Introduction to logical frameworks

Brigitte Pientka

School of Computer Science
McGill University
Montreal, Canada

So far we have explored basic principles underlying programming language design. A formal definition of a programming language provides a precise specification not only for programmers, but also for implementors of these languages. Moreover, it allows the rigorous analysis of its properties. But a language definition is also an intricate artifact, which is carefully designed, and the proof of its properties are often complex and subtle – above all many cases are tedious. How then can we trust our language design? How can we trust that indeed the properties we claim about a language are true? How do we know that a given program indeed satisfies a certain safety property? How can we compare different properties? – Realistic languages have many cases to be considered, and while many of them will be straightforward, the task of verifying them all can be complex. Consequently it can be difficult to define a language correctly, and prove the appropriate theorems – let alone maintain the definition and the associated proofs when the language evolves and changes.

Fortunately, the burden can be alleviated by mechanizing the definition of a language together with its meta-theory. In this note, we will give a brief introduction to the logical framework LF and its implementation in the Twelf system – a programming environment which supports the implementation of language definitions and their meta-theory. It supports the definition of formal systems given via axioms and inference rules, as well as proofs about these formal systems.

1 Mechanizing Definitions

Language formalization frequently start an informal, on-paper definition of the language. It mostly consists of 3 distinct parts:

- Represent the grammar/ syntax of a language
- Represent its operational and static semantic
- Represent its meta-theory, i.e. proofs about the semantics such as progress and preservation.

Each layer brings up different questions we must address. The choice we make in each of the questions substantially influences how easy it is to attack the next layer.

1.1 How to represent the syntax of a language?

If we think about representing the grammar of a given language, one of the first questions we must address is the one related to variable binding. How should we represent bound variables which arise in the object language? – Related is the question to substitution. Do we need to implement renaming and substitution operations explicitly or are these operations provided "for free" by the framework?

Concrete Approaches In concrete approaches we typically represent variables using names of numbers. Capture avoiding substitution must then be defined explicitly as a function on terms, and in the case where bound variables are named, alpha-equivalence must also be defined explicitly. Concrete approaches can be subdivided roughly into 3 categories: 1) names to represent variables 2) de Bruijn indices 3) distinguishing bound and free variables. More importantly, once we start reasoning about our language, we may need to verify explicitly that the only terms we characterize are indeed the ones which are well-formed. This is often done using explicit well-formedness predicates which clutter the subsequent development .

Nominal approaches These approaches provide another way to address the problem of alpha conversion inherent in the named representation. It leaves however the task of implementing substitution to the user. There are also various problems in some systems that names can escape their scope – something which is really detrimental.

Higher-order approaches In the higher-order approach, we represent binding-constructs using lambda-abstraction in the meta-language. In other words, our meta-language is so powerful that it provides us with lambda-abstraction, which we use to represent binding constructs. This is possible, if we choose as our representation language the lambda-calculus. In this approach we typically do not need to explicitly implement predicates which check that all expressions are well-formed – instead we can prove that the encoding is indeed adequate separately.

1.2 How to represent operational and static semantics?

We face again the question whether to follow a first-order or a higher-order approach. Recall, our definition of typing rules. The typing judgement says "$\Gamma \vdash t : T$" which means given the assumptions $\Gamma = x_1{:}T_1, \ldots, x_n{:}T_n$ we can establish that the term t has type T. When we implement typing rules, we must decide how to handle the context Γ. How should we handle assumptions? – In a first-order approach we may choose to represent the assumptions as a list together with operations such as looking up an element in a list. In a higher-order approach, we represent hypothetical and parameteric derivations as functions.

1.3 How can we verify and construct proofs about formal systems?

This is the last and hardest question. The choices we made when we represented the syntax and the semantics of a language, will directly influence our ability to verify and construct proofs about it. Ideally, we would like to be as close to informal practice as possible, but there is a tension between what is desirable and what is possible.

If we choose first-order representations of our language, we can simply use existing theorem provers to verify the properties about our language – however, these proofs get large, are extremely tedious, and do not scale well.

