Formalizing Rust Traits

by

Jonatan Milewski

B. Sc., McGill University, 2013

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Computer Science)

The University of British Columbia
(Vancouver)

November 2015

Abstract

Rust is a new systems programming language designed with a focus on bare metal performance, safe concurrency and memory safety. It features a robust abstraction mechanism in the form of traits, which provide static overloading and dynamic dispatch. In this thesis, we present MiniRust—a formal model of a subset of Rust. The model focuses on the trait system and includes some advanced features of traits such as associated types and trait objects. In particular, we discuss the notion of object safety—the suitability of a particular trait for creating trait objects—and we formally determine very general conditions under which it can be guaranteed. To represent the runtime semantics of MiniRust programs, we develop an explicitly-typed internal language RustIn, for which we prove type safety, and we show that well-typed MiniRust programs can be translated to well-typed RustIn programs. Finally, we adapt the informally-described Rust trait coherence rules to our model and we show that they are sufficient to ensure that overloads are always well-determined, even in the presence of library extensions.

Preface

This thesis is original, unpublished, independent work by the author, Jonatan Milewski.

Table of Contents

List of Figures

Acknowledgments

First and foremost, I would like to thank my supervisor Ron Garcia for his guidance, support and encouragement throughout the research and writing process. His vast expertise is an invaluable resource and his ever-positive attitude helped me stay motivated to keep going. His thorough feedback was a crucial contribution to the writing process. Also, I would like to thank Reid Holmes for providing insightful comments and suggestions on this thesis.

Big thanks are also due to Niko Matsakis from Mozilla Research for his useful insights on the inner workings of Rust and to The Mozilla Corporation for funding this research. The knowledgeable Rust community and the plethora of Rust resources stemming from it were incredibly helpful as well.

Last but not least, I would like to thank my family and friends who provided indispensable emotional support and encouragement at times when they were most needed.

Chapter 1

Introduction

Modern programming languages present a high degree of complexity, incorporating advanced constructs and type system features. As such, reasoning about properties of a language, such as its static and dynamic semantics, is often quite challenging. Moreover, details of the language implementation, containing numerous optimizations driven by pragmatic concerns, can make it more difficult to understand what the intended semantics are.

A formal model of a programming language can abstract away its implementation details and allow us to focus our attention on its core. It can facilitate an in-depth study of specific aspects of the language and of the way in which they interact with one another. It can also be used to present complex—difficult to grasp—language features in terms of constructs and ideas that are already well understood. Moreover, a formal model can serve as a base for prototyping future extensions and exploring how they would interact with the rest of the language. Perhaps more importantly, the rigorous formal treatment and the availability of various mathematical tools let us obtain formal proofs of certain properties of the language—in a language that strives to provide certain guarantees to the programmer, it is particularly desirable to be able to prove that those guarantees do indeed hold. As such, a formalization may provide a justification for the decisions made by the language designers but it may also expose gaps in the design that were overlooked.

One important and commonly studied aspect of a given programming language is its type system. A type system can serve multiple purposes [27]. One obvious goal is to statically detect and report errors caused by code that would inevitably produce an undesired result at runtime. However, the range of errors caught by the type-checker—an implementation of the type system—varies greatly from language to language. Another goal of a type system is to provide the ability to create abstractions, which enable the developer to write code that is clearer, more concise and reusable. Such abstractions may include for instance the ability to define new datatypes and to employ *generics*, which allow us to write code that may be

used with values of different types. The type system can also protect such abstractions by ensuring that they behave as intended.

In commonly-used programming languages, the rules of the type system are seldom specified formally. In an effort to gain a good understanding of the type system, users of a language may refer to its documentation, which often provides a rather broad view of the system. For more detailed information, one would then have to refer to the implementation of the type-checker, where the various optimizations present in the code can make reasoning about the type system very difficult. A formal definition of the type system can strip those implementation details away and thus it facilitates reasoning about what constitutes a well-typed program. Combined with a model of the runtime semantics of the system, it allows us to understand how such programs evolve at runtime and it can be used to formally prove properties such as type safety—that the type system prevents the type errors that it is designed to prevent at runtime. More generally, we may use the formal model of the type system to prove that the abstractions provided by the language act as intended.

In this thesis we focus on the type system of the Rust programming language [6]. Rust is an open-source community project headed by Mozilla Research. It is also new, with version 1.0 released in May 2015, and many of its aspects have not yet been studied formally. Being a systems language, Rust is designed to be suitable for low-level programming requiring control, performance and predictability—properties desired in domains such as operating systems, embedded systems and browser engines (the browser engine Servo [8], written in Rust, is its primary real-world test case). As such, its primary areas of focus are memory safety, speed and concurrency. Many languages employ a runtime garbage collector to guarantee memory safety. Rust, however, does not have garbage collection, which gives the programmer low-level control over the memory stack and predictable runtime performance. The language relies instead on type system abstractions to ensure memory safety at compile-time without introducing runtime overhead. The relevant features of the type system include *lifetimes*, corresponding to *regions* in region-based memory management [39] [19], and the concept of *ownership* [13] with *move semantics*, having ties to *affine type systems* [29]. Finally, those type system features, along with other abstractions in the language enable language developers to add data structures and functions to the standard library that allow us to write concurrent programs guaranteed to avoid data races.

Despite being labeled a systems language, Rust includes a number of language abstractions similar to those found in other higher-level languages, including algebraic data types (called *enums* in Rust) and pattern matching. The language also supports type inference, which lets the programmer omit some type annotations that can be inferred statically. Last, but not least, the type system provides abstractions in the form of *generics* and *traits* that enable the programmer to write modular, reusable code. These abstractions introduce *type*

polymorphism in the language, where a single piece of code can be used with values of multiple types [36]. While generics are associated with *parametric polymorphism* (where a piece of polymorphic code behaves the same way regardless of the concrete type instantiation), traits, similar to type classes in Haskell [40], introduce another form of polymorphism called *ad-hoc polymorphism* or *overloading*, where the runtime semantics of polymorphic code vary depending on the concrete type(s) used.

In a nutshell, a trait is a way to abstract over types that share the same property. The property can be defined implicitly in a *marker trait* (a trait that does not provide any functionality) or explicitly through method signatures. Each type that wishes to implement a trait must implement that trait's declared interface. Traits also include *associated items*, including *associated types*—type-level functions similar to associated types in Haskell [12]—and they provide the ability to create *trait objects*. Trait-based overloads are normally resolved statically, so successful overload resolution is guaranteed at compile-time, which also enables the language implementers to avoid introducing runtime overhead through the use of traits. However, trait objects also allow *dynamic dispatch* of overloads and they provide a heterogeneous view on values implementing the same trait, regardless of those values' concrete types. All in all, traits are a powerful part of Rust's type system: they inform the design of *closures* (anonymous functions that enclose their surrounding environment), they are the mechanism that enables operator overloading and they form an integral part of Rust's safety story.

The main contribution of this thesis is a formalization of Rust's trait system. Specifically, we develop MiniRust: a formal model of a small subset of Rust's type system including traits. For compactness, we do not model the type system features related to memory safety but we include more advanced features of the trait ecosystem such as *supertraits*, *associated types* and *trait objects*, which allows us to gain insights into how those features interact with one another. To demonstrate the runtime semantics of programs with traits, we also develop an internal language, RustIn, which doesn't have traits. We present operational semantics for RustIn, we show how MiniRust programs can be translated to RustIn and we prove that well-typed MiniRust programs are type-safe. Finally, we adapt the Rust *trait coherence rules*, meant to prevent ambiguity of trait resolution in the presence of library dependencies. We prove that the rules are sufficient to satisfy the desired guarantees, which in turn brings us a step closer to ensuring full type system coherence—that a program will behave the same way regardless of the way it was typed.

The rest of this thesis is organized as follows: in the next chapter we provide a more detailed introduction to traits and how are they used in Rust, including the features that are part of the formalization. Then, in chapter 3 we delve into the formalization, starting with NanoRust: a formal model of a subset of the Rust type system including traits but omitting some of their more advanced features. We then present MicroRust in chapter 4, which

introduces associated types to NanoRust and finally MiniRust in chapter 5, which adds dynamically sized types and trait objects. In chapter 6 we present RustIn, the internal language for MiniRust. The presentation includes the language's type system, operational semantics and a proof of type safety. Chapter 7 ties MiniRust and RustIn together by presenting type-directed translation rules from MiniRust to RustIn along with a proof that the translation preserves well-typedness of programs. Then, in chapter 8 we discuss the notion of trait coherence along with a proof of certain properties of the coherence rules. Finally, in chapter 9 we discuss limitations of our work and possible extensions to the formalization, in chapter 10 we present related work, and in chapter 11 we conclude the thesis.

Chapter 2

Traits in Rust

This chapter presents an introduction to traits in Rust. Most of the material in this chapter is adapted from the official Rust book [7], where we refer the reader for more details.

2.1 Functions

We first present some basics of the Rust language that are relevant to our formalization.

A Rust program consists of a collection of *items*. Among others, items may include *function declarations*, *datatype declarations*, *traits* and *impls*. We first introduce function declarations.

Below is a simple Rust program:

```
fn square(x: f64) -> f64 {
    x * x
}

fn main() {
    let y = 1.2;
    let y_squared = square(y);
    println!("The square of {} is {}.", y, y_squared);
}
```

The program computes the square of 1.2 and prints it to standard output. At the top of the program is a declaration of the function `square`. Function declarations are explicitly typed in Rust: the types of a function's arguments and return value must be provided by the programmer. The *function signature* `fn square(x: f64) -> f64` contains the name of the function, the name and type of its argument `x: f64` and the function's return type on the right side of the arrow (`-> f64`). `f64` is one of the primitive types in Rust—it ranges over 64-bit floating point numbers. Other primitive types include signed and unsigned integers of

different sizes (`i8`, `u16`, etc.), characters `char`, booleans `bool` and the unit type `()`, which is associated with the unit expression `()`. The body of `square` is just a single expression. The return value of the function is the result of evaluating that expression, which in this case is the square of `x`.

The starting point of a Rust program is the `main` function. Its function signature is equivalent to `fn main() -> ()`, since the last statement in the function's body can be considered as an expression that returns the unit value (the absence of a return type in the function signature is syntactic sugar for `-> ()`).

The `let` keyword is used to declare new local variables. Local variable declarations do not require explicit type annotations in most cases, as long as the compiler has enough information to infer their types.

2.2 Structs and pointers

Rust also allows creating user-defined datatypes. For instance, a *struct* is a composite data type with named fields. A *struct declaration* declares the name of a struct along with its fields and their respective types:

```
struct Point {
    x: i32,
    y: i32
}

fn get_x(p: &Point) -> i32 {
    p.x
}

fn main() {
    let my_point = Point { x: 4, y: 6 };
    let my_x = get_x(&my_point);
    assert_eq!(my_x, 4);
}
```

In the above program, we first define a new struct `Point`, which consists of two 32-bit integer fields `x` and `y`. Having defined the struct, we can use `Point` as a type. In the function signature of `get_x`, the type annotation `&Point` signals that the argument `p` is a *reference* (the basic kind of pointer in Rust) to a `Point`. In the function's body, the dot operator in `p.x` is used to access the field `x` of `p`. The dot operator also automatically dereferences the receiver as necessary. Therefore, since `p` is a pointer to a `Point`, `p.x` is equivalent to `(*p).x`, where the `*` operator first dereferences `p` before its `x` field is accessed.

In the `main` function in the example above we create a new instance of `Point` and we assign it to the local variable `my_point`. We then pass a reference to `my_point` (denoted by the reference operator `&`) to `get_x` and we use the `assert_eq!` macro to assert that the call to `get_x` returns the expected result of 4 (the symbol `!` denotes macros in Rust).

2.3 Traits

In its simplest form, a *trait* is a collection of method interfaces parameterized by an implicit `Self` type variable. An *impl* contains implementations of all of the trait's methods for some specific type instantiation of `Self`.

```
struct Point {
    x: i32,
    y: i32
}

trait Eq {
    fn eq(&self, &Self) -> bool;
}

impl Eq for Point {
    fn eq(&self, other: &Point) -> bool {
        self.x == other.x && self.y == other.y
    }
}
```

In the above example, we define the trait `Eq` [1], whose body consists of the type signature of method `eq`, which checks for equality between two values of type `Self` (note that the method receives pointers to those values and not the values themselves). Since `eq` is a method, the first argument in its type signature, `&self`, stands for the receiver of the method and it has type `&Self` (`self` is a special keyword used for the method receiver of type `Self`). Then, we have an impl of `Eq` for our new type `Point`, where we must provide an implementation of `eq`, with all instances of the type variable `Self` replaced with `Point` (`&self` in the function signature is equivalent to `self: &Point`).

The trait declaration allows us to call `eq` with values of any type that implements `Eq`, which includes `Point`:

```
let (p1, p2) = (Point {x: 3, y: 4}, Point {x: 4, y: 5});
let are_equal = p1.eq(&p2);
```

[1] Our `Eq` trait is a simplified version of the equivalence relation traits in the standard library, which are used to overload the == operator: https://doc.rust-lang.org/stable/std/cmp/

The dot operator in the above example is used to call the method `eq` on the receiver `p1`. When using the method call notation, the receiver is automatically referenced when needed, so the last line is equivalent to

```
let are_equal = (&p1).eq(&p2);
```

Note that the call to `eq` is dispatched statically: the concrete implementation of `eq` used in the above call is determined at compile-time.

We can also constrain type parameters of generic functions by traits:

```
fn triple_equal<T>(first: &T, second: &T, third: &T) -> bool
    where T: Eq {
    first.eq(second) && second.eq(third)
}
```

In the first line of the function signature of `triple_equal` we learn that the function is generic over type variable `T`. The *where-clause* on the second line contains a constraint `T: Eq`, which means "`T` implements `Eq`" and restricts the types allowed to instantiate `T` to only those that implement `Eq`. In consequence, we are able to use this additional information about `T` in the body of the function. Concretely, we gain access to the functionality provided by `Eq` and we can call the trait's `eq` method on a reference to a value of type `T`.

Impls themselves can also be generic and constrained as in the following example:

```
struct Pair<T> {
    first: T,
    second: T
}

impl<T> Eq for Pair<T> where T: Eq {
    fn equals(&self, other: &Pair<T>) -> bool {
        self.first.equals(other.first) &&
        self.second.equals(other.second)
    }
}
```

In the above example, `Pair` is a generic `struct` used to create pairs of values of any type (as long as the type is the same for both values). The corresponding impl is generic over the type `T` of the values in the pair. The where-clause restricts `T` to be instantiable only with a type that implements `Eq`. The constraint in the where-clause is then available to the functions in the impl's body.

2.4 Associated functions & UFCS

In Rust, the body of a trait may also include *associated functions*, which do not require `self` (or a pointer to `self`) to appear as the first argument in their type signature. In consequence,

associated functions cannot be invoked using method syntax. A call to an associated function must be prefixed by the name of the trait, using the so-called *universal function call syntax* (UFCS). For example, the Rust standard library contains a trait `Default`, used for types that have a default value:

```
trait Default {
    fn default() -> Self;
}
```

The type signature of `default` does not include `self` as its first argument. Thus, `default` is an associated function and it can only be called using UFCS. The concrete type, whose impl of `Default` we would like to use, can appear in the function call prefix:

```
let x = <i32 as Default>::default();
```

or it can be omitted as long as the surrounding context contains enough type information:

```
let y: i32 = Default::default();
```

Trait methods can also be called using UFCS. The receiver of the method is then simply passed as the first argument to the function call.

2.5 Supertraits

Traits in Rust allow a form of interface inheritance through *supertraits*:

```
trait Ord: Eq {
    fn lt(&self, &Self) -> bool;
    fn le(&self, &Self) -> bool;
}

impl Ord for Point {
    fn lt(&self, other: &Point) -> bool {
        self.x < other.x ||
        (self.x == other.x && self.y < other.y)
    }
    fn le(&self, other: &Point) -> bool {
        self.lt(other) || self.eq(other)
    }
}
```

In the above example, the header of the `Ord` trait declaration contains a bound `: Eq`, which asserts that `Eq` is a supertrait of `Ord` (the trait header is equivalent to `trait Ord where Self: Eq`). The supertrait constraint acts as an obligation for impls of `Ord`: only types that already implement `Eq` can implement `Ord` (the obligation is enforced by the compiler). At the

same time, the constraint acts as a guarantee to users of `Ord` (generic functions constrained by `Ord`) that any type that implements `Ord` also implements `Eq`. The functionality provided by `Eq` then becomes available for values of any type implementing `Ord`. This allows us to write functions like the following:

```
fn eq_and_le<T>(first: &T, second: &T, third: &T) -> bool
    where T: Ord {
    first.eq(second) && first.le(third)
}
```

where we call `eq` (a method of the trait `Eq`) on `first`, even though the where-clause in the function signature does not explicitly contain the constraint `T: Eq`.

2.6 Multi-parameter traits & associated types

So far we have only seen single-parameter traits, parameterized by the implicit `Self` type variable. Traits, however, can have an arbitrary number of additional type parameters. We can, for example, define the trait `WeightedGraph` using multiple type parameters:

```
trait WeightedGraph<N, E> where E: HasWeight {
    fn edges(&self) -> &Vec<E>;
    ...
}
```

`WeightedGraph` is parameterized by `Self`, corresponding to the graph itself, and by type variables `N` and `E`, corresponding to the node and edge types respectively.

We can then use the trait in a generic function as in the example below:

```
fn graph_size<G, N, E>(graph: &G) -> usize
    where G: WeightedGraph<N, E> {
    graph.edges().len()
}
```

where we must also explicitly parameterize the function by the node and edge types.

However, if the concrete instantiations of `N` and `E` are always uniquely determined by the concrete type instantiation of `G`, we can declare the types of nodes and edges as *associated types* instead:

```
struct MyNode { ... }
struct MyWeightedEdge { ... }

trait WeightedGraph2 {
    type Node;
    type Edge: HasWeight;
    fn edges(&self) -> &Vec<Self::Edge>;
}

impl WeightedGraph2 for MyGraph {
    type Node = MyNode;
    type Edge = MyWeightedEdge;
    fn edges(&self) -> &Vec<MyWeightedEdge> {
        ...
    }
}

fn graph_size<G>(graph: &G) -> u32 where G: WeightedGraph2 {
    graph.edges().len()
}
```

An associated type acts as a named output parameter of a trait. Only `Self` and the other input parameters declared in the trait header are used to determine the specific impl used to resolve a call to an overloaded method or function. The concrete associated type instantiation is uniquely determined by those parameters (assuming no overlap in impl declarations). An associated type can thus be interpreted as a type-level function: its arguments are the name of the trait and its input parameters, and its return value is the associated type's instantiation provided in the impl.

We declare a new associated type in the body of a trait declaration using the `type` keyword. For example, in the code above, the trait `WeightedGraph2` declares two associated types: `Node` and `Edge`. An impl must also "implement" all associated types defined in its implemented trait by providing their instantiations. To use an associated type of a trait, we specify the name of the trait along with its type parameters, e.g. we use the syntax `<T as WeightedGraph2>::Node` to denote the associated type `Node` instantiated in the impl of `WeightedGraph2` for type `T`. In the case where there is no ambiguity as to which trait is being referenced, Rust lets us omit the trait name, e.g. `T::Node`.

Associated types defined in trait declarations can also be constrained. For example, the associated type `Edge` in `WeightedGraph2` is bounded by the `HasWeight` trait. Like supertrait bounds, associated type bounds serve as obligations for all impls of a trait to make sure that the types that instantiate the associated type satisfy the constraints. Those constraints are *propagated* similarly to supertrait constraints, meaning that a satisfied trait

constraint is sufficient to automatically infer that the constraints on that trait's associated types (and the trait's supertrait constraints) are satisfied as well. Therefore, the constraint `Edge: HasWeight` is available to users of `WeightedGraph2`:

```
trait HasWeight {
    fn weight(&self) -> u32;
}

fn weighted_graph_size<G>(graph: &G) -> u32 where G: WeightedGraph2 {
    graph.edges()
         .iter()
         .fold(0u32, |acc: u32, item: &G::Edge| acc + item.weight())
}
```

In the body of `weighted_graph_size` we use `fold`: a method of the `Iterator` trait. The method performs the reduce or combine operation on the elements of the iterator that is passed to it as the receiver. The arguments to `fold` are the initial value of the accumulator (`0u32` in this case) and a function that specifies the combine operation. In the above example, the combine operation is specified by the closure (an anonymous function) `|acc: u32, item: &G::Edge| acc + item.weight()`. The closure takes two arguments: an accumulator `acc` and an edge `item` (the type annotations in the example are not actually necessary as they can be inferred by the compiler), and returns the result of evaluating `acc + item.weight()`. We know that `item`'s type, `&<G as WeightedGraph2>::Edge`, is a reference to an associated type of the trait `WeightedGraph2`. Meanwhile, the method `weight()` belongs to the trait `HasWeight`, and so can only be called on a reference to a value whose type implements `HasWeight`. We can, however, call the method `weight()` on `item` without explicitly including the constraint `<G as WeightedGraph2>::Edge: HasWeight` in the where-clause of the function because the constraint `Edge: HasWeight` in the trait declaration of `WeightedGraph2` is propagated.

Finally, the programmer can write generic functions constrained by associated types. For example, the `Iterator` trait in the Rust standard library [2] has an associated type that corresponds to the type of the values of the iterator:

```
trait Iterator {
    type Item;
    ...
}
```

If we want to write a function that can be used with any iterator of 32-bit integers, we can simply specify the requirement in the where-clause by extending the trait constraint with an *associated type equality*:

[2] https://doc.rust-lang.org/std/iter/trait.Iterator.html

```
fn iterate<T>(iterator: &T) where T: Iterator<Item=i32> { ... }
```

2.7 Trait objects

Abstraction without overhead is an important design principle in Rust: abstractions in the language should not incur any performance penalties. To avoid generating run-time overhead from the use of generics and traits, the Rust compiler *monomorphizes* its generic functions and methods. The process consists of turning each generic function into multiple monomorphic functions: one for each instantiation of the function's type variables that occurs in the program. In addition to avoiding indirection and overhead caused by handling generics, the process allows the compiler to apply the function inlining optimization, which can make the resulting code more efficient. This way of translating generics can however lead to code bloat, which can result in large binaries. As an alternative to monomorphization, Rust provides the ability to dispatch trait methods dynamically through *trait objects*.

In Rust's type system, a trait declaration creates a related type of the same name. Such a type, which we call a *trait type*, describes values whose concrete types implement the trait (i.e., the concrete types are the `Self` parameters of the trait). The trait type is effectively an *existential type* [28]: the concrete type of its value is existentially quantified—hidden from the type system's field of view.

In Rust's memory model, where program values are allocated directly on the stack, the type of a value informs the compiler of the value's size and alignment requirements. However, in the case of a trait type, since the concrete type of the value is unknown, the size and alignment of the value are not known either, as a trait type may classify values of varying sizes. The trait type is thus an instance of a *dynamically sized type* (DST)—a type whose values do not have a statically known size. As such, DST values must appear behind some kind of pointer.[3]

A trait object is thus an object that contains a pointer to some value whose concrete type implements the trait, as well as a pointer to a *vtable* that contains pointers to the concrete trait methods that may be invoked on the object. The notation for the type of a trait object looks like a pointer to a trait type. For example, a trait object of the trait `HasWeight` can have type `&HasWeight` (in Rust documentation, the implementation of a trait object is referred to as a *fat pointer*).

To illustrate how trait objects work in Rust, consider the following example:

[3]Another instance of a DST is the type of an unknown size array of 32-bit integers `[i32]`. The type `&[i32]` denotes a *slice*, which is an object that contains a pointer to the array as well as metadata including the array's length.

```
struct Ball {
    radius: f64,
    weight: f64
}

struct Cube {
    width: f64,
    weight: f64
}

trait HasWeight {
    fn weight(&self) -> f64;
}

impl HasWeight for Ball {
    fn weight(&self) -> f64 { self.weight }
}
impl HasWeight for Cube {
    fn weight(&self) -> f64 { self.weight }
}

fn obj_weight(obj: &HasWeight) -> f64 { obj.weight() }
```

The function `obj_weight` does not have to be monomorphized at compile-time because it is already monomorphic. The concrete implementation of the `weight` method that is used in the body of `obj_weight` is the one referenced in the trait object `obj`'s vtable.

To create a trait object we can explicitly cast a reference to a value whose type implements the trait (as its `Self` type) to a trait object:

```
let ball = Ball { radius: 6.2, weight: 2.7 };
let ball_object = &ball as &HasWeight;
let ball_weight = obj_weight(ball_object);
```

We can also implicitly coerce a reference to an object by passing it to a piece of code that expects a trait object:

```
let ball = Ball { radius: 6.2, weight: 2.7 };
let ball_weight = obj_weight(&ball);
```

The 'existential type' nature of trait objects also allows us to have a heterogeneous view of values that have different types but share a common trait. This is particularly useful in creating collections of such values:

```
let ball = Ball { radius: 6.2, weight: 2.7 };
let cube = Cube { width: 2.5, weight: 9.9 };
let object_vector: Vec<&HasWeight> = vec!(&ball, &cube);
for item in object_vector {
    println!("I weigh {} kg.", item.weight());
}
```

Since all trait objects of `HasWeight` have the same type, we can create a vector of all objects of `HasWeight`, regardless of their concrete underlying types.

To conclude discussion of trait objects, we must note that not all traits can be used to create an object—only traits that are deemed *object-safe* can be used to do so. Roughly speaking, a trait is object-safe if all of its methods are object-safe. A method is object-safe if it is not generic and if in its type signature `Self` only appears behind a pointer as the receiver of the method (the restriction is necessary to prevent runtime type errors in operations on trait objects). However, the requirements for object safety get more complex in the presence of supertraits and associated types. We discuss those requirements in more detail when we introduce trait objects to MiniRust in chapter 5.

Chapter 3

NanoRust: a language with traits

In this chapter we begin the formal development of a subset of Rust with a focus on traits. Our goal is to have the formal model include some of the more advanced features of the trait system, namely associated types and trait objects. However, in an attempt to keep the presentation accessible to the reader, we defer including those features until later chapters. Therefore, in this chapter we introduce NanoRust: a simple language with traits but without associated types and trait objects, both of which will be introduced in subsequent chapters as part of MicroRust and MiniRust respectively.

3.1 Some syntactic conventions

We first set up some syntactic conventions, which we use throughout the remainder of this thesis.

We use the overline, e.g. $\overline{\tau}$, to denote sequences of zero or more elements. We use lowercase subscripts to distinguish between different sequences. For example, $\overline{\tau_i}$ and $\overline{\tau_j}$ represent different sequences. We reuse the same subscript to denote sequences of the same length: $\overline{\tau_i}$ has the same length as $\overline{T_i}$. If an overline is extended over both subscripted and non-subscripted items, the non-subscripted items remain constant. For example, $\overline{\Gamma \vdash_{\text{WF}} \tau_i}$ represents repetition of $\Gamma \vdash_{\text{WF}} \tau$ for each $\tau \in \overline{\tau_i}$. In the case where we want to specify that a sequence forms an unordered set, we use curly braces: $\{\overline{\tau_i}\}$.

In type schemes of the form $\forall \overline{T}.\overline{\pi} \Rightarrow \tau$, we can omit the for-all quantifier $\forall$ if the sequence of type variables $\overline{T}$ is empty. Similarly, we can remove the qualification symbol $\Rightarrow$ if the sequence of constraints $\overline{\pi}$ is empty as well. Therefore, assuming that () denotes an empty sequence, we consider $\forall().() \Rightarrow \tau$ equivalent to τ. The same convention applies to constraint schemes $\forall \overline{T}.\overline{\pi} \Rightarrow \pi$.

We use $[\tau/T]$ to denote a substitution. For instance, $[\tau/T]\pi$ represents the result of substituting all free occurrences of the type variable T in π with the type τ.

$$
\begin{array}{rcll}
 & & e \in \textsc{Term},\quad T, \mathsf{Self} \in \textsc{TVar},\quad S \in \textsc{SCons},\quad \tau \in \textsc{Type} & \\
 & & x, f \in \textsc{Var},\quad D \in \textsc{Trait},\quad \sigma \in \textsc{TScheme},\quad \theta \in \textsc{CScheme} & \\
\mathit{pgm} & ::= & \overline{\mathit{item}} & \text{(programs)} \\
\mathit{item} & ::= & \mathsf{struct}\ S\,\langle\overline{T}\rangle\,\{\overline{x:\tau}\} & \text{(items)} \\
 & \mid & \mathit{fun} & \\
 & \mid & \mathsf{trait}\ D\,\langle\overline{T}\rangle\ \mathsf{for}\ \mathsf{Self}\ \mathsf{where}\ \overline{\pi}\,\{\,f:\sigma\,\} & \\
 & \mid & \mathsf{impl}\,\langle\overline{T}\rangle\,D\,\langle\overline{\tau}\rangle\ \mathsf{for}\ \tau\ \mathsf{where}\ \overline{\pi}\,\{\mathit{fun}\,\} & \\
\mathit{fun} & ::= & \mathsf{fn}\ f\,\langle\overline{T}\rangle\,(\overline{x:\tau}) \to \tau\ \mathsf{where}\ \overline{\pi}\,\{\,e\,\} & \text{(functions)} \\
e & ::= & \mathsf{let}\ x = e\ \mathsf{in}\ e \mid lv := e \mid e;e \mid \&lv \mid *e & \text{(terms)} \\
 & \mid & e(\overline{e}) \mid x \mid () \mid e\ \mathsf{as}\ \tau \mid S\{\overline{x:e}\} \mid e.x & \\
lv & ::= & x \mid *e \mid lv.x & \text{(lvalues)} \\
\tau & ::= & () \mid T \mid \mathsf{fn}(\overline{\tau}) \to \tau \mid \&\tau \mid S\,\langle\overline{\tau}\rangle & \text{(types)} \\
\sigma & ::= & \forall\overline{T}.\overline{\pi} \Rightarrow \tau & \text{(type schemes)} \\
\theta & ::= & \forall\overline{T}.\overline{\pi} \Rightarrow \pi & \text{(constraint schemes)} \\
\pi & ::= & \tau : \beta & \text{(trait constraints)} \\
\beta & ::= & D\,\langle\overline{\tau}\rangle & \text{(trait bounds)} \\
\Gamma & ::= & \emptyset \mid \Gamma, x:\sigma \mid \Gamma, T \mid \Gamma, S\,\langle\overline{T}\rangle\,\{\overline{x:\tau}\} & \text{(typing environments)} \\
 & \mid & \Gamma, (D\,\langle\overline{T}\rangle\ \mathsf{where}\ \overline{\pi}, \mathsf{Self}:\overline{\beta}, f) & \\
\Theta & ::= & \emptyset \mid \Theta, \theta & \text{(constraint environments)}
\end{array}
$$

Figure 3.1: NanoRust: syntax

3.2 Syntax

Figure 3.1 presents the syntax of NanoRust. The syntax gives us a glimpse of the features of Rust modeled in this thesis. Note that, to keep our formalization relatively lightweight, we only model a small subset of the language. As a general rule, we include features that are relevant to traits and some features that—we believe—capture the essence of Rust. For instance, imperative features, such as references and assignment expressions, demonstrate the systems language nature of Rust. References also form an integral part of our formalization of trait objects in chapter 5. We also include structs, which let us create new types. We use structs in the translation of MiniRust to its internal language in chapter 7. We notably omit the features of Rust's type system that focus on memory safety. For instance, lifetimes play a significant role in Rust's trait system, however they are omitted from the model due to the complexity that they bring to the language. We do believe, however, that they would be an interesting addition to the formalization, worth exploring in future work.

3.2.1 Items

Programs in NanoRust consist of a collection of items. The items we model in NanoRust include struct declarations, functions, traits, and impls.

A *struct declaration* defines a new composite data type along with its structure: the names of its fields and their respective types. Struct declarations may also be polymorphic. A declaration struct $S\,\langle\overline{T_j}\rangle\,\{\overline{x_i : \tau_i}\}$ defines the polymorphic type $\forall\overline{T_j}.S\,\langle\overline{T_j}\rangle$. We can interpret the polymorphic type as a family of types $S\,\langle\overline{\tau_j}\rangle$. The types $\overline{\tau_i}$ of the struct fields $\overline{x_i}$ are then parametric over the instantiations $\overline{\tau_j}$ of $\overline{T_j}$.

Every *function declaration* is explicitly typed and, like structs, can be polymorphic. In the *function signature* $\mathsf{fn}\ f\,\langle\overline{T_j}\rangle\,(\overline{x_i : \tau_i}) \to \tau\ \mathsf{where}\ \overline{\pi_k}$, f is the name of the function, $\overline{T_j}$ are the type variables over which the function is parameterized, $\overline{x_i : \tau_i}$ are the names and types of the function arguments, and τ is the return type. The where-clause $\mathsf{where}\ \overline{\pi_k}$ contains constraints on the function's type parameters, which must be satisfied by any instantiation of the function. The return value of the function is the result of evaluating its body e, which is a single expression. We require programs to have a function f_{MAIN}, corresponding to Rust's `main` function, which serves as the starting point of a NanoRust program.

A *trait declaration* consists of two parts: the *header* and the *body*. The header $\mathsf{trait}\ D\,\langle\overline{T_i}\rangle\,\mathsf{for\ Self\ where}\ \overline{\pi_j}$ declares the name of the trait D, the name of its Self parameter Self and non-Self parameters $\overline{T_i}$ (note that in contrast to Rust, trait headers in NanoRust explicitly declare the Self type variable in anticipation of kinds in chapter 5). The where-clause $\mathsf{where}\ \overline{\pi_j}$ contains supertrait constraints $\mathsf{Self} : \beta$ and any other constraints $\tau : \beta$ that must be satisfied by every impl that implements the trait.

In Rust, the body of a trait may include both *trait methods*, where `self` (or a reference to it) appears as the first argument in the method signature and acts as a receiver, and *associated functions*, which do not have the restriction on `self` but in consequence can only be called using UFCS, described in the previous chapter. To keep the syntax of terms in NanoRust as concise as possible, we do not support method call syntax. This lets us unify trait methods and associated functions in the form of *trait functions*. To call a trait function, we use standard function syntax (without a trait name prefix), following the lead of class methods in Haskell [40]. To avoid ambiguities related to multiple traits declaring functions of the same name, we simply require that all functions in a program, including trait functions, have unique names.

The body of a trait declaration in NanoRust consists of a single trait function. Specifically, it contains the function's name and its type signature in the form of a type scheme. The restriction on the number of functions in a given trait is purely motivated by the clarity of presentation of the model. It should be fairly straightforward to add support for an arbitrary number of trait functions; we believe that doing so, however, would increase the complexity of the formal model without shedding new conceptual light on the language.

An *impl declaration* provides an implementation of a trait. The impl *header* $\mathsf{impl}\,\langle\overline{T_j}\rangle\,D\,\langle\overline{\tau_i}\rangle\,\mathsf{for}\ \tau\ \mathsf{where}\ \overline{\pi_k}$ provides an instantiation of the trait parameters for which the

implementation is written: τ is an instantiation of `Self` and $\overline{\tau_i}$ are instantiations of the non-Self trait parameters. The impl, along with its type instantiations, is parameterized by type variables $\overline{T_j}$, which are constrained by constraints $\overline{\pi_k}$ in the where-clause. The *body* of the impl provides a concrete implementation of the trait function.

Rust also allows declaring nested items, which effectively restricts where such items are visible. However, semantics of nested items are the same as the semantics of top-level items. In particular, an inner function nested inside an outer function does not capture the local variables from the outer function in its scope (Rust does provide however the ability to write closures—anonymous functions that enclose their surrounding environment, which are not part of this formalization). For simplicity, in NanoRust we only permit items declared at the top level of the program.

3.2.2 Terms

e ranges over terms (or expressions) in NanoRust. Terms include:

- let $x = e_1$ in e_2: a local variable declaration equivalent to `let x = e; e` in Rust,
- $lv := e$: an assignment expression (equivalent to `lv = e` in Rust), which assigns the result of evaluating the expression e to the location represented by the lvalue lv [1],
- $e;e$: which permits chaining expressions and can be used to emulate statements, since the result of evaluating the first term is discarded,
- $\&lv$: the reference operation, which returns a reference to the value stored at the memory location represented by the lvalue lv (all references in NanoRust are mutable),
- $*e$: the dereference operation, which returns the value stored at the memory location represented by e,
- $e(\overline{e})$: a function call,
- x: an identifier ranging over term variable names (local variables, functions, etc.),
- $()$: the unit expression associated with no computation,
- $S\{\overline{x : e}\}$: a struct instance,
- $e.x$: a struct field access expression and

[1] Local variables are *immutable* by default in Rust. However, since the NanoRust type system does not keep track of mutability of individual variables and we want to have a notion of mutable state in the model, local variables in NanoRust are *always mutable*.

- e as τ: a type ascription. Type ascriptions in Rust are used to cast the type of a term to another type. In NanoRust, they are used to simulate programmer-supplied type annotations since the syntax does not include explicitly annotated let-expressions (e.g. `let x: i32 = 4;` can be written as let $x = 4$ as `i32` in ...), or explicitly parameterized struct instantiations and function calls (e.g. `<i32>::print(4)` can be written as (`print` as fn (`i32`) → ())(4)).

The metavariable lv ranges over *lvalues*: terms that represent memory locations. As such, they can be assigned to and they can be referenced. They include identifiers x, dereference operations $*x$ and field accesses $lv.x$.

3.2.3 Types

Every term that occurs in a NanoRust program has a type represented by the metavariable τ. Types include:

- unit type (): the type of the unit expression (),
- type variable T, which we can interpret as a type placeholder, used to depict generics,
- function type $\mathsf{fn}(\overline{\tau_i}) \to \tau$, which ranges over functions that take arguments of types $\overline{\tau_i}$ and return a value of type τ,
- reference type $\&\tau$, and
- struct type $S\,\langle\overline{\tau_i}\rangle$: type of instances of struct S whose type parameters are instantiated to types $\overline{\tau_i}$.

We omit primitive machine types such as integers and booleans. We believe that their inclusion would not bring any useful insight into the semantics of the NanoRust. Adding primitive types with their corresponding terms and operations should however be fairly straightforward.

3.2.4 Polymorphic types

The types we have seen so far are *monomorphic*: they correspond to a single 'concrete' type representation. In order to reason about generic language constructs, we also include the notion of a polymorphic type (a type that may take on multiple concrete representations) through *type schemes*. In a system without traits (or similar constructs) and constraints, a type scheme σ is generally of the form $\forall\overline{T_i}.\tau$. We can understand it to mean that a value that is assigned a type scheme σ can have type $[\overline{\tau_i/T_i}]\tau$ for any choice of types $\overline{\tau_i}$. This form of type scheme is sufficient to support parametric polymorphism, which is expressed in Rust through generics.

To enable ad-hoc polymorphism through traits, we need to be able to constrain the type variables occurring in a type scheme. Thus, type schemes in NanoRust are *qualified*: they include *predicates* (which we also call *constraints*) that must be satisfied before the type scheme's type variables can be instantiated. A qualified type scheme [20] has form $\forall\overline{T_i}.\overline{\pi_j} \Rightarrow \tau$. The type scheme is qualified by constraints (or predicates) $\overline{\pi_j}$. As such, a value with a type scheme $\forall\overline{T_i}.\overline{\pi_j} \Rightarrow \tau$ can have type $[\overline{\tau_i/T_i}]\tau$ for any choice of types $\overline{\tau_i}$ such that the predicates $[\overline{\tau_i/T_i}]\overline{\pi_j}$ are satisfied. In NanoRust, those predicates are trait constraints of the form $\tau : D\,\langle\overline{\tau_k}\rangle$, which we read as "type τ implements trait D with parameters $\overline{\tau_k}$". From this point on, we simply refer to qualified type schemes as type schemes.

