Exam

Essentials of Programming Languages, 2023 WS

Prof. Dr. Peter Thiemann
Hannes Saffrich

March 3, 2024

Submission Deadline: March 31, 2024 at 23:59 (via email[1])

Overview

The exam is divided into four parts. Each part provides you with the specification of a language (syntax, typing, semantics) in Agda-code, and requires you to prove that this language satisfies certain properties. To help you prove these properties, we provide you with a proof skeleton, i.e. the types of the lemmas, which we used to prove the properties ourselves.

Each part has its own subdirectory, which contains the specification of the language (e.g. `Exam/Part1/Specification.agda`) and for each task (property you need to prove) a separate `.agda`-file containing the statement of the property and its lemmas.

The individual parts are summarized as followed:

- In Part 1 (20 points) you are presented with two simple languages that consist only of natural numbers and addition and your task is to prove properties, which relate the two languages and their semantics.
- In Part 2 (20 points) your task is to prove type soundness for a simply typed lambda calculus with the unit type.
- In Part 3 (45 points) your task is to prove type soundness for a minimal dependently typed lambda calculus.
- In Part 4 (15 points) your task is to prove type soundness for the language from Part 3, but extended with dependent pairs and an equality type.

The lambda calculi in parts 2 to 4 are formalized with intrinsic scoping (similar as we did in the *Untyped* chapter) and extrinsic typing (see the recording of the last tutorial session). Part 2 serves as an introduction to this style of formalization. The parts are designed to be worked on in the order that they're presented and increase in difficulty. Part 3 probably has the largest cognitive overhead as you need to think yourself into a larger system with many definitions.

[1] `saffrich@informatik.uni-freiburg.de`

Rules

- It is **not** allowed to change the specification of the languages in any way, i.e. the `Specification.agda` files must remain unchanged.
- It is **not** allowed to change the properties you need to prove, e.g. the type of `progress` in `Exam/Part1/Progress.agda` must remain unchanged.
- It is **not** allowed to use postulates, except for functional extensionality, which is already defined in the `Specification.agda` files of the parts, where this postulate is relevant.
- Proving the property required by a task completely gives full points for that task, independently of code style and code complexity. The property is proven completely, if there are no holes left and also no unsolved meta variables or constraints (code highlighted with yellow background, e.g. from implicit arguments that Agda cannot resolve).
- If you are not able to finish the proof of the property required by a task, then you may still get partial credit for progress on the properties and the lemmas we provided. It also counts as progress if you successfully use other lemmas we provided, even if you weren't able to prove those other lemmas themselves.
- It is allowed to change or ignore the lemmas we provide you, e.g. in `Exam/Part1/Preservation.agda` everything except the `preservation` theorem could be changed or deleted. **However, this implies that partial progress may not be graded:** if you are not able to finish a task, then it is up to us to decide if your own lemmas actually contribute in a meaningful way to the task. Hence, we strongly recommend to use the proof skeleton we provided.
- All parts together are worth 100 points in total. To **pass the exam**, at least 50 points have to be achieved.

Hints

In some files you may notice the following comment in the first line:

```
{-# OPTIONS --allow-unsolved-metas #-}
```

This comment is a pragma, which allows you to import files that still have holes in them, so you can do the proofs in any order you wish.

Part 1: Natural Numbers and Addition

The file `Exam/Part1/Specification.agda` defines two tiny languages that support only natural numbers and addition.

The first language models natural number literals as primitive terms (the `#_` constructor of `Expr`), whereas the second language models them via zero and successor terms (the `'suc` and `'zero` constructors of `Expr'`).

For both languages a small-step semantics (`_↪_`) and a denotational semantics (`⟦_⟧`) are provided. The denotational semantics is basically an interpreter for the languages written in Agda, i.e. a function that maps terms to natural numbers.

Furthermore, translation functions are defined, which allow to convert terms between the two languages.

We want to show for each language that the denotational semantics behaves like the small-step semantics, as well as that the translation functions preserve semantics, i.e. that a term in one language evaluates to the same number as its translation evaluates to in the other language.

