 \\ \vdots \\ \texttt{output}_k\ \{e'_k\}\ e''_k \end{array}\right]\!\!\right] = \begin{array}{l} \mathsf{letrec}\ f_1 = e'''_1 \\ \quad\ \ldots \\ \quad\ f_m = e'''_m \\ \mathsf{in\ output}_1\ (\mathsf{concretize}\ [\![e''_1]\!]\ \mathsf{with}\ [\![e'_1]\!]) \\ \quad\ \ldots \\ \quad\ \mathsf{output}_k\ (\mathsf{concretize}\ [\![e''_k]\!]\ \mathsf{with}\ [\![e'_k]\!]) \end{array}$$

where

$$e'''_i = \begin{cases} \mathsf{thunk}\ [\![e_i]\!] & \text{if } n_i = 0 \ \wedge \ [\![\, e_i \,]\!] \notin ValExp \\ \lambda x_{i1}. \ldots \lambda x_{in_i}.[\![e_i]\!] & \text{otherwise} \end{cases}$$
$$1 \leq i \leq m, \ m \in \mathbb{N}, \ n_i \in \mathbb{N}_0$$

and

$$k, m \in \mathbb{N}, \ f, x \in Var, \ e, e', e'', e''' \in Exp, \ \mathsf{output} \in Outputkind$$

Using the introduced notation, we begin by explaining the specifics of a constant function (that is a function with no function arguments):

4.2 Remark (constant function). We tacitly assume that given $m \in \mathbb{N}$ functions, originally defined by m let-statements, and given some function 'f_i, $1 \leq i \leq m$', we have that '$n_i = 0$', which corresponds to 'f_i' being a constant function. In particular it entails that '$e_i''' = [\![e_i]\!]$', where the expression-translation '$[\![e_i]\!]$' is assumed to be some λ_J expression.

The where-clause specifies the shape of the translated expressions, symbolized by 'e_i'''', as it is statically bound in the recursive (function) binding environment by the equation '$f_i = e_i'''$' (for some i where $m \in \mathbb{N}$, $1 \leq i \leq m$). A problematic scoping situation might occur during translation, when 'f_i' defines a constant function as discussed in detail in Section 3. Because 'e_i'''' may equal any expression form, we have to confine any impending static evaluation by wrapping all non-value expressions with a 'thunk'. It means vice versa, that constant functions which *are* in fact value expressions can be safely bound:

4.3 Remark (constant function translation). If for some $m \in \mathbb{N}$, $1 \leq i \leq m$ we have $n_i = 0$ (no function arguments), and $[\![e_i]\!] \in ValExp$ (value expression), then the where-clause of the translation rule entails $e_i''' = [\![e_i]\!]$ (function is a constant value expression).

From Definition 3.11 follows immediately the following invariant:

4.4 Lemma (binding environment invariant). *The right hand side of the letrec-function-bindings are all value expressions, i.e., for some $m \in \mathbb{N}$ we have*

$$\forall i \in \mathbb{N},\, 1 \leq i \leq m,\, n_i \in \mathbb{N}_0 \,:\, e_i''' \in ValExp$$

.

4.5 Haskell (translation of Jeeves program).

```haskell
programParser :: FreshVars → Parser Program
programParser xs = do recb ← manyParser recbindParser xs1 success
                      psts ← manyParser outputstatParser xs2 success
                      return (P_LETREC recb psts)
  where ~(xs1,xs2) = splitVars xs

recbindParser :: FreshVars → Parser Binding
recbindParser xs = do token (word "let")
                      f ← token ident
                      e ← argumentAndExpThunkParser xs
                      optional (token (word ";"))
                      return (BIND (VAR f) e)

argumentAndExpThunkParser :: FreshVars → Parser Exp
argumentAndExpThunkParser xs = do vs ←many (token ident)  -- accumulates function parameters
                                  token (word "=")
                                  e ← expParser xs
                                  if ((null vs) && not (isValue e))
                                    then return (E_THUNK e)  -- constant, non-value expression
```

```
                                        else return (foldr f e vs)  -- guaranteed to be a value by the guard
  where
    f v1 e1 = E_LAMBDA (VAR v1) e1

outputstatParser :: FreshVars → Parser Statement
outputstatParser xs = do output ← outputToken
                         token (word "{")
                         e1 ← expParser xs1      -- should evaluate to concrete value
                         token (word "}")
                         e2 ← expParser xs2
                         optional (token (word ";"))
                         return (CONCRETIZE_WITH output e2 e1)
  where ~(xs1,xs2) = splitVars xs
```

The expression translation follows the concrete expression syntax structure in Definition 2.4, from which we have tacitly adopted all algebraic specifications.

4.6 Definition (translation of Jeeves expressions).

$$
\begin{aligned}
[\![\, e_1 ;\ \ldots\ e_n ;\ e \,]\!] &= \mathsf{let}\ x_1 = [\![e_1]\!]\ \mathtt{in}\ \ldots\ \mathsf{let}\ x_n = [\![e_n]\!]\ \mathtt{in}\ [\![e]\!] \\
&\quad \text{where } x_1 \ldots x_n \text{ fresh, } 0 \leq n \\
[\![\, \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 \,]\!] &= \mathsf{if}\ [\![e_1]\!]\ \mathsf{then}\ [\![e_2]\!]\ \mathsf{else}\ [\![e_3]\!] \\
[\![\, \mathtt{let}\ x\ x_1 \ldots x_n = e_1\ \mathtt{in}\ e_2 \,]\!] &= \mathsf{let}\ x = \lambda\, x_1\ \ldots \lambda\, x_n\, .\, [\![e_1]\!]\ \mathtt{in}\ [\![e_2]\!] \\
&\quad \text{where } 0 \leq n \\
[\![\, \mathtt{level}\ lx_1\ ,\ \ldots\ ,\ lx_n\ \mathtt{in}\ e \,]\!] &= \mathsf{defer}\ lx_1\ \mathsf{in}\ \ldots\ \mathsf{in}\ \mathsf{defer}\ lx_n\ \mathsf{in}\ [\![e]\!] \\
&\quad \text{where } 1 \leq n \\
[\![\, \mathtt{policy}\ lx : e_1\ \mathtt{then}\ lv\ \mathtt{in}\ e_2 \,]\!] &= \mathsf{assert}\ ([\![e_1]\!] \Rightarrow (lx = [\![lv]\!]))\ \mathsf{in}\ [\![e_2]\!] \\
[\![\, e\ op\ e \,]\!] &= [\![e]\!]\ op\ [\![e]\!] \\
[\![\, fe\ pe \,]\!] &= [\![fe]\!]\ [\![pe]\!] \\
[\![\, \mathtt{context} \,]\!] &= \mathsf{context} \\
[\![\, \mathtt{<} ae_1 \,\mathtt{|}\, ae_2 \mathtt{>}\ (lx) \,]\!] &= \mathsf{if}\ lx\ \mathsf{then}\ [\![ae_2]\!]\ \mathsf{else}\ [\![ae_1]\!] \\
[\![\, \{\, x_1 = e_1 ; \ldots ; x_n = e_n\ \} \,]\!] &= \mathsf{record}\ x_1 = [\![e_1]\!]\ \ldots\ x_n = [\![e_n]\!] \\
&\quad \text{where } 0 \leq n \\
[\![\, pe\ .\ x \,]\!] &= [\![\, pe]\!]\ .\ x \\
[\![\ !\ pe \,]\!] &= !\,[\![\, pe]\!] \\
[\![\, (e) \,]\!] &= [\![e]\!] \\
[\![\, lit \,]\!] &= lit
\end{aligned}
$$

4.7 Remark (simple expression sequence translation). An expression sequence 'e' with only one expression is described by index '$n = 0$.

4.8 Remark (simple let expression translation). A let expession '`let` $x = e_1$ `in` e_2' with only one variable binding is described by index '$n = 0$'.

4.9 Remark (empty record translation). We represent an empty record by the index '$n = 0$', and its translation by the keyword record.

The expression translation is implemented as a *Jeeves expression parser* that builds abstract λ_J expression syntax trees, *c.f.*, Definition 3.5. Recall that all parsers are implemented using the Haskell monadic parser combinator library [10], which is explicitly included in Appendix B.2.

4.10 Haskell (translation of Jeeves expressions).

```
expParser :: FreshVars → Parser Exp
expParser xs = do es ← manyParser1 semiUnitParser xs1 (token (word ";"))
                  return (snd (foldr1 f (zip xs2 es)))
  where
    f (x1,e1) (x2,e2) = (x1, E_LET x1 e1 e2)
    (xs1,xs2) = splitVars xs
    semiUnitParser xs = ifParser xs +++ letParser xs  +++ levelParser xs +++ policyParser xs
         +++ logicalImplyParser xs

ifParser xs = do token (word "if")
                 e1 ← expParser xs1
                 token (word "then")
                 e2 ← expParser xs2
                 token (word "else")
                 e3 ← expParser xs3
                 return (E_IF e1 e2 e3)
  where ~(xs1,xs2,xs3) = splitVars3 xs

letParser xs = do token (word "let")
                  x ← token ident
                  xse1 ← argumentAndExpParser xs1
                  token (word "in")
                  e2 ← expParser xs2
                  return (E_LET (VAR x) xse1 e2)
  where ~(xs1,xs2) = splitVars xs

argumentAndExpParser xs = do vs ←many (token ident)
                             token (word "=")
                             e ← expParser xs
                             return (foldr f e vs)
  where
    f v1 e1 = E_LAMBDA (VAR v1) e1

levelParser xs = do token (word "level")
                    lx ← levelIdent
                    lxs ← many commaTokenLevelIdent
                    token (word "in")
                    e ← expParser xs1
                    return (foldr f e (lx:lxs))
  where
    commaTokenLevelIdent = do token (word ",")
                              lx ← levelIdent
                              return lx
```

```
    f lx e = E_DEFER lx e
    ~(xs1, lys) = splitVars xs

policyParser xs = do token (word "policy")
                     lx ← levelIdent
                     token (word ":")
                     e1 ← expParser xs1
                     token (word "then")
                     lv ← levelToken
                     token (word "in")
                     e2 ← expParser xs2
                     return (E_ASSERT (E_OP OP_IMPLY e1 (E_OP OP_EQ (E_VAR lx) lv))
                         e2)
  where
    ~(xs1,xs2) = splitVars xs

logicalImplyParser xs = do loe ← logicalOrParser xs1
                           loes ← optional ( logicalImplyTailParser xs2)
                           return ( foldl f loe loes)
  where
    f loe1 loe2 = E_OP OP_IMPLY loe1 loe2
    ~(xs1,xs2) = splitVars xs

logicalImplyTailParser xs = do token (word "⇒")
                               loe ← logicalOrParser xs
                               return loe

logicalOrParser xs = do lae ← logicalAndParser xs1
                        laes ← many ( logicalOrTailParser xs2)
                        return ( foldl f lae laes)
  where
    f lae1 lae2 = E_OP OP_OR lae1 lae2
    ~(xs1,xs2) = splitVars xs

logicalOrTailParser xs = do token (word "||")
                            lae ← logicalAndParser xs
                            return lae

logicalAndParser xs = do ce ← compareParser xs1
                         ces ← many (logicalAndTailParser xs2)
                         return ( foldl f ce ces)
  where
    f ce1 ce2 = E_OP OP_AND ce1 ce2
    ~(xs1,xs2) = splitVars xs

logicalAndTailParser xs = do token (word "&&")
                             ce ← compareParser xs
                             return ce
```

```
compareParser xs = do ae ← additiveParser xs1
                      copae ← optional (compareTailParser xs2)
                      if (null copae) then return ae
                        else return (E_OP (fst (head copae)) ae (snd (head copae)))
  where ~(xs1,xs2) = splitVars xs

compareTailParser :: FreshVars → Parser (Op,Exp)
compareTailParser xs = do cop ← compareOperator
                          ae ← additiveParser xs
                          return (cop,ae)

compareOperator = wordToken "=" OP_EQ +++ wordToken "<" OP_LESS +++ wordToken ">"
    OP_GREATER

additiveParser xs = (do fe ← functionParser xs1
                        aopae ← optional ( additiveTailParser xs2)
                        if (null aopae) then return fe else return ((head aopae) fe))
                    +++
                    (do aopae ← additiveTailParser xs
                        return (aopae (E_NAT 0)))
  where ~(xs1,xs2) = splitVars xs

additiveTailParser :: FreshVars → Parser (Exp → Exp)
additiveTailParser xs = do aop ← additiveOperator
                           fe ← functionParser xs1
                           aopae ← optional ( additiveTailParser xs2)
                           if (null aopae) then return (λx → E_OP aop x fe)
                             else return (λx → (head aopae) (E_OP aop x fe))
  where ~(xs1,xs2) = splitVars xs

additiveOperator = wordToken "+" OP_PLUS +++ wordToken "-" OP_MINUS

functionParser xs = do pe ← primaryParser xs1
                       pes ← many (primaryParser xs2)
                       return ( foldl E_APP pe pes)
  where ~(xs1,xs2) = splitVars xs


primaryParser xs = do pe ← primaryTailParser xs
                      fis ← fLookup
                      return ( foldl E_FIELD pe fis)

fLookup :: Parser [FieldName]
fLookup = many (do word "."
                   fi ← ident
                   return (FIELD_NAME fi))
```

```
primaryTailParser xs = literalParser xs +++ regularIdent +++
                       wordToken "context" E_CONTEXT +++
                       sensiValParser xs +++ recordParser xs +++
                       unaryParser xs +++ groupingParser xs

sensiValParser xs = do token (word "<")
                       e1 ← additiveParser xs1
                       token (word "|")
                       e2 ← additiveParser xs2
                       token (word ">")
                       token (word "(")
                       lx ← levelIdent
                       token (word ")")
                       return (E_IF (E_VAR lx) e2 e1)
  where ~(xs1,xs2) = splitVars xs

recordParser xs = do token (word "{" )
                     fies ← manyParser fieldParser xs (token (word ";"))
                     token (word "}" )
                     return (E_RECORD fies)

fieldParser :: FreshVars → Parser (FieldName,Exp)
fieldParser xs = do fi ← token ident
                    token (word "=")
                    pe ← primaryParser xs
                    return (FIELD_NAME fi,pe)

unaryParser xs = do token (word "!")
                    pe ← primaryParser xs
                    return (E_UOP OP_NOT pe)

groupingParser xs = do token (word "(")
                       e ← expParser xs
                       token (word ")")
                       return e
```

4.11 Definition (translation of Jeeves lexical tokens). The Jeeves lexical tokens, specified in Definition 2.5, formally carries over to λ_J as the identical token sets, except for $Level$ tokens, which maps to $Boolean$ in the following way:

$$[\![\texttt{top}]\!] = \mathsf{true} \qquad [\![\texttt{bottom}]\!] = \mathsf{false}$$

4.12 Haskell (translation of Jeeves lexical tokens).

The identity mapping of the Jeeves token set (except for level-tokens) to λ_J token set, is implemented by letting the parser "build" the equivalent implementation of those tokens (Haskell 2.6) directly as represented in λ_J (Haskell 3.10). $Level$ tokens, however, are represented as boolean expressions *c.f.* Definition 4.11.

