Semester Project Report: Machine-checked typesafety proofs

Samuel Grütter   Supervisor: Nada Amin

EPFL, June 2014

Abstract

In this semester project, I learnt the proof assistants Twelf [5] and Coq [4], and how to use them to formalize proofs about programming languages. Then, I applied the acquired skills to translate the "transitivity pushing" proof [6], a proof needed for the DOT [1] formalization, from Twelf to Coq. The representation chosen in the Twelf proof is not general enough, whereas the representation chosen in the Coq proof is, so the contribution of this translation is to show that the proof still works in the more general representation. Additionally, one part of the proof could be simplified and made more understandable.

Contents

1 Overview

In the first part of the semester project, I learnt the proof assistants Twelf and Coq, and how to use them to formalize proofs about programming languages. The insights I gained can be presented as answers to the following two orthogonal questions:

- Independently of the proof assistant we're using, what's the best way to represent a programming language in a formal proof language? In particular, how can/should we represent language constructs which bind variables?
- What are the advantages and disadvantages of Twelf and Coq?

Section 2 treats the first question, and section 3 treats the second.

In the second part of the semester project, I applied the acquired skills to translate the "transitivity pushing" proof [6], a proof needed for the DOT [1] formalization, from Twelf to Coq. This part is treated in section 4.

The motivation for doing this translation is the following: In the Twelf proof, the reversed De Bruijn indices representation was chosen, but as explained in section 2.2, this representation is not general enough. So the question was whether the proof would still work in a general representation. To verify this, the locally nameless representation (section 2.3), which is general enough, was chosen, and, based on the experiences described in section 3, Coq was chosen over Twelf.

Referenced source code files

This report refers to several source code files by using the filenames shown in the left column of this table. The right column gives their complete URLs:

Name	URL
`RecordsLang.elf`	`https://github.com/samuelgruetter/typesafety-proofs-spring14/blob/master/RecordsLang/RecordsLang.elf`
`DotConcrete.elf`	`https://github.com/samuelgruetter/typesafety-proofs-spring14/blob/master/DotConcrete/DotConcrete.elf`
`DotTransitivity.v`	`https://github.com/samuelgruetter/typesafety-proofs-spring14/blob/master/DotTransitivity/DotTransitivity.v`
`AutoTypechecker.v`	`https://github.com/samuelgruetter/typesafety-proofs-spring14/blob/master/AutoTypechecker/AutoTypechecker.v`

Many parts of this report are intended to be read together with the source code of these files.

2 Choosing the representation

2.1 Concrete syntax

The obvious way of encoding a programming language in a proof language is to use concrete syntax, that is, variables are represented using identifiers. The identifiers can be represented as string names, or as natural numbers, or by any other set, provided that we have a simple way to decide equality.

Since I did not even know that there are other representations than concrete syntax, I started my semester project by formalizing in concrete syntax a simple records language, which is described in the next subsection.

2.1.1 Records language in concrete syntax in Twelf

To get started with Twelf, I implemented a simple language with records of immutable values and lambdas. Its complete definition and typesafety proof can be found in the file `RecordsLang.elf`. Here, we only print its grammar:

x, v	identifier	r ::=	runtime expression
t ::=	type	val	normal form
$\{\overline{v : t}\}$	record type	(st, u)	e being evaluated,
$t \to t$	function type		together with store
e ::=	general expression	val ::=	fully evaluated value
u	user expression	$(st, (v : t) \Rightarrow e)$	closure
r	runtime expression	$\{\overline{v = val}\}$	fully evaluated record
u ::=	user expression	st ::=	store
v	variable	$\overline{v = val}$	maps variables to vals
$\{\overline{v = e}\}$	record construction	Γ ::=	typechecking environment
$(v : t) \Rightarrow e$	anonymous function	$\overline{v : t}$	maps variables to types
$e(e)$	function application		
$e.v$	field selection		

We distinguish between "user expressions", which are the expressions that a program written by a user consists of, and "runtime expressions", which are introduced during the small-step evaluation of the program. A store st maps identifiers to fully evaluated values (val). Closures $(st, (v : t) \Rightarrow e)$ are used to capture the full store of the site where the function $(v : t) \Rightarrow e$ was defined. Pairing a store with a user expression gives the runtime expression (st, u), which stands for a subexpression that has to be evaluated in the context st instead of the outer context. During the small-step evaluation, the term being evaluated is always wrapped in an (st, u), no matter whether the context changes or not, to allow uniform treatment.

There are typing rules for values (called `vty`) and typing rules for expressions (called `ty`). The typing rules for values don't need an environment, because values cannot depend on other values.

The evaluation rules are in small-step style and the corresponding relation is called `step`. Before evaluating a subexpression, it is always wrapped in a (st, u). This has two purposes: First, it tells us in which environment to evaluate it, and second, it is used as a marker to indicate that we've already started evaluating the subexpression. This is important for the evaluation of record initializations, which have to be evaluated in order. Once all fields of a record initialization are evaluated, the partial function `vis>rvs` succeeds, and the `step/vis/done` rule can be applied, which converts the record initialization from type e to type val.

In many places, we use the `(st-eq-g S G)` judgment, which says that the environment `G` binds exactly the same identifiers as the store `S`, that they appear in the same order, and that the types of the values in `S` correspond to the types in `G`.

We prove typesafety for this language by proving progress and preservation. First, we prove progress:

```
progress-thm:     % in:  S, U, and (U types in S)
                  st-eq-g S G -> ty G (e/u U) T
                  % out: there's a step
                  -> step S U R -> type.
```

It states that if we have an expression `U`, which types in a store `S`, then it can always take a step. Note that we needn't have a conclusion of the form "can take a step or is a value", because values are syntactically distinguished from expressions.

To prove preservation, we need a special case of a weakening lemma, called `weaken-ty-gnil`, which states that if a typing judgment holds in the empty environment, then it also holds in any environment. To prove it, we first prove a lemma called `weaken-ty-1`, which states that if a typing judgment holds in an environment `G`, it also holds if we add an arbitrary binding to the *beginning* of the environment, i.e. if the binding we add has the same variable name as one already in the environment, the one already in the environment shadows the added binding. This is exactly the opposite of standard weakening lemmas, and easier to prove, but sufficient for this simple language.

