An Introduction to Logical Relations

Proving Program Properties Using Logical Relations

Lau Skorstengaard
lask@cs.au.dk

Contents

1 Introduction **2**
1.1 Simply Typed Lambda Calculus (STLC) 2
1.2 Logical Relations 3
1.3 Categories of Logical Relations 5

2 Normalization of the Simply Typed Lambda Calculus **5**
2.1 Strong Normalization of STLC 5
2.2 Exercises 10

3 Type Safety for STLC **11**
3.1 Type safety - the classical treatment 11
3.2 Type safety - using logical predicate 12
3.3 Exercises 15

4 Universal Types and Relational Substitutions **15**
4.1 System F (STLC with universal types) 16
4.2 Contextual Equivalence 19
4.3 A Logical Relation for System F 20
4.4 Exercises 28

5 Existential types **29**

6 Recursive Types and Step Indexing **34**
6.1 A motivating introduction to recursive types 34
6.2 Simply typed lambda calculus extended with μ 36
6.3 Step-indexing, logical relations for recursive types 37
6.4 Exercises 41

1 Introduction

The term logical relations stems from Gordon Plotkin's memorandum *Lambda-definability and logical relations written* in 1973. However, the spirit of the proof method can be traced back to Wiliam W. Tait who used it to show strong normalization of *System T* in 1967.

Names are a curious thing. When I say "chair", you immediately get a picture of a chair in your head. If I say "table", then you picture a table. The reason you do this is because we denote a chair by "chair" and a table by "table", but we might as well have said "giraffe" for chair and "Buddha" for table. If we encounter a new word composed of known words, it is natural to try to find its meaning by composing the meaning of the components of the name. Say we encounter the word "tablecloth" for the first time, then if we know what "table" and "cloth" denotes we can guess that it is a piece of cloth for a table. However, this approach does not always work. For instance, a "skyscraper" is not a scraper you use to scrape the sky. Likewise for logical relations, it may be a fool's quest to try to find meaning in the name. Logical relations are relations, so that part of the name makes sense. They are also defined in a way that has a small resemblance to a logic, but trying to give meaning to logical relations only from the parts of the name will not help you understand them. A more telling name might be Type Indexed Inductive Relations. However, Logical Relations is a well-established name and easier to say, so we will stick with it (no one would accept "giraffe" to be a chair).

The remainder of this note is based on the lectures of Amal Ahmed at the Oregon Programming Languages Summer School, 2015. The videos of the lectures can be found at `https://www.cs.uoregon.edu/research/summerschool/summer15/curriculum.html`.

1.1 Simply Typed Lambda Calculus (STLC)

The language we use to present logical predicates and relations is the simply typed lambda calculus. In the first section, it will be used in its basic form. In the later, sections the simply typed lambda calculus will be used as a base language. If the text says that we extend with some construct, then it is the simply typed lambda calculus that we extend with this construct. The simply typed lambda calculus is defined as follows:

Types:	$\tau ::= \mathit{bool} \mid \tau \to \tau$
Terms:	$e ::= x \mid \text{true} \mid \text{false} \mid \text{if } e \text{ then } e \text{ else } e \mid \lambda x : \tau.\ e \mid e\ e$
Values:	$v ::= \text{true} \mid \text{false} \mid \lambda x : \tau.\ e$
Evaluation contexts:	$E ::= [] \mid \text{if } E \text{ then } e \text{ else } e \mid E\ e \mid v\ E$
Evaluations:	$\text{if true then } e_1 \text{ else } e_2 \mapsto e_1$ $\text{if false then } e_1 \text{ else } e_2 \mapsto e_2$ $(\lambda x : \tau.\ e)\ v \mapsto e[^v/_x]$ $\dfrac{e \mapsto e'}{E[e] \mapsto E[e']}$
Typing Contexts:	$\Gamma ::= \bullet \mid \Gamma, x : \tau$
Typing rules:	$\dfrac{}{\Gamma \vdash \text{false} : \mathit{bool}}$ T-False $\qquad \dfrac{}{\Gamma \vdash \text{true} : \mathit{bool}}$ T-True $\dfrac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$ T-Var $\qquad \dfrac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1.\ e : \tau_1 \to \tau_2}$ T-Abs $\dfrac{\Gamma \vdash e_1 : \tau_2 \to \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau}$ T-App $\dfrac{\Gamma \vdash e : \mathit{bool} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : \tau}$ T-If

For the typing contexts, it is assumed that the binders are distinct. So if $x \in \text{dom}(\Gamma)$, then $\Gamma, x : \tau$ is not a legal context.

1.2 Logical Relations

Logical relations are used to prove properties about programs in a language. Logical relations are proof methods and can be used as an alternative to proving properties directly. Examples of properties one can show using logical relations are:

- Termination (Strong normalization)
- Type safety
- Equivalence of programs

- Correctness of programs
- Representation independence
- Parametricity and free theorems, e.g.,

  $$f : \forall\alpha.\ \alpha \to \alpha$$

  The program cannot inspect α as it has no idea which type it will be, therefore f must be identity function.

  $$\forall\alpha.\ int \to \alpha$$

  A function with this type cannot exist (the function would need to return something of type α, but it only has something of type *int* to work with so it cannot possibly return a value of the proper type).
- Security-Typed Languages (for Information Flow Control (IFC))
  Example: All types in the code snippet below are labeled with their security level. A type can be labeled with either L for *low* or H for *high*. We do not want any flow from variables with a *high* labeled type to a variable with a *low* labeled type. The following is an example of an insecure *explicit flow* of information:

  ```
  x : int^L
  y : int^H
  x = y     //This assignment is insecure.
  ```

  Further, information may leak through a *side channel*. That is the value denoted by a variable with a *low* labeled type depends on the value of a variable with a *high* labeled type. If this is the case, we may not have learned the secret value, but we may have learned some information about it. An example of a side channel:

  ```
  x : int^L
  y : int^H
  if y > 0 then x = 0 else x = 1
  ```

  The above examples show undesired programs or parts of programs, but if we want to generally state behavior we do not want a program to exhibit, then we state it as non-interference:

  $$\vdash P : int^L \times int^H \to int^L$$
  $$P(v_L, v_{1H}) \approx_L P(v_L, v_{2H})$$

  If we run P with the same *low* value and with two different *high* values, then the *low* result of the two runs of the program should be equal. That is the *low* result does not depend on *high* values.

1.3 Categories of Logical Relations

We can split logical relations into two: logical predicates and logical relations. Logical predicates are unary and are usually used to show properties of a program. Logical relations are binary and are usually used to show equivalences:

Logical Predicates	Logical Relations
(Unary)	(Binary)
$P_\tau(e)$	$R_\tau(e_1, e_2)$
- One property	- Program Equivalence
- Strong normalization	
- Type safety	

The following describes some properties we want a logical *predicate* to have in general. These properties can be generalized to logical relations. In general, for a logical predicate P_τ and an expression e, we want e to be accepted by the predicate if it satisfies the following properties[1]:

1. $\bullet \vdash e : \tau$
2. The property we wish e to have.
3. The condition is preserved by eliminating forms.

2 Normalization of the Simply Typed Lambda Calculus

2.1 Strong Normalization of STLC

In this section, we wish to show that the simply typed lambda calculus has strong normalization which means that every term is strongly normalizing. Normalization of a term is the process of reducing a term into its normal form. If a term is strongly normalizing, then it reduces to its normal form. In our case, we define the normal forms of the language to be the values of the language.

[1]Note: when we later want to prove type safety, the well-typedness property is weakened to only require e to be closed.

A first try on normalization of STLC

We start with a couple of abbreviations:

$$e \Downarrow v \stackrel{\text{def}}{=} e \mapsto^* v$$
$$e \Downarrow \stackrel{\text{def}}{=} \exists v.\, e \Downarrow v$$

Where v is a value. What we want to prove is:

Theorem (Strong Normalization)**.**
If $\bullet \vdash e : \tau$ *then* $e \Downarrow$

We first try to prove the above property directly to see it fail.

Proof. ¡ This proof gets stuck and is not complete. !
Induction on the structure of the typing derivation.
Case $\bullet \vdash \text{true} : \mathit{bool}$, this term has already terminated.
Case $\bullet \vdash \text{false} : \mathit{bool}$, same as for true.
Case $\bullet \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : \tau$, simple, but requires the use of canonical forms of bool[2].
Case $\bullet \vdash \lambda x : \tau_1.\, e : \tau_1 \to \tau_2$, it is a value already and it has terminated.
Case $\dfrac{\Gamma \vdash e_1 : \tau_2 \to \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau}\ \text{T-App}$,
by the induction hypothesis, we get $e_1 \Downarrow v_1$ and $e_2 \Downarrow v_2$. By the type of e_1, we conclude $e_1 \Downarrow \lambda x : \tau_2.\, e'$. What we need to show is $e_1\ e_2 \Downarrow$. We know $e_1\ e_2$ takes the following steps:

$$\begin{aligned} e_1\ e_2 &\mapsto^* (\lambda x : \tau_2.\, e')\ e_2 \\ &\mapsto^* (\lambda x : \tau_2.\, e')\ v_2 \\ &\mapsto e'[v_2/x] \end{aligned}$$

Here we run into an issue as we do not know anything about e'. Our induction hypothesis is not strong enough.[3] ⊠

A logical predicate for strongly normalizing expressions

We want to define a logical predicate, $\text{SN}_\tau(e)$. We want SN_τ to accept the expressions of type τ that are strongly normalizing. In the introduction, we considered

[2] See Pierce's Types and Programming Languages for more about canonical forms.
[3] :(

some properties a logical predicate in general should have. Keep these properties in mind when we define the logical predicate for strong normalization:

$$\begin{aligned}\mathrm{SN}_{bool}(e) &\Leftrightarrow \bullet \vdash e : bool \ \wedge \ e \Downarrow \\ \mathrm{SN}_{\tau_1 \to \tau_2}(e) &\Leftrightarrow \bullet \vdash e : \tau_1 \to \tau_2 \ \wedge \ e \Downarrow \ \wedge \ (\forall e'.\ \mathrm{SN}_{\tau_1}(e') \implies \mathrm{SN}_{\tau_2}(e\ e'))\end{aligned}$$

It is here important to consider whether the logical predicate is well-founded. $\mathrm{SN}_\tau(e)$ is defined over the structure of τ, so it is indeed well-founded.

Strongly normalizing using a logical predicate

We are now ready to show strong normalization using $\mathrm{SN}_\tau(e)$. The proof is done in two steps:

ⓐ $\bullet \vdash e : \tau \implies \mathrm{SN}_\tau(e)$

ⓑ $\mathrm{SN}_\tau(e) \implies e \Downarrow$

The structure of this proof is common to proofs that use logical relations. We first prove that well-typed terms are in the relation. Then we prove that terms in the relation actually have the property we want to show (in this case strong normalization).

The proof of ⓑ is by induction on τ. This should not be difficult, as we baked the property we want into the relation. That was the second property we in general wanted a logical relation to satisfy.

We could try to prove ⓐ by induction over $\bullet \vdash e : \tau$, but the case

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1.\ e : \tau_1 \to \tau_2}\ \text{T-Abs}$$

gives issues. Instead we prove a generalization of ⓐ

Theorem (ⓐ Generalized)**.** *If* $\Gamma \vdash e : \tau$ *and* $\gamma \models \Gamma$ *then* $\mathrm{SN}_\tau(\gamma(e))$

Here γ is a substitution, $\gamma = \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$. We define the substitution to work as follows:

$$\begin{aligned}&\emptyset(e) = e \\ &\gamma[x \mapsto v](e) = \gamma(e[^x/_v])\end{aligned}$$

In English, the theorem reads: If e is well-typed with respect to some type τ and we have some closing substitution that satisfy the typing environment, then if we close of e with γ, then this closed expression is in SN_τ.