Your Tasks

- Prove all the lemmas in `Exam/Part1/Properties.agda`;
  Points: 20; Difficulty: ⋆

Part 2: STLC with Unit

Intrinsic Typing

In the *DeBruijn* chapter of the *PLFA* book, we saw a formalization of a simply typed lambda calculus with intrinsically typed terms. *Intrinsically typed* means that there are no separate definitions of terms and the typing relation, but instead a type of type-correct terms is defined directly:

$$\_\vdash\_ : \mathsf{Context} \to \mathsf{Type} \to \mathsf{Set}$$

Given $\Gamma : \mathsf{Context}$ and $\mathsf{t} : \mathsf{Type}$, the type $\Gamma \vdash \mathsf{t}$ describes all terms, which have type t assuming their free variables have types according to Γ. This means there is no way to even talk about terms, which are ill-typed, e.g. a term that uses a number as a function like $2 \cdot 3$.

As a consequence, not only the definition of terms and the typing relation are merged into one definition, but all operations and relations, we define on terms, also implicitly state that they preserve the typings of terms, e.g.

- the operation of substitution also states that if we substitute a well-typed term into another well-typed term, then the result will again be a well-typed term.
- the definition of the small-step reduction relation does not just relate terms with terms, but well-typed terms with well-typed terms, which amounts to proving the preservation lemma, i.e. that if a well-typed term reduces, then the result is again a well-typed term.

While formalizing a language using intrinsic typing has many upsides, it also has the downside that it is not possible to talk about untyped terms anymore (which is sometimes necessary), and that the specification of the language is mixed up with the proofs about its properties, which can make it harder to see if the language is actually specified correctly. Furthermore, languages with dependent types cannot be formalized with intrinsic types at all.

Extrinsic Typing with Intrinsic Scoping

In the *Lambda* and *Properties* chapters of the *PLFA* book, we saw a simply typed lambda calculus with extrinsically typed terms. *Extrinsically typed* means that separate definitions are used for terms and the typing relation. Consequently, for each operation or relation we define, we need to prove a separate lemma, which states that the operation or relation preserves types, e.g. if we apply a well-typed substitution to a well-typed term, the result will again be a well-typed term.

In this exam, we also use extrinsic typing, but instead of representing variables as strings, as we did in the *Lambda* chapter, we use the DeBruijn representation for variables, similar as we did in the *Untyped* chapter.

In contrast to the *Untyped* chapter, we model intinsically scoped terms not as intrinsically typed terms with a trivial type system, where every term has

the same "fake type" $\star$, but simply index our `Term` type with a natural number that specifies how many different free variables the term has:

$$\mathsf{Term} : \mathbb{N} \to \mathsf{Set}$$

Consequently, `Term 0` represents all terms with no free variables, i.e. complete programs without undefined variables. If a lambda term has type `Term n`, then its body has type `Term (suc n)`, since it is allowed to use one additional variable. Since variables are represented as DeBruijn-Indices, a variable term of type `Term n` corresponds to a number between 0 and `n-1`. To describe natural numbers between `0` and `n-1`, we use the type `Fin n` as defined in `Exam/Util/Fin.agda`.

To properly define the typing relation, we also need to add a new index to the typing context:

$$\mathsf{Context} : \mathbb{N} \to \mathsf{Set}$$

A `Context n` represents a list of exactly n `Type`s. The additional index for the length of the context allows us to ensure that in the typing relation the context Γ provides types for each of the free variables in the term:

$$\_\vdash\_:\_ : \forall\ \{\mathsf{n} : \mathbb{N}\} \to \mathsf{Context\ n} \to \mathsf{Term\ n} \to \mathsf{Type} \to \mathsf{Set}$$

A value of type $\Gamma \vdash \mathtt{e} : \mathtt{t}$ is a proof that the term `e` has type `t` assuming the free variables of `e` have the types described in the context Γ.

Hint: Modelling intrinsically-scoped terms this way was also discussed in the last tutorial session. See the recording for more information.