The preservation theorem states that if an expression `U` types in a store `S` and steps to the runtime-expression `R`, then `R` still typechecks. Note that we can demand that it typechecks in the empty environment, because runtime expressions are either values or of the form (st, u), i.e. they wrap the store they need to typecheck.

```
preservation-thm: % in: U types and can step
                  st-eq-g S G -> ty G (e/u U) T -> step S U R
                  % out: what it steps to still types
                  -> ty gnil (e/r R) T -> type.
```

2.1.2 DOT in concrete syntax in Twelf

Given that it was not too hard to formalize the simple records language in Twelf and prove it typesafe, it seemed that this should also be possible for DOT. So I formalized the specification of full DOT in the file `DotConcrete.elf`, following the rules in [1], but to avoid list handling, types with several declarations were expressed as intersection types, as one can see in the grammar for types:

$t ::=$	type	id	identifier/label
$\top$	top type	$v ::=$	value
$\bot$	bottom type	id	bound variable
$\{\textbf{type } id : t..t\}$	type with one type member	o	reference to store
$\{\textbf{val } id : t\}$	type with one value member	$p ::=$	path
$\{\textbf{def } id : t \to t\}$	type with one method member	v	value
$p.id$	path type	$p.id$	field selection on path
$t \wedge t$	intersection type		
$t \vee t$	union type		
$\{id \Rightarrow t\}$	bind type (give self reference)		

So, a refinement type with several declarations, which in [1] would be written as

$$T\{z \Rightarrow L_1 : S_1..U_1, ..., L_k : S_k..U_k, l_1 : T_1, ..., l_m : T_m, m_1 : V_1 \to W_1, ..., m_n : V_n \to W_n\}$$

would be represented as

$$\begin{aligned}\{z \Rightarrow \{T &\wedge \{\textbf{type } L_1 : S_1..U_1\} \wedge ... \wedge \{\textbf{type } L_k : S_k..U_k\}\\ &\wedge \{\textbf{val } l_1 : T_1\} \wedge ... \wedge \{\textbf{val } l_k : T_k\}\\ &\wedge \{\textbf{def } m_1 : V_1 \to W_1\} \wedge ... \wedge \{\textbf{def } m_n : V_m \to W_m\}\}\}\end{aligned}$$

An advantage of this approach is that we don't need the concept of *expansion*: In [1], expansion is used in the following situations, and for each of them, there's a way of representing the same without needing expansion:

- When constructing a **new** $T_c\{\overline{d}\}$: We write (**cast** T_c (**new** $\{\overline{d}\}$)) instead, **cast** being part of the syntactic sugar defined in [1]. It's also possible to write just **new** $\{\overline{d}\}$, then the type is calculated from the types of the initializations in $\overline{d}$.

- For subtyping of refinement types: Since we can represent refinement types using bind types and intersection types, we don't need expansion for this.

- For membership judgments: [1] has the $\Gamma \vdash t \ni D$ judgment which says that a term t contains declaration D as member. Instead, we have a $\Gamma, s \vdash T \ni \{...\}$ judgment (called `mem` in `DotConcrete.elf`), which says that *type* T has as a **type**, **val**, or **def** as member. To decide whether a term t contains a declaration, we first assign it a type T and then check membership of the declaration on that T. As one can see in `DotConcrete.elf`, that way it's possible to define membership without depending on expansion.

The reduction rules, typing rules and subtyping rules mostly follow [1]. Since they require substitution and fresh name generation, they are quite verbose: Without the arithmetics part and the examples, the whole specification of DOT takes 570 lines of Twelf code.

Attempting to prove progress soon ended up in an explosion of boilerplate code, because uniqueness and totality proofs for the substitution and fresh name generation functions were required. Even though conceptually these are functions, and thus implicitly unique and total, we have to prove this in Twelf, because Twelf does not have functions which are powerful enough to express substitution and fresh name generation, so we have to define them as relations, and relations don't have built-in uniqueness and totality.

Moreover, it was foreseeable that many proofs of the form "if predicate P holds for a term t, and t is alpha-equivalent to t', then P also holds for t'" whould be needed, which would again require very verbose code.

Finally, I concluded that it was not worth pursuing this approach any further.

2.2 Reversed De Bruijn indices

Another approach to represent bound variables is one that we could call "Reversed de Bruijn indices". In general, code written in this representation is much less verbose, and fewer infrastructure lemmas are needed.

Most Twelf developments done at LAMP so far used this representation. The goal of this section is to present this representation and then to show why it is not general enough.

Contrary to ordinary de Bruijn indices [3], where bound variables are replaced by a natural number indicating the distance from the variable occurrence to the place where the variable is bound, the "Reversed de Bruijn indices" approach replaces each variable by a natural number which indicates at which depth the variable was bound, i.e. each variable z is replaced by a natural i such that i is the number of binders we encounter when going from the root to the place where z is bound. For example, the type

```
type T1 = { z =>
  type A <: { a => type E; val e: a.E }
  type T <: { y =>
    val a: z.A
  }
}
```

would be represented as

```
type T1 = { (0) =>
  type A <: { (1) => type E; val e: (1).E }
  type T <: { (1) =>
    val a: (0).A
  }
}
```

Environments, which are usually mappings from variable names to types, can simply be represented as lists of types, where the i-th type in the list is the type of the variable with index i. When typechecking or subtype checking enters a scope which binds a variable to a type, all we have to do is to append this type to the environment.

This representation is very convenient to deal with in Twelf, but, as we shall see, severely restricts the expressiveness of the language under study.

