Formalization of the Catala language

Denis Merigoux, Nicolas Chataing

November 2020

Contents

1 Introduction

Tax law defines how taxes should be computed, depending on various characteristic of a fiscal household. Government agencies around the world use computer programs to compute the law, which are derived from the local tax law. Translating tax law into an unambiguous computer program is tricky because the law is subject to interpretations and ambiguities. The goal of the Catala domain-specific language is to provide a way to clearly express the interpretation chosen for the computer program, and display it close to the law it is supposed to model.

To complete this goal, our language needs some kind of *locality* property that enables cutting the computer program in bits that match the way the legislative text is structured. This subject has been extensively studied by Lawsky [4, 3, 2], whose work has greatly inspired our approach.

The structure exhibited by Lawsky follows a kind of non-monotonic logic called default logic [5]. Indeed, unlike traditional programming, when the law defines a value for a variable, it does so in a *base case* that applies only if no *exceptions* apply. To determine the value of a variable, one needs to first consider all the exceptions that could modify the base case.

It is this precise behavior which we intend to capture when defining the semantics of Catala.

2 Default calculus

We choose to present the core of Catala as a lambda-calculus augmented by a special "default" expression. This special expression enables dealing with the logical structure underlying tax law. Our lambda-calculus has only unit and boolean values, but this base could be enriched with more complex values and traditional lambda-calculus extensions (such as algebraic data types or Λ-polymorphism).

2.1 Syntax

Type	τ	$::=$	$\texttt{bool} \mid \texttt{unit}$	boolean and unit types
		$\mid$	$\tau \rightarrow \tau$	function type
Expression	e	$::=$	$x \mid \texttt{true} \mid \texttt{false} \mid ()$	variable, literal
		$\mid$	$\lambda\ (x : \tau) .\ e \mid e\ e$	λ-calculus
		$\mid$	d	default term
Default	d	$::=$	$\langle e :\!-\ e \mid [e^*] \rangle$	default term
		$\mid$	$\circledast$	conflict error term
		$\mid$	$\varnothing$	empty error term

Compared to the regular lambda calculus, we add a construction coming from default logic. Particularly, we focus on a subset of default logic called categorical, prioritized default logic [1]. In this setting, a default is a logical rule of the form $A :\!- B$ where A is the justification of the rule and B is the consequence. The rule can only be applied if A is consistent with the current knowledge W: from $A \wedge W$, one should not derive $\bot$. If multiple rules $A :\!- B_1$ and $A :\!- B_2$ can be applied at the same time, then only one of them is applied through an explicit ordering of the rules.

To incorporate this form of logic inside our programming language, we set A to be an expression that can be evaluated to `true` or `false`, and B the expression that the default should reduce to if A is true. If A is false, then we look up for other rules of lesser priority to apply. This priority is encoded trough a syntactic tree data structure[1]. A node of the tree contains a default to consider first, and then a list of lower-priority defaults that don't have a particular ordering between them. This structure is sufficient to model the base case/exceptions structure or the law, and in particular the fact that exceptions are not always prioritized in the legislative text.

In the term $\langle e_{\text{just}} :\!- e_{\text{cons}} \mid e_1, \ldots, e_n \rangle$, e_{just} is the justification A, e_{cons} is the consequence B and $e_1, \ldots, e_n$ is the list of rules to be considered if e_{just} evaluates to `false`.

Of course, this evaluation scheme can fail if no more rules can be applied, or if two or more rules of the same priority have their justification evaluate to `true`. The error terms $\circledast$ and $\varnothing$ encode these failure cases. Note that if a Catala program correctly derived from a legislative source evaluates to $\circledast$ or $\varnothing$, this could mean a flaw in the law itself. $\varnothing$ means that the law did not specify what happens in a given situation, while $\circledast$ means that two or more rules specified in the law conflict with each other on a given situation.

2.2 Typing

Our typing strategy is an extension of the simply-typed lambda calculus. The typing judgment $\boxed{\Gamma \vdash e : \tau}$ reads as "under context Γ, expression e has type τ".

Typing context	Γ	$::=$	$\varnothing$	empty context
(unordered map)		$\mid$	$\Gamma, x : \tau$	typed variable

[1]Thanks to Pierre-Évariste Dagand for this insight.

We start by the usual rules of simply-typed lambda calculus.