$\gamma \models \Gamma$ is read "the substitution γ satisfies the type environment, Γ." It is defined as follows:

$$\gamma \models \Gamma \stackrel{\text{def}}{=} \mathrm{dom}(\gamma) = \mathrm{dom}(\Gamma) \ \wedge \ \forall x \in \mathrm{dom}(\Gamma).\ \mathrm{SN}_{\Gamma(x)}(\gamma(x))$$

To prove the generalized theorem we need further two lemmas

Lemma (Substitution Lemma)**.** *If* $\Gamma \vdash e : \tau$ *and* $\gamma \models \Gamma$*, then* $\bullet \vdash \gamma(e) : \tau$

Lemma (SN preserved by forward/backward reduction)**.** *Suppose* $\bullet \vdash e : \tau$ *and* $e \mapsto e'$

1. *if* $\mathrm{SN}_\tau(e')$*, then* $\mathrm{SN}_\tau(e)$
2. *if* $\mathrm{SN}_\tau(e)$*, then* $\mathrm{SN}_\tau(e')$

Proof. Probably also left as an exercise (not proved during the lecture). □

Proof. (Substitution Lemma). Left as an exercise. □

Proof. (@ Generalized). Proof by induction on $\Gamma \vdash e : \tau$.
Case $\Gamma \vdash \mathrm{true} : \mathit{bool}$,
We have:

$$\gamma \models \Gamma$$

We need to show:

$$\mathrm{SN}_{\mathit{bool}}(\gamma(\mathrm{true}))$$

If we do the substitution, we just need to show $\mathrm{SN}_{\mathit{bool}}(\mathrm{true})$ which is true as $\mathrm{true} \Downarrow \mathrm{true}$.
Case $\Gamma \vdash \mathrm{false} : \mathit{bool}$, similar to the true case.
Case $\dfrac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ \textsc{T-Var}$,
We have:

$$\gamma \models \Gamma$$

We need to show:

$$\mathrm{SN}_\tau(\gamma(x))$$

This case follows from the definition of $\Gamma \models \gamma$. We know that x is well-typed, so it is in the domain of Γ. From the definition of $\Gamma \models \gamma$, we then get $\mathrm{SN}_{\Gamma(x)}(\gamma(x))$. From well-typedness of x, we have $\Gamma(x) = \tau$ which then gives us what we needed to show.

Case $\Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : \tau$, left as an exercise.

Case $\dfrac{\Gamma \vdash e_1 : \tau_2 \to \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau}\ \textsc{T-App}$,

We have:

$$\gamma \models \Gamma$$

We need to show:

$$\mathrm{SN}_\tau(\gamma(e_1\ e_2)) \equiv \mathrm{SN}_\tau(\gamma(e_1)\ \gamma(e_2))$$

By the induction hypothesis we have

$$\mathrm{SN}_{\tau_2 \to \tau}(\gamma(e_1)) \tag{1}$$
$$\mathrm{SN}_{\tau_2}(\gamma(e_2)) \tag{2}$$

By the 3rd property of (1), $\forall e'.\ \mathrm{SN}_{\tau_2}(e') \implies \mathrm{SN}_\tau(\gamma(e_1)\ e')$, instantiated with (2), we get $\mathrm{SN}_\tau(\gamma(e_1)\ \gamma(e_2))$ which is the result we need.

Case $\dfrac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1.\ e : \tau_1 \to \tau_2}\ \textsc{T-Abs}$,

We have:

$$\gamma \models \Gamma$$

We need to show:

$$\mathrm{SN}_{\tau_1 \to \tau_2}(\gamma(\lambda x : \tau_1.\ e)) \equiv \mathrm{SN}_{\tau_1 \to \tau_2}(\lambda x : \tau_1.\ \gamma(e))$$

Our induction hypothesis in this case reads:

$$\Gamma, x : \tau_1 \vdash e : \tau_2\ \wedge\ \gamma' \models \Gamma, x : \tau_1 \quad \implies \quad \mathrm{SN}_{\tau_2}(\gamma'(e))$$

It suffices to show the following three things:

1. $\bullet \vdash \lambda x : \tau_1.\ \gamma(e) : \tau_1 \to \tau_2$
2. $\lambda x : \tau_1.\ \gamma(e) \Downarrow$
3. $\forall e'.\, \mathrm{SN}_{\tau_1}(e') \implies \mathrm{SN}_{\tau_2}((\lambda x : \tau_1.\ \gamma(e))\ e')$

If we use the substitution lemma[4] and push the γ in under the λ-abstraction, then we get 1. 2 is okay as the lambda-abstraction is a value.

It only remains to show 3. To do this, we want to somehow apply the induction hypothesis for which we need a γ' such that $\gamma' \models \Gamma, x : \tau_1$. We already have γ and $\gamma \models \Gamma$, so our γ' should probably have have the form $\gamma' = \gamma[x \mapsto v_?]$ for some $v_?$ of type τ_1. Let us move on and see if any good candidates for $v_?$ present themselves.

Let e' be given and assume $\mathrm{SN}_{\tau_1}(e')$. We then need to show $\mathrm{SN}_{\tau_2}((\lambda x : \tau_1.\ \gamma(e))\ e')$. From $\mathrm{SN}_{\tau_1}(e')$, it follows that $e' \Downarrow v'$ for some v'. v' is a good candidate for $v_?$ so let $v_? = v'$. From the forward part of the preservation lemma, we can further conclude $\mathrm{SN}_{\tau_1}(v')$. We use this to conclude $\gamma[x \mapsto v'] \models \Gamma, x : \tau_1$ which we use with the assumption $\Gamma, x : \tau_1 \vdash e : \tau_2$ to instantiate the induction hypothesis and get $\mathrm{SN}_{\tau_2}(\gamma[x \mapsto v'](e))$.

Now consider the following evaluation:

$$\begin{aligned}(\lambda x : \tau_1.\ \gamma(e))\ e' &\mapsto^* (\lambda x : \tau_1.\ \gamma(e))\ v' \\ &\mapsto \gamma(e)[v'/x] \equiv \gamma[x \mapsto v'](e)\end{aligned}$$

We already concluded that $e' \mapsto^* v'$, which corresponds to the first series of steps. We can then do a β-reduction to take the next step and finally we get something that is equivalent to $\gamma[x \mapsto v'](e)$. That is we have the evaluation

$$(\lambda x : \tau_1.\ \gamma(e))\ e' \mapsto^* \gamma[x \mapsto v'](e)$$

From $\mathrm{SN}_{\tau_1}(e')$, we have $\bullet \vdash e' : \tau_1$ and we already argued that $\bullet \vdash \lambda x : \tau_1.\ \gamma(e) : \tau_1 \to \tau_2$ so from the application typing rule we get $\bullet \vdash (\lambda x : \tau_1.\ \gamma(e))\ e' : \tau_2$. We can use this with the above evaluation and the forward part of the preservation lemma to argue that every intermediate expressions in the steps down to $\gamma[x \mapsto v'](e)$ are closed and well typed.

If we use $\mathrm{SN}_{\tau_2}(\gamma[x \mapsto v'](e))$ with $(\lambda x : \tau_1.\ \gamma(e))\ e' \mapsto^* \gamma[x \mapsto v'](e)$ and the fact that every intermediate step in the evaluation is closed and well typed, then we can use the backward reduction part of the SN preservation lemma to get $\mathrm{SN}_{\tau_2}((\lambda x : \tau_1.\ \gamma(e))\ e')$ which is the result we wanted. □

2.2 Exercises

1. Prove SN preserved by forward/backward reduction.

2. Prove the substitution lemma.

[4]Substitution has not been formally defined here, but one can find a sound definition in Pierce's Types and Programming Languages.

3. Go through the cases of "ⓐ Generalized" shown here by yourself.

4. Prove the if-case of "ⓐ Generalized".

5. Extend the language with pairs and adjust the proofs.

   (a) See how the clauses, we generally wanted our logical predicate to have, play out when we extend the logical predicate. Do we need to add anything for the third clause or does it work out without putting anything there, like we did with the *bool* case?

3 Type Safety for STLC

In the following section, we want to prove type safety for the simply typed lambda calculus. We do not want to prove it directly as one normally does. We want to prove it using a logical predicate.

First we need to consider what type safety is. The classical mantra for type safety is "Well-typed programs do not *go wrong*." It depends on the language and type system what *go wrong* means, but in our case a program has *gone wrong* if it is stuck[5] (an expression is stuck if it is irreducible but not a value).

3.1 Type safety - the classical treatment

Type safety for simply typed lambda calculus is stated as follows:

Theorem (Type Safety for STLC)**.** *If* $\bullet \vdash e : \tau$ *and* $e \mapsto^* e'$*, then* $\text{Val}(e')$ *or* $\exists e''.\ e' \mapsto e''$.

Traditionally type safety is proven with two lemmas: progress and preservation.

Lemma (Progress)**.** *If* $\bullet \vdash e : \tau$*, then* $\text{Val}(e)$ *or* $\exists e'.\ e \mapsto e'$.

Progress is normally proved by induction on the typing derivation.

Lemma (Preservation)**.** *If* $\bullet \vdash e : \tau$ *and* $e \mapsto e'$*, then* $\bullet \vdash e' : \tau$.

Preservation is normally proved by induction on the evaluation. Preservation is also known as *subject reduction*. Progress and preservation talk about one step, so to prove type safety we have to do induction on the evaluation. Here we do not want to prove type safety the traditional way. We want to prove it using a logical predicate. We use a logical predicate rather than a logical relation because type safety is a unary property.

[5]If we consider language-based security for information flow control the notion of *go wrong* would be that there is an undesired flow of information

3.2 Type safety - using logical predicate

The notation will here be changed compared to the one from lecture 1. We define the logical predicate in two parts: a value interpretation and an expression interpretation. The value interpretation is a function from types to the power set of closed values:

$$\mathcal{V}[\![-]\!] : \mathit{type} \to \mathcal{P}(\mathit{ClosedVal})$$

The value interpretation is defined as:

$$\begin{aligned} \mathcal{V}[\![\mathit{bool}]\!] &= \{\text{true}, \text{false}\} \\ \mathcal{V}[\![\tau_1 \to \tau_2]\!] &= \{\lambda x : \tau_1.\ e \mid \forall v \in \mathcal{V}[\![\tau_1]\!].\ e[v/x] \in \mathcal{E}[\![\tau_2]\!]\} \end{aligned}$$

We define the expression interpretation as:

$$\mathcal{E}[\![\tau]\!] = \{e \mid \forall e'.\ e \mapsto^* e' \ \wedge\ \text{irred}(e') \implies e' \in \mathcal{V}[\![\tau]\!]\}$$

Notice that neither $\mathcal{V}[\![\tau]\!]$ nor $\mathcal{E}[\![\tau]\!]$ requires well-typedness. Normally this would be a part of the predicate, but as the goal is to prove type safety we do not want it as a part of the predicate. In fact, if we did include a well-typedness requirement, then we would end up having to prove preservation for some of the proofs to go through. We do, however, require the value interpretation to only contain closed values. An expression is irreducible if it is unable to take any reduction steps according to the evaluation rules. The predicate irred captures whether an expression is irreducible:

$$\text{irred}(e) \stackrel{\text{def}}{=} \nexists e'.\ e \mapsto e'$$

The sets are defined on the structure of the types. $\mathcal{V}[\![\tau_1 \to \tau_2]\!]$ contains $\mathcal{E}[\![\tau_2]\!]$, but $\mathcal{E}[\![\tau_2]\!]$ uses τ_2 directly in $\mathcal{V}[\![\tau_2]\!]$, so the definition is structurally well-founded. To prove type safety, we first define a new predicate, safe:

$$\text{safe}(e) \stackrel{\text{def}}{=} \forall e'.\ e \mapsto^* e' \implies \text{Val}(e') \vee \exists e''.\ e' \mapsto e''$$

An expression e is safe if it can take a number of steps and end up either as a value or as an expression that can take another step.

We are now ready to prove type safety. Just like we did for strong normalization, we prove type safety in two steps:

ⓐ $\quad \bullet \vdash e : \tau \implies e \in \mathcal{E}[\![\tau]\!]$

ⓑ $\quad e \in \mathcal{E}[\![\tau]\!] \implies \text{safe}(e)$

Rather than proving ⓐ directly we prove a more general theorem and get ⓐ as a corollary. But we are not yet in a position to state the theorem. First we need to define define the interpretation of environments:

$$\begin{aligned}\mathcal{G}[\![\bullet]\!] &= \{\emptyset\} \\ \mathcal{G}[\![\Gamma, x:\tau]\!] &= \{\gamma[x \mapsto v] \mid \gamma \in \mathcal{G}[\![\Gamma]\!] \ \wedge \ v \in \mathcal{V}[\![\tau]\!]\}\end{aligned}$$

Further we need to define semantic type safety:

$$\Gamma \models e : \tau \stackrel{\text{def}}{=} \forall \gamma \in \mathcal{G}[\![\Gamma]\!].\ \gamma(e) \in \mathcal{E}[\![\tau]\!]$$

We can now define our generalized version of ⓐ.

Theorem (Fundamental Property)**.** *If* $\Gamma \vdash e : \tau$*, then* $\Gamma \models e : \tau$

A theorem like this would typically be the first you prove after defining a logical relation. The theorem says that every syntactic type safety implies semantic type safety.

