Master's Thesis

Mechanized Type Soundness Proofs using Definitional Interpreters

Author
Hannes Saffrich

Supervisor
Prof. Dr. Peter Thiemann

Albert-Ludwigs-Universität Freiburg im Breisgau
Technische Fakultät
Institut für Informatik

September 1, 2019

Abstract

Type soundness is a property of a typed programming language stating that a program's type faithfully describes the program's runtime behavior. The statement and proof structure of a type soundness theorem depend not only on the features of the programming language, but also on how the semantics is formalized. While formalizations using a small-step semantics are reasonably well explored, big-step semantics have received less attention, as they do not allow reasoning about non-terminating programs. However, this property can be regained by augmenting the big-step semantics with a simple step-counter, leading to a concise representation as a monadic definitional interpreter.

This master's thesis examines the use of step-indexed definitional interpreters as semantics for mechanized type soundness proofs. The general approach to the problem is briefly presented, followed by 5 case studies covering the simply typed lambda calculus and its extensions with mutable references, substructural types, subtyping, and parametric polymorphism. Each case study presented in this thesis is accompanied by a corresponding mechanization using the Coq proof assistant. The mechanizations can be found at `https://github.com/m0rphism/definitional`.

Zusammenfassung

Type Soundness ist eine Eigenschaft von getypten Programmiersprachen die aussagt, dass der Typ eines Programms auch wirklich das Laufzeitverhalten des Programms beschreibt. Die Formulierung und Beweisstruktur eines Type Soundness-Theorems sind nicht nur von den Merkmalen der Programmiersprache abhängig, sondern auch davon wie die Semantik formalisiert wird. Während Formalisierungen mit Small-Step Semantiken bereits ausgiebig erforsch sind, haben Big-Step Semantiken weniger Aufmerksamkeit erhalten, da diese es nicht erlauben Aussagen über nicht-terminierende Programme zu treffen. Diese Eigenschaft kann aber zurückgewonnen werden indem man die Big-Step Semantik um einen einfachen Schritt-Zähler erweitert, was sich zu einer präzisen Repräsentation als Monadic Definitional Interpreter eignet.

Diese Masterarbeit untersucht die Benutzung von schritt-indizierten Definitional Interpretern als Semantik für mechanisierte Type Soundness-Beweise. Der allgemeine Ansatz wird kurz präsentiert, gefolgt von 5 Fallstudien. Diese umfassen den Simply Typed Lambda Calculus und seine Erweiterungen mit Mutable References, Substructural Types, Subtyping und parametrischem Polymorphismus. Zu jeder Fallstudie, die in dieser Arbeit vorgestellt wird, gibt es eine entsprechende Mechanisierung mit dem Coq Beweisassistenten. Die Mechanisierungen sind verfügbar unter `https://github.com/m0rphism/definitional`.

Erklärung

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Datum Unterschrift

Contents

Chapter 1

Introduction

1.1 Motivation

The type system of a statically typed programming language is supposed to serve two purposes:

– it rules out certain classes of ill-formed programs, allowing implementations to avoid unnecessary runtime checks without risking undefined behavior; and

– it classifies the well-formed programs by certain aspects of their runtime behavior, allowing the programmer to rule out programs that exhibit well-defined but unintended behavior.

For both purposes, it is vital that the type system faithfully describes the runtime behavior of programs.

As an example, consider a language supporting basic operations on strings and integer numbers. In such a language one can formulate well-formed expressions like `2 + 3`, but also ill-formed expressions like `2 + "foo"`.

Without a type system, both forms are valid, so an implementation of the language would either have to dynamically check if the arguments to `+` are indeed integers, causing a runtime error for `2 + "foo"`, or assume that the arguments of `+` are always integers, causing undefined behavior for `2 + "foo"`, potentially leading to violated memory safety and security risks.

With a type system, we can rule out such ill-formed expressions, by giving the addition function `+` the type $\texttt{Int} \times \texttt{Int} \to \texttt{Int}$, but `"foo"` the type `String`. As the implementation has to deal only with well-typed programs, it can omit the runtime checks for `+` without risking undefined behavior.

However, this relies crucially on the fact that a well-typed program can only exhibit the runtime behavior expected of its type: if the type system would permit giving `"foo"` the type `Int`, even though it evaluates to a non-integral value, then both purposes would be violated.

1.2 Type Soundness

Statically typed programming languages are usually formalized by their syntax, semantics, and type system. The syntax describes the structure of programs, the semantics describes how programs can be evaluated in a specific environment, and the type system categorizes programs without assuming a specific environment.

A type system is sometimes also called *static semantics*[6], stressing that the types it assigns to a program are intended to capture aspects of the program's semantics that are valid for all possible environments.

However, there is no inherent connection between type systems and semantics, that makes the assigned types automatically describe the semantics of a program. This correspondence has to be proved first and is called *type soundness.*

Wright and Felleisen[18] describe type soundness in the context of a partial evaluation function

$$\mathsf{eval} : \mathsf{Programs} \to \mathsf{Answers} \cup \{\mathsf{wrong}\}$$

that maps erroneous programs (type errors) to wrong, and is undefined for nonterminating programs. Given a typing relation $\triangleright e : t$, they state two forms of type soundness:

$$\textsc{WeakSoundness}\quad \frac{\triangleright e : t}{\mathsf{eval}(e) \neq \mathsf{wrong}} \qquad\qquad \textsc{StrongSoundness}\quad \frac{\triangleright e : t \qquad \mathsf{eval}(e) = v}{v \in V^t}$$

– weak soundness asserts that if a program e has a type t, then evaluating e does not lead to a type error; and

– strong soundness asserts that if a program e has a type t, and evaluating e does terminate, then the result is not only not wrong, but also belongs to the set of values related to type t.

When the term *type soundness* is used unqualified, it usually refers to strong type soundness. Note, that specifying a type soundness theorem, does also require to specify the form of type errors by giving the semantics, and the form of well typed results by defining V^t, the set of values of type t.

For an implementation, weak type soundness means, that well-typed programs do not evaluate to wrong, so there's no need to generate code that dynamically checks for wrong. Note, that this does not rule out runtime errors in general: checked runtime errors can still be added to the semantics, they just have to use another encoding than wrong.

1.3 Influence of Semantics

The statement and proof structure of a type soundness theorem strongly depend on how the semantics of the language is formalized. In this section, we formalize the language fragment from Section 1.1 using different forms of semantics and examine the influence on the statement of a type soundness theorem.

We start by formalizing the syntax of program expressions:

```
Inductive Exp : Type :=
| e_num : ℕ → Exp
| e_str : String → Exp
| e_add : Exp → Exp → Exp.
```

An expression e : Exp is defined to be either

– a number literal e_num n for some number n;

– a string literal e_str s for some string s; or

– the addition e_add e1 e2 of two subexpressions e1, e2.

For example, we can represent the ill-formed term 2 + "foo" from the previous chapter as the expression e_add (e_num 2) (e_str "foo").

We give the syntax of types as

```
Inductive Typ : Type :=
| t_nat : Typ
| t_str : Typ.
```

i.e. a type t : Typ is defined to be either the type of natural numbers t_nat, or the type of strings t_str .

Next we formulate the type system, where ExpTyp e t stands for $\triangleright e : t$:

```
Inductive ExpTyp : Exp → Typ → Prop :=
| et_num :
    ∀ n, ExpTyp (e_num n) t_nat
| et_str :
    ∀ s, ExpTyp (e_str s) t_str
| et_add :
    ∀ e1 e2,
    ExpTyp e1 t_nat →
    ExpTyp e2 t_nat →
    ExpTyp (e_add e1 e2) t_nat.
```

Each constructor corresponds to a typing rule, and we have one constructor for each kind of expression:

– et_num states that for any number n the expression e_num n has type t_nat;

– et_str states that for any string s the expression e_str s has type t_str ;

– et_add states that an addition expression e_add e1 e2 has type t_nat, if both e1 and e2 have type t_nat.

For example, we can use the et_add and et_num constructors to derive

```
et_add _ _ (et_num 1) (et_num 2)
  : ExpTyp (e_add (e_num 1) (e_num 2)) t_nat
```

but no combination of constructors is able to derive

```
ExpTyp (e_add (e_num 1) (e_str "foo")) t
```

for any type t.

1.3.1 Small-Step Semantics

Small-step semantics describe evaluation through a binary relation $\_ \hookrightarrow \_$ between expressions and a notion of when an expression is considered a value.

The statement $e_1 \hookrightarrow e_2$ denotes that e_2 can be obtained from e_1 in a single evaluation step. The evaluation of an expression e is then viewed as the repeated application of the relation

```
e ↪ e1 ↪ e2 ↪ ...
```

Either the chain never stops - then e is considered non-terminating - or the chain stops at an expression en for some n. In the latter case, either en is considered a value, then the evaluation succeeds with that value, or the evaluation is stuck, representing a type error.

For our example language, we would expect such a relation to evaluate the expression $(1 + 2) + 3$ in two steps to the value 6

```
(1 + 2) + 3   ↪   3 + 3   ↪   6
```

whereas we would expect the ill-formed expression 2 + "foo" to be directly stuck.

We formally define the semantics, by giving two relations IsValue and Step:

```
Inductive IsValue : Exp → Prop :=
| iv_num : ∀ n, IsValue (e_num n)
| iv_str : ∀ s, IsValue (e_str s).
```

We consider e_num n and e_str s expressions as values, but not addition e_add e1 e2, as such an expression represents an unfinished computation.

```
Inductive Step : Exp → Exp → Prop :=
| s_add :
    ∀ n1 n2,
    Step (e_add (e_num n1) (e_num n2)) (e_num (n1 + n2)).
| s_add1 :
    ∀ e1 e2 e1',
    Step e1 e1' →
    Step (e_add e1 e2) (e_add e1' e2)
| s_add2 :
    ∀ e1 e2 e2',
    IsValue e1 →
    Step e2 e2' →
    Step (e_add e1 e2) (e_add e1 e2')
```

We define the semantics relation by three rules related to addition:

- the s_add rule states that an addition of two number values can be evaluated by simply adding the numbers;
- the s_add1 rule states that e_add e1 e2, can be evaluated to e_add e1' e2, if e1 can be evaluated to e1'; and
- the s_add2 rule states that e_add e1 e2, can be evaluated to e_add e1 e2', if e1 is already a value and e2 can be evaluated to e2'.

To be able to talk about sequences of evaluation steps, we define the reflexive, transitive closure Multi R of a binary relation R as

```
Inductive Multi {X : Type} (R : X → X → Prop) : X → X → Prop :=
| m_refl  : ∀ x,  Multi R x x
| m_step : ∀ x y z,  Multi R x y → R y z → Multi R x z.
```

This allows us to write Multi Step e e' to denote that e can be evaluated to e' in zero or more steps.

Wright and Felleisen[18] introduced the standard approach of proving soundness via small-step semantics with two lemmas:

Lemma 1.1 (Preservation)**.**

```
∀ e1 e2 t,  ExpTyp e1 t → Step e1 e2 → ExpTyp e2 t.
```

Lemma 1.2 (Progress)**.**

```
∀ e1 t,  ExpTyp e1 t → IsValue e1 ∨ ∃ e2, Step e1 e2.
```

The first lemma states that typing is preserved under evaluation, i.e. that if an expression e1 has type t, and e1 evaluates in one step to e2, then e2 also has type t.

The second lemma states that typed expressions are not type errors, i.e. that if an expression e has a type t, then either e is a value or it can be further reduced to some expression e2.

Together, they lead towards a syntactic soundness theorem:

Theorem 1.1 (Syntactic Type Soundness)**.**

```
∀ e t,
ExpTyp e t →
Diverges e ∨ ∃ v,  IsValue v ∧ Multi Step e v ∧ ExpTyp v t.
```

where Diverges e is defined as

```
Definition Diverges (e : Exp) : Prop :=
  ∀ e',  Multi Step e e' → ∃ e'',  Step e' e''.
```

The Syntactic Type Soundness theorem is close to Wright and Felleisen's statement of strong soundness. However, in most mechanizations with small-step semantics only the preservation and progress lemma are proved, but not a syntactic soundness theorem.

Wright and Felleisen describe the type soundness proofs via preservation and progress lemmas as "lengthy but simple, requiring only basic inductive techniques"[18].

1.3.2 Big-Step Semantics

Big-step semantics describe evaluation through a binary relation $\_ \Downarrow \_$ directly between expressions and the values they evaluate to.

For example, in our language we would expect $(1 + 2) + 3 \Downarrow 6$ to hold.

To formally describe the big-step semantics, we first give a notion of value. In contrast to small-step semantics, values are not a sub-class of expressions, but a separate syntactic entity. In our simple language, the only values are numbers and strings:

```
Inductive Val : Type :=
| v_num : ℕ → Val
| v_str : String → Val.
```

We then define the semantics relation with one constructor for each kind of expression:

```
Inductive BigStep : Exp → Val → Prop :=
| bs_num :
    ∀ n,
    BigStep (e_num n) (v_num n)
| bs_str :
    ∀ s,
    BigStep (e_str s) (v_str s)
| bs_add :
    ∀ e1 e2 n1 n2,
    BigStep e1 (v_num n1) →
    BigStep e2 (v_num n2) →
    BigStep (e_add e1 e2) (v_num (n1 + n2)).
```

- bs_num states that a number expression e_num n evaluates to the number value v_num n;
- bs_str states that a string expression e_str s evaluates to the string value v_str s; and
- bs_add states that an addition expression e_add e1 e2 evaluates to v_num (n1 + n2) if e1 evaluates to v_num n1 and e2 evaluates to v_num n2.

Before we come to type soundness, we need to specify a typing relation between values and types, as values are now a separate syntactic entity. The typing relation simply states, that any number or string value has a number or string type, respectively:

```
Inductive ValTyp : Val → Typ → Prop :=
| vt_num : ∀ n, ValTyp (v_num n) t_nat
| vt_str : ∀ s, ValTyp (v_str s) t_str .
```

There are now two obvious choices for trying to formulate type soundness, which are unfortunately both insufficient:

$$\frac{\mathsf{ExpTyp\ e\ t}}{\exists\ \mathsf{v} \qquad \mathsf{BigStep\ e\ v} \qquad \mathsf{ValTyp\ v\ t}} \qquad\qquad \frac{\mathsf{ExpTyp\ e\ t} \qquad \mathsf{BigStep\ e\ v}}{\mathsf{ValTyp\ v\ t}}$$

The left theorem states, that if an expression e has type t, then e evaluates to some value v of type t. While correct for our simple language, this statement is too strong in general: as soon as we have well-typed, non-terminating expressions, it is not true anymore that BigStep e v holds for all well-typed expressions.

The right theorem states, that if an expression e has type t and e evaluates to value v, then v has type t. While correct, this statement is too weak in general: it only guarantees the abscence of type errors for terminating programs. For non-terminating programs, the assumption BigStep e v can not be satisfied, so type errors are not proved impossible in those cases.

1.3.3 Definitional Interpreter

The problem with big-step semantics is, that to formulate a type soundness theorem of the right strength, it is necessary to distinguish between non-terminating programs and type errors.

While we could change the semantics relation, such that it is still undefined for non-terminating programs, but returns a special value wrong for type errors, and right v for regular values, this would lead to ugly artifacts in the formalization of the semantics, as a lot of rules would have to be added, just to propagate wrong through subexpressions.

A cleaner representation can be achieved by encoding the semantics relation directly as a definitional interpreter in Coq, which allows the propagation of type errors to be hidden behind a monad.

As Coq is a total meta language, it is not possible to implement a definitional interpreter for languages that are not strongly normalizing as a regular Coq-function. However, this can be worked around by extending the interpreter with a step-counter, that restricts the maximal recursive depth of the interpreter.

We represent the error and non-termination conditions each through the Maybe type:

```
Inductive Maybe (X : Type) : Type :=
| none : Maybe X
| some : X → Maybe X.
```

To increase readability, we use the following notations:

CanTimeout :≡ Maybe	timeout :≡ none	done :≡ some
CanErr :≡ Maybe	error :≡ none	noerr :≡ some

We then state the definitional interpreter as a Coq function

```
eval : ℕ → Exp → CanTimeout (CanErr Val)
```

such that eval n e corresponds to trying to evaluate the expression e in n steps, returning

- timeout if the number of steps n was too small;
- done error if the evaluation caused a type error; and
- done (noerr v) if the evaluation succeeded with value v.