To express generic constraints, we also include *constraint schemes*, which also can be qualified. Similar to type schemes, they have form form $\forall\overline{T_i}.\overline{\pi_j} \Rightarrow \pi$ which we can read as: "for any type instantiation $\overline{\tau_i}$ of types $\overline{T_i}$ such that the predicates $[\overline{\tau_i/T_i}]\overline{\pi_j}$ are all satisfied, $[\overline{\tau_i/T_i}]\pi$ is satisfied as well". In a constraint scheme $\forall\overline{T_i}.\overline{\pi_j} \Rightarrow \pi$, we refer to $\overline{\pi_j}$ as the *qualifying constraints*, and we refer to π as the *principal constraint* of the constraint scheme.

3.2.5 Environments

In order to type expressions, we carry two environments containing our current assumptions about what is in scope. The typing environment Γ is a set that contains variable typings $x : \sigma$, type variables that are in scope, available struct definitions, and tuples that contain information about traits that have been declared. A trait information tuple $(D\,\langle\overline{T}\rangle\,\mathsf{where}\,\overline{\pi}, \mathsf{Self} : \overline{\beta}, f)$ is implicitly quantified by the type variables Self and $\overline{T}$. It consists of the trait header $D\,\langle\overline{T}\rangle\,\mathsf{where}\,\overline{\pi}$, supertrait bounds $\mathsf{Self} : \overline{\beta}$, and an identifier f corresponding to the name of the trait function. We also have a constraint environment Θ that is a set of constraint schemes corresponding to impls that we can assume to be in scope.

3.2.6 Example

To show how a Rust program may be represented in NanoRust, consider the following example from chapter 2:

```
struct Point {
    x: i32,
    y: i32
}

trait Eq {
    fn eq(&self, &Self) -> bool;
}

impl Eq for Point {
    fn eq(&self, other: &Point) -> bool {
        self.x == other.x && self.y == other.y
    }
}

fn main() {
    let (p1, p2) = (Point {x: 3, y: 4}, Point {x: 4, y: 5});
    let are_equal = p1.eq(&p2)
}
```

An equivalent representation of the above program in NanoRust would look as follows:

```
struct Point{ x : i32, y : i32 }
trait Eq for Self { eq : fn(&Self, &Self) → bool }
impl Eq for Point{
  fn eq(self : &Point, other : &Point) → bool {
    &&(==((∗self).x, (∗other).x), ==((∗self).y, (∗other).y))
  }
}
fn main() → () {
  let p1 = Point{x : 3, y : 4} in
  let p2 = Point{x : 4, y : 5} in
  let are_equal = eq(&p1, &p2) in ()
}
```

where && and == are some two-argument functions in scope. Note that NanoRust does not include automatic referencing and dereferencing—such operations must be made explicit.

3.3 Well-formedness and constraint entailment

Figure 3.2 contains the well-formedness and constraint entailment judgments.

At the top of the figure we have an auxiliary relation $\Gamma \vdash \tau : D\,\langle\overline{\tau_i}\rangle$, which ensures that the trait D is in scope and is provided the right number of parameters. The single rule of the

$$\frac{(D\,\langle\overline{T_i}\rangle \text{ where } \_,\_,\_) \in \Gamma}{\Gamma \vdash \tau : D\,\langle\overline{\tau_i}\rangle}$$

$\boxed{\Gamma \vdash_{\text{WF}} \sigma}$ (Well-formed type schemes)

$$\text{(wf-var)}\frac{T \in \Gamma}{\Gamma \vdash_{\text{WF}} T} \qquad \text{(wf-unit)}\frac{}{\Gamma \vdash_{\text{WF}} ()} \qquad \text{(wf-fun)}\frac{\overline{\Gamma \vdash_{\text{WF}} \tau_i} \quad \Gamma \vdash_{\text{WF}} \tau}{\Gamma \vdash_{\text{WF}} \mathsf{fn}(\overline{\tau_i}) \to \tau}$$

$$\text{(wf-ref)}\frac{\Gamma \vdash_{\text{WF}} \tau}{\Gamma \vdash_{\text{WF}} \&\tau} \qquad \text{(wf-struct)}\frac{(S\,\langle\overline{T_i}\rangle\,\{\overline{x_j : \tau_j}\}) \in \Gamma \quad \overline{\Gamma \vdash_{\text{WF}} \tau_i}}{\Gamma \vdash_{\text{WF}} S\,\langle\overline{\tau_i}\rangle}$$

$$\text{(wf-tscheme)}\frac{\overline{\Gamma, \overline{T_i} \vdash_{\text{WF}} \pi_j}^{\,j} \quad \Gamma, \overline{T_i} \vdash_{\text{WF}} \tau}{\Gamma \vdash_{\text{WF}} \forall \overline{T_i}.\overline{\pi_j} \Rightarrow \tau}$$

$\boxed{\Gamma \vdash_{\text{WF}} \theta}$ (Well-formed constraint schemes)

$$\text{(wf-cscheme)}\frac{\overline{\Gamma, \overline{T_i} \vdash_{\text{WF}} \pi_j} \quad \Gamma, \overline{T_i} \vdash_{\text{WF}} \pi}{\Gamma \vdash_{\text{WF}} \forall \overline{T_i}.\overline{\pi_j} \Rightarrow \pi} \qquad \text{(wf-trcons)}\frac{\Gamma \vdash \tau : D\,\langle\overline{\tau_i}\rangle \quad \Gamma \vdash_{\text{WF}} \tau \quad \overline{\Gamma \vdash_{\text{WF}} \tau_i}}{\Gamma \vdash_{\text{WF}} \tau : D\,\langle\overline{\tau_i}\rangle}$$

$\boxed{\Gamma \mid \Theta \Vdash \pi}$ (Constraint entailment)

$$\text{(c-entail)}\frac{(\forall \overline{T_i}.\overline{\pi_j} \Rightarrow \pi) \in \Theta \quad \overline{\Gamma \vdash_{\text{WF}} \tau_i} \quad \overline{\Gamma \mid \Theta \Vdash [\overline{\tau_i/T_i}]\pi_j}}{\Gamma \mid \Theta \Vdash [\overline{\tau_i/T_i}]\pi}$$

Figure 3.2: NanoRust: well-formedness and constraint entailment

relation uses the trait tuple in the typing environment Γ to obtain the relevant information. Note that the tuple is implicitly quantified by the type variables $\overline{T_i}$. Throughout this thesis we maintain a convention that bound variables can be renamed, so the choice of names of the variables $\overline{T_i}$ does not matter (e.g. $(D\,\langle T_1\rangle, \mathsf{Self}, f)$ is equivalent to $(D\,\langle T_2\rangle, \mathsf{Self}, f)$).

The well-formed type schemes judgment $\Gamma \vdash_{\text{WF}} \sigma$ can be interpreted as: "type scheme σ is well-formed with respect to Γ". Intuitively, a type or type scheme is well-formed if it can plausibly be assigned to some expression or item under the current set of assumptions. Specifically, the well-formedness judgment ensures that the type variables (in rule (wf-tvar)) and structs (in rule (wf-struct)) used in the type are in the current set of assumptions in Γ. In rule (wf-tscheme), the type variables over which the type scheme is universally quantified are added into the set of assumptions in the premises, which verify well-formedness of the type scheme's components. This enables the body of the type scheme to mention those type

variables.

The judgment is principally used to verify that type annotations provided by the programmer "make sense". Strictly speaking, the well-formedness checks are not necessary to ensure type safety in NanoRust. However, a term with an ill-formed type will not be very useful as the typing rules prevent it from being used in any practical manner. Checking for well-formedness can however help weed out ill-formed type annotations in item declarations and thus reduce the amount of dead code in well-typed programs. It also allows us to reason about well-formed environments, which will be important when we discuss the dynamic semantics of our language and trait coherence in chapters 7 and 8.

The well-formed constraint schemes judgment $\Gamma \vdash_{\text{WF}} \theta$ verifies well formedness of constraints and constraint schemes. In particular, the rule (wf-trcons) verifies that a trait constraint refers to a trait that has actually been declared in the program and that the number of type parameters in the constraint matches the trait declaration.

The constraint entailment judgment $\Gamma \mid \Theta \Vdash \pi$ verifies that the constraint π is satisfied under the set of constraint assumptions Θ and typing environment Γ. The judgment consists of a single rule (c-entail), which looks up the relevant constraint scheme in the constraint environment Θ and verifies that the qualifying predicates $\overline{\pi_j}$ are satisfied, under the appropriate instantiation of the type variables $\overline{T_i}$, over which the constraint scheme is quantified. Note that due to our syntactic convention that lets us remove unneeded quantifiers $\forall$ and qualifiers $\Rightarrow$, the rule

$$\frac{\pi \in \Theta}{\Gamma \mid \Theta \vdash \pi}$$

is just a special case of (c-entail).

3.4 Well-typed terms

Figure 3.3 presents the typing rules for terms in NanoRust. The typing judgment $\Gamma \mid \Theta \vdash e : \tau$ can be read as: "term e has type τ under sets of assumptions Γ and Θ". One thing to note about this judgment is that, while the variable bindings in Γ may be assigned polymorphic types (or type schemes), the typing judgment assigns a monomorphic type τ to each term. This is due to the fact that the use of generics in Rust is limited: only top-level items may declare new polymorphic constructs and all type parameters of a generic item have to be instantiated with some monomorphic types before the item can be used.

The rule (var), used for typing term variables, reflects this principle. If the variable is bound to a type scheme in Γ, then the type variables $\overline{T_i}$ in the type scheme must be instantiated to some types $\overline{\tau_i}$. If the type scheme is also qualified, we must also check that the constraints $[\overline{\tau_i/T_i}]\overline{\pi_j}$ are satisfied.

Note that the rule does not tell us which specific types $\overline{\tau_i}$ to use for the type variable

$\boxed{\Gamma \mid \Theta \vdash e : \tau}$ (Well-typed terms)

$$\text{(as)}\ \frac{\Gamma \mid \Theta \vdash e : \tau}{\Gamma \mid \Theta \vdash e \text{ as } \tau : \tau} \qquad \text{(unit)}\ \frac{}{\Gamma \mid \Theta \vdash () : ()} \qquad \text{(let-un)}\ \frac{\Gamma \mid \Theta \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \mid \Theta \vdash e_2 : \tau_2}{\Gamma \mid \Theta \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2}$$

$$\text{(seq)}\ \frac{\Gamma \mid \Theta \vdash e_1 : \tau_1 \quad \Gamma \mid \Theta \vdash e_2 : \tau_2}{\Gamma \mid \Theta \vdash e_1; e_2 : \tau_2} \qquad \text{(var)}\ \frac{(x : \forall \overline{T_i}.\overline{\pi_j} \Rightarrow \tau) \in \Gamma \quad \overline{\Gamma \vdash_{\text{WF}} \tau_i} \quad \overline{\Gamma \mid \Theta \Vdash [\overline{\tau_i/T_i}]\pi_j}^{\,j}}{\Gamma \mid \Theta \vdash x : [\overline{\tau_i/T_i}]\tau}$$

$$\text{(app)}\ \frac{\Gamma \mid \Theta \vdash e : \mathsf{fn}(\overline{\tau_i}) \to \tau \quad \overline{\Gamma \mid \Theta \vdash e_i : \tau_i}}{\Gamma \mid \Theta \vdash e(\overline{e_i}) : \tau} \qquad \text{(ref)}\ \frac{\Gamma \mid \Theta \vdash lv : \tau}{\Gamma \mid \Theta \vdash \&lv : \&\tau} \qquad \text{(deref)}\ \frac{\Gamma \mid \Theta \vdash e : \&\tau}{\Gamma \mid \Theta \vdash *e : \tau}$$

$$\text{(asgn)}\ \frac{\Gamma \mid \Theta \vdash lv : \tau \quad \Gamma \mid \Theta \vdash e : \tau}{\Gamma \mid \Theta \vdash lv := e : ()} \qquad \text{(new-struct)}\ \frac{S\langle \overline{T_k} \rangle \{\overline{x_i : \tau_i}\} \in \Gamma \quad \overline{\Gamma \mid \Theta \vdash e_i : [\overline{\tau_k/T_k}]\tau_i} \quad \overline{\Gamma \vdash_{\text{WF}} \tau_k}}{\Gamma \mid \Theta \vdash S\{\overline{x_i : e_i}\} : S\langle \overline{\tau_k} \rangle}$$

$$\text{(proj)}\ \frac{\Gamma \mid \Theta \vdash e : S\langle \overline{\tau_j} \rangle \qquad S\langle \overline{T_j} \rangle \{\overline{x_m : \tau_m}, x : \tau, \overline{x_n : \tau_n}\} \in \Gamma}{\Gamma \mid \Theta \vdash e.x : [\overline{\tau_j/T_j}]\tau}$$

Figure 3.3: NanoRust: well-typed terms

instantiations. NanoRust's type system can thus allow multiple typings of the same program. An implementation of the type system would have to perform some type inference in order to choose type instantiations appropriate in the given context.

Finally, note that our syntactic conventions make the rule

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau}$$

a special case of an application of (var).

3.5 Well-typed items & programs

Figure 3.4 presents the typing rules for items and programs.

The purpose of the well-typed items judgment is to populate the top-level typing and constraint environments with relevant assumptions obtained from program items. The judgment $\Gamma_I \mid \Theta_I \vdash \mathit{item} : \Gamma_O \mid \Theta_O$ takes as input environments Γ_I, Θ_I and it produces output environments Γ_O, Θ_O. In the well-typed programs judgment $\vdash_P \mathit{pgm} : \Gamma \mid \Theta$ we see that the output environments of each item are in fact subsets of the input environments. As such, the input

$\boxed{\Gamma \mid \Theta \vdash \mathit{item} : \Gamma \mid \Theta}$ (Well-typed items)

$$\text{(struct)}\ \frac{\overline{\Gamma, \overline{T_j} \vdash_{\text{WF}} \tau_i}^{\,i}}{\Gamma \mid \Theta \vdash \mathsf{struct}\ S\ \langle \overline{T_j} \rangle\ \{\overline{x_i : \tau_i}\} : [S\ \langle \overline{T_j} \rangle\ \{\overline{x_i : \tau_i}\}] \mid \emptyset}$$

$$\text{(fun)}\ \frac{\sigma = \forall \overline{T_k}.\overline{\pi_j} \Rightarrow \mathsf{fn}(\overline{\tau_i}) \to \tau \qquad \Gamma \vdash_{\text{WF}} \sigma \qquad \Gamma, \overline{T_k}, \overline{x_i : \tau_i} \mid \Theta, \overline{\pi_j} \vdash e : \tau}{\Gamma \mid \Theta \vdash \mathsf{fn}\ f\ \langle \overline{T_k} \rangle\ (\overline{x_i : \tau_i}) \to \tau\ \mathsf{where}\ \overline{\pi_j}\ \{\, e \,\} : [f : \sigma] \mid \emptyset}$$

$$\text{(trait)}\ \frac{\begin{array}{c}\{\overline{\pi_j}\} \equiv \{\overline{\mathsf{Self} : \beta_s}, \overline{\pi_h}\} \qquad \overline{T_p} = \mathsf{Self}, \overline{T_i} \qquad \overline{\Gamma \vdash_{\text{WF}} \forall \overline{T_p}.(\mathsf{Self} : D\ \langle \overline{T_i} \rangle) \Rightarrow \pi_j}^{\,j} \\ \sigma = \forall \overline{T_k}.\overline{\pi_l} \Rightarrow \mathsf{fn}(\overline{\tau_m}) \to \tau_R \qquad \sigma' = \forall \overline{T_p}.(\mathsf{Self} : D\ \langle \overline{T_i} \rangle) \Rightarrow \sigma \qquad \Gamma \vdash_{\text{WF}} \sigma'\end{array}}{\Gamma \mid \Theta \vdash \begin{array}{l}\mathsf{trait}\ D\ \langle \overline{T_i} \rangle\ \mathsf{for}\ \mathsf{Self} \\ \mathsf{where}\ \overline{\pi_j}\ \{\, f : \sigma \,\}\end{array} : \begin{array}{l}[(D\ \langle \overline{T_i} \rangle\ \mathsf{where}\ \overline{\pi_h}, \mathsf{Self} : \overline{\beta_s}, f), (f : \sigma')] \mid \\ [\overline{\forall \overline{T_p}.(\mathsf{Self} : D\ \langle \overline{T_i} \rangle) \Rightarrow \mathsf{Self} : \beta_s}^{\,s}\ ]\end{array}}$$

$$\text{(impl)}\ \frac{\begin{array}{c}(D\ \langle \overline{T_i} \rangle\ \mathsf{where}\ \overline{\pi_h}, \mathsf{Self} : \overline{\beta_s}, f) \in \Gamma \qquad \overline{\tau_p} = \tau, \overline{\tau_i} \qquad \overline{T_p} = \mathsf{Self}, \overline{T_i} \qquad \overline{T_k \in FV(\tau_p)} \\ \Gamma \mid \Theta \vdash_{\text{WF}} \forall \overline{T_k}.\overline{\pi_j} \Rightarrow \tau : D\ \langle \overline{\tau_i} \rangle \qquad \Theta^* = \Theta \setminus \{\overline{\forall \overline{T_p}.(\mathsf{Self} : D\ \langle \overline{T_i} \rangle) \Rightarrow \mathsf{Self} : \beta_s}^{\,s}\} \\ \overline{\Gamma, \overline{T_k} \mid \Theta^*, \overline{\pi_j} \Vdash [\overline{\tau_p / T_p}](\mathsf{Self} : \beta_s)}^{\,s} \qquad \overline{\Gamma, \overline{T_k} \mid \Theta^*, \overline{\pi_j} \Vdash [\overline{\tau_p / T_p}]\pi_h}^{\,h} \\ (f : \forall \overline{T_p}.(\mathsf{Self} : D\ \langle \overline{T_i} \rangle) \Rightarrow \sigma) \in \Gamma \qquad \Gamma, \overline{T_k} \mid \Theta^*, \overline{\pi_j} \vdash \mathit{fun} : [f : \sigma'] \mid \emptyset \qquad \sigma' = [\overline{\tau_p / T_p}]\sigma\end{array}}{\Gamma \mid \Theta \vdash \begin{array}{l}\mathsf{impl}\ \langle \overline{T_k} \rangle\ D\ \langle \overline{\tau_i} \rangle\ \mathsf{for}\ \tau\ \mathsf{where}\ \overline{\pi_j}\ \{ \\ \quad \mathit{fun}\ \}\end{array} : \emptyset \mid [\forall \overline{T_k}.\overline{\pi_j} \Rightarrow \tau : D\ \langle \overline{\tau_i} \rangle]}$$

$\boxed{\vdash_P \mathit{pgm} : \Gamma \mid \Theta}$ (Well-typed programs)

$$\text{(pgm)}\ \frac{\overline{\Gamma \mid \Theta \vdash \mathit{item}_i : \Gamma_i \mid \Theta_i} \qquad \Gamma = \overline{\Gamma_i} \qquad \Theta = \overline{\Theta_i}}{\vdash_P \overline{\mathit{item}_i} : \Gamma \mid \Theta}$$

Figure 3.4: NanoRust: well-typed items and programs

environments to program items are unions of the output environments of each item (we use the notation $\overline{\Gamma_i}$ as syntactic sugar for $\bigcup\{\overline{\Gamma_i}\}$). This allows us to declare mutually dependent items, such as mutually recursive functions.

The typing rule for structs is straightforward: it checks that the types of the struct fields are well-formed and it adds the struct definition to the typing environment. The typing rule for functions, (fun), generates a type scheme σ from the function signature and it uses the well-typed terms judgment to ensure that the body of the function is well-typed while extending the sets of assumptions with the type parameters of the function, the function arguments, and the constraints $\overline{\pi_j}$ from the function's where-clause.

3.5.1 Well-typed traits

The typing rule for traits, (trait), produces three results:

- a tuple that represents the obligations that impls of the trait must fulfill,
- a binding of the trait function name to its type signature, and
- propagated constraint schemes corresponding to the trait's supertraits, which imply that, given a satisfied trait constraint, its corresponding supertrait constraints are also satisfied.

Supertrait constraints are part of the trait declaration's where-clause. As of version 1.3 of the compiler, Rust restricts supertrait constraints to be of the form $\mathsf{Self} : \beta$. In consequence, any additional constraints in the where-clause (in the form of bounds on a non-Self type) are not propagated: the user of the trait is not able to assume that non-supertrait constraints from the where-clause are satisfied. For example, given the following traits:

```
struct Wrapper<T> { ... }

trait Copy {
    fn copy(self) -> Self;
}

trait Wrappable where Wrapper<Self>: Copy {
    fn wrap(self) -> Wrapper<Self>;
}
```

the following function would not type-check:

```
fn wrap_and_copy<T>(arg: T) -> Wrapper<T> where T: Wrappable {
    arg.wrap().copy()
}
```

For `wrap_and_copy` to be well-typed, the constraint `Wrapper<T>: Copy` has to be explicitly included in its where-clause. The restriction seems to be a side effect of the history of development of traits and where-clauses in Rust as supertrait constraints (in the form of bounds on `Self`) predate where-clauses.

In NanoRust, we inherit the same restriction on supertraits and we do not propagate non-supertrait constraints. In the first side condition of the rule, $\{\overline{\pi_j}\} \equiv \{\overline{\mathsf{Self} : \beta_s}, \overline{\pi_h}\}$, we separate supertrait constraints $\overline{\mathsf{Self} : \beta_s}$ from the other constraints. We then verify that the constraints in the where-clause are well-formed.

To verify well-formedness of the programmer-provided trait function type signature σ, we first build a generalized type scheme σ'. To do so, we quantify σ by the trait parameters and we qualify the resulting type scheme with the trait constraint. We then verify

that σ' is well-formed. Note that the notation $\forall\overline{T_p}.(\mathsf{Self} : D\,\langle\overline{T_i}\rangle) \Rightarrow \sigma$ is syntactic sugar for $\forall\overline{T_p}, \overline{T_k}.(\mathsf{Self} : D\,\langle\overline{T_i}\rangle, \overline{\pi_l}) \Rightarrow \tau$ where $\sigma = \forall\overline{T_k}.\overline{\pi_l} \Rightarrow \tau$.

3.5.2 Well-typed impls

The rule (impl) ensures that the impl declaration fulfills the obligations set out by the trait that it is implementing and it adds the appropriate constraint scheme to the set of assumed constraints. The impl itself can be universally quantified by type variables $\overline{T_k}$ and constrained by predicates $\overline{\pi_j}$ in the impl's where-clause.

In the side condition $\overline{T_k \in FV(\overline{\tau_p})}$, we use a function FV that returns the set of free type variables in a given type or sequence of types. The side condition ensures that the type variables, over which the impl is quantified are all part of the impl's trait parameter instantiations.

When verifying entailment of constraints in the implemented trait's where-clause and when typing the trait function implementation, we use a modified constraint environment Θ^*, which omits the propagated supertrait constraint schemes of the implemented trait from Θ. Doing so prevents trivial entailment of the supertrait constraints: since the output constraint scheme of the impl is also part of the input constraint environment Θ, we could use it in conjunction with the supertrait constraint scheme from the trait typing rule to conclude that the supertrait constraint is satisfied. For example, consider the following program:

```
trait Bar { ... }
trait Foo: Bar { ... }
impl Foo for i32 { ... }
```

The top-level constraint environment Θ generated for the program would be
$\{\forall\mathsf{Self}.\mathsf{Self} : \mathtt{Foo} \Rightarrow \mathsf{Self} : \mathtt{Bar},\quad \mathtt{i32} : \mathtt{Foo}\}$.
We would then be able to obtain a derivation of $\Gamma \mid \Theta \Vdash \mathtt{i32} : \mathtt{Bar}$ and falsely assume that the supertrait constraint is satisfied.

Chapter 4

MicroRust: NanoRust with associated types

In this chapter we extend NanoRust with associated types and equality constraints, creating a new language, which we call MicroRust. Our formal treatment of associated types and type equality constraints presented in this chapter is largely inspired by work on Haskell associated type synonyms [12].

4.1 Syntax

Figure 4.1 contains the syntax of MicroRust. The highlighted parts represent new additions to the language.

4.1.1 Associated types

The bodies of trait and impl declarations in MicroRust include *associated types*. As with trait functions, we require each trait to have exactly one associated type. It should however be fairly straightforward to add support for an arbitrary number of associated types in a trait. A trait declaration declares an *associated type constructor* A in its body using the syntax type $A : \overline{\beta}$, where $\overline{\beta}$ are bounds on the associated type. In an impl, the associated type A is *instantiated* to some type τ using the declaration type $A \mapsto \tau$. The impl must also ensure that τ satisfies the bounds $\overline{\beta}$ on A that occur in the trait declaration.

The syntax of types is extended with an associated type $A_D \langle \tau, \overline{\tau_i} \rangle$, where, in addition to the associated type constructor A, we include the name of the trait D that A is associated with, the trait's Self parameter τ and the additional trait parameters $\overline{\tau_i}$. In particular, an associated type $A_{\texttt{Foo}} \langle T_1, T_2, T_3 \rangle$ is equivalent to `<T1 as Foo<T2, T3>>::A` in Rust. All of the components of the associated type application are necessary to ensure that the type is matched to the appropriate trait and impl in the general case—to simplify our type system, we do not allow omitting the trait name or the trait parameters, even if they could, in princi-

$e \in \text{TERM}, \quad T, \text{Self} \in \text{TVAR}, \quad S \in \text{SCONS}, \quad \tau \in \text{TYPE}$
$A \in \text{ACONS}, \quad x, f \in \text{VAR}, \quad D \in \text{TRAIT}, \quad \sigma \in \text{TSCHEME}, \quad \theta \in \text{CSCHEME}$

pgm	::=	$\overline{item}$	(programs)
$item$	::=	$\text{struct } S\,\langle\overline{T}\rangle\,\{\overline{x:\tau}\}$	(items)
	|	fun	
	|	$\text{trait } D\,\langle\overline{T}\rangle \text{ for Self where } \overline{\pi}\,\{\text{ type } A:\overline{\beta};\ f:\sigma\,\}$	
	|	$\text{impl}\,\langle\overline{T}\rangle\,D\,\langle\overline{\tau}\rangle \text{ for } \tau \text{ where } \overline{\pi}\,\{\text{ type } A \mapsto \tau;\ fun\,\}$	
fun	::=	$\text{fn } f\,\langle\overline{T}\rangle\,(\overline{x:\tau}) \to \tau \text{ where } \overline{\pi}\,\{e\}$	(functions)
e	::=	$\text{let } x = e \text{ in } e \mid lv := e \mid e;e \mid \&lv \mid *e$	(terms)
	|	$e(\overline{e}) \mid x \mid () \mid e \text{ as } \tau \mid S\{\overline{x:e}\} \mid e.x$	
lv	::=	$x \mid *e \mid lv.x$	(lvalues)
τ	::=	$() \mid T \mid \text{fn}(\overline{\tau}) \to \tau \mid \&\tau \mid S\,\langle\overline{\tau}\rangle \mid A_D\,\langle\tau,\overline{\tau}\rangle$	(types)
σ	::=	$\forall\overline{T}.\overline{\pi} \Rightarrow \tau$	(type schemes)
θ	::=	$\forall\overline{T}.\overline{\pi} \Rightarrow \pi$	(constraint schemes)
π	::=	$\tau : \beta$	(trait constraints)
β	::=	$D\,\langle\overline{\tau},\ A \mapsto \tau\ \rangle \mid D\,\langle\overline{\tau},\ A \mapsto \star\ \rangle$	(trait bounds)
Γ	::=	$\emptyset \mid \Gamma, x:\sigma \mid \Gamma, T \mid \Gamma, S\,\langle\overline{T}\rangle\,\{\overline{x:\tau}\}$	(typing environments)
	|	$\Gamma, (D\,\langle\overline{T}\rangle \text{ where } \overline{\pi}, \text{Self}:\overline{\beta},\ A:\overline{\beta},\ f)$	
Θ	::=	$\emptyset \mid \Theta, \theta$	(constraint environments)

Figure 4.1: MicroRust: syntax

ple, be inferred.

Note that the syntax does not prevent impl declarations from instantiating an associated type with another associated type. This is also allowed by our typing rules, however, to prevent circularity in associated type instantiations, we externally require that each concrete associated type (without unsolved type variables) can be *normalized*, i.e., converted to an equivalent concrete type expression without associated types. For instance, in an impl of trait D_1, it is possible to have an associated type instantiation type $A \mapsto A_{D_2}\,\langle\texttt{bool}\rangle$ as long as $A_{D_2}\,\langle\texttt{bool}\rangle$ normalizes to some concrete type (such as `i32` for example).

4.1.2 Type equality constraints

With the addition of associated types to the language we also need a notion of *type equality* or *equivalence of type expressions*. For example, given an impl

$$\text{impl } \texttt{Foo} \text{ for } \texttt{i32}\ \{\text{type } A \mapsto \texttt{bool};\}$$

we want to be able to conclude that the type expression $\mathtt{A_{Foo}}\langle \mathtt{i32} \rangle$ is equivalent to `bool`. To reflect this idea, we introduce *type equality constraints* of the form $\tau_1 \sim \tau_2$, which we use to convert between equivalent type expressions (such as in the type of a term).

We also extend the syntax of trait bounds to allow parameterizing trait constraints by their associated type: the *extended trait constraint* $\tau : D\,\langle \overline{\tau_i}, A \mapsto \tau_A \rangle$ tells us that type τ implements trait D with additional type parameters $\overline{\tau_i}$, and the corresponding associated type $A_D\,\langle \tau, \overline{\tau_i} \rangle$ is instantiated with the type expression τ_A. A MicroRust trait bound $D\,\langle T_1, A \mapsto T_2 \rangle$ is equivalent to the bound `D<T1, A=T2>` in Rust.[1] The extended trait constraint lets us constrain functions by an associated type as in the following function that is parametric over iterators of 32-bit integers:

```
fn iterate<T>(iterator: T) where T: Iterator<Item=i32> { ... }
```

Rust does not require trait constraints to include their associated type instantiations. To account for such cases, where the associated type instantiation is not known and/or not needed, we include constraints of the form $\tau : D\,\langle \overline{\tau_i}, A \mapsto \star \rangle$ where $\star$ denotes an identity associated type instantiation—an instantiaton of an associated type with itself. In the typing rules we will often use the notation $\tau : D\,\langle \overline{\tau_i} \rangle$ as syntactic sugar for $\tau : D\,\langle \overline{\tau_i}, A \mapsto \star \rangle$.

4.2 Typing rules

4.2.1 Well-formedness judgments

The well-formedness judgments in MicroRust are presented in figure 4.2. At the top of the figure we define a new auxiliary relation $\Gamma \vdash A_D$, which verifies that the associated type constructor A is associated with trait D.

In a departure from NanoRust, the well-formedness judgments include the constraint environment Θ. The environment is needed in the new rule (wf-atype) of the well-formed types judgment, which verifies that an associated type $A_D\,\langle \tau, \overline{\tau_i} \rangle$ is well-formed. All associated types are related to a trait constraint and an associated type only has meaning if its corresponding trait constraint is satisfied. As such, the rule (wf-type) invokes the constraint entailment judgment to verify that its corresponding constraint $\tau : D\,\langle \overline{\tau_i}, A \mapsto \star \rangle$ is entailed. Note that the dependence on the constraint entailment relation in the rule effectively makes the well-formed types and constraint entailment judgments mutually dependent on one another.

Rule (wf-tscheme), used to verify well-formedness of type schemes, extends the constraint environment in its premises with the qualifying constraints of the type scheme. Doing so

[1] A trait constraint in Rust may also include instantiations of associated types of supertraits of the constraint's trait. We do not allow this in our model, however it is a feature we would consider in future work.

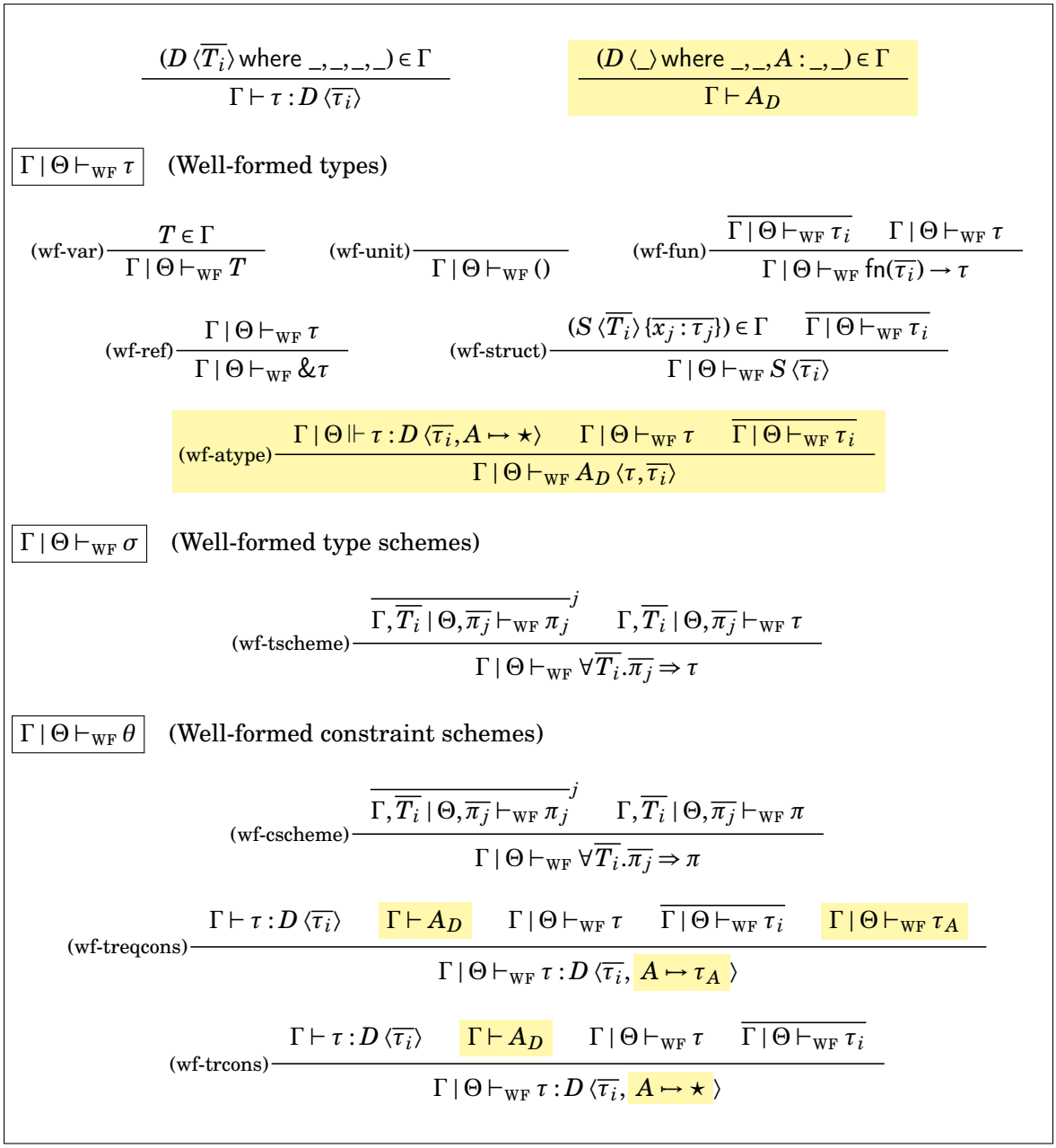

Figure 4.2: MicroRust: well-formedness judgments

allows us to have type schemes that include associated types that are associated with traits by which the type scheme is constrained. For example, in the type scheme

$$\forall T.(T : \mathsf{Foo},\, A_{\mathsf{Foo}}\langle T\rangle : \mathsf{Bar}) \Rightarrow \tau$$

the associated type $A_{\mathsf{Foo}}\langle T\rangle$ is well-formed because the trait constraint $T : \mathsf{Foo}$ is added to the set of impl assumptions in rule (wf-tscheme) and it is in the scope of the rule instance of (wf-atype) that verifies well-formedness of $A_{\mathsf{Foo}}\langle T\rangle$. The rule (wf-cscheme), used to verify well-formedness of constraint schemes, extends Θ similarly in its premises. Rules (wf-treqcons) and (wf-trcons) verify well-formedness of constraints. If a constraint is of the form $\tau : D\langle\overline{\tau_i}, A \mapsto \tau_A\rangle$ then we also verify that the associated type instantiation τ_A is well-formed.

4.2.2 Constraint entailment

Figure 4.3 presents rules used to prove constraint satisfaction. The rules are part of two mutually inductive judgments: the trait constraint entailment judgment $\Gamma \mid \Theta \Vdash \pi$ and the equality constraint entailment judgment $\Gamma \mid \Theta \Vdash \tau \sim \tau$.

The trait constraint entailment judgment contains new rules (c-treq1) and (c-treq2), which prove entailment of a trait constraint, whose type parameters and associated type instantiation (if present) are equivalent to some type expressions, for which the constraint is satisfied. The rule (c-astar) also lets us infer entailment of a trait constraint with an identity associated type instantiation ($A \mapsto \star$) from a corresponding satisfied trait constraint with any associated type instantiation τ_A.

The equality constraint entailment judgment $\Gamma \mid \Theta \Vdash \tau_1 \sim \tau_2$ asserts that type expression τ_1 is equivalent to τ_2 under assumptions in Γ and Θ. Rule (eq-sep) infers a type equality from an associated type instantiation $A \mapsto \tau_A$ in a trait constraint. Rules (eq-refl), (eq-trans) and (eq-sym) respectively reflect the reflexivity, transitivity and symmetry properties of the type equality relation. The remaining rules in the judgment are used to prove equivalence of inductively defined type expressions.

4.2.3 Well-typed terms

Figure 4.4 presents the well-typed terms judgment in MicroRust. The only addition to the judgment is the rule (sub), which allows converting the type expression of a term e to another equivalent type expression. Note that, unlike other rules in the well-typed terms judgment, rule (sub) does not depend on the structure of the term e and thus it can be used at any point in a typing derivation. It would be up to an implementation or a corresponding type inference algorithm to decide when to convert the type of a term.