For reasons of efficiency, we do distinguish between the representation of "regular" variables (‘x’) and "level" variables (‘lx’) in our implementation, except when translating sensitive values.

Notice the definition of a "helper", the literalParser, which parses Jeeves literals directly.

```
literalParser  xs = booleanToken +++ naturalToken +++ stringToken +++ constantToken

booleanToken = wordToken "true" (E_BOOL True)
               +++ wordToken "false" (E_BOOL False)

naturalToken = do n ← token nat
                  return (E_NAT n)

stringToken = do cs ← token string
                 return (E_STR cs)

constantToken = do cs ← token constant
                   return (E_CONST cs)

regularIdent :: Parser Exp
regularIdent = do x ← token ident
                  return (E_VAR (VAR x))

levelIdent :: Parser Var
levelIdent = do lx ← token ident
                return (VAR lx)

levelToken :: Parser Exp
levelToken = wordToken "top" (E_BOOL True) +++ wordToken "bottom" (E_BOOL False)

outputToken = token (word "print") +++ token (word "sendmail")
```

We exploit that Haskell is a lazy language that permits cyclic data definitions to maintain an infinite supply of fresh variable names (a need reflected by Definition 4.6 and Definition 7.36).

4.13 Haskell (fresh variables). We implement an infinite supply of distinct variables (and infinite, disjoined, derived sublists) by the variable generator iterate. (The definition of iterate is in fact cyclic/infinite in its definition.)

```
type FreshVars = [Var]

vars :: FreshVars
vars = map (\n→VAR ("x"++show n)) (iterate (\n→n+1) 1)

splitVars :: FreshVars → (FreshVars,FreshVars)
splitVars xs = (odds xs, evens xs) where
  odds ~(x:xs) = x : evens xs
  evens ~(x:xs) = odds xs

splitVars3 :: FreshVars → (FreshVars,FreshVars,FreshVars)
splitVars3 vs = (xs, ys, zs) where
   (xs,yzs) = splitVars vs
   (ys,zs) = splitVars yzs
```

Finally, we present a formal translation of the first of our canonical examples: the Jeeves naming policy program from Example 1.4 and 2.8.

4.14 Example (Name policy program translation).

$$\left[\!\!\left[\begin{array}{l}\texttt{let}\ name = \texttt{level}\ a\ in\ \texttt{policy}\ a : !(\texttt{context} = alice)\ \texttt{then}\ bottom\ \texttt{in}\ \texttt{<}"Anonymous"\ \texttt{|}\ "Alice"\texttt{>}(a)\ ]\!] \\ \texttt{let}\ msg = "Author\ is\ " + name \\ \texttt{print alice}\ msg \\ \texttt{print bob}\ msg\end{array}\right]\!\!\right] =$$

$$\begin{array}{l}\mathsf{letrec}\ name{=}\mathsf{thunk}(\mathsf{defer}\ a\ \mathsf{in}\ (\mathsf{assert}\ (!(\mathsf{context} = alice) => (a = false))\ in\ [\![\texttt{<}"Anonymous"\ \texttt{|}\ "Alice"\texttt{>}(a)]\!])) \\ \qquad msg{=}\mathsf{thunk}\ ("Author\ is\ " + name) \\ \quad \mathsf{in}\ \mathsf{print}\ (\mathsf{concretize}\ msg\ \mathsf{with}\ alice) \\ \qquad \mathsf{print}\ (\mathsf{concretize}\ msg\ \mathsf{with}\ bob)\end{array}$$

where

$$[\![\texttt{<}"Anonymous"\ \texttt{|}\ "Alice"\texttt{>}(a)]\!] \quad = \mathsf{if}\ a\ \mathsf{then}\ "Alice"\ \mathsf{else}\ "Anonymous"$$

5 Scoping and symbolic normal forms

In this section we specify the notions of scope and symbolic normal forms of λ_J for use in later sections. According to Yang et al [23, Figure 3], dynamic expression evaluation generally speaking happens in 3 consecutive steps:

1. reduction all the way to temporary *normal form* that may still contain dynamic, unresolved symbolic sub-expressions and constraints, followed by
2. *constraint resolution*, which resolves the consequences of knowing the value of the input variable "context", to find a solution to the program constraint set, and finally,
3. completing the reduction of the temporary normal forms, instantiated with the constraint solution.

The semantic set of temporary normal forms, which are denoted symbolic normal forms in accordance with Yang et al [23, Figure 2], is specified by the algebraic $Value$ sort in Definition 5.1. Depending on whether they contain unresolved residues, they are either categorized as *symbolic values* or *concrete values*. In order to semantically reflect lexical scoping during expression reduction, we have added the notion of a *closure* compared to [23, Fig. 2]). Generally speaking, a closure consists of a *function expression*, constant or non-constant, together with an *environment* component ρ, which holds the set of (static) variable bindings of that expression. In λ_J, such closures take the form: $(\mathsf{thunk}\ e, \rho)$, $(\lambda x.e, \rho)$. We define closures as concrete (symbolic) normal forms, *i.e.*, as concrete values of the $Value$ sort.

In the remainer of this section we formally present the symbolic normal forms followed by a specification of the static λ_J binding environment, all in tandem with their Haskell implementations. The former specification is presented as an algebraic specification in Definition 5.1, the latter as as a partial domain function in Definition 5.5.

5.1 Definition (symbolic normal forms).

$$
\begin{aligned}
\upsilon \in Value &::= \kappa \mid \sigma \\
\kappa \in ConcreteValue &::= b \mid n \mid s \mid c \mid \mathsf{error} \\
&\mid (\lambda x.e, \rho) \mid (\mathsf{thunk}\, e, \rho) \\
&\mid \mathsf{record}\ x{:}\kappa \cdots x{:}\kappa \\
\sigma \in SymbolicValue &::= x \mid lx \mid \mathsf{context} \mid \sigma\,.\,x \\
&\mid \sigma\ op\ \upsilon \mid \upsilon\ op\ \sigma \mid uop\ \sigma \\
&\mid \mathsf{if}\ \sigma\ \mathsf{then}\ \upsilon\ \mathsf{else}\ \upsilon \\
&\mid \mathsf{record}\ x{:}\sigma\ x{:}\upsilon \cdots x{:}\upsilon \\
&\mid \mathsf{record}\ x{:}\upsilon\ x{:}\sigma \cdots x{:}\upsilon \\
&\ \vdots \\
&\mid \mathsf{record}\ x{:}\upsilon\ x{:}\upsilon \cdots x{:}\sigma
\end{aligned}
$$

$$
\text{where } b \in Boolean, n \in Natural,\ s \in String,\ c \in Constant, \\
\text{and } x \in Identifier, \rho \in Environment.
$$

5.2 Remark (error normal form). Following Yang et al [23, Fig. 2], we have added error as a concrete normal form to reflect a semantically erroneous evaluation state.

5.3 Remark (record normal forms). We have added two distinct normal forms of the record data structures. A record where all fields are on concrete normal form (κ) is itself on concrete normal form (κ). A record where "at least" one field is on symbolic normal form (σ) is on symbolic normal form (σ).

5.4 Haskell (symbolic normal forms). The algebraic $Value$ constructors for the $Value$ sort are implemented as Haskell constructors for the Value datatype. The distinction between concrete and symbolic is implemented by the predicates isConcrete and isSymbolic over $Value$.

```haskell
data Value = -- Concrete values
        V_BOOL Bool | V_NAT Int | V_STR String | V_CONST String | V_ERROR
      | V_LAMBDA Var Exp Environment | V_THUNK Exp Environment
      | V_RECORD [(FieldName,Value)]
        -- Symbolic values
      | V_VAR Var | V_CONTEXT
      | V_OP Op Value Value | V_UOP UOp Value
      | V_IF Value Value Value | V_FIELD Value FieldName
      deriving (Ord,Eq)

isConcrete (V_BOOL _)       = True
isConcrete (V_NAT _)        = True
isConcrete (V_STR _)        = True
isConcrete (V_CONST _)      = True
isConcrete (V_ERROR)        = True
isConcrete (V_LAMBDA _ _ _) = True
isConcrete (V_THUNK _ _)    = True
isConcrete (V_RECORD xvs) = all (λb→b) [isConcrete v | (_,v) ← xvs]
```

```
isConcrete _                  = False

isSymbolic v = not (isConcrete v)
```

5.5 Definition (static binding environment). The concept of a static binding environment ρ is formalized in terms of new semantic meta-notation on λ_J variables and values:

- ρ denotes an *environment* that maps variables to (constant or symbolic) values,
- $\rho[x \mapsto \upsilon]$ denotes an environment obtained by extending the environment ρ with the map x to υ, and
- $\rho(x)$ denotes the value obtained by looking up x in the environment.

Environment ρ is recursively defined as a *partial domain function c.f.* Schmidt [20]:

$$\rho : \text{variables} \rightarrow Value_{\perp}$$

$$\text{For all } y \in \text{DOM}(\rho[x \mapsto \upsilon]) :$$

$$\rho[x \mapsto \upsilon](y) =_{\text{def}} \begin{cases} \upsilon & \text{if } y = x \\ \rho(y) & \text{if } y \neq x \end{cases}$$

$$\epsilon(y) =_{\text{def}} \lambda y.\perp$$

where ϵ denotes the empty environment, and the co-domain $Value_{\perp}$ is the (lifted) domain of semantic values.

5.6 Haskell (static binding environment). We use standard Haskell maps to implement the static binding environment in a straight forward manner.

```
type Environment = Map Var Value
```

- $\rho(x)$ is implemented by rho!x
- $\rho[x \mapsto v]$ is implemented by insert x v rho
- ϵ, aka $\lambda y.\perp$, is implemented by empty

6 The constraint environment

In this section, we describe the constraint environment which is created at the λ_J-level during program execution, in accordance with Yang et al [23, Fig. 3]. The ensemble of constraints has been defined as an additional component to the (static) binding environment of the dynamic λ_J semantics. As mentioned in the three step description of Section 5, the first part of a λ_J-evaluation causes constraints to be accumulated as the privacy enforcing expressions get evaluated, followed by a constraint resolution step, conditioned by the known value of the input. The actual constraint resolution is side stepped in the original semantics by Yang et al [23, Fig. 3], and simply reduced to the question of whether there exists a solution which solves the constraint set or not. Constraint programming systems in fact combines a constraint solver and a search engine in a very (monadic) flexible way as described by others [21]. In this report, however, we simply analyse the monadic structure of the constraint set semantics.

A *constraint environment* is divided into two base sets of constraints: the *current set of constraints* denoted by the algebraic Σ sort (hard constraints), and the *constraints on default values* for logical variables, denoted by the algebraic Δ sort (soft constraints), following standard constraint programming conventions [14, 18].

The specification of the hard constraints, Σ, is a result of constraints build up in connection with a defer and assert expression evaluation, *c.f.* Yang et al [23, Fig. 3,(E-DEFER),(E-ASSERT)] as "*the set of constraints that must hold for all derived outputs*". An assert expression is specified by '$\mathsf{assert}\ e_1\ \mathsf{in}\ e_2$', where '$e_1$' is a logical expression by which privacy policies get introduced *c.f.* Yang et al [23, Fig. 6,(T-POLICY)] as hard constraints. The extension of Σ with privacy policies 'e_1' is reflected by the (E-ASSETCONSTRAINT) and (E-ASSERT) rule. The extensions have the form '$\mathcal{G} \Rightarrow \upsilon_{e_1}$', where '$\upsilon_{e_1}$' is the result value from evaluating 'e_1', and '$\mathcal{G}$' called the path condition is explained below. With the modifications and assumptions in Remark 3.7, a defer expression is specified by '$\mathsf{defer}\ lx\ \mathsf{in}\ e$', where '$\{\upsilon\}$' in the original syntax is left unspecified by the translation [23, Fig. 6,(TR-LEVEL),Fig. 3,(E-DEFER)]. In this syntax form, a defer expression merely has become a reflection of the introduction of level variables *c.f.* [23, Fig. 3,(E-DEFER)]. The extension of Σ thus becomes reflected by the logic expression '$\mathcal{G} \Rightarrow [x \mapsto x']$'. The ($\alpha$) *renaming* '$[x \mapsto x']$' of '$x$' with a fresh (logical) variable 'x'', follows from the fact that *the constraint sets have no notion of scope*. Thus, all logical variable names must be declared as globally unique.