Our first definition of the interpreter is without monadic notation:

```
Fixpoint eval (n : ℕ) (e : Exp) : CanTimeout (CanErr Val) :=
  match n with
  | 0 ⇒ timeout
  | S n ⇒
      match e with
      | e_num n ⇒ done (noerr (v_num n))
      | e_str s ⇒ done (noerr (v_str s))
      | e_add e1 e2 ⇒
          match eval n e1 with
          | done (noerr (v_num n1)) ⇒
```

```
                match eval n e2 with
                | done (noerr (v_num n2)) ⇒
                    done (noerr (v_num (n1 + n2)))
                | done _ ⇒ done error
                | timeout ⇒ timeout
                end
            | done _ ⇒ done error
            | timeout ⇒ timeout
            end
        end
    end.
```

The interpreter first checks if there are any steps n left to perform. If this is not the case, evaluation is stopped by returning timeout. Otherwise, evaluation proceeds by pattern matching on the expression e: If e is a number or string literal, then the corresponding value is returned, requiring no further steps. If e is the addition e_add e1 e2 of two other expressions, then we try to evaluate both subexpressions in at most n−1 steps. If both subexpressions evaluated successfully to number values v_num, then the evaluation of the addition succeeds by returning the sum of the number values. However, if one of the subexpressions fails to evaluate, then evaluation of the addition has to fail accordingly, causing a lot of noise through branches for simple error propagation.

To hide the propagation of the timeout and error cases, we introduce a notation for the monadic sequencing of the CanTimeout ∘ CanErr monad:

```
Notation "' p ← e1 ; e2" :=
  (match e1 with
   | done (noerr p) ⇒ e2
   | done _         ⇒ done error
   | timeout        ⇒ timeout
   end)
  (right associativity , at level 60, p pattern).
```

Note, that p is specified as a pattern parameter, so if p fails to match, then done error is returned, similar to Haskell's MonadFail concept.

We can now reformulate the interpreter in a much more concise way:

```
Fixpoint eval (n : ℕ) (e : Exp) : CanTimeout (CanErr Val) :=
  match n with
  | 0 ⇒ timeout
  | S n ⇒
      match e with
      | e_num n ⇒ done (noerr (v_num n))
      | e_str s ⇒ done (noerr (v_str s))
      | e_add e1 e2 ⇒
          ' v_num n1 ← eval n e1;
          ' v_num n2 ← eval n e2;
          done (noerr (v_num (n1 + n2)))
      end
  end.
```

Finally, we state the type soundness theorem:

Theorem 1.2 (Type Soundness)**.**

$$\forall\ \mathsf{n\ e\ mv\ t},$$
$$\mathsf{eval}\ \ \mathsf{n\ e} = \mathsf{done\ mv} \to$$
$$\mathsf{ExpTyp\ e\ t} \to$$
$$\exists\ \mathsf{v},\ \ \mathsf{mv} = \mathsf{noerr\ v} \wedge \mathsf{ValTyp\ v\ t}.$$

This theorem is closely related to Wright and Felleisen's strong soundness. The only difference is the step index n.

Similar to Wright and Felleisen's approach for small-step semantics, the type soundness proofs using step-indexed definitional interpreters require only basic inductive proof techniques.

In contrast to small-step semantics, it's straightforward to derive an implementation from the semantics: it suffices to omit the step-index from the definitional interpreter, such that it runs as much steps as needed.

1.4 Related Work

Milner famously gave an informal definition of type soundness in 1978: "Well-typed programs cannot go wrong"[7].

Wright and Felleisen's seminal paper from 1994 restated this as: "Well-typed programs do not get stuck", and coined the usage of preservation and progress for type soundness with small-step semantics that's widely used until today[18].

The approach with step-indexed definitional interpreters was recently used in 2015 for Coq mechanizations of type soundness proofs related to Scala's Dot Calculus[13] and second-class values[8]. The formalization of the Dot Calculus with a definitional interpreter, instead of a small-step semantics, brings the benefit that no workaround for a *substitution preserves typing* lemma is necessary, which doesn't hold for the Dot Calculus in general. The technique of step-indexing a definitional interpreter was presented in two blog posts by Siek[15, 16], who dates it back to a book by Ernst[5] from 2006.

The simply typed lambda calculus with small step semantics and many of its extensions have been proved sound in the foundational books[9, 10, 6, 11].

SML has been proved type sound, up to unsafe system operations provided by the implementations[4].

Rust's core has been proved type sound[17].

Java's and Scala's type systems have been proved unsound[14, 1]. The second paper derives a Java function that can coerce any type to any other type, without making use of type casting. Fortunately, the soundness hole only leads to a runtime exception in the Java Virtual Machine.

1.5 Outline & Contributions

The rest of this thesis is structured according to Figure 1.1:

– Chapter 2 presents a small foundational framework for definitional interpreters on which the formalizations in the subsequent chapters are based. The framework contains definitions of and lemmas about basic data structures like lists, natural numbers, and the Maybe type;

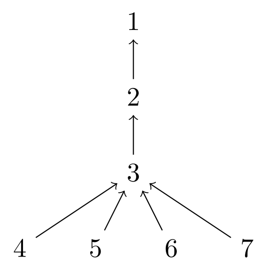

Figure 1.1: Dependencies between chapters

- Chapter 3 presents a formalization of the simply typed lambda calculus with a step-indexed definitional interpreter, and states and proves the corresponding type soundness theorem. A big-step semantics is formalized and proved equivalent to the definitional interpreter. The type soundness theorem requires only a single lemma, which is part of the framework;
- Chapter 4 extends the formalization of the simply typed lambda calculus with subtyping;
- Chapter 5 extends the formalization of the simply typed lambda calculus with substructural types, such that lambda abstractions with both affine and unrestricted multiplicities are supported;
- Chapter 6 extends the formalization of the simply typed lambda calculus with mutable references; and
- Chapter 7 extends the formalization of the simply typed lambda calculus with parametric polymorphism à la System F. The formalization is inspired by the mechanization of System $F_{<:}$ by Rompf and Amin[13].

The formalizations are presented using actual Coq code, but the proofs are presented informally, as Coq's tactic scripts are very hard to read without an interactive support system. To help the reader to relate the presentations in this thesis to the actual Coq mechanizations, we use the same names for definitions, lemmas, and variables as in the actual Coq files.

To the best of our knowledge, the soundness theorems for subtyping, substructural types, and System F have not been proved with definitional interpreters before.

As an additional contribution, not presented in this thesis, we have also created an alternative version to Rompf and Amin's System $F_{<:}$ proof, which proves the equivalence between the logical and the algorithmic subtyping of System $F_{<:}$, instead of using a workaround with "transitivity pushback". While this equivalence has been proved before[9], it hasn't been proved in the context of definitional interpreters, where the subtyping relation for values has to incorporate a type equivalence modulo environments, which causes non-trivial complications. We choose not to present the formalization, as it would exceed the bounds of a master's thesis.

Chapter 2

Framework

In this chapter, we introduce a set of general definitions and lemmas, that are helpful for the formalizations presented in all subsequent chapters. Those are pretty standard and should be largely included in the standard libraries of most proof assistents.

2.1 Maybe Monad

As we have already covered the Maybe monad in Section 1.3.3, we only repeat the definitions for completeness, and introduce one new definition: the map function.

The Maybe type is given by

```
Inductive Maybe (X : Type) : Type :=
| none : Maybe X
| some : X → Maybe X.
```

We use the following notations in the context of definitional interpreters:

CanTimeout :≡ Maybe	timeout :≡ none	done :≡ some
CanErr :≡ Maybe	error :≡ none	noerr :≡ some

The monadic sequencing of the CanTimeout ∘ CanErr monad is given by:

```
Notation "' p ← e1 ; e2" :=
  (match e1 with
   | done (noerr p) ⇒ e2
   | done _         ⇒ done error
   | timeout        ⇒ timeout
   end)
  (right associativity , at level 60, p pattern).
```

The map operation is given by

```
Definition mmap {X Y : Type} (f : X → Y) (mx : Maybe X) : Maybe Y :=
  match mx with
  | none ⇒ none
  | some x ⇒ some (f x)
  end.
```

2.2 Natural Numbers

The systems covered in this thesis require a notion of variables. In most presentations, the precise definition of variables is left opaque, and instead the existence of some countably infinite set is assumed, together with a notion of decidable equality on its elements.

We choose to identify this set of variables simply with the natural numbers:

```
Inductive ℕ : Set :=
| O : ℕ
| S : ℕ → ℕ.
```

To define a decidable equality, we make use of the booleans

```
Inductive 𝔹 : Set :=
| true  : 𝔹
| false : 𝔹.
```

We use a standard decision procedure to decide equality of natural numbers:

```
beq_nat : ℕ → ℕ → 𝔹
```

and a corresponding reflection principle:

```
beq_eq_iff : ∀ (x y : ℕ), beq_nat x y = true ↔ x = y
```

2.3 Lists

Working with languages that provide variables or memory locations, requires defining our relations with respect to the types and values of those variables or memory locations.

As we are going to represent variables and memory locations as natural numbers, it is natural to represent environments, e.g. mappings from variables to values, as lists of values indexed by their variables.

Hence, we introduce a list data type with a few basic operations and lemmas about them.

```
Inductive List (X : Type) : Type :=
| nil  : List A
| cons : A → List A → List A
```

We use the notation [] for nil, and x :: xs for cons x xs.

2.3.1 Basic Operations

The length function computes the number of elements in the list.

```
Fixpoint length {X : Type} (xs : List X) : ℕ =
  match xs with
  | [] ⇒ 0
  | _ :: xs ⇒ S (length xs)
  end.
```

The indexr function retrieves the n-th element counting from the right of the list, e.g. indexr 0 (x :: y :: []) = some y.

```
Fixpoint indexr {X : Type} (n : ℕ) (xs : List X) : Maybe X :=
  match xs with
  | []        ⇒ none
  | x :: xs' ⇒ if beq_nat n (length xs') then some x else indexr n xs'
  end.
```

The append function concatenates two lists. We use the notation xs1 ++ xs2 to denote append xs1 xs2.

```
Fixpoint append {X : Type} (xs1 xs2 : List X) : List X :=
  match xs1 with
  | [] ⇒ xs2
  | x :: xs1 ⇒ x :: append xs1 xs2
  end.
```

The update function replaces the n-th element counting from the right of the list, e.g. update 0 y' (x :: y :: []) = x :: y' :: [].

```
Fixpoint update {X : Type} (n : ℕ) (x' : X) (xs : List X) : List X :=
  match xs with
  | [] ⇒ []
  | x :: xs ⇒ if beq_nat n (length xs)
              then x' :: xs
              else x :: update n x' xs
  end.
```

Next, we proof two lemmas related to indexr:

Lemma 2.1 (indexr_max)**.**

```
∀ X (xs : List X) (n : ℕ) (x : X),
 indexr n xs = some x →
 n < length xs.
```

Proof. Straightforward induction over xs, followed by a case analysis on n in the cons case. □

Lemma 2.2 (indexr_extend)**.**

```
∀ X xs n x' (x : X),
indexr n xs = some x →
indexr n (x' :: xs) = some x.
```

Proof. Straightforward reasoning using Lemma 2.1. □

2.3.2 Forall2

When we represent values and types of variables as lists of values and types, then we often need to state that a binary relation R holds between the value and type of each variable.

We cover this scenario generally by introducing the Forall2 R xs ys type, which states that the binary relation R : X →Y →Prop holds between each pair of the zipping of the two lists xs and ys, i.e. R x1 y1 ∧ ... ∧ R xn yn.

```
Inductive Forall2 {X Y : Type} (R : X → Y → Prop) : List X → List Y →
    Prop :=
| fa2_nil :
    Forall2 R [] []
| fa2_cons :
    ∀ (x : X) (y : Y) (xs : List X) (ys : List Y),
    R x y →
    Forall2 R xs ys →
    Forall2 R (x :: xs) (y :: ys).
```

If two lists are related by Forall2 R xs ys, then by construction they have the same length:

Lemma 2.3 (fa2_length)**.**

```
∀ {X Y} {R : X → Y → Prop} {xs ys},
Forall2 R xs ys →
length xs = length ys.
```

Proof. Straightforward induction over the evidence for Forall2 R xs ys. □

While the next lemma is intuitively obvious, it will be essential in the variable cases of all type soundness theorems presented in this thesis:

Lemma 2.4 (fa2_indexr)**.**

```
∀ {X Y} {R : X → Y → Prop} {xs ys} {y} {n},
Forall2 R xs ys →
indexr n ys = some y →
∃ x, indexr n xs = some x ∧ R x y.
```

Proof. We start by induction over Forall2 R xs ys:

– **Case** fa2_nil **.** By definition of fa2_nil , we have ys = [], so by definition of indexr, the assumption indexr n ys = some t reduces to none = some t, so we can discard this case by contradiction.

– **Case** fa2_cons**.** By definition of fa2_cons, we have some xs', ys', x', y' such that

$$\text{xs} = \text{x'} :: \text{xs'} \qquad \text{ys} = \text{y'} :: \text{ys'} \qquad \text{Forall2 R xs' ys'} \qquad \text{R x' y'}$$

We proceed by case analysis on the index n:

  – **Case** 0**.** By definition of indexr, we have

$$\text{indexr 0 xs} = \text{some x'} \qquad \text{indexr 0 ys} = \text{some y'} = \text{some y}$$

  so we choose x = x', and are done by assumptions.

  – **Case** n+1**.** By definition of indexr, we have

$$\text{indexr (n + 1) xs} = \text{indexr n xs'}$$
$$\text{indexr (n + 1) ys} = \text{indexr n ys'} = \text{some y}$$

  so we apply the induction hypothesis to conclude the goal.

□

We prove a similar lemma using update instead of indexr, which will be used by the formalization of mutable references presented in Chapter 6:

Lemma 2.5 (fa2_update_l)**.**

```
∀ {X Y} (R : X → Y → Prop)
  (xs : List X) (ys : List Y) (n : ℕ) (x : X) (y : Y),
indexr n ys = some y →
Forall2 R xs ys →
R x y →
Forall2 R (update n x xs) ys.
```

Proof. Very similar structure to fa2_indexr. □

2.3.3 Suffixes

When we come to the formalization of mutable references in Chapter 6, we need to state what it means for a list to be the suffix of another list.

We define the suffix-relation by stating that a list xs1 is the suffix of a list xs2, if there exists some list xs such that appending xs to xs1 results in xs2:

```
Definition IsSuffixOf {X} (xs1 xs2 : List X) : Prop :=
  ∃ xs, xs ++ xs1 = xs2.
```

Next, we prove that the suffix-relation is reflexive and transitive:

Lemma 2.6 (suffix_refl)**.**

```
∀ {X} {xs : List X},
IsSuffixOf xs xs.
```

Proof. Immediate by choosing [] for the existential variable from IsSuffix. □

Lemma 2.7 (suffix_trans)**.**

```
∀ {X} {xs1 xs2 xs3 : List X},
 IsSuffixOf xs1 xs2 →
 IsSuffixOf xs2 xs3 →
 IsSuffixOf xs1 xs3.
```

Proof. Straightforward reasoning using associativity of append. □

The next lemma states that if xs1 is a suffix of xs2, then right-indexing the lists at their common entry yields common results:

Lemma 2.8 (indexr_suffix)**.**

```
∀ {X} n (xs1 xs2 : List X) (x : X),
 indexr n xs1 = some x →
 IsSuffixOf xs1 xs2 →
 indexr n xs2 = some x.
```

Proof. Straightforward induction over xs2, followed by a case analysis on n in the cons case, and an application of Lemma 2.1. □

Chapter 3

Simply Typed Lambda Calculus

In this chapter, we give a formalization of the Simply Typed Lambda Calculus (STLC) with the empty base type using a stepped definitional interpreter semantics, and then state and proof the corresponding type soundness theorem. This chapter forms the basis on which all formalizations and proofs from later chapters build on.

3.1 Syntax

The syntax of the STLC is usually given by a grammar like

$$t ::= \emptyset \mid t \to t \qquad \text{(Types)}$$

$$e ::= x \mid \lambda x : t.e \mid e\ e \qquad \text{(Expressions)}$$

stating that

- a type t is either the void type $\emptyset$; or a function type $t_1 \to t_2$ between two other types t_1 and t_2; and
- an expression e is either a variable x; a lambda abstraction $\lambda x : t.e$ binding a variable x of type t in body e; or a lambda application $e_1\ e_2$ applying e_1 to argument e_2.