If we choose higher-order representations of our language, then there are two approaches possible: 1) we implement the proof as a relation and post-hoc verify that the relation establishes a total function. 2) we use a theorem prover to construct the proof. We will focus on the first of these approaches, since it is more mature, and has been used extensively in the formalization of SML, foundational proof carrying code, and many other projects.

Questions we should keep in mind when we formalize these meta-theoretic properties are: How do we do induction? Case analysis? Substitution lemmas?

Adequacy An important question we must keep in mind in this endeavour is the following: What does then does it mean to correctly represent such a language definition in a formal framework? – In general, we aim for a *adequate representation*, i.e. the objects represented formally in the framework describe exactly those we were talking about on paper. More precisely, the representation of the language is isomorphic to the informal definition of the language we had on paper. But in practice we even want more: we want that the structure of the language is preserved as well – this will mean that we want a bi-jection (in fact a compositional bijection).

To establish adequacy, we require two tools: 1) Adequacy proof: induction proofs on the canonical forms of LF 2) Modularity of adequacy proofs based on subordination (we must understand under what assumptions/circumstances is an encoding adequate and when it is adequate with respect to one set of assumptions, how do we know it remains adequate given some other set of assumptions)

While we omit here a detailed discussion of how to prove adequacy, we will refer the interested reader to the article (**?**).

2 Simply typed subset of LF

We begin with a simply-typed lambda calculus which is essentially a subset of the logical framework LF **?**. We will only characterize normal forms in the simply-typed lambda calculus, since only those will characterize adequately our on paper formalization and hence only those are representing meaningful terms in our object-language (i.e. on paper formulation).

Types	A, B, C	$::= a \mid A \to B$
Normal Terms	M, N	$::= \lambda x.\, M \mid R$
Neutral Terms	R	$::= x \mid c \mid R\, N$
Contexts	Ψ, Φ	$::= \bullet \mid \Psi, x{:}A$
Signature	Σ	$::= \bullet \mid \Sigma, a : \mathit{type} \mid \Sigma, c : A$

Object-level terms

$$\frac{x{:}A \in \Psi}{\Psi \vdash x \Leftarrow A} \qquad \frac{c{:}A \in \Sigma}{\Psi \vdash c \Leftarrow A} \qquad \frac{\Psi \vdash R \Rightarrow a' \quad a = a'}{\Psi \vdash R \Leftarrow a}$$

$$\frac{\Psi, x{:}A \vdash M \Leftarrow B}{\Psi \vdash \lambda x.M \Leftarrow A \to B} \qquad \frac{\Psi \vdash M \Rightarrow A \to B \quad \Psi \vdash N \Leftarrow A}{\Psi \vdash M\, N \Rightarrow B}$$

A signature contains essentially only constant definitions similar to a data-type definitions. For example:

```
exp: type.

z      : exp.                          fn : (exp → exp) → exp.
suc    : exp → exp.                    app: exp → exp → exp.
pred   : exp → exp.                    let: exp → (exp → exp) → exp.
iszero : exp → exp.
if     : exp → exp → exp.
true   : exp.
false  : exp.
```

The simply-typed lambda-calculus supports higher-order encodings, where object-level variables are represented by meta-level variables. The type of a data-constructor (like fn, let, ...) which describes a binding construct is modelled as a higher-order function. One key advantages is that we do not need to worry about α-renaming of bound variables. Let us look at some examples to understand better how our on paper language is represented in the meta-language of the simply-typed lambda-calculus.

On paper	Formal representation in LF
suc z	`suc z`
if (iszero (suc z)) then z else (suc z)	`(if (iszero (suc z)) z (suc z))`
fn $x \Rightarrow x$	fn λx. x
fn $x \Rightarrow$ fn $y \Rightarrow x\ y$	fn λx. fn λy. (app x y)
let $f =$ fn $x \Rightarrow 0$ in f 1 end	let (fn λx. z) (λf. app f (suc z))

We can exactly represent those objects which are meaningful. Here are some examples which are not well-typed and which are not meaningful.