We also alter the ⓑ part of the proof, so we prove

$$\bullet \models e : \tau \implies \text{safe}(e)$$

Proof. (Altered ⓑ). Suppose $e \mapsto^* e'$ for some e', then we need to show $\text{Val}(e')$ or $\exists e''.\ e' \mapsto e''$. We proceed by casing on whether or not $\text{irred}(e')$:
Case $\neg\,\text{irred}(e')$, this case follows directly from the definition of irred. $\text{irred}(e')$ is defined as $\nexists e''.\ e' \mapsto e''$ and as the assumption is $\neg\,\text{irred}(e')$, we get $\exists e''.\ e' \mapsto e''$.
Case $\text{irred}(e')$, by assumption we have $\bullet \models e : \tau$. As the typing context is empty, we choose the empty substitution and get $e \in \mathcal{E}[\![\tau]\!]$. We now use the definition of $e \in \mathcal{E}[\![\tau]\!]$ with what we supposed, $e \mapsto^* e'$, and the case assumption, $\text{irred}(e')$, to conclude $e' \in \mathcal{V}[\![\tau]\!]$. As e' is in the value interpretation of τ, we can conclude $\text{Val}(e')$. □

To prove the Fundamental Property, we need a substitution lemma:

Lemma (Substitution)**.** *Let* e *be syntactically well-formed term, let* v *be a closed value and let* γ *be a substitution that map term variables to closed values, and let* x *be a variable not in the domain of* γ*, then*

$$\gamma[x \mapsto v](e) = \gamma(e)[^{x}/_{v}]$$

Proof. By induction on the size of γ.
Case $\gamma = \emptyset$, this case is immediate by how substitution is defined. That is by

definition we have $[x \mapsto v]e = e[^v/_x]$.
Case $\gamma = \gamma'[y \mapsto v']$, $x \neq y$, in this case our induction hypothesis is:

$$\gamma'[x \mapsto v]e = \gamma'(e)[^v/_x]$$

We wish to show

$$\gamma'[y \mapsto v'][x \mapsto v]e = \gamma'[y \mapsto v'](e)[^v/_x]$$

$$\begin{aligned}
\gamma'[y \mapsto v'][x \mapsto v]e &= \gamma'[x \mapsto v][y \mapsto v']e && (3)\\
&= \gamma'[x \mapsto v](e[^{v'}/_y]) && (4)\\
&= \gamma'(e[^{v'}/_y])[^x/_v] && (5)\\
&= \gamma'[y \mapsto v'](e)[^x/_v] && (6)
\end{aligned}$$

In the first step (3), we swap the two mappings. It is safe to do so as both v and v' are closed so we know that no variable capturing will occur. In the second step (4), we just use the definition of substitution (as specified in the first lecture note). In the third step (5), we use the induction hypothesis[6]. Finally in the last step (6), we use the definition of substitution to get the y binding out as an extension of γ'. □

Proof. (Fundamental Property). Proof by induction on the typing judgment.
Case $\dfrac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1.\ e : \tau_1 \to \tau_2}$ T-Abs,
We need to show $\Gamma \models \lambda x : \tau_1.\ e : \tau_1 \to \tau_2$. First suppose $\gamma \in \mathcal{G}[\![\Gamma]\!]$. Then we need to to show

$$\gamma(\lambda x : \tau_1.\ e) \in \mathcal{E}[\![\tau_1 \to \tau_2]\!] \equiv (\lambda x : \tau_1.\ \gamma(e)) \in \mathcal{E}[\![\tau_1 \to \tau_2]\!]$$

Now suppose that $\lambda x : \tau_1.\ \gamma(e) \mapsto^* e'$ and irred(e'). We then need to show $e' \in \mathcal{V}[\![\tau_1 \to \tau_2]\!]$. Since $\lambda x : \tau_1.\ \gamma(e)$ is a value, it is irreducible, and we can conclude it took no steps. In other words $e' = \lambda x : \tau_1.\ \gamma(e)$. So we need to show $\lambda x : \tau_1.\ \gamma(e) \in \mathcal{V}[\![\tau_1 \to \tau_2]\!]$. Now suppose $v \in \mathcal{V}[\![\tau_1]\!]$ then we need to show $\gamma(e)[v/x] \in \mathcal{E}[\![\tau_2]\!]$.

Keep the above proof goal in mind and consider the induction hypothesis:

$$\Gamma, x : \tau_1 \models e : \tau_2$$

[6]The induction hypothesis actually has a number of premises, as an exercise convince yourself that they are satisfied.

Instantiate this with $\gamma[x \mapsto v]$. We have $\gamma[x \mapsto v] \in \mathcal{G}[\![\Gamma, x : \tau_1]\!]$ because we started by supposing $\gamma \in \mathcal{G}[\![\Gamma]\!]$ and we also had $v \in \mathcal{V}[\![\tau_2]\!]$. The instantiation gives us $\gamma[x \mapsto v](e) \in \mathcal{E}[\![\tau_2]\!] \equiv \gamma(e)[v/x] \in \mathcal{E}[\![\tau_2]\!]$. The equivalence is justified by the substitution lemma we proved. This is exactly the proof goal we kept in mind.

Case $\dfrac{\Gamma \vdash e_1 : \tau_2 \to \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau}$ T-App, show this case as an exercise.

The remaining cases were not proved during the lecture. □

Now consider what happens if we add pairs to the language (exercise 5 in exercise section 2.2). We need to add a clause to the value interpretation:

$$\mathcal{V}[\![\tau_1 \times \tau_2]\!] = \{\langle v_1, v_2\rangle \mid v_1 \in \mathcal{V}[\![\tau_1]\!] \ \wedge\ v_2 \in \mathcal{V}[\![\tau_2]\!]\}$$

There is nothing surprising in this addition to the value relation, and it should not be a challenge to show the pair case of the proofs.

If we extend our language with sum types.

$$e ::= \ldots \mid \text{inl}\ v \mid \text{inr}\ v \mid \text{case}\ e\ \text{of}\ \text{inl}\ x => e_1 \quad \text{inr}\ x => e_2$$

Then we need to add the following clause to the value interpretation:

$$\mathcal{V}[\![\tau_1 + \tau_2]\!] = \{\text{inl}\ v \mid v \in \mathcal{V}[\![\tau_1]\!]\} \cup \{\text{inr}\ v \mid v \in \mathcal{V}[\![\tau_2]\!]\}$$

It turns out this clause is sufficient. One might think that is is necessary to require the body of the match to be in the expression interpretation, which looks something like $\forall e_1 \in \mathcal{E}[\![\tau]\!]$. This requirement will, however, give well-foundedness problems, as τ is not a structurally smaller type than $\tau_1 + \tau_2$. It may come as a surprise that we do not need to relate the expressions as the slogan for logical relations is "Related inputs to related outputs."

3.3 Exercises

1. Prove the T-App case of the Fundamental Property.

4 Universal Types and Relational Substitutions

In the previous sections, we considered safety and termination, but now we shift our focus to program equivalences. To prove program equivalences, we will use logical relations as our proof method. To motivate the need for arguing about program equivalence, we first introduce universal types.

Say we have a function that sorts integer lists:

$$\text{sortint} \ : \ \mathit{list}\ \mathit{int} \to \mathit{list}\ \mathit{int}$$

sortint takes a list of integers and returns a sorted version of that list. Say we now want a function that sorts lists of strings, then instead of implementing a separate function we could factor out the code responsible for sorting and have just one function. The type signature of such a generic sort function is:

$$\text{sort} \ : \ \forall\alpha.\ (\mathit{list}\ \alpha) \times (\alpha \times \alpha \to \mathit{bool}) \to \mathit{list}\ \alpha$$

sort takes a type, a list of elements of this type, and a comparison function that compares to elements of the type argument and returns a list sorted according to the comparison function. An example of an application of this function could be

$$\text{sort}\ [\mathit{int}]\ (3, 7, 5)\ <$$

Whereas sort instantiated with the string type, but given an integer list would not be a well typed instantiation.

$$\text{sort}\ [\text{string}]\ \cancel{(3, 7, 5)}\ (\text{"}a\text{"}, \text{"}c\text{"}, \text{"}b\text{"})\ \text{string}<$$

Here the application with the list $(3, 7, 5)$ is not well typed, but if we instead use a list of strings, then it type checks.

We want to extend the simply typed lambda calculus with functions that abstract over types in the same way lambda abstractions, $\lambda x : \tau.\ e$, abstract over terms. We do that by introducing a type abstraction:

$$\Lambda\alpha.\ e$$

This function abstracts over the type α which allows e to depend on α.

4.1 System F (STLC with universal types)

$$\begin{aligned}
\tau &::= \ldots \mid \forall\alpha.\ \tau \\
e &::= \ldots \mid \Lambda\alpha.\ e \mid e[\tau] \\
v &::= \ldots \mid \Lambda\alpha.\ e \\
E &::= \ldots \mid E[\tau]
\end{aligned}$$

$$(\Lambda\alpha.\ e)[\tau] \mapsto e[\tau/\alpha]$$

Type environment:

$$\Delta ::= \bullet \mid \Delta,\ \alpha$$

(The type environment is assumed to consist of distinct type variables. For instance, the environment Δ, α is only well-formed if $\alpha \notin \mathrm{dom}(\Delta)$)[7]. With the addition of type environments of type variables our typing judgments now have the following form:

$$\Delta, \Gamma \vdash e : \tau$$

We now need a notion of well-formed types. If τ is well formed with respect to Δ, then we write:

$$\Delta \vdash \tau$$

We do not include the formal rules here, but they amount to $\mathrm{FTV}(\tau) \subseteq \Delta$, where $\mathrm{FTV}(\tau)$ is the set of free type variables in τ.

We further introduce a notion of well formed environments. An environment is well formed if all the types that appear in the range of Γ are well formed.

$$\Delta \vdash \Gamma \stackrel{\text{def}}{=} \forall x \in \mathrm{dom}(\Gamma).\ \Delta \vdash \Gamma(x)$$

For any type judgment $\Delta, \Gamma \vdash e : \tau$, we have as an invariant that τ is well formed in Δ and Γ is well formed in Δ. The old typing system modified to use the new form of the typing judgment looks like this:

$$\frac{}{\Delta; \Gamma \vdash \text{false} : \mathit{bool}}\ \text{T-False} \qquad \frac{}{\Delta; \Gamma \vdash \text{true} : \mathit{bool}}\ \text{T-True} \qquad \frac{\Gamma(x) = \tau}{\Delta; \Gamma \vdash x : \tau}\ \text{T-Var}$$

$$\frac{\Delta; \Gamma \vdash e : \mathit{bool} \qquad \Delta; \Gamma \vdash e_1 : \tau \qquad \Delta; \Gamma \vdash e_2 : \tau}{\Delta; \Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : \tau}\ \text{T-If}$$

$$\frac{\Delta; \Gamma, x : \tau_1 \vdash e : \tau_2}{\Delta; \Gamma \vdash \lambda x : \tau_1.\ e : \tau_1 \to \tau_2}\ \text{T-Abs} \qquad \frac{\Delta; \Gamma \vdash e_1 : \tau_2 \to \tau \qquad \Delta; \Gamma \vdash e_2 : \tau_2}{\Delta; \Gamma \vdash e_1\ e_2 : \tau}\ \text{T-App}$$

Notice that the only thing that has changed is that Δ has been added to the environment in the judgments. We further extend the typing rules with the following two rules to account for our new language constructs:

$$\frac{\Delta; \Gamma \vdash e : \forall \alpha.\tau \qquad \Delta \vdash \tau'}{\Delta; \Gamma \vdash e[\tau'] : \tau[{}^{\tau'}/_{\alpha}]}\ \text{T-TApp} \qquad \frac{\Delta, \alpha; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \Lambda\alpha.\ e : \forall \alpha.\tau}\ \text{T-TAbs}$$

[7] We do not annotate α with a kind as we only have one kind in this language.

Properties of System-F

In System-F, certain types reveal the behavior of the functions with that type. Let us consider terms with the type $\forall\alpha.\ \alpha \to \alpha$. Recall from the Logical Relations section that this had to be the identity function. We can now phrase this as a theorem:

Theorem. *If* $\bullet;\bullet \vdash e : \forall\alpha.\ \alpha \to \alpha$,
$\bullet \vdash \tau$, *and*
$\bullet;\bullet \vdash v : \tau$,
then $e[\tau]\ v \mapsto^* v$

This is a free theorem in this language. Another free theorem that was mentioned in the motivation of lecture 1 was about expressions with type $\forall\alpha.\alpha \to \mathit{bool}$. Here all expressions with this type had to be constant functions. We can also phrase this as a theorem

Theorem. *If* $\bullet \vdash \tau$, $\bullet \vdash v_1 : \tau$,
and $\bullet \vdash v_1 : \tau$,
then $e[\tau]\ v_1 \approx^{ctx} e[\tau]\ v_2$.

Or in a slightly more general fashion where we allow different types:

Theorem. *If* $\bullet \vdash \tau$,
$\bullet \vdash \tau'$,
$\bullet \vdash v_1 : \tau$,
and $\bullet \vdash v_1 : \tau'$,
then $e[\tau]\ v_1 \approx^{ctx} e[\tau']\ v_2$.[8]

We get these free theorems because the functions have no way of inspecting the argument as they do not know what type it is. As the function has to treat its argument as an unknown "blob", it has no choice but to return the same value every time.