In our formalization, we make two changes to this representation:

- we use a nameless representation of variables as DeBruijn Levels[3]; and
- we omit the type annotation in lambda abstractions, as those play no role for type soundness.

The variable representation as DeBruijn Levels encodes variables as natural numbers n referring to the n-th outmost lambda abstraction. This makes the variable names in the binders of lambda abstractions redundant, as the variables themselves state to which binder they belong. For example, the lambda term $\lambda f.\ \lambda x.\ f\ x$ has the DeBruijn Level encoding $\lambda.\ \lambda.\ 0\ 1$.

While nameless variable representations enjoy many interesting properties, like α-equivalence being the same as syntactic equality, those properties are only relevant in our last case study of parametric polymorphism. For the simply typed lambda calculus, the choice is irrelevant to the proof structure of type soundness, as we will discuss in Section 3.7. The reason, why we still choose this representation, is that it allows us to present all case studies in a uniform manner, and to reuse various basic lemmas for different language features.

Thus, we formalize the syntax as

```
Inductive Typ : Type :=
  | t_void  : Typ
  | t_arr   : Typ → Typ → Typ.

Inductive Exp : Type :=
  | e_var  : ℕ → Exp
  | e_app  : Exp → Exp → Exp
  | e_abs  : Exp → Exp.
```

The lambda term $\lambda f.\ \lambda x.\ f\ x$ is then encoded as

```
e_abs (e_abs (e_app (e_var 0) (e_var 1))).
```

3.2 Type System

In this section, we specify the type system of the STLC.

In Section 1.3, we specified the type system of our example language as a binary relation $\triangleright e : t$, stating that expression e has type t. If we try the same for the STLC, we run into problems with the variable case: a variable e_var x hasn't a fixed type by itself, but instead has a type determined by the variable's context.

To solve this problem, we specify the type system as a tertiary relation between expressions, types, and a so called type environment, that records the types of variables. The basic idea is then, that the typing relation extends the type environment when it goes inside an abstraction, such that the contained variables can refer to the type environment for their type.

As we represent variables as DeBruijn Levels, we define type environments simply as lists of types, which are indexed by variables:

```
Definition TypEnv := List Typ.
```

We then define the type system by giving one constructor for each expression:

```
Inductive ExpTyp : TypEnv → Exp → Typ → Prop :=
| et_var :
    ∀ x te t,
    indexr x te = some t →
    ExpTyp te (e_var x) t
| et_app :
    ∀ te e1 e2 t1 t2,
    ExpTyp te e1 (t_arr t1 t2) →
    ExpTyp te e2 t1 →
    ExpTyp te (e_app e1 e2) t2
```

```
| et_abs :
    ∀ te e t1 t2,
    ExpTyp (t1 :: te) e t2 →
    ExpTyp te (e_abs e) (t_arr t1 t2).
```

- The `et_var` constructor states that a variable `e_var x` has type `t`, if the type environment `te` records that `x` has indeed type `t`;
- The `et_abs` constructor states that a lambda abstraction `e_abs e` has type `t_arr t1 t2`, if its body `e` has type `t2` in type environment `t1 :: te`, i.e. in the type environment `te` that now also records that the variable bound by the abstraction has type `t1`; and
- The `et_app` constructor states that a lambda application `e_app e1 e2` has type `t2`, if `e1` has a function type `t_arr t1 t2`, and `e2` has the corresponding argument type `t1`.

3.3 Big-Step Semantics

In this section, we specify the big-step semantics of the STLC. While we don't need the big-step semantics for our formalization of type soundness, which will be strictly in terms of a definitional interpreter, we still define the big-step semantics for comparison and to state an equivalence theorem with respect to the definitional interpreter in the next section.

When we try to state the big-step semantics as a binary relation, as we did in Section 1.3, we run into the same problems as for the type system: just like the type of a variable depends on the variable's context, so does its value.

Hence, we use the same strategy as before and introduce the semantics relation as a tertiary relation between expressions, values, and value environments.

Similar to type environments, we represent value environments simply as lists of values indexed by variables:

```
Definition ValEnv := List Val.
```

The only values we have are closures resulting from the evaluation of lambda abstractions:

```
Inductive Val :=
| v_abs (ve : ValEnv) (e : Exp).
```

In contrast to a lambda abstraction, which only carries its body `e`, a closure also carries the value environment `ve` in which the original lambda abstraction was evaluated. The reason for this is, that lambda abstractions may capture variables from the outside. If we then apply such an abstraction later to an argument, we need to access the values of those captured variables to evaluate the body of the abstraction.

We are now equipped to specify the semantics relation:

```
Inductive BigStep : ValEnv → Exp → Val → Prop :=
| bs_var :
    ∀ ve x v,
    index x ve = some v →
    BigStep ve (e_var x) v
| bs_abs :
    ∀ ve e,
    BigStep ve (e_abs e) (v_abs ve e)
| bs_app :
    ∀ ve e1 e2 ve' e' v2 v,
    BigStep ve e1 (v_abs ve' e') →
    BigStep ve e2 v2 →
    BigStep (v2 :: ve') e' v →
    BigStep ve (e_app e1 e2) v.
```

- the bs_var constructor states that a variable e_var x evaluates to a value v, if the value environment ve maps x to that value;
- the bs_abs constructor states that a lambda abstraction e_abs e evaluates to its closure v_abs ve e in the current environment ve; and
- the bs_app constructor states that a lambda application e_app e1 e2 evaluates to a value v, if e1 evaluates to a closure v_abs ve' e', e2 evaluates to some value v2, and the closure's body e1' evaluates to v in its captured environment ve' extended by the argument value v2 for the closure's variable.

3.4 Definitional Interpreter

In this section, we derive a monadic definitional interpreter for the STLC from the big-step semantics presented in the previous section.

Compared to the definitional interpreter from Subsection 1.3.3, our interpreter function now requires an additional argument for the value environment.

The translation from the big-step semantics is straightforward:

```
Fixpoint eval (n : ℕ) (ve : ValEnv) (e : Exp) : CanTimeout (CanErr Val)
    :=
  match n with
  | 0 ⇒ none
  | S n ⇒
      match e with
      | e_var x ⇒ done (indexr x ve)
      | e_abs e ⇒ done (noerr (v_abs ve e))
      | e_app e1 e2 ⇒
          ' v_abs ve1' e1' ← eval n ve e1;
          ' v2 ← eval n ve e2;
          eval n (v2 :: ve1') e1'
      end
  end.
```

– variables e_var x are evaluated in one step that is successful exactly if the value environment ve contains some value for x, i.e. indexr x ve = some v;

– abstractions e_abs e are always evaluated successfully in one step to their closure v_abs ve e in the current environment; and

– applications e_app e1 e2 are evaluated successfully in n+1 steps, if both their arguments and the closure body evaluate successfully in n steps to values of the expected form. If one of the evaluations of the subexpressions timeouts or fails, then the monadic sequencing ensures that the whole computation timeouts or fails as expected.

It's straightforward to proof, that the big-step semantics is equivalent to the definitional interpreter, in the sense that the big-step semantics evaluates an expression to a value if and only if the definitional interpreter evaluates the expression to the same value in some number of steps:

Theorem 3.1 (sem_eq)**.**

```
∀ ve e v,
BigStep ve e v ↔ (∃ n, eval n ve e = done (noerr v)).
```

Proof. We choose to omit the proof from the presentation, as this equivalence is not central to this thesis. It's a simple proof using only induction in both directions. The interested reader can refer to the Coq mechanization in the accompanied file Chap_3_STLC_SemEq.v. □

3.5 Type Soundness

Before we state a type soundness theorem, we have to specify the value typing. Our only kind of values is closures v_abs ve e, which result from lambda abstractions e_abs e. Hence, the value typing of closures is similar to the expression typing of abstractions, but has to additionally take care of the value environment ve. For each value of the captured variables from ve, we need a witness that the value indeed has the variable's type in the type environment te of the closure's body e. To model this well-formedness relationship between value and type environments, we make use of the Forall2 type from Chapter 2

```
Definition WfEnv : ValEnv → TypEnv → Prop :=
  Forall2 ValTyp.
```

and state the value typing as

```
Inductive ValTyp : Val → Typ → Prop :=
| vt_abs :
    ∀ ve te e t1 t2,
    Forall2 ValTyp ve te →
    ExpTyp (t1 :: te) e t2 →
    ValTyp (v_abs ve e) (t_arr t1 t2).
```

The vt_abs constructor states, that a closure v_abs ve e has type t_arr t1 t2, if there is some type environment te, such that the values of ve have their corresponding type in te, and the closure's body e has type t2 in te extended by t1.

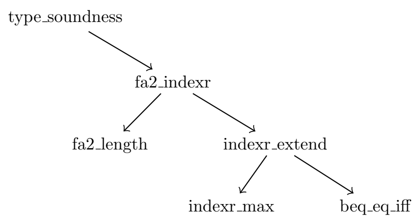

Figure 3.1: Proof Graph for STLC Soundness

We are now equipped to state type soundness as

Theorem (Type Soundness)**.**

```
∀ (n : ℕ) (e : Exp) (mv : CanErr Val) (t : Typ),
eval n [] e = done mv →
ExpTyp [] e t →
∃ v, mv = noerr v ∧ ValTyp v t.
```

While correct, this formulation does not give us a suitable induction hypothesis. The evaluation of an application `e_app e1 e2` requires us to reason about the body of the closure value resulting from the evaluation of `e1`. This body is typed and evaluated in environments that are different from `[]`.

Hence, we strengthen the theorem as follows (changes are marked red):

Theorem (Type Soundness)**.**

```
∀ (n : ℕ) (e : Exp) (te : TypEnv) (ve : ValEnv) (mv : CanErr Val)
  (t : Typ),
eval n ve e = done mv →
ExpTyp te e t →
WfEnv ve te →
∃ v, mv = noerr v ∧ ValTyp v t.
```

3.6 Type Soundness Proof

Figure 3.1 shows the proof graph of the type soundness theorem. The proof of the theorem itself requires only a single lemma, which is used in the variable case. As we have already proved this lemma in the framework as Lemma 2.4, we start directly with the proof of the type soundness theorem:

Theorem 3.2 (Type Soundness)**.**

```
∀ (n : ℕ) (e : Exp) (te : TypEnv) (ve : ValEnv) (mv : CanErr Val)
  (t : Typ),
eval n ve e = done mv →
ExpTyp te e t →
WfEnv ve te →
∃ v, mv = noerr v ∧ ValTyp v t.
```

Proof. We start by induction over the number of steps `n`:

- **Case** `0`**.** By definition of `eval`, the assumption `eval 0 ve e = done mv` reduces to `timeout = done mv`, so we can discard this case by contradiction.

- **Case** `n + 1`**.** We proceed by case analysis on the typing derivation `ExpTyp te e t`:

– **Case** et_var. By definition of et_var, we have some x such that

    e = e_var x   indexr x te = noerr t.

By definition of eval, the assumption eval (n + 1) ve (e_var x) = done mv reduces to

    done (indexr x ve) = done mv

Thus, by substituting indexr x ve for mv, we are left to prove

```
WfEnv ve te →
indexr x te = noerr t →
∃ v, indexr x ve = noerr v ∧ ValTyp v t
```

which is an instance of Lemma 2.4 (fa2_indexr).

– **Case** et_abs. By definition of et_abs, we have some e', t1, t2 such that

    e = e_abs e'   t = t_arr t1 t2   ExpTyp (t1 :: te) e' t2

By definition of eval, the assumption eval (n + 1) ve (e_abs e') = done mv reduces to

    done (noerr (v_abs ve e')) = done mv

Thus, by substituting for mv, we are left to prove

    ∃ v, v_abs ve e' = v ∧ ValTyp v (t_arr t1 t2)

so we choose v = v_abs ve e' and construct the value typing from our assumptions:

$$\frac{\mathsf{WfEnv\ ve\ te} \qquad \mathsf{ExpTyp\ (t1 :: te)\ e'\ t2}}{\mathsf{ValTyp\ (v\_abs\ ve\ e')\ (t\_arr\ t1\ t2)}}\ \text{VT\_ABS}$$

– **Case** et_app. By definition of et_app, we have some e1, e2, t1, t2 such that

    e = e_app e1 e2   t = t2   ExpTyp te e1 (t_arr t1 t2)   ExpTyp te e2 t1

By definition of eval, the assumption

    eval (n + 1) ve (e_app e1 e2) = done mv

reduces to

```
' v_abs ve' e1' ← eval n ve e1;
' v2 ← eval n ve e2;
eval n (v2 :: ve') e1'          = done mv
```

Next, we observe that there must be some mv1 and mv2 such that

    eval n ve e1 = done mv1   eval n ve e2 = done mv2 because otherwise our definition of monadic sequencing would cause the whole left hand side to evaluate to timeout, leading to the contradiction timeout = done mv.

We are now equipped to apply our induction hypothesis to the evaluation of both subexpressions:

$$\frac{\mathsf{eval\ n\ ve\ e1 = done\ mv1} \qquad \mathsf{ExpTyp\ te\ e1\ (t\_arr\ \ t1\ \ t2)} \qquad \mathsf{WfEnv\ ve\ te}}{\exists\ \mathsf{v1,\ mv1 = noerr\ v1} \wedge \mathsf{ValTyp\ v1\ (t\_arr\ \ t1\ \ t2)}}\ \text{IH}$$

$$\frac{\mathsf{eval\ n\ ve\ e2 = done\ mv2} \qquad \mathsf{ExpTyp\ te\ e2\ t1} \qquad \mathsf{WfEnv\ ve\ te}}{\exists\ \mathsf{v2,\ mv2 = noerr\ v2} \wedge \mathsf{ValTyp\ v2\ t1}}\ \text{IH}$$

By inversion of the value typing ValTyp v1 (t_arr t1 t2), we find some te', ve', e1' such that

$$\mathsf{v1 = v\_abs\ ve'\ e1'} \quad \mathsf{ExpTyp\ (t1\ ::\ te')\ e1'\ t2} \quad \mathsf{WfEnv\ ve'\ te'}$$

By substituting for mv1, mv2, and v1, we now know

$$\mathsf{eval\ n\ ve\ e1 = done\ (noerr\ (v\_abs\ ve'\ e1'))}$$
$$\mathsf{eval\ n\ ve\ e2 = done\ (noerr\ v2)}$$

so the monadic sequencing in our assumption about eval lets us deduce

$$\mathsf{eval\ n\ (v2\ ::\ ve')\ e1' = done\ mv}$$

To conclude the proof, we want to apply the induction hypothesis again

$$\frac{\begin{array}{c}\mathsf{eval\ n\ (v2\ ::\ ve')\ e1' = done\ mv} \\ \mathsf{ExpTyp\ (t1\ ::\ te')\ e1'\ t2} \qquad \mathsf{WfEnv\ (v2\ ::\ ve')\ (t1\ ::\ te')}\end{array}}{\exists\ \mathsf{v,\ mv = noerr\ v} \wedge \mathsf{ValTyp\ v\ t}}\ \text{IH}$$

but we are still missing the well-formedness of the extended environment. We derive this last missing piece by

$$\frac{\mathsf{WfEnv\ ve'\ te'} \qquad \mathsf{ValTyp\ v2\ t1}}{\mathsf{WfEnv\ (v2\ ::\ ve')\ (t1\ ::\ te')}}\ \textsc{fa2\_cons}$$

□

3.7 Variable Representations

Allthough we used DeBruijn Indices to model variables, the proof of the soundness theorem has exactly the same structure for DeBruijn Levels and named variables. The only difference concerns the sublemmas of Lemma 2.4 (fa2_indexr), which for DeBruijn levels require additional lemmas to compensate for missing definitional equalities.

The reader is encouraged to compare the Coq formalizations via the `diff` tool for more information:

– the file `Chap_3_STLC_VarIndices.v` uses DeBruijn Indicies;

– the file `Chap_3_STLC_VarLevels.v` uses DeBruijn Levels; and

– the file `Chap_3_STLC_VarNames.v` uses explicit names in binders.

Chapter 4

Subtyping

Subtyping introduces a binary relation $\sqsubseteq$ between types, such that if $t \sqsubseteq t'$, then any expression of type t can also be given type t'.