Let us define a type `T2` as follows:

```
type T2 = { (0) =>
  type T <: { (1) =>
    val a: { (2) => type E; val e: (2).E }
  }
}
```

It's the same as `T1`, except that the definition of `A` was "inlined". Clearly, `T1` should be a subtype of `T2` (this was also verified with `scalac` 2.10.3). For `val a`, this subtype check needs to check that in the environment `{T1, {(1)` $\Rightarrow$ `val a: (0).A }}` [1], the type `(0).A` is a subtype of `{(2)` $\Rightarrow$ `type E; val e: (2).E}`, and after looking up `(0).A` among the members of `T1`, that

```
{(1) => type E; val e: (1).E} <: {(2) => type E; val e: (2).E}
```

Now we need to check that the declarations on the left-hand side are sub-declarations of the declarations on the right-hand side, and to do so, we need to put a binding for the self-reference into the environment. Since the self-reference does not have the same index on both sides, the Twelf developments resort to using one environment for each side in subtype checks, and to prove a statement of the form $\Gamma_1 \vdash$ `{(i)` $\Rightarrow \ldots$`}` $<:$ `{(j)` $\Rightarrow \ldots$`}` $\dashv \Gamma_2$, Γ_1 is shrunk to size i (i.e. we only take its elements from 0 to $i-1$), and Γ_2 is shrunk to size j, before adding the bindings for the self-reference. This shrinking is safe because a type defined at depth i cannot refer to variables which are bound at a deeper level.

So, in our example, for `val e` we would need to check that

$\Gamma_1 \vdash$ `(1).E` $<:$`(2).E` $\dashv \Gamma_2$

where

$\Gamma_1 =$ `{T1, {(1)` $\Rightarrow$`type E; val e: (1).E}}`
$\Gamma_2 =$ `{`
```
  T1,
  {(1) ⇒ val a: {(2) ⇒ type E; val e: (2).E }},
  {(2) ⇒ type E; val e: (2).E}
}
```

To prove this, we would need to apply reflexivity, but we lost the information that `(1)` and `(2)` are the same path, so we cannot prove this subtype relationship.

To work around this restriction, we can allow that every self reference variable can be replaced by an arbitrary number, but before typechecking or subtype checking enters a bind type whose self reference is represented by number i, the environment is shrunk to size i, or extended with dummy entries to size i, and we require that all terms are written such that this environment shrinking does not discard bindings that we need to typecheck the terms.

With this work-around, we could rewrite `T2` from the previous example as follows:

```
type T2 = { (0) =>
  type T <: { (1) =>
    val a: { (1) => type E; val e: (1).E }
  }
}
```

and then, the comparison that did not work before would work now:

```
{(1) => type E; val e: (1).E} <: {(1) => type E; val e: (1).E}
```

However, we can also find an example which does not work even with the work-around:

[1] `T1` is just used as a shorthand, in reality, the right-hand side of `T1`'s definition would be put into the environment.

```
type T1 = { z =>
  type A <: { a1 =>
    type E = Any
    type F
    val f: a1.F
    val e: Any
    val b: z.B
  }

  type B <: { b1 =>
    type E = Any
    type F
    val f: b1.F
    val e: Any
    val a: z.A
  }
}
```

```
type T2 = { z =>
  type A <: { a2 =>
    type E = Any
    type F
    val f: a2.F
    val b: { b3 =>
      type F; val f: b3.F; val e: a2.E; val a: z.A
    }
  }
  type B <: { b2 =>
    type E = Any
    type F
    val f: b2.F
    val a: { a3 =>
      type F; val f: a3.F; val e: b2.E; val b: z.B
    }
  }
}
```

One can verify that `T1` is a subtype of `T2` (and `scalac` 2.10.3 confirms this). Now let us try to replace the self references by numbers, such that the subtyping relationship can be shown.

For `val f` in `T1#A` and `T2#A`, we must be able to show by reflexivity that `a1.F` <: `a2.F`, so we must have `a1 = a2`, and symmetrically `b1 = b2`. Further, since in `T2` the declarations in the type of `val b` refer to members of `a2`, we must have `b3 > a2`, because otherwise environment shrinking would discard the binding for `a2`. Symmetrically, we must have `a3 > b2`. Moreover, to be able to compare the type of `val f` in the type of `val b` in `T1#A` to the type of `val f` in the type of `val b` in `T2#A`, we must have that `b1 = b3`, and symmetrically, that `a1 = a3`. Putting all these (in)equalities together, we get

```
a1 = a2 < b3 = b1 = b2 < a3 = a1
```

which is a contradiction. This means that there are examples which typecheck if we check them according to the rules, but don't typecheck anymore if we represent them with reversed De Bruijn indices. So, this representation is not adequate, and we have to look for a better representation.

2.3 The locally nameless representation

The *locally nameless* representation [2] tries to address the problems we saw in section 2.1. The key idea is to represent bound variables and free variables differently: Bound variables are represented with a De Bruijn index, and free variables are represented with an identifier (which could be a string, a natural number, ...). The advantage of this approach is that closed terms have a unique representation, so we're always working up to alpha-equivalence. The concept of "opening" is used to turn bound variables into free variables. We'll illustrate it by the example of the bind type from DOT, which binds a self reference: When subtype checking enters a bind type, a mapping for the bound variable has to be put into the environment, and the bound variable becomes a free variable. In `DotTransitivity.v`, the signature of the opening function for types is as follows:

```
open_rec_typ (k: nat) (u: var) (t: typ) : typ
```

It takes a natural number `k`, the index of the bound variable to replace, and an identifier `u`, the name of the free variable to be substituted for `k`, and a type `t`, in which the substitution has to be carried out. To open a bind type, we use `open_rec_typ` with $k = 0$, so we define `open_typ u t` as `open_rec_typ 0 u t`. Analogously, we define `open_rec_dec` and `open_dec` to open a declaration.

Now we can write the subtyping rule for bind types as follows:

```
forall (G: ctx) (ds1: decs) (ds2: decs),
  (forall z, ok (G & (z ~ (typ_bind ds1))) ->
        subdecs (G & (z ~ (typ_bind ds1)))
                (map (open_dec z) ds1)
                (map (open_dec z) ds2)) ->
  subtyp G (typ_bind ds1) (typ_bind ds2)
```

Here, `&` is environment concatenation, `z ~ t` creates a singleton environment mapping variable `z` to type `t`, and `ok E` means that in environment `E`, no duplicate variables occur (this makes the proofs simpler without restricting the language's expressiveness). To check if `ds1` are sub-declarations of `ds2`, we open them with the variable `z`, for which we put a mapping into the environment. [2]

The locally nameless representation has many advantages:

- It's quite close to the way proofs are presented on paper. In particular, we have an explicit environment (which is not the case in higher order abstract syntax approaches, for instance).
- Each alpha-equivalence class of binding terms/types has a unique representation, so there's no need for lemmas of the form "if predicate P holds for a term t, and t is alpha-equivalent to t', then P also holds for t'".
- Since bound and free variables are represented differently, substitution can be defined easily, without any variable capture issues.