The question now is: "how do we prove these free theorems?" The two last theorems both talk about program equivalence which we prove using logical relations. The first theorem did not mention equivalence, but the proof technique of choice is still a logical relation.

[8]We have not yet defined $\approx^{ctx}$ so for now just treat it as the two programs are equivalent without thinking too much about what equivalence means.

4.2 Contextual Equivalence

To define a contextual equivalence, we first define the notion of a program context. A program context is a complete program with exactly one hole in it. It is defined as follows:

$$
\begin{aligned}
C ::= \ & [\cdot] \\
& \mid \text{if } C \text{ then } e \text{ else } e \\
& \mid \text{if } e \text{ then } C \text{ else } e \\
& \mid \text{if } e \text{ then } e \text{ else } C \\
& \mid \lambda x : \tau.\ C \\
& \mid C\ e \\
& \mid e\ C \\
& \mid \Lambda\alpha.\ C \\
& \mid C[\tau]
\end{aligned}
$$

We need a notion of context typing. For simplicity, we just introduce it for simply typed lambda calculus. The context typing is written as:

$$
\frac{\Gamma \vdash e : \tau \qquad \Gamma' \vdash C[e] : \tau'}{C :\ (\Gamma \vdash \tau) \implies (\Gamma' \vdash \tau')}
$$

This means that for any expression e of type τ under Γ if we embed it into C, then the type of the embedding is τ' under Γ'.

Informally we want contextual equivalence to say that no matter what program context we embed either of the two expressions in, it gives the same result. This is also called as observational equivalence as the program context is unable to observe any difference no matter what expression we embed in it. We can of course not plug an arbitrary term into the hole, so we annotate the equivalence with the type of the hole which means that the two contextual equivalent expressions have to have that type.

$$
\begin{aligned}
& \Delta; \Gamma \vdash e_1 \approx^{ctx} e_2 : \tau \\
& \qquad \stackrel{\text{def}}{=} \\
& \forall C :\ (\Delta; \Gamma \vdash \tau) \implies (\bullet; \bullet \vdash \tau').\ (C[e_1] \Downarrow v \iff C[e_2] \Downarrow v)
\end{aligned}
$$

This definition assumes that e_1 and e_2 has type τ under the specified contexts.

Contextual equivalence is handy because we want to be able to reason about the equivalence of two implementations. Say we have two implementations of a stack, one is implemented using an array and the other using a list. If we can show that the two implementations are contextual equivalent, then we can use the more efficient one over the less efficient one and know that the complete program will behave the same. A way this could be used would be to take the simpler stack implementation as a "specification" of what a stack is supposed to do. If the other implementation is a highly optimized stack, then the equivalence proof could be taken as a correctness proof *with respect to* the specification.

In the next lecture, we will introduce a logical relation such that

$$\Delta; \Gamma \vdash e_1 \approx^{LR} e_2 : \tau \implies \text{contextual equivalence } \approx^{ctx}$$

That is we want to show that the logical relation is sound with respect to contextual equivalence.

If we can prove the above soundness, then we can state our free theorems with $\approx^{LR}$ rather than $\approx^{ctx}$ and get the same result if we can prove the logical equivalence. We would like to do this as it is difficult to directly prove two things are contextual equivalent. A direct proof has to talk about all possible program contexts which we could do using induction, but the lambda-abstraction case turns out to be difficult. This motivates the use of other proof methods where using a logical relation is one of them.

4.3 A Logical Relation for System F

Now we need to build a logical relation for System F. With this logical relation, we would like to be able to prove the free theorems from lecture 3. Our value interpretation will now consist of pairs as we are defining a relation. The value relation will have the following form:

$$\mathcal{V}[\![\tau]\!] = \{(v_1, v_2) \mid \bullet; \bullet \vdash v_1 : \tau \ \wedge \ \bullet; \bullet \vdash v_2 : \tau \ \wedge \ \ldots\}$$

In our value interpretation, we require v_1 and v_2 to be closed and well typed, but for succinctness we do not write this in the definitions below. Let us try to naively build the logical relation the same way we build the logical predicates:

$$\begin{aligned}
\mathcal{V}[\![\mathit{bool}]\!] &= \{(\text{true}, \text{true}), (\text{false}, \text{false})\} \\
\mathcal{V}[\![\tau \to \tau']\!] &= \{(\lambda x : \tau.\ e_1, \lambda x : \tau.\ e_2) \mid \\
&\qquad \forall (v_1, v_2) \in \mathcal{V}[\![\tau]\!].\ (e_1[^{v_1}/_x], e_2[^{v_2}/_x]) \in \mathcal{E}[\![\tau']\!]\}
\end{aligned}$$

The value interpretation of the function type is defined based on the slogan for logical relations: "Related inputs to related outputs." If we had chosen to use equal inputs rather than related, then our definition would be more restrictive than necessary.

We did not define a value interpretation for type variables in lecture 3, so let us try to push on without defining that part.

The next type is $\forall\alpha.\ \tau$. When we define the value interpretation, we consider the elimination forms which in this case is type application. Before we proceed, let us consider one of the free theorems from lecture 3 that we wanted to be able to prove:

Theorem. *If* $\bullet \vdash \tau$, $\bullet \vdash \tau'$, $\bullet \vdash v_1 : \tau$, *and* $\bullet \vdash v_1 : \tau'$, *then* $e[\tau]\ v_1 \approx^{ctx} e[\tau']\ v_2 :$ *bool*.

There are some important points to notice in this free theorem. First of all, we want to be able to apply Λ-terms to different types, so in our value interpretation we will have to pick two different types. Further, normally we pick related expressions, so it would probably be a good idea to pick related types. We do not, however, have a notion of related types, and in the theorem there is no relation between the two types used, so relating them might not be a good idea after all. With these points in mind, we can make a first attempt at defining the value interpretation of $\forall\alpha.\ \tau$:

$$\mathcal{V}[\![\forall\alpha.\ \tau]\!] = \{(\Lambda\alpha.\ e_1, \Lambda\alpha.\ e_2) \mid \forall\tau_1, \tau_2.\ (e_1[^{\tau_1}/_\alpha], e_2[^{\tau_2}/_\alpha]) \in \mathcal{E}[\![\tau[^{?}/_\alpha]]\!]\}$$

Now the question is what type to relate the two expressions under. We need to substitute ? for some type, but if we use either τ_1 or τ_2, then the well-typedness requirement will be broken. We choose to leave τ as it is and not do the substitution. We do, however, need to keep track of what types we picked in the left and right part of the pair. To do so, we use a relational substitution:

$$\rho = \{\alpha_1 \mapsto (\tau_{11}, \tau_{12}), \dots\}$$

Which we parameterize the interpretations with.

$$\mathcal{V}[\![\forall\alpha.\ \tau]\!]_\rho = \{(\Lambda\alpha.\ e_1, \Lambda\alpha.\ e_2) \mid \forall\tau_1, \tau_2.\ (e_1[^{\tau_1}/_\alpha], e_2[^{\tau_2}/_\alpha]) \in \mathcal{E}[\![\tau]\!]_{\rho[\alpha\mapsto(\tau_1,\tau_2)]}\}$$

We need to parameterize the entire logical relation with the relational substitution, otherwise we will not know what type to pick when we interpret the polymorphic type variable and we will not know how to close off the values. Which leads us to

the next issue. We are now interpreting types with free type variables, so we need to have a value interpretation of type variable α. It will look something like

$$\mathcal{V}[\![\alpha]\!]_\rho = \{(v_1, v_2) \mid \rho(\alpha) = (\tau_1, \tau_2) \dots\}$$

We need to say that the values are related, but the question is how to relate them. To figure this out, we again look to the free theorem. In the free theorem, the two values are related at the argument type we choose. We therefore pick a relation on these types when we pick the types. We remember the relation we pick in the relational substitution. We finally reach our definition of the value interpretation of $\forall\alpha.\ \tau$:

$$\mathcal{V}[\![\forall\alpha.\ \tau]\!]_\rho = \{(\Lambda\alpha.\ e_1, \Lambda\alpha.\ e_2) \mid \forall\tau_1, \tau_2, R \in \mathrm{Rel}[\tau_1, \tau_2].\\ (e_1[^{\tau_1}/_\alpha], e_2[^{\tau_2}/_\alpha]) \in \mathcal{E}[\![\tau]\!]_{\rho[\alpha\mapsto(\tau_1,\tau_2,R)]}\}$$

We do not require much of the relation, R. It has to be a set of pairs of values, and the values in every pair of the relation have to be closed and well typed under the corresponding type. So we define $\mathrm{Rel}[\tau_1, \tau_2]$ as:

$$\mathrm{Rel}[\tau_1, \tau_2] = \{R \in \mathcal{P}(Val \times Val) \mid \forall(v_1, v_2) \in R.\ \bullet \vdash v_1 : \tau_1\ \wedge\ \bullet \vdash v_2 : \tau_2\}$$

In the interpretation of α, we require the values to be related under the relation we choose in the value interpretation of $\forall\alpha.\ \tau$:

$$\mathcal{V}[\![\alpha]\!]_\rho = \{(v_1, v_2) \mid \rho(\alpha) = (\tau_1, \tau_2, R)\ \wedge\ (v_1, v_2) \in R\}$$

For convenience, we introduce the following notation for projection in ρ. Given

$$\rho = \{\alpha_1 \mapsto (\tau_{11}, \tau_{12}, R_1), \alpha_2 \mapsto (\tau_{21}, \tau_{22}, R_2),\ \dots\}$$

Define the following projections:

$$\rho_1 = \{\alpha_1 \mapsto \tau_{11},\ \alpha_2 \mapsto \tau_{21},\ \dots\}$$
$$\rho_2 = \{\alpha_1 \mapsto \tau_{12},\ \alpha_2 \mapsto \tau_{22},\ \dots\}$$
$$\rho_R = \{\alpha_1 \mapsto R_1,\ \alpha_2 \mapsto R_2,\ \dots\}$$

Notice that ρ_1 and ρ_2 now are type substitutions, so we write $\rho_1(\tau)$ to mean τ where all the type variables mentioned in the substitution has been substituted with the appropriate types. We can now write the value interpretation for type variables in a more succinct way:

$$\mathcal{V}[\![\alpha]\!]_\rho = \rho_R(\alpha)$$

We need to add ρ to the other parts of the value interpretation as well. Moreover, as we now interpret open types, we require the pairs of values in the relation to be well typed under the type closed off using the relational substitution. So all value interpretations have the form

$$\mathcal{V}[\![\tau]\!]_\rho = \{(v_1, v_2) \mid \bullet;\bullet \vdash v_1 : \rho_1(\tau) \ \wedge \ \bullet;\bullet \vdash v_2 : \rho_2(\tau) \ \wedge \ \dots\}$$

We further need to close of the type annotation of the variable in functions, so our value interpretations end up as:

$$\begin{aligned}\mathcal{V}[\![\mathit{bool}]\!]_\rho =& \{(\text{true}, \text{true}), (\text{false}, \text{false})\} \\ \mathcal{V}[\![\tau \to \tau']\!]_\rho =& \{(\lambda x : \rho_1(\tau).\ e_1, \lambda x : \rho_2(\tau).\ e_2) \mid \\ & \qquad \forall (v_1, v_2) \in \mathcal{V}[\![\tau]\!]_\rho.\ (e_1[^{v_1}/_x], e_2[^{v_2}/_x]) \in \mathcal{E}[\![\tau']\!]_\rho\}\end{aligned}$$

We define our interpretation of expressions as follows:

$$\begin{aligned}\mathcal{E}[\![\tau]\!]_\rho = \{(e_1, e_2) \mid & \bullet;\bullet \vdash e_1 : \rho_1(\tau) \wedge \\ & \bullet;\bullet \vdash e_2 : \rho_2(\tau) \wedge \\ & \exists v_1, v_2.\ e_1 \mapsto^* v_1 \ \wedge \ e_2 \mapsto^* v_2 \ \wedge \ (v_1, v_2) \in \mathcal{V}[\![\tau]\!]_\rho\}\end{aligned}$$

We now need to give an interpretation of the contexts Δ and Γ:

$$\begin{aligned}\mathcal{D}[\![\bullet]\!] &= \{\emptyset\} \\ \mathcal{D}[\![\Delta, \alpha]\!] &= \{\rho[\alpha \mapsto (\tau_1, \tau_2, R)] \mid \rho \in \mathcal{D}[\![\Delta]\!] \ \wedge \ R \in \text{Rel}[\tau_1, \tau_2]\} \\ \mathcal{G}[\![\bullet]\!]_\rho &= \{\emptyset\} \\ \mathcal{G}[\![\Gamma, x : \tau]\!]_\rho &= \{\gamma[x \mapsto (v_1, v_2)] \mid \gamma \in \mathcal{G}[\![\Gamma]\!]_\rho \ \wedge \ (v_1, v_2) \in \mathcal{V}[\![\tau]\!]_\rho\}\end{aligned}$$