Subtyping is characteristically used in object-oriented languages, where it plays a central part of class inheritance. For example, if a class Circle inherits from a class Shape, then Circle is also considered a subtype of Shape, which allows Circles to be used in place of Shapes, e.g. in Java

```
Shape s = new Circle();
```

In this chapter, the formalization of the simply typed lambda calculus from Chapter 3 is extended with subtyping. For a minimalistic scenario, the types are only extended by the top type - the common supertype of all other types.

4.1 Syntax

The extension to the syntax is straightforward. All we need is to add a new type t_top to the type syntax:

```
Inductive Typ : Type :=
| t_top : Typ
| t_arr : Typ → Typ → Typ.

Inductive Exp : Type :=
| e_var : ℕ → Exp
| e_app : Exp → Exp → Exp
| e_abs : Exp → Exp.
```

4.2 Type System

To extend the type system, we first need to define the subtyping relation:

```
Inductive ExpSubTyp : Typ → Typ → Prop :=
| est_top :
    ∀ t,
    ExpSubTyp t t_top
```

```
| est_arr :
    ∀ t11 t12 t21 t22,
    ExpSubTyp t21 t11 →
    ExpSubTyp t12 t22 →
    ExpSubTyp (t_arr t11 t12) (t_arr t21 t22).
```

- the est_top constructor states that any type t is a subtype of t_top;
- the est_arr constructor states that a function type t_arr t11 t12 is the subtype of another function type t_arr t21 t22, if t21 is a subtype of t11, and t12 is a subtype of t22.

We then extend the typing relation by adding a new constructor for subtyping:

```
Definition TypEnv := List Typ.

Inductive ExpTyp : TypEnv → Exp → Typ → Prop :=
| et_var :
    ∀ x te t1,
    indexr x te = some t1 →
    ExpTyp te (e_var x) t1
| et_app :
    ∀ te e1 e2 t1 t2,
    ExpTyp te e1 (t_arr t1 t2) →
    ExpTyp te e2 t1 →
    ExpTyp te (e_app e1 e2) t2
| et_abs :
    ∀ te e t1 t2,
    ExpTyp (t1 :: te) e t2 →
    ExpTyp te (e_abs e) (t_arr t1 t2)
| et_sub :
    ∀ te e t1 t2,
    ExpTyp te e t1 →
    ExpSubTyp t1 t2 →
    ExpTyp te e t2 .
```

The et_sub constructor states that subtyping preserves the typing relation, i.e. that if an expression e has type t1, and t1 is a subtype of t2, then e has also type t2.

4.3 Semantics

The semantics is precisely the same as for the STLC from Chapter 3:

```
Inductive Val : Type :=
| v_abs : List Val → Exp → Val.

Definition ValEnv := List Val.

Fixpoint eval (n : ℕ) (ve : ValEnv) (e : Exp) : CanTimeout (CanErr Val)
    :=
  match n with
  | 0 ⇒ timeout
  | S n ⇒
      match e with
      | e_var x ⇒ done (indexr x ve)
      | e_abs e ⇒ done (noerr (v_abs ve e))
      | e_app e1 e2 ⇒
          ' v_abs ve1' e1' ← eval n ve e1;
          ' v2 ← eval n ve e2;
          eval n (v2 :: ve1') e1'
      end
  end.
```

4.4 Type Soundness

As subtyping allows expressions to be evaluated to values, which have a subtype of the expression's type, we extend the value typing, such that closures now not only can have their arrow type `t_arr t1 t2`, but also any larger type `t`:

```
Inductive ValTyp : Val → Typ → Prop :=
| vt_abs :
    ∀ ve te e t1 t2 t ,
    Forall2 ValTyp ve te →
    ExpTyp (t1 :: te) e t2 →
    ExpSubTyp (t_arr t1 t2) t →
    ValTyp (v_abs ve e) t .

Definition WfEnv : ValEnv → TypEnv → Prop :=
  Forall2 ValTyp.
```

The statement of the actual soundness theorem stays the same:

Theorem (Type Soundness)**.**

```
∀ n e te ve res t,
eval n ve e = some res →
ExpTyp te e t →
WfEnv ve te →
∃ v, res = some v ∧ ValTyp v t.
```

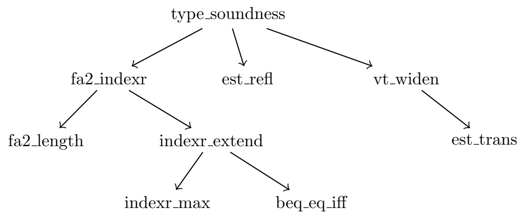

Figure 4.1: Proof Graph for $\text{STLC}_{<:}$ Soundness

4.5 Type Soundness Proof

Figure 4.1 shows the proof graph for the type soundness theorem. The only dependencies not covered in the framework from Chapter 2 are:

- est_refl and est_trans, which state the reflexivity and transitivity of the subtyping relation; and
- vt_widen, which states that if a value v has a type t, then v has also any supertype of t.

We start with the type soundness proof to motivate the lemmas:

Theorem 4.1 (Type Soundness)**.**

```
∀ n e te ve res t,
eval n ve e = some res →
ExpTyp te e t →
WfEnv ve te →
∃ v, res = some v ∧ ValTyp v t.
```

Proof. We start by induction over the number of steps n:

- **Case 0.** Contradiction; same as for the STLC.
- **Case n + 1.** In contrast to the STLC, we proceed by induction over the typing derivation ExpTyp te e t instead of simple case analysis, as we need the induction hypothesis for the new subtyping rule.
  - **Case** et_var**.** Same as for the STLC.
  - **Case** et_abs**.** Same as for the STLC, except that to prove ValTyp v (t_arr t1 t2) in the last step, we need an additional assumption about subtyping, as the vt_abs constructor changed:

    ```
    WfEnv ve te      ExpTyp (t1 :: te) e' t2
    ExpSubTyp (t_arr t1 t2) (t_arr t1 t2)
    ----------------------------------------- VT_ABS
    ValTyp (v_abs ve e') (t_arr t1 t2)
    ```

    This assumption is a special case of the reflexivity of subtyping proved in Lemma 4.2 (est_refl).
  - **Case** et_app**.** Same as for the STLC, except that the inversion of the closure's value typing now doesn't give us

    ```
    ExpTyp (t1 :: te') e1' t2
    ```

    but instead some t1', t2' such that

ExpTyp (t1' :: te') e1' t2' ∧
ExpSubTyp t1 t1' ∧
ExpSubTyp t2' t2

This leads to problems in the last proof step, where we want to proof well-formedness of the extended closure environment:

$$\frac{\text{WfEnv ve' te'} \qquad \text{ValTyp v2 } \color{red}{\text{t1'}}}{\text{WfEnv (v2 :: ve') (} \color{red}{\text{t1'}} \text{ :: te')}} \text{ FA2\_CONS}$$

Due to the subtyping, the environment is now extended by a subtype t1' of t1 instead of t1 itself. This in turn requires us to proof ValTyp v2 t1' instead of just ValTyp v2 t1, which we would have already known. We introduce Lemma 4.1 (vt_widen) to show that

$$\frac{\text{ValTyp v2 t1} \qquad \text{ExpSubTyp t1 t1'}}{\text{ValTyp v2 t1'}} \text{ VT\_WIDEN}$$

– **Case** et_sub**.** By definition of et_sub, we have some t' such that

ExpSubTyp t' t   ExpTyp te e t'

Our goal is to show

∃ v : Val, mv = noerr v ∧ ValTyp v t

We apply the inner induction hypothesis to our assumptions:

$$\frac{\text{eval (S n) ve e = done mv} \qquad \text{WfEnv ve te}}{\exists \text{ v, mv = noerr v} \wedge \text{ValTyp v t'}} \text{ IH'}$$

We conclude by using Lemma 4.1 (vt_widen):

$$\frac{\text{ValTyp v t'} \qquad \text{ExpSubTyp t' t}}{\text{ValTyp v t}} \text{ VT\_WIDEN}$$

□

We first prove that value typing is preserved under subtyping:

Lemma 4.1 (vt_widen)**.**

```
∀ v t1 t2,
ValTyp v t1 →
ExpSubTyp t1 t2 →
ValTyp v t2.
```

Proof. By inverting and reassembling the value typing using Lemma 4.3 (est_trans) to extend the contained subtyping relations. □

We then prove that subtyping is both reflexive and transitive. As the proofs are not specific to the definitional interpreter semantics, we only outline them.

Lemma 4.2 (est_refl)**.**

```
∀ t,
ExpSubTyp t t.
```

Proof. Straightforward induction over the type t. □

Lemma 4.3 (est_trans)**.**

```
∀ t1 t2 t3,
ExpSubTyp t1 t2 →
ExpSubTyp t2 t3 →
ExpSubTyp t1 t3.
```

Proof. Straightforward induction over the sum of the sizes of both subtyping derivation trees. □

Chapter 5

Substructural Types

Substructural type systems impose restrictions on how often variables are allowed to be used.

The most common classes of substructural type systems are

- *unrestricted*, allowing variables to be used arbitrarily often;
- *linear*, requiring variables to be used exactly once;
- *affine*, requiring variables to be used at most once; and
- *relevant*, requiring variables to be used at least once.

Those restrictions, especially linear and affine types, turn out to be useful in a variety of API's, where certain steps of a protocol are not allowed to happen multiple times, e.g. freeing memory, closing of a file handle, etc.

Rust is probably the most famous example of a real world language employing substructural typing. In Rust, affine types are used to model ownership, and unrestricted types are used for references[17].

In this chapter, the formalization of the simply typed lambda calculus from Chapter 3 is extended with substructural typing, such that both unrestricted and affine lambda abstractions are possible.

5.1 Syntax

The syntax extension is straightforward: both t_arr and e_abs are annotated by a multiplicity Mul, which can be either affine or unrestricted.

```
Inductive Mul : Type :=
| aff  :  Mul
| unr  :  Mul.

Inductive Typ : Type :=
| t_void  :  Typ
| t_arr   :  Mul → Typ → Typ → Typ.

Inductive Exp : Type :=
| e_var  :  ℕ → Exp
```

```
| e_app : Exp → Exp → Exp
| e_abs : Mul → Exp.
```

5.2 Type System

The type system extensions are relatively subtle, as the form of the typing relation remains the same, and we have no new syntactic forms to care for.

However, the restriction on variable usage raises two new concerns:

– when typing applications e_app e1 e2, then it is no longer correct to simply propagate the type environment to both sub-expressions, as this would allow both e1 and e2 to make use of the same variable that might be restricted.

– when typing unrestricted abstractions e_abs unr e, then it is no longer correct to simply capture the whole environment, as the environment may contain restricted variables, which may be used multiple times, as the unrestricted abstraction is allowed to be called multiple times.

To cover the first concern, we introduce the Split ting of type environments, such that the et_app constructor can be stated as

```
| et_app :
    ∀ te te1 te2 e1 e2 t1 t2 m,
    Split te te1 te2 →
    ExpTyp te1 e1 (t_arr m t1 t2) →
    ExpTyp te2 e2 t1 →
    ExpTyp te (e_app e1 e2) t2
```

To cover the second concern, we introduce a restrict function that removes all affine variables from the type environment.

We define Split and restrict, such that entries are not actually removed from the type environment, but rather marked as inaccessible. This greatly simplifies the proofs, as variables keep their meaning as DeBruijn Levels under splitting and restriction, and thus do not need to be renamed.

5.2.1 Type Environments

We define a type environment to be a list of types annotated with multiplicity and accessibility:

```
Inductive Acc : Type :=
| here : Acc
| gone : Acc.

Inductive Bind : Type :=
| bind : Acc → Mul → Typ → Bind.

Definition TypEnv := List Bind.
```

5.2.2 Splitting Type Environments

We define the splitting of the type environment as a tertiary relation between the input environment and two output environments:

```
Inductive Split : TypEnv → TypEnv → TypEnv → Prop :=
| sp_nil :
    Split [] [] []
| sp_gone :
    ∀ bs bs1 bs2 t m,
    Split bs bs1 bs2 →
    Split (bind gone m t :: bs)
          (bind gone m t :: bs1) (bind gone m t :: bs2)
| sp_left :
    ∀ bs bs1 bs2 t,
    Split bs bs1 bs2 →
    Split (bind here aff t :: bs)
          (bind here aff t :: bs1) (bind gone aff t :: bs2)
| sp_right :
    ∀ bs bs1 bs2 t,
    Split bs bs1 bs2 →
    Split (bind here aff t :: bs)
          (bind gone aff t :: bs1) (bind here aff t :: bs2)
| sp_both :
    ∀ bs bs1 bs2 t,
    Split bs bs1 bs2 →
    Split (bind here unr t :: bs)
          (bind here unr t :: bs1) (bind here unr t :: bs2).
```

- the sp_nil constructor states that the empty environment can be split into two empty environments;
- the sp_gone constructor states that if an entry is marked as gone, then it stays gone in both output environments;
- the sp_left and sp_right constructors state that if an entry is marked as affine, then it can be split into one of the output environments, but must be marked gone in the other; and
- the sp_both constructor states that if an entry is marked as unrestricted, then it may appear in both output environments.

5.2.3 Restricting Type Environments

We define the restriction of a type environment to simply mark all entries, that have affine multiplicity, as gone:

```
Definition restrict_entry (b : Bind) : Bind :=
  match b with
  | bind here aff t ⇒ bind gone aff t
  | b               ⇒ b
  end.
```

```
Definition restrict (m : Mul) (bs : TypEnv) : TypEnv :=
  match m with
  | unr ⇒ map restrict_entry bs
  | aff ⇒ bs
  end.
```

5.2.4 Typing Relation

We first define a kinding relation that relates types with their multiplicities:

```
Inductive TypKind : Typ → Mul → Prop :=
| tk_void : TypKind t_void unr
| tk_arr  : ∀ m t1 t2, TypKind (t_arr m t1 t2) m.
```

- the tk_void constructor states that the t_void type is of unrestricted kind; and
- the tk_arr constructor states that an arrow type t_arr m t1 t2 has the multiplicity of its annotation m as its kind.

We then extend the typing relation from the STLC as follows:

```
Inductive ExpTyp : TypEnv → Exp → Typ → Prop :=
| et_var :
    ∀ x te t m,
    indexr x te = some (bind here m t ) →
    ExpTyp te (e_var x) t
| et_app :
    ∀ te te1 te2 e1 e2 t1 t2 m,
    Split te te1 te2 →
    ExpTyp te1 e1 (t_arr m t1 t2) →
    ExpTyp te2 e2 t1 →
    ExpTyp te (e_app e1 e2) t2
| et_abs :
    ∀ te e t1 t2 m m1,
    TypKind t1 m1 →
    ExpTyp (bind here m1 t1 :: restrict m te ) e t2 →
    ExpTyp te (e_abs m e) (t_arr m t1 t2).
```

- the et_var constructor remains the same, except that the type environment entries now contain additional, irrelevant information;
- the et_app constructor now requires the type environment te to be split between both subderivations; and
- the et_abs constructor now forbids the use of affine variables in unrestricted abstractions by restricting the type environment accordingly.