T-UnitLit $\Gamma \vdash () : \texttt{unit}$

T-TrueLit $\Gamma \vdash \texttt{true} : \texttt{bool}$

T-FalseLit $\Gamma \vdash \texttt{false} : \texttt{bool}$

T-Var $\Gamma, x : \tau \vdash x : \tau$

T-Abs
$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda\ (x : \tau).\ e : \tau \to \tau'}$$

T-App
$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_1}$$

Then we move to the special default terms. First, the error terms that stand for any type.

ConflictError $\Gamma \vdash \circledast : \tau$

EmptyError $\Gamma \vdash \varnothing : \tau$

Now the interesting part for the default terms. As mentioned earlier, the justification e_{just} is a boolean, while e_{cons} can evaluate to any value. DefaultBase specifies how the tree structure of the default should be typed.

T-Default
$$\frac{\Gamma \vdash e_{\text{just}} : \texttt{bool} \qquad \Gamma \vdash e_{\text{cons}} : \tau \qquad \Gamma \vdash e_1 : \tau \qquad \cdots \qquad \Gamma \vdash e_n : \tau}{\Gamma \vdash \langle e_{\text{just}} :- e_{\text{cons}} \mid e_1, \ldots, e_n \rangle : \tau}$$

The situation becomes more complex in the presence of functions. Indeed, want our default expressions to depend on parameters. By only allowing e_{just} to be `bool`, we force the user to declare the parameters in a λ that wraps the default from the outside. Using this scheme, all the expressions inside the tree structure of the default will depend on the same bound variable x.

2.3 Evaluation

We give this default calculus small-step, structured operational semantics. The one-step reduction judgment is of the form $\boxed{e \longrightarrow e'}$.

In our simple language, values are just booleans, functions or error terms. We use a evaluation contexts to efficiently describe the evaluation order. Evaluation contexts are expression with a hole indicating the sub-term currently being reduced.

Values	v	$::=$	$\lambda\ (x : \tau).\ e$	functions
		$\mid$	$\texttt{true} \mid \texttt{false}$	booleans
		$\mid$	$\circledast \mid \varnothing$	errors
Evaluation	C_λ	$::=$	$\cdot\ e \mid v\ \cdot$	function application
contexts		$\mid$	$\langle \cdot :- e \mid [e^*] \rangle$	default justification evaluation
	C	$::=$	C_λ	regular contexts
		$\mid$	$\langle \texttt{true} :- \cdot \mid [e^*] \rangle$	default consequence evaluation
		$\mid$	$\langle \texttt{false} :- e \mid [v^*], \cdot, [e^*] \rangle$	sub-default evaluation

We choose a call-by-value reduction strategy. First, we present the usual reduction rules for beta-reduction and evaluation inside a context hole. Note that D-Context does not deal with error terms, which will have a special treatment for error propagation later.

D-Context
$$\frac{e \longrightarrow e' \qquad e' \notin \{\circledast, \varnothing\}}{C[e] \longrightarrow C[e']}$$

D-β
$$(\lambda\ (x : \tau).\ e)\ v \longrightarrow e[x \mapsto v]$$

Now we have to describe how the default terms reduce. Thanks to a the D-Context rule, we can suppose that the justification of the default is already reduced to a variable v. By applying the beta-reduction rule D-β, we can further reduce to the case where v is a boolean. This is where we encode our default logic evaluation semantics. If v is `true`, then this rule applies and we reduce to the consequence. We don't even have to consider rules of lower priority lower in the tree. This behavior is similar to short-circuit reduction rules of boolean operators, that enable a significant performance gain at execution.

$$\textsc{D-DefaultTrueNoError}\quad \frac{v \neq \varnothing}{\langle \texttt{true} :- v \mid e_1, \dots, e_n \rangle \longrightarrow v}$$

However, if the consequence of the first default evaluates to $\varnothing$, then we fall back on the rules of lower priority, as if the justification had evaluated to `false`. This behavior is useful when the consequence of the first default is itself a "high-priority" default tree. In this case, $e_1, \dots, e_n$ acts a the "low-priority" default tree. The chaining behavior from the high-priority tree to the low-priority tree defined by D-DefaultTrueError, will be very useful in §3.3.

$$\textsc{D-DefaultTrueError}\quad \langle \texttt{true} :- \varnothing \mid e_1, \dots, e_n \rangle \longrightarrow \langle \texttt{false} :- \varnothing \mid e_1, \dots, e_n \rangle$$