We need the relational substitution in the interpretation of Γ, because τ might contain free type variables now. We introduce a convenient notation for the projections of γ similar to the one we did for ρ:

$$\gamma = \{\alpha_1 \mapsto (v_{11}, v_{12}), \alpha_2 \mapsto (v_{21}, v_{22}), \ \dots\}$$

Define the projections as follows:

$$\begin{aligned}\gamma_1 &= \{\alpha_1 \mapsto v_{11}, \ \alpha_2 \mapsto v_{21}, \ \dots\} \\ \gamma_2 &= \{\alpha_1 \mapsto v_{12}, \ \alpha_2 \mapsto v_{22}, \ \dots\}\end{aligned}$$

We are now ready to define when two terms are logically related. We define it in a similar way to the logical predicate we already have defined. First we pick ρ and

γ to close off the terms, then we require the closed off expressions to be related under the expression interpretation of the type in question:

$$\begin{aligned}\Delta;\Gamma \vdash e_1 \approx e_2 : \tau \stackrel{\text{def}}{=} & \Delta;\Gamma \vdash e_1 : \tau \wedge \\ & \Delta;\Gamma \vdash e_2 : \tau \wedge \\ & \forall \rho \in \mathcal{D}[\![\Delta]\!]. \\ & \quad \forall \gamma \in \mathcal{G}[\![\Gamma]\!]_\rho. \\ & \quad\quad (\rho_1(\gamma_1(e_1)), \rho_2(\gamma_2(e_2))) \in \mathcal{E}[\![\tau]\!]_\rho\end{aligned}$$

Now we have defined our logical relation, the first thing we want to do is to prove the fundamental property:

Theorem (Fundamental Property)**.** *If* $\Delta;\Gamma \vdash e : \tau$ *then* $\Delta;\Gamma \vdash e \approx e : \tau$

This theorem may seem a bit mundane, but it is actually quite strong. In the definition of the logical relation, Δ and Γ can be seen as maps of place holders that needs to be replaced in the expression. So when we choose a ρ and γ, we may pick different types and terms to put in the expression. Closing the expression off can then give us two very different programs.

In some presentations, this is also known as the parametricity lemma. It may even be stated with out the short-hand notation for equivalence we use here.

We could prove the theorem directly by induction over the typing derivation, but we will instead prove it by means of compatibility lemmas.

Compatibility Lemmas

We state a compatibility for each of the typing rules we have. Each of the lemmas will correspond to a case in the induction proof of the Fundamental Property so the theorem will follow directly from the compatibility lemmas. We state the compatibility lemmas as rules to highlight the connection to the typing rules. The premises of the lemma are over the horizontal line, and the conclusion is below:

1. $\Gamma;\Delta \vdash \text{true} \approx \text{true} : \mathit{bool}$
2. $\Gamma;\Delta \vdash \text{false} \approx \text{false} : \mathit{bool}$
3. $\Gamma;\Delta \vdash x \approx x : \Gamma(x)$
4. $$\frac{\Delta;\Gamma \vdash e_1 \approx e_2 : \tau' \to \tau \qquad \Delta;\Gamma \vdash e_1' \approx e_2' : \tau'}{\Delta;\Gamma \vdash e_1\ e_1' \approx e_2\ e_2' : \tau}$$

$$5.\ \frac{\Delta; \Gamma, x : \tau \vdash e_1 \approx e_2 : \tau'}{\Delta; \Gamma \vdash \lambda x : \tau.\ e_1 \approx \lambda x : \tau.\ e_2 : \tau \to \tau'}$$

$$6.\ \frac{\Delta; \Gamma \vdash e_1 \approx e_2 : \forall \alpha.\tau \qquad \Delta \vdash \tau'}{\Delta; \Gamma \vdash e_1[\tau'] \approx e_2[\tau'] : \tau[^{\tau'}/_{\alpha}]}$$

The rule for if has been omitted here. Notice some of the lemmas are more general than what we actually need. Take for instance the compatibility lemma for expression application. To prove the fundamental property, we really just needed to have the same expressions on both sides of the equivalence. It turns out that the slightly more general version helps when we want to prove that the logical relation is sound with respect to contextual equivalence.

We will only prove the compatibility lemma for type application. To do so, we are going to need the following lemma:

Lemma (Compositionality)**.** *Let* $\Delta \vdash \tau'$*,* $\Delta, \alpha \vdash \tau$*,* $\rho \in \mathcal{D}[\![\Delta]\!]$*, and* $R = \mathcal{V}[\![\tau']\!]_\rho$*, then*

$$\mathcal{V}[\![\tau[^{\tau'}/_{\alpha}]]\!]_\rho = \mathcal{V}[\![\tau]\!]_{\rho[\alpha \mapsto (\rho_1(\tau'), \rho_2(\tau'), R)]}$$

The lemma says syntactically substituting some type for α in τ and then interpreting it is the same as semantically substituting the type for α. To prove this lemma, we would need to show $\mathcal{V}[\![\tau]\!]_\rho \in \mathrm{Rel}[\rho_1(\tau), \rho_2(\tau)]$ which is fairly easy given how we have defined our value interpretation.

Proof. (Compatibility, Lemma 6). What we want to show is

$$\frac{\Delta; \Gamma \vdash e_1 \approx e_2 : \forall \alpha.\tau \qquad \Delta \vdash \tau'}{\Delta; \Gamma \vdash e_1[\tau'] \approx e_2[\tau'] : \tau[^{\tau'}/_{\alpha}]}$$

So we assume (1) $\Delta; \Gamma \vdash e_1 \approx e_2 : \forall \alpha.\tau$ and (2) $\Delta \vdash \tau'$. According to our definition of the logical relation, we need to show three things:

$$\Delta; \Gamma \vdash e_1[\tau'] : \tau[^{\tau'}/_{\alpha}]$$
$$\Delta; \Gamma \vdash e_2[\tau'] : \tau[^{\tau'}/_{\alpha}]$$
$$\forall \rho \in \mathcal{D}[\![\Delta]\!].\ \forall \gamma \in \mathcal{G}[\![\Gamma]\!]_\rho.\ (\rho_1(\gamma_1(e_1[\tau'])), \rho_2(\gamma_2(e_2[\tau']))) \in \mathcal{E}[\![\tau[^{\tau'}/_{\alpha}]]\!]_\rho$$

The two first follows from the well-typedness part of (1) together with (2) and the appropriate typing rule. So it only remains to show the last one.

Suppose we have a ρ in $\mathcal{D}[\![\Delta]\!]$ and a γ in $\mathcal{G}[\![\Gamma]\!]_\rho$. We then need to show:

$$(\rho_1(\gamma_1(e_1[\tau'])), \rho_2(\gamma_2(e_2[\tau']))) \in \mathcal{E}[\![\tau[^{\tau'}/_\alpha]]\!]_\rho$$

From the $\mathcal{E}$-relation, we find that to show this we need to show that the two terms run down to two values and those values are related.

We keep this goal in mind and turn our attention to our premise, (1). This gives us by definition:

$$\forall \rho \in \mathcal{D}[\![\Delta]\!].\ \forall \gamma \in \mathcal{G}[\![\Gamma]\!]_\rho.\ (\rho_1(\gamma_1(e_1)), \rho_2(\gamma_2(e_2))) \in \mathcal{E}[\![\tau]\!]_\rho$$

If we instantiate this with the ρ and γ we supposed previously, then we get

$$(\rho_1(\gamma_1(e_1)), \rho_2(\gamma_2(e_2))) \in \mathcal{E}[\![\tau]\!]_\rho$$

which means that e_1 and e_2 runs down to some value v_1 and v_2 where $(v_1, v_2) \in \mathcal{V}[\![\forall\alpha.\ \tau]\!]_\rho$. As (v_1, v_2) is in the value interpretation of $\forall\alpha.\ \tau$, we know that the values are of type $\forall\alpha.\ \tau$. From this, we know that v_1 and v_2 are type abstractions, so there must exist e_1' and e_2' such that $v_1 = \Lambda\alpha.\ e_1'$ and $v_2 = \Lambda\alpha.\ e_2'$. We can now instantiate $(v_1, v_2) \in \mathcal{V}[\![\forall\alpha.\ \tau]\!]_\rho$ with two types and a relation. We choose $\rho_1(\tau')$ and $\rho_2(\tau')$ as the two type for the instantiation and $\mathcal{V}[\![\tau']\!]_\rho$ as the relation[9]. This gives us

$$(e_1'[^{\rho_1(\tau')}/_\alpha], e_2'[^{\rho_2(\tau')}/_\alpha]) \in \mathcal{E}[\![\tau]\!]_{\rho[\alpha \mapsto (\rho_1(\tau'), \rho_2(\tau'), \mathcal{V}[\![\tau']\!]_\rho)]}$$

For convenience, we write $\rho' = \rho[\alpha \mapsto (\rho_1(\tau'), \rho_2(\tau'), \mathcal{V}[\![\tau']\!]_\rho)]$. From the two expressions membership of the expression interpretation, we know that $e_1'[^{\rho_1(\tau)}/_\alpha]$ and $e_2'[^{\rho_2(\tau)}/_\alpha]$ run down to some values say v_{1_f} and v_{2_f} respectively where $(v_{1_f}, v_{2_f}) \in \mathcal{V}[\![\tau]\!]_{\rho'}$.

Let us take a step back and see what we have done. We have argued that the following evaluation takes place

$$\begin{aligned}
\rho_i(\gamma_i(e_i))[\rho_i(\tau')] &\mapsto^* (\Lambda\alpha.\ e_i')[\rho_1(\tau')] \\
&\mapsto e_i'[^{\rho_i(\tau')}/_\alpha] \\
&\mapsto^* v_{i_f}
\end{aligned}$$

where $i = 1, 2$. The single step in the middle is justified by the type application reduction. The remaining steps are justified in our proof above. If we further note that $\rho_i(\gamma_i(e_i[\tau'])) \equiv \rho_i(\gamma_i(e_i))[\rho_1(\tau')]$, then we have shown that the two expressions

[9]Here we use $\mathcal{V}[\![\tau]\!]_\rho \in \mathrm{Rel}[\rho_1(\tau), \rho_2(\tau)]$ to justify using the value interpretation as our relation.

from our goal in fact do run down to two values, and they are related. More precisely we have:

$$(v_{1_f}, v_{2_f}) \in \mathcal{V}[\![\tau]\!]_{\rho'}$$

but that is not exactly what we wanted them to be related under. We are, however, in luck and can apply the compositionality lemma to obtain

$$(v_{1_f}, v_{2_f}) \in \mathcal{V}[\![\tau[^{\tau'}/_{\alpha}]]\!]_{\rho}$$

which means that they are related under the relation we needed. □

We call theorems that follows as a consequence of parametricity for free theorems. Next we will show a free theorem that says that an expression of the type $\forall\alpha.\ \alpha \to \alpha$ must be the identity function.

Theorem (Free Theorem (I))**.** *If* $\bullet;\bullet \vdash e : \forall\alpha.\ \alpha \to \alpha$*,* $\bullet \vdash \tau$*, and* $\bullet;\bullet \vdash v : \tau$*, then*

$$e[\tau]\ v \mapsto^* v$$

System-F is a terminating language[10], so in the free theorem it suffices to say that when it terminates, then it is with the value passed as argument. If we had been in a non-terminating language such as System F with recursive types, then we would have had to state a weaker theorem namely: the expression terminates with the value given as argument as result, or the computation diverges.

Proof. From the fundamental property and the well-typedness of e, we know $\bullet \vdash e \approx e : \forall\alpha.\ \alpha \to \alpha$. By definition this gives us

$$\forall\rho \in \mathcal{D}[\![\Delta]\!].\ \forall\gamma \in \mathcal{G}[\![\Gamma]\!]_{\rho}.\ (\rho_1(\gamma_1(e)), \rho_2(\gamma_2(e))) \in \mathcal{E}[\![\forall\alpha.\ \alpha \to \alpha]\!]_{\rho}$$

We instantiate this with an empty ρ and an empty γ to get $(e, e) \in \mathcal{E}[\![\forall\alpha.\ \alpha \to \alpha]\!]_{\emptyset}$. From the definition of this, we know that e evaluates to some value F and $(F, F) \in \mathcal{V}[\![\forall\alpha.\ \alpha \to \alpha]\!]_{\emptyset}$. As F is a value of type $\forall\alpha.\ \alpha \to \alpha$, we know $F = \Lambda\alpha.\ e_1$ for some e_1. Now use the fact that $(F, F) \in \mathcal{V}[\![\forall\alpha.\ \alpha \to \alpha]\!]_{\emptyset}$ by instantiating it with the type τ twice and the relation $R = \{(v, v)\}$ to get $(e_1[^{\tau}/_{\alpha}], e_1[^{\tau}/_{\alpha}]) \in \mathcal{E}[\![\alpha \to \alpha]\!]_{\emptyset[\alpha \mapsto (\tau,\tau,R)]}$. We notice that this instantiation is all right as $R \in \mathrm{Rel}[\tau, \tau]$.