5.3 Semantics

The semantics remains identical to the STLC, except that closure values now also carry their multiplicity:

```
Inductive Val : Type :=
| v_abs : List Val → Mul → Exp → Val.

Definition ValEnv := List Val.

Fixpoint eval (n : ℕ) (ve : ValEnv) (e : Exp) : CanTimeout (CanErr Val)
    :=
  match n with
  | 0 ⇒ timeout
  | S n ⇒
      match e with
      | e_var x ⇒ done (indexr x ve)
      | e_abs m e ⇒ done (noerr (v_abs ve m e))
      | e_app e1 e2 ⇒
          ' v_abs env1' m' e1' ← eval n ve e1;
          ' v2 ← eval n ve e2;
          eval n (v2 :: env1') e1'
      end
  end.
```

5.4 Type Soundness

The extension to the value typing is straightforward: closures and function types are now both annotated with multiplicities that have to match. As the structure of type environments has changed, we also need to make small changes to ignore the annotations, for which we define a function bind_typ that extracts the Typ of an annotated type environment entry.

```
 Definition bind_typ (b : Bind) : Typ :=
  match b with
  | bind a m t ⇒ t
  end.

Inductive ValTyp : Val → Typ → Prop :=
| vt_abs :
    ∀ ve te e t1 t2 m m1,
     Forall2 (λ v b ⇒ ValTyp v (bind_typ b)) ve te →
    TypKind t1 m1 →
    ExpTyp (bind here m1 t1 :: te) e t2 →
    ValTyp (v_abs ve m e) (t_arr m t1 t2).

Definition WfEnv : ValEnv → TypEnv → Prop :=
   Forall2 (λ v b ⇒ ValTyp v (bind_typ b)) .
```

The statement of type soundness remains unchanged:

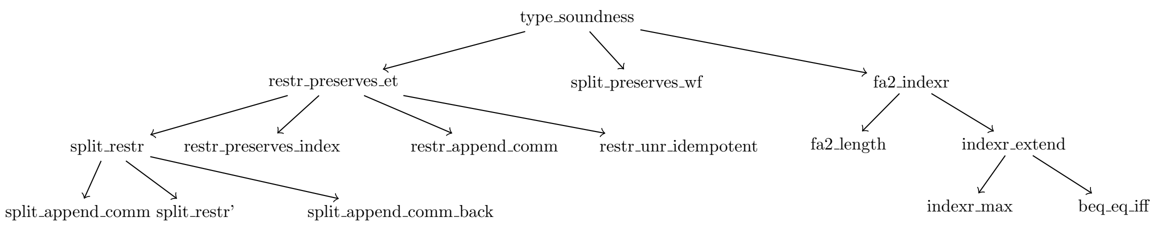

Figure 5.1: Proof Graph for STLC with Substructural Types Soundness

Theorem (Type Soundness)**.**

```
∀ n e te ve res t,
eval n ve e = some res →
ExpTyp te e t →
WfEnv ve te →
∃ v a, res = some v ∧ ValTyp v t.
```

5.5 Type Soundness Proof

Figure 5.1 shows the proof graph for the type soundness theorem. The only dependencies not covered in the framework from Chapter 2 are:

- split_preserves_wf, which is used in the e_abs case, and states that if well-formed environments WfEnv ve te are split, then both halves are again well-formed; and
- restr_preserves_et, which is used in the e_app case, and states that if an expression has a typing in an restricted type environment restrict te, then it has the same type in te.

We start with the type soundness proof to motivate the lemmas:

Theorem 5.1 (Type Soundness)**.**

```
∀ n e te ve res t,
eval n ve e = some res →
ExpTyp te e t →
WfEnv ve te →
∃ v a, res = some v ∧ ValTyp v t.
```

Proof. We start by induction over the number of steps n:

- **Case** 0**.** Contradiction; same as for the STLC.
- **Case** n + 1**.** We proceed by case analysis on the typing derivation ExpTyp te e t:
  - **Case** et_var**.** Same as for the STLC.
  - **Case** et_abs**.** In contrast to the STLC, the construction of the value typing with vt_abs now requires a proof for

$$\frac{\text{ExpTyp (bind here m1 t1 :: restrict m te) e t2}}{\text{ExpTyp (bind here m1 t1 :: te) e t2}}$$

    instead of having the conclusion already as an assumption.

This is due to the type system changes in et_abs.
We cover this case with Lemma 5.2 (restr_preserves_typing).
The rest of the proof remains the same.

– **Case et_app.** In contrast to the STLC, et_app now splits the type environment te between both subexpressions e1 and e2:

  Split te te1 te2   ExpTyp te1 e1 (t_arr t1 t2)   ExpTyp te2 e2 t1

  To apply the induction hypothesis to both subexpressions, we now need WfEnv evidence with respect to te1 and te2:

$$\frac{\text{eval n ve e1} = \text{done mv1} \quad \text{ExpTyp te1 e1 (t\_arr t1 t2)} \quad \text{WfEnv ve te1}}{\exists\ \text{v1, mv1} = \text{noerr v1} \wedge \text{ValTyp v1 (t\_arr t1 t2)}}\ \text{IH}$$

$$\frac{\text{eval n ve e2} = \text{done mv2} \quad \text{ExpTyp te2 e2 t1} \quad \text{WfEnv ve te2}}{\exists\ \text{v2, mv2} = \text{noerr v2} \wedge \text{ValTyp v2 t1}}\ \text{IH}$$

  We get the missing WfEnv evidence from Lemma 5.1 (split_preserves_wf).

$$\frac{\text{WfEnv ve te} \quad \text{Split te te1 te2}}{\text{WfEnv ve te1} \quad \text{WfEnv ve te2}}\ \text{SPLIT\_PRESERVES\_WF}$$

  The rest of the proof remains the same.

□

5.5.1 Splitting of Environments

Proving that environment splitting preserves well-formedness is simple, requiring no further sub-lemmas:

Lemma 5.1 (split_preserves_wf)**.**

```
∀ ve te te1 te2,
WfEnv ve te →
Split te te1 te2 →
WfEnv ve te1 ∧ WfEnv ve te2.
```

Proof. Straightforward induction over the environment splitting. □

5.5.2 Restricted Typing

While it is intuitively clear, that undeleting entries from the type environment preserves the typing relation, the mechanization requires 4 lemmas. As their proofs are not very interesting, we merely outline them for reference.

Lemma 5.2 (restr_preserves_typing)**.**

```
∀ m e t te te',
ExpTyp (te' ++ restrict m te) e t →
ExpTyp (te' ++ te) e t.
```

Proof. We start by case analysis on m:

- **Case** aff**.** Immediate, as restrict aff is just the identity.
- **Case** unr**.** We proceed by induction over the typing derivation:
  - **Case** et_var**.** Follows from Lemma 5.3 (restr_preserves_indexr).
  - **Case** et_abs**.** Follows from Lemma 5.4 (restr_unr_idempotent) and Lemma 5.5 (restr_append_comm).
  - **Case** et_app**.** Follows from Lemma 5.6 (split_restr). □

Lemma 5.3 (restr_preserves_indexr)**.**

```
∀ (te te' : TypEnv) (i : ℕ) m t ,
indexr i (te' ++ restrict unr te) = some (bind here m t) →
indexr i (te' ++ te) = some (bind here m t).
```

Proof. Straightforward induction over te'. □

Lemma 5.4 (restr_unr_idempotent)**.**

```
∀ (te : TypEnv) m,
 restrict m ( restrict unr te) = restrict unr te.
```

Proof. Case analysis on m, followed by induction on te in the affine case. □

Lemma 5.5 (restr_append_comm)**.**

```
∀ (te1 te2 : TypEnv) m,
 restrict m (te1 ++ te2) = restrict m te1 ++ restrict m te2.
```

Proof. Case analysis on m, followed by induction on te1 in the affine case. □

Lemma 5.6 (split_restr)**.**

```
∀ (i1 i2 l r : TypEnv),
Split (i1 ++ map restrict_entry i2) l r →
∃ l1 r1 l2 r2,
  Split (i1 ++ i2) (l1 ++ l2) (r1 ++ r2) ∧
  l1 ++ map restrict_entry l2 = l ∧
  r1 ++ map restrict_entry r2 = r.
```

Proof. We first apply Lemma 5.7 (split_append_comm), then Lemma 5.9 (split_restr), and finally Lemma 5.8 (split_append_comm_back). □

Lemma 5.7 (split_append_comm)**.**

```
∀ (i1 i2 l r : TypEnv),
Split (i1 ++ i2) l r →
∃ l1 r1 l2 r2,
  Split i1 l1 r1 ∧
  Split i2 l2 r2 ∧
  l1 ++ l2 = l ∧
  r1 ++ r2 = r.
```

Proof. Straightforward induction over te1. □

Lemma 5.8 (split_append_comm_back)**.**

```
∀ (i1 l1 r1 i2 l2 r2 : TypEnv),
 Split i1 l1 r1 →
 Split i2 l2 r2 →
 Split (i1 ++ i2) (l1 ++ l2) (r1 ++ r2).
```

Proof. Straightforward induction over te1. □

Lemma 5.9 (split_restr')**.**

```
∀ (te te1 te2 : TypEnv),
 Split (map restrict_entry te) te1 te2 →
∃ te1' te2',
   Split te te1' te2' ∧
   map restrict_entry te1' = te1 ∧
   map restrict_entry te2' = te2.
```

Proof. Straightforward induction over te. □

Chapter 6

Mutable References

In this chapter, the formalization of the simply typed lambda calculus from Chapter 3 is extended with mutable references.

This extension allows for the creation, observation, and mutation of so called locations, i.e. values representing references to other values. As such, mutable references are at the core of any imperative programming language.

6.1 Syntax

We extend the syntax as follow:

```
Inductive Typ : Type :=
| t_void  : Typ
| t_arr   : Typ → Typ → Typ
| t_unit  : Typ
| t_ref   : Typ → Typ.

Inductive Exp : Type :=
| e_var : ℕ → Exp
| e_app : Exp → Exp → Exp
| e_abs : Exp → Exp
| e_ref : Exp → Exp
| e_get : Exp → Exp
| e_set : Exp → Exp → Exp.
```

There are three new forms of expressions:

- e_ref e introduces a new reference to the value of e;
- e_get e retrieves the value of a reference e;
- e_set e1 e2 reassigns a reference e1 the value of e2.

There are two new forms of types:

- t_unit is the type with only a single inhabitant, and used as a return type for e_set; and
- t_ref t is the type of references to values of type t created through e_ref.

6.2 Type System

Extending the type system with mutable references is straightforward. The typing relation keeps its form as a tertiary relation between type environments, expressions, and types, and the rules for the old expressions remain the same. For each of the three new expressions, we add one new rule to the typing relation:

```
Definition TypEnv := List Typ.

Inductive ExpTyp : TypEnv → Exp → Typ → Prop :=
| et_var :
    ∀ x te t1,
    indexr x te = some t1 →
    ExpTyp te (e_var x) t1
| et_app :
    ∀ te e1 e2 t1 t2,
    ExpTyp te e1 (t_arr t1 t2) →
    ExpTyp te e2 t1 →
    ExpTyp te (e_app e1 e2) t2
| et_abs :
    ∀ te e t1 t2,
    ExpTyp (t1 :: te) e t2 →
    ExpTyp te (e_abs e) (t_arr t1 t2)
| et_ref :
    ∀ te e t,
    ExpTyp te e t →
    ExpTyp te (e_ref e) (t_ref t)
| et_get :
    ∀ te e t,
    ExpTyp te e (t_ref t) →
    ExpTyp te (e_get e) t
| et_set :
    ∀ te e1 e2 t,
    ExpTyp te e1 (t_ref t) →
    ExpTyp te e2 t →
    ExpTyp te (e_set e1 e2) t_unit .
```

- The et_ref constructor states that if an expression e has type t, then the reference e_ref e to the value of e has type t_ref t.
- The et_get constructor states that if an expression e has type t_ref t, then extracting the referenced value via e_get e has type t.
- The et_set constructor states that if an expression e1 has type t_ref t, and an expression e2 has type t, then updating the reference value from e1 to point to the value from e2 via e_set e1 e2 has type Unit, i.e. is welltyped, but does not return any interesting result, as all that is supposed to happen is the sideeffect of the store update.

6.3 Semantics

We start by adding two new forms of values:

```
Inductive Val :=
| v_abs  : List Val → Exp → Val
| v_unit : Val
| v_loc  : ℕ → Val.
```

– v_unit is the single inhabitant of the t_unit type; and

– v_loc n represents the reference cell created by the n-th use of e_ref.

To evaluate an expression e_get e, where e evaluates to some location v_loc n, we need to be able to access the value referenced by that location. Hence, we parameterize the semantics with a so called value store, that records the values referenced by location values. Analogously to value environments, we represent that store as a list of values indexed by their location:

```
Definition ValEnv := List Val.
Definition ValStore := List Val.
```

The definitional interpreter is extended with a value store as an additional argument and return value, allowing the value store to be threaded through the evaluation of subexpressions:

```
Fixpoint eval (n : ℕ) (ve : ValEnv) (vs : ValStore) (e : Exp) :
  CanTimeout (CanErr (Val ∗ ValStore ))
:=
  match n with
  | 0 ⇒
      timeout
  | S n ⇒
      match e with
      | e_var x ⇒
          done ( mmap (λ v ⇒ (v, vs)) (indexr x ve))
      | e_abs e ⇒
          done (noerr (v_abs ve e, vs ))
      | e_app e1 e2 ⇒
          '(v_abs ve1' e1', vs ) ← eval n ve vs e1;
          '(v2, vs ) ← eval n ve vs e2;
          eval n (v2 :: ve1') vs e1'
      | e_ref e ⇒
          '(v, vs) ← eval n ve vs e;
          done (noerr (v_loc (length vs), v :: vs))
      | e_get e ⇒
          '( v_loc l, vs) ← eval n ve vs e;
          done (mmap (λ v ⇒ (v, vs)) (indexr l vs))
      | e_set e1 e2 ⇒
          '( v_loc l, vs) ← eval n ve vs e1;
          '(v2, vs) ← eval n ve vs e2;
          done (noerr (v_unit , update l v2 vs))
```

```
    end
  end.
```

- the old cases are only adjusted to propagate the value store through the evaluation: in the e_var and e_abs cases, the value store is simply returned unmodified, and in the e_app case, the value store is threaded through the evaluation of both subexpressions;
- expressions of form e_ref e are evaluated by extending the value store by the value of e, and returning the location of that value. Recall, that we use right-indexing to access a value store vs, as we did for DeBruijn Levels, so the value v can be accessed by the largest list index length vs.
- expressions of form e_get e are evaluated by first evaluating e to some location v_loc l and new store vs, and then returning the value referenced by l.
- expressions of form e_set e1 e2 are evaluated by first evaluating e1 to some location v_loc l and e2 to some value v2, and then updating the store such that l references v2.

6.4 Type Soundness

We start by extending the value typing. To assign a type to a location v_loc l, we need to know the type of the value referenced by l.

For this purpose, we introduce a type store as a list of types, analogously to type environments:

```
Definition TypStore := List Typ.
```

We then extend the value typing as follows:

```
Inductive ValTyp : TypStore → Val → Typ → Prop :=
| vt_abs :
    ∀ ts ve te e t1 t2,
    Forall2 (ValTyp ts ) ve te →
    ExpTyp (t1 :: te) e t2 →
    ValTyp ts (v_abs ve e) ( t_arr t1 t2)
| vt_unit :
    ∀ ts,
    ValTyp ts v_unit t_unit
| vt_loc :
    ∀ ts l t,
    indexr l ts = some t →
    ValTyp ts (v_loc l) ( t_ref t).

Definition WfEnv (ve : ValEnv) (te : TypEnv) (ts : TypStore) : Prop :=
  Forall2 (ValTyp ts ) ve te.
```

- the vt_abs constructor remains independent from the type store, and only propagates the type store to the typing of the captured environment;

– the vt_unit constructor simply states that v_unit has type t_unit ; and

– the vt_loc constructor states that a location v_loc l has type t if the type store recorded that this is the case.

Next, we state a well-formedness relation between value stores and type stores, as we did with WfEnv for environments:

```
Definition WfStore (vs : ValStore) (ts : TypStore) : Prop :=
  Forall2 (ValTyp ts) vs ts.
```

As the type stores used in the value typing get larger during evaluation, we need to state when a type store ts1 is a substore of another type store ts2, in the sense that all locations present in ts1, have the same type in both ts1 and ts2. We define this relation simple as the list-suffix-relation from Chapter 2:

```
Notation SubStore := IsSuffixOf.
```

We are now equipped to state type soundness:

Theorem (Type Soundness)**.**

```
∀ n e te ve vs ts mv t,
eval n ve vs e = done mv →
ExpTyp te e t →
WfStore vs ts →
WfEnv ve te ts →
∃ v vs' ts',
  mv = noerr (v, vs') ∧
  WfStore vs' ts' ∧
  SubStore ts ts' ∧
  ValTyp ts' v t.
```

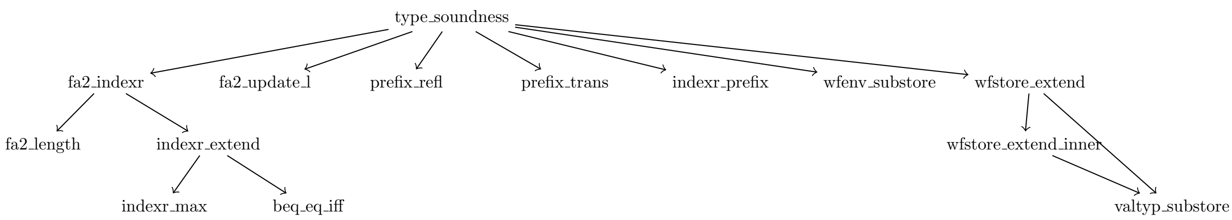

Figure 6.1: Proof Graph for STLC + Mutable References Soundness

6.5 Type Soundness Proof

Figure 6.1 shows the proof graph for the type soundness theorem. The only dependencies not covered in the framework from Chapter 2 are:

- wfenv_substore, which states that well-formed environments WfEnv ve te ts1 stay well-formed, if ts1 is replaced by a larger store ts2; and
- wfstore_extend, which states that a well-formed store WfStore vs ts can be extended by a value typing ValTyp ts v t to WfStore (v :: vs) (t :: ts).