Your Tasks

- Prove that the language satisfies the `progress` property.
  File: `Exam/Part2/Progress.agda`; Points: 5; Difficulty: $\star$
- Prove that the language satisfies the `preservation` property.
  File: `Exam/Part2/Preservation.agda`; Points: 15; Difficulty: $\star$

Part 3: Minimal Dependent Types

During this semester, we have been using Agda – a dependently-typed language – to model other languages and prove properties about them. However, we never looked at how a dependently-typed language itself can be modeled. The rest of this exam is concerned with doing exactly that: in this part, we start out with a minimal dependently typed language and then extend it in Part 4 and 5 with dependent pairs and an equality type.

Types are Terms

The languages, which we modeled during this course, always had a clear distinction between terms and types, i.e. we had one `data` definition for terms, one for types, and the typing relation was relating terms with types:

```
data Term : Set where ...
data Type : Set where ...
data _⊢_:_ : Context → Term → Type → Set where ...
```

In dependently typed languages this distinction is gone: types are also considered terms, and the typing relation relates terms with terms:

```
data Term : Set where ...
data _⊢_:_ : Context → Term → Term → Set where ...
```

Which terms are conceptionally considered as types is entirely up to how the typing relation is defined.

Syntax

The syntax of our language is defined by the following grammar:

$$t ::= x \mid \lambda x.t \mid t\ t \mid \forall(x : t) \to t \mid \texttt{Set}$$

A term t can be either a variable x, a lambda term $\lambda x.t$, a function application $t_1\ t_2$, a dependent function type $\forall(x : t_1) \to t_2$, or the type of types `Set`.

A dependent function type $\forall(x : t_1) \to t_2$ allows x to be used in t_2, just as we are used to it in Agda. If x is not used in t_2, then $\forall(x : t_1) \to t_2$ is equivalent to the non-dependent function type $t_1 \Rightarrow t_2$, which we already know from the simply typed lambda calculus. Since our language is very minimalistic, the only way that we can write a dependent function, which actually uses x in t_2 is by writing polymorphic functions. Consider for example the polymorphic identity function in Agda:

```
id : ∀ (A : Set) → A → A
id A x = x
```

This identity function corresponds in our language to the term $\lambda A.\lambda x.x$, to which our typing relation can assign the type $\forall(A : \texttt{Set}) \to \forall(x : A) \to A$.

Typing

The typing relation $\_\vdash\_:\_$ is defined by the following inference rules:

$$\frac{(x : t) \in \Gamma}{\Gamma \vdash x : t}\ \textsc{TVar} \qquad \frac{\Gamma \vdash t_1 : \texttt{Set} \qquad \Gamma, x : t_1 \vdash e : t_2}{\Gamma \vdash \lambda x.e : \forall(x : t_1) \to t_2}\ \textsc{TLam}$$

$$\frac{\Gamma \vdash e_1 : \forall(x : t_1) \to t_2 \qquad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1\ e_2 : t_2[x \mapsto e_2]}\ \textsc{TApp} \qquad \frac{\Gamma \vdash t_1 : \texttt{Set} \qquad \Gamma, x : t_1 \vdash t_2 : \texttt{Set}}{\Gamma \vdash \forall(x : t_1) \to t_2 : \texttt{Set}}\ \textsc{TAll}$$

$$\Gamma \vdash \texttt{Set} : \texttt{Set}\ \textsc{TSet} \qquad \frac{\Gamma \vdash e : t_1 \qquad t_1 \approx t_2}{\Gamma \vdash e : t_2}\ \textsc{TConv}$$

The rule TVar is exactly as we know it from the simply typed lambda calculus.

The rule TAbs is similar as in the simply typed lambda calculus, but requires additionally that the function's parameter type t_1 has type `Set`.[2] Just like in Agda, if a term has type `Set`, this means that the term is conceptionally a type, and not some other term like a lambda-function.