$\boxed{\Gamma \mid \Theta \Vdash \pi}$ (Trait constraint entailment)

$$\text{(c-ext)}\ \frac{(\forall \overline{T_i}.\overline{\pi_j} \Rightarrow \pi) \in \Theta \qquad \overline{\Gamma \mid \Theta \vdash_{\text{WF}} \tau_i} \qquad \overline{\Gamma \mid \Theta \Vdash [\overline{\tau_i/T_i}]\pi_j}}{\Gamma \mid \Theta \Vdash [\overline{\tau_i/T_i}]\pi}$$

$$\text{(c-treq1)}\ \frac{\Gamma \mid \Theta \Vdash \tau_1 : D\,\langle \overline{\tau_{1i}}, A \mapsto \tau_3 \rangle \qquad \Gamma \mid \Theta \Vdash \tau_1 \sim \tau_2 \qquad \overline{\Gamma \mid \Theta \Vdash \tau_{1i} \sim \tau_{2i}}^{\,i} \qquad \Gamma \mid \Theta \Vdash \tau_3 \sim \tau_4}{\Gamma \mid \Theta \Vdash \tau_2 : D\,\langle \overline{\tau_{2i}}, A \mapsto \tau_4 \rangle}$$

$$\text{(c-treq2)}\ \frac{\Gamma \mid \Theta \Vdash \tau_1 : D\,\langle \overline{\tau_{1i}}, A \mapsto \star \rangle \qquad \Gamma \mid \Theta \Vdash \tau_1 \sim \tau_2 \qquad \overline{\Gamma \mid \Theta \Vdash \tau_{1i} \sim \tau_{2i}}^{\,i}}{\Gamma \mid \Theta \Vdash \tau_2 : D\,\langle \overline{\tau_{2i}}, A \mapsto \star \rangle}$$

$$\text{(c-astar)}\ \frac{\Gamma \mid \Theta \Vdash \tau : D\,\langle \overline{\tau_i}, A \mapsto \tau_A \rangle}{\Gamma \mid \Theta \Vdash \tau : D\,\langle \overline{\tau_i}, A \mapsto \star \rangle}$$

$\boxed{\Gamma \mid \Theta \Vdash \tau \sim \tau}$ (Equality constraint entailment)

$$\text{(eq-sep)}\ \frac{\Gamma \mid \Theta \Vdash \tau : D\,\langle \overline{\tau_i}, A \mapsto \tau_A \rangle}{\Gamma \mid \Theta \Vdash A_D\,\langle \tau, \overline{\tau_i} \rangle \sim \tau_A} \qquad \text{(eq-refl)}\ \frac{}{\Gamma \mid \Theta \Vdash \tau \sim \tau} \qquad \text{(eq-trans)}\ \frac{\Gamma \mid \Theta \Vdash \tau_1 \sim \tau_3 \qquad \Gamma \mid \Theta \Vdash \tau_3 \sim \tau_2}{\Gamma \mid \Theta \Vdash \tau_1 \sim \tau_2}$$

$$\text{(eq-sym)}\ \frac{\Gamma \mid \Theta \Vdash \tau_2 \sim \tau_1}{\Gamma \mid \Theta \Vdash \tau_1 \sim \tau_2} \qquad \text{(eq-struct)}\ \frac{\overline{\Gamma \mid \Theta \Vdash \tau_{1i} \sim \tau_{2i}}^{\,i}}{\Gamma \mid \Theta \Vdash S\,\langle \overline{\tau_{1i}} \rangle \sim S\,\langle \overline{\tau_{2i}} \rangle}$$

$$\text{(eq-ref)}\ \frac{\Gamma \mid \Theta \Vdash \tau_1 \sim \tau_2}{\Gamma \mid \Theta \Vdash \&\tau_1 \sim \&\tau_2} \qquad \text{(eq-fun)}\ \frac{\overline{\Gamma \mid \Theta \Vdash \tau_{1i} \sim \tau_{2i}}^{\,i} \qquad \Gamma \mid \Theta \Vdash \tau_1 \sim \tau_2}{\Gamma \mid \Theta \Vdash \mathsf{fn}(\overline{\tau_{1i}}) \to \tau_1 \sim \mathsf{fn}(\overline{\tau_{2i}}) \to \tau_2}$$

$$\text{(eq-atype)}\ \frac{\Gamma \mid \Theta \Vdash \tau_1 \sim \tau_2 \qquad \overline{\Gamma \mid \Theta \Vdash \tau_{1i} \sim \tau_{2i}}}{\Gamma \mid \Theta \Vdash A_D\,\langle \tau_1, \overline{\tau_{1i}} \rangle \sim A_D\,\langle \tau_2, \overline{\tau_{2i}} \rangle}$$

Figure 4.3: MicroRust: constraint entailment

4.2.4 Well-typed items

In the well-typed items judgment in figure 4.5, rules (trait) and (impl) contain additions necessary to accommodate the inclusion of associated types (those additions are highlighted in the figure).

In the trait typing rule, (trait), the bounds on the associated type $A : \overline{\beta_a}$ are propagated much like supertrait bounds and thus are treated similarly. The trait information tuple output by the rule also includes the name of the trait's associated type constructor and its bounds.

The impl typing rule, (impl), verifies that the associated type instantiation provided in

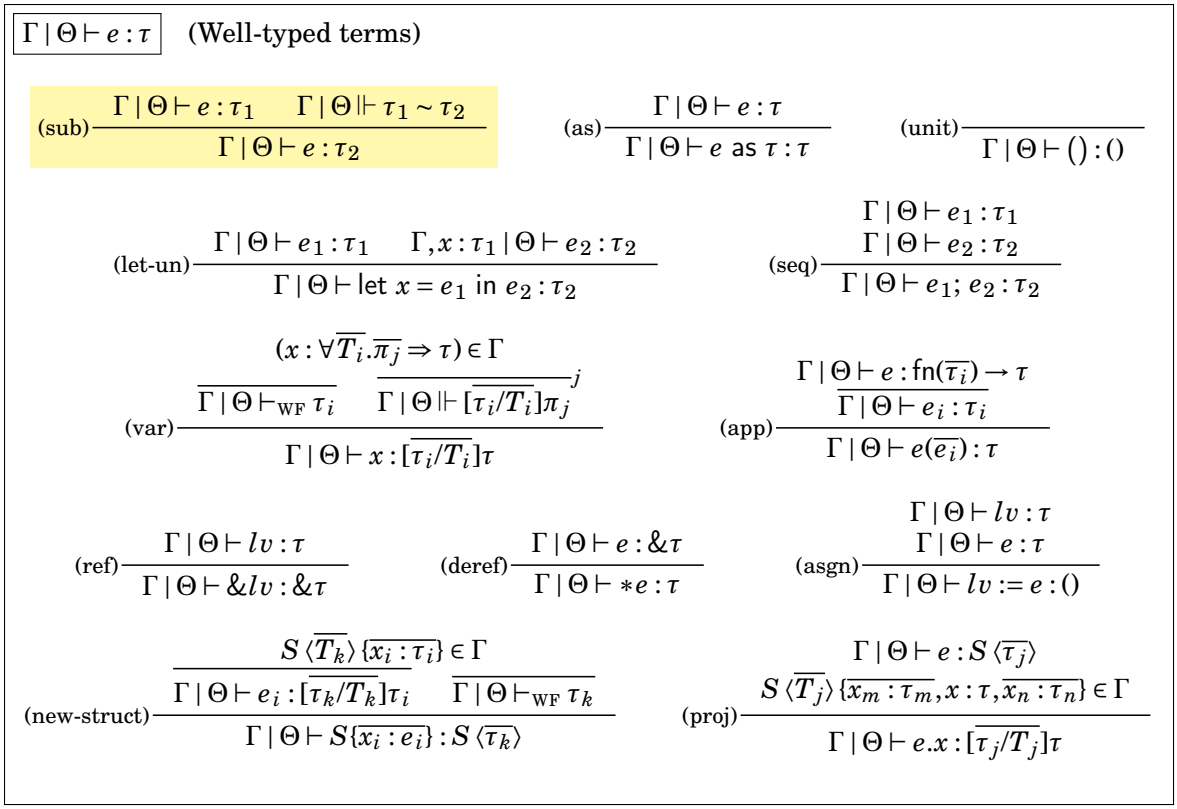

Figure 4.4: MicroRust: well-typed terms

the body of an impl satisfies the bounds on the associated type specified in the implemented trait declaration. To prevent trivial resolution of those constraints through use of the propagated associated type constraint schemes produced by the trait typing rule, we use a trimmed constraint environment Θ^*, where the propagated associated type constraint schemes (along with supertrait constraint schemes) are omitted. Finally, the rule produces a trait constraint that contains the associated type instantiation defined in the impl.

$\boxed{\Gamma \mid \Theta \vdash \mathit{item} : \Gamma \mid \Theta}$ (Well-typed items)

$$\text{(struct)}\ \frac{\overline{\Gamma, \overline{T_j} \mid \Theta \vdash_{\text{WF}} \tau_i}^{\,i}}{\Gamma \mid \Theta \vdash \mathsf{struct}\ S\ \langle \overline{T_j} \rangle\ \{\overline{x_i : \tau_i}\} : [S\ \langle \overline{T_j} \rangle\ \{\overline{x_i : \tau_i}\}] \mid \emptyset}$$

$$\text{(fun)}\ \frac{\begin{array}{c}\sigma = \forall \overline{T_k}.\overline{\pi_j} \Rightarrow \mathsf{fn}(\overline{\tau_i}) \to \tau \qquad \Gamma \mid \Theta \vdash_{\text{WF}} \sigma \\ \Gamma' = \Gamma, \overline{T_k}, \overline{x_i : \tau_i} \qquad \Theta' = \Theta, \overline{\pi_j} \qquad \Gamma' \mid \Theta' \vdash e : \tau\end{array}}{\Gamma \mid \Theta \vdash \mathsf{fn}\ f\ \langle \overline{T_k} \rangle\ (\overline{x_i : \tau_i}) \to \tau\ \mathsf{where}\ \overline{\pi_j}\ \{\, e \,\} : [f : \sigma] \mid \emptyset}$$

$$\text{(trait)}\ \frac{\begin{array}{c}\{\overline{\pi_j}\} \equiv \{(\overline{\mathsf{Self} : \beta_s}), \overline{\pi_h}\} \qquad \overline{T_p} = \mathsf{Self}, \overline{T_i} \qquad \overline{\Gamma \mid \Theta \vdash_{\text{WF}} \forall \overline{T_p}.(\mathsf{Self} : D\ \langle \overline{T_i} \rangle) \Rightarrow \pi_j}^{\,j} \\ \sigma = \forall \overline{T_k}.\overline{\pi_l} \Rightarrow \mathsf{fn}(\overline{\tau_m}) \to \tau_R \qquad \sigma' = \forall \overline{T_p}.(\mathsf{Self} : D\ \langle \overline{T_i} \rangle) \Rightarrow \sigma \\ \Gamma \mid \Theta \vdash_{\text{WF}} \sigma' \qquad \overline{\Gamma \mid \Theta \vdash_{\text{WF}} \forall \overline{T_p}.(\mathsf{Self} : D\ \langle \overline{T_i} \rangle) \Rightarrow A_D\ \langle \overline{T_p} \rangle : \beta_a}^{\,a}\end{array}}{\Gamma \mid \Theta \vdash \begin{array}{l}\mathsf{trait}\ D\ \langle \overline{T_i} \rangle\ \mathsf{for}\ \mathsf{Self} \\ \mathsf{where}\ \overline{\pi_j}\ \{ \\ \quad \mathsf{type}\ A : \overline{\beta_a}; \\ \quad f : \sigma;\ \}\end{array} : \begin{array}{l}[(D\ \langle \overline{T_i} \rangle\ \mathsf{where}\ \overline{\pi_h}, \mathsf{Self} : \overline{\beta_s}, \ A : \overline{\beta_a}, \ f), f : \sigma'] \mid \\ [\overline{\forall \overline{T_p}.(\mathsf{Self} : D\ \langle \overline{T_i} \rangle) \Rightarrow \mathsf{Self} : \beta_s}^{\,s}, \\ \quad \overline{\forall \overline{T_p}.(\mathsf{Self} : D\ \langle \overline{T_i} \rangle) \Rightarrow A_D\ \langle \overline{T_p} \rangle : \beta_a}^{\,a}\ ]\end{array}}$$

$$\text{(impl)}\ \frac{\begin{array}{c}(D\ \langle \overline{T_i} \rangle\ \mathsf{where}\ \overline{\pi_h}, \mathsf{Self} : \overline{\beta_s}, \ A : \overline{\beta_a}, \ f) \in \Gamma \qquad \overline{\tau_p} = \tau, \overline{\tau_i} \qquad \overline{T_p} = \mathsf{Self}, \overline{T_i} \\ \overline{T_k \in FV(\overline{\tau_p})} \qquad \Gamma \mid \Theta \vdash_{\text{WF}} \forall \overline{T_k}.\overline{\pi_j} \Rightarrow \tau : D\ \langle \overline{\tau_i} \rangle \\ \Theta^* = \Theta \setminus \{\overline{\forall \overline{T_p}.(\mathsf{Self} : D\ \langle \overline{T_i} \rangle) \Rightarrow \mathsf{Self} : \beta_s}^{\,s}, \ \overline{\forall \overline{T_p}.(\mathsf{Self} : D\ \langle \overline{T_i} \rangle) \Rightarrow A_D\ \langle \overline{T_p} \rangle : \beta_a}^{\,a}\} \\ \overline{\Gamma, \overline{T_k} \mid \Theta^*, \overline{\pi_j} \Vdash [\overline{\tau_p / T_p}](\mathsf{Self} : \beta_s)}^{\,s} \qquad \overline{\Gamma, \overline{T_k} \mid \Theta^*, \overline{\pi_j} \Vdash [\overline{\tau_p / T_p}]\pi_h}^{\,h} \\ \Gamma, \overline{T_k} \mid \Theta^*, \overline{\pi_j} \vdash_{\text{WF}} \tau_A \qquad \overline{\Gamma, \overline{T_k} \mid \Theta^*, \overline{\pi_j} \Vdash \tau_A : [\overline{\tau_p / T_p}]\beta_a}^{\,a} \\ (f : \forall \overline{T_p}.(\mathsf{Self} : D\ \langle \overline{T_i} \rangle) \Rightarrow \sigma) \in \Gamma \qquad \Gamma, \overline{T_k} \mid \Theta^*, \overline{\pi_j} \vdash \mathit{fun} : [f : \sigma'] \mid \emptyset \qquad \sigma' = [\overline{\tau_p / T_p}]\sigma\end{array}}{\Gamma \mid \Theta \vdash \begin{array}{l}\mathsf{impl}\ \langle \overline{T_k} \rangle\ D\ \langle \overline{\tau_i} \rangle\ \mathsf{for}\ \tau\ \mathsf{where}\ \overline{\pi_j}\ \{ \\ \quad \mathsf{type}\ A \mapsto \tau_A; \ \mathit{fun}\ \}\end{array} : \emptyset \mid [\forall \overline{T_k}.\overline{\pi_j} \Rightarrow \tau : D\ \langle \overline{\tau_i}, \ A \mapsto \tau_A \rangle]}$$

$\boxed{\vdash_P \mathit{pgm} : \Gamma \mid \Theta}$ (Well-typed programs)

$$\text{(pgm)}\ \frac{\overline{\Gamma \mid \Theta \vdash \mathit{item}_i : \Gamma_i \mid \Theta_i} \qquad \Gamma = \overline{\Gamma_i} \qquad \Theta = \overline{\Theta_i}}{\vdash_P \overline{\mathit{item}_i} : \Gamma \mid \Theta}$$

Figure 4.5: MicroRust: well-typed items and programs

Chapter 5

MiniRust: MicroRust with trait objects

In this chapter we present MiniRust: an extension of MicroRust where we include trait objects: trait-based existential objects that introduce dynamic dispatch to the language.

5.1 Syntax

Figure 5.1 presents the syntax of MiniRust.

5.1.1 Trait objects

A trait object is an existential object that encapsulates the functionality provided by its trait. The Self type of the object's trait is *existentially quantified* (hidden from the type system's field of view) [28]. The type of a trait object, $\&(\exists T : D\,\langle \overline{u_j}, \overline{A_{D_i}\,\langle T, \overline{u}\rangle \sim u_i}\rangle)$, is composed of the reference operator & and the *trait object descriptor* $\exists T : D\,\langle \overline{u_j}, \overline{A_{D_i}\,\langle T, \overline{u}\rangle \sim u_i}\rangle$. The use of the reference operator & follows from the trait object often being called a *fat pointer* in Rust. However, a trait object is not really a pointer—it is really an object that contains additional information about its encapsulated value—thus we consider the reference type operator & as overloaded in MiniRust. The trait object descriptor (or just descriptor for short) looks like a trait constraint. This is intended to emphasize that the existentially quantified type T implements the trait D. Since only Self is existentially quantified, the additional parameters of the trait appear in the trait object type as $\overline{u_j}$ (u ranges over types in MiniRust). Finally, since associated type instantiations are determined by the trait parameters, they are also specified in the type of the trait object as $\overline{A_{D_i}\,\langle T, \overline{u}\rangle \sim u_i}$. They include not only types associated with the trait of the descriptor, but also associated types of all supertraits in the trait's supertrait hierarchy (in Rust, an object's supertrait associated types must also be explicitly declared in the object's type). This allows us, for example, to call trait functions that return a value whose type is an associated type of a supertrait of the object's trait—the

$e \in \text{TERM}$, $\mathit{Self}, X \in \text{XVAR}$, $\mathsf{Self}_{\mathsf{T}}, T \in \text{TVAR}$, $\mathsf{Self}_{\mathsf{U}}, U \in \text{UVAR}$, $S \in \text{SCONS}$,
$\tau \in \text{STYPE}$, $u \in \text{TYPE}$, $A^{\mathsf{T}} \in \text{TACONS}$, $A^{\mathsf{U}} \in \text{UACONS}$, $A \in \text{ACONS}$, $x, f \in \text{VAR}$,
$D \in \text{TRAIT}$, $\sigma \in \text{TSCHEME}$, $\theta \in \text{CSCHEME}$, $\mathit{obj} \in \{\text{obj-safe}, \text{obj-unsafe}\}$

pgm	::=	$\overline{\mathit{item}}$	(programs)
item	::=	$\mathsf{struct}\ S\,\langle\overline{X}\rangle\,\{\overline{x:\tau}\}$	(items)
	|	fun	
	|	$\mathsf{trait}\ D\,\langle\overline{X}\rangle\ \mathsf{for}\ \mathit{Self}\ \mathsf{where}\ \overline{\pi}\,\{\,\mathsf{type}\ A:\overline{\beta},\ f:\sigma\,\}$	
	|	$\mathsf{impl}\,\langle\overline{X}\rangle\,D\,\langle\overline{u}\rangle\ \mathsf{for}\ u\ \mathsf{where}\ \overline{\pi}\,\{\,\mathsf{type}\ A \mapsto u,\ \mathit{fun}\,\}$	
fun	::=	$\mathsf{fn}\ f\,\langle\overline{X}\rangle\,(\overline{x:\tau}) \to \tau\ \mathsf{where}\ \overline{\pi}\,\{e\}$	(functions)
e	::=	$\mathsf{let}\ x = e\ \mathsf{in}\ e \mid lv := e \mid e;e \mid \&lv \mid *e$	(terms)
	|	$e(\overline{e}) \mid x \mid () \mid e\ \mathsf{as}\ \tau \mid S\{\overline{x:e}\} \mid e.x$	
lv	::=	$x \mid *e \mid lv.x$	(lvalues)
τ	::=	$() \mid T \mid \mathsf{fn}(\overline{\tau}) \to \tau \mid \&u \mid S\,\langle\overline{u}\rangle \mid A^{\mathsf{T}}_D\,\langle u, \overline{u}\rangle$	(s-types)
u	::=	$\tau \mid U \mid \exists T : D\,\langle\overline{u}, \overline{A_D\,\langle T, \overline{u}\rangle \sim u}\rangle \mid A^{\mathsf{U}}_D\,\langle u, \overline{u}\rangle$	(types)
X	::=	$T \mid U$	(type variables)
Self	::=	$\mathsf{Self}_{\mathsf{T}} \mid \mathsf{Self}_{\mathsf{U}}$	(self type variables)
A	::=	$A^{\mathsf{T}} \mid A^{\mathsf{U}}$	(associated type constructors)
σ	::=	$\forall \overline{X}.\overline{\pi} \Rightarrow \tau$	(type schemes)
θ	::=	$\forall \overline{X}.\overline{\pi} \Rightarrow \pi$	(constraint schemes)
π	::=	$u : \beta$	(constraints)
β	::=	$D\,\langle\overline{u}, A^{\mathsf{T}} \mapsto \tau\rangle \mid D\,\langle\overline{u}, A^{\mathsf{U}} \mapsto u\rangle \mid D\,\langle\overline{u}, A \mapsto \star\rangle$	(trait bounds)
Γ	::=	$\emptyset \mid \Gamma, x:\sigma \mid \Gamma, X \mid \Gamma, S\,\langle\overline{X}\rangle\,\{\overline{x:\tau}\}$	(typing environments)
	|	$\Gamma, (D\,\langle\overline{X}\rangle\ \mathsf{where}\ \overline{\pi}, \mathit{Self} : \overline{\beta}, A : \overline{\beta}, f, \mathit{obj})$	
Θ	::=	$\emptyset \mid \Theta, \theta$	(constraint environments)

Figure 5.1: MiniRust: syntax

type equality in the type of the trait object can be used to convert the associated type to its concrete representation. Note that two trait objects of the same trait must have the same associated type instantiations to have the same type.

The syntax of the descriptor in MiniRust differs from its equivalent representation in Rust in that we explicitly introduce a type variable T that represents the hidden Self type. For example, given a trait `Graph` that has two associated types `Node` and `Edge`, its trait object in Rust has type `&Graph<Node=MyNode, Edge=MyEdge>`. Its equivalent MiniRust representation is $\&(\exists T : \mathtt{Graph}\langle \mathtt{Node}^{\mathsf{T}}_{\mathtt{Graph}}\,\langle T\rangle \sim \mathtt{MyNode}, \mathtt{Edge}^{\mathsf{T}}_{\mathtt{Graph}}\,\langle T\rangle \sim \mathtt{MyEdge}\rangle)$. While this notation is heavier, it makes certain typing rules simpler, since they don't need to rely on the typing environment to know the exact type parameters of a supertrait's associated type.

5.1.2 S-types

One purpose of types in Rust is to assist the compilation process by providing the size of program values, so that they can be properly allocated on the stack. In some cases, however, the type of a value does not provide that information. Such a type is said to be *dynamically sized*. A notable instance of a dynamically sized type (DST) is the *trait type*: it describes values whose concrete types implement a given trait. Such values may have different sizes, however their type is ignorant of them. As such, a trait type, and a DST in general, may not be assigned directly to a value.

In MiniRust we introduce a similar distinction between directly storable *s-types* τ, which model Rust's statically sized types, and indirectly storable *types* u, which model possibly-dynamically sized types. An s-type classifies values that can be stored directly in memory (specifically, in the *store*, as we will see in chapter 6). The trait object descriptor, which models Rust's trait type, is part of the set of types but it is not an s-type. However, a 'reference to a descriptor'—a trait object—is an s-type.

To allow generic items to be parameterized by types and s-types, we distinguish between type and s-type variables: U and T respectively. In Rust, the distinction between dynamically and statically sized types is expressed as a bound on a type variable: `T: Sized` represents a statically sized type variable, while `T: ?Sized` represents a possibly-dynamically sized type variable. In cases where the bound is omitted, a default is assumed (the default is usually `Sized`, except for the `Self` type of a trait, whose default bound is `?Sized`).

The syntactic approach to distinguishing between the two kinds of type variables in MiniRust is slightly more restrictive. In particular, Rust trait declarations may contain methods that restrict `Self` to be `Sized` even if at the trait level, `Self` is `?Sized`. In MiniRust there is no direct way to restrict the sizedness of a trait parameter on a per-function basis (although it's not really relevant, since each MiniRust trait only contains one function). However, by reducing the notational burden associated with sizedness bounds we obtain a more concise syntax.

For the cases where we do not know (or care about) the *kind* of a type variable—whether it is a type or s-type variable—we include a metavariable X that ranges over both kinds of variables. Similarly, the special variable *Self* is now a metavariable that ranges over $\mathsf{Self}_{\mathsf{T}}$ and $\mathsf{Self}_{\mathsf{U}}$, which represent an s-type and type variable respectively.

Associated types can also be instantiated with trait object descriptors in MiniRust (in Rust they can be instantiated with any DST). For that reason, similarly to type variables, we distinguish syntactically between associated type and s-type constructors as A^{U} and A^{T} respectively. A is then a metavariable that ranges over both kinds of associated type constructors. If an associated type represents an s-type ($A^{\mathsf{T}}_D\langle\ldots\rangle$), it can only be instantiated with an s-type. We define the following predicate, which we use in our typing rules to protect

$$\frac{(D\,\langle\overline{X_i}\rangle \text{ where } \_, Self : \_, \_, \_, \_) \in \Gamma \quad [u/Self][\overline{u_i/X_i}]\ defined}{\Gamma \vdash u : D\,\langle\overline{u_i}\rangle} \qquad \frac{(D\,\langle\_\rangle \text{ where } \_, \_, A : \_, \_, \_) \in \Gamma}{\Gamma \vdash A_D}$$

$$\frac{(D\,\langle\_\rangle \text{ where } \_, \_, \_, \_, \text{obj-safe}) \in \Gamma}{\Gamma \vdash D\ obj\text{-}safe} \qquad \frac{S\,\langle\overline{X_i}\rangle\,\{\ldots\} \in \Gamma \quad \overline{[u_i/X_i]\ defined}}{\Gamma \vdash S\,\langle\overline{u_i}\rangle}$$

Figure 5.2: MiniRust: auxiliary relations

that requirement:

$$\overline{[A^{\mathsf{T}} \mapsto \tau]\ defined} \qquad \overline{[A^{\mathsf{U}} \mapsto u]\ defined}$$

With the distinction of type and s-type variables, we must also revisit our definition of type substitution to prevent an s-type variable from being instantiated with a type that is not an s-type (recall that all s-types are types). A substitution $[u/X]$ is only defined when the predicate $[u/X]\,defined$ holds. The predicate is defined as follows:

$$\overline{[u/U]\ \ defined} \qquad \overline{[\tau/T]\ \ defined}$$

For example, a substitution $[u/T]$ where $u \notin \text{STYPE}$ is undefined.

5.1.3 Object-safe traits

Not all traits can be used to create trait objects. Rust (and MiniRust) has a notion of *object safety* that restricts which traits can be used to create trait objects. If a trait satisfies the object safety requirements then its resulting trait information tuple will reflect that in its last field, being obj-safe. Otherwise, a trait that's not object-safe will have obj-unsafe as the last field of its information tuple. We discuss object safety in detail in section 5.3.

5.2 Type system

5.2.1 Auxiliary relations

Figure 5.2 presents auxiliary relations that extract information from trait information tuples in the typing environment Γ. The relation $\Gamma \vdash u : D\,\langle\overline{u_i}\rangle$ is extended to verify that the trait parameter instantiations in the constraints $u : D\,\langle\overline{u_i}\rangle$ have kinds that are compatible with the trait declaration.

The relation $\Gamma \vdash A_D$ asserts that the type constructor A is associated with trait D. Since

A_D is a metavariable ranging over A_D^{T} and A_D^{U}, the relation informs us of the associated type's kind.

In the same figure we also find the relation $\Gamma \vdash D$ *obj-safe*, which asserts that a trait D is object-safe. Finally, the relation $\Gamma \vdash S\,\langle\overline{u_i}\rangle$ verifies that S is a struct in the typing environment Γ, and that its type parameter instantiations $\overline{u_i}$ have the right kinds.

5.2.2 Well-formed types

Figure 5.3 presents the MiniRust well-formedness judgments. A major difference from the corresponding judgments in MicroRust is that generic constructs are now parameterized by variables X, which can be either type (U) or s-type (T) variables. In rules (wf-struct) and (wf-trcons) we use the auxiliary relations $\Gamma \vdash S\,\langle\overline{u}\rangle$ and $\Gamma \vdash u : D\,\langle\overline{u}\rangle$ respectively to ensure that the type variables are instantiated with types of appropriate kind.

As the language of types in MiniRust is extended with trait objects, the well-formed types judgment contains a new rule (wf-desc), used to verify that a trait object descriptor is well-formed. In its premise $\overline{\Gamma, T \mid \Theta, T : D\,\langle\overline{u_i}\rangle \vdash_{\mathrm{WF}} A_{D_j}\,\langle T, \overline{u_s}_j\rangle}^{\,j}$ we add the existential type variable T and the constraint $T : D\,\langle\overline{u_i}\rangle$ to the sets of assumptions. Since supertrait constraints are propagated, having a constraint $T : D\,\langle\overline{u_i}\rangle$ in the set of assumptions allows us to infer that the corresponding supertrait constraints of the form $T : D_j\,\langle\overline{u}\rangle$ are satisfied, with an appropriate choice of $\overline{u}$. As the associated types on the left-hand side of the type equalities in the descriptor are associated with the descriptor's trait and its supertraits, this enables us to prove their well-formedness, based on rule (wf-type). When verifying well-formedness of the right-hand sides of the type equalities $\overline{u_j}$, we do not extend the sets of assumptions. By excluding T from the well-formed type judgment for $\overline{u_j}$, we specifically do not let T occur in $\overline{u_j}$ in the interest of preserving type safety. If some type $u \in \overline{u_j}$ is part of the return type of a trait function call and it contains T, then the existential variable T escapes its scope [28].

5.2.3 Constraint entailment

The constraint entailment judgments are presented in figure 5.4. The trait constraint entailment judgment is the same as in MicroRust, with the exception of using u instead of τ for trait parameters. In rule (c-ext), instead of using the predicate $[\overline{u_i/X_i}]\,\mathit{defined}$ we directly apply the substitution, as it's only defined when the substitution's components have compatible kinds. The equality constraint entailment judgment includes a new rule, (eq-obj), that proves equivalence of two trait object descriptors by verifying that their respective components are equal.

$\boxed{\Gamma \mid \Theta \vdash_{\text{WF}} u}$ (Well-formed types)

$$\text{(wf-var)}\ \frac{X \in \Gamma}{\Gamma \mid \Theta \vdash_{\text{WF}} X} \qquad \text{(wf-unit)}\ \frac{}{\Gamma \mid \Theta \vdash_{\text{WF}} ()} \qquad \text{(wf-fun)}\ \frac{\overline{\Gamma \mid \Theta \vdash_{\text{WF}} \tau_i} \quad \Gamma \mid \Theta \vdash_{\text{WF}} \tau}{\Gamma \mid \Theta \vdash_{\text{WF}} \mathsf{fn}(\overline{\tau_i}) \to \tau}$$

$$\text{(wf-ref)}\ \frac{\Gamma \mid \Theta \vdash_{\text{WF}} u}{\Gamma \mid \Theta \vdash_{\text{WF}} \&u} \qquad \text{(wf-struct)}\ \frac{\Gamma \vdash S\,\langle \overline{u_i} \rangle \quad \overline{\Gamma \mid \Theta \vdash_{\text{WF}} u_i}}{\Gamma \mid \Theta \vdash_{\text{WF}} S\,\langle \overline{u_i} \rangle}$$

$$\text{(wf-atype)}\ \frac{\Gamma \mid \Theta \Vdash u : D\,\langle \overline{u_i}, A \mapsto \star \rangle \quad \Gamma \mid \Theta \vdash_{\text{WF}} u \quad \overline{\Gamma \mid \Theta \vdash_{\text{WF}} u_i}}{\Gamma \mid \Theta \vdash_{\text{WF}} A_D\,\langle u, \overline{u_i} \rangle}$$

$$\text{(wf-desc)}\ \frac{\Gamma \vdash D\ \textit{obj-safe} \quad \overline{\Gamma \mid \Theta \vdash_{\text{WF}} u_i} \quad \Gamma \vdash \mathsf{Self}_{\mathsf{U}} : D\,\langle \overline{u_i} \rangle \quad \overline{\Gamma, T \mid \Theta, T : D\,\langle \overline{u_i} \rangle \vdash_{\text{WF}} A_{Dj}\,\langle T, \overline{u_{kj}} \rangle}^{\,j} \quad \overline{\Gamma \mid \Theta \vdash_{\text{WF}} u_j}}{\Gamma \mid \Theta \vdash_{\text{WF}} \exists T : D\,\langle \overline{u_i}, \overline{A_{Dj}\,\langle T, \overline{u_{kj}} \rangle \sim u_j} \rangle}$$

$\boxed{\Gamma \mid \Theta \vdash_{\text{WF}} \sigma}$ (Well-formed type schemes)

$$\text{(wf-tscheme)}\ \frac{\overline{\Gamma, \overline{X_i} \mid \Theta, \overline{\pi_j} \vdash_{\text{WF}} \pi_j}^{\,j} \quad \Gamma, \overline{X_i} \mid \Theta, \overline{\pi_j} \vdash_{\text{WF}} \tau}{\Gamma \mid \Theta \vdash_{\text{WF}} \forall \overline{X_i}.\overline{\pi_j} \Rightarrow \tau}$$

$\boxed{\Gamma \mid \Theta \vdash_{\text{WF}} \theta}$ (Well-formed constraint schemes)

$$\text{(wf-cscheme)}\ \frac{\overline{\Gamma, \overline{X_i} \mid \Theta, \overline{\pi_j} \vdash_{\text{WF}} \pi_j} \quad \Gamma, \overline{X_i} \mid \Theta, \overline{\pi_j} \vdash_{\text{WF}} \pi}{\Gamma \mid \Theta \vdash_{\text{WF}} \forall \overline{X_i}.\overline{\pi_j} \Rightarrow \pi}$$

$$\text{(wf-treqcons)}\ \frac{\Gamma \vdash u : D\,\langle \overline{u_i} \rangle \quad \Gamma \vdash A_D \quad \Gamma \mid \Theta \vdash_{\text{WF}} u \quad \overline{\Gamma \mid \Theta \vdash_{\text{WF}} u_i} \quad \Gamma \mid \Theta \vdash_{\text{WF}} u_A}{\Gamma \mid \Theta \vdash_{\text{WF}} u : D\,\langle \overline{u_i}, A \mapsto u_A \rangle}$$

$$\text{(wf-trcons)}\ \frac{\Gamma \vdash u : D\,\langle \overline{u_i} \rangle \quad \Gamma \vdash A_D \quad \Gamma \mid \Theta \vdash_{\text{WF}} u \quad \overline{\Gamma \mid \Theta \vdash_{\text{WF}} u_i}}{\Gamma \mid \Theta \vdash_{\text{WF}} u : D\,\langle \overline{u_i}, A \mapsto \star \rangle}$$

Figure 5.3: MiniRust: well-formedness judgments

$\boxed{\Gamma \mid \Theta \Vdash \pi}$ (Trait constraint entailment)

$$\text{(c-ext)}\ \frac{(\forall \overline{X_i}.\overline{\pi_j} \Rightarrow \pi) \in \Theta \qquad \overline{\Gamma \mid \Theta \vdash_{\text{WF}} u_i} \qquad \overline{\Gamma \mid \Theta \Vdash [\overline{u_i/X_i}]\pi_j}}{\Gamma \mid \Theta \Vdash [\overline{u_i/X_i}]\pi}$$

$$\text{(c-treq1)}\ \frac{\Gamma \mid \Theta \Vdash u_1 : D\,\langle \overline{u_{1i}}, A \mapsto u_3\rangle \qquad \Gamma \vdash u_2 : D\,\langle \overline{u_{2i}}\rangle \qquad \Gamma \mid \Theta \Vdash u_1 \sim u_2 \qquad \overline{\Gamma \mid \Theta \Vdash u_{1i} \sim u_{2i}}^{\,i} \qquad \Gamma \mid \Theta \Vdash u_3 \sim u_4}{\Gamma \mid \Theta \Vdash u_2 : D\,\langle \overline{u_{2i}}, A \mapsto u_4\rangle}$$

$$\text{(c-treq2)}\ \frac{\Gamma \mid \Theta \Vdash u_1 : D\,\langle \overline{u_{1i}}, A \mapsto u_3\rangle \qquad \Gamma \vdash u_2 : D\,\langle \overline{u_{2i}}\rangle \qquad \Gamma \mid \Theta \Vdash u_1 \sim u_2 \qquad \overline{\Gamma \mid \Theta \Vdash u_{1i} \sim u_{2i}}^{\,i}}{\Gamma \mid \Theta \Vdash u_2 : D\,\langle \overline{u_{2i}}, A \mapsto \star\rangle}$$

$$\text{(c-astar)}\ \frac{\Gamma \mid \Theta \Vdash u : D\,\langle \overline{u_i}, A \mapsto u_A\rangle}{\Gamma \mid \Theta \Vdash u : D\,\langle \overline{u_i}, A \mapsto \star\rangle}$$

$\boxed{\Gamma \mid \Theta \Vdash u \sim u}$ (Equality constraint entailment)

$$\text{(eq-sep)}\ \frac{\Gamma \mid \Theta \Vdash u : D\,\langle \overline{u_i}, A \mapsto u_A\rangle}{\Gamma \mid \Theta \Vdash A_D\,\langle u, \overline{u_i}\rangle \sim u_A} \qquad \text{(eq-refl)}\ \frac{}{\Gamma \mid \Theta \Vdash u \sim u} \qquad \text{(eq-trans)}\ \frac{\Gamma \mid \Theta \Vdash u_1 \sim u_3 \qquad \Gamma \mid \Theta \Vdash u_3 \sim u_2}{\Gamma \mid \Theta \Vdash u_1 \sim u_2}$$

$$\text{(eq-sym)}\ \frac{\Gamma \mid \Theta \Vdash u_2 \sim u_1}{\Gamma \mid \Theta \Vdash u_1 \sim u_2} \qquad \text{(eq-struct)}\ \frac{\Gamma \vdash S\,\langle \overline{u_{1i}}\rangle \qquad \Gamma \vdash S\,\langle \overline{u_{2i}}\rangle \qquad \overline{\Gamma \mid \Theta \Vdash u_{1i} \sim u_{2i}}^{\,i}}{\Gamma \mid \Theta \Vdash S\,\langle \overline{u_{1i}}\rangle \sim S\,\langle \overline{u_{2i}}\rangle}$$

$$\text{(eq-ref)}\ \frac{\Gamma \mid \Theta \Vdash u_1 \sim u_2}{\Gamma \mid \Theta \Vdash \&u_1 \sim \&u_2} \qquad \text{(eq-fun)}\ \frac{\overline{\Gamma \mid \Theta \Vdash \tau_{1i} \sim \tau_{2i}}^{\,i} \qquad \Gamma \mid \Theta \Vdash \tau_1 \sim \tau_2}{\Gamma \mid \Theta \Vdash \mathsf{fn}(\overline{\tau_{1i}}) \to \tau_1 \sim \mathsf{fn}(\overline{\tau_{2i}}) \to \tau_2}$$

$$\text{(eq-atype)}\ \frac{\Gamma \vdash u_1 : D\,\langle \overline{u_{1i}}\rangle \qquad \Gamma \vdash u_2 : D\,\langle \overline{u_{2i}}\rangle \qquad \Gamma \mid \Theta \Vdash u_1 \sim u_2 \qquad \overline{\Gamma \mid \Theta \Vdash u_{1i} \sim u_{2i}}}{\Gamma \mid \Theta \Vdash A_D\,\langle u_1, \overline{u_{1i}}\rangle \sim A_D\,\langle u_2, \overline{u_{2i}}\rangle}$$

$$\text{(eq-obj)}\ \frac{\begin{array}{c}\Gamma \vdash \mathsf{Self}_{\mathsf{U}} : D\,\langle \overline{u_{1i}}\rangle \qquad \Gamma \vdash \mathsf{Self}_{\mathsf{U}} : D\,\langle \overline{u_{2i}}\rangle \qquad \overline{\Gamma \vdash T : D_j\,\langle \overline{u_{1s_j}}^{\,s}\rangle}^{\,j} \qquad \overline{\Gamma \vdash T : D_j\,\langle \overline{u_{2s_j}}^{\,s}\rangle}^{\,j} \\ \overline{\Gamma \mid \Theta \Vdash u_{1i} \sim u_{2i}} \qquad \overline{\Gamma, T \mid \Theta, T : D\,\langle \overline{u_{1i}}\rangle \Vdash u_{1s_j} \sim u_{2s_j}} \qquad \overline{\Gamma \mid \Theta \Vdash u_{1j} \sim u_{2j}}\end{array}}{\Gamma \mid \Theta \Vdash \exists T : D\,\langle \overline{u_{1i}}, \overline{A_{D_j}\,\langle T, \overline{u_{1s_j}}\rangle \sim u_{1j}}\rangle \sim \exists T : D\,\langle \overline{u_{2i}}, \overline{A_{D_j}\,\langle T, \overline{u_{2s_j}}\rangle \sim u_{2j}}\rangle}$$

Figure 5.4: MiniRust: constraint entailment

$\boxed{\Gamma \mid \Theta \vdash e : \tau}$ (Well-typed terms)

$$\text{(sub)}\frac{\Gamma \mid \Theta \vdash e : \tau_1 \qquad \Gamma \mid \Theta \Vdash \tau_1 \sim \tau_2}{\Gamma \mid \Theta \vdash e : \tau_2} \qquad \text{(as)}\frac{\Gamma \mid \Theta \vdash e : \tau}{\Gamma \mid \Theta \vdash e \text{ as } \tau : \tau} \qquad \text{(unit)}\frac{}{\Gamma \mid \Theta \vdash () : ()}$$

$$\text{(let-un)}\frac{\Gamma \mid \Theta \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \mid \Theta \vdash e_2 : \tau_2}{\Gamma \mid \Theta \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2} \qquad \text{(seq)}\frac{\Gamma \mid \Theta \vdash e_1 : \tau_1 \quad \Gamma \mid \Theta \vdash e_2 : \tau_2}{\Gamma \mid \Theta \vdash e_1; e_2 : \tau_2}$$

$$\text{(var)}\frac{(x : \forall \overline{X_i}.\overline{\pi_j} \Rightarrow \tau) \in \Gamma \quad \overline{\Gamma \mid \Theta \vdash_{\text{WF}} u_i} \quad \overline{\Gamma \mid \Theta \Vdash [\overline{u_i/X_i}]\pi_j}^{\,j}}{\Gamma \mid \Theta \vdash x : [\overline{u_i/X_i}]\tau} \qquad \text{(app)}\frac{\Gamma \mid \Theta \vdash e : \text{fn}(\overline{\tau_i}) \to \tau \quad \overline{\Gamma \mid \Theta \vdash e_i : \tau_i}}{\Gamma \mid \Theta \vdash e(\overline{e_i}) : \tau}$$

$$\text{(ref)}\frac{\Gamma \mid \Theta \vdash lv : \tau}{\Gamma \mid \Theta \vdash \&lv : \&\tau} \qquad \text{(deref)}\frac{\Gamma \mid \Theta \vdash e : \&\tau}{\Gamma \mid \Theta \vdash *e : \tau} \qquad \text{(asgn)}\frac{\Gamma \mid \Theta \vdash lv : \tau \quad \Gamma \mid \Theta \vdash e : \tau}{\Gamma \mid \Theta \vdash lv := e : ()}$$

$$\text{(new-struct)}\frac{S\langle \overline{X_k} \rangle \{\overline{x_i : \tau_i}\} \in \Gamma \quad \overline{\Gamma \mid \Theta \vdash e_i : [\overline{u_k/X_k}]\tau_i} \quad \overline{\Gamma \mid \Theta \vdash_{\text{WF}} u_k}}{\Gamma \mid \Theta \vdash S\{\overline{x_i : e_i}\} : S\langle \overline{u_k} \rangle} \qquad \text{(proj)}\frac{\Gamma \mid \Theta \vdash e : S\langle \overline{u_j} \rangle \quad S\langle \overline{X_j} \rangle \{\overline{x_m : \tau_m}, x : \tau, \overline{x_n : \tau_n}\} \in \Gamma}{\Gamma \mid \Theta \vdash e.x : [\overline{u_j/X_j}]\tau}$$

$$\text{(obj-cast)}\frac{\Gamma \mid \Theta \vdash e : \&\tau \quad \overline{\Gamma \mid \Theta \Vdash [\tau/T]u_{1j} \sim u_{2j}}^{\,j} \qquad \Gamma \mid \Theta \Vdash \tau : D\langle \overline{u_i}, A \mapsto \star \rangle \quad \Gamma \mid \Theta \Vdash (\exists T : D\langle \overline{u_i}, \overline{u_{1j} \sim u_{2j}} \rangle) : D\langle \overline{u_i}, A \mapsto \star \rangle}{\Gamma \mid \Theta \vdash e : \&(\exists T : D\langle \overline{u_i}, \overline{u_{1j} \sim u_{2j}} \rangle)}$$

Figure 5.5: MiniRust: well-typed terms

5.2.4 Well-typed terms

The well-typed terms judgment, presented in figure 5.5, is extended with the rule (obj-cast) used to type coercions to a trait object. Note that the rule does not depend on the structure of the term e. This allows us to coerce pointers to trait objects implicitly when needed, without using the as keyword.