Formal representation in LF	Not meaningful because...
`if (iszero z)z`	Partial term: if (iszero z) then z
`let z`	Partial term: let $x = 0$in
fn λx. app x	Partial term: fn $x \Rightarrow (x\ ?)$

More importantly, we observe that we can only describe objects which are in *canonical form*. Moreover, recall that strong normalization holds for the simply typed lambda-calculus. This means we cannot write non-terminating objects! In fact, we cannot write any meaningful computation at all – the only computation one allows is applying a term to another term.

$$(\lambda x.M)\ N \to [N/x]M$$

Although this application rule does not allow for interesting computation, it has in fact a profound impact: It means that by choosing the lambda-calculus as a meta-language we inherit from it not only α-renaming, but also the substitution operation!
This makes the lambda-calculus an ideal *specification language*.

Summary of the key idea :Binding constructs in the object-language are represented via λ-abstraction in the meta-language. The type of a data-constructor (like `fn`, `let`, ...) which describes a binding construct is modelled as a higher-order function. This has several advantages: 1) We do not need to worry about α-renaming of bound variables. 2) We get substitution operation for free since we rely on β-reduction from the meta-language.

Consequently, it will be easier to ensure correctness since we do not need to prove certain properties about renaming and substitutions.

This technique is called *higher-order abstract syntax* or sometimes also *λ-tree syntax*.

3 Dependently typed lambda-calculus

In this section, we will extend the simply-typed lambda calculus to allow for dependent types. Dependent types allow us to index types with other objects. For example, we can index lists with their length to keep track of their size, we can index MiniML expressions with their appropriate MiniML types to only characterize well-typed expressions, et.c

The simply-typed lambda-calculus is not expressive enough to enforce these invariants. In the case of MiniML expressions, it captures all well-formed expressions, but clearly even `suc true` is an expression. What if we want to characterize only those terms which are indeed well-typed? – The only possible terms "syntactically" allowed should be those which are well-typed terms.

To achieve this, we allow LF-types to be indexed by other objects. In the case where we want to represent only well-typed expressions, we will index the type `exp` with MiniML-types.

3.1 Example: Typed MiniML Expressions

So let us begin by defining MiniML types.

```
tp    : type.
nat   : tp.
bool  : tp.
arrow : tp → tp → tp.
```

Next, we will refine the given type for expression by indexing it with MiniML types. The following declares a type constant `exp` which itself takes in an argument, namely a MiniML type represented by `tp`.

```
exp: tp → type.
```

The constant declarations for function abstraction, application, etc. then can be declared as follows:

```
z  : exp nat.
suc: exp nat → exp nat.
if : ΠT:tp. exp bool → exp T → exp T → exp T.

fn : ΠT1:tp.Π T2:tp. (exp T1 → exp T2) → exp (arrow T1 T2).
app: ΠT1:tp.Π T2:tp. exp (arrow T1 T2) → exp T1 → exp T2.
let: ΠT1:tp.Π T2:tp. exp T1 → (exp T1 → exp T2) → exp T2.
```

Well-typed objects of our MiniML language can then be described formally as follows:

On paper	Formal representation in LF
$\mathsf{fn}\ x : \mathsf{nat} \Rightarrow x$	`fn nat nat λx. x`
$\mathsf{fn}\ x : (\mathsf{nat} \rightarrow \mathsf{nat}) \Rightarrow \mathsf{fn}\ y : \mathsf{nat} \Rightarrow x\ y$	`fn (arrow nat nat) nat` `λx. fn nat nat λy. (app nat nat x y)`

$\mathsf{let}\ f = \mathsf{fn}\ x : \mathsf{bool} \Rightarrow 0\ \mathsf{in}\ f\ 1\ \mathsf{end}$	`let bool nat`
	 `(fn bool nat λx. z)`
	 `(λf. app bool nat f (suc z))`