The specifications of the soft constraints, Δ, is another result of constraint build up in connection with a defer expression evaluation, as described by Yang et al [23, Fig. 3,(E–DEFER)] as "*the constraints only used if consistent with the resulting logical environment*". This build up, however, is concerned with any logical constraints imposed directly on the variables in terms of default values, etc. As explained in Remark 3.7, we tacitly assume the logical 'x'' variable to take the default value 'true' during translation according to Yang et al [23, Fig. 6], something which is directly reflected in Definition 7.36, as well as in the Δ specification in Definition 6.1. Since hard and soft constraints are extended in tandem *c.f.* Yang et al [23, Fig. 3,(E-DEFER)], we tacitly assume the default constraint is only imposed on a globally unique (fresh) variable name which we denote 'x''. Because we have introduced an additional lexical scoping mechanism ('ρ') in our formalizations, we will handle renaming directly at the scoping level *c.f.* Definition 7.36, *i.e.*, with '$\rho[x \mapsto x']$' alone. This simplifies the specification of hard constraints and soft constraints as described by Definition 6.1.

A *path condition* consists of a conjunction of symbolic values and negated symbolic values, which is used to describe the trail (or path) of symbolic (unresolved) assumptions conditioning some expression evaluation. The only place during expression evaluation where the path condition is extended, *c.f.* Definition 7.32, is when a conditional expression in the style

$$\text{`}\mathsf{if}\ \sigma_1\ \mathsf{then}\ e_2\ \mathsf{else}\ (\mathsf{if}\ \sigma_1'\ \mathsf{then}\ e_2'\ \mathsf{else}\ e_3')\text{'}$$

is evaluated. In this case, the conditions are symbolic values, which will depend on the constraint resolution later to be resolved. There are thus two possible ways a symbolic evaluation of this if-expression can take place. If 'σ_1' is assumed to become true (the 'e_2' is evaluated), or if '$\neg\sigma_1$' is assumed to become true (the '$\mathsf{if}\ \sigma_1'\ \mathsf{then}\ e_2'\ \mathsf{else}\ e_3'$' is evaluated). The path condition simply keeps track of which assumptions have been made by making a conjunction of all such presumed conditions prior to an evaluation. In our example, we thus have that the path condition '$\neg\sigma_1 \wedge \sigma_1'$' holds prior to '$e_2'$' evaluation. In Definition 6.1, we specify a path condition this way and denote it $\mathcal{G}$. It is defined as an element of the algebraic $PathCondition$ sort, together with the algebraic notation for the constraint environment, Σ (hard constraints), and Δ (soft constraints).

6.1 Definition (hard constraints, soft constraints, and path condition).

$$\Sigma = \mathcal{P}(\mathcal{G} \Rightarrow v)$$
$$\Delta = \mathcal{P}(\mathcal{G} \Rightarrow x = v)$$
$$\mathcal{G} \in PathCondition ::= \sigma \mid \neg\sigma \mid \mathcal{G} \wedge \mathcal{G}$$

where $x \in Identifier$, $v \in Value$, $\sigma \in SymbolicValue$.

$\mathcal{P}$ denotes the powerset in accordance with usual mathematical convention.

6.2 Remark (default theory property). The pair (Δ, Σ) logically defines a (super-normal) default theory, where Δ is a set of default rules (soft constraints), and Σ is a set of first-order formulas (hard constraints) [1], [19].

The Haskell implementation of Σ and Δ are given straightforwardly as relational lists. The relations are established as lists of pairs and lists of triplets, respectively. A relation '$\mathcal{G} \Rightarrow v$' is thus implemented by the data type (PathCondition,Value), and '$\mathcal{G} \Rightarrow\ x = v$' is implemented by the data type (PathCondition,Var,Value). The Haskell implementation of a path condition is also given as a list. This is a list of Haskell representations of formulas or negated formulas which are presumed to hold during some specific expression evaluation.

6.3 Haskell (hard constraints, soft constraints, and path condition).

```
data Sigma = SIGMA [(PathCondition,Value)]
emptySigma = SIGMA []
unitSigma g v = SIGMA [(g,v)]
unionSigma (SIGMA map1) (SIGMA map2) = SIGMA (map1++map2)

data Delta = DELTA [(PathCondition,Var,Value)]
emptyDelta = DELTA []
unitDelta g (x,v) = DELTA [(g,x,v)]
unionDelta (DELTA map1) (DELTA map2) = DELTA (map1++map2)

data PathCondition = P_COND [Formula] deriving (Ord,Eq)
emptyPath = P_COND []

data Formula = F_IS Value
             | F_NOT Value
             deriving (Ord,Eq)

formulaConjunction f (P_COND fs) = P_COND (f:fs)
```

We design the Haskell implementation of the constraint sets to explicitly restrict modifications to *extensions* with new constraints, because the evaluation rules (in the following section) only extend. To this end, we implement the constraint environment in Haskell by Constraints a, a *monad* over Sigma and Delta. We recall that a monad in Haskell is represented by a type class with two operators, return and bind ($\gg=$) [22]. We implement two instances on the monad, unitSigmaConstraints and unitDeltaConstraints. The goal of these instances is to update /reset Sigma and Delta respectively.

6.4 Haskell (constraint environment).

```
-- Monadic notation...
data Constraints a = CONSTRAINTS Sigma Delta a
instance Monad Constraints where
  return v = CONSTRAINTS emptySigma emptyDelta v -- the trivial monad, returning value v
  (CONSTRAINTS sigma1 delta1 v1) >>= f =        -- the sequencing of two instances
    CONSTRAINTS (unionSigma sigma1 sigma2) (unionDelta delta1 delta2) v2
      where (CONSTRAINTS sigma2 delta2 v2) = f v1

unitSigmaConstraints :: PathCondition → Value → Constraints Value
unitSigmaConstraints g v = CONSTRAINTS (unitSigma g v) emptyDelta V_ERROR

unitDeltaConstraints :: PathCondition → Var → Value → Constraints Value
unitDeltaConstraints g x v = CONSTRAINTS emptySigma (unitDelta g (x,v)) V_ERROR
```

6.5 Remark (constraint environment updates). From the evaluation semantics in Yang et al [23, Fig. 3,(E-DEFER),(E-ASSERT)] we observe that the only semantic (expression) rules that potentially will affect the constraint monad directly are those concerning the *privacy policy rules*, *i.e.*, assert (when policy constraints are being semantically enforced), and defer (when confidentiality levels are being semantically differentiated/deferred) at the λ_J-level.

7 The λ_J evaluation semantics

In this section we specify the dynamic λ_J semantics, which implements Jeeves as an eager constraint functional language. The specification of the evaluation engine follows the original idea by Yang et al [23, Fig. 3], but differs on a number of issues. Most significantly, we have reformulated the semantics as a *compositional*, *environment-based*, *big step semantics*, as opposed to the original *non-compositional*, *substitution-based*, *small-step semantic* formulation [23, Fig. 3]. Primarily, in order to enhance the ability to proof semantical statements, because proofs then can be carried inductively over the height of the proof trees (something which breaks down in general when substitution into subterms is allowed like in the original λ_J semantics). As something new, we have added a formal notion of a Jeeves, aka a λ_J program evaluation. Finally, we have added the notion of lexical variable scoping to manage static bindings.[4] This has been done by enhancing the semantics with a (new) binding environment feature (ρ and closures) as discussed in Section 5. The Haskell implementation is presented alongside each individual evaluation rule.

We begin by formalizing three peripheral semantic λ_J concepts needed to proceed with the actual evaluation semantics presentation. The *input-output domain*, the final set of *solution constrains* to be resolved, and the *runtime (side) effects* from running a λ_J program. We then proceed by a reformalization of the dynamic semantics as a big step, compositional, non-substitutional semantics as discussed above, alongside the associated Haskell implementation.

The first thing to formally consider is the input-output functionality of Jeeves. According to Yang et al [23, Fig. 3] the input and output at the Jeeves source level is specified by

$$\texttt{print}\,\{\,\textit{some-input}\,\}\,\textit{some-output}$$

statements, where the input is specified between the syntactical braces ({}), and the output is specified right after the braces. Thus, no input enters a Jeeves aka λ_J program at runtime but is given a

[4] *Lexical* or *static scoping* means that declared variables only occur within the text of the declared program structure.

priori, as a static part of the program structure. A program outcome amounts semantically to "the effect" of running a set of Jeeves print statements. (In our setting, '`print`' is in fact generalized to '`outputkind`', thus accounts for several different channels like '`print`', '`sendmail`', etc.) According to Yang et al [23, Fig. 3, Fig. 6], the print statement translates to

$$\mathsf{print}\ (\ \mathsf{concretize}\ e_v\ \mathsf{with}\ \upsilon_c\ )$$

where 'υ_c' is the translation of the *some-input* value, and 'e_v' is the translation of the *some-output* expression. Input values are semantically concrete values 'υ_c' (as hinted by the subscript 'c'), that is either a *literal* or a *record*. Output values are semantically defined by the outcome of the 'e_v' evaluation, which we here assume results in either a *literal*, a *record*, or *error* (all *concrete, printable values*) being channeled out. The input and output value domains are recursively defined by the algebras $InputValue$ and $OutputValue$.

7.1 Definition (semantic input-output values).

$$\begin{aligned} iv \in InputValue &::= \ \mathsf{lit} \mid \mathsf{record}\ fi_1 : iv_1 \dots fi_m : iv_m \\ ov \in OutputValue &::= \ \mathsf{lit} \mid \mathsf{record}\ fi_1 : ov_1 \dots fi_m : ov_m \mid \mathsf{error} \\ &\mathsf{where\ lit} \in Literal, \mathsf{error} \in ConcreteValue \end{aligned}$$

$Error$ is the algebraic specification for erroneous program states.

7.2 Remark (related value domain). Formally we have that $InputValue, OutputValue \subset ConcreteValue$. Notice, however, that the latter inclusion breaks slighly down as we extend the $OutputValue$ domain in Definition 7.9.

7.3 Remark (output outcome). Though not explicitly stated by Yang et al, we have decided only to consider data structures as part of our semantic output value domain and omit (function) closures, despite '$\lambda x.e$' expressions technically are "first class citizens" in Jeeves. Whence only including values which are printable.

7.4 Remark (implementation). We do not include an explicit Haskell implementation of the input-output domains. The specification merely serves as an overview of this functionality.

The second thing to formally consider is the *final set of solution constraints* to be resolved upon completion of the evaluation of a print statement. According to Yang et al [23, Fig. 3], the dynamic evaluation of a print statement terminates with the application of either of two rules, the (E-CONCRETIZESAT) or the (E-CONCRETIZEUNSAT). The decision upon which of the rules apply, depends on whether there exists a unique solution '$\mathcal{M}$' (for model) which solves the constrainst set, as expressed by the premise '$\text{MODEL}(\Delta, \Sigma \cup \{\mathcal{G} \wedge \mathsf{context}{=}\upsilon_c\}) = \mathcal{M}$' such that the constraint solution run on the (possibly symbolic) output expression 'υ_v', instantiates to a (concrete) output value, as the premise '$c = \mathcal{M}[\![\upsilon_v]\!]$' suggests.[5] We formalize the structure '$\text{MODEL}(\Delta, \Sigma \cup \{\mathcal{G} \wedge \mathsf{context}{=}\upsilon_c\})$' over the elements Σ (hard constraints), Δ (soft constraints), '$\mathcal{G}$' (path condition) and 'υ_c' (concrete input value, here renamed 'κ').

7.5 Definition (solution model).

$$sol \in Solution ::= \text{MODEL}\,(\Delta, \Sigma \cup \{\mathcal{G} \wedge \mathsf{context}{=}\kappa\})$$

where $\mathcal{G} \in PathCondition, \quad \kappa \in ConcreteValue.$

[5] A correct premise would have been '$true \vdash \langle \emptyset, \emptyset, \mathcal{M}[\![\upsilon_v]\!]\rangle \rightarrow \langle \emptyset, \emptyset, c\rangle$' in Yang et al [23, Fig. 3,(E-CONCRETIZESAT)].

*7.6 Remark (*MODEL *tag).* Because we do not specify a constraint solver in this formalization, we apply the tag MODEL as a *syntactic constructor* with no semantic meaning associated.

7.7 Remark (default theory property). We notice that the constraint set defined by '$(\Delta, \Sigma \cup \{\mathcal{G} \wedge \text{context} = \upsilon_c\})$' equally forms a (super-normal) default theory.