3 Choosing the tool

An orthogonal question to the question which representation to choose is the question which proof assistant to choose. In this semester project, I learnt to use Twelf [5] and Coq [4], and in the following subsections, I'll describe a few desirable properties of proof assistants, and discuss which of these properties Twelf and Coq have or don't have.

This list of desirable properties is not meant to be exhaustive. Rather, I'll focus on some which were important in this semester project.

3.1 Automatic typechecking of sample terms

Each machine-checked proof contains a certain number of "trusted definitions" [3], that is, definitions needed to state the theorems to prove. In the case of typesafety proofs, these are the specification of the language under study. As their name suggests, readers of the proof have to trust these definitions in that they contain no mistakes. For instance, a worst-case scenario would be that the typing rules of a typesafety proof are contradictory, so that assuming that a term typechecks would be a contradiction, and from this contradiction, everything (including typesafety) could be proved.

[2]Note that we could also put an opened version of `(typ_bind ds1)` into the environment, as it's done in `DotTransitivity.v`.

[3]We use the term the same way as [2].

To address this problem, it's helpful to write some sample terms in the language being formalized and to use the proof assistant to prove that these terms typecheck according to the given typechecking rules and that they reduce to the expected normal form according to the given evaluation rules. This provides some empirical evidence that the language specification expresses what we meant.

3.1.1 Twelf

For this, Twelf is very powerful thanks to its `%solve` command: We can give it a type (which, by the Curry-Howard correspondence, can for instance correspond to the judgment "this sample term typechecks"), and Twelf automatically searches for a proof term having this type.

3.1.2 Coq

With Coq, we don't get this "for free": For each sample term we want to test, we have to prove a lemma stating that it typechecks and reduces. However, Coq offers powerful automation techniques, which can be used to write a typechecker which constructs a proof object for typable terms and gives a precise error message in for untypable expressions.

An example for such a typechecker can be found in the file `AutoTypechecker.v`, which implements the simple language described in [7]. Once the typechecker is written, checking that an expression named `e_test` has type `tyNat` is as easy as this (`tc` is the name of the typechecker tactic):

```
Fact e_test_types: Types nil e_test tyNat.
Proof.
  unfold e_test.
  tc.
Qed.
```

Moreover, the typechecker can be implemented such that it gives precise error messages in case of untypable expressions. For instance, if one attempts to typecheck the expression `(expApp ( expConst (constNat 1)) (expConst (constNat 1)))` with `tc`, it produces the error message "Error: Tactic failure: cannot apply `(expConst (constNat 1))` of type `tyNat` to argument `( expConst (constNat 1))` of type `tyNat`".

In this simple language, typechecking can be done without any backtracking. This is not the case for DOT, where we have a subsumption rule, which might need to be applied at any time. Doing backtracking and provide precise error messages for DOT typechecking might be quite challenging, because in case of failure, one would have to know whether this is a definite failure which needs to be reported, or just a failure indicating that we've taken a wrong branch somewhere. But there are two ways to make the goal simpler, such that it should be feasible:

- Accept a typechecker which does not return error messages in case of untypable expressions.
- Accept a typechecker which only applies the subsumption rule in order to upcast the argument given to a function. Such a typechecker would be able to typecheck most "normal" examples, and only fail on special cases which arise typically after performing some reduction steps on a term. An example [4] for such a situation is

```
val o' = new { m: Top -> R' } ( m(x) = o'.m(x) /* infinite loop */ )
val o  = new { v: { m: Top -> R} } ( v = o')
o.v.m(o).a.l
```

[4]given by N. Amin in a different context

where

```
R  = { z ⇒ X: Top .. { val l: Top }; a : z.X }
R' = { z ⇒ X: Top .. { val l: Top }; a : Top }
```

Here, `o.v.m(o).a.l` typechecks, and would also be accepted by a simplified typechecker which only applies subsumption to upcast the argument given to a function, but it steps to `o'.m(o).a.l`, which still typechecks, but would not be accepted by such a simplified typechecker, because we have to apply subsumption in order to get from `o' : { m: Top -> R' }` to `o' : { m: Top -> R }`.

3.1.3 Comparison

If the language specification and the sample terms we want to test are correct, then typechecking sample terms is definitely much simpler in Twelf. However, if there are mistakes, debugging them in Twelf is certainly possible, but it's a bit easier in Coq: If the mistakes are in the typechecking rules, doing an interactive proof in Coq can quickly reveal the problem, and if the mistakes are in the sample terms, the automatic typechecker (if one has written it...) can give precise error messages.

3.2 Termination checking of mutually inductive proofs

In typesafety proofs, many lemmas are proven by structural induction on a proof object, and proof objects often have mutually inductive types. For instance, DOT has a `subtyp` judgment, a `subdec` judgment and a `subdecs` judgment, which are defined in terms of each other.

In this section, we will discuss proofs by induction on such mutually inductive types.

3.2.1 Induction in Coq

There are three ways to write proofs by induction in Coq:

1. Using the primitive tactic `fix`.
2. Using an induction scheme generated by Coq.
3. Using a custom induction scheme, which has to be proven manually.

Let's discuss each of them:

1. The `fix` tactic is the most powerful and flexible. The user has to specify on which argument of the proof the induction will be done, and gets the theorem to prove as a hypothesis to prove the theorem. When there are no more subgoals in the proof, and the user writes `Qed`, Coq invokes its termination checker to verify that the proof uses the theorem being proved only on structurally smaller values.

The problem with this tactic is that the termination checker can require a very long time to do the checking, because it unfolds all definitions it encounters. For instance, to check the weakening and narrowing proofs for `subtyp`/`subdec`/`subdecs`, it took several minutes.

2. For all inductive types, Coq automatically generates an induction scheme called `T_ind`, where `T` is replaced by the name of the type being defined. The induction scheme for type `T` says that if for each branch of `T`, supposing that a predicate `P` holds for all constructor arguments of type `T` of the branch implies that `P` also holds for the branch, then `P` holds for all instances of type `T`. [5] For

[5]The reader is encouraged to run the command `Check subtyp_ind` in the file `DotTransitivity.v`, and to inspect the formal statement of this induction scheme, which is too long to be printed here.

most proofs, the `subtyp_ind` induction scheme cannot be used, because it only gives us induction hypotheses for wrapped instances of type `subtyp`, but the `subtyp_bind` branch wraps a `subdecs` instance, for which we are given no induction hypothesis.