The purpose of the first premise $\Gamma \mid \Theta \vdash e : \&\tau$ in the rule (obj-cast) is to obtain the type of the reference that is to be coerced to a trait object. The premise $\Gamma \mid \Theta \Vdash \tau : D\langle \overline{u_i}, A \mapsto \star \rangle$ verifies that the referenced type τ, obtained in the preceding premise, implements the object's trait, along with the other type parameters specified. The premise $\overline{\Gamma \mid \Theta \Vdash [\tau/T]u_{1j} \sim u_{2j}}^{\,j}$ ensures that the type equalities in the trait object type are satisfied when instantiating the existential type variable T with the concrete type τ. Finally, to ensure that the trait operations are available on the object, the premise $\Gamma \mid \Theta \Vdash (\exists T : D\langle \overline{u_i}, \overline{u_{1j} \sim u_{2j}} \rangle) : D\langle \overline{u_i}, A \mapsto \star \rangle$

verifies that the trait object descriptor implements its trait (we elaborate on this point in section 5.3).

5.2.5 Well-typed items

The well-typed items judgment in figure 5.6 contains the same rules as the corresponding judgment in MicroRust. The main difference between the judgments is that in MicroRust we use variables X and types u as type parameters instead of s-type variables T and s-types τ, which allows us to parameterize items by types or s-types, as desired. Rule (trait) is used to type non-object-safe traits. The trait information tuple in its output typing environment contains the flag obj-unsafe, which represents that the trait is not object-safe. Also, in rule (impl), we include the side condition $[A \mapsto u_A]$ *defined* that ensures that the associated type instantiation in the impl has the correct kind. The judgment also has a new rule (obj-trt) for typing object-safe traits, presented in figure 5.7. We discuss the new rule in detail in the next section.

5.3 Object-safe traits

A trait object lets us reason about its encapsulated value in terms of the functionality provided by its trait. In fact, since the concrete type of the value is hidden from the type system's field of view, the only operations allowed on a trait object are those defined by its trait (and its supertraits). To permit these operations on trait objects, the trait object descriptor effectively implements its trait—hence the premise $\Gamma \mid \Theta \Vdash (\exists T : D\,\langle \overline{u_i}, \overline{u_{1j} \sim u_{2j}} \rangle) : D\,\langle \overline{u_i} \rangle$ in the rule (obj-cast) in figure 5.5.

However, some operations that may be defined in a trait are not feasible with a trait object. For example, recall the trait `Eq` from chapter 2:

```
trait Eq {
    fn eq(&self, &Self) -> bool;
}
```

Suppose that we have the following implementations:

```
impl Eq for i32 { ... }
impl Eq for char { ... }
```

and a hypothetical impl for `Eq`'s trait type (using Rust syntax):

```
impl Eq for Eq {
    fn eq(&self, &Eq) -> bool {
        // unpack objects and call the appropriate version of eq
    }
}
```

$\boxed{\Gamma \mid \Theta \vdash \mathit{item} : \Gamma \mid \Theta}$ (Well-typed items)

$$\text{(struct)}\ \frac{\overline{\Gamma, \overline{X_j} \mid \Theta \vdash_{\text{WF}} \tau_i}^{\,i}}{\Gamma \mid \Theta \vdash \mathsf{struct}\ S\ \langle \overline{X_j} \rangle\ \{\overline{x_i : \tau_i}\} : [S\ \langle \overline{X_j} \rangle\ \{\overline{x_i : \tau_i}\}] \mid \emptyset}$$

$$\text{(fun)}\ \frac{\begin{array}{c}\sigma = \forall \overline{X_k}.\overline{\pi_j} \Rightarrow \mathsf{fn}(\overline{\tau_i}) \to \tau \qquad \Gamma \mid \Theta \vdash_{\text{WF}} \sigma \\ \Gamma' = \Gamma, \overline{X_k}, \overline{x_i : \tau_i} \qquad \Theta' = \Theta, \overline{\pi_j} \qquad \Gamma' \mid \Theta' \vdash e : \tau\end{array}}{\Gamma \mid \Theta \vdash \mathsf{fn}\ f\ \langle \overline{X_k} \rangle\ (\overline{x_i : \tau_i}) \to \tau\ \mathsf{where}\ \overline{\pi_j}\ \{\, e \,\} : [f : \sigma] \mid \emptyset}$$

$$\text{(trait)}\ \frac{\begin{array}{c}\{\overline{\pi_j}\} \equiv \{\overline{\mathit{Self} : \beta_s}, \overline{\pi_h}\} \qquad \overline{X_p} = \mathit{Self}, \overline{X_i} \qquad \overline{\Gamma \mid \Theta \vdash_{\text{WF}} \forall \overline{X_p}.(\mathit{Self} : D\ \langle \overline{X_i} \rangle) \Rightarrow \pi_j}^{\,j} \\ \sigma = \forall \overline{X_k}.\overline{\pi_l} \Rightarrow \mathsf{fn}(\overline{\tau_m}) \to \tau_R \qquad \sigma' = \forall \overline{X_p}.(\mathit{Self} : D\ \langle \overline{X_i} \rangle) \Rightarrow \sigma \\ \Gamma \mid \Theta \vdash_{\text{WF}} \sigma' \qquad \overline{\Gamma \mid \Theta \vdash_{\text{WF}} \forall \overline{X_p}.(\mathit{Self} : D\ \langle \overline{X_i} \rangle) \Rightarrow A_D\ \langle \overline{X_p} \rangle : \beta_a}^{\,a}\end{array}}{\Gamma \mid \Theta \vdash \begin{array}{l}\mathsf{trait}\ D\ \langle \overline{X_i} \rangle\ \mathsf{for}\ \mathit{Self} \\ \mathsf{where}\ \overline{\pi_j}\ \{ \\ \quad \mathsf{type}\ A : \overline{\beta_a}; \\ \quad f : \sigma;\ \}\end{array} : \begin{array}{l}[(D\ \langle \overline{X_i} \rangle\ \mathsf{where}\ \overline{\pi_h}, \mathit{Self} : \overline{\beta_s}, A : \overline{\beta_a}, \mathsf{obj\text{-}unsafe}), f : \sigma'] \mid \\ [\overline{\forall \overline{X_p}.(\mathit{Self} : D\ \langle \overline{X_i} \rangle) \Rightarrow \mathit{Self} : \beta_s}^{\,s}, \\ \overline{\forall \overline{X_p}.(\mathit{Self} : D\ \langle \overline{X_i} \rangle) \Rightarrow A_D\ \langle \overline{X_p} \rangle : \beta_a}^{\,a}]\end{array}}$$

$$\text{(impl)}\ \frac{\begin{array}{c}(D\ \langle \overline{X_i} \rangle\ \mathsf{where}\ \overline{\pi_h}, \mathit{Self} : \overline{\beta_s}, A : \overline{\beta_a}, f, \_) \in \Gamma \qquad \overline{u_p} = u, \overline{u_i} \qquad \overline{X_p} = \mathit{Self}, \overline{X_i} \\ \overline{X_k \in FV(\overline{u_p})} \qquad \overline{\Gamma \mid \Theta \vdash_{\text{WF}} \forall \overline{X_k}.\overline{\pi_j} \Rightarrow u : D\ \langle \overline{u_i} \rangle}^{\,a} \\ \Theta^* = \Theta \setminus \{\overline{\forall \overline{X_p}.(\mathit{Self} : D\ \langle \overline{X_i} \rangle) \Rightarrow \mathit{Self} : \beta_s}^{\,s}, \overline{\forall \overline{X_p}.(\mathit{Self} : D\ \langle \overline{X_i} \rangle) \Rightarrow A_D\ \langle \overline{X_p} \rangle : \beta_a}^{\,a}\} \\ \overline{\Gamma, \overline{X_k} \mid \Theta^*, \overline{\pi_j} \Vdash [\overline{u_p/X_p}](\mathit{Self} : \beta_s)}^{\,s} \qquad \overline{\Gamma, \overline{X_k} \mid \Theta^*, \overline{\pi_j} \Vdash [\overline{u_p/X_p}]\pi_h}^{\,h} \\ [A \mapsto u_A]\ \mathit{defined} \qquad \Gamma, \overline{X_k} \mid \Theta^*, \overline{\pi_j} \vdash_{\text{WF}} u_A \qquad \overline{\Gamma, \overline{X_k} \mid \Theta^*, \overline{\pi_j} \Vdash u_A : [\overline{u_p/X_p}]\beta_a}^{\,a} \\ (f : \forall \overline{X_p}.(\mathit{Self} : D\ \langle \overline{X_i} \rangle) \Rightarrow \sigma) \in \Gamma \qquad \Gamma, \overline{X_k} \mid \Theta^*, \overline{\pi_j} \vdash \mathit{fun} : [f : \sigma'] \mid \emptyset \qquad \sigma' = [\overline{u_p/X_p}]\sigma\end{array}}{\Gamma \mid \Theta \vdash \begin{array}{l}\mathsf{impl}\ \langle \overline{X_k} \rangle\ D\ \langle \overline{u_i} \rangle\ \mathsf{for}\ u\ \mathsf{where}\ \overline{\pi_j}\ \{ \\ \quad \mathsf{type}\ A \mapsto u_A; \mathit{fun}\ \}\end{array} : \emptyset \mid [\forall \overline{X_k}.\overline{\pi_j} \Rightarrow u : D\ \langle \overline{u_i}, A \mapsto u_A \rangle]}$$

... continued in figure 5.7

$\boxed{\vdash_P \mathit{pgm} : \Gamma \mid \Theta}$ (Well-typed programs)

$$\text{(pgm)}\ \frac{\overline{\Gamma \mid \Theta \vdash \mathit{item}_i : \Gamma_i \mid \Theta_i} \qquad \Gamma = \overline{\Gamma_i} \qquad \Theta = \overline{\Theta_i}}{\vdash_P \overline{\mathit{item}_i} : \Gamma \mid \Theta}$$

Figure 5.6: MiniRust: well-typed items and programs

Then, suppose that we call `eq` with a trait object:

```
fn object_eq(obj: &Eq, other_obj: &Eq) -> bool {
    obj.eq(other)
}
```

If the concrete type behind `obj` is `i32` and the one behind `other_obj` is `char`, then there is no appropriate concrete version of `eq` that we can call. [1]

To prevent such cases, Rust has a set of requirements for *object-safe* traits: traits that can be used as bases to create trait objects. According to the documentation of the language [5], a trait is object-safe if its `Self` parameter is not marked as `Sized` and if each of its methods is object-safe. A method is object-safe only if `Self` does not appear in the types of its arguments and its return value. Also, an object-safe method may not be generic—a restriction due to *monomorphization* that is part of Rust's compilation model (we explore this in more detail in chapter 7). Moreover, since supertrait and associated type constraints are propagated, the allowed occurrences of `Self` inside them are restricted as well. However, the documentation does not currently address them and it is unclear what the rules are in that respect [5].

The restrictions on the occurrences of `Self` must be relaxed somewhat when it comes to associated types, since an associated type parameterized by `Self` may be instantiated with a concrete type that does not mention `Self`. In particular, since a trait object descriptor includes instantiations of the associated types of the object's trait and all of its supertraits, it appears reasonable to allow such associated types, parameterized by `Self`, to occur in the the body of the trait.

For instance, suppose that we have a trait `Iterator` implemented for a user-defined type `MyIterator` (using Rust syntax):

```
trait Iterator {
    type Item;
    fn next(&self) -> Self::Item;
}

impl Iterator for MyIterator {
    type Item = MyItem;
    fn next(&self) -> Self::Item { ... }
}

let my_iter: MyIterator = ...
let iter_obj = &my_iter as &Iterator<Item=MyItem>;
let my_item = my_iter.next();
```

It is clear from the type of the trait object `iter_obj` that the call to `my_item.next()` returns a value of type `MyItem`. As such, `next()` is considered object-safe and `my_item` has type `MyItem`.

[1] In fact, this is an instance of the binary method problem [11].

5.3.1 Typing object-safe traits

The rule (obj-trt), used for typing object-safe traits, is presented in figure 5.7. Note that the type system does not prevent an object-safe trait from being typed using rule (trait) instead. As long as the trait is not used to create trait objects in the program, using (trait) to type its declaration will result in a valid typing.

The main differences between (obj-trt) and (trait) are highlighted in the rule (obj-trt). One notable difference is that in (obj-trt), the *Self* variable is restricted to be a type variable $\mathsf{Self}_{\mathsf{U}}$. This restriction is necessary because we consider object-safe traits to be implemented for their trait object descriptors, which are not s-types. Also, to mirror the restriction in Rust that prevents declaring generic methods in an object-safe trait, we require the type of the trait function f to be monomorphic.

The primary results of the (obj-trt) rule for a trait D are the *trait object constraint schemes* $\overline{\forall \overline{X_i}, \overline{X_d}.\overline{\pi_{dm}}^{m} \Rightarrow \pi_d}^{d}$. Each constraint π_d corresponds to some trait D_d in D's supertrait hierarchy (including D itself) and it signals that u_{obj}—the trait descriptor of D—implements D_d. Each constraint scheme is universally quantified by the non-Self type parameters of the trait ($\overline{X_i}$), and the instantiations of the associated types of D and all of its supertraits ($\overline{X_d}$). It is also qualified by the constraints $\overline{\pi_{dm}}$, representing D_d's supertrait and associated type constraints. Recall that, using rule (obj-cast), when we create an instance of a trait object of D, we must prove that its descriptor implements its trait. To do that, we use the trait object constraint scheme that corresponds to the descriptor's trait D. In particular, when we create a trait object of D, we must ensure that the qualifying constraints in D's trait object constraint scheme are satisfied, under appropriate instantiations of the constraint scheme's type variables. Specifically, we verify that the associated type constraints are satisfied by the concrete associated type instantiations declared in the descriptor and that the descriptor also implements D's supertraits (since the trait object constraint schemes output by (obj-trt) correspond to all traits in D's supertrait hierarchy, we use them to prove that the descriptor implements D's supertraits).

Building the trait object descriptor

Recall that the descriptor of the trait D is of the form $\exists T : D\,\langle \overline{u_i}, \overline{A_{D_d}\,\langle T, \overline{u}\rangle \sim u_d}\rangle$ where $\overline{u_i}$ are the non-Self parameters of the described trait, $\overline{A_{D_d}\,\langle T, \overline{u}\rangle}$ are the associated types of D and all its supertraits, and $\overline{u_d}$ are their respective instantiations. In the rule (obj-trt) we use the relation $\Gamma \vdash D\,\langle \overline{u_i}\rangle \Uparrow \{\overline{\beta_s}\}$ to obtain all supertrait bounds in the supertrait hierarchy of the trait D (including the bound on D itself).[2] From the resulting supertrait bounds we can infer the associated types that are part of the trait descriptor u_{obj} used in the output

[2]Note that to obtain a derivation of $\Gamma \vdash D\,\langle \overline{u_i}\rangle \Uparrow \{\overline{\beta_s}\}$ for any $\overline{u_i}$ and $\overline{\beta_s}$ we must ensure that there are no cycles in D's supertrait relationship graph.

$\boxed{\Gamma \mid \Theta \vdash \mathit{item} : \Gamma \mid \Theta}$ (Well-typed items continued)

$$\frac{\begin{array}{c}\{\overline{\pi_j}\} \equiv \{(\overline{\mathsf{Self}_\mathbf{U} : \beta_s}), \overline{\pi_h}\} \quad \overline{X_p} = \mathsf{Self}_\mathbf{U}, \overline{X_i} \quad \overline{\Gamma \mid \Theta \vdash_{\mathrm{WF}} \forall \overline{X_p}.\mathsf{Self}_\mathbf{U} : D\,\langle \overline{X_i}\rangle \Rightarrow \pi_j}^{\,j} \\ \tau = \mathsf{fn}(\&\mathsf{Self}_\mathbf{U}, \overline{\tau_m}) \to \tau_R \quad \sigma' = \forall \overline{X_p}.(\mathsf{Self}_\mathbf{U} : D\,\langle \overline{X_i}\rangle) \Rightarrow \tau \\ \Gamma \mid \Theta \vdash_{\mathrm{WF}} \sigma' \quad \overline{\Gamma \mid \Theta \vdash_{\mathrm{WF}} \forall \overline{X_p}.(\mathsf{Self}_\mathbf{U} : D\,\langle \overline{X_i}\rangle) \Rightarrow A_D\,\langle \overline{X_p}\rangle : \beta_a}^{\,a} \\ \Gamma \vdash D\,\langle \overline{X_i}\rangle \Uparrow \{\overline{D_d\,\langle \overline{u_{dn}}^{\,n}, A_d \mapsto \star\rangle}^{\,d}\} \quad \overline{A_d :: X_d} \\ u_{\mathsf{obj}} = \exists T : D\langle \overline{X_i}, \overline{A_{D_d}\,\langle T, \overline{[T/\mathsf{Self}_\mathbf{U}]u_{dn}}^{\,n}\rangle \sim X_d}^{\,d}\rangle \\ \overline{\pi_d = u_{\mathsf{obj}} : D_d\,\langle \overline{[u_{\mathsf{obj}}/\mathsf{Self}_\mathbf{U}]u_{dn}}^{\,n}, A_d \mapsto X_d\rangle}^{\,d} \quad \overline{\theta_d = \forall \overline{X_i}, \overline{X_d}.\pi_d}^{\,d} \\ \overline{\Gamma \mid \Theta \vdash_{\mathrm{WF}} \theta_d} \quad \overline{\Gamma, \overline{X_i}, \overline{X_d} \mid \Theta \mid \overline{\theta_d} \vdash_{\mathsf{os}} \pi_d \rightsquigarrow \overline{\pi_{dm}}^{\,m}}^{\,d}\end{array}}{\Gamma \mid \Theta \vdash \begin{array}{l}\mathsf{trait}\ D\,\langle \overline{X_i}\rangle\ \mathsf{for}\ \mathsf{Self}_\mathbf{U} \\ \mathsf{where}\ \overline{\pi_j}\,\{ \\ \quad \mathsf{type}\ A : \overline{\beta_a} \\ \quad f : \tau\ \}\end{array} : \begin{array}{l}[(D\,\langle \overline{X_i}\rangle \ \mathsf{where}\ \overline{\pi_h},\ \mathsf{Self}_\mathbf{U} : \overline{\beta_s},\ A : \overline{\beta_a},\ f, \mathsf{obj\text{-}safe}), \\ f : \sigma'] \mid \\ [\overline{\forall \overline{X_p}.(\mathsf{Self}_\mathbf{U} : D\,\langle \overline{X_i}\rangle) \Rightarrow \mathsf{Self}_\mathbf{U} : \beta_s}^{\,s}, \\ \overline{\forall \overline{X_p}.(\mathsf{Self}_\mathbf{U} : D\,\langle \overline{X_i}\rangle) \Rightarrow A_D\,\langle \overline{X_p}\rangle : \beta_a}^{\,a}, \\ \overline{\forall \overline{X_i}, \overline{X_d}.\overline{\pi_{dm}}^{\,m} \Rightarrow \pi_d}^{\,d}\ ]\end{array}}\ (\text{obj-trt})$$

$$\frac{(D\,\langle \overline{X_i}\rangle \ \mathsf{where}\ \_, \mathsf{Self}_\mathbf{U} : \overline{D_s\,\langle \overline{u_{ns}}, A_s \mapsto \_\rangle}, A : \_, \_, \_) \in \Gamma \quad \overline{\Gamma \vdash D_s\,\langle \overline{[u_i/X_i]u_{ns}}\rangle \Uparrow \{\beta_{j_s}\}}^{\,s}}{\Gamma \vdash D\,\langle \overline{u_i}\rangle \Uparrow \{\overline{\beta_{j_s}}, D\,\langle \overline{u_i}, A \mapsto \star\rangle\}}$$

$$\frac{\Gamma \mid \Theta_1, \Theta_2 \Vdash u_1 \sim u_2 \quad \overline{\Gamma \mid \Theta_1, \Theta_2 \Vdash u_{1i} \sim u_{2i}} \quad \Gamma \mid \Theta_1 \vdash_{\mathrm{WF}} u_2 : D\,\langle \overline{u_{2i}}, A \mapsto \star\rangle}{\Gamma \mid \Theta_1 \mid \Theta_2 \vdash u_1 : D\,\langle \overline{u_{1i}}, A \mapsto \star\rangle \Rrightarrow u_2 : D\,\langle \overline{u_{2i}}, A \mapsto \star\rangle}$$

$$\frac{\begin{array}{c}\Gamma \mid \Theta_1, \Theta_2 \Vdash u_1 \sim u_2 \quad \overline{\Gamma \mid \Theta_1, \Theta_2 \Vdash u_{1i} \sim u_{2i}} \\ \Gamma \mid \Theta_1, \Theta_2 \Vdash u_3 \sim u_4 \quad \Gamma \mid \Theta_1 \vdash_{\mathrm{WF}} u_2 : D\,\langle \overline{u_{2i}}, A \mapsto u_4\rangle\end{array}}{\Gamma \mid \Theta_1 \mid \Theta_2 \vdash u_1 : D\,\langle \overline{u_{1i}}, A \mapsto u_3\rangle \Rrightarrow u_2 : D\,\langle \overline{u_{2i}}, A \mapsto u_4\rangle}$$

$\Gamma \mid \Theta \mid \overline{\theta_d} \vdash_{\mathsf{os}} u_{\mathsf{obj}} : D\,\langle \overline{u_i}, A \mapsto X\rangle \rightsquigarrow \overline{\pi_s}, \overline{\pi_a}$
where

$$\frac{\begin{array}{l}u_{\mathsf{obj}} = \exists T : D_{\mathsf{obj}}\,\langle \overline{X_h}, \overline{A_{D_d}\,\langle T, \overline{u_{nd}}\rangle \sim X_d}\rangle \\ (D\,\langle \overline{X_i}\rangle \ \mathsf{where}\ \_, \mathsf{Self}_\mathbf{U} : \overline{\beta_s}, A : \overline{\beta_a}, f, \mathsf{obj\text{-}safe}) \in \Gamma \\ \overline{\Gamma \mid \Theta \mid \overline{\theta_d} \vdash u_{\mathsf{obj}} : [u_{\mathsf{obj}}/\mathsf{Self}_\mathbf{U}][\overline{u_i/X_i}]\beta_s \Rrightarrow \pi_s}^{\,s} \\ \overline{\Gamma \mid \Theta \mid \overline{\theta_d} \vdash A_D\,\langle u_{\mathsf{obj}}, \overline{u_i}\rangle : [u_{\mathsf{obj}}/\mathsf{Self}_\mathbf{U}][\overline{u_i/X_i}]\beta_a \Rrightarrow \pi_a}^{\,a} \\ (f : \forall \mathsf{Self}_\mathbf{U}, \overline{X_i}.(\mathsf{Self}_\mathbf{U} : D\,\langle \overline{X_i}\rangle) \Rightarrow \mathsf{fn}(\&\mathsf{Self}_\mathbf{U}, \overline{\tau_m}) \to \tau_R) \in \Gamma \\ (A_D\,\langle T, \overline{u'_i}\rangle \sim X) \in \{A_{D_d}\,\langle T, \overline{u_{nd}}\rangle \sim X_d\} \\ \overline{\Gamma, T \mid \Theta, \overline{\theta_d}, \overline{T : D_d\,\langle \overline{u_{nd}}, A \sim X_d\rangle} \Vdash [T/\mathsf{Self}_\mathbf{U}][\overline{u'_i/X_i}]\tau_R \sim [u_{\mathsf{obj}}/\mathsf{Self}_\mathbf{U}][\overline{u_i/X_i}]\tau_R}^{\,m}\end{array}}{\Gamma, T \mid \Theta, \overline{\theta_d}, \overline{T : D_d\,\langle \overline{u_{nd}}, A \sim X_d\rangle} \Vdash [u_{\mathsf{obj}}/\mathsf{Self}_\mathbf{U}][\overline{u_i/X_i}]\tau_m \sim [T/\mathsf{Self}_\mathbf{U}][\overline{u'_i/X_i}]\tau_m}$$

Figure 5.7: MiniRust: well-typed object-safe traits

trait object constraint schemes. We use type variables $\overline{X_d}$ to stand in for the associated type instantiations (they're instantiated when a trait object instance is created). The side condition $A :: X$ ensures that A and X have the same kind. We define the relation $A :: X$ as follows:

$$\overline{A^{\mathsf{T}} :: T} \qquad \overline{A^{\mathsf{U}} :: U}$$

Fulfilling trait obligations

A nice advantage of having a trait object descriptor implement its trait is that we can write a generic function constrained by a trait and apply it not only to a concrete value that implements the trait but also to a trait object. For example, consider the following function (in Rust syntax):

```
fn area<T>(shape: &T) -> f64
    where T: ?Sized + HasArea { ... }
```

Since `T` does not have to be a statically sized type, we can pass to `area` a reference to some value that implements the `HasArea` trait or a `HasArea` trait object.

With this added flexibility, we must take additional care to ensure that each trait object behaves like a value that implements the object's trait—specifically, it must support the same operations. To achieve that, we must make sure that if a trait object descriptor implements a trait then it fulfills the obligations set out by the trait, since most of them are propagated. Intuitively, the concrete type encapsulated in the trait object already fulfills those obligations since it implements the object's trait. As long as the obligations don't concern `Self`, then we can assume that an implementation of the trait for its descriptor parameterized by the same non-`Self` parameters and associated types fulfills them as well. However, if `Self` is mentioned in a trait's obligations then we must take care to make sure that the trait's obligations are fulfilled with `Self` instantiated with the descriptor.

The primary obligation that an impl of a trait must fulfill is that the impl must provide an implementation of the trait function. As such, we must also make sure that we can always apply a trait function to a trait object whose descriptor implements the trait.

Moreover, the trait obligations include three kinds of constraints that must be satisfied:

- **Supertrait constraints** ($\overline{\mathsf{Self}_{\mathsf{U}} : \beta_s}$): These constraints are propagated and they allow supertrait functions to be called on trait objects. Therefore, we must ensure that the trait descriptor implements each supertrait in the descriptor's supertrait hierarchy. Thus, when an instance of a trait object is created, we verify that the object's descriptor implements its trait and that it implements its supertraits, using the trait object constraint schemes output by rule (obj-trt).

- **Associated type constraints** ($\overline{A : \beta_a}$): These constraints are also propagated. In this case, however, the bounds are on an associated type, whose concrete instantiation is determined when a trait object is created. As such, we defer checking whether those constraints are satisfied until then.
- **Other constraints** ($\overline{\pi_h}$): These are additional constraints that impls of a trait must satisfy. The bounds in these constraints are applied on types other than the trait's `Self` type or its associated type. However, they are not propagated—they do not increase the power or expressiveness of a trait object. As such, we do no extra work related to them.

Building the trait object constraint schemes

In the side conditions $\pi_d = u_{\mathsf{obj}} : D_d \langle \overline{[u_{\mathsf{obj}}/\mathsf{Self}_{\mathsf{U}}]u_{dn}}^{n}, A_d \mapsto X_d \rangle$ ^{d} in rule (obj-trt) we obtain the constraints corresponding to traits in D's supertrait hierarchy, which are used as the principal constraints of the rule's output constraint schemes. The associated type in each constraint is instantiated with variable X_d, mirroring the associated type instantiations in the object descriptor u_{obj}. In the side condition $\Gamma, \overline{X_i}, \overline{X_d} \mid \Theta \mid \overline{\theta_d} \vdash_{\mathsf{os}} \pi_d \rightsquigarrow \overline{\pi_{dm}}$ we use the trait object auxiliary relation to obtain the qualifying constraints of each trait object constraint scheme output by the rule. The side condition also includes constraint schemes $\overline{\theta_d}$, which are the aformentioned trait object constraint schemes with their qualifying constraints $\overline{\pi_{dm}}$ removed: intuitively, to obtain the constraints $\overline{\pi_{dm}}$, we temporarily assume that the trait descriptor already implements the traits in its supertrait hierarchy.

The auxiliary relation $\Gamma \mid \Theta \mid \overline{\theta_d} \vdash_{\mathsf{os}} u_{\mathsf{obj}} : D \langle \overline{u_i}, A \mapsto X \rangle \rightsquigarrow \overline{\pi_s}, \overline{\pi_a}$ is defined at the bottom of figure 5.7 (we will also refer to the relation as the relation $\vdash_{\mathsf{os}}$ for brevity). It relates the constraint $u_{\mathsf{obj}} : D \langle \overline{u_i}, A \mapsto X \rangle$, where the trait D is part of the descriptor u_{obj}'s supertrait hierarchy, with constraints $\overline{\pi_s}$ and $\overline{\pi_a}$ that are type parameter-wise equivalent to the D's supertrait and associated type constraints. The side conditions

$$\frac{}{\Gamma \mid \Theta \mid \overline{\theta_d} \vdash u_{\mathsf{obj}} : [u_{\mathsf{obj}}/\mathsf{Self}_{\mathsf{U}}][\overline{u_i/X_i}]\beta_s \Rrightarrow \pi_s}\,{}^{s}$$

$$\frac{}{\Gamma \mid \Theta \mid \overline{\theta_d} \vdash A_D \langle u_{\mathsf{obj}}, \overline{u_i} \rangle : [u_{\mathsf{obj}}/\mathsf{Self}_{\mathsf{U}}][\overline{u_i/X_i}]\beta_a \Rrightarrow \pi_a}\,{}^{a}$$

use a helper relation, defined earlier in the figure, that transforms the constraint on the left-hand side of the arrow to some equivalent constraint that is well-formed with respect to the environments Γ and Θ (if the constraint on the left-hand side of the arrow is already well-formed with respect to Γ, Θ then the relation may output the same constraint on the right-hand side of the arrow). The constraint schemes $\overline{\theta_d}$ can be used in the proof of type equality but not in the proof of well-formedness of the resulting constraints. The above side conditions thus help ensure that the trait object constraint schemes output by (obj-trt) are

well-formed. For example, consider the following trait:

$$\mathsf{trait}\ D_1\ \mathsf{for}\ \mathsf{Self}_{\mathsf{U}}\ \{\mathsf{type}\ A^{\mathsf{T}} : D_2\langle A_{D_1}^{\mathsf{T}}\langle \mathsf{Self}_{\mathsf{U}}\rangle\rangle;\ \ldots\}$$

Its corresponding descriptor would be $u_{\mathsf{obj}} = \exists T : D_1\langle A_{D_1}^{\mathsf{T}}\langle T\rangle \sim T_A\rangle$ parameterized by the type variable T_A. If we build a corresponding trait object constraint scheme using the trait's associated type constraint directly without using the helper relation $\cdot \mid \cdot \mid \cdot \vdash \cdot \Rrightarrow \cdot$ then we obtain the constraint scheme

$$\forall T_A.(A_{D_1}^{\mathsf{T}}\langle u_{\mathsf{obj}}\rangle : D_2\langle A_{D_1}^{\mathsf{T}}\langle u_{\mathsf{obj}}\rangle\rangle) \Rightarrow u_{\mathsf{obj}} : D_1\langle A^{\mathsf{T}} \mapsto T_A\rangle.$$

The constraint is not well-formed because $A_{D_1}^{\mathsf{T}}\langle u_{\mathsf{obj}}\rangle$, which appears in the associated type constraint, is only well-formed if the constraint $u_{\mathsf{obj}} : D_1\langle A^{\mathsf{T}} \mapsto \star\rangle$ is satisfied (by rule (wf-atype)). To prove that, we need to use that same constraint scheme. Therefore, in the relation $\vdash_{\mathsf{os}}$ we can use the side condition

$$\Gamma, T_A \mid \Theta \mid \forall T_A.u_{\mathsf{obj}} : D_1\langle A^{\mathsf{T}} \mapsto T_A\rangle \vdash A_{D_1}^{\mathsf{T}}\langle u_{\mathsf{obj}}\rangle : D_2\langle A_{D_1}^{\mathsf{T}}\langle u_{\mathsf{obj}}\rangle\rangle \Rrightarrow T_A : D_2\langle T_A\rangle,$$

which gives us an associated type constraint $T_A : D_2\langle T_A\rangle$ that is well-formed with respect to Γ and Θ (note that the constraint scheme $\forall T_A.u_{\mathsf{obj}} : D_1\langle A^{\mathsf{T}}\langle u_{\mathsf{obj}}\rangle \mapsto T_A\rangle$ lets us prove $A_{D_1}^{\mathsf{T}}\langle u_{\mathsf{obj}}\rangle \sim T_A$ for all instantiations of the type variable T_A). Intuitively, to obtain the qualifying constraints $\overline{\pi_s}, \overline{\pi_a}$ of the trait object constraint schemes, we temporarily assume that the trait descriptor already implements the traits in its supertrait hierarchy.

Object-safe trait functions

The relation $\vdash_{\mathsf{os}}$ also verifies that the trait function of the trait D, implemented for the descriptor u_{obj}, is object-safe—that any application of the function to the descriptor's object preserves type safety. The side conditions

$$\frac{\Gamma, T \mid \Theta, \overline{\theta_d}, \overline{T : D_d\,\langle \overline{u_n}_d, A \sim X_d\rangle} \Vdash [T/\mathsf{Self}_{\mathsf{U}}][\overline{u_i'/X_i}]\tau_R \sim [u_{\mathsf{obj}}/\mathsf{Self}_{\mathsf{U}}][\overline{u_i/X_i}]\tau_R}{\Gamma, T \mid \Theta, \overline{\theta_d}, \overline{T : D_d\,\langle \overline{u_n}_d, A \sim X_d\rangle} \Vdash [u_{\mathsf{obj}}/\mathsf{Self}_{\mathsf{U}}][\overline{u_i/X_i}]\tau_m \sim [T/\mathsf{Self}_{\mathsf{U}}][\overline{u_i'/X_i}]\tau_m}\ m$$

help ensure this, as we will see in chapter 7. They verify that the trait function's argument and return types, where $\mathsf{Self}_{\mathsf{U}}$ is instantiated with the existential type variable T, are respectively equivalent to the argument and return types where $\mathsf{Self}_{\mathsf{U}}$ is instantiated with the trait descriptor u_{obj}. The environments Γ and Θ are extended to ensure well-formedness of the type expressions.