This step is an important part of a proof of any free theorem namely choosing the relation. Before we chose the relation, we picked two types. We did this based on the theorem we want to show. In the theorem, we instantiate e with τ, so we pick τ. Likewise with the relation, in the theorem we give v to the function with

[10]For more on this see *Types and Programming Languages* by Benjamin Pierce.

the domain α, so we pick the singleton relation consisting of (v, v). Picking the correct relation is what requires some work in the proof of a free theorem. The remaining work done in the proof is simply unfolding of definitions.

Now let us return to the proof. From $(e_1[^{\tau}/_{\alpha}], e_1[^{\tau}/_{\alpha}]) \in \mathcal{E}[\![\alpha \to \alpha]\!]_{\emptyset[\alpha \mapsto (\tau,\tau,R)]}$, we know that $e_1[^{\tau}/_{\alpha}]$ evaluates to some value g and $(g, g) \in \mathcal{V}[\![\alpha \to \alpha]\!]_{\emptyset[\alpha \mapsto (\tau,\tau,R)]}$. From the type of g, we know that it must be a λ-abstraction, so $g = \lambda x : \tau.\ e_2$ for some expression e_2. Now instantiate $(g, g) \in \mathcal{V}[\![\alpha \to \alpha]\!]_{\emptyset[\alpha \mapsto (\tau,\tau,R)]}$ with $(v, v) \in \mathcal{V}[\![\alpha]\!]_{\emptyset[\alpha \mapsto (\tau,\tau,R)]}$ to get $(e_2[^{v}/_{x}], e_2[^{v}/_{x}]) \in \mathcal{E}[\![\alpha]\!]_{\emptyset[\alpha \mapsto (\tau,\tau,R)]}$. From this we know that $e_2[^{v}/_{x}]$ steps to some value v_f and $(v_f, v_f) \in \mathcal{V}[\![\alpha]\!]_{\emptyset[\alpha \mapsto (\tau,\tau,R)]}$. We have that $\mathcal{V}[\![\alpha]\!]_{\emptyset[\alpha \mapsto (\tau,\tau,R)]} \equiv R$ so $(v_f, v_f) \in R$ which mean that $v_f = v$ as (v, v) is the only pair in R.

Now let us take a step back and consider what we have shown above.

$$
\begin{aligned}
e[\tau]\ v &\mapsto^* F[\tau]\ v \\
&\equiv (\Lambda\alpha.\ e_1)[\tau]\ v \\
&\mapsto (e_1[^{\tau}/_{\alpha}])\ v \\
&\mapsto^* g\ v \\
&\equiv (\lambda x : \tau.\ e_2)\ v \\
&\mapsto e_2[^{v}/_{x}] \\
&\mapsto^* v_f \\
&\equiv v
\end{aligned}
$$

First we argued that $e[\tau]$ steps to some F and that F was a type abstraction, $\Lambda\alpha.\ e_1$. Then we performed the type application to get $e_1[^{\tau}/_{\alpha}]$. We then argued that this steps to some g of the form $\lambda x : \tau.\ e_2$ which further allowed us to do a β-reduction to obtain $\mapsto e_2[^{v}/_{x}]$. We then argued that this reduced to v_f which was the same as v. In summation we argued $e[\tau]\ v \mapsto^* v$ which is the result we wanted. □

4.4 Exercises

1. Prove the following free theorem:

   Theorem (Free Theorem (II))**.** *If* $\bullet;\bullet \vdash e : \forall\alpha.\ ((\tau \to \alpha) \to \alpha)$ *and* $\bullet;\bullet \vdash k : \tau \to \tau_k$*, then*

   $$\bullet;\bullet \vdash e[\tau_k]\ k \approx k(e[\tau]\ \lambda x : \tau.\ x) : \tau_k$$

   This theorem is a simplified version of the one found in *Theorems For Free* by Philip Wadler[1].

5 Existential types

An existential type is reminiscent of a Java interface. It describes some functionality that someone can go off and implement. You can use the existential type without knowing what the actual implementation is going to be.

Take for example a stack. We would expect a stack to have the following functions:

mk creates a new stack.

push puts an element on the top of the stack. It takes a stack and an element and returns the resulting stack.

pop removes the top element of the stack. It takes a stack and returns the new stack along with the element that was popped from it.

An interface would define the above signature which you then would go off and implement[11]. If we wanted to write an interface for a stack, we would write something like (this is meant to be suggestive, so it is in an non-formal notation):

$$\begin{aligned} stack = \exists\alpha.\ \langle & mk : 1 \to \alpha, \\ & push : \alpha \times int \to \alpha, \\ & pop : \alpha \to \alpha \times int \rangle \end{aligned}$$

where α stands for the type that is used in the actual implementation. The above is an interface, it hides all the α's which means that a client cannot see the actual type of the stack which means that they do not know how the stack is actually implemented.

We formally write existentials in a similar fashion to how we wrote universal types:

$$\exists\alpha.\ \tau$$

Here τ is the same as the record in the stack example. The interface is just a type, so now we need to define how one implements something of an existential type. If we were to implement the stack interface, then we would implement a package of functions that are supposed to be used together. This could look something like

[11] There is a famous paper called *Abstract Data Types Have Existential Type* from '86 by Mitchell and Plotkin. The title says it all.

(again this is meant to be suggestive):

$$\begin{array}{l}\text{pack } array[int], \\ \quad \langle \lambda x : \_.\ \ldots\ , \\ \quad\ \lambda x : \_.\ \ldots\ , \\ \quad\ \lambda x : \_.\ \ldots \rangle\end{array}$$

Here $array[int]$ is the type we want to use for the concrete implementation and the record of functions is the concrete implementation that uses $array[int]$ to implement a stack. Let us introduce an example that we can use in the in the rest of this note. Suppose we have the following type:

$$\tau = \exists \alpha.\ \alpha \times (\alpha \to bool)$$

And two terms that we for now claim is of this type:

$$\begin{array}{l} e_1 = \text{pack } \langle int, \langle 1, \lambda x : int.\ x = 0 \rangle \rangle \text{ as } \tau \\ e_2 = \text{pack } \langle bool, \langle \text{true}, \lambda x : bool.\ \text{not } x \rangle \rangle \text{ as } \tau \end{array}$$

Here *int* and *bool* are called the *witness types*. We claim that these two implementations are equivalent and our goal in this note is to show this.

Before we can do that, we need to introduce a bit more. pack is how we create something of existential type, it is our introduction form. We also need an elimination form which is unpack. unpack takes apart a package so that we can use its components. A package consists of a witness type and an expression that implements an existential type. We also need typing rules for these two constructs:

$$\frac{\Delta; \Gamma \vdash e : \tau[^{\tau'}/_{\alpha}] \qquad \Delta \vdash \tau'}{\Delta; \Gamma \vdash \text{pack } \langle \tau', e \rangle \text{ as } \exists \alpha.\tau : \exists \alpha.\tau}$$

$$\frac{\Delta; \Gamma \vdash e_1 : \exists \alpha.\tau \qquad \Delta, \alpha; \Gamma, x : \tau \vdash e_2 : \tau_2 \qquad \Delta \vdash \tau_2}{\Delta; \Gamma \vdash \text{unpack } \langle \alpha, x \rangle = e_1 \text{ in } e_2 : \tau_2}$$

Intuitively, the typing rule of pack says that provided an implementation of the existential type that implementation has to be well-typed when the witness type is plugged in for α. In the typing rule for unpack, it is of importance that α is not free in τ_2 which is ensured by $\Delta \vdash \tau_2$. This is important because the point of a package is to hide away the witness type. Within a certain scope, the witness type can be pulled out of the package using unpack if α could be returned, then it would be exposed to the outer world which would defeat the purpose of hiding it. unpack

takes out the components of e_1 and calls them α and x. The two components can then be used in the body, e_2, of the unpack-expression.

With the typing rules we can type check e_1 and e_2 to verify that they in fact have type τ. Typing of e_1:

$$\dfrac{\dfrac{\dfrac{}{\bullet; x : int \vdash x : int} \quad \dfrac{}{\bullet; x : int \vdash 0 : int}}{\bullet; x : int \vdash x = 0 : bool}}{\dfrac{\dfrac{}{\bullet; \bullet \vdash 1 : int} \quad \bullet; \bullet \vdash \lambda x : int.\ x = 0 : int \to bool}{\bullet; \bullet \vdash \langle 1, \lambda x : int.\ x = 0 \rangle : int \times (int \to bool)} \quad \dfrac{}{\bullet \vdash int}}{\bullet; \bullet \vdash \text{pack } \langle int, \langle 1, \lambda x : int.\ x = 0 \rangle \rangle \text{ as } \tau : \tau}$$

Typing of e_2:

$$\dfrac{\dfrac{\dfrac{}{\bullet; x : bool \vdash x : bool}}{\bullet; x : bool \vdash \text{not } x : bool}}{\dfrac{\dfrac{}{\bullet; \bullet \vdash \text{true} : bool} \quad \bullet; \bullet \vdash \lambda x : bool.\ \text{not } x : bool \to bool}{\bullet; \bullet \vdash \langle \text{true}, \lambda x : bool.\ \text{not } x \rangle : bool \times (bool \to bool)} \quad \dfrac{}{\bullet \vdash bool}}{\bullet; \bullet \vdash \text{pack } \langle bool, \langle \text{true}, \lambda x : bool.\ \text{not } x \rangle \rangle \text{ as } \tau : \tau}$$

To use a package constructed with pack, we need to unpack it with an unpack. If we for instance try to unpack e_1, then we do it as follows:

$$\begin{aligned} &\text{unpack} \langle \alpha, p \rangle = e_1 \text{ in} \\ &\qquad \cancel{(\text{snd } p)\ 5} \\ &\qquad (\text{snd } p)(\text{fst } p) \end{aligned}$$

Here the type int is bound to α, and the pair $(1, \lambda x : int.\ x = 0)$ is bound to p. When we take the second projection of p to get the function out, we cannot apply it to 5 because that would not type check. The environment in which we type check the body of the unpack is $\alpha, p : \alpha \times \alpha \to bool$. So for the expression to type check, we need to apply the function to something of type α. We cannot use 5 as it has type int rather than α. The only thing available of type α and thus the only thing we can give to the function is the first projection of p. We can further not return $(\text{fst}\, p)$ directly as we require α not to be free in the body of the unpack, remember the requirement $\Delta \vdash \tau_2$ in the typing rule.

Likewise for e_2 we can only pass the first projection to the function in the second projection of the package. So the only way we can apply the function in e_2 is:

$$\text{unpack}\langle\alpha, p\rangle = e_2 \,\text{in}$$
$$(\text{snd } p)(\text{fst } p)$$

We can now informally argue why the e_1 and e_2 are equivalent. In e_1, the only value of type α is 1 and in e_2 it is only true. So the related values, R, must be $\{(1, \text{true})\}$. As already stated, these are the only values we can apply the functions to, so we can quickly find the possible values that can be returned. In e_1 it is $(\lambda x : x = 0.\ )\ 1 \mapsto \text{false}$ and in e_2 it is $(\lambda x : \text{not}\, x.\ )\ \mathit{true} \mapsto \text{false}$. The only value that is ever exposed from the package is false. If this claim is true, then it is impossible for a client to observe a difference and thus which package is in fact in use.