To solve this problem, we can ask Coq to generate a mutually recursive induction scheme:

```
Scheme subtyp_mut  := Induction for subtyp  Sort Prop
with   subdec_mut  := Induction for subdec  Sort Prop
with   subdecs_mut := Induction for subdecs Sort Prop.
```

As we can see by running `Check subtyp_mut`, the `subtyp_bind` branch now gets an induction hypothesis for the `subdecs` instance that it wraps.

But there remains another problem: If we define `subdecs` in the following (high-level) way,

```
... with subdecs : mode -> ctx -> decs -> decs -> Prop :=
  | subdecs_def : forall m G ds1 ds2,
      (forall l d2, binds l d2 ds2 ->
         (exists d1, binds l d1 ds1 /\ subdec m G d1 d2)) ->
      subdecs m G ds1 ds2.
```

then the generated `subdecs_mut` induction scheme has no induction hypothesis for the `subdec` instance that it wraps, because it's hidden behind an existential quantification, and the automatic induction scheme generator cannot handle this.

The solution to this is to define `subdecs` without using existential quantification:

```
... with subdecs : mode -> ctx -> decs -> decs -> Prop :=
  | subdecs_empty : forall m G ds,
      subdecs m G ds empty
  | subdecs_push : forall m G l ds1 ds2 d1 d2,
      binds l d1 ds1 ->
      subdec m G d1 d2 ->
      subdecs m G ds1 ds2 ->
      subdecs m G ds1 (ds2 & l ~ d2).
```

Then, we can create a "combined scheme" using

```
Combined Scheme subtyp_subdec_subdecs_mutind
   from subtyp_mut, subdec_mut, subdecs_mut.
```

whose conclusion is the conjunction of the conclusions of the `subtyp_mut`, `subdec_mut`, and `subdecs_mut` schemes.

Using this "combined scheme", we can write the weakening and narrowing proofs for `subtyp`/`subdec`/`subdecs`, and Coq can check them within less than a second, which is considerably faster than the solution using `fix`, which took several minutes.

3. Proving custom induction schemes could be a solution to get an induction scheme for `subtyp`/`subdec`/`subdecs` where `subdecs` is defined using existential quantification. However, writing down these induction schemes explicitly takes a lot of space and leads to code which is hard to maintain, because each change in the definition of `subtyp`/`subdec`/`subdecs` would require an update in the induction scheme.

3.2.2 Two kinds of proofs

Now that we have seen how induction can be done in Coq, let us distinguish between two kinds of proofs by induction on mutually inductive types:

A) Proofs which consist of one statement per inductive type. In our example, weakening would be such a proof, because it consists of the three statements

- "If a `subtyp` derivation holds in the environment `G1 & G3`, then it also holds in the environment `G1 & G2 & G3`".
- "If a `subdec` derivation holds in the environment `G1 & G3`, then it also holds in the environment `G1 & G2 & G3`".
- "If a `subdecs` derivation holds in the environment `G1 & G3`, then it also holds in the environment `G1 & G2 & G3`".

B) Proofs for which we cannot give a statement for each of the inductive types. The lemma `prepend_chain` in the "transitivity pushing" proof (described in section 4) is an example for this. It uses a data structure called `chain`, which can be thought of as a list of subtyping judgments. We say that a chain "goes from type `A` to type `D`" if the left hand side type of the first subtyping judgment in the list is `A` and the right hand side type of the last subtyping judgment in the list is `D`. The lemma `prepend_chain` says the following:

- "If we are given a subtyping derivation for `A1` <: `A2`, and a chain which goes from `A2` to `D`, then we can construct a chain which goes from `A1` to `D`".

Note that in this case, it's impossible to add a similar judgment for `subdec` and `subdecs`, because `chain` is a list of `subtyp` derivations, so we cannot prepend `subdec` or `subdecs` judgments to it.

For proofs of type A), we can use an auto-generated induction scheme, but for proofs of type B), we cannot, because to apply an auto-generated induction scheme, we need to give a statement we want to prove for each inductive type.

3.2.3 Induction in Twelf

In Twelf, proofs by induction on such mutually inductive types are quite easy to write: We have to indicate the argument on which we want to decrease, and when processing the `%total` command, Twelf's termination checker verifies that to prove a theorem, we only invoke other theorems being proved with a structurally smaller argument. This is very similar to the `fix` tactic of Coq, but since Twelf is a less rich language, Twelf's termination checking tends to be faster than Coq's.

3.2.4 Comparison

It seems that proofs by induction on mutually inductive types work "just out of the box" in Twelf, but confront the user with some issues in Coq. These can be solved, but solving them requires some effort. We could summarize them as follows:

- Termination checking for proofs using the `fix` tactic can take a very long time, and there are situations (type B proofs) where we're forced to use `fix` (however, all proofs made in this project where we're forced to use `fix` can be checked within a few seconds).
- Coq cannot always extract the appropriate induction scheme, and fixing it manually leads to verbose and hard to maintain code, or it requires to renounce existential quantification in the definition of the inductive types.

3.3 Expressiveness of predicates

Coq's logical predicates are much more powerful than Twelf's. For instance, we can express the predicate that the declarations `ds1` are subdeclarations of the declarations `ds2` as follows:

```
forall l d2,
  binds l d2 ds2 -> (exists d1, binds l d1 ds1 /\ subdec G d1 d2)
```

Here, `binds l d ds` means that the declaration `(l: d)` belongs to the set of declarations `ds`.

Twelf, however, can only express such forall-exists statements as theorems (which must be proven to be always true), but not as a predicate which could be used as a hypothesis for another theorem.

Of course this does not mean that Twelf cannot express the predicate that the declarations `ds1` are subdeclarations of `ds2`, because we can also define this predicate by induction on `ds2`, for instance like

```
subdecs/nil  : subdecs G DS dnil.
subdecs/cons : subdecs G (dcons D1 DS1) (dcons D2 DS2)
               <- subdec G D1 D2
               <- subdecs G DS1 DS2.
```

Note that this representation imposes a restriction on the ordering of the declarations, which might be undesirable, but it should be possible to fix that.