3.2 Theory: Dependently typed lambda-calculus

To achieve a dependently-typed lambda-calculus we only require a small change in the typing rules above. Instead of the simple function type $A \rightarrow B$ we introduce the dependent function type $\Pi x{:}A.B$. We can interpret the simple function type as a special case of the dependent one where x does not occur in B.

$$
\begin{array}{lll}
\text{Types} & A, B, C & ::= a\ M_1 \ldots M_n \mid \Pi x{:}A.B \\
\text{Normal Terms} & M, N & ::= \lambda x.\, M \mid R \\
\text{Neutral Terms} & R & ::= x \mid c \mid R\, N \\
\text{Contexts} & \Psi, \Phi & ::= \bullet \mid \Psi, x{:}A \\
\\
\text{Signature} & \Sigma & ::= \bullet \mid \Sigma, a : \mathit{type} \mid \Sigma, c : A
\end{array}
$$

The other important changes are in the typing rules. Since types can now be indexed by terms, the atomic type is not a simple constant anymore, but is a function symbol. In the rule for lambda-abstraction we simply change the function type to be dependent.

Object-level terms

$$
\frac{x{:}A \in \Psi}{\Psi \vdash x \Leftarrow A}
\qquad
\frac{c{:}A \in \Sigma}{\Psi \vdash c \Leftarrow A}
\qquad
\frac{\Psi \vdash R \Rightarrow a'\ M_1 \ldots M_n \quad a'\ M_1 \ldots M_n = a\ N_1 \ldots N_k}{\Psi \vdash R \Leftarrow a\ N_1 \ldots N_k}
$$

$$
\frac{\Psi, x{:}A \vdash M \Leftarrow B}{\Psi \vdash \lambda x.M \Leftarrow \Pi x{:}A.B}
\qquad
\frac{\Psi \vdash M \Rightarrow \Pi x{:}A.B \quad \Psi \vdash N \Leftarrow A}{\Psi \vdash M\ N \Rightarrow [N/x]B}
$$

The most important change happens in the application rule. Because types are now dependent, the type we infer for M is $\Pi x{:}A.B$. We now cannot simply return B in the conclusion of this rule because x would be free in it! Instead we must substitute the term N into B.

However, this substitution may violate our intention to only describe canonical forms, and in fact if we define substitution naively then indeed this will happen.

For example: $[\lambda y.y/y]\lambda x.y\ x$ will give us $\lambda x.(\lambda y.y)\ x$ which is not even well-typed according to our typing rules!

The solution is to consider a special substitution, called *hereditary substitution* which will preserve canonical forms under substitution.

We will omit here the exact definition, but it is important to note that substitution here is special!

A final remark: Our type theory not only forces our objects to be in β-normal form, but also in η-long form. This means when we have *an object of function type must be an abstraction*!

Some more remarks concerning kinds and types In a setting where we have dependent types (types are more complex!) we must ensure that types are "well-formed". Just as types classify well-typed (meaningful) expressions, we need to be able to classify the meaningful types.

We already have a very simple way of classifying them. In the signature we have $a : \mathsf{type}$, which simply declares a new type constant. But now things are more complex, and it makes sense to include rules which allow us to check for well-formed types.

$$\text{Kind } K ::= \mathsf{type} \mid A \to K$$

We use the judgement $\vdash A \Leftarrow K$ to verify that a type A checks against a given kind K.

$$\frac{\Sigma(a) = A_1 \to \ldots A_n \to \mathsf{type} \quad \text{for all } M_i \text{ we have} \vdash M_i \Leftarrow A_i}{\vdash a\ M_1 \ldots M_n \Leftarrow \mathsf{type}}$$

3.3 Example: Lists indexed by their length

Let's get a bit more familiar with this idea of dependent types. First, how we would usually define lists.

```
lst: type.
nil: lst.
cons: el → lst → lst.
```

What about indexing lists with their length? – That's easy, we simply say the following:

```
lst: nat → type.
nil: lst z.
cons: ΠN:nat el → lst N → lst (suc N).
```

How then do objects belonging to this type look like? – Here are some examples where `a` is an element of the type `el`.

```
(cons (suc z) a (cons z a nil))
```

Dependent types can track much more expressive invariants.