We start with the type soundness proof to motivate the lemmas:

Theorem 6.1 (Type Soundness)**.**

```
∀ n e te ve vs ts mv t,
eval n ve vs e = done mv →
ExpTyp te e t →
WfStore vs ts →
WfEnv ve te ts →
∃ v vs' ts',
  mv = noerr (v, vs') ∧
  WfStore vs' ts' ∧
  SubStore ts ts' ∧
  ValTyp ts' v t.
```

Proof. We start by induction over the number of steps n:

- **Case** 0**.** Contradiction; same as for the STLC.
- **Case** n + 1**.** We proceed by case analysis on the typing derivation ExpTyp te e t:
  - **Case** et_var**.** By definition of et_var, we have some x such that

    e = e_var x   indexr x te = noerr t.

    As before, this allows us to apply Lemma 2.4 (fa2_indexr)

    $$\frac{\text{WfEnv ts ve te} \qquad \text{indexr x te = noerr t}}{\exists\ \text{v},\ \text{indexr x ve = noerr v} \wedge \text{ValTyp ts v t}}\ \text{FA2\_INDEXR}$$

    By definition of eval and mmap, and substitution of noerr v for indexr x ve, the assumption eval (n + 1) ve vs (e_var x) = done mv reduces to

    done (noerr (v, vs)) = done mv

    so we substitute for mv, instantiate the existential variables of our goal with v := v, vs' := vs, ts' := ts and are left to prove

WfStore vs ts ∧ SubStore ts ts ∧ ValTyp ts v t

The first and last conjuncts follow by assumption and as the conclusion from fa2_indexr. The second conjunct follows directly from Lemma 2.6 (suffix_refl).

- **Case** et_abs**.** Same as for the STLC. As in the et_var case, the value store doesn't change, so we use suffix_refl to establish Substore ts ts.

  By definition of et_abs, we have some e', t1, t2 such that

  e = e_abs e'   t = t_arr t1 t2   ExpTyp (t1 :: te) e' t2

  By definition of eval, the assumption eval (n + 1) ve vs (e_abs e') = done mv reduces to

  done (noerr (v_abs ve e', vs)) = done mv

  Thus, by substituting for mv, we are left to prove

  ∃ v, v_abs ve e' = v ∧ ValTyp v (t_arr t1 t2)

  so we choose v = v_abs ve e' and construct the value typing from our assumptions:

$$\frac{\text{WfEnv ve te} \qquad \text{ExpTyp (t1 :: te) e' t2}}{\text{ValTyp (v\_abs ve e') (t\_arr t1 t2)}}\ \text{VT\_ABS}$$

- **Case** et_app**.** By definition of et_app, we have some e1, e2, t1, t2 such that

  e = e_app e1 e2   t = t2   ExpTyp te e1 (t_arr t1 t2)   ExpTyp te e2 t1

  By definition of eval, the assumption

  eval (n + 1) ve vs (e_app e1 e2) = done mv

  reduces to

```
' (v_abs ve' e1', vs) ← eval n ve vs e1;
' (v2, vs) ← eval n ve vs e2;
eval n (v2 :: ve') vs e1'         = done mv
```

  As before, we observe that there must be some mv1 and mv2 such that

  eval n ve e1 = done mv1   eval n ve e2 = done mv2

  We are now equipped to apply our induction hypothesis to the evaluation of both subexpressions:

$$\frac{\begin{array}{c}\text{eval n ve vs e1 = done mv1}\\ \text{ExpTyp te e1 (t\_arr t1 t2)} \qquad \text{WfStore vs ts} \qquad \text{WfEnv ts ve te}\end{array}}{\begin{array}{c}\exists\ \text{v1 vs1 ts1,} \qquad \text{mv1 = noerr (v1, vs1)} \qquad \text{WfStore vs1 ts1}\\ \text{SubStore ts ts1} \qquad \text{ValTyp ts1 v1 (t\_arr t1 t2)}\end{array}}\ \text{IH}$$

$$\frac{\begin{array}{c}\text{eval n ve vs1 e2} = \text{done mv2}\\ \text{ExpTyp te e2 t1} \quad \text{WfStore vs1 ts1} \quad \text{WfEnv ts1 ve te}\end{array}}{\begin{array}{c}\exists\ \text{v2 vs2 ts2}, \quad \text{mv2} = \text{noerr (v2, vs2)} \quad \text{WfStore vs2 ts2}\\ \text{SubStore ts1 ts2} \quad \text{ValTyp ts2 v2 t2}\end{array}}\ \text{IH}$$

For the second application, we get the WfEnv ts1 ve te from WfEnv ts ve te and SubStore ts ts1 using Lemma 6.4 (wfenv_substore).

$$\frac{\text{WfEnv ts ve te} \quad \text{SubStore ts ts1}}{\text{WfEnv ts1 ve te}}\ \textsc{wfenv\_substore}$$

By inversion of the value typing ValTyp ts1 v1 (t_arr t1 t2), we find some te', ve', e1' such that

v1 = v_abs ve' e1'   ExpTyp (t1 :: te') e1' t2   WfEnv ve' te'

By substituting for mv1, mv2, and v1, we now know

eval n ve e1 = done (noerr (v_abs ve' e1'))
eval n ve e2 = done (noerr v2)

so the monadic sequencing in our assumption about eval lets us deduce

eval n (v2 :: ve') e1' = done mv

To conclude the proof, we want to apply the induction hypothesis again

$$\frac{\begin{array}{c}\text{eval n (v2 :: ve') e1'} = \text{done mv}\\ \text{ExpTyp (t1 :: te') e1' t2} \quad \text{WfEnv (v2 :: ve') (t1 :: te')}\end{array}}{\exists\ \text{v},\ \text{mv} = \text{noerr v} \wedge \text{ValTyp v t}}\ \text{IH}$$

but we are still missing the well-formedness of the extended environment. We derive this last missing piece by

$$\frac{\text{WfEnv ve' te'} \quad \text{ValTyp v2 t1}}{\text{WfEnv (v2 :: ve') (t1 :: te')}}\ \textsc{fa2\_cons}$$

- **Case et_ref.** By definition of et_ref, we have some e', t' such that

  e = e_ref e'   t = t_ref t'   ExpTyp te e' t'

  By definition of eval, the assumption

  eval (n + 1) ve vs (e_ref e') = done mv

  reduces to

  ' (v', vs') ← eval n ve vs e';
  done (noerr (v_loc (length vs'), v' :: vs')) = done mv

As before, we observe that there must be some mv' such that

```
eval n ve vs e' = done mv'
```

so we can apply the induction hypothesis as

$$\frac{\begin{array}{c}\text{eval n ve vs e' = done mv'} \\ \text{ExpTyp te e' t'} \quad \text{WfStore vs ts} \quad \text{WfEnv ts ve te}\end{array}}{\begin{array}{c}\exists\ \text{v' vs' ts',} \quad \text{mv' = noerr (v', vs')} \quad \text{WfStore vs' ts'} \\ \text{SubStore ts ts'} \quad \text{ValTyp ts' v' t'}\end{array}}\ \text{IH}$$

By substituting for mv', the assumption about evaluation further reduces to

```
done (noerr (v_loc (length vs'), v' :: vs')) = done mv
```

By substituting for mv, and instantiating the existential variables of our goal with v := v_loc (length vs'), vs' := v' :: vs', ts' := t' :: ts', we are left to prove

```
WfStore (v' :: vs') (t' :: ts') ∧
SubStore ts (t' :: ts') ∧
ValTyp (t' :: ts') v' (t_ref t')
```

The first conjunct is proved via Lemma 6.1 (wfstore_extend):

$$\frac{\text{WfStore vs' ts'} \quad \text{ValTyp ts' v' t'}}{\text{WfStore (v' :: vs') (t' :: ts')}}\ \text{WFSTORE\_EXTEND}$$

The second conjunct follows via Lemma 2.7 (suffix_trans) from the assumptions.
The third conjunct follows via Lemma 2.3 (fa2_length).

– **Case** et_get. By definition of et_ref, we have some e' such that

```
e = e_get e'                ExpTyp te e' (t_ref t)
```

By definition of eval, the assumption

```
eval (n + 1) ve vs (e_get e') = done mv
```

reduces to

```
' (v', vs') ← eval n ve vs e';
done (noerr (v_loc (length vs'), v' :: vs)) = done mv
```

As before, we observe that there must be some mv' such that

```
eval n ve vs e' = done mv'
```

so we can apply the induction hypothesis as

$$\frac{\begin{array}{c}\text{eval n ve vs e' = done mv'} \\ \text{ExpTyp te e' (t\_ref t)} \quad \text{WfStore vs ts} \quad \text{WfEnv ts ve te}\end{array}}{\begin{array}{c}\exists\ \text{v' vs' ts',} \quad \text{mv' = noerr (v', vs')} \quad \text{WfStore vs' ts'} \\ \text{SubStore ts ts'} \quad \text{ValTyp ts' v' (t\_ref t)}\end{array}}\ \text{IH}$$

By inversion of the value typing $\mathsf{ValTyp\ ts'\ v'\ (t\_ref\ \ t)}$, we find some n such that

$$\mathsf{v' = v\_loc\ n} \qquad\qquad \mathsf{indexr\ n\ ts' = some\ t}$$

We then apply Lemma 2.4 on the second result, yielding

$$\frac{\mathsf{WfStore\ vs'\ ts'} \qquad \mathsf{indexr\ n\ ts' = some\ t}}{\exists\ \mathsf{v},\ \mathsf{indexr\ n\ vs' = some\ v} \wedge \mathsf{ValTyp\ ts'\ v\ t}}\ \textsc{fa2\_indexr}$$

By substituting for v, the assumption about evaluation reduces further to

$$\mathsf{done\ (noerr\ (v,\ vs')) = done\ mv}$$

and after substituting and instantiating we are left to prove

$$\mathsf{WfStore\ vs'\ ts'} \wedge \mathsf{SubStore\ ts\ ts'} \wedge \mathsf{ValTyp\ ts'\ v\ t}$$

which we have already done.

- **Case** $\mathsf{et\_set}$. By definition of $\mathsf{et\_set}$, we have some $\mathsf{e1,\ e2,\ t'}$ such that

$$\mathsf{e = e\_set\ e1\ e2} \quad \mathsf{t = Unit} \quad \mathsf{ExpTyp\ te\ e1\ (t\_ref\ \ t')} \quad \mathsf{ExpTyp\ te\ e2\ t'}$$

By definition of eval, the assumption

$$\mathsf{eval\ (n+1)\ ve\ vs\ (e\_set\ e1\ e2) = done\ mv}$$

reduces to

```
' (v_loc l, vs) ← eval n ve vs e1;
' (v2, vs) ← eval n ve vs e2;
done (noerr (v_unit, update l v2 vs)) = done mv
```

As before, we observe that there must be some $\mathsf{mv1}$ and $\mathsf{mv2}$ such that

$$\mathsf{eval\ n\ ve\ vs\ e1 = done\ mv1} \qquad \mathsf{eval\ n\ ve\ vs\ e2 = done\ mv2}$$

We are now equipped to apply our induction hypothesis to the evaluation of both subexpressions:

$$\frac{\begin{array}{c}\mathsf{eval\ n\ ve\ vs\ e1 = done\ mv1}\\ \mathsf{ExpTyp\ te\ e1\ (t\_ref\ \ t')} \qquad \mathsf{WfStore\ vs\ ts} \qquad \mathsf{WfEnv\ ts\ ve\ te}\end{array}}{\begin{array}{c}\exists\ \mathsf{v1\ vs1\ ts1}, \qquad \mathsf{mv1 = noerr\ (v1,\ vs1)} \qquad \mathsf{WfStore\ vs1\ ts1}\\ \mathsf{SubStore\ ts\ ts1} \qquad \mathsf{ValTyp\ ts1\ v1\ (t\_ref\ \ t')}\end{array}}\ \mathrm{IH}$$

$$\frac{\begin{array}{c}\mathsf{eval\ n\ ve\ vs1\ e2 = done\ mv2}\\ \mathsf{ExpTyp\ te\ e2\ t'} \qquad \mathsf{WfStore\ vs1\ ts1} \qquad \mathsf{WfEnv\ ts1\ ve\ te}\end{array}}{\begin{array}{c}\exists\ \mathsf{v2\ vs2\ ts2}, \qquad \mathsf{mv2 = noerr\ (v2,\ vs2)} \qquad \mathsf{WfStore\ vs2\ ts2}\\ \mathsf{SubStore\ ts1\ ts2} \qquad \mathsf{ValTyp\ ts2\ v2\ t'}\end{array}}\ \mathrm{IH}$$

For the second application, we get the WfEnv ts1 ve te from WfEnv ts ve te and SubStore ts ts1 using Lemma 6.4 (wfenv_substore).

$$\frac{\mathsf{WfEnv\ ts\ ve\ te} \qquad \mathsf{SubStore\ ts\ \ ts1}}{\mathsf{WfEnv\ ts1\ ve\ te}}\ \textsc{wfenv\_substore}$$

By inversion of the value typing ValTyp ts1 v1 (t_ref t'), we find some l such that

v1 = v_loc l indexr l ts1 = some t'

By substituting for mv1, mv2, and v1, we now know

eval n ve e1 = done (noerr (v_loc l))
eval n ve e2 = done (noerr v2)

so the monadic sequencing in our assumption about eval lets us deduce

done (noerr (v_unit, update l v2 vs2)) = done mv

By substituting for mv and instantiating v := v_unit, vs' := update l v2 vs2, ts' := ts2, we are left to prove

WfStore (update l v2 vs2) ts2 ∧
SubStore ts ts2 ∧
ValTyp ts2 v_unit t_unit

To prove the first conjunct WfStore (update l v2 vs2) ts2, we use Lemma 2.5 (fa2_update_l) and Lemma 2.8 (indexr_suffix).

The second conjunct SubStore ts ts2 follows simply from Lemma 2.7 (suffix_trans) applied to SubStore ts ts1 and SubStore ts1 ts2 which resulted from the induction hypotheses.

The third conjunct ValTyp ts2 v_unit t_unit follows directly from the vt_unit constructor.

□

Lemma 6.1 (wfstore_extend)**.**

```
∀ (v : Val) (vs : ValStore) (t : Typ) (ts : TypStore),
WfStore vs ts →
ValTyp ts v t →
WfStore (v :: vs) (t :: ts).
```

Proof. Follows directly from Lemma 6.3 (valtype_substore) and Lemma 6.2 (wfstore_extend_inner). □

Lemma 6.2 (wfstore_extend_inner)**.**

```
∀ (ts ts' : TypStore) (vs : ValStore) (t : Typ),
  Forall2 (ValTyp ts') vs ts →
  Forall2 (ValTyp (t :: ts')) vs ts.
```

Proof. Straightforward induction over the Forall2 evidence using Lemma 6.3 (valtype_substore). □

The remaining two lemmas state that value typings ValTyp and well-formed environments WfEnv persist to hold for larger TypStores:

Lemma 6.3 (valtype_substore)**.**

```
∀ (v : Val) (t : Typ) (ts1 ts2 : TypStore),
ValTyp ts1 v t →
SubStore ts1 ts2 →
ValTyp ts2 v t.
```

Lemma 6.4 (wfenv_substore)**.**

```
∀ (te : TypEnv) (ve : ValEnv) (ts1 ts2 : TypStore),
WfEnv ve te ts1 →
SubStore ts1 ts2 →
WfEnv ve te ts2.
```

As ValTyp and WfEnv have a mutually inductive structure, we need to prove both lemmas together[1]:

Proof. Straightforward mutual induction over the WfEnv evidence from wfenv_substore together with the ValType evidence from valtype_substore . The case of a location value v_loc l requires Lemma 2.8 (indexr_suffix) from the framework. □

[1] In our Coq formalization, we were not able to derive the correct mutual induction schemes with a definition of WfEnv based on Forall2 . We worked around this issue by representing WfEnv with a more specialized, but structurally isomorphic type. See the implementation for more details.