The rule TApp is similar as in the simply typed lambda calculus, but concludes a different return type for the application. This is because e_1 has a dependent function type, so t_2 may contain the variable x, which stands for the argument to which we apply e_1. Consequently, if we apply e_1 to the argument e_2, we don't just get back a value of type t_2, but of t_2 where x has been replaced by the argument e_2. This behavior can also be observed in our identity example in Agda:

```
id-nat : ℕ → ℕ
id-nat = id ℕ
```

Here, the application is `id ℕ`, where `id` has type $\forall(A : \texttt{Set}) \to \forall(x : A) \to A$, so t_2 is $\forall(x : A) \to A$ and the result type of the application is $t_2[A \mapsto \mathbb{N}]$, which is $\forall(x : \mathbb{N}) \to \mathbb{N}$.

The rule TAll states that a dependent function type has type `Set`, if both its parameter type t_1 and its return type t_2 have type `Set`. As the return type t_2 is allowed to use x, we need to extend the typing context Γ with the type of x, which follows the same reasoning as for the body of a lambda term in TLam.

The rule TSet states that `Set` itself is a type.[3] This is important for example

[2] If you are wondering why we don't require also t_2 to have type `Set`: this is because our typing relation enjoys the invariant that if all types in Γ have type `Set` and $\Gamma \vdash e : t$, then also t has type `Set`. This makes it sufficient to assert only that the types that we put into the typing context have type `Set`.

[3] Here we differ from Agda, where `Set` has type $\texttt{Set}_1$, and $\texttt{Set}_1$ has type $\texttt{Set}_2$ and so on. Having `Set : Set` makes our language logically inconsistent as it allows us to encode paradoxes aka non-terminating programs. We do this only to keep the language minimal, but it is not very complicated to add a universe hierarchy to the formalization to make it logically consistent again.

for our identity function, because the rule TAll says that $\forall(A : \texttt{Set}) \to \ldots$ is only a type, if `Set` is a type.

The rule TConv states that if a term has type t_1, then we can also give it a type t_2, if t_1 is convertible to t_2 (written as $t_1 \approx t_2$). This flexibility is crucial for dependent types and can also be observed in Agda:

```
example : 2 + 2 ≡ 4
example = refl {ℕ} {4}
```

For clarity, we explicitly wrote the implicit arguments for refl, which Agda normally infers for us. Here the term `refl {ℕ} {4}` has type $4 \equiv 4$, but the type we gave to example is $2 + 2 \equiv 4$. While the types $4 \equiv 4$ and $2 + 2 \equiv 4$ are syntactically not the same, they are what we call *convertible* or *definitionally equal*, i.e. they are terms which in zero or more steps reduce to the same term. To define the convertibility relation `_≈_` formally, we first need to introduce the semantics.

Semantics

The small-step reduction relation $\_\hookrightarrow\_$ is defined by the following inference rules:

$$\textsc{RBetaLam}\quad (\lambda x.t_1)\ t_2 \hookrightarrow t_1[x \mapsto t_2] \qquad \textsc{RXiLam}\quad \frac{t \hookrightarrow t'}{\lambda x.t \hookrightarrow \lambda x.t'} \qquad \textsc{RXiApp1}\quad \frac{t_1 \hookrightarrow t_1'}{t_1\ t_2 \hookrightarrow t_1'\ t_2} \qquad \textsc{RXiApp2}\quad \frac{t_2 \hookrightarrow t_2'}{t_1\ t_2 \hookrightarrow t_1\ t_2'}$$

$$\textsc{RXiAll1}\quad \frac{t_1 \hookrightarrow t_1'}{(\forall(x : t_1) \to t_2) \hookrightarrow (\forall(x : t_1') \to t_2)} \qquad \textsc{RXiAll2}\quad \frac{t_2 \hookrightarrow t_2'}{(\forall(x : t_1) \to t_2) \hookrightarrow (\forall(x : t_1) \to t_2')}$$

We have one β-reduction rule, RBetaLam, which is just as in the simply typed lambda calculus: an application eliminates a lambda term via substitution. The rest are congruency rules, which allow for the reduction in subterms. We allow full reduction as in the *Untyped* chapter of the PLFA book, i.e. there is no evaluation order enforced with additional `Value` requirements, and we also allow reduction in the body of a lambda term via RXiLam. As types are terms, we have the RXiAll1 and RXiAll2 rules to allow reduction in the subterms of a dependent function type.