One could argue that the forall-exists way of expressing the subdeclarations predicate is more intuitive and easier to understand, and thus prefer Coq over Twelf.

3.4 The concept of total functions

In order to express the concept of substitution and opening (see section 2.3), we need to define functions for them. Many parts of typesafety proofs depend on the fact that substitution and opening are indeed functions, i.e. they depend on the functions'

- Totality: The function returns a result for all inputs.
- Uniqueness: For a given input, there's only one possible output.

Twelf's functions are on the term level and cannot be recursive, so they are not powerful enough to express substitution and opening. Instead, we have to give inference rules for a relation which specifies the function we want to express. But since we define a relation, we need to prove its totality and uniqueness explicitly. Since Twelf has no automation, these proofs are *very* long and tedious, and distract from the interesting parts.

Coq is much easier to use here: It has `Fixpoint` definitions, which allow us to define (possibly mutually recursive) functions, and since they're defined as functions, we get their totality and uniqueness for free.

3.5 The concept of equality

Coq has a built-in concept of Leibniz equality, that is, two terms x and y are considered equal if and only if every property which is true for x is also true for y. Since Coq also supports negation, we can easily express that two terms are not equal. Using the tactic `discriminate H`, where `H` is a an equality proof which states equality for two structurally different terms, we can solve any goal by contradiction, just writing this single line of code.

None of this is possible with Twelf. Creating a judgment which states that two terms of a given type are not equal is especially tedious, because if the type has n branches, we need to add $n(n-1)$

cases to define non-equality. And for equality, we would also have to prove transitivity manually, because Twelf has no built-in concept of equality. Also note that we'd have to do this for each type separately, because Twelf has no polymorphism.

3.6 Automation

No comparison between Twelf and Coq should forget to mention that Coq offers powerful automation techniques. Altough they were not (yet) used extensively in this project, they seem to make Coq developments more scalable, and they will definitely help with bigger proofs.

Twelf, on the contrary, has no support for automation.

3.7 Conclusion

To summarize this comparison, let's state the "winner" for each of the desirable properties:

Property	"Winner"
Automatic typechecking of sample terms	Twelf
Termination checking of mutually inductive proofs	Twelf
Expressiveness of predicates	Coq
The concept of total functions	Coq
The concept of equality	Coq
Automation	Coq

Based on the experiences made in this project, I tend to prefer Coq, mainly because it's more expressive and less boilerplate-prone.

4 DOT subtyping transitivity in Coq

4.1 The problem

In the typesafety proof for DOT, we need a subtyping transitivity lemma which states that if $\Gamma \vdash T_1 <: T_2$ and $\Gamma \vdash T_2 <: T_3$, then $\Gamma \vdash T_1 <: T_3$. There is no explicit transitivity rule in the subtyping rules, but they are defined in such a way that they imply transitivity, but this has to be shown.

4.1.1 Why we need subtyping transitivity

In the typesafety proof for DOT, we need a so-called *inversion lemma* for each typing rule. These lemmas say that given a typing derivation, we can extract the hypotheses which were needed to construct the typing derivation. For instance (if there was no subsumption rule), the inversion lemma for the typing rule for variables would state that if we have a derivation for $\Gamma \vdash x : T$, where x is a variable, then we can get a derivation for $(x : T) \in \Gamma$. The proofs of these lemmas would be trivial if there was no subsumption rule, but if there is, we have to slightly change the statement of the inversion lemmas, because any typing derivation might be followed by the application of the subsumption rule. For instance, the inversion lemma for the typing rule for variables would now state that if we have a derivation for $\Gamma \vdash x : T$, where x is a variable, then there exists a T' such that $(x : T') \in \Gamma$ and T' is a subtype of T. Proving these inversion lemmas requires a lemma which states that the subtyping relation is transitive, because suppose we're proving the inversion lemma for variables by induction on the size of the $\Gamma \vdash x : T$ derivation that we get, then if we get a derivation whose last rule is subsumption using a $\Gamma \vdash x : T'$ derivation and a $\Gamma \vdash T' <: T$ derivation, we can apply the induction hypothesis on the $\Gamma \vdash x : T'$ derivation to get that there's a T'' such that $(x : T'') \in \Gamma$ and $\Gamma \vdash T'' <: T'$, so we need the transitivity lemma to turn $\Gamma \vdash T'' <: T'$ and $\Gamma \vdash T' <: T$ into $\Gamma \vdash T'' <: T$.

4.1.2 Why an explicit transitivity rule doesn't help

One might suggest that we just add an explicit transitivity rule to the subtyping rules. However, this causes problems in the proofs of the subtyping inversion lemmas. For instance, consider the subtyping inversion lemma for bind types: It states that if a bind type T_1 whose declaration set is $\overline{D_1}$ is a subtype of a bind type T_2 whose declaration set is $\overline{D_2}$, then $\overline{D_1}$ are subdeclarations of $\overline{D_2}$.

If there's no explicit transitivity rule, this is trivial to prove, because there's only one rule which can prove that one bind type is a subtype of another, and this rule's premise is that $\overline{D_1}$ are subdeclarations of $\overline{D_2}$.

However, if we have an explicit transitivity rule, this subtyping inversion cannot be proven any more, because the last rule in the derivation of $T_1 <: T_2$ that we get might be the transitivity rule, so it might happen that all we can extract is $T_1 <: p.L$ and $p.L <: T_2$ for some path type $p.L$, and it's not clear how to use this to prove that $\overline{D_1}$ are subdeclarations of $\overline{D_2}$.