If the consequence of the default is `false`, then we have to consider rules of lower priority $e_1, \dots, e_n$ that should be all evaluated (left to right), according to the sub-default evaluation context. Then, we consider all the values yielded by the sub-default evaluation and define two functions over these values. Let $\mathsf{empty\_count}(v_1, \dots, v_n)$ returns the number of empty error terms $\varnothing$ among the values. We then case analyze on this count:

- if $\mathsf{empty\_count}(v_1, \dots, v_n) = n$, then none of the sub-defaults apply and we return $\varnothing$;
- if $\mathsf{empty\_count}(v_1, \dots, v_n) = n - 1$, then only only one of the sub-default apply and we return its corresponding value;
- if $\mathsf{empty\_count}(v_1, \dots, v_n) < n - 1$, then two or more sub-default apply and we raise a conflict error $\circledast$.

$$\textsc{D-DefaultFalseNoSub}\quad \langle \texttt{false} :- e \mid \varnothing, \dots, \varnothing \rangle \longrightarrow \varnothing$$

$$\textsc{D-DefaultFalseOneSub}\quad \langle \texttt{false} :- e \mid \varnothing, \dots, \varnothing, v, \varnothing, \dots, \varnothing \rangle \longrightarrow v$$

$$\textsc{D-DefaultFalseSubConflict}\quad \frac{\mathsf{empty\_count}(v_1, \dots, v_n) < n - 1}{\langle \texttt{false} :- e \mid v_1, \dots, v_n \rangle \longrightarrow \circledast}$$

Last, we need to define how our error terms propagate. Because the rules for sub-default evaluation have to count the number of error terms in the list of sub-defaults, we cannot always immediately propagate the error term $\varnothing$ in all the evaluation contexts as it usually done. Rather, we rely on the distinction between the λ-calculus evaluation contexts C_λ and the sub-default evaluation context. Hence the following rules for error propagation:

$$\textsc{D-ContextEmptyError}\quad \frac{e \longrightarrow \varnothing}{C_\lambda[e] \longrightarrow \varnothing} \qquad \textsc{D-ContextConflictError}\quad \frac{e \longrightarrow \circledast}{C[e] \longrightarrow \circledast}$$

3 Scope language

Our core default calculus provides a value language adapted to the drafting style of tax law. Each article of the law will provide one or more rules encoded as defaults. But how to collect those defaults into a single expression that will compute the result that we want? How to reuse existing rules in different contexts?

These question point out the lack of an abstraction structure adapted to the legislative drafting style. Indeed, our λ functions are not convenient to compose together the rules scattered around the legislative text. Moreover, the abstractions defined in the legislative text exhibit a behavior quite different from λ functions.

First, the blurred limits between abstraction units. In the legislative text, objects and data are referred in a free variable style. It is up to us to put the necessary bindings for these free variables, but it is not trivial to do so. For that, one need to define the perimeter of each abstraction unit, a legislative *scope*, which might encompass multiple articles.

Second, the confusion between local variables and function parameters. The base-case vs. exception structure of the law also extends between legislative scopes. For instance, a scope A can define a variable x to have value a, but another legislative scope B can *call into* A but specifying that x should be b. In this setting, B defines an exception for x, that should be dealt with using our default calculus.

Based on these two characteristic, we propose a high-level *scope language*, semantically defined by its encoding in the default calculus.

3.1 Syntax

A scope S is a legislative abstraction unit that can encompass multiple articles. S is comprised of multiple rules that define a scope variable a to a certain expression under a condition that characterize the base case or the exception.

S can also call into another scope S', as a function can call into another. These calls are scattered in the legislative texts and have to be identified by the programmer. Since S can call S' multiple times with different "parameters", we have to distinguish between these sub-call and give them different names S'_1, S'_2, etc. A program P is a list of scope declarations σ.