We define the reflexive, transitive closure of the reduction relation as usual, i.e. $t_1 \hookrightarrow^* t_2$ means that t_2 can be reached from t_1 by taking zero or more $\hookrightarrow$-steps.

Two terms are convertible, $t_1 \approx t_2$, iff there exists some term t, such that $t_1 \hookrightarrow^* t$ and $t_2 \hookrightarrow^* t$.

As we allow reduction in the body of a lambda term, we need to define values mutually recursive with neutral terms as in the *Untyped* chapter of the PLFA book. The intuitive reason for this is that if our term contains a free

variable, then this free variable might block reduction, so we need to generalize our notion of Value.

$$\textsc{VLam}\ \frac{\mathsf{Value}\ t}{\mathsf{Value}\ (\lambda x.t)} \qquad \textsc{VAll}\ \frac{\mathsf{Value}\ t_1 \qquad \mathsf{Value}\ t_2}{\mathsf{Value}\ (\forall (x : t_1) \to t_2)} \qquad \textsc{VSet}\ \mathsf{Value}\ \mathtt{Set} \qquad \textsc{VNeu}\ \frac{\mathsf{Neutral}\ t}{\mathsf{Value}\ t}$$

$$\textsc{NVar}\ \mathsf{Neutral}\ x \qquad \textsc{NApp}\ \frac{\mathsf{Neutral}\ t_1 \qquad \mathsf{Value}\ t_2}{\mathsf{Neutral}\ (t_1\ t_2)}$$

Properties

We want to prove type soundness, i.e. that the type system correctly describes the semantics. As we saw during this course, this consists in proving a *progress* and a *preservation* lemma:

Lemma (Progress)**.** *If $\emptyset \vdash e : t$, then either e is a value or there exists some e' such that $e \hookrightarrow e'$.*

Lemma (Preservation)**.** *If $\emptyset \vdash e : t$ and $e \hookrightarrow e'$, then $\emptyset \vdash e' : t$.*

However, as we are using a semantics, which allows to reduce in the body of a lambda term, we need to generalize these lemmas, such that they also apply to terms with free variables by allowing an arbitrary typing context:

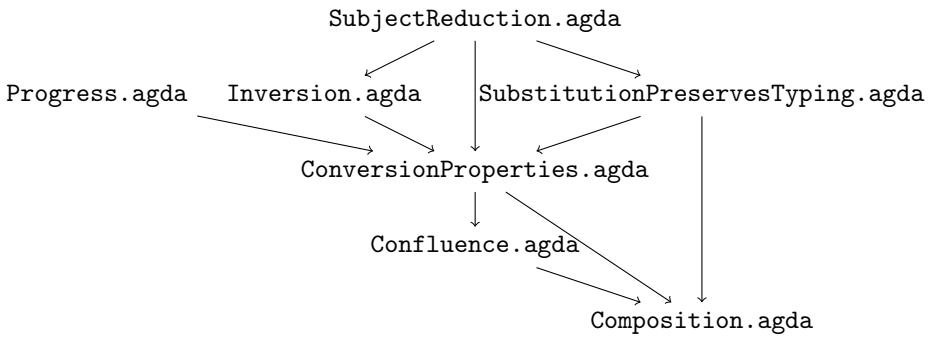
Lemma (Progress)**.** *If $\Gamma \vdash e : t$, then either e is a value or there exists some e' such that $e \hookrightarrow e'$.*

Lemma (Subject Reduction)**.** *If $\Gamma \vdash e : t$ and $e \hookrightarrow e'$, then $\Gamma \vdash e' : t$.*

The generalized version of *preservation* is called *subject reduction.*

Proof Structure

The dependencies between the `.agda`-files are shown in the following graph:



We will now motivate some of the required lemmas and dependencies.