Practice In practice, dependently typed arguments can be omitted in the actual term, and the can be reconstructed. Similarly, we do not write Π-quantifier explicitly. This is indeed very important to achieve a usable and readable language.

So in a system like Twelf we actually write:

```
lst: nat -> type.
nil: lst z.
cons: el -> lst N -> lst (suc N).
```

And we can then verify the following:

```
list1 = (cons a (cons a nil)) : lst (suc (suc z)).
```

The reconstruction problem is in general undecidable, because it requires higher-order unification. However, in practice we can reconstruct implicit arguments. Nevertheless, implicit type arguments must be kept around during runtime which causes a substantial runtime penalty.

Over the last decade, various forms of dependent types have found their way into mainstream functional programming languages to allow programmers to express stronger properties about their programs. Generalized algebraic data types (GADTs) can index types by other types and have entered mainstream languages such as Haskell. Other approaches, such as DML, use indexed types with a fixed constraint domain, such as integers with linear inequalities, for which efficient decision procedures exist.

In these cases all implicit type arguments can be uniquely reconstructed and indeed omitted during runtime.

3.4 Encoding judgements as types and derivations as objects

Dependent types are especially cool because they allow us to encode derivations as objects! Take for example our big-step operational semantics.

Example:Evaluation judgements as dependent types The idea is first to represent the evaluation judgement as a type which is indexed by expressions.

On Paper	Formalization
$e \Downarrow v$	`eval: exp → exp → type`
$\dfrac{}{\mathsf{z} \Downarrow \mathsf{z}}\ \textsf{ev\_z}$	`ev_z: eval z z`
$\dfrac{e \Downarrow v}{\mathtt{suc}\ e \Downarrow \mathtt{suc}\ v}\ \textsf{ev\_s}$	`ev_s: eval E V → eval (suc E) (suc V)`
$\dfrac{e \Downarrow \mathsf{z}}{\mathtt{pred}\ e \Downarrow \mathsf{z}}\ \textsf{ev\_p\_z}$	`ev_p_z: eval E z → eval (pred E) z.`
$\dfrac{e \Downarrow \mathtt{suc}\ v}{\mathtt{pred}\ e \Downarrow v}\ \textsf{ev\_p\_s}$	`ev_p_s: eval E (suc V) → eval (pred E) V.`
$\dfrac{e \Downarrow \mathsf{z}}{\mathtt{iszero}\ \ e \Downarrow \mathtt{true}}\ \textsf{ev\_isz}$	`ev_isz: eval E z → eval (iszero E) true.`
$\dfrac{e \Downarrow \mathtt{suc}\ v}{\mathtt{iszero}\ \ e \Downarrow \mathtt{false}}\ \textsf{ev\_iss}$	`ev_isz: eval E (suc V) → eval (iszero E) false.`

Next, we concentrate on the extension to include functions.

```
ev_fn : eval (fn λx. E x) (fn λx. E x).

ev_app : eval E1 (fn λx.E1' x) →
         eval E2 V2 →
         eval ((λx. E1' x) V2) V
      → eval (app E1 E2) V.

ev_let : eval E1 V1 →
         eval ((λx. E2 x) V1) V
      → eval (let E1 (λx. E2 x)) V.
```

The great part about having chosen a higher-order representation is that we get the substitution operation for free, since β-reduction is part of our meta-language.

Finally, let us consider some example derivation, where we omit implicit index arguments:

```
D1 = ev_z
   : eval z z.

D2 = (ev_let (ev_fn) (ev_app  ev_fn (ev_s ev_z) ev_z)) :
     eval (let  (fn  λx. z) (λf. app f (suc z))) z.
```

The important lessons here is that derivations can be encoded as data-objects using dependent types!