Scope name	S			
Scope call identifier	n			
Location	ℓ	$::=$	a	scope variable
		$\mid$	$S_n[a]$	sub-scope call variable
Expression	e	$::=$	ℓ	location
		$\mid$	$\cdots$	default calculus expressions
Rule	r	$::=$	$\texttt{rule}\ \ell : \tau = \langle e :\!- e \mid [e^*]\rangle$	Location definition
		$\mid$	$\texttt{call}\ S_n$	sub-scope call
Scope declaration	σ	$::=$	$\texttt{scope}\ S : [r^*]$	
Program	P	$::=$	$[\sigma^*]$	

3.2 Running example

Let's illustrate how the scope language plays out with a simple program that calls a sub-scope:

`Simple scope program`

```
1  scope X:
2    rule a = < true :- 0 | >
3    rule b = < true :- a + 1 | >
4
5  scope Y:
6    rule X_1[a] = < true :- 42 | >
7    call X_1
8    rule c = < X_1[b] == 1 :- false | >,  < X_1[b] == 43 :- true | >
```

Considered alone, the execution `X` is simple: `a` and `b` are defined by a single default whose justification is `true`. Hence, `a` should evaluate to `0` and `b` should evaluate to `1`.

Now, consider scope `Y`. It defines a single variable `c` with two defaults lines 8 and 9, but the justifications for these two defaults use the result of the evaluation (line 7) of variable `b` of the sub-scope `X_1`. Line 6 shows an example of providing an "argument" to the subscope call. The execution goes like this: at line 7 when calling the sub-scope, `X_1[a]` has two defaults, one coming from line 2, the other calling from line 6. Because the caller has priority over the callee, the default from line 6 wins and `X_1[a]` evaluates to `42`. Consequently, `X_1[b]` evaluates to `43`. This triggers the second default in the list of line 9 which evaluates `c` to `true`.

The goal is to provide an encoding of the scope language into the lambda calculus that is compatible with this intuitive description of how scopes should evaluate. To get a high-level picture of the translation, we first show what the previous simple program will translate to, using ML-like syntax for the target default calculus:

`Simple default program`

```
1   let X (a: unit -> int) (b: unit -> int) : (int * int) =
2     let a : unit -> int = a ++ (fun () -> < true :- 0 | >) in
3     let a : int = a () in
4     let b : unit -> int = b ++ (fun () -> < true :- a + 1 | >) in
5     let b : int = b () in
6     (a, b)
7
8   let Y (c: unit -> bool) : bool =
9     let X_1[a] : unit -> int = fun () -> < true :- 42 | > in
10    let X_1[b] : unit -> int = fun () -> EmptyError in
11    let (X_1[a], X_1[b]) : int * int = X(X_1[a], X_1[b]) in
12    let c : unit -> bool = c ++ (fun () ->
13      < X_1[b] == 1 :- false>, < X_1[b] == 43 :- true >)
14    in
15    let c : bool = c () in
16    c
```

We start unravelling this translation with the scope `X`. `X` has been turned into a function whose arguments are all the local variables of the scope. However, the arguments have type `unit -> <type>`. Indeed, we want the arguments of `X` (line 1) to be the default expression supplied by the caller of `X`, which will later be merged with a higher priority (operator `++`) to the default expression defining the local variables of `X` (lines 2 and 4). But since defaults are not values in our default calculus, we have to thunk them in the arguments of `X` so that their evaluation is delayed. After the defaults have been merged, we apply `()` to the thunk (lines 3 and 5) to force evaluation and get back the value. Finally, X returns the tuple of all its local variables (line 6).

The translation of Y exhibits the pattern for sub-scope calls. Lines 9 translates the assignment of the sub-scope argument X_1[a]. Before calling X_1 (line 11), the other argument X_1[b] is initialized to the neutral $\varnothing$ that will be ignored at execution because X provides more defaults for b. The sub-scope call is translated to a regular function call (line 11). The results of the call are then used in the two defaults for c (lines 13), which have been turned into a default tree taking into account that no priority has been declared between the two defaults. Finally, c is evaluated (line 15).

3.3 Formalization of the translation

The main judgment of reduction from scope language to default calculus is $\boxed{P \vdash \sigma \rightsquigarrow e \Rrightarrow \Delta_{\text{own}}}$, which reduces a scope declaration to a function in the default calculus, while providing the list of its own variables.

Translation context	Δ	$::=$	$\varnothing$	empty context
(unordered map)		$\mid$	$\Delta_{\text{own}}, \Delta_{\text{sub}}$	own and sub-scopes contexts
	Δ_{own}	$::=$	$\varnothing \mid \Delta_{\text{own}}, a : \tau$	typed scope variable
	Δ_{sub}	$::=$	$\varnothing \mid \Delta_{\text{sub}}, S_n[a] : \tau$	typed sub-scope variable

The translation context Δ is similar to the typing context Γ of the default calculus, but it only takes into account the new scope-related location. At any point, Δ will contain the scope locations defined (and usable in expressions) so far. Δ is divided in Δ_{own} and Δ_{sub}, which contain respectively the scope's own variables and the variables of its sub-scopes.