- In `Composition.agda`, we prove lemmas about the interaction of multiple substitutions. For example, it doesn't matter if we first apply a substitution to a term and then weaken it, or if we first weaken the term and then apply a substitution to it, that has been lifted with $\mathtt{ext_s}$ over the variable introduced by the weakening:

  ```
  ren wk (sub σ e) ≡ sub (ext_s σ) (ren wk e)
  ```

  The standard way of proving such interaction lemmas is by making use of the fact that substitutions and renamings can be composed:

  ```
  _r∘s_ : ∀ {n1 n2 n3} → Ren n2 n3 → Sub n1 n2 → Sub n1 n3
  (ρ1 r∘s σ2) x = ren ρ1 (σ2 x)
  ```

  If we have a renaming ρ_1 and a substitution σ_2, then their composition ρ_1 $\mathtt{_s\circ_r}$ σ_2 is a substitution, which first performs the substitution σ_2 and then the renaming ρ_1. This intuition is formalized by the corresponding fusion lemma:

  ```
  ren ρ1 (sub σ2 e) ≡ sub (ρ1 r∘s σ2) e
  ```

  With the help of the fusion lemmas, we can prove the above interaction lemma by proving that

  ```
  wk r∘s σ ≡ ext_s σ s∘r wk
  ```

  As substitutions are functions, we need to make use of the functional extensionality axiom (PLFA, chapter Isomorphisms).

- In `Confluence.agda`, we prove that our semantics is confluent, i.e. that if $t \hookrightarrow^* t_1$ and $t \hookrightarrow^* t_2$, then there exists some t', such that $t_1 \hookrightarrow^* t'$ and $t_2 \hookrightarrow^* t'$. The proof of confluence follows the *Confluence* chapter from the PLFA book very closely.

- In `ConversionProperties.agda`, we prove various properties about the convertibility relation `_≈_`. We prove that convertibility is an equivalence relation, i.e. it is reflexive, symmetric, and transitive, and how convertibility interacts with substitution.

  Applying a substitution to two convertible terms, yields again convertible terms:

  ```
  ≈-sub : ∀ {m n} {e e' : Term m} (σ : Sub m n)
    → e ≈ e'
    → sub σ e ≈ sub σ e'
  ```

Substituting two convertible terms into another term, yields again convertible terms:

```
≈σ-sub₁ : ∀ {n} {e : Term (suc n)} {e₁ e₂ : Term n}
  → e₁ ≈ e₂
  → e [ e₁ ] ≈ e [ e₂ ]
```

Typing is preserved along convertible types in the context:

```
≈-Γ-⊢₁ : ∀ {n} {Γ : Context n} {t₁ t₂ : Term n} {e t : Term (suc n)}
  → t₁ ≈ t₂
  → Γ , t₁ ⊢ e : t
  → Γ , t₂ ⊢ e : t
```

Each of these lemmas require multiple sub-lemmas, which are also part of `ConversionProperties.agda`.

- In `SubstitutionPreservesTyping.agda`, we prove the ⊢`sub` lemma. This file is closely related to `Part2/Preservation.agda`. As we are working with dependent types, the ⊢`sub` lemma looks slightly different:

```
⊢sub : ∀ {m n} {Γ₁ : Context m} {Γ₂ : Context n} {e t σ}
  → σ : Γ₁ ⇒ₛ Γ₂
  → Γ₁ ⊢ e : t
  ------------------------
  → Γ₂ ⊢ sub σ e : sub σ t
```

  In contrast to the simply typed lambda calculus, applying a substitution to a term, requires us to also apply the substitution to the type. This is because if e is an application of a dependent function to a variable, then e will also have the variable in its type. Consequently, if we replace the variable in e, we also need to replace the variable in the type of e.

- In `Progress.agda`, we prove the *progress* lemma, which is largely independent from the rest of the code base. The only external lemmas required is that _≈_ is an equivalence relation.