7.8 Haskell (solution model). The MODEL construction is implemented as the special data type Solution, which is equivalent to the MODEL container, and a one-to-one implementation of the 'sol' (concretized constraint set) quadruple. We notice, that the implementation doesn't validate whether Value is concrete or not at this point (but the later evaluation rule does).

```
data Solution = MODEL Delta Sigma PathCondition Value
type Solutions = [Solution]

noSolutions :: Solutions
noSolutions = []
```

In accordance with Yang et al, we do not specify constraint resolution explicitly in our formalizations, but tacitly asume that the passage is deferred to later by delegating to an external, off-the-shelf SMT solver [3]. Thus, we have deliberately omitted the specification of the '$c = \mathcal{M}[\![\upsilon_v]\!]$' clause in our specifications. The ensemble, however, that is fed to the constraint solver, will take

the form of a new concrete value, which consists of two components, the final accumulated constraint set formalized by $Solution$ together with the 'υ_v' (the evaluated output expression feeding into '$\mathcal{M}[\![\upsilon_v]\!]$' upon constraint resolution).

7.9 Definition (instantiation). Extend the output value algebra of Definition 7.1 with an additional form:

$$ins \in OutputValue ::= \ldots \mid \text{INSTANTIATE}\,(sol, \upsilon)$$

where $sol \in Solution, \upsilon \in Value$

7.10 Remark (the INSTANTIATE *tag).* To increase readability, we apply the tag INSTANTIATE as a *syntactic constructor* with no semantic meaning associated.

7.11 Haskell (instantiation). We implement the instantiation concrete value with the special data type Instantiate because it is only used at the outermost level of the evaluation.

```
data Instantiate = INSTANTIATE Solution Value
```

The third thing to formally consider is the *runtime (side) effects* from running a λ_J program. The original semantics does not include an explicit evaluation rule for a complete λ_J program evaluation, but specify the evaluation of each individual print statement, hinting that constraint solving happens per individual output statement [23, Fig. 3]. In other words, λ_J only supports *constraint propagation* per output posting.[6] No constraints gets "carried over" from the runtime evaluation of one output statement to the other. Consequently, we formalize the effect of running a Jeeves aka λ_J program to be a list of independent writings to individual output channels. All formalized by the (program) $Effect$ algebra.

[6]Constraint propagation means that constraints are accumulated during the course of evaluation.

7.12 Definition (program effect).

$$\mathcal{E} \in Effect ::= (output, ins)$$

where $output \in OutputKind,\, ins \in OutputValue$

7.13 Haskell (Effects). The Effect algebra is implemented as the special data type Effect, which is equivalent to the EFFECT container, and a one-to-one implementation of '$output$' and the instantiate output value 'ins'.

```
data Effect = EFFECT Outputkind Instantiate
type Effects = [Effect]

noEffects :: Effects
noEffects = []
```

Notice that the concrete value returned uses the dedicated Instantiate type.

With all preliminary concepts formalized and implemented, we can then proceed by formalizing the actual program runtime semantics. In this work, we formulate the λ_{J} evaluation semantic as a *fixpoint semantics* in the environment 'ρ'. Because we have build the semantics with trivial constructs, we know the existence of a *least fixpoint*, which how we are formulating our semantics [20].

In Section 3, we introduced the notion of a λ_{J} program, to specifically include an explicit ('letrec') recursion construct at the λ_{J} level, with the intent of building a recursive function environment in the top-scope, at runtime. The dynamic semantics of the letrec expression is aimed at being defined as the so-called *ML letrec* with the difference from ML that in λ_{J}, the letrec is defined only to appear at the top level of a program [15]. [7]

We are furthermore assuming that all output statements are evaluated *after* the program's recursive binding environment has been set up (something which is unclear in the original formalization, where let statements and print statements are presented in any mixed combination in the given examples.) For a more detailed treatment on the recursive binding feature, we refer to Section 5.

7.14 Definition (program evaluation rule).

$$\frac{\begin{array}{l}\rho_0,\, \mathcal{G}_0 \vdash \langle\{\}, \{\}, s_0\rangle \Rightarrow \mathcal{E}_1 \\ \;\;\ldots \\ \rho_0,\, \mathcal{G}_0 \vdash \langle\{\}, \{\}, s_{m-1}\rangle \Rightarrow \mathcal{E}_m\end{array}}{\vdash\ \mathsf{letrec}\ f_1 = ve_1 \cdots f_n = ve_n\ \mathsf{in}\ s_0\ \ldots\ s_{m-1} \Rightarrow (\mathcal{E}_1,\, \ldots,\, \mathcal{E}_m)} \quad \text{(p-letrec)}$$

where

$$\rho_0 = [f_1 \mapsto \upsilon_1, \ldots, f_n \mapsto \upsilon_n] \tag{1}$$

$$\text{For all } 0 \leq i \leq n : \quad \upsilon_i = \begin{cases} (ve_i, \rho_0) & \text{if } ve_i = \lambda x.e \,\vee\, ve_i = \mathsf{thunk}\, e \,\vee\, ve_i = x \\ ve_i & \text{otherwise} \end{cases} \tag{2}$$

$$\mathcal{G}_0 = \{\} \tag{3}$$

and $\quad f, v, x \in Var,\ ve \in ValExp,\ e \in Exp,\ s \in Statement,\ \mathcal{E} \in Effect$

[7] ML's *letrec* combinator defines names by recursive functional equations.

7.15 Remark (notation). To ease readability, we simply state '$[f_1 \mapsto v_1, \ldots, f_n \mapsto v_n]$' for the equivalent '$\epsilon[f_1 \mapsto v_1, \ldots, f_n \mapsto v_n]$' notation as expected according to Definition 5.5.

The program evaluation rule is composed as follows. The static, recursive binding environment 'ρ_0', specifies the initial top-level scope of a λ_J program. The path condition '$\mathcal{G}_0$', specifies the initial path constraints before execution of an output statement. In accordance with our early discussion, the execution environment, '$\rho_0, \mathcal{G}_0$', is the same before the execution of any output statement, regardless of the sequence in which they appear as 1) the recursive environment is assumed to be build up prior to any output statement execution, 2) constraints are not propagated from one output execution to the next.

According to Lemma 4.4, all function bindings, after translation of a Jeeves program to λ_J, is ensured to be on the (weak head normal) form '$f = ve$', where 've' is a value expression.The "where" clause of the program rule describes when closures, formalized by '(ve, ρ)', are initially build during program evaluation, and when not. As expected, this happens when the binding is dispatched to either a λ-closure, a thunk-closure, or a free variable closure. Otherwise, the binding is to either a literal, context, or error.

7.16 Haskell (program evaluation rule).

```haskell
evalProgram :: FreshVars → Program → Effects

evalProgram xs (P_LETREC recbindings outputstms) = effects
  where
    (CONSTRAINTS sigma delta effects) = evalStms xs rho0 emptyPath outputstms noEffects
    rho0 = foldr g empty recbindings
    g (BIND fi (E_BOOL b)) rho    = insert fi (V_BOOL b) rho
    g (BIND fi (E_NAT n)) rho     = insert fi (V_NAT n) rho
    g (BIND fi (E_STR s)) rho     = insert fi (V_STR s) rho
    g (BIND fi (E_CONST c)) rho   = insert fi (V_CONST c) rho
    g (BIND fi (E_VAR x)) rho     = insert fi (V_THUNK (E_VAR x) rho0) rho -- closure
    g (BIND fi (E_LAMBDA x e)) rho = insert fi (V_LAMBDA x e rho0) rho -- closure
    g (BIND fi (E_THUNK e)) rho   = insert fi (V_THUNK e rho0) rho -- closure
    g (BIND fi (E_RECORD fes)) rho = insert fi (V_THUNK (E_RECORD fes) rho0) rho --
       closure


evalStms :: FreshVars → Environment → PathCondition → Statements → Effects → Constraints
     Effects

evalStms xs rho g [] effects = return effects

evalStms xs rho g (stm:stms) effects = do
   effect ← evalStm xs1 rho g stm
   effects2 ← evalStms xs2 rho g stms effects
   return ( effect : effects2 )
  where
    ~(xs1,xs2) = splitVars xs
```

7.17 Definition (evaluation of a statement). The big step rule for evaluation of an (output) statement corresponds to the evaluations by the small step rules E-ConcretizeExp, E-ConcretizeSat, E-CONCRETIZEUNSAT in Yang etal. [23, Fig. 3], except for the fact that we do *not* seek to solve the constraint set to generate a solution '$\mathcal{M}$', but only seek to generate the set of constraints: MODEL is here merely a syntactic constructor and has no semantic significance unlike in Yang etal. [23, Fig. 3].

$$\frac{\begin{array}{c}\rho, \mathcal{G} \vdash \langle \Sigma, \Delta, e_1 \rangle \Rightarrow \langle \Sigma_1, \Delta_1, \upsilon_1 \rangle \\ \rho, \mathcal{G} \vdash \langle \Sigma_1, \Delta_1, e_2 \rangle \Rightarrow \langle \Sigma_2, \Delta_2, \kappa_2 \rangle\end{array}}{\begin{array}{l}\rho, \mathcal{G} \vdash \langle \Sigma, \Delta, \mathsf{output}\ (\mathsf{concretize}\ e_1\ \mathsf{with}\ e_2) \rangle \\ \qquad \Rightarrow (\mathsf{output}, \mathrm{INSTANTIATE}(\mathrm{MODEL}(\Delta_2, \Sigma_2 \cup \{\mathcal{G} \wedge \mathsf{context}{=}\kappa_2\}), \upsilon_1))\end{array}} \quad \text{(e-concretize)}$$

7.18 Remark (extended concretize *syntax).* Because '`print`' at the Jeeves source-level has been generalized to '`output`' in our formalization (with the tacit assumption that $OutputKind$ carries over to λ_J), we have added 'output' as an explicit tag in our semantics compared to Yang et al [23, Fig. 3] to keep track of the writes to the various kinds of output channels.

7.19 Haskell (evaluation of a statement).

```
evalStm :: FreshVars → Environment → PathCondition → Statement → Constraints Effect

evalStm xs rho g (CONCRETIZE_WITH output e1 e2) =
  (CONSTRAINTS sigma delta effect)
  where
    (CONSTRAINTS sigma delta (c,v)) = do v1 ←evalExp xs1 rho g e1
                                         c2 ← evalExp xs2 rho g e2
                                         return (c2,v1)    -- = (c,v) by pattern matching
    effect | isConcrete c = EFFECT output (INSTANTIATE (MODEL delta sigma g c) v)
           | otherwise    = error ("Attempt to create MODEL with non-concrete final 
              value"++show c)
    ~(xs1,xs2) = splitVars xs
```

7.20 Definition (evaluation of expressions). The judgement

$$\rho, \mathcal{G} \vdash \langle \Sigma, \Delta, e \rangle \Rightarrow \langle \Sigma', \Delta', \upsilon \rangle$$

describes the evaluation of a λ_J expression 'e' to a value 'υ' in the static environment 'ρ', under pathcondition '$\mathcal{G}$', where Σ' and Δ' capture the privacy effects of the evaluation on the constraint sets Σ and Δ.

7.21 Haskell (evaluation of expressions).

```
evalExp :: FreshVars → Environment →PathCondition → Exp → Constraints Value
```

We proceed by presenting an environment-based, big step formulation and implementation of the dynamic expression semantics of λ_J. The semantics follows the syntax presented in Definition 2.1, and modifies and clarifies the original semantics [23, Figure 3].

7.22 Definition (evaluation of literals and context**).** There are no explicit rules for handling literals and context in [23, Figure 3]. We do, however, tacitly assume it to be the "identity mapping". The present rule evaluates a subset of simple normal form (expressions): 'b', 'n', 's', 'c', '$context$' to the eqivalent normal form (values).

$$\frac{}{\rho, \mathcal{G} \vdash \langle \Sigma, \Delta, \mathsf{ve} \rangle \Rightarrow \langle \Sigma, \Delta, \mathsf{ve} \rangle} \quad \text{where } ve \in \{b,\ n,\ s,\ c,\ context\} \qquad \text{(e-simple)}$$

7.23 Haskell (evaluation of literals and simple expressions). The distinction between (normal form) *expressions* and *values* in Definition 7.22 becomes apparent when E_ constructors are translated into V_ constructors.

```
evalExp xs rho g (E_BOOL b) = return (V_BOOL b)
evalExp xs rho g (E_NAT n) = return (V_NAT n)
evalExp xs rho g (E_STR s)  = return (V_STR s)
evalExp xs rho g (E_CONST c) = return (V_CONST c)
evalExp xs rho g (E_CONTEXT) = return (V_CONTEXT)
```

7.24 Definition (evaluation of variable expressions). There are no explicit rules for handling variables in [23, Figure 3]. The present rule shows how regular variables, but also level variables are handled in an environment-based semantics. For further specifics on the role of level variables in the environment, we refer to Definition 7.36.

$$\frac{}{\rho, \mathcal{G} \vdash \langle \Sigma, \Delta, x \rangle \Rightarrow \langle \Sigma, \Delta, \rho(x) \rangle} \quad \text{where } \rho(x) \neq (\mathsf{thunk}\, e', \rho') \qquad \text{(e-var1)}$$

$$\frac{\rho', \mathcal{G} \vdash \langle \Sigma, \Delta, e' \rangle \Rightarrow \langle \Sigma', \Delta', v' \rangle}{\rho, \mathcal{G} \vdash \langle \Sigma, \Delta, x \rangle \Rightarrow \langle \Sigma', \Delta', v' \rangle} \quad \text{where } \rho(x) = (\mathsf{thunk}\, e', \rho') \qquad \text{(e-var2)}$$