4.1.3 Dependencies

If we don't have a an explicit transitivity rule, we need to prove transitivity, and such a proof depends on the narrowing lemma, which states that if a statement about subtyping holds in the environment $(\Gamma, z : T_2)$ and T_1 is a subtype of T_2, then the statement also holds in the environment $(\Gamma, z : T_1)$. Conversely, the proof of the narrowing lemma also depends on the transitivity lemma, as we shall see in the two following paragraphs, where we informally sketch the relevant parts of the proofs which show the dependency:

4.1.4 Transitivity depends on narrowing

The transitivity proof would have to perform a case distinction on what the types T_1, T_2 and T_3 are. Let's consider the case where T_1, T_2 and T_3 all are bind types, let $\overline{D_1}$, $\overline{D_2}$ and $\overline{D_3}$ be their declaration sets, and let's call the self reference z. In this case, we are given proofs for $\Gamma, z : T_1 \vdash \overline{D_1} <: \overline{D_2}$, and for $\Gamma, z : T_2 \vdash \overline{D_2} <: \overline{D_3}$, and we have to show that $\Gamma, z : T_1 \vdash \overline{D_1} <: \overline{D_3}$. If we do our proof by induction, we have an induction hypothesis which states (directly or indirectly, depending on how the proof is set up) that $\Gamma, z : T_1 \vdash \overline{D_1} <: \overline{D_2} \wedge \Gamma, z : T_1 \vdash \overline{D_2} <: \overline{D_3} \Longrightarrow \Gamma, z : T_1 \vdash \overline{D_1} <: \overline{D_3}$. So, in order to apply it, we need to convert our $\Gamma, z : T_2 \vdash \overline{D_2} <: \overline{D_3}$ into a $\Gamma, z : T_1 \vdash \overline{D_2} <: \overline{D_3}$. Since we are given $\Gamma \vdash T_1 <: T_2$, we can (and have to) use the narrowing lemma for this.

4.1.5 Narrowing depends on transitivity

Let's consider the case where we are given a derivation for $\Gamma, z : T_2 \vdash p.L <: T$ and we have to show that $\Gamma, z : T_1 \vdash p.L <: T$, where T_1 is a subtype of T_2. We can do this as follows: We can decompose $\Gamma, z : T_2 \vdash p.L <: T$ into a declaration membership statement $\Gamma, z : T_2 \vdash p \ni (L : S..U)$ and a subtyping judgment $\Gamma, z : T_2 \vdash U <: T$. Since these are smaller derivations, we can use our induction hypothesis to narrow these two statements into $\Gamma, z : T_1 \vdash p \ni (L : S'..U')$ and $\Gamma, z : T_1 \vdash U <: T$. Notice that the membership statement might get more precise through narrowing, so $(L : S..U)$ becomes $(L : S'..U')$, and the narrowing we invoked also provides a proof that $(L : S'..U')$ is a subdeclaration of $(L : S..U)$, from which we can extract $\Gamma, z : T_1 \vdash U' <: U$.

Now, to construct a proof of $\Gamma, z : T_1 \vdash p.L <: T$, we need to provide $\Gamma, z : T_1 \vdash p \ni (L : S'..U')$ (which we have) and $\Gamma, z : T_1 \vdash U' <: T$, which we can construct by combining $\Gamma, z : T_1 \vdash U' <: U$ and $\Gamma, z : T_1 \vdash U <: T$, but this requires transitivity.

4.1.6 Proof by mutual induction

The obvious solution to this mutual dependency would be to prove these two lemmas by mutual induction. However, it is not clear what we could use as termination measure. Let's consider two of them and why they didn't work: [6]

- We cannot use the size of T_2 in the transitivity proof, because if T_2 is a path type, we need to prove $T_1 <: T_3$ from $(T_1 <: p.L) \wedge (p.L <: T_3)$, so we replace $p.L$ by its bounds $S..U$ to get $(T_1 <: S) \wedge (S <: U) \wedge (U <: T_3)$ and then we want to apply the induction hypothesis on these, but we cannot do so, because S and U are not wrapped in $p.L$, so they're not necessarily smaller, and trying to define a different size measure for types to solve this gave no results.
- We cannot use the size of the $T_1 <: T_2$ derivation and the $T_2 <: T_3$ derivation, because when the transitivity proof invokes narrowing, there's no guarantee that narrowing does not increase the size of the proof.

Of course there might be other termination measures which work, but since it seemed very hard to find one, an approach called "transitivity pushing" [6] was chosen, which is described in the next section.

4.2 The "transitivity pushing" approach

The problems described in the previous section are solved by [6], but unfortunately, its mechanized proof in Twelf uses the Reversed de Bruijn indices representation, which is not general enough, as shown in section 2.2. The contribution of this semester project is to rewrite the proof in Coq with the locally nameless representation, to ensure that the proof also works if we don't have the restriction that a bind type is a subtype of another bind type only if the definitions of both bind types use the same variable name for the self reference.

The proof is in the file `DotTransitivity.v`, and its idea is explained in this section.

4.2.1 Two subtyping modes

We add an explicit transitivity rule to the subtyping rules. However, this confronts us with the problems described in section 4.1.2. To address these, we introduce two subtyping modes:

- A subtyping judgment in `oktrans` mode can use all subtyping rules, including the explicit transitivity rule.
- A subtyping judgment in `notrans` mode may not use the explicit transitivity rule as its last rule, but is allowed to use it at deeper levels.

Then, we prove a theorem called `oktrans_to_notrans` which states that every subtyping judgment which holds in `oktrans` mode also holds in `notrans` mode. That is, we prove that we can "push" the explicit transitivity rule used as the last rule of a derivation down into deeper levels of the derivation.

[6] Based on an e-mail conversation from April 2014 between Tiark Rompf and Martin Odersky.

We can use this theorem to prove the subtyping inversion lemmas which we could not prove in section 4.1.2, because if the subtyping derivation that we get is in `oktrans` mode, we can just apply the theorem to convert it to a subtyping derivation in `notrans` mode.

So all which remains to do is to prove `oktrans_to_notrans`, and we'll see in the following subsections how to do this.

4.2.2 Transitivity in `notrans` mode without paths in the middle

First, we prove the lemma called `subtyp_trans_notrans`, which states that if we have a derivation of $T_1 <: T_2$ and a derivation of $T_2 <: T_3$, both in `notrans` mode, and T_2 is not a path type, then we can show $T_1 <: T_3$ in `notrans` mode. We can prove this lemma by case distinction on T_2, and there's nothing difficult in this proof.