To formally argue that e_1 and e_2 are equivalent we need to properly introduce the syntax we have been talking about so far:

$$\begin{aligned}
\tau &::= \ldots \mid \exists\alpha.\ \tau \\
e &::= \ldots \mid \text{pack } \langle\tau, e\rangle \text{ as } \exists\alpha.\tau \mid \text{unpack } \langle\alpha, x\rangle = e \text{ in } e \\
v &::= \ldots \mid \text{pack } \langle\tau, v\rangle \text{ as } \exists\alpha.\tau \\
E &::= \ldots \mid \text{pack } \langle\tau, E\rangle \text{ as } \exists\alpha.\tau \mid \text{unpack } \langle\alpha, x\rangle = E \text{ in } e
\end{aligned}$$

We also need to extend the operational semantics:

$$\text{unpack } \langle\alpha, x\rangle = \text{pack } \langle\tau', v\rangle \text{ as } \exists\alpha.\tau \text{ in } e \mapsto e[^{\tau'}/_{\alpha}][^{v}/_{x}]$$

Now finally we need to extend our value interpretation to consider $\exists\alpha.\ \tau$. The values we relate are of the form $(\text{pack } \langle\tau_1, v_1\rangle \text{ as } \exists\alpha.\tau,\ \text{pack } \langle\tau_2, v_2\rangle \text{ as } \exists\alpha.\tau)$ and as always our first instinct should be to look at the elimination form, so we want to consider $\text{unpack } \langle\alpha, x\rangle = \text{pack } \langle\tau_i, v_i\rangle \text{ as } \exists\alpha.\tau \text{ in } e_i$ for $i = 1, 2$. Now it would be tempting to relate the two bodies, but we get a similar issue to what we had for sum types. If we relate the two bodies, then what type should we relate them under? The type we get might be larger than the one we are interpreting which gives us a well-foundedness problem. So by analogy we do not require that the two unpack expressions have related bodies. Instead we relate v_1 and v_2 under some relation:

$$\begin{aligned}
\mathcal{V}[\![\exists\alpha.\tau]\!]_\rho = \{(&\,\text{pack } \langle\rho_1(\tau_1), v_1\rangle \text{ as } \rho_1(\exists\alpha.\tau), \\
&\text{pack } \langle\rho_2(\tau_2), v_2\rangle \text{ as } \rho_2(\exists\alpha.\tau)) \mid \\
&\qquad \exists R \in \text{Rel}[\rho_1(\tau_1), \rho_2(\tau_2)]. \\
&\qquad\quad (v_1, v_2) \in \mathcal{V}[\![\tau]\!]_{\rho[\alpha \mapsto (\rho_1(\tau_1), \rho_2(\tau_2), R)]}\}
\end{aligned}$$

The relation turns out to be somewhat dual to the one for universal types. Instead of saying $\forall \tau_1, \tau_2, R$, we say $\exists \tau_1, \tau_2, R$, but as we get τ_1 and τ_2 directly from the values, we omit them in the definition. We also relate the two values at τ and extend the relational substitution with the types we have for α. Notice that we use ρ to close of the type variables in the two values we related.

With this extension to the value interpretation, we are ready to show that e_1 and e_2 are logically related. We reuse the definition of logical equivalence we defined previously. What we wish to show is:

Theorem.

$$\bullet;\bullet \vdash e_1 \approx e_2 : \exists\alpha.\alpha \times (\alpha \to \mathit{bool})$$

Proof. With an empty environment, this amounts to show $(e_1, e_2) \in \mathcal{E}[\![\exists\alpha.\alpha\times(\alpha \to \mathit{bool})]\!]_\emptyset$. To show this, we need to establish that e_1 and e_2 evaluates down to some value and that these two values are related under the same type and relational substitution. e_1 and e_2 are pack-expressions so they are already values, so we just need to show $(e_1, e_2) \in \mathcal{V}[\![\exists\alpha.\alpha \times (\alpha \to \mathit{bool})]\!]_\emptyset$. We now need to pick a relation and show that the implementations are related under $\alpha \times (\alpha \to \mathit{bool})$ that is

$$(\langle 1, \lambda x : \mathit{int}.\ x = 0\rangle, \langle \text{true}, \lambda x : \mathit{bool}.\ \text{not}\ x\rangle) \in \mathcal{V}[\![\alpha \times (\alpha \to \mathit{bool})]\!]_{\emptyset[\alpha\mapsto(\mathit{int},\mathit{bool},R)]}$$

We pick $R = \{(1, \text{true})\}$ as the relation. To show that two tuples are related we show that their components are related[12]. So we need to show two things the first is

$$(1, \text{true}) \in \mathcal{V}[\![\alpha]\!]_{\emptyset[\emptyset\mapsto\alpha](\mathit{int},\mathit{bool},R)}$$

Which amounts to showing $(1, \text{true}) \in R$ which is true. The other thing we need to show is:

$$(\lambda x : \mathit{int}.\ x = 0, \lambda x : \mathit{bool}.\ \text{not}\ x) \in \mathcal{V}[\![\alpha \to \mathit{bool}]\!]_{\emptyset[\alpha\mapsto(\mathit{int},\mathit{bool},R)]}$$

Suppose $(v_1, v_2) \in \mathcal{V}[\![\alpha]\!]_{\emptyset[\alpha\mapsto(\mathit{int},\mathit{bool},R)]}$ which is the same as $(v_1, v_2) \in R$. Due to our choice of R, we have $v_1 = 1$ and $v_2 = \mathit{false}$. Now we need to show $(v_1 = 0, \text{not}\ v_2) \in \mathcal{E}[\![\mathit{bool}]\!]_{\emptyset[\alpha\mapsto(\mathit{int},\mathit{bool},R)]}$. Which means that we need to show that the two expressions evaluate to two values related under bool. $v_1 = 0$ evaluates to false as v_1 is 1 and not v_2 evaluates to false as well as $v_2 = \text{true}$, so we need to show $(\text{false}, \text{false}) \in \mathcal{V}[\![\mathit{bool}]\!]_{\emptyset[\alpha\mapsto(\mathit{int},\mathit{bool},R)]}$ which is true by definition of the value interpretation of bool. □

[12] We defined this for logical predicates, but not for logical relations.

6 Recursive Types and Step Indexing

6.1 A motivating introduction to recursive types

First consider the following program in the *untyped* lambda calculus:

$$\Omega = (\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$$

The interested reader can now try to evaluate the above expression. After a β-reduction and a substitution, we end up with Ω again, so the evaluation of this expression diverges. Moreover, it is not possible to assign a type to Ω (again the interested reader may try to verify this by attempting to assign a type). It can hardly come as a surprise that it cannot be assigned a type, as we previously proved that the simply typed lambda calculus is strongly normalizing so if we could assign Ω a type, then it would not diverge.

To type Ω, we need recursive types. If we are able to type Ω, then we do not have strong normalization (as Ω is not strongly normalizing). With recursive types, we can type structures that are inherently inductive such as lists, trees, and streams. In an ML-like language, a declaration of a tree type would look like this:

```
type tree = Leaf
          | Node of int * tree * tree
```

In Java, we could define a tree class with an int field and fields for the sub trees:

```
class Tree {
  int value;
  Tree left, right;
}
```

So we can define trees in our programming languages, but we cannot define them in the lambda calculus. Let us try to find a reasonable definition for recursive types by considering what properties are needed to define trees. We want a type that can either be a node or a leaf. A leaf can be represented by unit (as it here does not carry any information), and a node is the product of an int and two nodes. We put the two constructs together with the sum type, as it is:

$$tree = 1 + (int * tree * tree)$$

This is what we want, but we cannot specify this. We try to define *tree*, but *tree* appears on the right hand side, which is self-referential. Instead of writing *tree*,

we use a type variable α:

$$\begin{aligned}\alpha &= 1 + (int \times \alpha \times \alpha) \\ &= 1 + (int \times (int \times \alpha \times \alpha) \times (int \times \alpha \times \alpha)) \\ &\vdots\end{aligned}$$

All the sides of the above equations are equal, and they are all trees. We could keep going and get an infinite system of equations. If we keep substituting the definition of α for α ,we keep getting bigger and bigger types. All of the types are trees, and all of them are finite. If we take the limit of this process, then we end up with an infinite tree, and that tree is the tree we conceptually have in our minds. So what we need is the fixed point of the above equation.

Let us define a recursive function for which we want to find a fixed point:

$$F = \lambda\alpha :: type.1 + (int \times \alpha \times \alpha)$$

We want the fixed point which by definition is t such that

$$t = F(t)$$

So we want

$$tree = F(tree)$$

The fixed point of this function is written:

$$\mu\alpha.\ F(\alpha)$$

Here μ is a fixed-point type constructor. As the above is the fixed point, then by definition it should be equal to F applied to it:

$$\mu\alpha.\ F(\alpha) = F(\mu\alpha.\ F(\alpha))$$

Now let us make this look a bit more like types by substituting $F(\alpha)$ for τ.

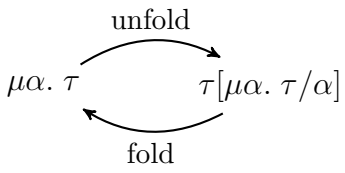
$$\mu\alpha.\ \tau = F(\mu\alpha.\ \tau)$$

The right hand side is really just τ with $\mu\alpha.\ \tau$ substituted with τ:

$$\mu\alpha.\ \tau = \tau[\mu\alpha.\ \tau/\alpha]$$

We are going to introduce the recursive type $\mu\alpha.\ \tau$ to our language. When we have a recursive type, we can shift our view to an expanded version $\tau[\mu\alpha.\ \tau/\alpha]$

and contract back to the original type. Expanding the type is called unfold and contracting is is called fold.



With recursive types in hand, we can now define our tree type:

$$tree \stackrel{\text{def}}{=} \mu\alpha.\ 1 + (int \times \alpha \times \alpha)$$

When we want to work with this, we would like to be able to get under the μ. Say we have $e : tree$ that is an expression e with type $tree$, then we want to be able to say whether it is a leaf or a node. To do so, we unfold the type to get the type where α has been substituted with the definition of $tree$ and the outer $\mu\alpha.$ has been removed. With the outer $\mu\alpha.$ gone, we can match on the sum type to find out whether it is a leaf or a node. When we are done working with the type, we can fold it back to the original tree type.

$$\begin{gathered} tree = \mu\alpha.\ 1 + (int \times \alpha \times \alpha) \\ \text{fold} \updownarrow \text{unfold} \\ 1 + (int \times (\mu\alpha.\ 1 + (int \times \alpha \times \alpha)) \times (\mu\alpha.\ 1 + (int \times \alpha \times \alpha))) \end{gathered}$$

This kind of recursive types is called iso-recursive types, because there is an isomorphism between a $\mu\alpha.\ \tau$ and its unfolding $\tau[\mu\alpha.\ \tau/\alpha]$.

6.2 Simply typed lambda calculus extended with μ

STLC extended with recursive types is defined as follows:

$$\begin{aligned} \tau &::= \ldots \mid \mu\alpha.\ \tau \\ e &::= \ldots \mid \text{fold } e \mid \text{unfold } e \\ v &::= \ldots \mid \text{fold } v \\ E &::= \ldots \mid \text{fold } E \mid \text{unfold } E \end{aligned}$$

$$\frac{}{\text{unfold } (\text{fold } v) \mapsto v} \qquad \frac{\Gamma \vdash e : \tau[\mu\alpha.\ \tau/\alpha]}{\Gamma \vdash \text{fold } e : \mu\alpha.\ \tau}\ \text{T-Fold}$$

$$\frac{\Gamma \vdash e : \mu\alpha.\ \tau}{\Gamma \vdash \text{unfold } e : \tau[\mu\alpha.\ \tau/\alpha]}\ \text{T-Unfold}$$

With this, we could define the type of an integer list as:

$$int\ list \stackrel{\text{def}}{=} \mu\alpha.\ 1 + (int \times \alpha)$$

6.3 Step-indexing, logical relations for recursive types

In a naive first attempt to make the value interpretation, we could write something like

$$\mathcal{V}[\![\mu\alpha.\ \tau]\!] = \{\text{fold } v \mid \text{unfold } (\text{fold } v) \in \mathcal{E}[\![\tau[^{\mu\alpha.\ \tau}/_{\alpha}]]\!]\}$$

We can simplify this slightly; first we use the fact that unfold (fold v) reduces to v. Next we use the fact that v must be a value and the fact that we want v to be in the expression interpretation of $\tau[\mu\alpha.\ \tau/\alpha]$. By unfolding the definition of the expression interpretation, we conclude that it suffices to require v to be in the value interpretation of the same type. We then end up with the following definition:

$$\mathcal{V}[\![\mu\alpha.\ \tau]\!] = \{\text{fold } v \mid v \in \mathcal{V}[\![\tau[^{\mu\alpha.\ \tau}/_{\alpha}]]\!]\}$$

This gives us a well-foundedness issue. The value interpretation is defined by induction on the type, but $\tau[^{\mu\alpha.\ \tau}/_{\alpha}]$ is not a structurally smaller type than $\mu\alpha.\ \tau$.