Chapter 7

Parametric Polymorphism

In this chapter, the formalization of the simply typed lambda calculus from Chapter 3 is extended with parametric polymorphism resulting in a formalization of System F[12], also known as *the second-order lambda calculus* or *Girard-Reynolds polymorphic lambda calculus*.

Just as the simply typed lambda calculus allows to introduce variables ranging over values, the parametric polymorphism in System F allows to introduce variables ranging over types. For example, we can write a polymorphic identity function as

$$\Lambda\alpha.\lambda(x:\alpha).x \;:\; \forall\alpha.\alpha \to \alpha,$$

and instantiate it to a given type τ as

$$\big(\Lambda\alpha.\lambda(x:\alpha).x\big)[\tau] \;\equiv\; \lambda(x:\tau).x \;:\; \tau \to \tau.$$

While still being strongly normalizing, System F is much more expressive than the simply typed lambda calculus, allowing to encode many other language features[9].

The formalization of System F is significantly more complex than the other case studies we have seen so far. We specify the semantics of a type application $e[\tau]$ not by substituting τ for the type variable in e, but instead by pushing τ into the value environment, leaving the variable in e intact. As a consequence, we need to introduce a type equivalence, that relates types with respect to their value environments. For example, a type τ with respect to the empty environment is equivalent to a type variable α with respect to the environment that maps α to τ. Thus, the core lemmas of the soundness theorem are about the interaction of type equivalence with substitution used in the type system.

7.1 Syntax

The syntax of types is extended by universal quantification and variables. In our formalization, we represent type variables using a special form of the locally nameless encoding[2], that requires three different kinds of variables:

```
Inductive Typ : Type :=
| t_arr (t1 t2 : Typ)
| t_all (t : Typ)
| t_var_b (x : ℕ)
| t_var_c (x : ℕ)
| t_var_a (x : ℕ) .
```

- the t_all t is a universal type quantifying over a type variable in body t;
- the t_var_b variable represents a variable that's bound by a universal type;
- the t_var_c variable represents a free variable, caused by a type application;
- the t_var_a variable represents a free variable, that's used if the type equivalence relation goes under a binder.

The syntax of expressions is extended by two new forms:

```
Inductive Exp : Type :=
| e_var (x : ℕ)
| e_abs (e : Exp)
| e_app (e1 e2 : Exp)
| e_tabs (e : Exp)
| e_tapp (e : Exp) (t : Typ) .
```

- the e_tabs e expression represents a type abstraction with body e; and
- the e_tapp e t expression represents a type application of type t to expression e.

7.2 Type System

We start by adjusting the definition of type environments. Instead of assigning a type to a variable referring to a regular value, an entry in the type environment can now also state, that the variable refers to a type value, which itself has no type.

```
Inductive TypBind : Type :=
| bind_exp : Typ → TypBind
| bind_typ : TypBind.

Definition TypEnv := List TypBind.
```

Next, we define a relation HasVars, such that HasVars b a c t states that type t has at most b bound variables t_var_b that are not under a binder, at most a free variables t_var_a from the type equivalence relation, and at most c free variables t_var_c caused by type applications:

```
Inductive HasVars : ℕ → ℕ → ℕ → Typ → Prop :=
| hv_arr :
    ∀ b a c t1 t2,
    HasVars b a c t1 →
    HasVars b a c t2 →
    HasVars b a c (t_arr t1 t2)
| hv_all :
    ∀ b a c t2,
    HasVars (S b) a c t2 →
    HasVars b a c (t_all t2)
| hv_var_c :
    ∀ b a c x,
    c > x →
    HasVars b a c (t_var_c x)
| hv_var_a :
    ∀ b a c x,
    a > x →
    HasVars b a c (t_var_a x)
| hv_var_b :
    ∀ b a c x,
    b > x →
    HasVars b a c (t_var_b x).
```

To specify the typing of a type application `e_tapp e t'`, we need to substitute the type variable bound by the universal type of `e` with `t'`. For this purpose we define what it means to open a bound type variable `b'` with type `t'` in type `t`:

```
Fixpoint open_rec (b' : ℕ) (t' : Typ) (t : Typ) : Typ :=
  match t with
  | t_arr t1 t2 ⇒ t_arr (open_rec b' t' t1) (open_rec b' t' t2)
  | t_all t2    ⇒ t_all (open_rec (S b') t' t2)
  | t_var_c c   ⇒ t_var_c c
  | t_var_a a   ⇒ t_var_a a
  | t_var_b b   ⇒ if beq_nat b' b then t' else t_var_b b
  end.

Definition open t' t := open_rec 0 t' t.
```

The typing relation is then extended as follows:

```
Inductive ExpTyp : TypEnv → Exp → Typ → Prop :=
| et_var :
    ∀ x te t,
    HasVars 0 0 (length te) t →
    indexr x te = some (bind_exp t) →
    ExpTyp te (e_var x) t
| et_app :
    ∀ te e1 e2 t1 t2,
    ExpTyp te e1 (t_arr t1 t2) →
    ExpTyp te e2 t1 →
    ExpTyp te (e_app e1 e2) t2
| et_abs :
```

```
    ∀ te e t1 t2,
    HasVars 0 0 (length te) (t_arr t1 t2) →
    ExpTyp (bind_exp t1 :: te) e t2 →
    ExpTyp te (e_abs e) (t_arr t1 t2)
  | et_tapp :
    ∀ te e t1 t2,
    HasVars 0 0 (length te) t1 →
    ExpTyp te e (t_all t2) →
    ExpTyp te (e_tapp e t1) (open t1 t2)
  | et_tabs :
    ∀ te e t2,
    HasVars 0 0 (length te) (t_all t2) →
    ExpTyp (bind_typ :: te) e (open (t_var_c (length te)) t2) →
    ExpTyp te (e_tabs e) (t_all t2) .
```

- the old constructors remain the same, except that we require HasVars 0 0 (length te) t evidence at multiple places. The purpose of this evidence, is to exclude ill-formed types from the typing relation, that result from our variable encoding. The evidence ensures, that types have no bound variables that are not actually under any binder, and also no free variables related to type equivalence, as we are not in a situation, where the type equivalence has gone under a binder;
- the et_tapp constructor states that a type application e_tapp e t has type open t1 t2, if e has a universal type t_all t2, and t1 is a well-formed type, as witnessed by HasVars; and
- the et_tabs constructor states that a type abstraction e_tabs e t has type t_all t2, if t_all t2 is a well-formed type, i.e. t2 has only a single t_var_b that is not yet bound, and if its body e has the type of t2, where the yet unbound variable of t2 is opened by a free variable t_var_c .

7.3 Semantics

The extension to the semantics is straightforward. We have two new forms of values:

```
Inductive Val :=
| v_abs  : List Val → Exp → Val
| v_tabs : List Val → Exp → Val
| v_typ  : List Val → Typ → Val.
```

- a type abstraction closure v_tabs ve t results from evaluating a type abstraction, just like a regular closure results from evaluating a lambda abstraction; and
- a type closure v_typ ve t occurs in the evaluation of a type application, and represents a type t that may have free t_var_c occurences referring to other type closures in ve.

The value environment remains the same:

```
Definition ValEnv := List Val.
```

The definitional interpreter is extended by two new cases for the new expression forms:

```
Fixpoint eval (n : ℕ) (ve : ValEnv) (t : Exp) : CanTimeout (CanErr Val)
    :=
  match n with
  | 0 ⇒ timeout
  | S n ⇒
      match t with
      | e_var x ⇒ done (indexr x ve)
      | e_abs e ⇒ done (noerr (v_abs ve e))
      | e_tabs e ⇒ done (noerr (v_tabs ve e))
      | e_app e1 e2 ⇒
          ' v2 ← eval n ve e2;
          ' v_abs ve' e1' ← eval n ve e1;
          eval n (v2 :: ve') e1'
      | e_tapp e t ⇒
          ' v_tabs ve' e' ← eval n ve e;
          eval n (v_typ ve t :: ve') e'
      end
  end.
```

– a type abstraction e_tabs e is evaluated to a closure v_tabs ve e, just like a regular abstraction; and

– a type application e_tapp e t is evaluated by first evaluating e to a closure v_tabs ve' e', and then evaluating the closure's body e' in it's captured environment ve' extended by the argument type t closed in the current environment ve.

7.4 Type Soundness

As the type application puts the argument type as a type closure in the value environment, we need to define a type equivalence relation, which relates types with respect to their value environment. The type equivalence between universal types is defined in terms of the type equivalence of their bodys. For this purpose the bound variable is opened with a free variable t_var_a specific to the type equivalence relation. To count those variables, we introduce an environment AbsEnv as a list of Unit values:

```
Definition AbsEnv := List Unit.
```

We then state the type equivalence TEq, where TEq ve1 t1 ve2 t2 ae states that the type t1 is in value environment ve1 equivalent to type t2 in value environment ve2, where both types make use of at most length ae variables of form t_var_a . When we use the type equivalence outside of its own definition, we only need to compare types that have no t_var_a variables.

```
Inductive TEq : ValEnv → Typ → ValEnv → Typ → AbsEnv → Prop :=
| teq_arr :
    ∀ ve1 ve2 t1 t2 t1' t2' ae,
    TEq ve1 t1 ve2 t2 ae →
    TEq ve1 t1' ve2 t2' ae →
    TEq ve1 (t_arr t1 t1') ve2 (t_arr t2 t2') ae
| teq_all :
    ∀ ve1 ve2 t1 t2 x ae,
    x = length ae →
    HasVars 1 (length ae) (length ve1) t1 →
    HasVars 1 (length ae) (length ve2) t2 →
    TEq ve1 (open (t_var_a x) t1) ve2 (open (t_var_a x) t2) (tt :: ae)
        →
    TEq ve1 (t_all t1) ve2 (t_all t2) ae
| teq_var_c1 :
    ∀ ve1 ve2 ve1' t1' x t2 ae,
    indexr x ve1 = some (v_typ ve1' t1') →
    HasVars 0 0 (length ve1') t1' →
    TEq ve1' t1' ve2 t2 ae →
    TEq ve1 (t_var_c x) ve2 t2 ae
| teq_var_c2 :
    ∀ ve1 ve2 ve2' t2' x t1 ae,
    indexr x ve2 = some (v_typ ve2' t2') →
    HasVars 0 0 (length ve2') t2' →
    TEq ve1 t1 ve2' t2' ae →
    TEq ve1 t1 ve2 (t_var_c x) ae
| teq_var_c12 :
    ∀ ve1 ve2 v x1 x2 ae,
    indexr x1 ve1 = some v →
    indexr x2 ve2 = some v →
    TEq ve1 (t_var_c x1) ve2 (t_var_c x2) ae
| teq_var_a12 :
    ∀ ve1 ve2 x ae,
    indexr x ae = some tt →
    TEq ve1 (t_var_a x) ve2 (t_var_a x) ae.
```

- the teq_arr constructor states, that two arrow types are equivalent if their components are equivalent in the same environments;

- the teq_all constructor states, that two universal types are equivalent if their bodys are equivalent, after opening them with the same free variable t_var_a x, and extending the abstract environment ae by another unit value tt to witness the new free variable;

- the teq_var_c1 constructor states, that a free type variable t_var_c x in environment ve1, is equivalent to some other type t1 in environment ve2, if x is mapped to another type closure v_typ ve1' t1', that's equivalent to t1 in environment ve2;

- the teq_var_c2 constructor is symmetric to teq_var_c1;

- the teq_var_c12 constructor covers the case where both sides are variables of form teq_var_c. If the variables are syntactically equal, then no other evidence for equivalence is required; and
- the teq_var_a12 constructor is analogous to teq_var_c12, but for free variables introduced by the teq_all constructor. As those variables are abstract, i.e. do not relate to any concrete type from a value environment, syntactic equality is the only meaningful way to compare them.

For subtyping in Chapter 4, we extended the value typing, such that any value can be given any supertype of its actual type. For System F, we extend the value typing similarly, but with respect to type equivalence instead of subtyping. A value typing ValTyp ve v t now states that value v has a type that's equivalent to t in value environment ve. The additonal ve index of ValTyp prevents the use of Forall2 to model the well-formedness of value and type environments. We thus define WfEnv from scratch, together with ValTyp as mutually inductive types:

```
Inductive WfEnv : ValEnv → TypEnv → Prop :=
| wfe_nil :
    WfEnv nil nil
| wfe_cons :
    ∀ v t ve te,
    ValTyp (v :: ve) v t →
    WfEnv ve te →
    WfEnv (v :: ve) (t :: te)
with ValTyp : ValEnv → Val → TypBind → Prop :=
| vt_abs :
    ∀ ve1 ve2 te2 e t1 t2 t ,
    WfEnv ve2 te2 →
    ExpTyp ( bind_exp t1 :: te2) e t2 →
    TEq ve2 (t_arr t1 t2) ve1 t [] →
    ValTyp ve1 (v_abs ve2 e) (bind_exp t)
| vt_tabs :
    ∀ ve1 ve2 te2 e t2 t,
    WfEnv ve2 te2 →
    ExpTyp (bind_typ :: te2) e (open (t_var_c (length ve2)) t2) →
    TEq ve2 (t_all t2) ve1 t [] →
    ValTyp ve1 (v_tabs ve2 e) (bind_exp t)
| vt_ty :
    ∀ ve1 ve2 te2 t,
    WfEnv ve2 te2 →
    ValTyp ve1 (v_typ ve2 t) bind_typ .
```

- the vt_abs constructor previously stated, that a lambda closure v_abs ve2 e simply has the arrow type t_arr t1 t2 corresponding to its body. For System F, the arrow type t_arr t1 t2 may contain type variables referring to type closures in the captured value environment ve2. Hence, the closure can now be given any type t in value environment ve1, such that t in ve1 is equivalent to t_arr t1 t2 in ve2;

– the vt_tabs constructor states the typing of type abstraction closures v_tabs ve2 e. It is completely analogous to vt_abs, requiring the a typing of body e as stated by the type system; and

– the vt_typ constructor states a type closure v_typ ve2 t is well-formed, if it's captured value environment ve2 is well-formed with respect to some type environment te2.