- In `Inversion.agda`, we prove so-called *inversion lemmas* for the typing relation. This is necessary, because the typing rule TConv makes our typing relation not syntax-directed. For example, in the simply typed lambda calculus, if we have a proof that Γ ⊢ 'λ e : t, then we can match on the proof, and find out, that this proof was constructed with the typing rule for lambda terms. In our dependently typed lambda calculus, the proof could have also been constructed with the TConv rule from some Γ ⊢ 'λ e : t' with t ≈ t'. But then how was the proof for Γ ⊢ 'λ e : t' constructed? Ultimately, the typing rule for lambda terms has to be used, but it could be under arbitrary many uses of the TConv rule. This fact is captured by the inversion lemma for lambda terms:

```
invert-⊢λ : ∀ {n} {Γ : Context n} {e : Term (suc n)}
                  {t₁ : Term n} {t₂ : Term (suc n)}
  → Γ ⊢ ‘λ e : ∀[x: t₁ ] t₂
  → ∃[ t₁’ ] ∃[ t₂’ ]
      t₁ ≈ t₁’ ×
      t₂ ≈ t₂’ ×
      Γ ⊢ t₁’ : ‘Set ×
      Γ , t₁’ ⊢ e : t₂’
```

Whereas the TLam and TConv rules allow us to build a typing for a lambda term from certain assumptions, the invert-⊢λ lemma takes a typing of a lambda term, and tells us from which assumptions it was built.

- In `SubjectReduction.agda`, we finally prove the subject-reduction lemma.

In some proofs, you need to make use of the `subst`-lemma from the PLFA chapter *Equality*:

```
subst : ∀ {A : Set} {x y : A} (P : A → Set) → x ≡ y → P x → P y
subst P refl px = px
```

The `subst` lemma allows you to apply equalities to types. For example, if you have a proof `p` of type `Γ ⊢ e : t`, but you need to produce a proof of type `Γ ⊢ e : t'`. If you also have a proof `q` of type `t ≡ t'`, then you can use `subst` to adjust the type of `p`:

```
p' : Γ ⊢ e : t'
p' = subst (λ T → Γ ⊢ e : T) q p
```

Your Tasks

- Prove the interaction lemmas for renamings and substitutions.
  File: `Exam/Part3/Composition.agda`; Points: 10; Difficulty: ⋆⋆
- Prove that the language satisfies the `confluence` property.
  File: `Exam/Part3/Confluence.agda`; Points: 10; Difficulty: ⋆⋆
- Prove the properties about the conversion relation.
  File: `Exam/Part3/ConversionProperties.agda`; Points: 5; Difficulty: ⋆⋆
- Prove that substitution preserves typing.
  File: `Exam/Part3/SubstitutionPreservesTyping.agda`; Points: 5; Difficulty: ⋆⋆
- Prove the inversion lemmas.
  File: `Exam/Part3/Inversion.agda`; Points: 5; Difficulty: ⋆⋆
- Prove that the language satisfies the `subject-reduction` property.
  File: `Exam/Part3/SubjectReduction.agda`; Points: 5; Difficulty: ⋆⋆
- Prove that the language satisfies the `progress` property.
  File: `Exam/Part3/Progress.agda`; Points: 5; Difficulty: ⋆⋆

Part 4: Extension with Dependent Pairs and Equality

In this part, we want to extend our dependently typed lambda calculus from Part 3 with dependent pair types and equality types.

Syntax

The syntax of our language is defined by the following grammar:

$$t ::= \ldots \mid (t, t) \mid \texttt{proj}_1\, t \mid \texttt{proj}_2\, t \mid \exists[x : t]\ t \mid \texttt{refl} \mid \texttt{subst}\, t\ t\ t \mid t \equiv t$$

A term t can be either a term as described in Part 3, a pair constructor (t_1, t_2), a projection $\texttt{proj}_i\, t$, a dependent pair type $\exists[x : t_1]\ t_2$, the `refl` constructor for equality, an application of the `subst` function to eliminate equalities, or an equality type $t_1 \equiv t_2$.