Note that if we could prove this lemma without requiring that T_2 is not a path type, we could use it to prove the final `oktrans_to_notrans` theorem by simple induction on the size of the subtyping derivation. So let us study why we cannot prove it if T_2 is a path type:

We are given a derivation for $T_1 <: p.L$ and a derivation for $p.L <: T_3$, both in `notrans` mode, and we have to prove $T_1 <: T_3$ in `notrans` mode. From the two given derivations, we can extract the bounds $S..U$ of $p.L$, as well as derivations for $T_1 <: S$, $S <: U$, and $U <: T_3$. However, these derivations are all in `oktrans` mode, and if we wanted to do the proof by induction on the size of the derivations, we would need them to be in `notrans` mode. So we might need to repeatedly replace derivations in `oktrans` mode by their two premises, until we get a chain of only `notrans` derivations. That's the idea of the next subsection.

4.2.3 Going from `oktrans` to a "chain" by an algorithm

Imagine an algorithm called `step` [7] whose input is a "chain" of subtyping derivations of the form $T_1 <: T_2 <: ... <: T_n$, where each subtyping proof can be in `oktrans` or `notrans` mode. The algorithm traverses the chain, and replaces each subtyping proof in `oktrans` mode by its two premises, i.e. $... <: T_i <: T_{i+1} <: ...$ would be replaced by $... <: T_i <: U <: T_{i+1} <: ...$. Further, it replaces all path types by their bounds, i.e. $... <: p.L <: ...$ would be replaced by $... <: S <: U <: ...$, where S and U are the lower and upper bound of $p.L$.

Now we could start with a chain consisting of one `oktrans` derivation, and repeatedly apply `step` to it, until we get a chain which contains no path types (except maybe the first and the last type in the chain), and whose subtyping judgments all are in `notrans` mode. By repeatedly applying the `subtyp_trans_notrans` lemma, we could collapse this chain into a single `notrans` subtyping derivation. So we have sketched an algorithm which performs exactly the conversion that the `oktrans_to_notrans` theorem has to do. The only problem is that it's not clear how to prove that our algorithm terminates.

4.2.4 Refine the idea to make termination provable

So let us refine this idea to make termination provable. Observe that the chain can be seen as an intermediate data structure between the proof in `oktrans` mode and the proof in `notrans` mode. That is, it has to satisfy two criteria:

1) It must be possible to convert a proof in `oktrans` mode into a chain.

2) It must be possible to convert a chain into a proof in `notrans` mode.

[7] This is only a thought experiment and therefore not implemented in `DotTransitivity.v`.

Further, it must be possible to prove that both of these conversions terminate.

So we have to change the definition of a chain such that it satisfies these two criteria. To do so, we require that all derivations in a chain are in `notrans` mode, and that a chain contains no path types except maybe the first and the last type in the chain. This ensures that 2) is possible.

For 1), we write a lemma called `prepend_chain`, which takes a chain and prepends an `oktrans` derivation to it. Applying this lemma with an empty chain allows us to prove 1).

To prove `prepend_chain`, we would like to require that the first type in the chain is not a path type, so that we can prepend the `oktrans` derivation in a way similar to how the `subtyp_trans_notrans` lemma works. But if we require that, 1) does not hold any more for derivations of the form $p.L <: T$. So we must find a compromise between allowing and disallowing a path type at the beginning of the chain.

It turns out that this compromise can be encoded in a data structure we call `follow_ub`, which we put at the beginning of the chain. The purpose of `(follow_ub G p1.X1 B)` is, given an environment `G`, to take us from the path type $p_1.X_1$ to a type B, which is not a path type, by repeatedly taking the upper bound of the path type. That is, it encodes a list of the form

$$(p_1.X_1 : \_..p_2.X_2), (p_2.X_2 : \_..p_3.X_3), ...(p_N.X_N : \_..B)$$

where underscores are wildcards for types which do not matter. There's also a "pass-through" case in the definition of `follow_ub` for the case where the type at the beginning of the chain is not a path type.

Further, to satisfy 1), we have to introduce a "follow lower bound" structure symmetric to `follow_ub`, which we add at the end of the chain, and in the middle of the chain, we have to take into account two special cases related to reflexivity and the top type. So, our final definition of the chain looks as follows:

```
Definition st_middle (G: ctx) (B C: typ): Prop :=
  B = C \/
  subtyp notrans G typ_top C \/
  (notsel B /\ subtyp notrans G B C).

Definition chain (G: ctx) (A D: typ): Prop :=
   (exists B C, follow_ub G A B /\ st_middle G B C /\ follow_lb G C D)
      .
```

One might wonder why `st_middle` only contains one subtyping derivation, and not a list of them, as described before. This is because when constructing the chain, if we have two `notrans` derivations $S <: T$ and $T <: U$, where T is not a path type, we can immediately collapse them into one derivation by applying the `subtyp_trans_notrans` lemma.

It turns out that this definition of `chain` satisfies 1) and 2), so we can use it to prove our `oktrans_to_notrans` theorem.

4.3 Using the "transitivity pushing" approach

This "transitivity pushing" approach breaks the mutual dependency between subtyping transitivity and narrowing described in section 4.1.3, because we can prove narrowing in `oktrans` mode, where we can use transitivity wherever we want. When we want to prove subtyping inversion lemmas, we don't run into the problem described in section 4.1.2, because we can apply the `oktrans_to_notrans` theorem to turn any `oktrans` derivation into a `notrans` derivation.

References

[1] N. Amin, A. Moors, and M. Odersky. Dependent object types. In *19th International Workshop on Foundations of Object-Oriented Languages*, 2012.

[2] B. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. In *ACM SIGPLAN Notices*, volume 43, pages 3–15. ACM, 2008.

[3] N. G. De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. Elsevier, 1972.

[4] The Coq development team. *The Coq proof assistant reference manual.* LogiCal Project, 2014. Version 8.4, `http://coq.inria.fr`.

[5] F. Pfenning and C. Schürmann. System description: Twelf – a meta-logical framework for deductive systems. In *Automated Deduction – CADE-16*, pages 202–206. Springer, 1999.

[6] T. Rompf. "Transitivity pushing" proof in Twelf, 2014. `https://github.com/TiarkRompf/minidot/blob/master/dev2014/fsub-mini1h.elf`.

[7] J. Siek. Type safety in three easy lemmas, 2013. `http://siek.blogspot.ch/2013/05/type-safety-in-three-easy-lemmas.html`.