We will describe the translation from top to bottom, in order to keep the big picture in mind. We will assume the default calculus has been expanded with the usual ML `let in` construction, as well as tuples. Here is the top-level rule for translating scopes.

T-Scope
$$\frac{P; \varnothing \vdash_S r_1, \ldots, r_n \rightsquigarrow e \Rrightarrow a_1 : \tau_1, \ldots, a_m : \tau_m, \Delta_{\text{sub}}}{\begin{array}{c} P \vdash \texttt{scope}\ S : r_1, \ldots, r_n \rightsquigarrow \\ \texttt{let}\ S\ (a_1 : \texttt{unit} \rightarrow \tau_1) \cdots (a_m : \texttt{unit} \rightarrow \tau_m) : (\tau_1 * \cdots * \tau_m) = e[\cdot \mapsto (a_1, \ldots, a_m)] \Rrightarrow \\ a_1 : \tau_1, \ldots, a_m : \tau_m \end{array}}$$

This rule has a lot to unpack, but it is just the formal description of the translation scheme described earlier. To translate scope declaration S with associated rules $r_1, \ldots, r_n$, we use a helper judgment $\boxed{P; \Delta \vdash_S r_1, \ldots, r_n \rightsquigarrow e \Rrightarrow \Delta'}$ which reads as "given a program P and a translation context Δ, the rules $r_1, \ldots, r_n$ belonging to scope S translate to the expression e, producing a new typing context Δ'". In this T-Scope rule, we isolate in the resulting Δ' all the scope variables $a_1,\ldots,a_m$ from the sub-scope variables. Indeed, those variable will be the arguments and the return values of the function corresponding to the scope S. The expression e that stands for rules $r_1, \ldots, r_n$ is a series of `let` bindings, the last one finishing by a hole $(\cdot)$. We use this hole as a placeholder to be filled with the return value of the function, which is the tuple $(a_1, \ldots, a_n)$. Note that in accordance to the translation scheme and the need for a delayed evaluation of defaults, the arguments of S have a thunked type.

T-Rules
$$\frac{P; \Delta \vdash_S r_1 \rightsquigarrow e_1 \Rrightarrow \Delta' \qquad P; \Delta' \vdash_S r_2, \ldots, r_n \rightsquigarrow e_2 \Rrightarrow \Delta''}{P; \Delta \vdash_S r_1, \ldots, r_n \rightsquigarrow e_1[\cdot \mapsto e_2] \Rrightarrow \Delta''}$$

The translation of the sequence of rules consists of chaining the different `let in` expressions together with the same hole $(\cdot)$ substitution as the previous rule. Now, we can define the translation for individual rules, starting with the definitions of scope variables.

$$\textsc{T-DefScopeVar}\quad \frac{a \notin \Delta \qquad \Delta \vdash \langle e_{\text{just}} \mathrel{:-} e_{\text{cons}} \mid e_1, \dots, e_n \rangle : \tau}{\begin{array}{c} P; \Delta \vdash_S \texttt{rule}\ a : \tau = \langle e_{\text{just}} \mathrel{:-} e_{\text{cons}} \mid e_1, \dots, e_n \rangle \rightsquigarrow \\ \texttt{let}\ a : \tau = (a \mathbin{++} \lambda\,(() : \texttt{unit}).\ \langle e_{\text{just}} \mathrel{:-} e_{\text{cons}} \mid e_1, \dots, e_n \rangle)\ ()\ \texttt{in}\ \cdot \Rrightarrow a : \tau, \Delta \end{array}}$$

The premise of T-DefScopeVar, $a \notin \Delta$, indicates that our scope language allows each scope variable to be defined only once, with one default tree. This single default tree can incorporate multiple prioritized definitions of the same variable scattered around various legislative articles, but we assume in our scope language that these scattered definitions have been already collected. Therefore, the ordering of rules is very important in our scope language, because it should be compatible with the dependency graph of the scope locations. As the underlying default calculus is decidable and does not allow fixpoint definitions, the dependency graph of the scope locations should not be cyclic and therefore the topological ordering of its nodes should correspond to the order of the rules inside the scope declaration. This dependency ordering is enforced by the premise $\Delta \vdash \langle e_{\text{just}} \mathrel{:-} e_{\text{cons}} \mid e_1, \dots, e_n \rangle : \tau$, which seeds the typing judgment of §2.2 with Δ (the scope locations defined so far).