To illustrate how these equality constraints can be satisfied when the function's type signature mentions an associated type contained in the trait object descriptor (forcibly param-

eterized by $\mathsf{Self}_\mathbf{U}$), consider the trait `Iterator` from page 47. The type scheme of its method `next` in MiniRust notation is

$$\forall \mathsf{Self}_\mathbf{U}.(\mathsf{Self}_\mathbf{U} : \texttt{Iterator}\langle \texttt{Item}^\mathbf{T} \mapsto \star\rangle) \Rightarrow \mathsf{fn}(\&\mathsf{Self}_\mathbf{U}) \to \texttt{Item}^\mathbf{T}_{\texttt{Iterator}}\langle \mathsf{Self}_\mathbf{U}\rangle$$

and its trait object descriptor is

$$u_{\mathsf{obj}} = \exists T : \texttt{Iterator}\langle \texttt{Item}^\mathbf{T}_{\texttt{Iterator}}\langle T\rangle \sim T_1\rangle,$$

parametric over type variable T_1. To satisfy the side conditions in the auxiliary relation, we need to show that

$$\Gamma' \mid \Theta' \Vdash \texttt{Item}^\mathbf{T}_{\texttt{Iterator}}\langle T\rangle \sim \texttt{Item}^\mathbf{T}_{\texttt{Iterator}}\langle u_{\mathsf{obj}}\rangle$$

where Γ' and Θ' are as follows:

$$\begin{aligned}\Gamma' =& \Gamma, T, T_1\\ \Theta' =& \Theta, T : \texttt{Iterator}\langle \texttt{Item}^\mathbf{T} \sim T_1\rangle, \forall T_1.(u_{\mathrm{OBJ}} : \texttt{Iterator}\langle \texttt{Item}^\mathbf{T} \sim T_1\rangle)\end{aligned}$$

Using (c-ext) with constraint $T : \texttt{Iterator}\langle \texttt{Item} \sim T_1\rangle$ and (eq-sep) we can prove

$$\Gamma' \mid \Theta' \Vdash \texttt{Item}^\mathbf{T}_{\texttt{Iterator}}\langle T\rangle \sim T_1.$$

Using (c-ext) again with constraint $u_{\mathsf{obj}} : \texttt{Iterator}\langle \texttt{Item}^\mathbf{T} \sim T_1\rangle$, and (eq-sep) with (eq-sym), we can prove

$$\Gamma' \mid \Theta' \Vdash T_1 \sim \texttt{Item}^\mathbf{T}_{\texttt{Iterator}}\langle u_{\mathsf{obj}}\rangle$$

Then, we can compose the two equalities using (eq-trans) to obtain the desired result. In fact, as long as there are no overlapping impl declarations, the type equality constraints in the auxiliary relation are satisfiable only when $\mathsf{Self}_\mathbf{U}$ is a parameter to an associated type or when $\mathsf{Self}_\mathbf{U}$ is in neither of the related types.

5.3.2 Discussion

In our formalization of object safety in MiniRust, we refrain from setting syntactic restrictions on the appearances of the type variable `Self` in an object-safe declaration. Instead, we use the existing constraint mechanism of MiniRust's type system to constrain object-safe traits in an attempt to allow as many object-safe traits as possible, without compromising type safety. As a result of this approach, the concept of object safety is two-tiered in MiniRust: the ability to create an instance of a trait object hinges on the trait declaration being object-safe (as decided by the rule (obj-trt)), and on the qualifying constraints in the corresponding trait object constraint scheme being satisfied at object creation. Therefore, an object-safe

trait does not guarantee that all pointers to values that implement the trait can be safely cast to a trait object.

For example, consider the following traits (the trait functions are omitted):

$$\mathsf{trait}\ D_1\ \mathsf{for}\ \mathsf{Self}_{\mathbf{U}}\ \{\mathsf{type}\ A^{\mathbf{T}} : D_2\langle \mathsf{Self}_{\mathbf{U}}\rangle; \ldots\}$$
$$\mathsf{trait}\ D_2\langle U\rangle\ \mathsf{for}\ \mathsf{Self}_{\mathbf{U}}\{\ldots\}$$

The object trait constraint scheme generated by the (obj-trt) rule for the trait D_1 would be:

$$\forall T_A.(T_A : D_2\langle u_{\mathsf{obj}}\rangle) \Rightarrow u_{\mathsf{obj}} : D_1\langle A^{\mathbf{T}} \mapsto T_A\rangle$$

where $u_{\mathsf{obj}} = \exists T : D_1\langle A_{D_1}^{\mathbf{T}}\langle T\rangle \sim T_A\rangle$.

Now suppose that D_1 has the following impl:

$$\mathsf{impl}\ D_1\ \mathsf{for}\ \mathtt{i32}\ \{\mathsf{type}\ A^{\mathbf{T}} \mapsto \mathtt{bool}\}$$

and that the constraint $\mathtt{bool} : D_2\langle \mathtt{i32}\rangle$ is satisfied. If we want to cast an `i32` pointer to a trait object of D_1 (with descriptor $u_{D1} = \exists T : D_1\langle A_{D_1}^{\mathbf{T}}\langle T\rangle \sim \mathtt{bool}\rangle$) then we must ensure that the constraint $\mathtt{bool} : D_2\langle u_{D1}\rangle$ is satisfied. To be able to create the trait object, the programmer would also need to provide an impl corresponding to that constraint (unless the impl is auto-generated by the system, which we believe to be difficult, if at all possible).

We conjecture that if in an object-safe trait we only let `Self` appear in associated types that normalize to concrete types that do not mention `Self`, then we should always be able to create an object of the trait (of course, instances of `Self` in the receiver position of the trait method signature would still be allowed and required). We can make the restriction more strict by only allowing `Self` to appear as a parameter to an associated type that is in the trait's descriptor. Since such a restriction is defined at the syntactic level, it might be easier for programmers to understand.

Chapter 6

RustIn: an internal language for MiniRust

Our formal study of Rust's traits has so far only concerned its static semantics. Specifically, we have examined what constitutes a well-typed MiniRust program. To show how such a program behaves at run-time, we must develop its operational semantics. Designing operational semantics for MiniRust directly is challenging in the presence of trait-based overloading. One way to solve this challenge is to translate MiniRust programs to another language where trait constraints have been resolved. In this chapter, we present such a language—RustIn—for which we present a type system as well as operational semantics and we prove that RustIn is type-safe: that well-typed programs don't exhibit undefined behavior. In chapter 7, we show how well-typed MiniRust programs can be translated to RustIn and we prove that the resulting RustIn programs are also well-typed.

6.1 RustIn at a glance

Our approach to designing the internal language for MiniRust is similar to Haskell's internal language, System FC [38]. System FC, based on System F, is designed to be explicitly-typed in the way that terms encode their typing derivations. System FC's novel feature is support for non-syntactic type equality, in the form of *coercions*. In Haskell, coercions are used as the result of translation of type equality constraints generated from associated type synonyms.

Like System FC, RustIn is also explicitly typed. Our intention is to leave the bulk of the typechecking to MiniRust. This leaves us with straightforward, syntax-directed typechecking in RustIn and it lets us develop clear operational semantics that do not depend on the type of a term.

As MiniRust also includes the notion of equivalence of type expressions, we introduce coercions in RustIn. A coercion has a type that consists of the two equivalent type expressions that the coercion relates. For example, a coercion γ of type $A\,\langle \texttt{i32} \rangle \sim \texttt{bool}$ relates the types

$A\,\langle \texttt{i32} \rangle$ and `bool`. We can also abstract over coercions as we do with terms in function declarations and with types in type abstractions. For example, the term $\Lambda(c : A\,\langle \texttt{i32} \rangle \sim \texttt{bool}).e$ abstracts over some *coercion variable* c that has type $A\,\langle \texttt{i32} \rangle \sim \texttt{bool}$. Note, however, that unlike most terms, coercions have no computation associated with them so they can, in principle, be erased at runtime.

Coercions are used to convert the type of a term. For instance, if term e has type $A\,\langle \texttt{i32} \rangle$ and coercion γ has type $A\,\langle \texttt{i32} \rangle \sim \texttt{bool}$ then we can apply the coercion γ to term e to obtain a term $e \blacktriangleright \gamma$ of type `bool`.

Just as terms encode their typing derivations, coercions encode proofs of type expression equivalence. Therefore, a coercion can be used to unambiguously reconstruct the proof of equivalence of two type expressions.

6.2 Syntax

Figure 6.1 presents the syntax of RustIn.

6.2.1 Items

As in MiniRust, programs are collections of items. There are four kinds of items in RustIn.

Type declarations $\texttt{type}\ A\,\langle \overline{X} \rangle$ are used to declare new abstract types or type aliases. The type constructor A is parameterized by zero or more types $\overline{X}$.

Axiom declarations are used to declare top-level type equivalence axioms, which can be used in coercions. Specifically, an axiom $\texttt{axiom}\ c : \forall \overline{X}.A\,\langle \overline{u} \rangle \mapsto u$ declares a new *coercion constant* c and specifies its type. The axiom acts similarly to an axiom in proof theory—it is used to build proofs of type equivalence. Note that we restrict the left-hand side of the arrow $\mapsto$ in the axiom declaration to be an application of an abstract type declared in a type declaration. The universal quantifier $\forall \overline{X}$ is bound over both sides of the coercion type.

Struct declarations define new structs as in MiniRust. RustIn structs are also used to represent trait dictionaries, whose fields may include polymorphic functions. Therefore, unlike their MiniRust counterparts, RustIn structs may include polymorphic fields.

Term declarations are a generalization of top-level function declarations: they are used to declare (possibly polymorphic) terms at the top level of the program. Similarly to MiniRust's function declarations, the semantics of top-level terms are not affected by the order in which they are declared, as all top-level terms are visible at all points of the program. Thus, we restrict the top-level terms to be *values*, which do not reduce at runtime. The operational semantics, which we present later in the chapter, require one of the top-level variables to be called `main` and have type $\texttt{fn}() \to ()$.

$e \in \text{TERM}$, $x, f, \mathsf{main} \in \text{VAR}$, $lv \in \text{LVAL}$, $l \in \text{LOC}$, $cl \in \text{CLOC}$, $v \in \text{VAL}$, $w \in \text{STOREVAL}$, $pv \in \text{PVAL}$, $pw \in \text{PSTOREVAL}$, $\tau \in \text{STYPE}$, $u \in \text{TYPE}$, $X \in \text{XVAR}$, $T \in \text{TVAR}$, $U \in \text{UVAR}$, $A \in \text{ACONS}$, $A^{\mathsf{T}} \in \text{TACONS}$, $A^{\mathsf{U}} \in \text{UACONS}$, $\sigma \in \text{TSCHEME}$, $\eta \in \text{USCHEME}$, $\gamma \in \text{COER}$, $c \in \text{CVAR}$, $n \in \mathbb{N}^+$, $\omega \in \text{CTYPE}$, $\vartheta \in \text{POLYCTYPE}$, $\mu \in \text{LOC} \rightharpoonup \text{STOREVAL}$

pgm	::=	$\overline{item}$	(programs)
$item$	::=	$\mathsf{type}\ A\ \langle \overline{X} \rangle$	(items)
	|	$\mathsf{axiom}\ c : \forall \overline{X}.A\ \langle \overline{u} \rangle \mapsto u$	
	|	$\mathsf{struct}\ S\ \langle \overline{X} \rangle\ \{\overline{x : \sigma}\}$	
	|	$x : \sigma = v;$	
e	::=	$\mathsf{let}\ x : \tau = e\ \mathsf{in}\ e$	(terms)
	|	$lv := e$	
	|	$e(\overline{e})$	
	|	$\mathsf{fn}\ (\overline{x : \tau})\{e\}$	
	|	$\&lv$	
	|	$*e$	
	|	x	
	|	l	
	|	$()$	
	|	$e; e$	
	|	$\mathsf{pack}\ (\tau, \overline{e}, \overline{\gamma})\ \mathsf{as}\ \&(\exists T, \overline{\tau}, \overline{\omega})$	
	|	$\mathsf{let}\ (T, \overline{x}, \overline{c}) = \mathsf{unpack}\ e\ \mathsf{in}\ e$	
	|	$S\ \langle \overline{u} \rangle\ \{\overline{x : e}\}$	
	|	$e.x$	
	|	$e[\overline{u}]$	
	|	$\Lambda \overline{X}.e$	
	|	$e[\![\overline{\gamma}]\!]$	
	|	$\Lambda \overline{c : \omega}.e$	
	|	$e \blacktriangleright \gamma$	
lv	::=	x	(lvalues)
	|	$*e$	
	|	$lv.x$	
	|	cl	
cl	::=	l	(c-locations)
	|	$l \blacktriangleright \gamma$	
v	::=	$pv \blacktriangleright \gamma$	(values)
	|	pv	
pv	::=	$\&l$	(plain values)
	|	$()$	
	|	$\mathsf{fn}\ (\overline{x : \tau})\{e\}$	
	|	$S\ \langle \overline{u} \rangle\ \{\overline{x : v}\}$	
	|	$\mathsf{pack}\ (\tau, \overline{v}, \overline{\gamma})\ \mathsf{as}\ \tau$	
	|	$\Lambda \overline{X}.e$	
	|	$\Lambda \overline{c : \omega}.e$	
w	::=	$pw \blacktriangleright \gamma$	(store values)
	|	pw	
pw	::=	$\&l$	(plain store values)
	|	$()$	
	|	$\mathsf{fn}\ (\overline{x : \tau})\{e\}$	
	|	$S\ \langle \overline{u} \rangle\ \{\overline{x : l}\}$	
	|	$\mathsf{pack}\ (\tau, \overline{v}, \overline{\gamma})\ \mathsf{as}\ \tau$	
	|	$\Lambda \overline{X}.e$	
	|	$\Lambda \overline{c : \omega}.e$	
τ	::=	$()$	(s-types)
	|	T	
	|	$A^{\mathsf{T}}\ \langle \overline{u} \rangle$	
	|	$\mathsf{fn}(\overline{\tau}) \to \tau$	
	|	$\&u$	
	|	$S\ \langle \overline{u} \rangle$	
u	::=	τ	(types)
	|	U	
	|	$A^{\mathsf{U}}\ \langle \overline{u} \rangle$	
	|	$(\exists T, \overline{\tau}, \overline{\omega})$	
X	::=	T	(type variables)
	|	U	
A	::=	A^{T}	(abstract type constructors)
	|	A^{U}	
σ	::=	$\forall \overline{X}.\sigma$	(s-type schemes)
	|	$\forall \overline{\omega}.\sigma$	
	|	τ	
η	::=	$\forall \overline{X}.\eta$	(type schemes)
	|	$\forall \overline{\omega}.\eta$	
	|	u	
γ	::=	c	(coercions)
	|	$\mathsf{sym}\ \gamma$	
	|	$\gamma \circ \gamma$	
	|	$S\ \langle \overline{\gamma} \rangle$	
	|	$\&\gamma$	
	|	$\mathsf{fn}(\overline{\gamma}) \to \gamma$	
	|	$()$	
	|	X	
	|	$A\ \langle \overline{\gamma} \rangle$	
	|	$(\exists T, \overline{\gamma}, \overline{\gamma \sim \gamma})$	
	|	$*\gamma$	
	|	$\mathsf{spar}\ (n, \gamma)$	
	|	$\mathsf{farg}\ (n, \gamma)$	
	|	$\mathsf{fret}\ \gamma$	
	|	$\mathsf{objf}\ (\tau, n, \gamma)$	
	|	$\mathsf{objc\text{-}l}\ (\tau, n, \gamma)$	
	|	$\mathsf{objc\text{-}r}\ (\tau, \gamma)$	
	|	$\mathsf{coer}\ \gamma$	
	|	$\mathsf{coer\text{-}l}\ (n, \gamma)$	
	|	$\mathsf{coer\text{-}r}\ (n, \gamma)$	
	|	$\gamma[\overline{u}]$	
	|	$\forall \overline{X}.\gamma$	
	|	$\forall \overline{\gamma \sim \gamma}.\gamma$	
ω	::=	$u \sim u$	(simple coercion types)
ϑ	::=	$\eta \sim \eta$	(polymorphic coercion types)
Ω	::=	$\Omega, \mathsf{type}\ A\ \langle \overline{X} \rangle$	
	|	$\Omega, S\ \langle \overline{u} \rangle\ \{\overline{x : \sigma}\}$	
	|	$\Omega, c : \vartheta$	
	|	$\emptyset$	
Γ	::=	Γ, X	
	|	$\Gamma, x : \sigma$	
	|	$\emptyset$	
Σ	::=	$\Sigma, l : \sigma$	
	|	$\emptyset$	

Figure 6.1: RustIn: syntax

6.2.2 Terms

The syntax of terms in RustIn overlaps significantly with its counterpart in MiniRust. A notable difference between the two is that RustIn terms are explicitly typed. For example, a let-expression $\mathsf{let}\ x : \tau = e\ \mathsf{in}\ e$ explicitly includes the type τ of the declared variable x. The instantiations of type parameters of a struct instance $S\,\langle\overline{u}\rangle\,\{\overline{x : e}\}$ are also explicitly included in the term.

New terms, not included in MiniRust, include:

- anonymous functions $\mathsf{fn}\ (\overline{x : \tau})\{e\}$, which are similar to Rust's closures: they capture the variables that are in their surrounding scope,
- location l, which denote store locations that we use in our operational semantics (locations are not, however, part of RustIn source programs—they are only introduced at runtime),
- packing terms $\mathsf{pack}\ (\tau, \overline{e}, \overline{\gamma})\ \mathsf{as}\ \tau$, which 'pack' the contents of the tuple $(\tau, \overline{e}, \overline{\gamma})$ into an existential object, where the type τ is existentially quantified,
- unpacking terms $\mathsf{let}\ (T, \overline{x}, \overline{c}) = \mathsf{unpack}\ e_1\ \mathsf{in}\ e_2$, which bind the unpacked contents of the existential object e_1 to the variables of the tuple $(T, \overline{x}, \overline{c})$ in e_2,
- type abstractions $\Lambda\overline{X}.e$, which abstract types $\overline{X}$ out of term e,
- type applications $e[\overline{u}]$, which instantiate the type variables of a type abstraction with types $\overline{u}$ in term e,
- coercion abstractions $\Lambda\overline{c : \omega}.e$, which abstract coercions $\overline{c}$ of types $\overline{\omega}$ out of term e, and
- coercion applications $e[\![\overline{\gamma}]\!]$, which instantiate the coercion variables of a coercion abstraction with coercions $\overline{\gamma}$ in term e.
- term coercions $e \blacktriangleright \gamma$, which convert the type of term e using the coercion γ.

Lvalues include c-locations, which in turn include store locations l, or coerced store locations $l \blacktriangleright \gamma$: locations that we want to use at a different type, specified by the coercion γ. Just like locations, c-locations are only introduced at runtime—they do not appear in source programs.

We also introduce values v, which range over terms that can be the final result of evaluation. They include plain values pv and coerced plain values $pv \blacktriangleright \gamma$.

Plain values include location references ($\&l$), the unit expression ($()$), function declarations ($\mathsf{fn}\ (\overline{x : \tau})\{e\}$), struct instances with value fields ($S\,\langle\overline{u}\rangle\,\{\overline{x : v}\}$), value-saturated packing terms ($\mathsf{pack}\ (\tau, \overline{v}, \overline{\gamma})\ \mathsf{as}\ \tau$), type abstractions $\Lambda\overline{X}.e$, and coercion abstractions $\Lambda\overline{c : \omega}.e$.

Store values w and plain store values pw range over values that can be stored directly in the store. They are mostly the same as values and plain values with the exception of struct instances, where fields are store location labels instead values. Storing each struct instance field in its own location allows us to directly reference it.

6.2.3 Types

S-types τ and types u are for the most part the same as their MiniRust counterparts. The only exception is that the MiniRust trait descriptor $\exists T : D\,\langle \overline{u}, \overline{\pi^{\sim}} \rangle$ is replaced in RustIn with the more general *existential object descriptor* $(\exists T, \overline{\tau}, \overline{\omega})$. We refer to a reference to a descriptor $\&(\exists T, \overline{\tau}, \overline{\omega})$ as an *existential object* (like a trait object in MiniRust, a RustIn existential object is not exactly a pointer—the reference operator is thus overloaded). In the existential object descriptor, $\exists T$ introduces the existentially quantified type variable T bound to the descriptor, $\overline{\tau}$ are the types of terms encapsulated in the existential object and $\overline{\omega}$ are the types of coercions encapsulated in the object.

As in MiniRust, X is a metavariable ranging over s-type variables T and type variables U. A is a metavariable ranging over s-type constructors A^{T} and type constructors A^{U} that are used to declare abstract types used in coercions.

Type schemes in RustIn are not qualified, unlike their MiniRust counterparts. A RustIn *s-type scheme* can be quantified over types $\forall \overline{X_i}.\sigma$ (classifying type abstractions) and over coercions $\forall \overline{\omega}.\sigma$ (classifying coercion abstractions). We also include *type scheme* η, which is quantified over types that are not necessarily s-types. Note that the set of type schemes USCHEME is a superset of the set of type schemes TSCHEME. Type substitutions are defined in the same way as in MiniRust—a substitution $[u/T]$ is defined only if $u \in \text{STYPE}$.

6.2.4 Coercions

γ ranges over coercions. We can separate them into the following categories:

- c: a coercion variable introduced either in a top-level axiom (as a constant) or in a coercion abstraction,
- $\mathsf{sym}\ \gamma$ and $\gamma \circ \gamma$, which reflect respectively the symmetry and transitivity properties of the equivalence relation,
- constructing coercions $(S\,\langle \overline{\gamma} \rangle, \&\gamma, \mathsf{fn}\ (\overline{\gamma}) \to \gamma, (), X, A\,\langle \overline{\gamma} \rangle, (\exists T, \overline{\gamma}, \overline{\gamma \sim \gamma}), \forall \overline{X}.\gamma, \forall \overline{\gamma \sim \gamma}.\gamma)$, which are constructed following a type expression structure (they can also represent reflexivity of the equivalence relation), and
- deconstructing coercions, where the above coercions are decomposed $(*\gamma$, $\mathsf{spar}\ (n, \gamma)$, $\mathsf{farg}\ (n, \gamma)$, $\mathsf{fret}\ \gamma$, $\mathsf{objf}\ (\tau, n, \gamma)$, $\mathsf{objc\text{-}l}\ (\tau, n, \gamma)$, $\mathsf{objc\text{-}r}\ (\tau, n, \gamma)$, $\mathsf{coer}\ \gamma$, $\mathsf{coer\text{-}l}\ (n, \gamma)$,

coer-r (n, γ), $\gamma[\overline{u}]$).

ϑ is a polymorphic coercion type, which relates two equivalent type schemes η. We also include a simple coercion type, which relates two equivalent types u.

6.2.5 Environments

To type terms, we carry around three distinct environments. The *coercion and struct environment* Ω contains coercion variable typings and defined structs. The *typing environment* Γ contains term variable typings and free type variables in scope. The *store typing environment* Σ is a partial function mapping store locations to their types.

We distinguish between environments Ω and Γ as the evaluation rules, which we discuss in section 6.4, rely on the struct declarations and coercion typings that are contained in the top-level environment Ω. Variable typings and the set of free type variables in scope, contained in the typing environment Γ, are not needed at runtime.

6.3 Type system

6.3.1 Well-typed coercions

Figures 6.2 and 6.3 present the well-typed coercions judgment. The judgment, $\Omega \vdash \gamma : \vartheta$, reads: "coercion γ has type ϑ under environment Ω". An important insight into the judgment is that every member of the set TYPE is also a member of the set COER. Specifically, a type u is also an *identity coercion* u of type $u \sim u$.

Rule (co-var) simply looks up the type of a coercion variable (or constant) c in the environment Ω. Rules (co-sym) and (co-trans) apply the symmetry and transitivity properties of the equivalence relation respectively. (co-unit) and (co-tvar) are identity coercions of the unit type and type variables respectively. Rules (co-tabs), (co-cabs), (co-atype), (co-struct), (co-ref), (co-obj) and (co-fun) are used to type constructing coercions by induction on each of the respective coercion's components. In each of those rules, if the coercions in the premises are types, then the coercion in the conclusion of the rule is an identity coercion (relating a type to itself). Rule (co-tapp) is used for typing type applications $\gamma[\overline{u_i}]$ where the type of γ relates two type schemes η_1 and η_2 quantified by some types $\overline{X_i}$.

The typing rules in figure 6.3 are used to type deconstructing coercions. Such coercions are parametric over some coercion γ that has a type that follows some specific structure. For instance, in rule (co-deref) used to type coercions of form $*\gamma$, the coercion γ relates reference types. In rule (co-spar), the coercion spar (n, γ) relates the n^{th} type arguments of the structs related by the coercion γ. The coercions typed in rule (co-farg) and (co-fret) relate respectively the n^{th} arguments and return types of the functions related by γ. Rules (co-objf), (co-objc-l)

$\boxed{\Omega \vdash \gamma : \vartheta}$ (Well-typed coercions)

$$\text{(co-var)}\frac{(c : \vartheta) \in \Omega}{\Omega \vdash c : \vartheta} \qquad \text{(co-sym)}\frac{\Omega \vdash \gamma : \eta_1 \sim \eta_2}{\Omega \vdash \mathsf{sym}\ \gamma : \eta_2 \sim \eta_1}$$

$$\text{(co-trans)}\frac{\Omega \vdash \gamma_1 : \eta_1 \sim \eta_3 \qquad \Omega \vdash \gamma_2 : \eta_3 \sim \eta_2}{\Omega \vdash \gamma_1 \circ \gamma_2 : \eta_1 \sim \eta_2} \qquad \text{(co-tabs)}\frac{\Omega \vdash \gamma : \eta_1 \sim \eta_2}{\Omega \vdash \forall \overline{X_i}.\gamma : \forall \overline{X_i}.\eta_1 \sim \forall \overline{X_i}.\eta_2}$$

$$\text{(co-tapp)}\frac{\Omega \vdash \gamma : \forall \overline{X_i}.\eta_1 \sim \forall \overline{X_i}.\eta_2}{\Omega \vdash \gamma[\overline{u_i}] : [\overline{u_i/X_i}]\eta_1 \sim [\overline{u_i/X_i}]\eta_2}$$

$$\text{(co-cabs)}\frac{\Omega \vdash \gamma : \eta_1 \sim \eta_2 \qquad \overline{\Omega \vdash \gamma_{1i} : u_{1i} \sim u_{3i}} \qquad \overline{\Omega \vdash \gamma_{2i} : u_{2i} \sim u_{4i}}}{\Omega \vdash \forall \overline{\gamma_{1i} \sim \gamma_{2i}}.\gamma : (\forall \overline{u_{1i} \sim u_{2i}}.\eta_1) \sim (\forall \overline{u_{3i} \sim u_{4i}}.\eta_2)}$$

$$\text{(co-unit)}\frac{}{\Omega \vdash () : () \sim ()} \qquad \text{(co-tvar)}\frac{}{\Omega \vdash X : X \sim X}$$

$$\text{(co-atype)}\frac{\overline{\Omega \vdash \gamma_i : u_{1i} \sim u_{2i}}}{\Omega \vdash A\ \langle \overline{\gamma_i} \rangle : A\ \langle \overline{u_{1i}} \rangle \sim A\ \langle \overline{u_{2i}} \rangle} \qquad \text{(co-struct)}\frac{\overline{\Omega \vdash \gamma_i : u_{1i} \sim u_{2i}}}{\Omega \vdash S\ \langle \overline{\gamma_i} \rangle : S\ \langle \overline{u_{1i}} \rangle \sim S\ \langle \overline{u_{2i}} \rangle}$$

$$\text{(co-ref)}\frac{\Omega \vdash \gamma : u_1 \sim u_2}{\Omega \vdash \&\gamma : \&u_1 \sim \&u_2}$$

$$\text{(co-obj)}\frac{\overline{\Omega \vdash \gamma_i : \tau_{1i} \sim \tau_{2i}}^{\,i} \qquad \overline{\Omega \vdash \gamma_{1j} : u_{1j} \sim u_{3j}}^{\,j} \qquad \overline{\Omega \vdash \gamma_{2j} : u_{2j} \sim u_{4j}}^{\,j}}{\Omega \vdash (\exists T, \overline{\gamma_i}, \overline{\gamma_{1j} \sim \gamma_{2j}}) : (\exists T, \overline{\tau_{1i}}, \overline{u_{1j} \sim u_{3j}}) \sim (\exists T, \overline{\tau_{2i}}, \overline{u_{2j} \sim u_{4j}})}$$

$$\text{(co-fun)}\frac{\overline{\Omega \vdash \gamma_i : \tau_{1i} \sim \tau_{2i}} \qquad \Omega \vdash \gamma : \tau_1 \sim \tau_2}{\Omega \vdash \mathsf{fn}(\overline{\gamma_i}) \to \gamma : \mathsf{fn}(\overline{\tau_{1i}}) \to \tau_1 \sim \mathsf{fn}(\overline{\tau_{2i}}) \to \tau_2}$$

Figure 6.2: RustIn: well-typed coercions

and (co-objc-r) are used to type coercions that relate components of some pair of existential object descriptors. The coercions in the conclusion of these rules also include a type τ, which instantiates the type variable T over which the descriptor is existentially quantified. Finally, rules (co-coer), (co-coer-l) and (co-coer-r) are used to type coercions that relate components of a type scheme quantified over a coercion.

6.3.2 Well-typed terms

The well-typed terms judgment is presented in figure 6.4. The judgment, $\Omega \mid \Gamma \mid \Sigma \vdash e : \sigma$, reads: "under environments Ω, Γ and Σ, term e has type scheme σ". As opposed to the typing judgments in MiniRust, the types of terms in RustIn may be polymorphic, which brings more

$\boxed{\Omega \vdash \gamma : \vartheta}$ (Well-typed coercions)

$$\text{(co-deref)}\ \frac{\Omega \vdash \gamma : \&u_1 \sim \&u_2}{\Omega \vdash *\gamma : u_1 \sim u_2} \qquad \text{(co-spar)}\ \frac{\Omega \vdash \gamma : S\,\langle \overline{u_{1i}} \rangle \sim S\,\langle \overline{u_{2i}} \rangle \quad u_1 = n^{th}(\overline{u_{1i}}) \quad u_2 = n^{th}(\overline{u_{2i}})}{\Omega \vdash \mathsf{spar}\ (n,\gamma) : u_1 \sim u_2}$$

$$\text{(co-farg)}\ \frac{\Omega \vdash \gamma : \mathsf{fn}(\overline{\tau_{1i}}) \to \tau_{R1} \sim \mathsf{fn}(\overline{\tau_{2i}}) \to \tau_{R2} \quad \tau_1 = n^{th}(\overline{\tau_{1i}}) \quad \tau_2 = n^{th}(\overline{\tau_{2i}})}{\Omega \vdash \mathsf{farg}\ (n,\gamma) : \tau_1 \sim \tau_2} \qquad \text{(co-fret)}\ \frac{\Omega \vdash \gamma : \mathsf{fn}(\overline{\tau_{1i}}) \to \tau_1 \sim \mathsf{fn}(\overline{\tau_{2i}}) \to \tau_2}{\Omega \vdash \mathsf{fret}\ (\gamma) : \tau_1 \sim \tau_2}$$

$$\text{(co-objf)}\ \frac{\Omega \vdash \gamma : (\exists T, \overline{\tau_{1i}}, \overline{\omega_{1i}}) \sim (\exists T, \overline{\tau_{2i}}, \overline{\omega_{2i}}) \quad \tau_1 = [\tau/T]n^{th}(\overline{\tau_{1i}}) \quad \tau_2 = [\tau/T]n^{th}(\overline{\tau_{2i}})}{\Omega \vdash \mathsf{objf}\ (\tau,n,\gamma) : \tau_1 \sim \tau_2}$$

$$\text{(co-objc-l)}\ \frac{\Omega \vdash \gamma : (\exists T, \overline{\tau_{1i}}, \overline{u_{1j} \sim u_{3j}}) \sim (\exists T, \overline{\tau_{2i}}, \overline{u_{2j} \sim u_{4j}}) \quad u_1 = [\tau/T]n^{th}(\overline{u_{1j}}) \quad u_2 = [\tau/T]n^{th}(\overline{u_{2j}})}{\Omega \vdash \mathsf{objc\text{-}l}\ (\tau,n,\gamma) : u_1 \sim u_2}$$

$$\text{(co-objc-r)}\ \frac{\Omega \vdash \gamma : (\exists T, \overline{\tau_{1i}}, \overline{u_{1j} \sim u_{3j}}) \sim (\exists T, \overline{\tau_{2i}}, \overline{u_{2j} \sim u_{4j}}) \quad u_1 = [\tau/T]n^{th}(\overline{u_{3j}}) \quad u_2 = [\tau/T]n^{th}(\overline{u_{4j}})}{\Omega \vdash \mathsf{objc\text{-}r}\ (\tau,n,\gamma) : u_1 \sim u_2}$$

$$\text{(co-coer)}\ \frac{\Omega \vdash \gamma : \forall \overline{\omega_{1i}}.\eta_1 \sim \forall \overline{\omega_{2i}}.\eta_2}{\Omega \vdash \mathsf{coer}\ (\gamma) : \eta_1 \sim \eta_2}$$

$$\text{(co-coer-l)}\ \frac{\Omega \vdash \gamma : (\forall \overline{u_{1i} \sim u_{2i}}.\eta_1) \sim (\forall \overline{u_{3i} \sim u_{4i}}.\eta_2) \quad u_1 = n^{th}(\overline{u_{1i}}) \quad u_2 = n^{th}(\overline{u_{3i}})}{\Omega \vdash \mathsf{coer\text{-}l}\ (n,\gamma) : u_1 \sim u_2}$$

$$\text{(co-coer-r)}\ \frac{\Omega \vdash \gamma : (\forall \overline{u_{1i} \sim u_{2i}}.\eta_1) \sim (\forall \overline{u_{3i} \sim u_{4i}}.\eta_2) \quad u_1 = n^{th}(\overline{u_{2i}}) \quad u_2 = n^{th}(\overline{u_{4i}})}{\Omega \vdash \mathsf{coer\text{-}r}\ (n,\gamma) : u_1 \sim u_2}$$

Figure 6.3: RustIn: well-typed coercions continued

flexibility as to where type and coercion abstractions are permitted. Note that we work with terms and types up to alpha-conversion (up to bound variable renaming). In particular, this means that $[u/X]\forall X.\sigma$ is equivalent to $[u/X]\forall X_1.[X_1/X]\sigma$ (which in turn is equivalent to $\forall X_1.[X_1/X]\sigma$).

Many of the rules are similar to their counterparts in MiniRust. Some notable differences include (t-var), where the bound variables in the type scheme of x are not instantiated right away. Type and coercion instantiations are explicit—the instantiations are part of the term—and their corresponding typing rules are (t-tapp) and (t-capp) respectively. The coercion instantiation rule is similar to the function application rule (t-fapp), which demonstrates

$\boxed{\Omega \mid \Gamma \mid \Sigma \vdash e : \sigma}$ (Well-typed terms)

$$\text{(t-let)}\frac{\Omega \mid \Gamma \mid \Sigma \vdash e_1 : \tau_1 \quad \Omega \mid \Gamma, x : \tau_1 \mid \Sigma \vdash e_2 : \tau_2}{\Omega \mid \Gamma \mid \Sigma \vdash \mathsf{let}\ x : \tau_1 = e_1\ \mathsf{in}\ e_2 : \tau_2} \qquad \text{(t-upd)}\frac{\Omega \mid \Gamma \mid \Sigma \vdash lv : \tau \quad \Omega \mid \Gamma \mid \Sigma \vdash e : \tau}{\Omega \mid \Gamma \mid \Sigma \vdash lv := e : ()}$$

$$\text{(t-fapp)}\frac{\Omega \mid \Gamma \mid \Sigma \vdash e : \mathsf{fn}(\overline{\tau_i}) \to \tau \quad \overline{\Omega \mid \Gamma \mid \Sigma \vdash e_i : \tau_i}}{\Omega \mid \Gamma \mid \Sigma \vdash e(\overline{e_i}) : \tau} \qquad \text{(t-fabs)}\frac{\Omega \mid \Gamma, \overline{x_i : \tau_i} \mid \Sigma \vdash e : \tau}{\Omega \mid \Gamma \mid \Sigma \vdash \mathsf{fn}\ (\overline{x_i : \tau_i})\{e\} : \mathsf{fn}(\overline{\tau_i}) \to \tau}$$

$$\text{(t-ref)}\frac{\Omega \mid \Gamma \mid \Sigma \vdash lv : \tau}{\Omega \mid \Gamma \mid \Sigma \vdash \&lv : \&\tau} \qquad \text{(t-deref)}\frac{\Omega \mid \Gamma \mid \Sigma \vdash e : \&\tau}{\Omega \mid \Gamma \mid \Sigma \vdash *e : \tau} \qquad \text{(t-var)}\frac{(x : \sigma) \in \Gamma}{\Omega \mid \Gamma \mid \Sigma \vdash x : \sigma}$$

$$\text{(t-unit)}\frac{}{\Omega \mid \Gamma \mid \Sigma \vdash () : ()} \qquad \text{(t-newstruct)}\frac{S\,\langle \overline{X_j} \rangle\,\{\overline{x_i : \sigma_i}\} \in \Omega \quad \overline{\Omega \mid \Gamma \mid \Sigma \vdash e_i : [\overline{u_j/X_j}]\sigma_i}^{\,i}}{\Omega \mid \Gamma \mid \Sigma \vdash S\,\langle \overline{u_j} \rangle\,\{\overline{x_i : e_i}\} : S\,\langle \overline{u_j} \rangle}$$

$$\text{(t-proj)}\frac{\Omega \mid \Gamma \mid \Sigma \vdash e : S\,\langle \overline{u_j} \rangle \quad S\,\langle \overline{X_j} \rangle\,\{\overline{x_i : \sigma_i}\} \in \Omega \quad (x : \sigma) \in \overline{x_i : \sigma_i}}{\Omega \mid \Gamma \mid \Sigma \vdash e.x : [\overline{u_j/X_j}]\sigma} \qquad \text{(t-seq)}\frac{\Omega \mid \Gamma \mid \Sigma \vdash e_1 : \tau_1 \quad \Omega \mid \Gamma \mid \Sigma \vdash e_2 : \tau_2}{\Omega \mid \Gamma \mid \Sigma \vdash e_1; e_2 : \tau_2}$$

$$\text{(t-unpack)}\frac{\Omega \mid \Gamma \mid \Sigma \vdash e_1 : \&(\exists T, \overline{\tau_i}, \overline{\omega_j}) \quad \Omega, \overline{c_j : \omega_j} \mid \Gamma, T, \overline{x_i : \tau_i} \mid \Sigma \vdash e_2 : \tau \quad T \notin FV(\tau)}{\Omega \mid \Gamma \mid \Sigma \vdash \mathsf{let}\ (T, \overline{x_i}, \overline{c_j}) = \mathsf{unpack}\ e_1\ \mathsf{in}\ e_2 : \tau}$$

$$\text{(t-pack)}\frac{\overline{\Omega \mid \Gamma \mid \Sigma \vdash e_i : [\tau/T]\tau_i}^{\,i} \quad \overline{\Omega \vdash \gamma_j : [\tau/T]\omega_j}^{\,j}}{\Omega \mid \Gamma \mid \Sigma \vdash \mathsf{pack}\ (\tau, \overline{e_i}, \overline{\gamma_j})\ \mathsf{as}\ \&(\exists T, \overline{\tau_i}, \overline{\omega_j}) : \&(\exists T, \overline{\tau_i}, \overline{\omega_j})}$$

$$\text{(t-tapp)}\frac{\Omega \mid \Gamma \mid \Sigma \vdash e : \forall \overline{X_i}.\sigma}{\Omega \mid \Gamma \mid \Sigma \vdash e[\overline{u_i}] : [\overline{u_i/X_i}]\sigma} \qquad \text{(t-tabs)}\frac{\Omega \mid \Gamma, \overline{X_i} \mid \Sigma \vdash e : \sigma}{\Omega \mid \Gamma \mid \Sigma \vdash \Lambda \overline{X_i}.e : \forall \overline{X_i}.\sigma}$$

$$\text{(t-capp)}\frac{\Omega \mid \Gamma \mid \Sigma \vdash e : \forall \overline{\omega_i}.\sigma \quad \overline{\Omega \vdash \gamma_i : \omega_i}}{\Omega \mid \Gamma \mid \Sigma \vdash e[\![\overline{\gamma_i}]\!] : \sigma} \qquad \text{(t-cabs)}\frac{\Omega, \overline{c_i : \omega_i} \mid \Gamma \mid \Sigma \vdash e : \sigma}{\Omega \mid \Gamma \mid \Sigma \vdash \Lambda \overline{c_i : \omega_i}.e : \forall \overline{\omega_i}.\sigma}$$

$$\text{(t-loc)}\frac{(l : \sigma) \in \Sigma}{\Omega \mid \Gamma \mid \Sigma \vdash l : \sigma} \qquad \text{(t-coerce)}\frac{\Omega \mid \Gamma \mid \Sigma \vdash e : \sigma_1 \quad \Omega \vdash \gamma : \sigma_1 \sim \sigma_2}{\Omega \mid \Gamma \mid \Sigma \vdash e \blacktriangleright \gamma : \sigma_2}$$

Figure 6.4: RustIn: well-typed terms

$\boxed{\Omega \mid \Gamma \vdash \mathit{item} : \Omega \mid \Gamma}$ (Well-typed items)

$$\text{(t-type)}\ \frac{}{\Omega \mid \Gamma \vdash \mathsf{type}\ A\ \langle \overline{X_i} \rangle : [\mathsf{type}\ A\ \langle \overline{X_i} \rangle] \mid \emptyset}$$

$$\text{(t-axiom)}\ \frac{(\mathsf{type}\ A\ \langle \overline{X_i} \rangle) \in \Omega \qquad [\overline{u_i / X_i}]\ \mathit{defined} \qquad [A \mapsto u]\ \mathit{defined}}{\Omega \mid \Gamma \vdash \mathsf{axiom}\ c : \forall \overline{X_j}.A\ \langle \overline{u_i} \rangle \mapsto u : [c : \forall \overline{X_j}.A\ \langle \overline{u_i} \rangle \sim \forall \overline{X_j}.u] \mid \emptyset}$$

$$\text{(t-struct)}\ \frac{}{\Omega \mid \Gamma \vdash \mathsf{struct}\ S\ \langle \overline{X_i} \rangle\ \{\overline{x_j : \sigma_j}\} : [S\ \langle \overline{X_i} \rangle\ \{\overline{x_j : \sigma_j}\}] \mid \emptyset}$$

$$\text{(t-decl)}\ \frac{\Omega \mid \Gamma \mid \emptyset \vdash v : \sigma}{\Omega \mid \Gamma \vdash x : \sigma = v; : \emptyset \mid [x : \sigma]}$$

$\boxed{\vdash \mathit{pgm} : \Omega \mid \Gamma}$ (Well-typed programs)

$$\text{(t-pgm)}\ \frac{\overline{\Omega \mid \Gamma \vdash \mathit{item}_i : \Omega_i \mid \Gamma_i} \qquad \Omega = \overline{\Omega_i} \qquad \Gamma = \overline{\Gamma_i}}{\vdash \overline{\mathit{item}_i} : \Omega \mid \Gamma}$$

Figure 6.5: RustIn: well-typed items and programs

the likeness between coercions and terms.