7.25 Haskell (evaluation of variable expressions).

```
evalExp xs rho g (E_VAR x) = evalExp_VAR (if x `member` rho then rho!x else error ("Undefined
    !"++show x))
  where
    evalExp_VAR (V_THUNK e' rho') = evalExp xs rho' g e'
    evalExp_VAR v                 = return v
```

7.26 Definition (evaluation of lambda expressions). There is no specific rule for lambda expressions alone in Yang etal. [23, Fig. 3]. The present big step rule, however, partially correspond to the binding-part of E-APPLAMBDA. In the current semantics, lambda expression evaluation builds a (concrete) closure normal form with the current environment and returns it as semantic value *c.f.* Definition 5.1.

$$\frac{}{\rho, \mathcal{G} \vdash \langle \Sigma, \Delta, \lambda x.e \rangle \Rightarrow \langle \Sigma, \Delta, (\lambda x.e, \rho) \rangle} \qquad \text{(e-lambda)}$$

7.27 Haskell (evaluation of lambda expressions).

```
evalExp xs rho g (E_LAMBDA x e) = return (V_LAMBDA x e rho)
```

7.28 Definition (evaluation of binary operator expressions). The big step rule for evaluation of a binary operator expression corresponds to the evaluations by the small step rules E-OP, E-OP1, and E-OP2 in Yang etal. [23, Fig. 3]. Definition 2.5 specifies the token set of the operator sort that we have included in this formalization.

$$\frac{\begin{array}{c}\rho, \mathcal{G} \vdash \langle \Sigma, \Delta, e_1 \rangle \Rightarrow \langle \Sigma', \Delta', \kappa_1 \rangle \\ \rho, \mathcal{G} \vdash \langle \Sigma', \Delta', e_t \rangle \Rightarrow \langle \Sigma'', \Delta'', \kappa_2 \rangle\end{array}}{\rho, \mathcal{G} \vdash \langle \Sigma, \Delta, e_1 \; op \; e_2 \rangle \Rightarrow \langle \Sigma'', \Delta'', \kappa \rangle} \quad \kappa \equiv \kappa_1 \, op \, \kappa_2 \qquad \text{(e-op1)}$$

$$\frac{\rho, \mathcal{G} \vdash \langle \Sigma, \Delta, e_1 \rangle \Rightarrow \langle \Sigma', \Delta', \upsilon_1 \rangle \quad \rho, \mathcal{G} \vdash \langle \Sigma', \Delta', e_t \rangle \Rightarrow \langle \Sigma'', \Delta'', \upsilon_2 \rangle}{\rho, \mathcal{G} \vdash \langle \Sigma, \Delta, e_1 \; op \; e_2 \rangle \Rightarrow \langle \Sigma'', \Delta'', \upsilon_1 \; op \; \upsilon_2 \rangle} \quad \upsilon_1 \equiv \sigma_1 \vee \upsilon_2 \equiv \sigma_2 \qquad \text{(e-op2)}$$

7.29 Haskell (evaluation of binary operator expressions). Haskell 3.10 shows the implementation of the Op binary operator data type. Notice how we have implemented list concatenation by overloading the definition of OP_PLUS.

```haskell
evalExp xs rho g (E_OP op e1 e2) = do
  v1 ← evalExp xs1 rho g e1
  v2 ← evalExp xs2 rho g e2
  return (evalExp_OP rho g op v1 v2)
  where
    ~(xs1,xs2) = splitVars xs

    evalExp_OP rho g op v1 v2 | isConcrete v1 && isConcrete v2 = (evalOpCC op v1 v2)
                              | isSymbolic v1 || isSymbolic v2 = (V_OP op v1 v2)

    evalOpCC :: Op → Value → Value → Value

    evalOpCC OP_PLUS (V_NAT n1) (V_NAT n2) = V_NAT (n1+n2)
    evalOpCC OP_PLUS (V_STR s1) (V_STR s2) = V_STR (s1++s2)

    evalOpCC OP_MINUS (V_NAT n1) (V_NAT n2) = V_NAT (n1−n2)

    evalOpCC OP_AND (V_BOOL b1) (V_BOOL b2) = V_BOOL (b1&&b2)
    evalOpCC OP_OR (V_BOOL b1) (V_BOOL b2) = V_BOOL (b1||b2)
    evalOpCC OP_IMPLY (V_BOOL b1) (V_BOOL b2) = V_BOOL ((not b1)||b2)

    evalOpCC OP_EQ v1 v2   = V_BOOL (v1≡v2)
    evalOpCC OP_LESS v1 v2 = V_BOOL (v1<v2)
    evalOpCC OP_GREATER v1 v2 = V_BOOL (v1>v2)
```

7.30 Definition (evaluation of unary operator expressions). There are no specific rules concerning unary operator expressions in Yang etal. [23, Fig. 3]. The big step rules, however, are simpel to construct and require no further commenting. Definition 2.5 specifies the token set of the operator sort, which currently is the singleton set {!} (negation).

$$\frac{\rho, \mathcal{G} \vdash \langle \Sigma, \Delta, e \rangle \Rightarrow \langle \Sigma', \Delta', \kappa \rangle}{\rho, \mathcal{G} \vdash \langle \Sigma, \Delta, uop \; e \rangle \Rightarrow \langle \Sigma'', \Delta'', \kappa' \rangle} \quad \kappa' \equiv uop \, \kappa \qquad \text{(e-uop1)}$$

$$\frac{\rho, \mathcal{G} \vdash \langle \Sigma, \Delta, e \rangle \Rightarrow \langle \Sigma', \Delta', \sigma \rangle}{\rho, \mathcal{G} \vdash \langle \Sigma, \Delta, uop \; e \rangle \Rightarrow \langle \Sigma'', \Delta'', uop \; \sigma \rangle} \qquad \text{(e-uop2)}$$

7.31 Haskell (evaluation of unary operator expressions). Definition 3.10 shows the implementation of the UOp unary operator data type. (Currently a singleton with the OP_NOT constructor).

```haskell
evalExp xs rho g (E_UOP uop e) = do
  v ← evalExp xs rho g e
  return (evalExp_UOP rho g uop v)
```

```
  where
    evalExp_UOP rho g uop v | isConcrete v = evalUOpC uop v
                            | isSymbolic v = V_UOP uop v

    evalUOpC :: UOp →Value → Value
    evalUOpC OP_NOT (V_BOOL b) = V_BOOL (not b)
```

7.32 Definition (evaluation of conditional expressions). The big step rules for evaluation of a conditional expression corresponds to the evaluations by the small step rules E-COND, E-CONDTRUE, E-CONDFALSE, E-CONDSYMT, and E-CONDSYMF. Depending on the conditional, the semantics is implemented in two way: provided it evaluates to a boolean value, then the if-expression *behaves in a non-strict fashion*. Provided the conditional evaluates to a symbolic normal form , however, then the if-expression *behaves in a strict fashion* as both branches are evaluated to normal forms. The latter underpins the primary reason for symbolic if-evaluation: to implement the semantics of sensitive values. The evaluation of each branch is in fact performed as separate evaluation steps under (opposing) symbolic/ logical conditions: '$\sigma \wedge \mathcal{G}$', and '$\neg\sigma \wedge \mathcal{G}$', and the generated constraint sets are successively being assembled into Σ''' and Δ'''.[8].

$$\frac{\rho, \mathcal{G} \vdash \langle \Sigma, \Delta, e_1 \rangle \Rightarrow \langle \Sigma', \Delta', \mathsf{true} \rangle \qquad \rho, \mathcal{G} \vdash \langle \Sigma, \Delta, e_2 \rangle \Rightarrow \langle \Sigma'', \Delta'', \upsilon_2 \rangle}{\rho, \mathcal{G} \vdash \langle \Sigma, \Delta, \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \rangle \Rightarrow \langle \Sigma'', \Delta'', \upsilon_2 \rangle} \quad \text{(e-cond1)}$$

$$\frac{\rho, \mathcal{G} \vdash \langle \Sigma, \Delta, e_1 \rangle \Rightarrow \langle \Sigma', \Delta', \mathsf{false} \rangle \qquad \rho, \mathcal{G} \vdash \langle \Sigma', \Delta', e_3 \rangle \Rightarrow \langle \Sigma'', \Delta'', \upsilon_3 \rangle}{\rho, \mathcal{G} \vdash \langle \Sigma, \Delta, \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \rangle \Rightarrow \langle \Sigma'', \Delta'', \upsilon_3 \rangle} \quad \text{(e-cond2)}$$

$$\frac{\rho, \mathcal{G} \vdash \langle \Sigma, \Delta, e_1 \rangle \Rightarrow \langle \Sigma', \Delta', \sigma \rangle \qquad \rho, \sigma \wedge \mathcal{G} \vdash \langle \Sigma', \Delta', e_2 \rangle \Rightarrow \langle \Sigma'', \Delta'', \upsilon_2 \rangle \qquad \rho, \neg\sigma \wedge \mathcal{G} \vdash \langle \Sigma'', \Delta'', e_3 \rangle \Rightarrow \langle \Sigma''', \Delta''', \upsilon_3 \rangle}{\rho, \mathcal{G} \vdash \langle \Sigma, \Delta, \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \rangle \Rightarrow \langle \Sigma''', \Delta''', \mathsf{if}\ \sigma\ \mathsf{then}\ \upsilon_2\ \mathsf{else}\ \upsilon_3 \rangle} \quad \text{(e-cond3)}$$

The if expession evaluation rule is implemented as follows.

7.33 Haskell (evaluation of conditional expressions).

```
evalExp xs rho g (E_IF e1 e2 e3) = do
  v1 ← evalExp xs1 rho g e1
  evalExp_IF v1
  where

    -- (e-cond1)
    evalExp_IF (V_BOOL True) = evalExp xs2 rho g e2

    -- (e-cond2)
    evalExp_IF (V_BOOL False) = evalExp xs2 rho g e3

```

[8]Because constraints are assembled through set union, the order by which the branches are evaluated is insignificant.

```
-- (e-cond3)
evalExp_IF s1 | isSymbolic s1 = do
  v2 ← evalExp xs21 rho (formulaConjunction (F_IS s1) g) e2
  v3 ← evalExp xs22 rho (formulaConjunction (F_NOT s1) g) e3
  return (V_IF s1 v2 v3)

~(xs1,xs2) = splitVars xs
~(xs21,xs22) = splitVars xs2
```

7.34 Definition (evaluation of application expressions). The big step rule for evaluation of an application expression corresponds to the evaluations described by the small step rules E-APP1, E-APP2, and E-APPLAMBDA in Yang etal. [23, Fig. 3]. It specifies how function application is carried out through *call-by-value evaluation*, but with the important difference that variable binding during β-reduction is handled on an *environment basis* ($\rho'[x \mapsto \upsilon_2]$) instead of a *substitution basis* ($e[x \mapsto \upsilon]$), *c.f.* Henderson [9].[9] The present application rule reformulation is a direct consequence of letting lexical scoping be handled with closures as described in Section 5. Finally, we allow the capturing of an erroneous λ_J application upon which the error normal form is returned as a semantic result.

$$\frac{\begin{array}{c}\rho, \mathcal{G} \vdash \langle \Sigma, \Delta, e_1 \rangle \Rightarrow \langle \Sigma', \Delta', \upsilon_1 \rangle \\ \rho, \mathcal{G} \vdash \langle \Sigma', \Delta', e_2 \rangle \Rightarrow \langle \Sigma'', \Delta'', \upsilon_2 \rangle \\ \rho'[x \mapsto \upsilon_2], \mathcal{G} \vdash \langle \Sigma'', \Delta'', e' \rangle \Rightarrow \langle \Sigma''', \Delta''', \upsilon_3 \rangle\end{array}}{\rho, \mathcal{G} \vdash \langle \Sigma, \Delta, e_1\ e_2 \rangle \Rightarrow \langle \Sigma''', \Delta''', \upsilon_3 \rangle} \quad \upsilon_1 \equiv (\lambda x.e', \rho') \qquad \text{(e-app1)}$$

$$\frac{\begin{array}{c}\rho, \mathcal{G} \vdash \langle \Sigma, \Delta, e_1 \rangle \Rightarrow \langle \Sigma', \Delta', \sigma_1 \rangle \\ \rho, \mathcal{G} \vdash \langle \Sigma', \Delta', e_2 \rangle \Rightarrow \langle \Sigma'', \Delta'', \upsilon_2 \rangle\end{array}}{\rho, \mathcal{G} \vdash \langle \Sigma, \Delta, e_1\ e_2 \rangle \Rightarrow \langle \Sigma'', \Delta'', \mathsf{error} \rangle} \qquad \text{(e-app2)}$$

7.35 Haskell (evaluation of application expressions).

```
evalExp xs rho g (E_APP e1 e2) = do
    v1 ← evalExp xs1 rho g e1
    v2 ← evalExp xs2 rho g e2
    v3 ← evalExp_APP v1 v2
    return v3
  where
    ~(xs1,xs2,xs3) = splitVars3 xs

    evalExp_APP (V_LAMBDA x e' rho') v2 = do
          v ← evalExp xs3 ( insert x v2 rho') g e'
          return v

    evalExp_APP _ _= return (V_ERROR)
```

[9] "Environment based" instead of "substitution based" semantics prevents unforseable expression expansion, when code is substituted into terms at runtime, thus ensures that inductive argumentation can be applied to prove properties of the semantics.