3.5 Example:Typing judgements as dependent types

Finally, let us see how we encode typing derivations. We proceed essentially as before. An essential question we must answer is what to do with the context of assumptions. Recall that the typing judgement says:

$$\Gamma \vdash t : T \text{ Term } t \text{ has type } T \text{ in the context } \Gamma$$

While most of the time this context didn't matter, it is important when we consider language-constructs which deal with variables. Recall the rule for functions:

$$\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \mathsf{fn}\ x{:}T_1 \Rightarrow t : T_1 \to T_2}$$

This rule is *parametric* in the parameter x, and *hypothetical* in the assumption $x{:}T_1$. We can read the premise of this rule as follows:

For any term x, if x has type T_1 then the body t has type T_2.

What does this mean? – It means really that parametric and hypothetical derivations can be viewed as functions! This makes actually perfect sense, since the derivation remains valid for any concrete term t and proof that indeed that t has type T_1.

In fact this is the substitution property we proved earlier.

Lemma 1 (Substitution lemma). *If* $x{:}T_1 \vdash t : T_2$ *and* $\vdash t_1 : T_1$ *then* $\vdash [t_1/x]t : T_2$.

The lesson to learn here is that we do not need an explicit encoding for contexts as a list, but we can view hypothetical and parametric rules as higher-order functions. This leads to the following encoding for the typing rules:

On Paper	Formalization
$\Gamma \vdash e : T$	`hastype : exp → tp → type`
$\dfrac{}{\Gamma \vdash \mathtt{z} : \text{NAT}}$	`tp_z: hastype z nat`
$\dfrac{\Gamma \vdash e : \text{NAT}}{\Gamma \vdash \mathtt{suc}\ (e) : \text{NAT}}$	`tp_s: ΠT:tp. ΠE:exp.` `hastype E nat → hastype (suc E) nat.`
$\dfrac{\Gamma \vdash e : \text{NAT}}{\Gamma \vdash \mathtt{iszero}\ \ (e) : \text{BOOL}}$	`tp_iz: ΠE: exp.` `hastype E nat → hastype (iszero E) bool.`

$\dfrac{\Gamma, x{:}T_1 \vdash e : T_2}{\Gamma \vdash \mathsf{fn}\ x{:}T_1 \Rightarrow e : T_2 : T_1 \rightarrow T_2}$	`tp_fn: ΠE:exp → exp. ΠT1:tp ΠT2:tp.`
	`(Πx:exp. hastype x T1→ hastype (E x)) T2`
	`→ hastype (fnλx. E x)(arrow T1T2).`

$\dfrac{\Gamma \vdash e : T \quad \Gamma, x{:}T \vdash e' : T'}{\Gamma \vdash \mathsf{let}\, x = e\, \mathsf{in}\, e' : T'}$	`tp_l: ΠE:exp. ΠE':exp → exp. ΠT:tp ΠT':tp.`
	`hastype E T →`
	`(Πx:exp. hastype x T → hastype (E' x) T')`
	`→ hastype (let E (λx. E' x)) T'.`

Examples Let's look at some sample derivations, so we get used to the idea of encoding derivations as functions: – **I will omit the implicit type arguments here when we write these examples! But they are there!**.

```
tp_fn (λx. λp. p) : hastype (fn (λy. y)) (arrow T1 T1)
tp_let tp_z (λx. λp. tp_s p) : (hastype (let z (λx. suc x)) nat).
```

Summary Let us summarize the important points from this section.