To solve this issue, we index the interpretation by a natural number, k, which we write as follows:

$$\mathcal{V}_k[\![\tau]\!] = \{v \mid \ldots\}$$

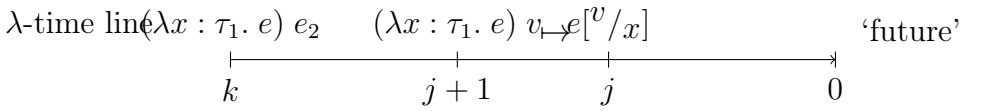
Hence $v \in \mathcal{V}_k[\![\tau]\!]$ is read as "v belongs to the interpretation of τ for k steps." We interpret this in the following way: given a value that we run for k or fewer steps (as in the value is used in some program context for fewer than k steps), then we will never notice that it does not have type τ. If we use the same value in a

program context that wants to run for more than k steps, then we might notice that it does not have type τ which means that we might get stuck. This gives us an approximate guarantee.

We use this as an inductive metric to make our definition well-founded, so we define the interpretation on induction on the step-index followed by an inner induction on the type structure. Let us start by adding the step-index to our existing value interpretation:

$$\begin{aligned}\mathcal{V}_k[\![\mathit{bool}]\!] &= \{\text{true}, \text{false}\} \\ \mathcal{V}_k[\![\tau_1 \to \tau_2]\!] &= \{\lambda x : \tau_1.\ e \mid \forall j \leq k.\ \forall v \in \mathcal{V}_j[\![\tau_1]\!].\ e[^v/_x] \in \mathcal{E}_j[\![\tau_2]\!]\}\end{aligned}$$

true and false are in the value interpretation of *bool* for any k, so true and false will for any k look like it has type *bool*. To illustrate how to understand the value interpretation of $\tau_1 \to \tau_2$, please consider the following time line:



Here we start at index k and as we run the program, we use up steps until we at some point reach 0 and run out of steps. At step k, we are looking at a lambda. A lambda is used by applying it but it is not certain that the application will happen right away. We only do a β-reduction when we try to apply a lambda to a value, but we might be looking at a context where we want to apply the lambda to an expressions, i.e. $(\lambda x : \tau_1.\ e)\ e_2$. We might use a bunch of steps to reduce e_2 down to a value, but we cannot say how many. So say that sometime in the future we have fully evaluated e_2 to v and say that we have $j+1$ steps left at this time, then we can do the β reduction which gives us $e[^v/_x]$ at step j.

We can now define the value interpretation of $\mu\alpha.\ \tau$:

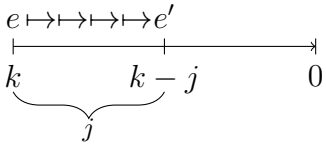
$$\mathcal{V}_k[\![\mu\alpha.\ \tau]\!] = \{\text{fold}\ \ v \mid \forall j < k.\ v \in \mathcal{V}_j[\![\tau[^{\mu\alpha.\ \tau}/_\alpha]]\!]\}$$

This definition is like the one we previously proposed, but with a step-index. This definition is well-founded because j is required to be *strictly* less than k and as we define the interpretation on induction over the step-index this is indeed well founded. We do not define a value interpretation for type variables α, as we have no polymorphism yet. The only place we have a type variable at the moment is in $\mu\alpha.\ \tau$, but in the interpretation we immediately close off the τ under the μ, so we will never encounter a free type variable.

Finally, we define the expression interpretation:

$$\mathcal{E}_k[\![\tau]\!] = \{e \mid \forall j < k.\, \forall e'.\, e \mapsto^j e' \ \wedge\ \text{irred}(e') \implies e' \in \mathcal{V}_{k-j}[\![\tau]\!]\}$$

To illustrate what is going on here, consider the following time line:



We start with an expression e, then we take j steps and get to expression e'. At this point, if e' is irreducible, then we want it to belong to the value interpretation of τ for $k-j$ steps. We use a strict inequality because we do not want to hit 0 steps. If we hit 0 steps, then we do not have any computational steps to observe a difference, so all bets are off.

We also need to lift the interpretation of type environments to step-indexing:

$$\begin{aligned}\mathcal{G}_k[\![\bullet]\!] &= \{\emptyset\} \\ \mathcal{G}_k[\![\Gamma, x:\tau]\!] &= \{\gamma[x \mapsto v] \mid \gamma \in \mathcal{G}_k[\![\Gamma]\!] \ \wedge\ v \in \mathcal{V}_k[\![\tau]\!]\}\end{aligned}$$

Finally we are in a position to lift the definition of semantic type safety to one with step-indexing.

$$\Gamma \models e : \tau \stackrel{\text{def}}{=} \forall k \geq 0.\, \forall \gamma \in \mathcal{G}_k[\![\Gamma]\!] \implies \gamma(e) \in \mathcal{E}_k[\![\tau]\!]$$

To actually prove type safety, we do it in two steps. First we state and prove the fundamental theorem:

Theorem (Fundamental property)**.**
If $\Gamma \vdash e : \tau$*, then* $\Gamma \models e : \tau$*.*

When we have proven the fundamental theorem, we prove that it entails type safety.

ⓑ $\bullet \models e : \tau \implies \text{safe}(e)$

Thanks to the way we defined the logical predicate, this second step should be trivial to prove.

To actually prove the fundamental theorem, which is the challenging part, we need to prove a monotonicity lemma:

Lemma (Monotonicity)**.**
If $v \in \mathcal{V}_k[\![\tau]\!]$ *and* $j \leq k$*, then* $v \in \mathcal{V}_j[\![\tau]\!]$*.*

Proof. The proof is by case on τ.
Case $\tau = \mathit{bool}$, assume $v \in \mathcal{V}_k[\![\mathit{bool}]\!]$ and $j \leq k$, we then need to show $v \in \mathcal{V}_j[\![\mathit{bool}]\!]$. As $v \in \mathcal{V}_k[\![\mathit{bool}]\!]$, we know that either $v = \text{true}$ or $v = \text{false}$. If we assume $v = \text{true}$, then we immediately get what we want to show, as true is in $\mathcal{V}_j[\![\mathit{bool}]\!]$ for any j. Likewise for the case $v = \text{false}$.
Case $\tau = \tau_1 \to \tau_2$, assume $v \in \mathcal{V}_k[\![\tau_1 \to \tau_2]\!]$ and $j \leq k$, we then need to show $v \in \mathcal{V}_j[\![\tau_1 \to \tau_2]\!]$. As v is a member of $\mathcal{V}_k[\![\tau_1 \to \tau_2]\!]$, we can conclude that $v = \lambda x : \tau_1.\ e$ for some e. By definition of $v \in \mathcal{V}_j[\![\tau_1 \to \tau_2]\!]$ we need to show $\forall i \leq j. \forall v' \in \mathcal{V}_i[\![\tau_1]\!].\ e[^{v'}/_x] \in \mathcal{E}_i[\![\tau_2]\!]$. Suppose $i \leq j$ and $v' \in \mathcal{V}_i[\![\tau_1]\!]$, we then need to show $e[^{v'}/_x] \in \mathcal{E}_i[\![\tau_2]\!]$.

By assumption, we have $v \in \mathcal{V}_k[\![\tau_1 \to \tau_2]\!]$ which gives us $\forall n \leq k.\ \forall v' \in \mathcal{V}_n[\![\tau_1]\!].\ e[^{v'}/_x] \in \mathcal{E}_n[\![\tau_2]\!]$. From $j \leq k$ and $i \leq j$, we get $i \leq k$ by transitivity. We use this with $v' \in \mathcal{V}_i[\![\tau_1]\!]$ to get $e[^{v'}/_x] \in \mathcal{E}_i[\![\tau_2]\!]$ which is what we needed to show.
Case $\tau = \mu\alpha.\ x$, assume $v \in \mathcal{V}_k[\![\mu\alpha.\ \tau]\!]$ and $j \leq k$, we then need to show $v \in \mathcal{V}_j[\![\mu\alpha.\ \tau]\!]$. From v's assumed membership of the value interpretation of τ for k steps, we conclude that there must exist a v' such that $v = \text{fold}\ v'$. If we suppose $i < j$, then we need to show $v' \in \mathcal{V}_i[\![\tau[^{\mu\alpha.\ \tau}/_\alpha]]\!]$. From $i < j$ and $j \leq k$, we can conclude $i < k$ which we use with $\forall n < k.\ v' \in \mathcal{V}_n[\![\tau[^{\mu\alpha.\ \tau}/_\alpha]]\!]$, which we get from $v \in \mathcal{V}_k[\![\mu\alpha.\ \tau]\!]$, to get $v' \in \mathcal{V}_i[\![\tau[^{\mu\alpha.\ \tau}/_\alpha]]\!]$. □

Proof (Fundamental Property). Proof by induction over the typing derivation.
Case $\dfrac{\Gamma \vdash e : \tau[\mu\alpha.\ \tau/\alpha]}{\Gamma \vdash \text{fold}\ e : \mu\alpha.\ \tau}$ T-Fold,
We need to show v

$$\Gamma \models \text{fold}\ e : \mu\alpha.\tau$$

So suppose we have $k \geq 0$ and $\gamma \in \mathcal{G}_k[\![\mu\alpha.\tau]\!]$, then we need to show $\gamma(\text{fold}\ e) \in \mathcal{E}_k[\![\mu\alpha.\tau]\!]$ which amounts to showing fold $\gamma(e) \in \mathcal{E}_k[\![\mu\alpha.\tau]\!]$.

So suppose that $j < k$ and that fold $\gamma(e) \mapsto^j e'$ and irred(e'), then we need to show $e' \in \mathcal{V}_{k-j}[\![\mu\alpha.\tau]\!]$. As we have assumed that fold $\gamma(e)$ reduces down to something irreducible and the operational semantics of this language are deterministic, we know that $\gamma(e)$ must have evaluated down to something irreducible. We therefore know that $\gamma(e) \mapsto^{j_1} e_1$ where $j_1 \leq j$ and irred(e_1). Now we use our induction hypothesis:

$$\Gamma \models e : \tau[^{\mu\alpha.\tau}/_\alpha]$$

We instantiate this with k and $\gamma \in \mathcal{G}_k[\![\Gamma]\!]$ to get $\gamma(e) \in \mathcal{E}_k[\![\tau[^{\mu\alpha.\tau}/_{\alpha}]]\!]$. Which we then can instantiate with j_1 and e_1 to get $e_1 \in \mathcal{V}_{k-j_1}[\![\tau[^{\mu\alpha.\tau}/_{\alpha}]]\!]$. Now let us take a step back and see what happened: We started with a fold $\gamma(e)$ which took j_1 steps to fold e_1. We have just shown that this e_1 is actually a value as it is in the value interpretation of $\mathcal{V}_{k-j_1}[\![\tau[^{\mu\alpha.\tau}/_{\alpha}]]\!]$. To remind us e_1 is a value let us henceforth refer to it as v_1. We further know that fold $\gamma(e)$ reduces to e' in j steps and that e' was irreducible. fold v_1 is also irreducible as it is a value and as our language is deterministic, it must be the case that $e' = \text{fold } v_1$ and thus $j = j_1$. Our proof obligation was to show $e' = \text{fold } v_1 \in \mathcal{V}_{k-j}[\![\mu\alpha.\tau]\!]$ to show this suppose we have $l < k - j$ (this also gives us $l < k - j_1$ as $j = j_1$). We then need to show $v_1 \in \mathcal{V}_l[\![\tau[^{\mu\alpha.\tau}/_{\alpha}]]\!]$, we obtain this result from the monotonicity lemma using $\mathcal{V}_{k-j_1}[\![\tau[^{\mu\alpha.\tau}/_{\alpha}]]\!]$ and $l < k - j_1$. □

The list example from the previous lecture used the sum type. Sums are a straight forward extension of the language. The extension of the value interpretation would be:

$$\mathcal{V}_k[\![\tau_1 + \tau_2]\!] = \{\text{inl } v_1 \mid v_1 \in \mathcal{V}_k[\![\tau_1]\!]\} \cup \{\text{inr } v_2 \mid v_2 \in \mathcal{V}_k[\![\tau_2]\!]\}$$

We can use k directly or k decremented by one. It depends on whether we want casing to take up a step. Either way the definition is well-founded.

6.4 Exercises

1. Do the lambda and application case of the *Fundamental Property* theorem.

2. Try to prove the *monotonicity* lemma where the definition of the value interpretation has been adjusted with:

   $$\mathcal{V}_k[\![\tau_1 \to \tau_2]\!] = \{\lambda x : \tau_1.\ e \mid \ \forall v \in \mathcal{V}_k[\![\tau_1]\!].\ e[^v/_x] \in \mathcal{E}_k[\![\tau_2]\!]\}$$

   This will fail, but it is instructive to see how it fails.

Acknowledgments

It is established practice for authors to accept responsibility for any and all mistakes in documents like this. I, however, do not. If you find anything amiss, please let me know so I can figure out who of the following are to blame: Amal Ahmed, Morten Krogh-Jespersen, Kent Grigo, or Kristoffer Just Andersen.

References

[1] Philip Wadler. Theorems for free! In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pages 347–359, New York, NY, USA, 1989. ACM.