Since scope variables are also arguments of the scope, T-DefScopeVar redefines a by merging the new default tree with the default expression a of type $\texttt{unit} \rightarrow \tau$ passed as an argument to S. The merging operator, $\mathbin{++}$, has the following definition :

$$\begin{array}{c} \texttt{let}\ (\mathbin{++})\ (e_1 : \texttt{unit} \rightarrow \tau)\ (e_2 : \texttt{unit} \rightarrow \tau) : \texttt{unit} \rightarrow \tau = \\ \langle \lambda\,(() : \texttt{unit}).\ \texttt{true} \mathrel{:-} e_1 \mid e_2 \rangle \end{array}$$

$\mathbin{++}$ is asymetric in terms of priority: in $e_1 \mathbin{++} e_2$, e_1 will have the highest priority and it is only in the case where none of the rules in e_1 apply that e_2 will be considered. In T-DefScopeVar, the left-hand-side of $\mathbin{++}$ is coherent with our objective of giving priority to the caller.

Finally, the evaluation of the merged default tree is forced by applying $()$, yielding a value of type τ for a which will be available for use in the rest of the translated program. Now that we have presented the translation scheme for rules defining scope variables, we can switch to the translation of sub-scope variables definitions and calls. We will start by the rules that define sub-scope variables, prior to calling the associated sub-scope.

$$\textsc{T-DefSubScopeVar}\quad \frac{S \neq S' \qquad S'_n[a] \notin \Delta \qquad \Delta \vdash \langle e_{\text{just}} \mathrel{:-} e_{\text{cons}} \mid e_1, \dots, e_n \rangle : \tau}{\begin{array}{c} P; \Delta \vdash_S \texttt{rule}\ S'_n[a] : \tau = \langle e_{\text{just}} \mathrel{:-} e_{\text{cons}} \mid e_1, \dots, e_n \rangle \rightsquigarrow \\ \texttt{let}\ S'_n[a] : \texttt{unit} \rightarrow \tau = \lambda\,(() : \texttt{unit}).\ \langle e_{\text{just}} \mathrel{:-} e_{\text{cons}} \mid e_1, \dots, e_n \rangle\ \texttt{in}\ \cdot \Rrightarrow \\ S'_n[a] : \texttt{unit} \rightarrow \tau, \Delta \end{array}}$$

This rule is very similar to T-DefScopeVar, and actually simpler. It does not feature the merge operator $\mathbin{++}$, because sub-scope variables like $S'_n[a]$ are not part of the scope's argument. The premise $S \neq S'$ means that a scope S cannot have a recursive definition; it cannot call into itself and define sub-scope variables of its own scope. Note that $S'_n[a] : \texttt{unit} \rightarrow \tau$ is added to Δ in the final part of the judgment; $S'_n[a]$ has been defined as a sub-scope argument but not as a value that can be used by the scope yet, its type is $\texttt{unit} \rightarrow \tau$ and not τ.

When all the arguments of sub-scope S' have been defined using, T-DefSubScopeVar, the sub-scope itself can be called.

T-SubScopeCall

$$\frac{S \neq S' \qquad P(S') = \sigma' \qquad P \vdash \sigma' \rightsquigarrow e' \Rrightarrow a'_1 : \tau'_1, \ldots, a'_n : \tau'_n, \Delta'_{\text{sub}} \qquad \mathsf{init\_subvars}(\Delta;\, S'_n\texttt{[}a'_1\texttt{]}, \ldots, S'_n\texttt{[}a'_n\texttt{]}) = e_{\text{init}}}{\begin{array}{c} P; \Delta \vdash_S \texttt{call}\ S'_n \rightsquigarrow e_{\text{init}}[\cdot \mapsto \texttt{let}\ \texttt{(}S'_n\texttt{[}a'_1\texttt{]},\ldots,S'_n\texttt{[}a'_n\texttt{])}\ \texttt{:}\ \texttt{(}\tau'_1\ \texttt{*}\ \cdots\ \texttt{*}\ \tau'_n\texttt{)}\ \texttt{=} \\ e'\ \texttt{(}S'_n\texttt{[}a'_1\texttt{])} \cdots \texttt{(}S'_n\texttt{[}a'_n\texttt{])}\ \texttt{in}\ \cdot\,] \Rrightarrow S'_n\texttt{[}a'_1\texttt{]} : \tau'_1, \ldots, S'_n\texttt{[}a'_n\texttt{]} : \tau'_n, \Delta \end{array}}$$