- *Dependent types allow us to index types by other objects.* We have seen several examples of dependent types: 1) We indexed expressions with their types to keep track of only well-typed expressions. 2) We indexed lists with their length to characterize stronger invariant about lists. 3) We defined the evaluation judgement as a dependent types `eval` which is indexed by two expressions. The constructors which define the elements of this type correspond to the evaluation rules, and evaluation derivations are described as data-objects.
- *Higher-order dependent functions for modelling hypothetical and parametric derivations.* Here we extend this idea that binding structures in our on-paper formalization are modelled via functions/λ-abstraction. Having taken the step of thinking of derivations as

data-objects which can be inspected, and manipulated, we think of a hypothetical and parametric derivation as a function!
It is the next natural step combining the power of dependent types and the power of higher-order abstract syntax. This has similar advantages as when we used higher-order abstract syntax for modelling the data-language. 1) We do not need to worry about α-renaming of parameters which occur in derivations or α-renaming of the label of assumptions. 2) More importantly, we get the substitution lemma for free. Since we do think of a hypothetical and parametric derivation as a function, we simply apply the function to the appropriate arguments.

4 Implementing proofs

We will now proceed to the last step: implementing proofs about operational and static semantics.

We will build on the ideas we have seen so far. Recall that dependent types characterize evaluation judgements, and hence we were able to describe derivations as objects of type `eval`. Here are the two sample derivations from the previous section.

```
D1 = ev_z
   : eval z z.

D2 = (ev_let (ev_app ev_z (ev_s ev_z) ev_fn) (ev_fn)) :
     eval (let  (fn  λx. z) (λf. app f (suc z))) z.
```

Inductive proofs about evaluation judgements can be implemented by pattern matching on the data-objects of the dependent type `eval`. The key idea to characterize proofs here is that we can think of a theorem as a relation between derivation. What we therefore want is a type constant which is indexed by objects describing derivations!

At the moment we are not quite ready to express this, since kinds which characterize types cannot be dependent. But we can of course make them dependent, in the same way we made types dependent.

We therefore declare kinds which classify types as follows:

$$\text{Kind } K ::= \mathsf{type} \mid \Pi x{:}A.K$$

This small change in our theory which is in fact fairly straightforward, has an enormous impact on what we can describe.

4.1 Example: Value soundness

Consider the value soundness theorem.

Theorem 1 (Value soundness). *If $e \Downarrow v$ then v is a value.*

This theorem can be expressed as a relation between two derivations, the derivation $e \Downarrow v$ and a derivation proving that v is a value.

```
vs: ΠE:exp. ΠV:exp. eval E V → value V → type.
```

This means that `vs` now denotes a type which is indexed by the derivation `eval E V` and the derivation `value V`. It is helpful to also actually define the dependent type `value` as follows.

```
value: exp → type.
v_z: value z.
v_s: value V → value (suc V).
v_f: value (fn λx. E x).
```

Let us return to the proof for value soundness and the dependent type `vs`. What are the constructors which are elements of this indexed type? – In fact the constructors we define characterize each case in the proof for value soundness. How did the proof proceed? It proceeded by structural induction on the evaluation judgement. This meant we considered all possible evaluation rules applied. In the formal representation this means we will inspect and analyze the evaluation derivations by pattern matching.

Case 1 $\mathcal{D} : \mathtt{z} \Downarrow \mathtt{z}$ by using $\mathsf{ev\_z}$

We know that $\mathtt{z}$ by definition of values.

This can be represented in our framework as:

```
vs_z: vs ev_z v_z.
```

Case 2 $\mathcal{D} : \mathtt{suc}\ e \Downarrow \mathtt{suc}\ v$ by using $\mathsf{ev\_s}$

$\mathcal{D}_1 : e \Downarrow v$ by inversion on $\mathsf{ev\_s}$

$\mathcal{E}_1 : v$ is a value by i.h. on $\mathcal{D}_1$

$\mathcal{E} : \mathtt{suc}\ v$ is a value by definition of values

This case is then represented as follows:

```
vs_s: vs D1 E1
   → vs (ev_s D1) (v_s E1).
```

Case 3 $\mathcal{D} : \mathsf{fn}\ x \Rightarrow e \Downarrow \mathsf{fn}\ x \Rightarrow e$ by using $\mathsf{ev\_fn}$

$\mathsf{fn}\ x \Rightarrow e$ is a value by definition of values.