Dependent pairs behave exactly as we know them from Agda (PLFA book, quantifiers chapter):

```
data Σ (A : Set) (B : A → Set) : Set where
  ⟨_,_⟩ : (x : A) → B x → Σ A B

proj₁ : ∀ {A : Set} {B : A → Set} → Σ A B → A
proj₁ ⟨ a , b ⟩ = a

proj₂ : ∀ {A : Set} {B : A → Set} → (p : Σ A B) → B (proj₁ p)
proj₂ ⟨ a , b ⟩ = b

syntax Σ A (λ x → Bx) = Σ[ x ∈ A ] Bx
```

In our language, the term $\exists[x : t_1]\ t_2$ corresponds to the Agda type $\Sigma[x \in t_1]\ t_2$, which is just syntactic sugar for $\Sigma\ t_1\ (\lambda x \to t_2)$ due to the `syntax` definition.

Similarly, equality proofs behave exactly as we know them from Agda (PLFA book, equality chapter):

```
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x

subst : ∀ {A : Set} {x y : A} (P : A → Set) → x ≡ y → P x → P y
subst P refl px = px
```

As we do not have pattern matching in our language, we use the `subst` function instead to eliminate equality proofs. The `subst` function cannot prove everything that we can prove with pattern matching in Agda, but it is a good starting point. For example, we can use it to prove that equality is symmetric and transitive:

```
sym : ∀ {A : Set} {x y : A} → x ≡ y → y ≡ x
sym {A} {x} {y} x≡y = subst (λ z → z ≡ x) x≡y refl

trans : ∀ {A : Set} {x y z : A} → x ≡ y → y ≡ z → x ≡ z
trans {A} {x} {y} {z} x≡y y≡z = subst (λ z → x ≡ z) y≡z x≡y
```

Typing

We extend the typing relation from Part 3 by adding rules for our new terms. The new rules correspond very closely to the types of the Agda definitions presented in the previous subsubsection, which you can use to guide your intuition.

$$\textsc{TEx}\ \frac{\Gamma \vdash t_1 : \texttt{Set} \qquad \Gamma, x : t_1 \vdash t_2 : \texttt{Set}}{\Gamma \vdash \exists[x : t_1]\ t_2 : \texttt{Set}} \qquad \textsc{TPair}\ \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2[x \mapsto e_1]}{\Gamma \vdash (e_1, e_2) : \exists[x : t_1]\ t_2}$$

$$\textsc{TProj1}\ \frac{\Gamma \vdash e : \exists[x : t_1]\ t_2}{\Gamma \vdash \texttt{proj}_1\, e : t_1} \qquad \textsc{TProj2}\ \frac{\Gamma \vdash e : \exists[x : t_1]\ t_2}{\Gamma \vdash \texttt{proj}_2\, e : t_2[x \mapsto \texttt{proj}_1\, e]}$$

$$\textsc{TEq}\ \frac{\Gamma \vdash e_1 : t \qquad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 \equiv e_2 : \texttt{Set}} \qquad \textsc{TRefl}\ \frac{\Gamma \vdash e : t}{\Gamma \vdash \texttt{refl} : e \equiv e}$$

$$\textsc{TSubst}\ \frac{\Gamma \vdash u_1 : t' \qquad \Gamma \vdash u_2 : t' \qquad \Gamma, x : t' \vdash t : \texttt{Set} \qquad \Gamma \vdash e_1 : u_1 \equiv u_2 \qquad \Gamma \vdash e_2 : t[x \mapsto u_1]}{\Gamma \vdash \texttt{subst}\, t\ e_1\ e_2 : t[x \mapsto u_2]}$$

Your Tasks

- Copy every `.agda`-file from Part 3 to Part 4, except for `Specification.agda` and `Progress.agda`. In the copied files change the imports accordingly, e.g. `open import Exam.Part3.Inversion` should be renamed to `open import Exam.Part4.Inversion`. Now, Agda will complain that many lemmas have missing pattern matching cases. Your task is to add and prove the missing cases, such that we end up with a working proof of subject-reduction for the extended language. In some places, you may also need to introduce a few new lemmas.

  You do *not* need to prove the *progress* lemma.

  Important: Once you started Part 4, we will not consider the Part 3 directory during grading, but instead check if the cases from Part 3 are covered in Part 4. This is easier for both you and us, because if you still have holes in Part 3, you can also fill them after you started Part 4, without needing to synchronize both parts, and we only have to grade one of the two directories.

  Points: 15; Difficulty: ⋆⋆