Rule (t-proj) is used for typing struct field projections, which can be polymorphic. Note that rule (t-ref) only lets us have references to values that have a monomorphic s-type τ, so polymorphic struct fields cannot be referenced in RustIn (the restriction follows from the semantics of Rust, where only monomorphic items can be referenced).

Rule (t-pack) is used to type object packing terms, which create an instance of an existential object. The rule (t-unpack) is used to type object unpacking terms. The side condition $T \notin FV(\tau)$ ensures that the existential variable T does not escape its scope, delimited by the term e_2.

Rule (t-loc), used for typing store locations, is the only rule that uses of the store typing environment Σ. There are no terms or items in the language that extend the store typing environment—the environment is a tool for proving type safety of RustIn in section 6.5.

Finally, the rule (t-coerce) is used to type coerced terms, whose types are converted from type σ_1 to σ_2 as dictated by the type of the coercion γ.

6.3.3 Well-typed items and programs

Figure 6.5 presents the well-typed items and well-typed programs judgments. The judgments are similar to the analogous judgments in MiniRust. Of interest are (t-type) and (t-axiom), which are used for typing type and axiom declarations respectively. (t-type) simply populates

the environment Ω with the name of the abstract type constructor, along with its parameters. In the side conditions of (t-axiom) we verify that the abstract type constructor A has been declared and that the abstract type's parameters are instantiated with types of appropriate kind. The predicates $[\overline{u/X}]$ *defined* and $[A \mapsto u]$ *defined* are the same as in MiniRust. The rule populates the environment Ω with the resulting coercion.

6.4 Operational semantics

We present the operational semantics of RustIn in the small-step (structural) style. We have two mutually dependent evaluation relations: one for evaluating terms, and another for evaluating lvalues. We treat lvalues in a similar way as the Cyclone operational semantics [19].

6.4.1 Evaluating terms

The relation $\langle \mu, e \rangle \xrightarrow{\Omega} \langle \mu, e \rangle$, presented in figures 6.6 and 6.7, represents a single step of computation in evaluating terms. The relation is parameterized by the top-level environment Ω, which contains coercion typings and struct definitions. Throughout the evaluation rules we carry the store μ, which is a mapping from locations to store values.

In RustIn, every term variable has a corresponding location in the store (this follows from Rust's memory model where the data assigned to a local variable is stored directly on the stack). For every variable declaration, we assign a location to the variable and in the declaration's scope we substitute all instances of the variable with the location (the substitution allows us to directly reference store locations). For instance, in rule (e-let1), in the side condition $\langle \mu', l \rangle = \mathbf{alloc}(\mu, v)$ we store the value v at location l (**alloc** is defined in figure 6.10). Then, in e_2, we substitute all instances of the variable x with l. Similarly, for function calls in rule (e-fapp1), we allocate the values of the function arguments in locations $\overline{l_i}$ and we substitute the formal parameters $\overline{x_i}$ with their respective locations in the body of the function. As such, locations l evaluate to the values to which they are mapped in the store, as shown in rule (e-loc). However, note that a reference to a location, $\&l$ is a value itself and cannot be reduced.

In the evaluation relation we take special care when coercions are involved. In particular, we want to make sure that the evaluated term has the same type after every computation step, as it is an important part of our type safety proof. For instance, we use rule (e-fapp2) to evaluate function applications where the function's type is coerced by the coercion γ. We want to get rid of the coercion γ so that we can apply the function to the arguments $\overline{e_i}$. However, we then need to coerce the types of the arguments $\overline{e_i}$ so that they match with the argument types $\overline{\tau_i}$ that the function $\mathsf{fn}\ (\overline{x_i : \tau_i})\{e\}$ expects. Therefore, we apply the coercions $\overline{\gamma_i}$, which we create based on γ, to the arguments $\overline{e_i}$. Similarly, as we need the function application

$\boxed{\langle \mu, e\rangle \xrightarrow{\Omega} \langle \mu, e\rangle}$ (Evaluation of terms)

$$\text{(e-let1)}\ \frac{\langle \mu', l\rangle = \mathbf{alloc}(\mu, v)}{\langle \mu, \mathsf{let}\ x : \tau = v\ \mathsf{in}\ e_2\rangle \xrightarrow{\Omega} \langle \mu', [l/x]e_2\rangle} \qquad \text{(e-upd1)}\ \frac{\langle \mu, lv\rangle \xrightarrow{lv(\Omega)} \langle \mu', lv'\rangle}{\langle \mu, lv := e\rangle \xrightarrow{\Omega} \langle \mu', lv' := e\rangle}$$

$$\text{(e-upd2)}\ \frac{}{\langle \mu, l := v\rangle \xrightarrow{\Omega} \langle \mathbf{update}_{\Omega}(\mu, l, v), ()\rangle}$$

$$\text{(e-upd3)}\ \frac{}{\langle \mu, (l \blacktriangleright \gamma) := e\rangle \xrightarrow{\Omega} \langle \mu, l := (e \blacktriangleright \mathsf{sym}\ \gamma)\rangle} \qquad \text{(e-loc)}\ \frac{}{\langle \mu, l\rangle \xrightarrow{\Omega} \langle \mu, \mathbf{getValue}(\mu, l)\rangle}$$

$$\text{(e-deref1)}\ \frac{}{\langle \mu, *\&l\rangle \xrightarrow{\Omega} \langle \mu, l\rangle} \qquad \text{(e-deref2)}\ \frac{}{\langle \mu, *(\&l \blacktriangleright \gamma)\rangle \xrightarrow{\Omega} \langle \mu, l \blacktriangleright *\gamma\rangle}$$

$$\text{(e-ref1)}\ \frac{\langle \mu, lv\rangle \xrightarrow{lv(\Omega)} \langle \mu', lv'\rangle}{\langle \mu, \&lv\rangle \xrightarrow{\Omega} \langle \mu', \&lv'\rangle} \qquad \text{(e-ref2)}\ \frac{}{\langle \mu, \&(l \blacktriangleright \gamma)\rangle \xrightarrow{\Omega} \langle \mu, \&l \blacktriangleright \&\gamma\rangle}$$

$$\text{(e-fapp1)}\ \frac{\mu_0 = \mu \qquad \overline{\langle \mu_i, l_i\rangle = \mathbf{alloc}(\mu_{i-1}, v_i)}^{\,i}}{\langle \mu, \mathsf{fn}\ (\overline{x_i : \tau_i})\{e\}(\overline{v_i})\rangle \xrightarrow{\Omega} \langle \mu_i, [\overline{l_i/x_i}]e\rangle}$$

$$\text{(e-fapp2)}\ \frac{\overline{\gamma_i = \mathsf{sym}\ (\mathsf{farg}\ (i, \gamma))} \qquad \gamma_R = \mathsf{fret}\ (\gamma)}{\langle \mu, (\mathsf{fn}\ (\overline{x_i : \tau_i})\{e\} \blacktriangleright \gamma)(\overline{e_i})\rangle \xrightarrow{\Omega} \langle \mu, \mathsf{fn}\ (\overline{x_i : \tau_i})\{e\}(\overline{e_i \blacktriangleright \gamma_i}) \blacktriangleright \gamma_R\rangle}$$

$$\text{(e-seq1)}\ \frac{}{\langle \mu, v; e_2\rangle \xrightarrow{\Omega} \langle \mu, e_2\rangle} \qquad \text{(e-proj1)}\ \frac{(x : v) \in \overline{x_i : v_i}}{\langle \mu, S\ \langle \overline{u}\rangle\ \{\overline{x_i : v_i}\}.x\rangle \xrightarrow{\Omega} \langle \mu, v\rangle}$$

$$\text{(e-proj2)}\ \frac{S\ \langle \overline{X_j}\rangle\ \{\overline{x_i : \sigma_i}\} \in \Omega \qquad \Omega \vdash \gamma : S\ \langle \overline{u_{1j}}\rangle \sim S\ \langle \overline{u_{2j}}\rangle \qquad \overline{\gamma_j = \mathsf{spar}\ (j, \gamma)}}{\langle \mu, (S\ \langle \overline{u_{1j}}\rangle\ \{\overline{x_i : v_i}\} \blacktriangleright \gamma).x\rangle \xrightarrow{\Omega} \langle \mu, S\ \langle \overline{u_{2j}}\rangle\ \overline{\{x_i : v_i \blacktriangleright [\![\overline{\gamma_j/X_j}]\!]\sigma_i\}}.x\rangle}$$

$$\text{(e-unpack1)}\ \frac{\mu_0 = \mu \qquad \overline{\langle \mu_i, l_i\rangle = \mathbf{alloc}(\mu_{i-1}, v_i)}^{\,i}}{\langle \mu, \mathsf{let}\ (T, \overline{x_i}, \overline{c_j}) = \mathsf{unpack}\ (\mathsf{pack}\ (\tau_1, \overline{v_i}, \overline{\gamma_j})\ \mathsf{as}\ \tau_2)\ \mathsf{in}\ e\rangle \xrightarrow{\Omega} \langle \mu_i, [\overline{l_i/x_i}][\tau_1/T]e\rangle}$$

$$\text{(e-unpack2)}\ \frac{\begin{array}{c}\Omega \vdash \gamma : \&(\exists T, \overline{\tau_{1i}}, \overline{\omega_{1j}}) \sim \&(\exists T, \overline{\tau_{2i}}, \overline{\omega_{2j}}) \\ \overline{\gamma_i = \mathsf{objf}\ (\tau_1, i, *\gamma)} \qquad \overline{\gamma_{2_j} = (\mathsf{sym}\ (\mathsf{objc\text{-}l}\ (\tau_1, j, *\gamma))) \circ \gamma_{1_j} \circ \mathsf{objc\text{-}r}\ (\tau_1, j, *\gamma)}^{\,j}\end{array}}{\begin{array}{l}\langle \mu, \mathsf{let}\ (T, \overline{x_i}, \overline{c_j}) = \mathsf{unpack}\ ((\mathsf{pack}\ (\tau_1, \overline{v_i}, \overline{\gamma_{1_j}})\ \mathsf{as}\ \&(\exists T, \overline{\tau_{1i}}, \overline{\omega_{1j}})) \blacktriangleright \gamma)\ \mathsf{in}\ e\rangle \\ \xrightarrow{\Omega} \quad \langle \mu, \mathsf{let}\ (T, \overline{x_i}, \overline{c_j}) = \mathsf{unpack}\ (\mathsf{pack}\ (\tau_1, \overline{v_i \blacktriangleright \gamma_i}, \overline{\gamma_{2_j}})\ \mathsf{as}\ \&(\exists T, \overline{\tau_{2i}}, \overline{\omega_{2j}}))\ \mathsf{in}\ e\rangle\end{array}}$$

... continued in figure 6.7

Figure 6.6: RustIn: evaluation of terms

$\boxed{\langle \mu, e\rangle \xrightarrow{\Omega} \langle \mu, e\rangle}$ (Evaluation of terms)

$$\frac{\langle \mu, e\rangle \xrightarrow{\Omega} \langle \mu', e'\rangle}{}$$

(e-let2)	$\langle \mu, \mathsf{let}\ x : \tau = e\ \mathsf{in}\ e_2\rangle$	$\xrightarrow{\Omega}$	$\langle \mu', \mathsf{let}\ x : \tau = e'\ \mathsf{in}\ e_2\rangle$
(e-upd4)	$\langle \mu, l := e\rangle$	$\xrightarrow{\Omega}$	$\langle \mu', l := e'\rangle$
(e-deref3)	$\langle \mu, *e\rangle$	$\xrightarrow{\Omega}$	$\langle \mu', *e'\rangle$
(e-fapp3)	$\langle \mu, e(\overline{e})\rangle$	$\xrightarrow{\Omega}$	$\langle \mu', e'(\overline{e})\rangle$
(e-fapp4)	$\langle \mu, \mathsf{fn}\ (\overline{x:\tau})\{e_1\}(\overline{v_i}, e, \overline{e_j})\rangle$	$\xrightarrow{\Omega}$	$\langle \mu', \mathsf{fn}\ (\overline{x:\tau})\{e_1\}(\overline{v_i}, e', \overline{e_j})\rangle$
(e-seq2)	$\langle \mu, e; e_2\rangle$	$\xrightarrow{\Omega}$	$\langle \mu', e'; e_2\rangle$
(e-proj3)	$\langle \mu, e.x\rangle$	$\xrightarrow{\Omega}$	$\langle \mu', e'.x\rangle$
(e-struct)	$\langle \mu, S\,\langle \overline{u}\rangle\,\{\overline{x_i : v_i}, x : e, \overline{x_j : e_j}\}\rangle$	$\xrightarrow{\Omega}$	$\langle \mu', S\,\langle \overline{u}\rangle\,\{\overline{x_i : v_i}, x : e', \overline{x_j : e_j}\}\rangle$
(e-tapp3)	$\langle \mu, e[\overline{u_i}]\rangle$	$\xrightarrow{\Omega}$	$\langle \mu, e'[\overline{u_i}]\rangle$
(e-capp3)	$\langle \mu, e[\![\overline{\gamma}]\!]\rangle$	$\xrightarrow{\Omega}$	$\langle \mu, e'[\![\overline{\gamma}]\!]\rangle$
(e-pack)	$\langle \mu, \mathsf{pack}\ (\tau_1, \overline{v_i}, e, \overline{e_j}, \overline{\gamma})\ \mathsf{as}\ \tau_2\rangle$	$\xrightarrow{\Omega}$	$\langle \mu', \mathsf{pack}\ (\tau_1, \overline{v_i}, e', \overline{e_j}, \overline{\gamma})\ \mathsf{as}\ \tau_2\rangle$
(e-unpack3)	$\langle \mu, \mathsf{let}\ (T, \overline{x}) = \mathsf{unpack}\ e\ \mathsf{in}\ e_2\rangle$	$\xrightarrow{\Omega}$	$\langle \mu', \mathsf{let}\ (T, \overline{x}) = \mathsf{unpack}\ e'\ \mathsf{in}\ e_2\rangle$
(e-coerce2)	$\langle \mu, e \blacktriangleright \gamma\rangle$	$\xrightarrow{\Omega}$	$\langle \mu', e' \blacktriangleright \gamma\rangle$

$$\text{(e-tapp1)}\ \frac{}{\langle \mu, (\Lambda\overline{X_i}.e)[\overline{u_i}]\rangle \xrightarrow{\Omega} \langle \mu, [\overline{u_i/X_i}]e\rangle}$$

$$\text{(e-tapp2)}\ \frac{}{\langle \mu, ((\Lambda\overline{X_i}.e) \blacktriangleright \gamma)[\overline{u_i}]\rangle \xrightarrow{\Omega} \langle \mu, [\overline{u_i/X_i}]e \blacktriangleright \gamma[\overline{u_i}]\rangle}$$

$$\text{(e-capp1)}\ \frac{}{\langle \mu, (\Lambda\overline{c_i : \omega_i}.e)[\![\overline{\gamma_i}]\!]\rangle \xrightarrow{\Omega} \langle \mu, [\![\overline{\gamma_i/c_i}]\!]e\rangle}$$

$$\text{(e-capp2)}\ \frac{\gamma' = \mathsf{coer}\ (\gamma) \qquad \overline{\gamma'_i = \mathsf{coer\text{-}l}\ (i, \gamma) \circ \gamma_i \circ \mathsf{sym}\ (\mathsf{coer\text{-}r}\ (i, \gamma))}^{\,i}}{\langle \mu, ((\Lambda\overline{c_i : \omega_i}.e) \blacktriangleright \gamma)[\![\overline{\gamma_i}]\!]\rangle \xrightarrow{\Omega} \langle \mu, [\![\overline{\gamma'_i/c_i}]\!]e \blacktriangleright \gamma'\rangle}$$

$$\text{(e-coerce1)}\ \frac{}{\langle \mu, (pv \blacktriangleright \gamma_1) \blacktriangleright \gamma_2\rangle \xrightarrow{\Omega} \langle \mu, pv \blacktriangleright (\gamma_1 \circ \gamma_2)\rangle}$$

Figure 6.7: RustIn: evaluation of terms continued

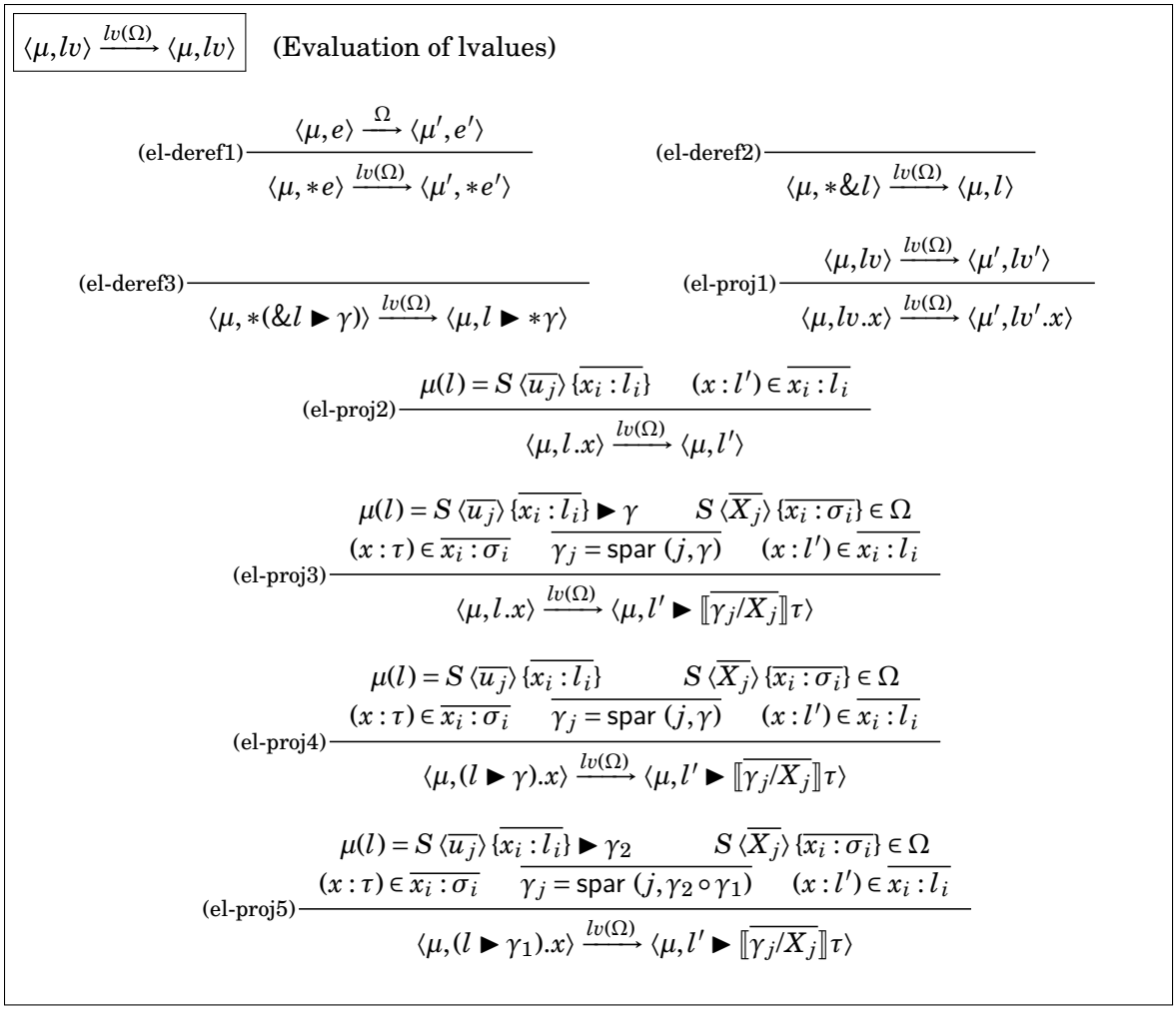

Figure 6.8: RustIn: evaluation of lvalues

terms on both sides of the relation to have the same types, we need to coerce the entire term using γ_R, which we again obtain from deconstructing γ.

6.4.2 Evaluating lvalues

Some rules of the term evaluation relation use in their premises the lvalue evaluation relation $\langle \mu, lv \rangle \xrightarrow{lv(\Omega)} \langle \mu', lv' \rangle$. Every step of this relation reduces an lvalue towards a c-location. We use the relation in the term evaluation rules (e-upd1) and (e-ref1). In rule (e-upd1) we use the lvalue relation to reduce the left-hand side of the term $lv := e$ to a c-location, so that we can update the resulting location with the value of e. In rule (e-ref1), where we evaluate a reference, we also use the lvalue relation to reduce the referenced lvalue to a location.

The lvalue evaluation relation is presented in figure 6.8. Note that in rule (el-deref1) we

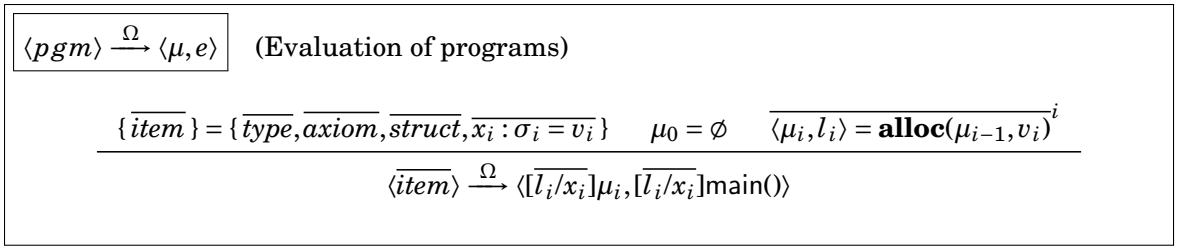

Figure 6.9: RustIn: evaluation of programs

use the term evaluation relation in its premise to reduce the term e. Rules (el-proj*) evaluate field access lvalues $lv.x$. An lvalue $lv.x$ intuitively reduces to the location of the field x of the struct instance that is stored at the location represented by lv. In rules (el-proj3), (el-proj4) and (el-proj5) we add coercions to the resulting lvalues to ensure that their types match the types of the lvalues on the left-hand side of the relation. In the applied coercions $[\![\overline{\gamma_j/X_j}]\!]\tau$ the double brackets $[\![$ and $]\!]$ denote coercion substitutions: $\overline{X_j}$ and τ are identity coercions in this context. We use a similar substitution in the term evaluation relation in figure 6.6 in rule (e-proj2). Also note that only monomorphic fields are evaluated as lvalues. This is due to the restriction in RustIn's type system where we can only have references to monomorphic values and the update operation $lv := e$ can only apply to monomorphic terms.

6.4.3 Evaluating programs

Figure 6.9 presents the program evaluation relation. The relation is defined by a single rule. As type, axiom and struct declarations don't have any computation associated with them, we only use top-level term declarations. In the rule's side conditions we allocate the declared values in the store and we substitute all top-level variable names with their corresponding locations in the resulting store. Then we apply the main function, which we assume to be a member of the top-level variables $\overline{x_i}$.

Note that the top-level terms need to already be values so that we can store them directly in the store. Otherwise, as the top-level terms may be mutually dependent, we would need to do more work to determine their evaluation order, and reduce them to values before storing them.

6.4.4 Evaluation metafunctions

In figure 6.10 we present metafunctions that we use throughout the evaluation rules for RustIn.

The function $\mathbf{alloc}(\mu, v)$ takes as inputs a store μ and a value v, and it outputs a tuple $\langle \mu', l \rangle$ that contains a *new* location l where the value v is allocated and the updated store

$$
\begin{array}{lcll}
\mathbf{alloc}(\mu, w) & = & \langle \mu[l \mapsto w], l \rangle & \text{where } l \notin dom(\mu) \\
\mathbf{alloc}(\mu_0, S\,\langle \overline{u} \rangle\,\{\overline{x_i : v_i}\}) & = & \langle \mu_i[l \mapsto S\,\langle \overline{u} \rangle\,\{\overline{x_i : l_i}\}], l \rangle & \text{where } \overline{\langle \mu_i, l_i \rangle = \mathbf{alloc}(\mu_{i-1}, v_i)}^{\,i} \\
 & & & \quad l \notin dom(\mu_i) \\
\mathbf{alloc}(\mu_0, S\,\langle \overline{u} \rangle\,\{\overline{x_i : v_i}\} \blacktriangleright \gamma) & = & \langle \mu_i[l \mapsto S\,\langle \overline{u} \rangle\,\{\overline{x_i : l_i}\} \blacktriangleright \gamma], l \rangle & \text{where } \overline{\langle \mu_i, l_i \rangle = \mathbf{alloc}(\mu_{i-1}, v_i)}^{\,i} \\
 & & & \quad l \notin dom(\mu_i)
\end{array}
$$

$$
\begin{array}{lcll}
\mathbf{update}_\Omega(\mu, l, w) & = & \mu[l \mapsto w] & \\
\mathbf{update}_\Omega(\mu_0, l, S\,\langle \overline{u_j} \rangle\,\{\overline{x_i : v_i}\}) & = & \mu_i \quad \text{where} & \mu_0(l) = S\,\langle \overline{u_j} \rangle\,\{\overline{x_i : l_i}\} \\
 & & & \overline{\mu_i = \mathbf{update}_\Omega(\mu_{i-1}, l_i, v_i)} \\
\mathbf{update}_\Omega(\mu_0, l, S\,\langle \overline{u_j} \rangle\,\{\overline{x_i : v_i}\}) & = & \mu_i \quad \text{where} & \mu_0(l) = S\,\langle \overline{u'_j} \rangle\,\{\overline{x_i : l_i}\} \blacktriangleright \gamma \\
 & & & \Omega \vdash \gamma : S\,\langle \overline{u'_j} \rangle \sim S\,\langle \overline{u_j} \rangle \\
 & & & \overline{S\,\langle \overline{X_j} \rangle\,\{\overline{x_i : \sigma_i}\} \in \Omega} \\
 & & & \overline{\gamma_j = \mathsf{spar}\ (j, \mathsf{sym}\ \gamma)} \\
 & & & \overline{\mu_i = \mathbf{update}_\Omega(\mu_{i-1}, l_i, v_i \blacktriangleright\!\blacktriangleright [\![\overline{\gamma_j / X_j}]\!] \sigma_i)}^{\,i} \\
\mathbf{update}_\Omega(\mu_0, l, S\,\langle \overline{u_j} \rangle\,\{\overline{x_i : v_i}\} \blacktriangleright \gamma) & = & \mu_i \quad \text{where} & \mu_0(l) = S\,\langle \overline{u'_j} \rangle\,\{\overline{x_i : l_i}\} \\
 & & & \Omega \vdash \gamma : S\,\langle \overline{u_j} \rangle \sim S\,\langle \overline{u'_j} \rangle \\
 & & & \overline{S\,\langle \overline{X_j} \rangle\,\{\overline{x_i : \sigma_i}\} \in \Omega} \\
 & & & \overline{\gamma_j = \mathsf{spar}\ (j, \gamma)} \\
 & & & \overline{\mu_i = \mathbf{update}_\Omega(\mu_{i-1}, l_i, v_i \blacktriangleright\!\blacktriangleright [\![\overline{\gamma_j / X_j}]\!] \sigma_i)}^{\,i} \\
\mathbf{update}_\Omega(\mu_0, l, S\,\langle \overline{u_j} \rangle\,\{\overline{x_i : v_i}\} \blacktriangleright \gamma) & = & \mu_i \quad \text{where} & \mu_0(l) = S\,\langle \overline{u'_j} \rangle\,\{\overline{x_i : l_i}\} \blacktriangleright \gamma' \\
 & & & \Omega \vdash \gamma : S\,\langle \overline{u_j} \rangle \sim \tau \\
 & & & \Omega \vdash \gamma' : S\,\langle \overline{u'_j} \rangle \sim \tau \\
 & & & \overline{S\,\langle \overline{X_j} \rangle\,\{\overline{x_i : \sigma_i}\} \in \Omega} \\
 & & & \overline{\gamma_j = \mathsf{spar}\ (j, \gamma \circ \mathsf{sym}\ \gamma')} \\
 & & & \overline{\mu_i = \mathbf{update}_\Omega(\mu_{i-1}, l_i, v_i \blacktriangleright\!\blacktriangleright [\![\overline{\gamma_j / X_j}]\!] \sigma_i)}^{\,i}
\end{array}
$$

undefined otherwise

$$
\begin{array}{lcll}
\mathbf{getValue}(\mu, l) & = & S\,\langle \overline{u} \rangle\,\{\overline{x_i : \mathbf{getValue}(\mu, l_i)}\} & \text{where } \mu(l) \equiv S\,\langle \overline{u} \rangle\,\{\overline{x_i : l_i}\} \\
\mathbf{getValue}(\mu, l) & = & S\,\langle \overline{u} \rangle\,\{\overline{x_i : \mathbf{getValue}(\mu, l_i)}\} \blacktriangleright \gamma & \text{where } \mu(l) \equiv S\,\langle \overline{u} \rangle\,\{\overline{x_i : l_i}\} \blacktriangleright \gamma \\
\mathbf{getValue}(\mu, l) & = & \mu(l) & \text{otherwise}
\end{array}
$$

$$
\begin{array}{lcl}
(pv \blacktriangleright \gamma') \blacktriangleright\!\blacktriangleright \gamma & = & pv \blacktriangleright (\gamma' \circ \gamma) \\
pv \blacktriangleright\!\blacktriangleright \gamma & = & pv \blacktriangleright \gamma
\end{array}
$$

Figure 6.10: RustIn: metafunctions

μ'. If the allocated value is a store value w then the allocation is simple: a new location is selected from the outside of the domain of μ, the value w is stored at that location, and the updated store that includes the new mapping is output by the function.

When **alloc** is applied to a struct instance, then more work is required because a struct instance value is not a store value. In that case we recursively store the fields of the struct instance in new locations and we allocate the struct instance value—with the fields $\overline{v_i}$ replaced with their respective locations $\overline{l_i}$—in the resulting store. We proceed similarly with coerced struct instances.

Note that, strictly speaking, **alloc** is not a function, as it can choose any location l from the outside of the domain of the store, and there is nothing that prevents it from choosing a different location each time the function is applied to the same arguments. In fact, **alloc** defines a *family of functions* that are equivalent up to the choice of location labels. We use the function in rules (e-let1), (e-fapp1) and (e-unpack1) in the evaluation relation in figure 6.6, and in the program evaluation relation in figure 6.9.

The function $\mathbf{update}_\Omega(\mu, l, v)$ is parameterized by the environment Ω. It takes as inputs a store μ, a location label l and a value v to be stored at l. We use the function **update** when the location l is already in the domain of μ and its type is the same as the type of the value v (specifically, we use it in rule (e-upd2) in the evaluation relation in figure 6.6). The output of the function is an updated store μ', where the value at location l is modified accordingly. When the input value is a store value w, then the update operation is straightforward. When the input value is a struct instance, then the exact store value chosen to update the store at location l depends on what is the existing store value at that location. To preserve the types of the affected locations, we may have to apply coercions to the stored values. Note that the **update** function also reuses the same locations for struct fields.

In the definition of **update** we use the binary helper function $\blacktriangleright\!\!\blacktriangleright$ defined at the bottom of the figure. We can interpret the helper function as a "value-preserving coercion". The function takes as inputs a value and a coercion, and it outputs the application of the coercion on the input value. If the input value is a coerced plain value $pv \blacktriangleright \gamma'$ then when we apply γ to it, we obtain $(pv \blacktriangleright \gamma') \blacktriangleright \gamma$, which is not a value. The function $(pv \blacktriangleright \gamma') \blacktriangleright\!\!\blacktriangleright \gamma$ thus takes an evaluation step, composing the coercions γ' and γ so that its result $pv \blacktriangleright (\gamma' \circ \gamma)$ is a value. We use the helper function in recursive calls to the **update** function to ensure that we apply the function to a value in accordance with its domain.

The function **getValue** takes as inputs a store μ and a location l, and it outputs a value v corresponding to the store value stored at location l in μ. If the stored value is a struct instance, then its fields are locations (and not values), so we recursively apply **getValue** on the struct's fields' locations to obtain the appropriate values. We use the function in rule (e-loc) of the evaluation relation in figure 6.6 to obtain the value stored at location l.

6.5 Type safety

Having defined the operational semantics for RustIn, we can reason about the type safety of well-typed programs. Before we present the relevant theorems, we first set up some definitions.

We first define a relation parametric on the store μ. The relation defines an ordering on store values related by struct instances that occur in μ.

Definition 6.1. *Let $A_\mu = \{w \mid \exists l.\mu(l) = w\}$. In other words, A_μ is the image of μ.*
Define $\sqsubset_\mu \subseteq A_\mu \times A_\mu$ as follows:

- $\mu(l) \sqsubset_\mu S\,\langle\overline{u}\rangle\,\{\overline{x_m : l_n}, x : l, \overline{x_n : l_n}\}$
- $\mu(l) \sqsubset_\mu S\,\langle\overline{u}\rangle\,\{\overline{x_m : l_n}, x : l, \overline{x_n : l_n}\} \blacktriangleright \gamma$

Then we introduce the notion of a well-typed store.

Definition 6.2 (Well-typed store)**.** *We say that $\Omega \mid \Gamma \mid \Sigma \vdash \mu$, meaning that the store μ is well typed with respect to the environments Ω, Γ and the store typing Σ if*

1. $dom(\mu) = dom(\Sigma)$ *and*
2. $\Omega \mid \Gamma \mid \Sigma \vdash \mu(l) : \Sigma(l)$ *for every* $l \in dom(\mu)$
3. $\sqsubset_\mu$ *is a well-founded relation on the set* $A_\mu = \{w \mid \exists l.\mu(l) = w\}$*, i.e., it has no infinite descending chains.*

Note that the definition requires the relation $\sqsubset_\mu$ to be well-founded. Intuitively, this means that the store should not contain circular struct instances where, for example, a field of a struct instance is that particular struct instance itself. In particular, this is required to ensure that the metafunctions **getValue** and **update** are terminating.[1]

The concept of *type equality axiom consistency*, is another important piece of type safety of RustIn. Its definition is adapted from System FC [38] whose type safety proof also relies on axiom consistency. The definition uses a notion of *constructed types* (called *value types* in System FC), defined below.

Definition 6.3 (Constructed type)**.** *A **constructed type** is a type η that is of the form* $(), S\,\langle\overline{u}\rangle, fn(\overline{\tau}) \to \tau, \&u, (\exists T, \overline{\tau}, \overline{\omega}), \forall\overline{X_i}.\eta$ *or* $\forall\overline{\omega_i}.\eta$.

Roughly speaking, a constructed type is a type whose outermost type constructor is some concrete type, i.e., a constructed type is neither an abstract type nor a type variable.

[1] Note that a struct instance may still contain a pointer to itself. For example, the assignment: $x_0 := S\{x : \&x_0\}$, where S is a struct type constructor, is allowed.