This case is then represented as:

```
vs_f: vs (ev_fn) (v_f).
```

Case 4 $\mathcal{D} : e_1\ e_2 \Downarrow v$ by rule $\mathsf{ev\_app}$

$\mathcal{D}_1 : e_1 \Downarrow (\mathsf{fn}\ x \Rightarrow e)$

$\mathcal{D}_2 : e_2 \Downarrow v_2$

$\mathcal{D}_3 : [v_2/x]e \Downarrow v$ by inversion on $\mathsf{ev\_app}$

v is a value by i.h. on $\mathcal{D}_3$

This case is represented as:

```
vs_app: vs D₃ E
        → vs (ev_app D₁ D₂ D₃) E.
```

4.2 How to ensure this relation constitutes a proof?

So far we have only used the dependently typed lambda-calculus to encode each case in the proof as a constant of a specific dependent type. An important question to consider is then what do these dependent types achieve? – Dependent types guarantee that certain constraints between the evaluation judgement and the value judgement are satisfied. For example, type checking will detect that the following case is wrong:

```
ev_s : vs D E → vs (ev_s D) ev_z
```

This case would correspond to proving

Wrong Case 2 $\mathcal{D} : \texttt{suc}\ e \Downarrow \texttt{suc}\ v$	by using `ev_s`
$\mathcal{D}_1 : e \Downarrow v$	by inversion on `ev_s`
$\mathcal{E}_1 : v$ is a value	by i.h. on $\mathcal{D}_1$
$\mathcal{E} : \texttt{z}$ is a value	by definition of values

Each line in this proof is of course correct, but it does not prove what we need to prove! The theorem says, we need to prove that `suc` v is a value, not that `z` is a value.

Types do therefore fulfil an important job. However, they do not quite ensure that we have an *inductive proof.* For an inductive proof to be correct we must in addition ensure that

1. Given the assumptions we are always able to construct the conclusion of the theorem. In other words, the relation is indeed a function. We call this property being well-moded.
2. We have covered all possible cases
3. All appeals to the induction hypothesis are valid.

These last three properties are not guaranteed by the dependently typed lambda-calculus, our meta-language. These are properties we must establish separately, and we must go outside of our dependently typed lambda-calculus to establish these properties.

Proving modes, coverage, termination The Twelf system is an implementation of the dependently typed lambda calculus. In addition to type checking, and type reconstruction, it provides external checkers which guarantee that a dependent type describes a total function, i.e. it satisfies exactly the three properties listed above. How it actually verifies these properties is beyond the scope of these notes, but we can give a brief introduction how to check the desired properties.

Modes To ensure that the type corresponding to our theorem is indeed a function, we will check that given the assumption `eval E V` we can always construct objects of type `value V`.

```
%mode vs +D -P.
```

The arguments annotated with `+` are describing the inputs or assumptions while the arguments annotated with `-` describe the output or conclusion of the theorem.

Termination In inductive proofs we must ensure that all appeals to the induction hypothesis were valid. We can think of an inductive proof as a recursive function. Consequently, the appeal to the induction hypothesis corresponds to the recursive call. To prove that the appeal to the induction hypothesis is valid, means to prove that the recursive call was made on smaller input arguments. More generally it means we must prove that the function terminates.

In Twelf, we can call a termination checker to verify that the implementation of `vs` indeed is a recursive function which terminates as follows using the keyword `%terminates`.

```
%terminates D (vs D _).
```

The keyword `%terminates` takes in two arguments: the first one describes the argument which is supposed to decrease in the recursive call. In other words, this is the index object you are doing induction on. The second argument gives the object denoting the theorem.

Coverage Finally, we must ensure that we have covered all cases. If our implementation of `vs` constitutes a proof then we must have specified cases for each evaluation rule. The coverage checker will verify that this is indeed the case.

```
%covers vs +D -P.
```

The keyword `%covers` is similar to the keyword `%mode`. It takes in the name of the type in question, the input arguments and the output arguments properly annotated to describe their roles.

4.3 Example: Type preservation

Bibliography