The statement of type soundness remains unchanged:

Theorem (Type Soundness)**.**

```
∀ n e te ve mv t,
eval n ve e = done mv →
ExpTyp te e t →
WfEnv ve te →
∃ v, mv = noerr v ∧ ValTyp ve v (bind_exp t).
```

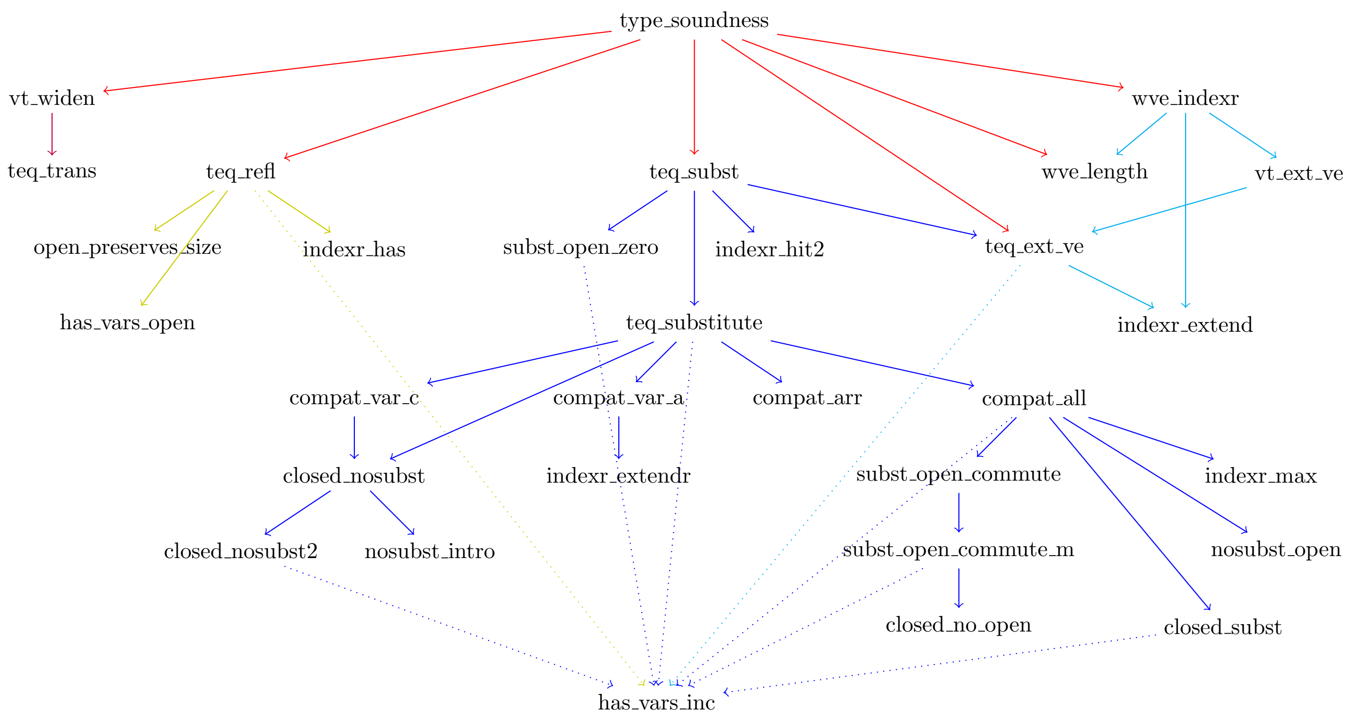

Figure 7.1: Proof Graph for System F Soundness

7.5 Type Soundness Proof

Figure 7.1 shows the proof graph of the type soundness theorem. As the full formal proof is rather lengthy, we only cover the theorem and its direct sublemmas in detail, and refer to the implementation for the complete proof:

– Similar to subtyping, we need a lemma vt_widen, that allows to transfer a value typing ValTyp ve v t along a type equivalence TEq ve t ve' t', yielding ValTyp ve' v t'. The lemma is used in the cases of lambda and type applications to relate the value typing from our goal to the value typing of the closure values produced by the induction hypothesis for the closure body.

– Similar to subtyping, we need a lemma teq_refl, that states the reflexivity of the type equivalence TEq. The lemma is used in the cases of lambda and type abstractions to build the value typing in the current environment.

– The teq_subst lemma is used in the type application case. It states the type equivalence between the direct type substitution performed by the type system and the delayed type substitution performed by the semantics through extending the value environment with a type closure. Proving this lemma requires a fair amount of extra machinery as witnessed by the proof graph.

- The teq_ext_ve lemma is used in the lambda application case. It states that type equivalence is preserved, if one of the involved value environments is extended by a new entry.
- The wve_indexr and wve_length lemmas are used in the variable case, and correspond to Lemma 2.4 (fa2_indexr) and Lemma 2.3 (fa2_length).

We start by proving the type soundness theorem:

Theorem 7.1 (Type Soundness)**.**

```
∀ n e te ve mv t,
eval n ve e = done mv →
ExpTyp te e t →
WfEnv ve te →
∃ v, mv = noerr v ∧ ValTyp ve v (bind_exp t).
```

Proof. We start by induction over the number of steps n:

- **Case 0.** Contradiction; same as for the STLC.
- **Case** n + 1**.** We proceed by case analysis on the typing derivation ExpTyp te e t:
  - **Case** et_var**.** Same as for the STLC.
  - **Case** et_abs**.** Same as for the STLC, except that to construct the value typing for the closure, we now have to proof reflexivity of type equivalence, similar as it was the case with subtyping in Chapter 4.
  - **Case** et_tabs**.** Same as the et_abs case.
  - **Case** et_app**.** Same as for the STLC, until we apply the induction hypothesis to both subexpressions e1 and e2, and then invert the resulting value typing ValTyp ve v1 (t_arr t1 t2).

    Whereas for the STLC, the inversion of the value typing revealed that the body e1' of the closure value is related directly to types t1 and t2:

    ```
    v1 = v_abs ve' e1'   WfEnv ve' te'   ExpTyp (t1 :: te') e1' t2
    ```

    It is now the case, that the body e1' relates to some t1' and t2', such that t_arr t1 t2 is in the current value environment ve equivalent to t_arr t1' t2' in the closure's value environment ve':

    ```
    v1 = v_abs ve' e1'   WfEnv ve' te'   ExpTyp (bind_exp t1' :: te') e1' t2'
    TEq ve (t_arr t1 t2) ve' (t_arr t1' t2') []
    ```

    Next, we apply the induction hypothesis to the closure body:

    $$\frac{\begin{array}{c}\mathsf{eval\ n\ (v2 :: ve')\ e1' = done\ mv}\\ \mathsf{WfEnv\ (v2 :: ve')\ (bind\_exp\ t1' :: te')}\\ \mathsf{ExpTyp\ (bind\_exp\ t1' :: te')\ e1'\ t2'}\end{array}}{\exists\ \mathsf{v,\ mv = noerr\ v} \land \mathsf{ValTyp\ (v2 :: ve')\ v\ (bind\_exp\ t2')}}\ \text{IH}$$

    We prove the missing WfEnv evidence in three steps:

* we use the wfe_cons constructor on the WfEnv ve' te' evidence from the closure inversion, requiring us to proof a value typing:

$$\frac{\mathsf{WfEnv\ ve'\ te'} \qquad \mathsf{ValTyp\ (v2\ ::\ ve')\ v2\ (bind\_exp\ t1')}}{\mathsf{WfEnv\ (v2\ ::\ ve')\ (bind\_exp\ t1'\ ::\ te')}}\ \text{WFE\_CONS}$$

* we proof the value typing using Lemma 7.1 (vt_widen) on the value typing that resulted from the induction hypothesis, requiring us to proof a type equivalence:

$$\frac{\mathsf{ValTyp\ ve\ v2\ (bind\_exp\ t1)} \qquad \mathsf{TEq\ ve\ t1\ (v2\ ::\ ve1)\ t1'\ []}}{\mathsf{ValTyp\ (v2\ ::\ ve')\ v2\ (bind\_exp\ t1')}}\ \text{VT\_WIDEN}$$

* we proof the type equivalence using Lemma 7.2 (teq_ext_ve) on the type equivalence we retrived from the closure inversion:

$$\frac{\mathsf{TEq\ ve\ t1\ ve1\ t1'\ []}}{\mathsf{TEq\ ve\ t1\ (v2\ ::\ ve1)\ t1'\ []}}\ \text{TEQ\_EXT\_VE}$$

In contrast to the STLC, the application of the induction hypothesis to the closure body didn't directly solve our goal

$$\exists\ \mathsf{v},\ \mathsf{mv} = \mathsf{noerr\ v} \wedge \mathsf{ValTyp\ ve\ v\ (bind\_exp\ t2)}$$

but instead produced

$$\exists\ \mathsf{v},\ \mathsf{mv} = \mathsf{noerr\ v} \wedge \mathsf{ValTyp\ (v2\ ::\ ve')\ v\ (bind\_exp\ t2')}$$

We use Lemma 7.1 (vt_widen) to instead proof that the types are equivalent in their environments, and Lemma 7.2 (teq_ext_ve) to build the type equivalence from a result of the closure inversion:

$$\frac{\mathsf{ValTyp\ ve\ v\ (bind\_exp\ t2)} \qquad \dfrac{\mathsf{TEq\ ve\ t2\ ve'\ t2'}}{\mathsf{TEq\ ve\ t2\ (v2\ ::\ ve')\ t2'}}\ \text{TEQ\_EXT\_VE}}{\mathsf{ValTyp\ (v2\ ::\ ve')\ v\ (bind\_exp\ t2')}}\ \text{VT\_WIDEN}$$

– **Case** et_tapp**.** By definition of et_tapp, we have some t1, t2, e1 such that

$$\mathsf{e = e\_tapp\ e1\ t} \quad \mathsf{t = open\ t1\ t2} \quad \mathsf{ExpTyp\ te\ e1\ (\ t\_all\ \ t2)}$$
$$\mathsf{HasVars\ 0\ 0\ (length\ \ te)\ t1.}$$

By definition of eval, the assumption

$$\mathsf{eval\ \ (n + 1)\ ve\ (e\_tapp\ e1\ t)} \quad = \mathsf{done\ mv}$$

reduces to

$$\mathsf{'\ \ v\_tabs\ \ ve'\ \ e'} \leftarrow \mathsf{eval\ n\ ve\ e1;}$$
$$\mathsf{eval\ \ n\ (v\_typ\ \ ve\ \ t\ \ ::\ \ ve')\ \ e'} \quad = \mathsf{done\ mv}$$

As before, we observe that there must be some mv1 such that

$$\text{eval n ve e1} = \text{done mv1}$$

and then apply our induction hypothesis accordingly

$$\frac{\begin{array}{c}\text{eval n ve e1} = \text{done mv1} \qquad \text{WfEnv ve te} \\ \text{ExpTyp te e1 (t\_all t2)}\end{array}}{\exists\ \text{v1, mv1} = \text{noerr v1} \land \text{ValTyp ve v1 (bind\_exp (t\_all t2))}}\ \text{IH}$$

By inversion of the value typing of v1, we find that v1 has to be a type abstraction closure with a body of type t2', such that t2' in the captured environment ve' is equivalent to t2 in the current environment ve, i.e. there are some te', ve', e1', t2' such that

$$\begin{array}{l}\text{v1} = \text{v\_tabs ve' e1'} \quad \text{WfEnv ve' te'} \quad \text{TEq ve' (t\_all t2') ve (t\_all t2)} \\ \text{ExpTyp (bind\_typ :: te') e1' (open (t\_var\_c (length ve')) t2')}\end{array}$$

By substituting for mv1, and v1, we find that

$$\text{eval n ve e1} = \text{done (noerr (v\_tabs ve' e1'))}$$

so the monadic sequencing in our assumption about eval lets us deduce

$$\text{eval n (v\_typ ve t :: ve') e1'} = \text{done mv}$$

We are now almost ready to apply the induction hypothesis to the closure body:

$$\frac{\begin{array}{c}\text{eval n (v\_typ ve t :: ve') e1'} = \text{done mv} \\ \text{ExpTyp (bind\_typ :: te') e1' (open (t\_var\_c (length ve')) t2')} \\ \text{WfEnv (v\_typ ve t :: ve') (bind\_typ :: te')}\end{array}}{\begin{array}{c}\exists\ \text{v, mv} = \text{noerr v} \land \\ \text{ValTyp (v\_typ ve t :: ve') v (open (t\_var\_c (length ve')) t2')}\end{array}}\ \text{IH}$$

all that's missing is the WfEnv evidence which we simply construct from our assumptions:

$$\frac{\text{WfEnv ve' te'} \qquad \dfrac{\text{WfEnv ve te}}{\text{ValTyp (v\_typ ve t :: ve') (v\_typ ve t) bind\_typ}}\ \text{VT\_TYP}}{\text{WfEnv (v\_typ ve t :: ve') (bind\_typ :: te')}}\ \text{WFENV\_CONS}$$

Whereas in the e_app case of the STLC, the application of the induction hypothesis to the closure body directly proved our goal, we now have a mismatch:

$$\frac{\text{ValTyp (v\_typ ve t :: ve') v (open (t\_var\_c (length ve')) t2')}}{\text{ValTyp ve v (bind\_exp (open t1 t2))}}\ ?$$

We use the vt_widen Lemma to instead proof the type equivalence

```
TEq (v_typ ve t1 :: ve1) (open (t_var_c (length ve1)) t2')
    ve (open t1 t2) []
```

which by the teq_subst Lemma requires only the type equivalence we already extracted from the closure's value typing

```
TEq ve' (t_all t2') ve (t_all t2)
```

□

Lemma 7.1 (vt_widen)**.**

```
∀ vf H1 H2 t1 t2,
ValTyp H1 vf (bind_exp t1) →
TEq H1 t1 H2 t2 [] →
ValTyp H2 vf (bind_exp t2).
```

Proof. Identical to the proof of Lemma 4.1 (vt_widen) for subtyping, but using transitivity of type equivalence instead of subtyping. □

Lemma 7.2 (teq_ext_ve)**.**

```
∀ (v : Val) (ve1 ve2 : ValEnv) (t1 t2 : Typ) ae,
TEq       ve1  t1       ve2  t2 ae →
TEq (v :: ve1) t1       ve2  t2 ae ∧
TEq       ve1  t1 (v :: ve2) t2 ae.
```

Proof. Straightforward induction over the TEq evidence, using 2 minor, technical lemmas. □

Lemma 7.3 (teq_refl)**.**

```
∀ ae (ve : ValEnv) (t : Typ),
HasVars 0 (length ae) (length ve) t →
TEq ve t ve t ae.
```

Proof. Induction over the size of t using minor, technical lemmas. □

Lemma 7.4 (teq_subst)**.**

```
∀ (t1 t2 t2' : Typ) (ve ve' : ValEnv),
HasVars 0 0 (length ve) t1 →
TEq ve (t_all t2)
    ve' (t_all t2') [] →
TEq ve                  (open t1 t2)
    (v_typ ve t1 :: ve') (open (t_var_c (length ve')) t2') [].
```

Proof. The main proof is by induction over the type equivalence, but a lot of auxiliary definitions and lemmas are required. □

Lemma 7.5 (wve_indexr)**.**

```
∀ ve te x t,
WfEnv ve te →
indexr x te = some t →
∃ v, indexr x ve = some v ∧ ValTyp ve v t.
```

Proof. Similar to Lemma 2.4 (fa2_indexr), but now the value typing retrieved for older variables, relates to a suffix of ve, so we use Lemma 7.2 (teq_ext_ve) to extend the value typing accordingly. □

Lemma 7.6 (wve_length)**.**

```
∀ ve te,
WfEnv ve te →
length ve = length te.
```

Proof. Identical to Lemma 2.3 (fa2_length), □

Bibliography

[1] N. Amin and R. Tate. Java and scala's type systems are unsound: the existential crisis of null pointers. *Acm Sigplan Notices*, 51(10):838–848, 2016.

[2] A. Charguéraud. The locally nameless representation. *Journal of automated reasoning*, 49(3):363–408, 2012.

[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] C. Dubois. Proving ml type soundness within coq. In *International Conference on Theorem Proving in Higher Order Logics*, pages 126–144. Springer, 2000.

[5] E. Ernst, K. Ostermann, and W. R. Cook. *A virtual class calculus*, volume 41. ACM, 2006.

[6] R. Harper. *Practical foundations for programming languages.* Cambridge University Press, 2016.

[7] R. Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[8] L. Osvald, G. Essertel, X. Wu, L. I. G. Alayón, and T. Rompf. Gentrification gone too far? affordable 2nd-class values for fun and (co-) effect. In *ACM SIGPLAN Notices*, volume 51, pages 234–251. ACM, 2016.

[9] B. C. Pierce. *Types and programming languages.* MIT press, 2002.

[10] B. C. Pierce. *Advanced topics in types and programming languages.* MIT press, 2005.

[11] B. C. Pierce, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjöberg, and B. Yorgey. Software foundations. *Webpage: https://softwarefoundations.cis.upenn.edu/*, 2010.

[12] J. C. Reynolds. Towards a theory of type structure. In *Programming Symposium*, pages 408–425. Springer, 1974.

[13] T. Rompf and N. Amin. From f to dot: Type soundness proofs with definitional interpreters. *arXiv preprint arXiv:1510.05216*, 2015.

[14] V. Saraswat. Java is not type-safe, 1997.

[15] J. Siek. Big-step, diverging, or stuck? https://siek.blogspot.com/2012/07/big-step-diverging-or-stuck.html, 2012. Accessed: 2019-08-09.

[16] J. Siek. Type safety in three easy lemmas. https://siek.blogspot.com/2013/05/type-safety-in-three-easy-lemmas.html, 2013. Accessed: 2019-08-09.

[17] A. Weiss, D. Patterson, N. D. Matsakis, and A. Ahmed. Oxide: The essence of rust. *arXiv preprint arXiv:1903.00982*, 2019.

[18] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and computation*, 115(1):38–94, 1994.