Definition 6.4 (Consistency of coercion environment Ω)**.** *We say that Ω is **consistent** if the following hold:*

1. *If $\Omega \vdash \gamma : S\,\langle \overline{u} \rangle \sim \eta$ and η is a constructed type then $\eta = S\,\langle \overline{u'} \rangle$ for some $\overline{u'}$.*
2. *If $\Omega \vdash \gamma : fn(\overline{\tau_i}) \to \tau \sim \eta$ and η is a constructed type then $\eta = fn(\overline{\tau'_i}) \to \tau'$ for some $\overline{\tau'_i}, \tau'$.*
3. *If $\Omega \vdash \gamma : \&u_1 \sim \eta$ and η is a constructed type then $\eta = \&u'_1$ for some u'_1.*
4. *If $\Omega \vdash \gamma : (\exists T, \overline{\tau_i}, \overline{\omega_j}) \sim \eta$ and η is a constructed type then $\eta = (\exists T, \overline{\tau'_i}, \overline{\omega'_j})$ for some $\overline{\tau'_i}, \overline{\omega'_j}$.*
5. *If $\Omega \vdash \gamma : \forall \overline{X_i}.\sigma \sim \eta'$ where η' is any type, then $\eta' = \forall \overline{X_i}.\sigma'$ for some σ'.*
6. *If $\Omega \vdash \gamma : \forall \overline{\omega_i}.\sigma \sim \eta'$ where η' is any type, then $\eta' = \forall \overline{\omega'_i}.\sigma'$ for some $\overline{\omega'_i}, \sigma'$.*

A consistent environment Ω prevents us from being able to conclude inconsistent type equalities such as `i32` $\sim$ `bool`.

We can now state the theorems that together demonstrate type safety of RustIn. We first prove that if a well-typed term steps to another term then both terms have the same type.

Theorem 6.1 (Preservation)**.** *Suppose Ω is a consistent environment.*

1. *If $\Omega \mid \Gamma \mid \Sigma \vdash e : \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and $\langle \mu, e \rangle \xrightarrow{\Omega} \langle \mu', e' \rangle$*
   then, there is some store typing $\Sigma' \supseteq \Sigma$ such that
   $\Omega \mid \Gamma \mid \Sigma' \vdash e' : \sigma$ and $\Omega \mid \Gamma \mid \Sigma' \vdash \mu'$.
2. *If $\Omega \mid \Gamma \mid \Sigma \vdash lv : \tau$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and $\langle \mu, lv \rangle \xrightarrow{lv(\Omega)} \langle \mu', lv' \rangle$*
   then, there is some store typing $\Sigma' \supseteq \Sigma$ such that
   $\Omega \mid \Gamma \mid \Sigma' \vdash lv' : \tau$ and $\Omega \mid \Gamma \mid \Sigma' \vdash \mu'$.

The second part of the type safety proof is the proof of progress, which roughly says that a well-typed term is either a value or it can be reduced. In other words, a well-typed term doesn't get 'stuck'.

Theorem 6.2 (Progress)**.** *Suppose Ω is a consistent environment.*

1. *If $\Omega \mid \emptyset \mid \Sigma \vdash e : \sigma$ then either $e \in$ VAL or, for any store μ such that $\Omega \mid \emptyset \mid \Sigma \vdash \mu$, there is some term e' and store μ' with $\langle \mu, e \rangle \xrightarrow{\Omega} \langle \mu', e' \rangle$*
2. *If $\Omega \mid \emptyset \mid \Sigma \vdash lv : \tau$ then either $lv \in$ CLOC or, for any store μ such that $\Omega \mid \emptyset \mid \Sigma \vdash \mu$, there is some lvalue lv' and store μ' with $\langle \mu, lv \rangle \xrightarrow{lv(\Omega)} \langle \mu', lv' \rangle$*

The previous two theorems together paint the type safety picture of MiniRust, stating that, unless evaluation diverges, every well-typed term reduces to a value with all types preserved.

The last theorem we present concerns the program evaluation rule. It roughly states that if a program is well-typed, produces a consistent environment Ω and contains the `main` function, then we can successfully take a type-preserving evaluation step.

Theorem 6.3. *Suppose* $\vdash pgm : \Omega \mid \Gamma$ *where* Ω *is consistent and* $\Gamma = \{\overline{x_i : \sigma_i}\}$.
If $(\mathsf{main} : \mathsf{fn}() \to ()) \in \Gamma$, *then there are some* μ, e, Σ *for which*
$\langle pgm \rangle \xrightarrow{\Omega} \langle \mu, e \rangle$, *and* $\Omega \mid \emptyset \mid \Sigma \vdash \mu$, *and* $\Omega \mid \emptyset \mid \Sigma \vdash e : ()$.

The proofs of the above theorems and all helper lemmas are found in appendix A.

Chapter 7

Translating MiniRust programs to RustIn

Rust traits introduce type-based overloading to the language, where the runtime semantics of overloaded terms depend on the types at which the terms are used. As mentioned in the previous chapter, defining the operational semantics directly in languages with type-based overloading is challenging. For instance, to define them directly in MiniRust, we would need to somehow encode impl lookup in the semantics to resolve constraints. A simpler way to do this, which is how Haskell type classes are implemented [16], is through translation to an internal language where the trait constraints have already been resolved. As such, in the previous chapter, we developed an internal language, RustIn, which doesn't have traits, and we provided operational semantics and a proof of type safety for it. In this chapter we show how MiniRust programs can be translated to RustIn programs. The translation that we present here is type-directed: the translation of a term is decided in part by its type. As such, we present the translation rules by extending the MiniRust typing rules presented in chapter 5. We then prove that the translation preserves well-typedness, which brings us closer to proving that MiniRust is type-safe.

7.1 Translation at a glance

The salient part of the type-directed translation from MiniRust to RustIn is the way in which constraints are translated. Satisfied MiniRust constraints translate to some expressions that provide *evidence of constraint entailment* [20]. In MiniRust, terms with qualified types can only be used if their qualifying constraints are entailed by the constraint environment. In turn, the RustIn translations of such terms are abstracted over the evidence that their qualifying constraints are satisfied. Having the evidence in the translation of a qualified term aligns with the explicit nature of expressions in our target language, RustIn, where terms

encode their typing derivation and coercions encode proof of equivalence of type expressions.

Recall that MiniRust trait constraints have form $u : D\,\langle \overline{u_i}, A \mapsto u_A \rangle$. For the purpose of translation we consider such constraints as combinations of a simple trait constraint $u : D\,\langle \overline{u_i} \rangle$ and an equality constraint $A_D\,\langle u, \overline{u_i} \rangle \sim u_A$. As such, the translation of an extended constraint produces two pieces of evidence.

The evidence of trait constraint entailment in RustIn takes the form of a dictionary that contains the trait function and the dictionaries corresponding to the trait's supertraits and associated type constraints. The dictionary is implemented using a RustIn struct. A MiniRust function that is qualified by one or more trait constraints is then translated to a higher-order function that takes as additional inputs the dictionaries corresponding to the qualifying constraints. Similarly, a MiniRust trait function is translated to a higher-order function that takes as input the dictionary corresponding to an impl of the trait. In the body of the function we take the appropriate concrete function from the input dictionary and apply it.

The evidence of equality constraint entailment in RustIn takes the form of a coercion. Recall that a RustIn coercion already encodes in it the proof of type equivalence of the two type expressions that it relates. An equality constraint entailment derivation then constructs the corresponding coercion. A MiniRust term qualified by an equality constraint (or rather by an extended constraint that contains an associated type instantiation) is translated to a coercion abstraction. To use the resulting RustIn term, we apply it to the concrete coercion that represents a proof of type equality.

We use RustIn abstract types to serve as translation targets of associated types. As such, an associated type declaration in a trait declaration is translated to an abstract type declaration and an associated type instantiation in an impl is translated to a type equality axiom.

7.2 Syntax

The formalization in this chapter relates two languages: MiniRust and RustIn. We use the syntax of both, presented in figures 5.1 and 6.1 respectively. Since the syntactic constructs in the two languages overlap, we add a superscript t to RustIn constructs.

To aid with the translation, we modify the MiniRust typing and constraint environments. The modified environments are defined below (new parts are highlighted):

Γ	::=	$\emptyset \mid \Gamma, x : \sigma \mid \Gamma, X \mid \Gamma, S\,\langle \overline{X} \rangle\,\{\overline{x : \tau}\}$	(typing environments)
	|	$\Gamma, (D\,\langle \overline{X} \rangle \text{ where } \overline{\pi}, \mathit{Self} : \overline{\beta \rightsquigarrow x}, A : \overline{\beta \rightsquigarrow x}, f, \mathit{obj})$	
Θ	::=	$\emptyset \mid \Theta, \theta \rightsquigarrow \langle x, c \rangle$	(constraint environments)

The trait information tuple in the typing environment Γ is extended with the names of

the dictionary fields that correspond to the trait's supertrait and associated type constraints. Specifically, in a trait information tuple $(D\,\langle\overline{X_i}\rangle \text{ where } \overline{\pi}, Self : \overline{\beta_s \rightsquigarrow x_s}, A : \overline{\beta_a \rightsquigarrow x_a}, f, obj)$, the variables $\overline{x_s}, \overline{x_a}$ are field names of the RustIn struct that represents the dictionary of the trait D. Specifically, $\overline{x_s}$ correspond to fields that contain its supertrait dictionaries and $\overline{x_a}$ correspond to fields that contain dictionaries that represent D's associated type constraints.

The constraint environment Θ is extended to bind constraint schemes to RustIn term and coercion variables. The variables correspond to the translations of the constraint schemes in the resulting RustIn program.

7.3 Translating types

We extend the MiniRust well-formedness judgments to translate MiniRust types (and type schemes) as well as constraints (and constraint schemes) to RustIn types. Figure 7.1 presents the judgments. We read the well-formed types relation, $\Gamma \mid \Theta \vdash_{\text{WF}} u \rightsquigarrow u^t$, as "under environments Γ and Θ, a well-formed MiniRust type u translates to RustIn type u^t". Most of the translation rules are straightforward. The most interesting rule in the relation is (wf-desc), used to translate trait object descriptors. MiniRust trait object descriptors translate to the more general RustIn existential object descriptors. The translation makes evident that a trait object encapsulates a reference to the concrete value that implements the object's trait and the dictionary $S_D\,\langle T, \overline{u_i^t}\rangle$ corresponding to the object's underlying concrete impl. The existential object also encapsulates coercions that correspond to the associated type equalities in the descriptor.

Note that in the premises of rule (wf-desc) and (wf-tscheme) we extend the dictionary environment Θ with some constraints without providing their corresponding translation variables. The translation variables are not used in the well-formedness judgments so we simply omit them to simplify the presentation. Similarly, in the premise $\Gamma \mid \Theta \Vdash u : D\,\langle\overline{u_i}, A \mapsto \star\rangle$ of rule (wf-atype) we omit the translation of the constraint since we do not use it.

The well-formed constraint schemes judgment $\Gamma \mid \Theta \vdash_{\text{WF}} \theta \rightsquigarrow \langle\sigma^t, \vartheta\rangle$ is used to obtain the types of the evidence produced by the constraint scheme θ. The resulting translation is a tuple that consists of the type scheme σ^t of the trait constraint portion of the constraint scheme and a polymorphic coercion type ϑ corresponding to the coercion that represents the equality constraint. Note that the identity associated type instantiation $A \mapsto \star$ in rule (wf-trcons) translates to an identity coercion.

7.4 Translating constraints

Figure 7.2 presents the constraint entailment judgments with translation. In the trait constraint entailment judgment, $\Gamma \mid \Theta \Vdash \pi \rightsquigarrow \langle e^t, \gamma\rangle$, a satisfied trait constraint π is translated to

$\boxed{\Gamma \mid \Theta \vdash_{\text{WF}} u \rightsquigarrow u^t}$ (Well-formed types)

$$\text{(wf-var)}\ \frac{X \in \Gamma}{\Gamma \mid \Theta \vdash_{\text{WF}} X \rightsquigarrow X} \qquad \text{(wf-unit)}\ \frac{}{\Gamma \mid \Theta \vdash_{\text{WF}} () \rightsquigarrow ()}$$

$$\text{(wf-fun)}\ \frac{\overline{\Gamma \mid \Theta \vdash_{\text{WF}} \tau_i \rightsquigarrow \tau_i^t} \qquad \Gamma \mid \Theta \vdash_{\text{WF}} \tau \rightsquigarrow \tau^t}{\Gamma \mid \Theta \vdash_{\text{WF}} \mathsf{fn}(\overline{\tau_i}) \to \tau \rightsquigarrow \mathsf{fn}(\overline{\tau_i^t}) \to \tau^t} \qquad \text{(wf-ref)}\ \frac{\Gamma \mid \Theta \vdash_{\text{WF}} u \rightsquigarrow u^t}{\Gamma \mid \Theta \vdash_{\text{WF}} \&u \rightsquigarrow \&u^t}$$

$$\text{(wf-struct)}\ \frac{\Gamma \vdash S\,\langle \overline{u_i} \rangle \qquad \overline{\Gamma \mid \Theta \vdash_{\text{WF}} u_i \rightsquigarrow u_i^t}}{\Gamma \mid \Theta \vdash_{\text{WF}} S\,\langle \overline{u_i} \rangle \rightsquigarrow S\,\langle \overline{u_i^t} \rangle} \qquad \text{(wf-atype)}\ \frac{\Gamma \mid \Theta \Vdash u : D\,\langle \overline{u_i}, A \mapsto \star \rangle \qquad \overline{\Gamma \mid \Theta \vdash_{\text{WF}} u \rightsquigarrow u^t} \quad \overline{\Gamma \mid \Theta \vdash_{\text{WF}} u_i \rightsquigarrow u_i^t}}{\Gamma \mid \Theta \vdash_{\text{WF}} A_D\,\langle u, \overline{u_i} \rangle \rightsquigarrow A_D\,\langle u^t, \overline{u_i^t} \rangle}$$

$$\text{(wf-desc)}\ \frac{\Gamma \vdash D\ \textit{obj-safe} \qquad \Gamma \vdash \mathsf{Self}_{\mathsf{U}} : D\,\langle \overline{u_i} \rangle \qquad \overline{\Gamma \mid \Theta \vdash_{\text{WF}} u_i \rightsquigarrow u_i^t} \qquad \overline{\Gamma, T \mid \Theta, T : D\,\langle \overline{u_i} \rangle \vdash_{\text{WF}} A_{D\,j}\,\langle T, \overline{u_{s\,j}} \rangle \rightsquigarrow A_{D\,j}\,\langle T, \overline{u_{s\,j}^t} \rangle}^{\,j} \quad \overline{\Gamma \mid \Theta \vdash_{\text{WF}} u_j \rightsquigarrow u_j^t}}{\Gamma \mid \Theta \vdash_{\text{WF}} \exists T : D \langle \overline{u_i}, \overline{A_{D\,j}\,\langle T, \overline{u_{s\,j}} \rangle \sim u_j} \rangle \rightsquigarrow \left(\exists T, \&T, S_D\,\langle T, \overline{u_i^t} \rangle, \overline{A_{D\,j}\,\langle T, \overline{u_{s\,j}^t} \rangle \sim u_j^t} \right)}$$

$\boxed{\Gamma \mid \Theta \vdash_{\text{WF}} \sigma \rightsquigarrow \sigma^t}$ (Well-formed type schemes)

$$\text{(wf-tscheme)}\ \frac{\overline{\Gamma, \overline{X_i} \mid \Theta, \overline{\pi_j} \vdash_{\text{WF}} \pi_j \rightsquigarrow \langle \tau_j^t, \omega_j \rangle}^{\,j} \qquad \Gamma, \overline{X_i} \mid \Theta, \overline{\pi_j} \vdash_{\text{WF}} \tau \rightsquigarrow \tau^t}{\Gamma \mid \Theta \vdash_{\text{WF}} \forall \overline{X_i}.\overline{\pi_j} \Rightarrow \tau \rightsquigarrow \forall \overline{X_i}.\forall \overline{\omega_j}.\mathsf{fn}(\overline{\tau_j^t}) \to \tau^t}$$

$\boxed{\Gamma \mid \Theta \vdash_{\text{WF}} \theta \rightsquigarrow \langle \sigma^t, \vartheta \rangle}$ (Well-formed constraint schemes)

$$\text{(wf-cscheme)}\ \frac{\overline{\Gamma, \overline{X_i} \mid \Theta, \overline{\pi_j} \vdash_{\text{WF}} \pi_j \rightsquigarrow \langle \tau_j^t, \omega_j \rangle}^{\,j} \qquad \Gamma, \overline{X_i} \mid \Theta, \overline{\pi_j} \vdash_{\text{WF}} \pi \rightsquigarrow \langle \tau^t, u_1^t \sim u_2^t \rangle}{\Gamma \mid \Theta \vdash_{\text{WF}} \forall \overline{X_i}.\overline{\pi_j} \Rightarrow \pi \rightsquigarrow \left\langle \forall \overline{X_i}.\forall \overline{\omega_j}.\mathsf{fn}(\overline{\tau_j^t}) \to \tau^t, \forall \overline{X_i}.u_1^t \sim \forall \overline{X_i}.u_2^t \right\rangle}$$

$$\text{(wf-treqcons)}\ \frac{\Gamma \vdash u : D\,\langle \overline{u_i} \rangle \qquad \Gamma \vdash A_D \qquad \Gamma \mid \Theta \vdash_{\text{WF}} u \rightsquigarrow u^t \qquad \overline{\Gamma \mid \Theta \vdash_{\text{WF}} u_i \rightsquigarrow u_i^t} \qquad \Gamma \mid \Theta \vdash_{\text{WF}} u_A \rightsquigarrow u_A^t}{\Gamma \mid \Theta \vdash_{\text{WF}} u : D\,\langle \overline{u_i}, A \mapsto u_A \rangle \rightsquigarrow \left\langle S_D\,\langle u^t, \overline{u_i^t} \rangle, A_D\,\langle u^t, \overline{u_i^t} \rangle \sim u_A^t \right\rangle}$$

$$\text{(wf-trcons)}\ \frac{\Gamma \vdash u : D\,\langle \overline{u_i} \rangle \qquad \Gamma \vdash A_D \qquad \Gamma \mid \Theta \vdash_{\text{WF}} u \rightsquigarrow u^t \qquad \overline{\Gamma \mid \Theta \vdash_{\text{WF}} u_i \rightsquigarrow u_i^t}}{\Gamma \mid \Theta \vdash_{\text{WF}} u : D\,\langle \overline{u_i}, A \mapsto \star \rangle \rightsquigarrow \left\langle S_D\,\langle u^t, \overline{u_i^t} \rangle, A_D\,\langle u^t, \overline{u_i^t} \rangle \sim A_D\,\langle u^t, \overline{u_i^t} \rangle \right\rangle}$$

Figure 7.1: MiniRust: well-formedness judgments with translation

$\boxed{\Gamma \mid \Theta \Vdash \pi \rightsquigarrow \langle e^t, \gamma \rangle}$ (Trait constraint entailment)

$$\text{(c-ext)}\ \frac{(\forall \overline{X_i}.\overline{\pi_j} \Rightarrow \pi \rightsquigarrow \langle x, c \rangle) \in \Theta \qquad \overline{\Gamma \mid \Theta \vdash_{\text{WF}} u_i \rightsquigarrow u_i^t} \qquad \overline{\Gamma \mid \Theta \Vdash [\overline{u_i/X_i}]\pi_j \rightsquigarrow \langle e_j^t, \gamma_j \rangle}}{\Gamma \mid \Theta \Vdash [\overline{u_i/X_i}]\pi \rightsquigarrow \langle x[\overline{u_i^t}][\![\overline{\gamma_j}]\!](\overline{e_j^t}), c[\overline{u_i^t}] \rangle}$$

$$\text{(c-treq1)}\ \frac{\begin{array}{c}\Gamma \mid \Theta \Vdash u_1 : D\,\langle \overline{u_{1i}}, A \mapsto u_3 \rangle \rightsquigarrow \langle e^t, \gamma_1 \rangle \qquad \Gamma \vdash u_2 : D\,\langle \overline{u_{2i}} \rangle \\ \Gamma \mid \Theta \Vdash u_1 \sim u_2 \rightsquigarrow \gamma_2 \qquad \overline{\Gamma \mid \Theta \Vdash u_{1i} \sim u_{2i} \rightsquigarrow \gamma_i}^{\,i} \qquad \Gamma \mid \Theta \Vdash u_3 \sim u_4 \rightsquigarrow \gamma_3\end{array}}{\Gamma \mid \Theta \Vdash u_2 : D\,\langle \overline{u_{2i}}, A \mapsto u_4 \rangle \rightsquigarrow \langle e^t \blacktriangleright S_D\,\langle \gamma_2, \overline{\gamma_i} \rangle, A_D\,\langle \mathsf{sym}\ (\gamma_2), \overline{\mathsf{sym}\ (\gamma_i)} \rangle \circ \gamma_1 \circ \gamma_3 \rangle}$$

$$\text{(c-treq2)}\ \frac{\begin{array}{c}\Gamma \mid \Theta \Vdash u_1 : D\,\langle \overline{u_{1i}}, A \mapsto \star \rangle \rightsquigarrow \langle e^t, \gamma_1 \rangle \qquad \Gamma \vdash u_2 : D\,\langle \overline{u_{2i}} \rangle \qquad \Gamma \mid \Theta \Vdash u_1 \sim u_2 \rightsquigarrow \gamma_2 \\ \overline{\Gamma \mid \Theta \Vdash u_{1i} \sim u_{2i} \rightsquigarrow \gamma_i}^{\,i} \qquad \Gamma \mid \Theta \vdash_{\text{WF}} u_2 \rightsquigarrow u_2^t \qquad \overline{\Gamma \mid \Theta \vdash_{\text{WF}} u_{2i} \rightsquigarrow u_{2i}^t}\end{array}}{\Gamma \mid \Theta \Vdash u_2 : D\,\langle \overline{u_{2i}}, A \mapsto \star \rangle \rightsquigarrow \langle e^t \blacktriangleright S_D\,\langle \gamma_2, \overline{\gamma_i} \rangle, A_D\,\langle u_2^t, \overline{u_{2i}^t} \rangle \rangle}$$

$$\text{(c-astar)}\ \frac{\Gamma \mid \Theta \Vdash u : D\,\langle \overline{u_i}, A \mapsto u_A \rangle \rightsquigarrow \langle e^t, \gamma \rangle \qquad \Gamma \mid \Theta \vdash_{\text{WF}} u \rightsquigarrow u^t \qquad \overline{\Gamma \mid \Theta \vdash_{\text{WF}} u_i \rightsquigarrow u_i^t}}{\Gamma \mid \Theta \Vdash u : D\,\langle \overline{u_i}, A \mapsto \star \rangle \rightsquigarrow \langle e^t, A_D\,\langle u^t, \overline{u_i^t} \rangle \rangle}$$

$\boxed{\Gamma \mid \Theta \Vdash u \sim u \rightsquigarrow \gamma}$ (Equality constraint entailment)

$$\text{(eq-sep)}\ \frac{\Gamma \mid \Theta \Vdash u : D\,\langle \overline{u_i}, A \mapsto u_A \rangle \rightsquigarrow \langle e^t, \gamma \rangle}{\Gamma \mid \Theta \Vdash A_D\,\langle u, \overline{u_i} \rangle \sim u_A \rightsquigarrow \gamma} \qquad \text{(eq-refl)}\ \frac{\Gamma \mid \Theta \vdash_{\text{WF}} u \rightsquigarrow u^t}{\Gamma \mid \Theta \Vdash u \sim u \rightsquigarrow u^t}$$

$$\text{(eq-trans)}\ \frac{\begin{array}{c}\Gamma \mid \Theta \Vdash u_1 \sim u_3 \rightsquigarrow \gamma_1 \\ \Gamma \mid \Theta \Vdash u_3 \sim u_2 \rightsquigarrow \gamma_2\end{array}}{\Gamma \mid \Theta \Vdash u_1 \sim u_2 \rightsquigarrow \gamma_1 \circ \gamma_2} \qquad \text{(eq-sym)}\ \frac{\Gamma \mid \Theta \Vdash u_2 \sim u_1 \rightsquigarrow \gamma}{\Gamma \mid \Theta \Vdash u_1 \sim u_2 \rightsquigarrow \mathsf{sym}\ (\gamma)}$$

$$\text{(eq-struct)}\ \frac{\begin{array}{c}\Gamma \vdash S\,\langle \overline{u_{1i}} \rangle \qquad \Gamma \vdash S\,\langle \overline{u_{2i}} \rangle \\ \overline{\Gamma \mid \Theta \Vdash u_{1i} \sim u_{2i} \rightsquigarrow \gamma_i}^{\,i}\end{array}}{\Gamma \mid \Theta \Vdash S\,\langle \overline{u_{1i}} \rangle \sim S\,\langle \overline{u_{2i}} \rangle \rightsquigarrow S\,\langle \overline{\gamma_i} \rangle} \qquad \text{(eq-ref)}\ \frac{\Gamma \mid \Theta \Vdash u_1 \sim u_2 \rightsquigarrow \gamma}{\Gamma \mid \Theta \Vdash \&u_1 \sim \&u_2 \rightsquigarrow \&\gamma}$$

$$\text{(eq-fun)}\ \frac{\overline{\Gamma \mid \Theta \Vdash \tau_{1i} \sim \tau_{2i} \rightsquigarrow \gamma_i}^{\,i} \qquad \Gamma \mid \Theta \Vdash \tau_1 \sim \tau_2 \rightsquigarrow \gamma}{\Gamma \mid \Theta \Vdash \mathsf{fn}(\overline{\tau_{1i}}) \to \tau_1 \sim \mathsf{fn}(\overline{\tau_{2i}}) \to \tau_2 \rightsquigarrow \mathsf{fn}(\overline{\gamma_i}) \to \gamma}$$

$$\text{(eq-atype)}\ \frac{\Gamma \vdash u_1 : D\,\langle \overline{u_{1i}} \rangle \qquad \Gamma \vdash u_2 : D\,\langle \overline{u_{2i}} \rangle \qquad \Gamma \mid \Theta \Vdash u_1 \sim u_2 \rightsquigarrow \gamma \qquad \overline{\Gamma \mid \Theta \Vdash u_{1i} \sim u_{2i} \rightsquigarrow \gamma_i}}{\Gamma \mid \Theta \Vdash A_D\,\langle u_1, \overline{u_{1i}} \rangle \sim A_D\,\langle u_2, \overline{u_{2i}} \rangle \rightsquigarrow A_D\,\langle \gamma, \overline{\gamma_i} \rangle}$$

$$\text{(eq-obj)}\ \frac{\begin{array}{c}\Gamma \vdash \mathsf{Self}_{\mathbf{U}} : D\,\langle \overline{u_{1i}} \rangle \qquad \Gamma \vdash \mathsf{Self}_{\mathbf{U}} : D\,\langle \overline{u_{2i}} \rangle \qquad \overline{\Gamma \vdash T : D_j\,\langle \overline{u_{1sj}}^{s} \rangle}^{\,j} \qquad \overline{\Gamma \vdash T : D_j\,\langle \overline{u_{2sj}}^{s} \rangle}^{\,j} \\ \overline{\Gamma \mid \Theta \Vdash u_{1i} \sim u_{2i} \rightsquigarrow \gamma_i} \qquad \overline{\Gamma, T \mid \Theta, T : D\,\langle \overline{u_{1i}} \rangle \Vdash u_{1sj} \sim u_{2sj} \rightsquigarrow \gamma_{sj}} \qquad \overline{\Gamma \mid \Theta \Vdash u_{1j} \sim u_{2j} \rightsquigarrow \gamma_j}\end{array}}{\begin{array}{l}\Gamma \mid \Theta \Vdash \exists T : D\,\langle \overline{u_{1i}}, \overline{A_{Dj}\,\langle T, \overline{u_{1sj}} \rangle \sim u_{1j}} \rangle \sim \exists T : D\,\langle \overline{u_{2i}}, \overline{A_{Dj}\,\langle T, \overline{u_{2sj}} \rangle \sim u_{2j}} \rangle \\ \quad \rightsquigarrow (\exists T, \&T, S_D\,\langle T, \overline{\gamma_i} \rangle, \overline{A_{Dj}\,\langle T, \overline{\gamma_{sj}} \rangle \sim \gamma_j})\end{array}}$$

Figure 7.2: MiniRust: constraint entailment with translation

a term e^t and a coercion γ, representing the constraint's trait and equality parts respectively.

In rule (c-ext), the side condition $(\forall \overline{X_i}.\overline{\pi_j} \Rightarrow \pi \rightsquigarrow \langle x, c\rangle) \in \Theta$ gives us the variables x and c that correspond to the translation of the constraint scheme $\forall \overline{X_i}.\overline{\pi_j} \Rightarrow \pi$. The term variable x represents a higher-order function quantified by type variables $\overline{X_i}$ and abstracted over the evidence expressions corresponding to the constraints $\overline{\pi_j}$. In the conclusion of the rule, the translation dictionary term, $x[\overline{u_i^t}][\![\overline{\gamma_j}]\!](\overline{e_j^t})$, instantiates x with types $\overline{u_i^t}$ and applies the result to the evidence of entailment of constraints $\overline{[\overline{u_i/X_i}]\pi_j}$ (in the form of coercions $\overline{\gamma_j}$ and dictionary terms $\overline{e_j^t}$). The resulting coercion in the conclusion of the rule is the constraint scheme's coercion variable c instantiated with types $\overline{u_i^t}$—qualifying constraints $\overline{\pi_j}$ are not part of the translation of the equality part of the constraint. In rule (c-treq2), the identity associated type instantiation $A \mapsto \star$ simply translates to an identity coercion $A_D\,\langle u_2^t, \overline{u_{2i}^t}\rangle$ (recall that the representations of RustIn types also represent identity coercions).

In the equality constraint entailment relation, $\Gamma \mid \Theta \Vdash u \sim u \rightsquigarrow \gamma$, we translate equality constraints to coercions. The language of coercions in RustIn lets us encode the proof of equivalence of type expressions directly in the resulting coercion.

7.5 Translating terms

Figure 7.3 presents the well-typed terms judgment with translation. The translation of terms is fairly straightforward. Note that, since terms in the target language are explicitly typed, translations of MiniRust terms contain additional information from their types (such as in rules (let-un), (var) and (new-struct)).

In rule (var), if the term variable x has a qualified type scheme in the typing environment Γ, then its type scheme's type variables must be instantiated and its qualifying constraints satisfied before we can use x. As such, the translation of x includes explicit application of the type scheme's type variables and of the evidence expressions that correspond to the qualifying constraints $\overline{\pi_j}$.

In rule (obj-cast) we translate trait object instantiations to existential object packing terms. In the premise $\Gamma \mid \Theta \Vdash (\exists T : D\,\langle \overline{u_i}, \overline{u_{1j} \sim u_{2j}}\rangle) : D\,\langle \overline{u_i}\rangle$ we omit the translation of the constraint because we do not need it.

7.6 Translating items

Figure 7.4 presents the well-typed items judgment with translation. In the judgment, $\Gamma \mid \Theta \vdash \mathit{item} : \Gamma \mid \Theta \rightsquigarrow \mathit{pgm}^t$, an item translates to a partial program (a collection of items). The translation of struct declarations in rule (struct) is straightforward. Function declarations in (fun) translate to top-level term declarations. If a MiniRust function is qualified by constraints $\overline{\pi_j}$ in its where-clause, then the resulting function in RustIn is abstracted over the

$\boxed{\Gamma \mid \Theta \vdash e : \tau \rightsquigarrow e^t}$ (Well-typed terms)

$$\text{(sub)}\ \frac{\Gamma \mid \Theta \vdash e : \tau_1 \rightsquigarrow e^t \qquad \Gamma \mid \Theta \Vdash \tau_1 \sim \tau_2 \rightsquigarrow \gamma}{\Gamma \mid \Theta \vdash e : \tau_2 \rightsquigarrow e^t \blacktriangleright \gamma} \qquad \text{(as)}\ \frac{\Gamma \mid \Theta \vdash e : \tau \rightsquigarrow e^t}{\Gamma \mid \Theta \vdash e \text{ as } \tau : \tau \rightsquigarrow e^t}$$

$$\text{(unit)}\ \frac{}{\Gamma \mid \Theta \vdash () : () \rightsquigarrow ()} \qquad \text{(let-un)}\ \frac{\Gamma \mid \Theta \vdash e_1 : \tau_1 \rightsquigarrow e_1^t \quad \Gamma, x : \tau_1 \mid \Theta \vdash e_2 : \tau_2 \rightsquigarrow e_2^t \quad \Gamma \mid \Theta \vdash_{\mathrm{WF}} \tau_1 \rightsquigarrow \tau^t}{\Gamma \mid \Theta \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2 \rightsquigarrow \mathsf{let}\ x : \tau^t = e_1^t\ \mathsf{in}\ e_2^t}$$

$$\text{(seq)}\ \frac{\Gamma \mid \Theta \vdash e_1 : \tau_1 \rightsquigarrow e_1^t \quad \Gamma \mid \Theta \vdash e_2 : \tau_2 \rightsquigarrow e_2^t}{\Gamma \mid \Theta \vdash e_1; e_2 : \tau_2 \rightsquigarrow e_1^t; e_2^t} \qquad \text{(var)}\ \frac{(x : \forall \overline{X_i}.\overline{\pi_j} \Rightarrow \tau) \in \Gamma \quad \overline{\Gamma \mid \Theta \vdash_{\mathrm{WF}} u_i \rightsquigarrow u_i^t} \quad \overline{\Gamma \mid \Theta \Vdash [\overline{u_i/X_i}]\pi_j \rightsquigarrow \langle e_j^t, \gamma_j \rangle}^{\,j}}{\Gamma \mid \Theta \vdash x : [\overline{u_i/X_i}]\tau \rightsquigarrow x[\overline{u_i^t}][\![\overline{\gamma_j}]\!](\overline{e_j^t})}$$

$$\text{(app)}\ \frac{\Gamma \mid \Theta \vdash e : \mathsf{fn}(\overline{\tau_i}) \to \tau \rightsquigarrow e^t \quad \overline{\Gamma \mid \Theta \vdash e_i : \tau_i \rightsquigarrow e_i^t}}{\Gamma \mid \Theta \vdash e(\overline{e_i}) : \tau \rightsquigarrow e^t(\overline{e_i^t})} \qquad \text{(ref)}\ \frac{\Gamma \mid \Theta \vdash lv : \tau \rightsquigarrow lv^t}{\Gamma \mid \Theta \vdash \&lv : \&\tau \rightsquigarrow \&lv^t}$$

$$\text{(deref)}\ \frac{\Gamma \mid \Theta \vdash e : \&\tau \rightsquigarrow e^t}{\Gamma \mid \Theta \vdash *e : \tau \rightsquigarrow *e^t} \qquad \text{(asgn)}\ \frac{\Gamma \mid \Theta \vdash lv : \tau \rightsquigarrow lv^t \quad \Gamma \mid \Theta \vdash e : \tau \rightsquigarrow e^t}{\Gamma \mid \Theta \vdash lv := e : () \rightsquigarrow lv^t := e^t}$$

$$\text{(new-struct)}\ \frac{S\,\langle \overline{X_k} \rangle\,\{\overline{x_i : \tau_i}\} \in \Gamma \quad \overline{\Gamma \mid \Theta \vdash e_i : [\overline{u_k/X_k}]\tau_i \rightsquigarrow e_i^t} \quad \overline{\Gamma \mid \Theta \vdash_{\mathrm{WF}} u_k \rightsquigarrow u_k^t}}{\Gamma \mid \Theta \vdash S\{\overline{x_i : e_i}\} : S\,\langle \overline{u_k} \rangle \rightsquigarrow S\,\langle \overline{u_k^t} \rangle\,\{\overline{x_i : e_i^t}\}}$$

$$\text{(proj)}\ \frac{\Gamma \mid \Theta \vdash e : S\,\langle \overline{u_j} \rangle \rightsquigarrow e^t \qquad S\,\langle \overline{X_j} \rangle\,\{\overline{x_m : \tau_m}, x : \tau, \overline{x_n : \tau_n}\} \in \Gamma}{\Gamma \mid \Theta \vdash e.x : [\overline{u_j/X_j}]\tau \rightsquigarrow e^t.x}$$

$$\text{(obj-cast)}\ \frac{\begin{array}{c}\Gamma \mid \Theta \vdash e : \&\tau \rightsquigarrow e^t \qquad \Gamma \mid \Theta \vdash_{\mathrm{WF}} \tau \rightsquigarrow \tau_1^t \\ \Gamma \mid \Theta \Vdash \tau : D\,\langle \overline{u_i} \rangle \rightsquigarrow e_D^t \qquad \overline{\Gamma \mid \Theta \Vdash [\tau/T]u_{1j} \sim u_{2j} \rightsquigarrow \gamma_j} \\ \Gamma \mid \Theta \Vdash (\exists T : D\,\langle \overline{u_i}, \overline{u_{1j} \sim u_{2j}} \rangle) : D\,\langle \overline{u_i} \rangle \qquad \Gamma \mid \Theta \vdash_{\mathrm{WF}} \exists T : D\,\langle \overline{u_i}, \overline{u_{1j} \sim u_{2j}} \rangle \rightsquigarrow \tau_2^t\end{array}}{\Gamma \mid \Theta \vdash e : \&(\exists T : D\,\langle \overline{u_i}, \overline{u_{1j} \sim u_{2j}} \rangle) \rightsquigarrow \mathsf{pack}\ (\tau_1^t, e^t, e_D^t, \overline{\gamma_j})\ \mathsf{as}\ \tau_2^t}$$