7.36 Definition (evaluation of defer expressions). The big step rule for evaluation of a defer expression basically corresponds to the evaluations by the small step rules E-DEFEERCONSTRAINT, and E-DEFER in Yang etal. [23, Fig. 3]. The current defer syntax, *i.e.* 'defer lx in e', presents three major differences from the original syntax, as described in Remark 3.7. We have modified the defer semantics accordingly, by making the evaluation step about "the body" e, whilst removing now void evaluation steps for syntax which is no longer present, notably '$\{e\}$', '$\{\upsilon_c\}$' and 'default υ_d'. The overall aim of the defer rule is to introduce (level) variables, say 'lx', and their default values 'true' into the semantics, in a way that prevents name clashing in the constraint scopes. In this setting, we manage (level) variable names 'lx' on the environment stack, by performing an α-renaming with "fresh" variables 'lx''. Default values 'true' for variables 'lx'' are weighing in on any associated (policy) hard constraints by registering as *soft contraints* in the collected constraint set '$\Delta \cup \{\mathcal{G} \Rightarrow (lx'{=}\mathsf{true})\}$'.

$$\frac{\rho[lx \mapsto lx'], \mathcal{G} \vdash \langle \Sigma, \Delta \cup \{\mathcal{G} \Rightarrow (lx'{=}\mathsf{true})\}, e\rangle \Rightarrow \langle \Sigma', \Delta', \upsilon\rangle}{\rho, \mathcal{G} \vdash \langle \Sigma, \Delta, \mathsf{defer}\ lx\ \mathsf{in}\ e\rangle \Rightarrow \langle \Sigma', \Delta', \upsilon\rangle}\quad \text{fresh } lx' \qquad \text{(e-defer)}$$

To ensure that no bound variables escape into the contraint set we observe the following.

7.37 Lemma (environment scope invariant). *For every instance of the judgement '$\rho, \mathcal{G} \vdash \langle \Sigma, \Delta, e\rangle \Rightarrow \langle \Sigma', \Delta', \upsilon\rangle$' we have that the domain of 'ρ' contains all free variables in 'e', and no free variables from 'υ'.*

Proof. Proven by induction over proofs, where the base cases are the premises of Definition 7.14 and the step is shown for every inference rule. □

The defer expression evaluation rule is implemented as follows.

7.38 Haskell (evaluation of defer expressions).

```
evalExp ~(x:xs) rho g (E_DEFER lx e) = do
  unitDeltaConstraints g x (V_BOOL True)
  v ← evalExp xs (insert lx lx' rho) g e
  return v
  where lx' = V_VAR x
```

7.39 Definition (evaluation of assert expressions). The big step rule for evaluation of an assert expression corresponds to the evaluations by the small step rules E-ASSERTCONSTRAINT, and E-ASSERT in Yang etal. [23, Fig. 3]. The current assert syntax, however, has extended the syntax with an 'in e_2' part, as described in Remark 3.8. We have extended the semantics accordingly, by adding a separate evaluation step for 'e_2'. The overall aim of assert is to introduce policy constraints, given by the (constraint) expression 'e_1', into the semantics. This is effectuated through evaluation of 'e_1' to a symbolic normal form 'υ_1', followed by the introduction of those as *hard constraints* into the constraint environment as '$\Sigma' \cup \{\mathcal{G} \Rightarrow \upsilon_1\}$'.

$$\frac{\rho, \mathcal{G} \vdash \langle \Sigma, \Delta, e_1\rangle \Rightarrow \langle \Sigma', \Delta', \upsilon_1\rangle \qquad \rho, \mathcal{G} \vdash \langle \Sigma' \cup \{\mathcal{G} \Rightarrow \upsilon_1\}, \Delta', e_2\rangle \Rightarrow \langle \Sigma'', \Delta'', \upsilon_2\rangle}{\rho, \mathcal{G} \vdash \langle \Sigma, \Delta, \mathsf{assert}\ e_1\ \mathsf{in}\ e_2\rangle \Rightarrow \langle \Sigma'', \Delta'', \upsilon_2\rangle} \qquad \text{(e-assert)}$$

The assert expression evaluation rule is implemented as follows.

7.40 Haskell (evaluation of assert expressions).

```
evalExp xs rho g (E_ASSERT e1 e2) = do
  v1 ← evalExp xs1 rho g e1
  unitSigmaConstraints g v1
  v2 ← evalExp xs2 rho g e2
  return v2
  where
    ~(xs1,xs2) = splitVars xs
```

7.41 Definition (evaluation of let expressions). There are no specific rules for λ_J let expressions in Yang etal. [23, Fig. 3]. In the current semantics, we implement dynamic let evaluation by *eager evaluation*, in that the binding argument 'e_1', always is evaluated to a normal form 'v_1' first, then stacked in the binding environment '$\rho[x_1 \mapsto v_1]$' as the context in which "the body" 'e_2' is evaluated. This is reflected by the order of the two separate evaluation steps in the following rule.

$$\frac{\rho, \mathcal{G} \vdash \langle \Sigma, \Delta, e_1 \rangle \Rightarrow \langle \Sigma', \Delta', v_1 \rangle \qquad \rho[x_1 \mapsto v_1], \mathcal{G} \vdash \langle \Sigma', \Delta', e_2 \rangle \Rightarrow \langle \Sigma'', \Delta'', v_2 \rangle}{\rho, \mathcal{G} \vdash \langle \Sigma, \Delta, \ \mathsf{let}\ x_1 = e_1 \ \mathsf{in}\ e_2 \rangle \Rightarrow \langle \Sigma'', \Delta'', v_2 \rangle} \quad \text{(e-let)}$$

7.42 Haskell (evaluation of let expressions).

```
evalExp xs rho g (E_LET x1 e1 e2) = do
  v1 ←  evalExp xs1 rho g e1
  evalExp xs2 ( insert x1 v1 rho) g e2
  where
    ~(xs1,xs2) = splitVars xs
```

7.43 Definition (evaluation of record expressions). There are no specific rules for record expressions in Yang etal. [23, Fig. 3]. In the current eager semantics, however, we implement record evaluation *strictly* in the field arguments, as a left-to-right evaluation of the field bodies $e_0 \dots e_n$ to symbolic normal forms $v_0 \dots v_n$.

$$\frac{\rho, \mathcal{G} \vdash \langle \Sigma_0, \Delta_0, e_1 \rangle \Rightarrow \langle \Sigma_1, \Delta_1, v_1 \rangle \ \cdots \ \rho, \mathcal{G} \vdash \langle \Sigma_{n-1}, \Delta_{n-1}, e_n \rangle \Rightarrow \langle \Sigma_n, \Delta_n, v_n \rangle}{\rho, \mathcal{G} \vdash \langle \Sigma_0, \Delta_0, \ \mathsf{record}\ x_1 = e_1 \dots x_n = e_n \rangle \Rightarrow \langle \Sigma_n, \Delta_n, \ \mathsf{record}\ x_1 = v_1 \dots x_n = v_n \rangle} \ n \geq 0 \ \text{(e-rec)}$$

7.44 Remark (empty record). We have deliberately allowed $n = 0$, as a way to signify the empty record.

7.45 Haskell (evaluation of record expressions).

```
evalExp xs rho g (E_RECORD fies) = do
   fivs ← mapM eval1 (insertXss xs fies )
  return (V_RECORD fivs)
  where
    insertXss xs [] = []
    insertXss xs ((x,e):xes) = (x,e,xs1) : insertXss xs2 xes where ~(xs1,xs2) = splitVars xs

    eval1 (x,e,xs) = do v ← evalExp xs rho g e
                        return (x,v)
```

7.46 Definition (evaluation of field expressions). There are no specific rules for field look up expressions in Yang etal. [23, Fig. 3]. In the current semantics, we implement field lookup *strictly*, in that the record expression part 'e' of '$e.f_i$' is evaluated completely to symbolic normal form. If the evaluation renders a 'record' with all fields on normal form, the indicated field content is returned as semantic value. Otherwise, we return the normalized field lookup entity '$\sigma.fi$' as semantic value.

$$\frac{\rho, \mathcal{G} \;\vdash\; \langle \Sigma, \Delta, e \rangle \;\Rightarrow\; \langle \Sigma_1, \Delta_1, \text{record } fi_1 = v_1 \;\ldots\; fi_n = v_n \rangle}{\rho, \mathcal{G} \;\vdash\; \langle \Sigma, \Delta, e.fi_i \rangle \;\Rightarrow\; \langle \Sigma_1, \Delta_1, v_i \rangle} \qquad \text{(e-field1)}$$

$$\frac{\rho, \mathcal{G} \;\vdash\; \langle \Sigma, \Delta, e \rangle \;\Rightarrow\; \langle \Sigma_1, \Delta_1, \sigma \rangle}{\rho, \mathcal{G} \;\vdash\; \langle \Sigma, \Delta, e.fi \rangle \;\Rightarrow\; \langle \Sigma_1, \Delta_1, \sigma.fi \rangle} \quad \sigma \neq \text{record } fi_1 = v_1 \;\ldots\; fi_n = v_n \qquad \text{(e-field2)}$$

7.47 Haskell (evaluation of field expressions).

```
evalExp xs rho g (E_FIELD e fi) = do
  v1 ← evalExp xs rho g e
  return (evalVar_FIELD v1)
  where
    evalVar_FIELD (V_RECORD fivs) = head [v' | (fi',v') ← fivs , fi' ≡ fi ]
    evalVar_FIELD v               = (V_FIELD v fi)
```

Like the semantics by Yang et al [23], we observe that the evaluation semantics constitutes a deterministic proof system.

Finally, we illustrate the program evaluation rule with the first of our canonical examples from Example 1.4, based on the translation to λ_J in Example 4.14. Because of the shere size, however, we only show selected parts of the proof tree.

7.48 Example (Name policy program evaluation).

The main judgement has the following form:

$$\frac{\begin{array}{c}\rho_0,\, \mathcal{G}_0 \vdash \langle \{\}, \{\}, \text{print}(\text{concretize}\, msg \text{ with } alice) \rangle \Rightarrow \mathcal{E}_1 \\ \rho_0,\, \mathcal{G}_0 \vdash \langle \{\}, \{\}, \text{print}(\text{concretize}\, msg \text{ with } bob) \rangle \Rightarrow \mathcal{E}_2\end{array}}{\begin{array}{l}\vdash \text{ letrec name}{=}ve_1, \text{msg}{=}ve_2 \text{ in print}(\text{concretize}\, msg \text{ with } alice) \\ \qquad\qquad\qquad\qquad\qquad \text{print}(\text{concretize}\, msg \text{ with } bob) \Rightarrow \mathcal{E}_1,\, \mathcal{E}_2\end{array}} \qquad \text{(p-letrec)}$$

where

$$\begin{aligned}\rho_0 &= [\text{name} \mapsto (ve_1, \rho_0),\ \text{msg} \mapsto (ve_2, \rho_0)] \\ \mathcal{G}_0 &= \{\}\end{aligned}$$

and

$$\begin{aligned}ve_1 &= \text{thunk}(\text{defer}\, a \text{ in } (\text{assert } (!(\text{context} = alice) => (a = false)) \; in \; [\![<\text{"}Anonymous\text{"} \mid \text{"}Alice\text{"}>(a)]\!])) \\ ve_2 &= \text{thunk } (\text{"}Author\ is\ \text{"} + name)\end{aligned}$$

and

$$[\![<\text{"}Anonymous\text{"} \mid \text{"}Alice\text{"}>(a)]\!] = \text{if } a \text{ then "}Alice\text{" else "}Anonymous\text{"}$$

8 Running a Jeeves program

In this section, we show how to run a Jeeves program as it pertains to this document as a literate Haskell implementation of a Jeeves compiler and a λ_J evaluation engine. The main program is the *Jeeves program evaluator*. It consists of a parsing step, which converts from the Jeeves source language to λ_J abstract syntax, followed by an evaluation phase of the generated λ_J terms *c.f.* Figure 1. We also provide a way to run just the compile step to λ_J terms (*i.e.*, without the output part in Figure 1 as the input part is a build-in feauture of Jeeves). We are dedicating the remainder of the section to show how to run the canonical "Naming Policy" program from Figure 2, and "Conference Management System" program from Figure 3, and how to interpret the results.

At first, we illustate the beginning of a session with the Hugs Haskell system [11], where this literate program [17] is loaded with the command `:load "jeeves-constraints.lhs"`. (The program also runs with Glasgow Haskell.) In the remainder of this section, we will tacitly assume that loading has been successfully completed.

```
__   __ __  __  ____   ___      _________________________________________
||   || ||  || ||  || ||__      Hugs 98: Based on the Haskell 98 standard
||___|| ||__|| ||__||  __||     Copyright (c) 1994-2005
||---||         ___||           World Wide Web: http://haskell.org/hugs
||   ||                         Bugs: http://hackage.haskell.org/trac/hugs
||   || Version: September 2006 _________________________________________

Haskell 98 mode: Restart with command line option -98 to enable extensions

Type :? for help
Hugs> :load "jeeves-constraints.lhs"
Main>
```

A Jeeves program (and input) is evaluated with the invocation of the Jeeves evaluator by giving the command:

evaluateFile <*filename*>

which results in a sequence of (non-interfeering) 'Effects' in accordance with Definition 7.14 and Haskell 7.16. In appendix B.3 it is outlined how the effect output is formatted. The implementation of `evaluateFile` is reflected in the following code snippet.