Again, this rule has a lot to unpack, but is meant as a generalization of the translation scheme illustrated in §3.2. Let us start with the premises. As earlier, $S \neq S'$ means that scope declarations cannot be recursive. Next, we fetch the declaration σ' of S' inside the program P. σ' is reduced into the function expression e', whose arguments correspond to the scope variables of S': $a'_1, \ldots, a'_n$. Then, we need to define all the arguments necessary to call e'. Some of these arguments have been defined earlier in the translation, and they were added to Δ. But some arguments may not have been defined yet, and is its precisely the job of the init_subvars helper to produce the e_{init} expression to define those missing arguments with the $\varnothing$ value.

The conclusion of T-SubScopeCall defines the reduction of `call` S'_n. After e_{init}, we translate the sub-scope call to the default calculus call of the corresponding expression e', which takes as arguments the defaults and returns the corresponding values after evaluation. Finally, the new translation context produced is Δ augmented with all the variables of sub-scope S', who are available for use in later definitions of the scope.

The last item we need to define in order to complete the translation is init_subvars. Its definition is quite simple, since it produces an expression defining to $\varnothing$ all the variables from a list not present in Δ.

T-InitSubVarsInDelta

$$\frac{S'_n\texttt{[}a_1\texttt{]} : \tau_1 \in \Delta \qquad \mathsf{init\_subvars}(\Delta;\, S'_n\texttt{[}a_2\texttt{]}, \ldots, S'_n\texttt{[}a_n\texttt{]}) = e}{\mathsf{init\_subvars}(\Delta;\, S'_n\texttt{[}a_1\texttt{]}, \ldots, S'_n\texttt{[}a_n\texttt{]}) = e}$$

T-InitSubVarsNotInDelta

$$\frac{S'_n\texttt{[}a_1\texttt{]} : \tau_1 \notin \Delta \qquad \mathsf{init\_subvars}(\Delta;\, S'_n\texttt{[}a_2\texttt{]}, \ldots, S'_n\texttt{[}a_n\texttt{]}) = e'}{\begin{array}{c}\mathsf{init\_subvars}(\Delta;\, S'_n\texttt{[}a_1\texttt{]}, \ldots, S'_n\texttt{[}a_n\texttt{]}) = \\ \texttt{let}\ S'_n\texttt{[}a_1\texttt{]}\ \texttt{:}\ \texttt{unit} \to \tau_1\ \texttt{=}\ \lambda\ \texttt{(()}\ \texttt{:}\ \texttt{unit).}\ \varnothing\ \texttt{in}\ e\end{array}}$$

T-InitSubVarsEmpty

$$\mathsf{init\_subvars}(\Delta) = \cdot$$

References

[1] Gerhard Brewka and Thomas Eiter. “Prioritizing Default Logic”. In: *Intellectics and Computational Logic: Papers in Honor of Wolfgang Bibel*. Ed. by Steffen Hölldobler. Dordrecht: Springer Netherlands, 2000, pp. 27–45. ISBN: 978-94-015-9383-0. DOI: 10.1007/978-94-015-9383-0_3. URL: https://doi.org/10.1007/978-94-015-9383-0_3.

[2] Sarah B Lawsky. “Form as Formalization”. In: *Ohio State Technology Law Journal* (2020).

[3] Sarah B. Lawsky. “A Logic for Statutes”. In: *Florida Tax Review* (2018).

[4] Sarah B. Lawsky. “Formalizing the Code”. In: *Tax Law Review* 70.377 (2017).

[5] R. Reiter. “Readings in Nonmonotonic Reasoning”. In: ed. by Matthew L. Ginsberg. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1987. Chap. A Logic for Default Reasoning, pp. 68–93. URL: http://dl.acm.org/citation.cfm?id=42641.42646.