8.1 Haskell (Jeeves evaluator).

```
-- -------------------
-- TOP EVALUATOR

evaluate :: String → Effects

evaluate jeeves = effects
  where
    programParse = parse (programParser xs1) jeeves
     effects = if null programParse then noEffects else evalProgram xs2 ( fst (head programParse)
        )
    (xs1,xs2) = splitVars vars

```

```
evaluateFile  filename = do jeeves ← readFile  filename    -- IO utility
                            putStr (show (evaluate  jeeves))
```

A Jeeves program (with input) is parsed/translated with the invocation of the Jeeves parser by giving the command:

parseFile <*filename*>

The parser output is a λ_J program that follows the specification in Definition 4.1 and Haskell 4.5. In appendix B.3 it is outlined how the λ_J output is formatted in Haskell. The code for `parseFile` is listed in the Haskell B.2 framework.

The program (with input) format has to adhere to the syntax specified in Definition 2.4, as illustrated by the Jeeves program examples in Figure 2 and Figure 3. In the following, we tacitly assume that two files have been created, `testp1.jeeves` and `testp2.jeeves`, which respectively contain those programs.

The (formatted) program output from running the program is a list of effects where each effect, according to Definition 7.17, is formally described by $(\mathsf{output}, \text{INSTANTIATE}(\text{MODEL}(\Delta, \Sigma \cup \{\mathcal{G} \wedge \mathsf{context}{=}\kappa\}), \upsilon))$. This output is formatted as follows by our implementation:

```
Effect "output"
   SOFT CONSTR = ...
   HARD CONSTR MODEL = ...
   SYMBOLIC VALUE = ...
```

where 'Effect' is a keyword, 'output' prints the value of output, 'SOFT CONSTR = ...' prints the soft constraint set Δ, 'HARD CONSTR MODEL = ...' prints the instantiated hard constraint set '$\Sigma \cup \{\mathcal{G} \wedge \mathsf{context}{=}\kappa\}$', and 'SYMBOLIC VALUE = ...' prints the symbolic value υ. The order in which the (non-interferring) effects appear, reflects directly the order in which the print statements appear in the Jeeves program. We obviously has chosen to keep that ordering in the formatted program output, which is printed as a vertical list of the form '[<*effect*>, . . . , <*effect*>]' where '<*effect*>' is formatted as described above. We depict how to run and what the formatted program output looks like for the Naming Policy Program from Figure 2. According to the theoretical program evaluation in Example 7.48, the program *exactly* evaluates to the expected constraint sets and values!

```
Main> evaluateFile "Tests/testp1.jeeves"
[
  EFFECT "print"
    SOFT CONSTR= {} ∪ {True ⇒ x10=true},
    HARD CONSTR MODEL = {} ∪ {True ⇒ (¬(context='alice') ⇒ (x10=false))}
                              ∪ {True ∧ context='alice'}
    SYMBOLIC VALUE = ('Author is ' + (if x10 then 'Alice' else 'Anonymous'))
  ,
  EFFECT "print"
    SOFT CONSTR = {} ∪ {True ⇒ x20=true},
    HARD CONSTR MODEL = {} ∪ {True ⇒ (¬(context='alice') ⇒ (x20=false))}
                              ∪ {True ∧ context='bob'}
    SYMBOLIC VALUE = ('Author is ' + (if x20 then 'Alice' else 'Anonymous'))
  ]
```

We also depict how to run and what the formatted program output looks like for the Conference Management Policy program from Figure 3. Eventhough we have not made a formal proof of the expected constraint sets and values, the result of the run at this point is relatively convincing according to common sense.

```
Main> evaluateFile "Tests/testp2.jeeves"
[
  EFFECT "print"
    SOFT CONSTR = {} ∪ {True ⇒ x90=true} ∪ {True ⇒ x58=true} ∪ {True ⇒ x26=true},
    HARD CONSTR MODEL = {}
          ∪ {True ⇒ (¬(((context.viewer.role=Reviewer)
                       ∨ (context.viewer.role=PC)) ∨ (context.stage=Public))
                     ⇒ (x90=false))}
          ∪ {True ⇒ (¬(((if x58 then 'Alice' else 'Anonymized')=context.viewer.name)
                       ∨ ((context.stage=Public) ∧ ¬((if x90 then Accepted else 'none')='none'))
                     ⇒ (x58=false))}
          ∪ {True ⇒ (¬(((context.viewer.name =(if x58 then 'Alice' else 'Anonymized'))
                       ∨ (context.viewer.role=Reviewer))
                       ∨ (context.viewer.role=PC))
                       ∨ ((context.stage=Public) ∧ ¬((if x90 then Accepted else 'none')='none'))
                     ⇒ (x26=false))}
          ∪ {True ∧ context=(record viewer=(record name='Alice' role=PC) stage=Public) }
    SYMBOLIC VALUE = (record title=(if x26 then 'MyPaper' else '')
                             author=(if x58 then 'Alice' else 'Anonymized')
                             accepted=(if x90 then Accepted else 'none')
                     )
  ,
  EFFECT "print"
    SOFT CONSTR = {} ∪ {True ⇒ x180=true} ∪ {True ⇒ x116=true} ∪ {True ⇒ x52=true},
    HARD CONSTR MODEL = {}
          ∪ {True ⇒ (¬(((context.viewer.role=Reviewer)
                       ∨ (context.viewer.role=PC)) ∨ (context.stage=Public))
                     ⇒ (x180=false))}
          ∪ {True ⇒ (¬(((if x116 then 'Alice' else 'Anonymized')=context.viewer.name)
                      ∨ ((context.stage=Public) ∧ ¬((if x180 then Accepted else 'none')='none'))
                     ⇒ (x116=false))}
          ∪ {True ⇒ (¬(((context.viewer.name=(if x116 then 'Alice' else 'Anonymized'))
                       ∨ (context.viewer.role=Reviewer))
                       ∨ (context.viewer.role=PC))
                       ∨ ((context.stage=Public)
                         ∧ ¬((if x180 then Accepted else 'none')='none')))
                     ⇒ (x52=false))}
          ∪ {True ∧ context=(record viewer=(record name='Bob' role=Reviewer) stage=Public)}
    SYMBOLIC VALUE = (record title=(if x52 then 'MyPaper' else '')
                             author=(if x116 then 'Alice' else 'Anonymized')
                             accepted=(if x180 then Accepted else 'none')
                     )
  ]
```

The formatted output from invoking the Jeeves parser is a λ_J program that follows the specification in Definition 4.1 and Haskell 4.5. In appendix B.3 it is outlined how the λ_J output is formatted. We depict how to run the Jeeves parser and what the formatted λ_J program looks like for the Naming Policy Program from Figure 2. According to the theoretical program translation in Example 4.14, the program *exactly* parses to the expected λ_J terms!

```
Main>  parseFile "Tests/testp1.jeeves"
[(
letrec
 name = thunk ( (defer a in (assert (¬ (context='alice') ⇒ (a=false)) in
                                             (if a then 'Anonymous' else 'Alice'))) );
 msg = thunk ( ('Author is '+name) );
in
 print: concretize msg with 'alice' ;
 print: concretize msg with 'bob' ;
,"")]
```

Because of the verbose nature of the parsing step, we will sidestep the equivalent outcome from parsing the Conference Mangagement Program.

9 Conclusion

We have presented the first complete implementation of the *Jeeves evaluation engine*. "Complete" in the sense that the evaluation of a program written in Jeeves syntax is in fact defined in terms of the λ_J evaluation semantics, as is directly reflected in our implementation. "Not-complete", however, in the sense that a static (type) verification step currently has been omitted. As part of the process, we have specifically *obtained a tool that is able to generate privacy constraints for a given Jeeves program*. The actual constraint solving phase, however, has in accordance with Yang et al [23] been assumed to happen at a later time and is thus not part of our formalization efforts directly.

The implementation consists the following Haskell components:

- abstract Haskell type definitions to define a concrete Jeeves syntax as well as the λ_J syntax;
- an LL(1)-parser that builds abstract λ_J syntax trees from the Jeeves source-language, thus translating Jeeves to λ_J terms;
- a λ_J-interpreter, implementing the operational evaluation semantics of λ_J;
- an implementation of constraint evaluation as monadic operations on a monadic constraint environment.

With this implementation, we were able to both run and parse the canonical examples from Figure 2 and Figure 3 as they (almost) appear in the original paper by Yang et al [23] (after some syntactical corrections and adjustments) with the expected results. All in an easy-to-use fashion as explained in Section 8. We have achieved an elegant, yet precise program documentation by making use of Haskells' "literate" programming feature to incorporate the theoretical part of the report together with the actual program, ie, the source LaTeX of this report also serves as the source code of the program, as accounted for in Notation 1.1.

We have corrected a number of inconsistencies and shortcomings in the original syntax and semantics, together with certain limitations, in order to support an implementation, notably:

- added explicit syntax for a Jeeves and λ_J program;

- introduced explicit semantics for the letrec recursive operator in λ_J
- only allowing recursive functions at the top-level of a program;
- disallowing recursively defined policies;
- introduced explicit semantics for output side-effects;
- reformulated the dynamic operational semantics of λ_J to one that is entirely de-compositional and non-substitutional for convincingly proving program and privacy properties.
- identified the constraint set handling as being monadic with policies as the only constructs with side-effects on the constraint set (as expected).

We have published the implementation as a github project [17].

10 Future Directions

First of all, it is desirable to have the implementation "hooked up" to a constraint solver (with a Haskell interphase).

Even though the interpreter component of the implementation has the advantage of serving as a "proof of concept" as much as a practical, and theoretically transparent tool (the implementation of an operational semantics is by definition an interpreter), efficiency is of inherent concern. Efficiency can, in fact, be improved considerably by replacing the λ_J interpreter with a compilation step, that translates λ_J syntax trees to some efficient target code, whilst incorporating the semantic evaluation rules directly. Joelle Despeyreaux, for example, has outlined how to perform such a systematic translation from mini-ML, while incorporating the languages' operational semantics [4].

Redefining some of the Haskell parser mechanisms such as "++" is another area of optimization gains to explore. Because many of these pre-defined parser mechanisms allow backtracking, we have not been able to optimize our parser further, other than ensuring that the grammar productions that are parsed is on LL(1) form, which we found is not enough to avoid backtracking completely.

A study of how to optimize on the generated constraints prior to any automated constraint solving phase, could possible increase the efficiency (and correctness) of thereof.

A Discrepancies from the original formalization

In this section, we list the modifications and formalization decisions we have made compared to Yang et al [23] in order to clarify the syntax and semantics sufficiently to support an implementation.

A.1 Discrepancy (Jeeves syntax). The original abstract syntax *c.f.* Yang et al [23, Fig. 1] has been extended in several ways *c.f.* Definition 2.1:

- the syntax of a program has been made explicit,
- let statements are made an explit part of the program syntax,
- let statements only appear at the top-level of a program,
- a policy expression must contain an "`in`" part,

- the syntax of let expressions has been made explicit,
- the syntax for expression sequences has been made explicit,
- generalized level expressions has been made explicit,
- record and field expressions have been made explicit.

As a consequence of only allowing (recursively defined) let statements at the top-level of a Jeeves program, we obtain the following notable limitations:

- we disallow recursively defined functions in symbolic values,
- we disallow cyclic data structures.

Finally, we have added a *concrete syntax* for Jeeves programs in Definition 2.4.

A.2 Discrepancy (λ_J syntax). The original abstract syntax *c.f.* Yang et al [23, Fig. 2] has been extended in several ways *c.f.* Definition 3.1, Definition 3.3, Definition 3.5, as well as Definition 5.1:

- the syntax of a program has been made explicit,
- the recursive combinator 'letrec' has been added as a statement,
- the recursive combinator 'letrec' has been removed as an expression,
- output statements have been generalized,
- an explicit output tag to concretize statements has been added,
- the notion of a thunk expression has been added,
- the defer expression has been simplified (to reflect the translation),
- the assert expressions must contain an "in" part,
- the unit ('()') entity has been removed,
- records have been added as expressions (when their fields are expressions),
- field look-up has been added as an expression,
- concrete and symbolic values are not automatically defined as expressions.

As a consequence of only allowing letrec and output statements at the top-level of a λ_J program, we obtain the following notable limitations:

- a static, recursive scope of a program is only established at the top-level,
- a static, recursive scope of a program is established globally prior to side effect statements (output).

As mentioned, the category of concrete and symbolic normal forms is defined separately, though some syntactic entities appear both as an expression and as a value *c.f.* Definition 5.1:

- closures have been added as concrete values,
- strings and constants have been added as concrete values,
- records over concrete fields have been added as concrete value,
- records over symbolic fields have been added as symbolic value,
- field look-up over a symbolic record has been added as a symbolic value,

A.3 Discrepancy (λ_J translation). The original translation *c.f.* Yang et al [23, Fig. 6] has been extended in several ways *c.f.* Definition 4.1, Definition 4.6, and Definition 4.11:

- the translation of a Jeeves program has been added,
- the translation of expression sequences has been added,
- the translation of if expressions has been added,
- the translation of let expressions has been added,
- a generalization of the level expression translation has been added,
- the (trivial) "default" part has been removed,
- binary operator expression translation has been added,
- function application translation has been added,
- record translation has been added,
- field look-up translation has been added,
- translation of literals and 'context' has been added,
- translation of logical (unary) negation has been added,
- translation of (syntactic sugary) paranthesis has been added.

A.4 Discrepancy (evaluation semantics). The original evaluation semantics *c.f.* Yang et al [23, Fig. 3] has been extended and modified in several ways *c.f.* Definition 7.14, Definition 7.17 , and Definition 7.20:

- adding the notion of a binding environment (to manage evaluation scopes),
- reformulating the semantics as a least fixpoint semantics in the environment,
- formulating an evaluation semantics of a program (as a series of effects),
- reformulation from small-step to big-step semantics,
- reformulation from non-compositional to compositional semantics,
- reformulation from substitution-based to non-substitution based semantics,
- adding evaluation semantics for variable lookup,

- adding evaluation semantics for unary operation,
- added level variable handling to happen by the binding environment,
- added evaluation semantics for let expressions,
- added evaluation semantics for record expressions,
- added evaluation semantics for field look-up expressions.

We have furthermore added formalizations for the λ_J input-output domains (Definition 7.1), and for the pre-constraint-solve output effect from running a program prior to any constraint solving (Definition 7.11).

B Additional code

In this appendix we include various fragments of code that were not deemed key to the main presentation.

B.1 Haskell (Literal lexical token parsers).

```
spaces = many myspace -- white space and Haskell style comments in Jeeves
  where
    myspace = sat isSpace
              +++
              (do word "--"
                  many (sat (≢ '\n'))
                  return '␣')

ident :: Parser String   -- a lower case letter followed by alphanumeric chars
ident = do xs ← ident2
           if (isKeyword xs) then failure else return xs
  where
    ident2 = do x ← sat isLower
                xs ← many (sat isAlphaNum)
                return (x:xs)

isKeyword idkey = elem idkey keywords
keywords = ["top","bottom","if","then","else","lambda",
            "level","in","policy","error","context","let",
            "true","false","print","sendmail"]

nat :: Parser Int   -- a sequence of digits
nat = do xs ← many1 (sat isDigit )
         return (read xs)

string :: Parser String   -- strings can be in "" or ''.
string = do sat (≡ '"')
            s ← many (sat (≢ '"'))
            sat (≡ '"')
```

```
             return s
           +++
           do sat (≡ '\'')
              s ← many (sat (≢ '\''))
              sat (≡ '\'')
              return s

constant   = do x ← sat isUpper
                xs ← many (sat isAlphaNum)
                return (x:xs)
```

B.2 Haskell (parser framework).

```
data Parser a = PARSER (String →[(a, String)])

parse :: Parser a → String → [(a,String)]
parse (PARSER p) inp = p inp

parseFile filename = do jeeves ← readFile filename      -- IO utility
                        putStr (show (parse (programParser vars) jeeves))

instance Monad Parser where
  return v = PARSER (λinp →[(v,inp)])
  p ≫= f = PARSER (λinp →case parse p inp of
                          [] → []
                          [(v,out)] → parse (f v) out)

failure :: Parser a
failure = PARSER (λinp →[])

success :: Parser ()
success = PARSER (λinp →[((),inp)])

item :: Parser Char
item = PARSER (λinp →case inp of
                      "" → []
                      (x:xs) → [(x,xs)] )

-- choice operator
(+++) :: Parser a → Parser a → Parser a
p +++ q = PARSER (λinp →case parse p inp of
                          [] → parse q inp
                          [(v,out)] → [(v,out)])

-- token parser builder
wordToken :: String → a → Parser a  -- builds a token parser for a word tok to return r on
    success
wordToken tok r = do token (word tok)
                     return r
```

```
-- derived primitives
sat :: (Char → Bool) → Parser Char
sat p = do x ← item
           if p x then return x else failure

-- basic token definitions
token :: Parser a → Parser a
token p = do spaces
             v ← p
             spaces
             return v

word :: String → Parser String      -- parses just the argument characters, incl. white spaces
word []       = return []
word (c:cs)   = do sat (≡ c)
                   word cs
                   return (c:cs)

-- generic combinators
many :: Parser a → Parser [a]
many p = many1 p +++ return []

many1 :: Parser a → Parser [a]
many1 p = do v ←p
             vs ← many p
             return (v:vs)

optional :: Parser a → Parser [a]
optional p = optional1 p +++ return []

optional1 p = do v ← p
                 return [v]

manyParser :: (FreshVars → Parser a) → FreshVars → Parser b → Parser [a]
manyParser p xs sp = manyParser1 p xs sp +++ return []

manyParser1 p xs sp = (do v ← p xs1
                          vs ← manyParserTail p xs2 sp
                          return (v:vs))
  where (xs1,xs2) = splitVars xs

manyParserTail p xs sp = (do sp                       -- parses separation tokens like ; , ∘ etc
                             v ← p xs1
                             vs ← manyParserTail p xs2 sp
                             return (v:vs))
                         +++
                         return []
```

```
  where (xs1,xs2) = splitVars xs
```

B.3 Haskell (pretty-printing λ_J syntax).

```
instance Show Effect where
  show (EFFECT output (INSTANTIATE (MODEL delta sigma g c) v)) =
    "\n␣␣EFFECT␣" ++show output ++
    "\n␣␣␣␣SOFT␣CONSTR␣=␣" ++show delta ++"," ++
    "\n␣␣␣␣HARD␣CONSTR␣INST␣=␣" ++show sigma ++"␣∪␣{" ++show g ++"␣∧␣context=" ++
        show c ++"␣}" ++
    "\n␣␣␣␣SYMBOLIC␣VALUE␣=␣" ++show v ++"\n␣␣"

instance (Show a)⇒Show (Constraints a) where
  show (CONSTRAINTS sigma delta e) =
    "CONSTRAINTS" ++
    "\n␣␣SIGMA␣=␣" ++show sigma ++
    "\n␣␣DELTA␣=␣" ++show delta ++
    "\n␣␣" ++show e

instance Show Value where    -- pretty printing lambda J values
  show (V_BOOL b)         = if b then "true" else "false"
  show (V_NAT i)          = show i
  show (V_STR s)          = "'" ++s ++"'"
  show (V_CONST s)        = s
  show (V_ERROR)          = "error"
  show (V_LAMBDA x e rho) = "(\\"++show x++"."++show e++",RHO)"
  show (V_THUNK e rho)    = "(thunk␣RHO)"
  show (V_RECORD fivs)    = "(record" ++(if null fivs then "" else foldr1 (++) (map (λ(fi,e)
     → ("␣"++show fi++"="++show e)) fivs)) ++")"
  show (V_VAR x)          = show x
  show (V_CONTEXT)        = "context"
  show (V_OP op v1 v2)    = "("++show v1++show op++show v2++")"
  show (V_UOP uop v)      = show uop++show v
  show (V_IF v1 v2 v3)    = "(if␣" ++show v1 ++"␣then␣" ++show v2 ++"␣else␣" ++show
     v3 ++")"
  show (V_FIELD v fi)     = show v++"."++show fi

instance Show Exp where -- pretty printing lambda J expressions
  show (E_BOOL True)     = "true"
  show (E_BOOL False)    = "false"
  show (E_NAT n )        = show n
  show (E_STR s )        = "'" ++s ++"'"  -- todo: remove escape quotes
  show (E_CONST s)       = s              -- no quotes in a constant by definition
  show (E_VAR v )        = show v
  show (E_CONTEXT)       = "context"
  show (E_LAMBDA v e)    = "lambda␣" ++(show v) ++"." ++(show e)
  show (E_THUNK e)       = "thunk␣" ++"(␣" ++(show e) ++"␣)"
  show (E_OP op e1 e2)   = "(" ++show e1 ++show op ++show e2 ++")"
  show (E_UOP uop e)     = show uop ++"␣" ++show e
```

```
  show (E_IF e1 e2 e3)     = "(if " ++show e1 ++" then " ++show e2 ++" else " ++show e3
      ++")"
  show (E_APP e1 e2)       = "(" ++show_APP e1 ++" " ++show e2 ++")"
    where
      show_APP (E_APP e1 e2) = "("++ show_APP e1 ++" " ++show e2 ++")"
      show_APP e             = show e
  show (E_DEFER v e)       = "(defer " ++show v ++" in " ++show e ++")"
  show (E_ASSERT e1 e2) = "(assert " ++show e1 ++" in " ++show e2 ++")"
  show (E_LET x e1 e2)   = "(let " ++show x ++" = " ++show e1 ++" in " ++show e2 ++")
      "
  show (E_RECORD fies) = "(record" ++(if null fies then "" else foldr1 (++) (map (λ(fi,e)→
      (" "++show fi++"="++show e)) fies)) ++")"
  show (E_FIELD e fi)      = show e ++"." ++show fi

instance Show Binding where
  show (BIND x e) = " " ++show x ++" = " ++show e ++";\n"

instance Show Statement where
  show (CONCRETIZE_WITH output e1 e2) = " " ++output ++" (concretize " ++show e1 ++
      " with " ++show e2 ++") ;\n"

instance Show Program where
  show (P_LETREC ls ps) = "\nletrec\n" ++concat (map show ls) ++"in\n" ++concat (map
      show ps)

instance Show Op where
  show OP_PLUS = "+"
  show OP_MINUS = "-"
  show OP_AND = " ∧ "
  show OP_OR = " ∨ "
  show OP_IMPLY = " ⇒ "
  show OP_EQ = "="
  show OP_LESS = "<"
  show OP_GREATER = ">"

instance Show UOp where
  show OP_NOT = "¬"

instance Show Var where
  show (VAR s) = s

instance Show FieldName where
  show (FIELD_NAME s) = s

instance Show PathCondition where
    show (P_COND []) = "True"
    show (P_COND ps) = "∧"++ show ps
```

```
instance  Show Sigma where
   show (SIGMA list) = foldr  f  "{}"  list
      where
         f  (g,v)  s  =  s  ++"␣∪␣{"  ++show g++"␣⇒␣"++show v ++"}"

instance  Show Delta where
   show (DELTA list) = foldr  f  "{}"  list
      where
         f  (g,x,v)  s  = s ++"␣∪␣{"  ++show g ++"␣⇒␣"++ show x ++"="++show v++"}"

instance  Show Formula where
   show (F_IS v) = show v
   show (F_NOT v) = "¬" ++show v
```

References

[1] Grigoris Antoniou. A tutorial on default logics. *ACM Comput. Surv.*, 31(4):337–359, December 1999.

[2] Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors. *STACS '87—4th Annual Symposium on Theoretical Aspects of Computer Science*, volume 247 of *Lecture Notes in Computer Science*, Passau, Germany, February 1987. Springer.

[3] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *TACAS*, pages 337–340, 2008.

[4] Joëlle Despeyroux. Proof of translation in Natural Semantics. In *proceedings of the first Symp. on Logic In Computer Science, LICS'86*. IEEE Computer Society, June 1986. Also appears as INRIA Research Report RR-514, April 1986.

[5] Peter Naur et al. Report on the algorithmic language algol 60. *Communications of the ACM*, 6(1):1–17, January 1963.

[6] Simon Marlow et al. Haskell 2010 language report. Technical report, http://www.haskell.org, 2010.

[7] John Field and Michael Hicks, editors. *POPL 2012—Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Philadelphia, Pennsylvania, USA, January 2012. ACM.

[8] Carsten K. Gomard. A self-applicable partial evaluator for the lambda calculus: correctness and pragmatics. *Transactions on Programming Languages and Systems*, 14(2):147–172, 1992.

[9] P. Henderson. *Functional Programming—Application and Implementation*. Prentice-Hall, 1980.

[10] G. Hutton and E. Meijer. Monadic parsing in haskell. *Journal of Functional Programming*, 8(4):437–444, July 1998. Cambridge University Press, New York, NY, USA.

[11] Graham Hutton. Hugs system. Technical report, Glasgow, 2000.

[12] Graham Hutton. *Programming in Haskell*. Cambridge Univ Press, 2007.

[13] Gilles Kahn. Natural semantics. In Brandenburg et al. [2], pages 22–39.

[14] K. Marriott and P.J. Stuckey. *Programming with Constraints: An introduction*. MIT Press, Cambridge, Massachusetts, USA, 1998.

[15] Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990.

[16] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, DAIMI, Aarhus University, Aarhus, Denmark, 1981. Reprinted with corrections in *J. Log. Algebr. Program* **60-61**: 17-139 (2004).

[17] Eva Rose. Constraint generation for the jeeves privacy language. https://github.com/drevarose/jeeves-in-haskell/blob/master/jeeves-constraints.lhs, 2014.

[18] F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.

[19] Chiaki Sakama. Inductive negotiation in answer set programming. In Matteo Baldoni, Tran Cao Son, M. Birna van Riemsdijk, and Michael Winikoff, editors, *DALT*, volume 5397 of *Lecture Notes in Computer Science*, pages 143–160. Springer, 2008.

[20] David Schmidt. *Denotational Semantics*. Allyn and Bacon, 1986.

[21] Tom Schrijvers, Peter Stuckey, and Philip Wadler. Monadic constraint programming. *J. Funct. Program.*, 19(6):663–697, November 2009.

[22] Philip Wadler and Peter Thiemann. The marriage of effects and monads. *ACM Trans. Comput. Logic*, 4(1):1–32, January 2003.

[23] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. A language for automatically enforcing privacy policies. In Field and Hicks [7], pages 85–96.

Index