Figure 7.3: MiniRust: well-typed terms with translation

$\boxed{\Gamma \mid \Theta \vdash \mathit{item} : \Gamma \mid \Theta \rightsquigarrow \mathit{pgm}^t}$ (Well-typed items)

$$\text{(struct)}\ \frac{\overline{\Gamma, \overline{X_j} \mid \Theta \vdash_{\text{WF}} \tau_i \rightsquigarrow \tau_i^t}^{\,i}}{\Gamma \mid \Theta \vdash \text{struct } S\,\langle \overline{X_j} \rangle\,\{\overline{x_i : \tau_i}\} : [S\,\langle \overline{X_j} \rangle\,\{\overline{x_i : \tau_i}\}] \mid \emptyset \rightsquigarrow \text{struct } S\,\langle \overline{X_j} \rangle\,\{\overline{x_i : \tau_i^t}\}}$$

$$\text{(fun)}\ \frac{\begin{array}{c}\sigma = \forall \overline{X_k}.\overline{\pi_j} \Rightarrow \text{fn}(\overline{\tau_i}) \to \tau \qquad \Gamma \mid \Theta \vdash_{\text{WF}} \sigma \rightsquigarrow \sigma^t \\ \sigma^t = \forall \overline{X_k}.\forall \overline{\omega_j}.\text{fn}(\overline{\tau_j^t}) \to \text{fn}(\overline{\tau_i^t}) \to \tau^t \qquad \Gamma, \overline{X_k}, \overline{x_i : \tau_i} \mid \Theta, \overline{\pi_j \rightsquigarrow \langle x_j, c_j \rangle} \vdash e : \tau \rightsquigarrow e^t\end{array}}{\Gamma \mid \Theta \vdash \text{fn } f\,\langle \overline{X_k} \rangle\,(\overline{x_i : \tau_i}) \to \tau \text{ where } \overline{\pi_j}\,\{\,e\,\} : [f : \sigma] \mid \emptyset \rightsquigarrow f : \sigma^t = \Lambda \overline{X_k}.\Lambda \overline{c_j : \omega_j}.\text{fn } (\overline{x_j : \tau_j^t})\{\,\text{fn } (\overline{x_i : \tau_i^t})\{\,e^t\,\}\,\}}$$

$$\text{(trait)}\ \frac{\begin{array}{c}\{\overline{\pi_j}\} \equiv \{(\overline{Self : \beta_s}), \overline{\pi_h}\} \qquad \overline{X_p} = Self, \overline{X_i} \qquad \overline{\Gamma \mid \Theta \vdash_{\text{WF}} \forall \overline{X_p}.(Self : D\,\langle \overline{X_i} \rangle) \Rightarrow \pi_h}^{\,h} \\ \sigma = \forall \overline{X_k}.\overline{\pi_l} \Rightarrow \text{fn}(\overline{\tau_m}) \to \tau_R \qquad \sigma' = \forall \overline{X_p}.(Self : D\,\langle \overline{X_i} \rangle) \Rightarrow \sigma \qquad \Gamma \mid \Theta \vdash_{\text{WF}} \sigma' \rightsquigarrow \sigma'^t \\ \Gamma, \overline{X_p} \mid \Theta, Self : D\,\langle \overline{X_i} \rangle \vdash_{\text{WF}} \sigma \rightsquigarrow \sigma^t \qquad \sigma^t = \forall \overline{X_k}.\forall \overline{\omega_l}.\text{fn}(\overline{\tau_l^t}) \to \text{fn}(\overline{\tau_m^t}) \to \tau_R^t \\ \dfrac{\overline{\Gamma \mid \Theta \vdash_{\text{WF}} \forall \overline{X_p}.(Self : D\,\langle \overline{X_i} \rangle) \Rightarrow Self : \beta_s \rightsquigarrow \langle \forall \overline{X_p}.\text{fn}(S_D\,\langle \overline{X_p} \rangle) \to \tau_s^t, \vartheta_s \rangle}^{\,s}}{\overline{\Gamma \mid \Theta \vdash_{\text{WF}} \forall \overline{X_p}.(Self : D\,\langle \overline{X_i} \rangle) \Rightarrow A_D\,\langle \overline{X_p} \rangle : \beta_a \rightsquigarrow \langle \forall \overline{X_p}.\text{fn}(S_D\,\langle \overline{X_p} \rangle) \to \tau_a^t, \vartheta_a \rangle}^{\,a}}\end{array}}{\begin{array}{l}\Gamma \mid \Theta \vdash \begin{array}{l}\text{trait } D\,\langle \overline{X_i} \rangle \text{ for } Self \\ \text{where } \overline{\pi_j}\,\{ \\ \quad \text{type } A : \overline{\beta_a}; \\ \quad f : \sigma;\,\}\end{array} : \begin{array}{l}[(D\,\langle \overline{X_i} \rangle \text{ where } \overline{\pi_h}, Self : \overline{\beta_s \rightsquigarrow x_s}, A : \overline{\beta_a \rightsquigarrow x_a}, \text{obj-unsafe}), f : \sigma'] \mid \\ [\overline{\forall \overline{X_p}.(Self : D\,\langle \overline{X_i} \rangle) \Rightarrow Self : \beta_s \rightsquigarrow \langle x_{D,s}, c_{D,s} \rangle}^{\,s}, \\ \overline{\forall \overline{X_p}.(Self : D\,\langle \overline{X_i} \rangle) \Rightarrow A_D\,\langle \overline{X_p} \rangle : \beta_a \rightsquigarrow \langle x_{D,a}, c_{D,a} \rangle}^{\,a}\ ]\end{array} \\ \rightsquigarrow \begin{array}{l}\text{type } A_D\,\langle \overline{X_p} \rangle; \overline{\text{axiom } c_{D,a} : \vartheta_a}^{\,a}; \overline{\text{axiom } c_{D,s} : \vartheta_s}^{\,s}; \text{struct } S_D\,\langle \overline{X_p} \rangle\,\{f : \sigma^t, \overline{x_s : \tau_s^t}, \overline{x_a : \tau_a^t}\} \\ f : \sigma'^t = \Lambda \overline{X_p}, \overline{X_k}.\Lambda \overline{c_l : \omega_l}.\text{fn } (x_D : S_D\,\langle \overline{X_p} \rangle, \overline{x_l : \tau_l^t})\{\,(x_D.f)[\overline{X_k}][\![\overline{c_l}]\!](\overline{x_l})\,\}; \\ \overline{x_{D,s} : \forall \overline{X_p}.\text{fn}(S_D\,\langle \overline{X_p} \rangle) \to \tau_s^t = \Lambda \overline{X_p}.\text{fn } (x_D : S_D\,\langle \overline{X_p} \rangle)\{x_D.x_s\}}^{\,s}; \\ \overline{x_{D,a} : \forall \overline{X_p}.\text{fn}(S_D\,\langle \overline{X_p} \rangle) \to \tau_a^t = \Lambda \overline{X_p}.\text{fn } (x_D : S_D\,\langle \overline{X_p} \rangle)\{x_D.x_a\}}^{\,a};\end{array}\end{array}}$$

$$\text{(impl)}\ \frac{\begin{array}{c}(D\,\langle \overline{X_i} \rangle \text{ where } \overline{\pi_h}, Self : \overline{\beta_s \rightsquigarrow x_s}^{\,s}, A : \overline{\beta_a \rightsquigarrow x_a}^{\,a}, f, \_) \in \Gamma \qquad \overline{u_p} = u, \overline{u_i} \qquad \overline{X_p} = Self, \overline{X_i} \\ \overline{X_k \in FV(\overline{u_p})} \qquad \Theta^* = \Theta \setminus \{\overline{\forall \overline{X_p}.(Self : D\,\langle \overline{X_i} \rangle) \Rightarrow Self : \beta_s \rightsquigarrow \_}^{\,s}, \overline{\forall \overline{X_p}.(Self : D\,\langle \overline{X_i} \rangle) \Rightarrow A_D\,\langle \overline{X_p} \rangle : \beta_a \rightsquigarrow \_}^{\,a}\} \\ \overline{\Gamma, \overline{X_k} \mid \Theta^*, \overline{\pi_j \rightsquigarrow \langle x_j, c_j \rangle} \Vdash [\overline{u_p/X_p}](Self : \beta_s) \rightsquigarrow \langle e_s^t, \gamma_s \rangle}^{\,s} \qquad \overline{\Gamma, \overline{X_k} \mid \Theta^*, \overline{\pi_j \rightsquigarrow \langle x_j, c_j \rangle} \Vdash [\overline{u_p/X_p}]\pi_h}^{\,h} \\ \Gamma, \overline{X_k} \mid \Theta^*, \overline{\pi_j \rightsquigarrow \langle x_j, c_j \rangle} \vdash_{\text{WF}} u_A \rightsquigarrow u_A^t \qquad \overline{\Gamma, \overline{X_k} \mid \Theta^*, \overline{\pi_j \rightsquigarrow \langle x_j, c_j \rangle} \Vdash (A_D\,\langle \overline{u_p} \rangle : [\overline{u_p/X_p}]\beta_a) \rightsquigarrow \langle e_a^t, \gamma_a \rangle}^{\,a} \\ (f : \forall \overline{X_p}.(Self : D\,\langle \overline{X_i} \rangle) \Rightarrow \sigma) \in \Gamma \qquad \Gamma, \overline{X_k} \mid \Theta^*, \overline{\pi_j \rightsquigarrow \langle x_j, c_j \rangle} \vdash \mathit{fun} : [f : \sigma'] \mid \emptyset \rightsquigarrow f : \sigma'^t = e_F^t \\ \sigma' = [\overline{u_p/X_p}]\sigma \qquad \Gamma \mid \Theta \vdash_{\text{WF}} \forall \overline{X_k}.\overline{\pi_j} \Rightarrow u : D\,\langle \overline{u_i} \rangle \rightsquigarrow \sigma_D^t \qquad \sigma_D^t = \forall \overline{X_k}.\forall \overline{\omega_j}.\text{fn}(\overline{\tau_j^t}) \to S_D\,\langle \overline{u_p^t} \rangle\end{array}}{\begin{array}{l}\Gamma \mid \Theta \vdash \begin{array}{l}\text{impl}\,\langle \overline{X_k} \rangle\,D\,\langle \overline{u_i} \rangle \text{ for } u \text{ where } \overline{\pi_j}\,\{ \\ \quad \text{type } A = u_A; \mathit{fun}\,\}\end{array} : \emptyset \mid [\forall \overline{X_k}.\overline{\pi_j} \Rightarrow u : D\,\langle \overline{u_i}, A \mapsto u_A \rangle \rightsquigarrow \langle x_{u:D\langle \overline{u_i} \rangle}, c_{u:D\langle \overline{u_i} \rangle} \rangle] \\ \rightsquigarrow \begin{array}{l}\text{axiom } c_{u:D\langle \overline{u_i} \rangle} : \forall \overline{X_k}.A_D\,\langle \overline{u_p^t} \rangle \mapsto u_A^t \\ x_{u:D\langle \overline{u_i} \rangle} : \sigma_D^t = \Lambda \overline{X_k}.\Lambda \overline{c_j : \omega_j}.\text{fn } (\overline{x_j : \tau_j^t})\,\{\,S_D\,\langle \overline{u_p^t} \rangle\,\{f : e_F^t, \overline{x_s : e_s^t}, \overline{x_a : e_a^t}\}\,\};\end{array}\end{array}}$$

...continued in figure 7.5

Figure 7.4: MiniRust: well-typed items with translation

trait dictionaries and coercions that correspond to the constraints $\overline{\pi_j}$.

7.6.1 Translating traits

Rule (trait) is used to type non-object-safe trait declarations and it provides their translations. The translation of a trait consists of multiple RustIn items. One of the resulting items is a struct declaration that defines the type of the trait's dictionary. The struct's fields include the trait function f, the dictionaries of the trait's supertraits $\overline{x_s}$ and the dictionaries of the trait's associated type constraints $\overline{x_a}$. Since supertrait and associated type constraints of a trait D are propagated, a satisfied constraint on D lets us infer that D's supertrait and associated type constraints are also satisfied. The fields $\overline{x_s}$ and $\overline{x_a}$ in D's dictionary are used as evidence of that. The RustIn functions $\overline{x_{D,s}}$ and $\overline{x_{D,a}}$ correspond to the propagated constraint schemes. They take as input a dictionary of D and they output their corresponding propagated dictionaries obtained from the input dictionary.

The translation of the declaration of a trait D also contains a top-level function f corresponding to D's trait function. It is a higher-order function that simply takes as input a dictionary of D and applies the concrete version of the trait function that is in the dictionary.

The associated type declared in the trait is translated to a corresponding abstract type declaration. Since the associated type instantiations in supertrait and associated type bounds are also propagated, they are translated to type equality axioms that declare the coercion constants $\overline{c_{D,a}}$ and $\overline{c_{D,s}}$.

7.6.2 Translating impls

The translation of an impl declaration in rule (impl) consists of two RustIn items that correspond to the rule's output constraint scheme. The first item is an axiom that declares the abstract type instantiation corresponding to the instantiation of the trait's associated type in the impl's body. The second item is a generic function that produces an instance of the struct declared by the implemented trait, representing the trait dictionary. If the impl is universally quantified by some variables $\overline{X_k}$ in its header, then its dictionary is also universally quantified by the type variables $\overline{X_k}$. Similarly, if the impl is qualified by some constraints $\overline{\pi_j}$ in its where-clause, then the dictionary is accordingly abstracted over coercions and dictionaries that represent those constraints. The supertrait and associated type constraint dictionaries e_s^t and e_a^t are obtained in the rule's side conditions that verify that that those constraints are satisfied by the impl.

$\boxed{\Gamma \mid \Theta \vdash \mathit{item} : \Gamma \mid \Theta \rightsquigarrow \mathit{pgm}^t}$ (Well-typed items)

$$\text{(obj-trt)}\ \frac{\begin{array}{c}\{\overline{\pi_j}\} \equiv \{(\mathsf{Self}_{\mathbf{U}} : \overline{\beta_s}), \overline{\pi_h}\} \qquad \overline{X_p} = \mathsf{Self}_{\mathbf{U}}, \overline{X_i} \qquad \overline{\Gamma \mid \Theta \vdash_{\mathrm{WF}} \forall \overline{X_p}.\mathsf{Self}_{\mathbf{U}} : D\,\langle \overline{X_i}\rangle \Rightarrow \pi_h \rightsquigarrow \_}^{\,h} \\ \tau = \mathsf{fn}(\&\mathsf{Self}_{\mathbf{U}}, \overline{\tau_m}) \to \tau_R \qquad \sigma' = \forall \overline{X_p}.(\mathit{Self} : D\,\langle \overline{X_i}\rangle) \Rightarrow \tau \qquad \Gamma \mid \Theta \vdash_{\mathrm{WF}} \sigma' \rightsquigarrow \sigma'^t \\ \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, \overline{X_p} \mid \Theta, \mathsf{Self}_{\mathbf{U}} : D\,\langle \overline{X_i}\rangle \vdash_{\mathrm{WF}} \tau \rightsquigarrow \sigma^t}{\Gamma \mid \Theta \vdash_{\mathrm{WF}} \forall \overline{X_p}.(\mathsf{Self}_{\mathbf{U}} : D\,\langle \overline{X_i}\rangle) \Rightarrow \mathsf{Self}_{\mathbf{U}} : \beta_s \rightsquigarrow \langle \forall \overline{X_p}.\mathsf{fn}(S_D\,\langle \overline{X_p}\rangle) \to \tau_s^t, \vartheta_s\rangle}^{\,s}}{\Gamma \mid \Theta \vdash_{\mathrm{WF}} \forall \overline{X_p}.(\mathsf{Self}_{\mathbf{U}} : D\,\langle \overline{X_i}\rangle) \Rightarrow A_D\,\langle \overline{X_p}\rangle : \beta_a \rightsquigarrow \langle \forall \overline{X_p}.\mathsf{fn}(S_D\,\langle \overline{X_p}\rangle) \to \tau_a^t, \vartheta_a\rangle}^{\,a} \\ \Gamma, \overline{X_p} \vdash D\,\langle \overline{X_i}\rangle \Uparrow \{\overline{D_d\,\langle \overline{u_{dn}}^{\,n}, A_d \mapsto \star\rangle}^{\,d}\} \qquad \overline{A_d :: X_d} \\ u_{\mathsf{obj}} = \exists T : D\,\langle \overline{X_i}, \overline{A_{Dd}\,\langle T, [T/\mathsf{Self}_{\mathbf{U}}]\overline{u_{dn}}^{\,n}\rangle} \sim X_d^{\,d}\rangle \\ \hline \pi_d = u_{\mathsf{obj}} : D_d\,\langle \overline{[u_{\mathsf{obj}}/\mathsf{Self}_{\mathbf{U}}]u_{dn}}^{\,n}, A_d \mapsto X_d\rangle}^{\,d} \\ \hline \Gamma, \overline{X_i}, \overline{X_d} \mid \Theta \mid \overline{\forall \overline{X_i}, \overline{X_d}.\pi_d \rightsquigarrow \langle x_d, c_d\rangle} \vdash_{\mathsf{os}} \pi_d \rightsquigarrow \langle e_d^t, \overline{\pi_{dm} \rightsquigarrow x_{dm}}\rangle}^{\,d} \\ \hline \Gamma \mid \Theta \vdash_{\mathrm{WF}} \forall \overline{X_i}, \overline{X_d}.\overline{\pi_{dm}}^{\,m} \Rightarrow \pi_d \rightsquigarrow \langle \forall \overline{X_i}, \overline{X_d}.\forall \overline{\omega_{dm}}.\mathsf{fn}(\overline{\tau_{dm}^t}) \to \tau_d^t,\ \forall \overline{X_i}.\overline{X_d}.u_d^t \sim \forall \overline{X_i}.\overline{X_d}.X_d\rangle}^{\,d} \end{array}}{\Gamma \mid \Theta \vdash \begin{array}{l}\mathsf{trait}\ D\,\langle \overline{X_i}\rangle\ \mathsf{for}\ \mathsf{Self}_{\mathbf{U}} \\ \mathsf{where}\ \overline{\pi_j}\,\{ \\ \quad \mathsf{type}\ A : \overline{\beta_a}; \\ \quad f : \tau;\ \}\end{array} : \begin{array}{l}[(D\,\langle \overline{X_i}\rangle\ \mathsf{where}\ \overline{\pi_h}, \mathsf{Self}_{\mathbf{U}} : \overline{\beta_s \rightsquigarrow x_s}, A : \overline{\beta_a \rightsquigarrow x_a}, f, \mathsf{obj\text{-}safe}), \\ \quad f : \sigma'] \mid \\ \quad \overline{[\forall \overline{X_p}.(\mathsf{Self}_{\mathbf{U}} : D\,\langle \overline{X_i}\rangle) \Rightarrow \mathsf{Self}_{\mathbf{U}} : \beta_s \rightsquigarrow \langle x_{D,s}, c_{D,s}\rangle}^{\,s}, \\ \quad \overline{\forall \overline{X_p}.(\mathsf{Self}_{\mathbf{U}} : D\,\langle \overline{X_i}\rangle) \Rightarrow A_D\,\langle \overline{X_p}\rangle : \beta_a \rightsquigarrow \langle x_{D,a}, c_{D,a}\rangle}^{\,a}, \\ \quad \overline{\forall \overline{X_i}, \overline{X_d}.\overline{\pi_{dm}}^{\,m} \Rightarrow \pi_d \rightsquigarrow \langle x_d, c_d\rangle}^{\,d}\ ]\end{array} \\ \rightsquigarrow \begin{array}{l}\mathsf{type}\ A_D\,\langle \overline{X_p}\rangle;\ \overline{\mathsf{axiom}\ c_{D,a} : \vartheta_a}^{\,a};\ \overline{\mathsf{axiom}\ c_{D,s} : \vartheta_s}^{\,s};\ \mathsf{struct}\ S_D\,\langle \overline{X_p}\rangle\,\{f : \sigma^t, \overline{x_s : \tau_s^t}, \overline{x_a : \tau_a^t}\} \\ f : \sigma'^t = \Lambda \overline{X_p}.\mathsf{fn}\ (x_D : S_D\,\langle \overline{X_p}\rangle)\{\,x_D.f\,\}; \\ \overline{x_{D,s} : \forall \overline{X_p}.\mathsf{fn}(S_D\,\langle \overline{X_p}\rangle) \to \tau_s^t = \Lambda \overline{X_p}.\mathsf{fn}\ (x_D : S_D\,\langle \overline{X_p}\rangle)\{x_D.x_s\};}^{\,s} \\ \overline{x_{D,a} : \forall \overline{X_p}.\mathsf{fn}(S_D\,\langle \overline{X_p}\rangle) \to \tau_a^t = \Lambda \overline{X_p}.\mathsf{fn}\ (x_D : S_D\,\langle \overline{X_p}\rangle)\{x_D.x_a\};}^{\,a} \\ \overline{x_d : \forall \overline{X_i}, \overline{X_d}.\forall \overline{\omega_{dm}}.\mathsf{fn}(\overline{\tau_{dm}^t}) \to \tau_d^t = \Lambda \overline{X_i}, \overline{X_d}.\Lambda \overline{c_{dm} : \omega_{dm}}.\mathsf{fn}\ (\overline{x_{dm} : \tau_{dm}^t})\,\{e_d^t\};}^{\,d} \\ \overline{\mathsf{axiom}\ c_d : \forall \overline{X_i}, \overline{X_d}.u_d^t \mapsto X_d;}^{\,d}\end{array} \end{array}$$

$$\frac{(D\,\langle \overline{X_i}\rangle\ \mathsf{where}\ \_, \mathsf{Self}_{\mathbf{U}} : \overline{D_s\,\langle \overline{u_{ns}}, A_s \mapsto \_\rangle \rightsquigarrow \_}, A : \_, \_, \_) \in \Gamma \qquad \overline{\Gamma \vdash D_s\,\langle \overline{[u_i/X_i]u_{ns}}\rangle \Uparrow \{\overline{\beta_{j_s}}\}}^{\,s}}{\Gamma \vdash D\,\langle \overline{u_i}\rangle \Uparrow \{\overline{\beta_{j_s}}, D\,\langle \overline{u_i}, A \mapsto \star\rangle\}}$$

Figure 7.5: MiniRust: well-typed object-safe traits with translation

$\boxed{\Gamma \mid \Theta \mid \Theta \vdash \pi \Rrightarrow \pi \rightsquigarrow \gamma}$

$$\frac{\Gamma \mid \Theta_1, \Theta_2 \Vdash u_1 \sim u_2 \rightsquigarrow \gamma_1 \qquad \overline{\Gamma \mid \Theta_1, \Theta_2 \Vdash u_{1i} \sim u_{2i} \rightsquigarrow \gamma_i}^{\,i} \qquad \Gamma \mid \Theta_1 \vdash_{\text{WF}} u_2 : D\,\langle \overline{u_{2i}}, A \mapsto \star \rangle}{\Gamma \mid \Theta_1 \mid \Theta_2 \vdash u_1 : D\,\langle \overline{u_{1i}}, A \mapsto \star \rangle \Rrightarrow u_2 : D\,\langle \overline{u_{2i}}, A \mapsto \star \rangle \rightsquigarrow S_D\,\langle \gamma_1, \overline{\gamma_i} \rangle}$$

$$\frac{\begin{array}{c}\Gamma \mid \Theta_1, \Theta_2 \Vdash u_1 \sim u_2 \rightsquigarrow \gamma_1 \qquad \overline{\Gamma \mid \Theta_1, \Theta_2 \Vdash u_{1i} \sim u_{2i} \rightsquigarrow \gamma_i}^{\,i} \\ \Gamma \mid \Theta_1, \Theta_2 \Vdash u_3 \sim u_4 \qquad \Gamma \mid \Theta_1 \vdash_{\text{WF}} u_2 : D\,\langle \overline{u_{2i}}, A \mapsto u_2 \rangle\end{array}}{\Gamma \mid \Theta_1 \mid \Theta_2 \vdash u_1 : D\,\langle \overline{u_{1i}}, A \mapsto u_3 \rangle \Rrightarrow u_2 : D\,\langle \overline{u_{2i}}, A \mapsto u_4 \rangle \rightsquigarrow S_D\,\langle \gamma_1, \overline{\gamma_i} \rangle}$$

$\boxed{\Gamma \vdash \pi \mapsto \beta \rightsquigarrow e^t}$

$$\text{(path1)}\frac{\begin{array}{c}(D_1\,\langle \overline{X_i} \rangle \text{ where } \_, \overline{\mathsf{Self}_{\mathbf{U}} : \beta_s \rightsquigarrow x_s}, \_, \_) \\ (D_3\,\langle \overline{u_k}, A \mapsto \_ \rangle \rightsquigarrow x) \in \overline{\{[u/\mathsf{Self}_{\mathbf{U}}][\overline{u_i/X_i}]\beta_s \rightsquigarrow x_s\}} \\ \Gamma \vdash u : D_3\,\langle \overline{u_k} \rangle \mapsto D_2\,\langle \overline{u_j} \rangle \rightsquigarrow e^t\end{array}}{\Gamma \vdash u : D_1\,\langle \overline{u_i} \rangle \mapsto D_2\,\langle \overline{u_j} \rangle \rightsquigarrow x.e^t} \qquad \text{(path2)}\frac{(D_1\,\langle \overline{X_i} \rangle \text{ where } \_, \_, \_, f, \_)}{\Gamma \vdash u : D_1\,\langle \overline{u_i} \rangle \mapsto D_1\,\langle \overline{u_i} \rangle \rightsquigarrow f}$$

$\Gamma \mid \Theta \mid \overline{\theta_d \rightsquigarrow \langle \_, c'_d \rangle} \vdash_{\text{os}} u_{\text{obj}} : D\,\langle \overline{u_i}, A \mapsto X \rangle \rightsquigarrow \langle e^t_D, \overline{\pi_s \rightsquigarrow x'_s}, \overline{\pi_a \rightsquigarrow x'_a} \rangle$
where

$u_{\text{obj}} = \exists T : D_{\text{obj}}\,\langle \overline{X_h}, \overline{A_{Dd}\,\langle T, \overline{u_{nd}} \rangle \sim X_d} \rangle$

$$\frac{(D\,\langle \overline{X_i} \rangle \text{ where } \_, \mathsf{Self}_{\mathbf{U}} : \overline{\beta_s \rightsquigarrow x_s}, A : \overline{\beta_a \rightsquigarrow x_a}, f, \text{obj-safe}) \in \Gamma}{\Gamma \mid \Theta \mid \overline{\theta_d \rightsquigarrow \langle \_, c'_d \rangle} \vdash u_{\text{obj}} : [u_{\text{obj}}/\mathsf{Self}_{\mathbf{U}}][\overline{u_i/X_i}]\beta_s \Rrightarrow \pi_s \rightsquigarrow \gamma_s}\,s$$

$$\frac{\Gamma \mid \Theta \mid \overline{\theta_d \rightsquigarrow \langle \_, c'_d \rangle} \vdash u_{\text{obj}} : [u_{\text{obj}}/\mathsf{Self}_{\mathbf{U}}][\overline{u_i/X_i}]\beta_s \Rrightarrow \pi_s \rightsquigarrow \gamma_s}{\Gamma \mid \Theta \mid \overline{\theta_d \rightsquigarrow \langle \_, c'_d \rangle} \vdash A_D\,\langle u_{\text{obj}}, \overline{u_i} \rangle : [u_{\text{obj}}/\mathsf{Self}_{\mathbf{U}}][\overline{u_i/X_i}]\beta_a \Rrightarrow \pi_a \rightsquigarrow \gamma_a}\,a$$

$(f : \forall \mathsf{Self}_{\mathbf{U}}, \overline{X_i}.(\mathsf{Self}_{\mathbf{U}} : D\,\langle \overline{X_i} \rangle) \Rightarrow \mathsf{fn}(\&\mathsf{Self}_{\mathbf{U}}, \overline{\tau_m}) \to \tau_R) \in \Gamma$

$\Gamma \mid \Theta \vdash_{\text{WF}} \forall \mathsf{Self}_{\mathbf{U}}, \overline{X_i}.(\mathsf{Self}_{\mathbf{U}} : D\,\langle \overline{X_i} \rangle) \Rightarrow \mathsf{fn}(\&\mathsf{Self}_{\mathbf{U}}, \overline{\tau_m}) \to \tau_R$
$\quad \rightsquigarrow \forall \mathsf{Self}_{\mathbf{U}}, \overline{X_i}.\mathsf{fn}(S_D\,\langle \mathsf{Self}_{\mathbf{U}}, \overline{X_i} \rangle) \Rightarrow \mathsf{fn}(\&\mathsf{Self}_{\mathbf{U}}, \overline{\tau^t_m}) \to \tau^t_R$

$\Gamma \mid \Theta, u_{\text{obj}} : D_{\text{obj}}\,\langle \overline{X_h} \rangle \vdash_{\text{WF}} u_{\text{obj}} : D\,\langle \overline{u_i} \rangle \rightsquigarrow S_D\,\langle u^t_{\text{obj}}, \overline{u^t_i} \rangle$

$(A_D\,\langle T, \overline{u'_i} \rangle \sim X) \in \{\overline{A_{Dd}\,\langle T, \overline{u_{nd}} \rangle \sim X_d}\}$

$$\frac{\Gamma' \mid \Theta' \Vdash [T/\mathsf{Self}_{\mathbf{U}}][\overline{u'_i/X_i}]\tau_R \sim [u_{\text{obj}}/\mathsf{Self}_{\mathbf{U}}][\overline{u_i/X_i}]\tau_R \rightsquigarrow \gamma}{\Gamma' \mid \Theta' \Vdash [u_{\text{obj}}/\mathsf{Self}_{\mathbf{U}}][\overline{u_i/X_i}]\tau_m \sim [T/\mathsf{Self}_{\mathbf{U}}][\overline{u'_i/X_i}]\tau_m \rightsquigarrow \gamma_m}\,m$$

$\Gamma' = \Gamma, T$

$\Theta' = \Theta, \overline{\theta_d \rightsquigarrow \langle \_, c'_d \rangle}, T : D_{\text{obj}}\,\langle \overline{X_h} \rangle, \overline{T : D_d\,\langle \overline{u_{nd}}, A \mapsto X_d \rangle \rightsquigarrow \langle \_, c_d \rangle}$

$\Gamma, T \vdash T : D_{\text{obj}}\,\langle \overline{X_h} \rangle \mapsto D\,\langle \overline{u'_i} \rangle \rightsquigarrow e^t_P$

$e^t_D = S_D\,\langle u^t_{\text{obj}}, \overline{u^t_i} \rangle\,\{$
$\quad f : \mathsf{fn}\,(x_{\text{obj}} : \&u^t_{\text{obj}}, \overline{x_m : [u^t_{\text{obj}}/\mathsf{Self}_{\mathbf{U}}][\overline{u^t_i/X_i}]\tau^t_m})\,\{$
$\qquad \mathsf{let}\,(T, x_{\text{val}}, x_{\text{dict}}, \overline{c_d}) = \mathsf{unpack}\,x_{\text{obj}}\,\mathsf{in}$
$\qquad\quad x_{\text{dict}}.e^t_P(x_{\text{val}}, \overline{x_m \blacktriangleright \gamma_m}) \blacktriangleright \gamma\},$
$\quad \overline{x_s : x'_s \blacktriangleright \mathsf{sym}\,(\gamma_s)},$
$\quad \overline{x_a : x'_a \blacktriangleright \mathsf{sym}\,(\gamma_a)}\,\}$

Figure 7.6: MiniRust: auxiliary object-safe trait relations with translation

7.6.3 Translating object-safe traits

Figure 7.5 presents the object-safe trait typing rule, (obj-trt). In addition to the RustIn items produced by non-object-safe traits in rule (trait), translation of object-safe traits contains items that correspond to the trait object constraint schemes output by the rule. The items include top-level term declarations $\overline{x_d}$ and type equality axioms $\overline{c_d}$. The terms assigned to $\overline{x_d}$ are polymorphic functions. They take as input the coercions and dictionaries corresponding to the supertrait and associated type constraints $\overline{\pi_{dm}}$, which we obtain using the auxiliary relation $\vdash_{\mathsf{os}}$. The functions' return values are dictionaries e^t_d, also obtained from the relation $\vdash_{\mathsf{os}}$, that serve as evidence that the trait's descriptor implements the traits $\overline{D_d}$ in the descriptor's supertrait hierarchy. The type equality axioms $\overline{c_d}$ are translations of the associated type instantiations in the trait object constraint schemes.

The auxiliary relation $\Gamma \mid \Theta \mid \overline{\theta_d \rightsquigarrow \langle \_, c'_d \rangle} \vdash_{\mathsf{os}} u_{\mathsf{obj}} : D\,\langle \overline{u_i}, A \mapsto X \rangle \rightsquigarrow \langle e^t_D, \overline{\pi_s \rightsquigarrow x'_s}, \overline{\pi_a \rightsquigarrow x'_a} \rangle$ is presented in figure 7.6. The relation is extended to include in its outputs the dictionary e^t_D corresponding to constraint $u_{\mathsf{obj}} : D\,\langle \overline{u_i}, A \mapsto X \rangle$. The relation also includes variable names $\overline{x'_s}, \overline{x'_a}$ for the dictionaries corresponding to the supertrait constraints $\overline{\pi_s}$ and associated type constraints $\overline{\pi_a}$. The variables are used in the construction of the trait object dictionary e^t, since in (obj-trt)'s output, the dictionary is abstracted over them.

The side conditions

$$\frac{}{\Gamma \mid \Theta \mid \overline{\theta_d \rightsquigarrow \langle \_, c'_d \rangle} \vdash u_{\mathsf{obj}} : [u_{\mathsf{obj}}/\mathsf{Self}_{\mathsf{U}}][\overline{u_i/X_i}]\beta_s \Rrightarrow \pi_s \rightsquigarrow \gamma_s}\, s$$

$$\frac{}{\Gamma \mid \Theta \mid \overline{\theta_d \rightsquigarrow \langle \_, c'_d \rangle} \vdash A_D\,\langle u_{\mathsf{obj}}, \overline{u_i} \rangle : [u_{\mathsf{obj}}/\mathsf{Self}_{\mathsf{U}}][\overline{u_i/X_i}]\beta_a \Rrightarrow \pi_a \rightsquigarrow \gamma_a}\, a$$

give us the constraints $\overline{\pi_s}$ and $\overline{\pi_a}$, which are type parameter-wise equivalent to the supertrait and associated type constraints of the trait D instantiated with types $u_{\mathsf{obj}}, \overline{u_i}$. The coercions $\overline{\gamma_s}$ and $\overline{\gamma_a}$ relate the dictionaries of the constraints from the left-hand side of the arrow $\Rrightarrow$ with the dictionaries of their equivalent constraints from the right-hand side of the arrow.

To construct the dictionary e^t_D we use the coercions from the helper relation as well as the coercions $\gamma, \overline{\gamma_m}$ that result from the type equality constraints on the trait function's argument and return types. The coercions help ensure that the the struct's fields are well-typed.

We also introduce a helper relation $\Gamma \vdash \pi \mapsto \beta \rightsquigarrow e^t$, which provides a path e^t from the dictionary of π to the trait function of the dictionary corresponding to the bound of β. The relation is defined for bounds β that are in the supertrait hierarchy of the constraint π.

In the body of e^t_D, the function f, representing D's trait function, unpacks the trait object x_{obj} and applies the function contained in the dictionary x_{dict} encapsulated in the object. We apply the coercions $\overline{\gamma_m}$ and γ to ensure that the body of f is well-typed.

The fields $\overline{x_s}$ and $\overline{x_a}$ in e^t_D correspond to D's supertrait and associated type constraint dic-

$\boxed{\vdash_P pgm : \Gamma \mid \Theta \rightsquigarrow pgm^t}$ (Well-typed programs)

$$\text{(pgm)}\ \frac{\overline{\Gamma \mid \Theta \vdash item_i : \Gamma_i \mid \Theta_i \rightsquigarrow pgm_i^t} \qquad \Gamma = \overline{\Gamma_i} \qquad \Theta = \overline{\Theta_i}}{\vdash_P \overline{item_i} : \Gamma \mid \Theta \rightsquigarrow \overline{pgm_i^t}}$$

Figure 7.7: MiniRust: well-typed programs with translation

tionaries respectively. We assign to them the term variables $\overline{x'_s}$, $\overline{x'_a}$, corresponding to translations of $\overline{\pi_s}, \overline{\pi_a}$, coerced with $\overline{\mathsf{sym}\ \gamma_s}$ and $\overline{\mathsf{sym}\ \gamma_a}$ respectively to ensure that the fields' types match the definition of the struct S_D.

7.6.4 Translating programs

The program typing and translation rule is presented in figure 7.7. The rule is straightforward. A well-typed program simply translates to a concatenation of the partial RustIn programs generated from its items.

7.7 Type preservation of translation

Before we present the theorems of type preservation, we need to be able to reason about what constitutes a well-formed environment. The well-formed environments judgment is presented in figure 7.8. Intuitively, in a well-formed environment, all of its members are well-formed. A well-formed trait information tuple in a typing environment means that the trait satisfies the side conditions as prescribed by the (trait) or (obj-trt) rule. The rules in the figure also include translations to corresponding RustIn environments.

We first show that well-typed MiniRust terms translate to well-typed RustIn terms and the types of both terms are also related by translation.

Theorem 7.1 (Translation of terms preserves well-typedness)**.** *If:*

- $\Gamma \mid \Theta \vdash e : \tau \rightsquigarrow e^t$ *and*
- $\Gamma \mid \Theta \vdash_{\text{WF}} \tau \rightsquigarrow \tau^t$ *and*
- $\langle \Gamma, \Theta \rangle \rightsquigarrow \langle \Omega^t, \Gamma^t \rangle$,

then $\Omega^t \mid \Gamma^t \mid \emptyset \vdash_T e^t : \tau^t$.

The subscript T in $\Omega^t \mid \Gamma^t \mid \emptyset \vdash_T e^t : \tau^t$ signals that we are referring to a RustIn judgment (specifically the well-typed terms judgment in figure 6.4).

We then show that well-typed MiniRust items translate to well-typed RustIn items:

$\boxed{\Gamma \mid \Theta \vdash \Gamma \rightsquigarrow \langle \Omega^t, \Gamma^t \rangle}$ (Well-formed typing environments)

$$(\Gamma\emptyset)\frac{}{\Gamma \mid \Theta \vdash \emptyset \rightsquigarrow \langle \emptyset, \emptyset \rangle} \qquad (\Gamma X)\frac{\Gamma \mid \Theta \vdash \Gamma' \rightsquigarrow \langle \Omega^t, \Gamma^t \rangle}{\Gamma \mid \Theta \vdash \Gamma', X \rightsquigarrow \langle \Omega^t, (\Gamma^t, X) \rangle}$$

$$(\Gamma\text{var})\frac{\Gamma \mid \Theta \vdash \Gamma' \rightsquigarrow \langle \Omega^t, \Gamma^t \rangle \qquad \Gamma \mid \Theta \vdash_{\text{WF}} \sigma \rightsquigarrow \sigma^t}{\Gamma \mid \Theta \vdash \Gamma', x : \sigma \rightsquigarrow \langle \Omega^t, (\Gamma^t, x : \sigma^t) \rangle}$$

$$(\Gamma\text{sct})\frac{\Gamma \mid \Theta \vdash \Gamma' \rightsquigarrow \langle \Omega^t, \Gamma^t \rangle \qquad \overline{\Gamma, \overline{X_j} \mid \Theta \vdash_{\text{WF}} \tau_i \rightsquigarrow \tau_i^t}}{\Gamma \mid \Theta \vdash \Gamma', S\,\langle \overline{X_j} \rangle\,\{\overline{x_i : \tau_i}\} \rightsquigarrow \langle (\Omega^t, S\,\langle \overline{X_j} \rangle\,\{\overline{x_i : \tau_i^t}\}), \Gamma^t \rangle}$$

$$(\Gamma\text{trt})\frac{\begin{array}{c}\dfrac{\Gamma \mid \Theta \vdash \Gamma' \rightsquigarrow \langle \Omega_1^t, \Gamma^t \rangle \qquad \overline{X_p} = \mathit{Self}, \overline{X_i} \qquad \overline{\Gamma, \overline{X_p} \mid \Theta, \mathit{Self} : D\,\langle \overline{X_i} \rangle \vdash_{\text{WF}} \mathit{Self} : \beta_s \rightsquigarrow \langle \tau_s^t, \omega_s \rangle}^{s}}{\Gamma, \overline{X_p} \mid \Theta, \mathit{Self} : D\,\langle \overline{X_i} \rangle \vdash_{\text{WF}} A_D\,\langle \overline{X_p} \rangle : \beta_a \rightsquigarrow \langle \tau_a^t, \omega_a \rangle}^{a} \qquad (f : \forall \overline{X_p}, \overline{X_k}.(\mathit{Self} : D\,\langle \overline{X_i} \rangle, \overline{\pi_l}) \Rightarrow \tau) \in \Gamma \\ \overline{(\forall \overline{X_p}.(\mathit{Self} : D\,\langle \overline{X_i} \rangle) \Rightarrow \mathit{Self} : \beta_s \rightsquigarrow \_)}^{s} \in \Theta \qquad \overline{(\forall \overline{X_p}.(\mathit{Self} : D\,\langle \overline{X_i} \rangle) \Rightarrow A_D\,\langle \overline{X_p} \rangle : \beta_a \rightsquigarrow \_)}^{a} \in \Theta \\ \Gamma, \overline{X_p} \mid \Theta, \mathit{Self} : D\,\langle \overline{X_i} \rangle \vdash_{\text{WF}} \forall \overline{X_k}.\overline{\pi_l} \Rightarrow \tau \rightsquigarrow \sigma^t \qquad \Omega^t = \Omega_1^t,\ S_D\,\langle \overline{X_p} \rangle\,\{f : \sigma^t, \overline{x_s : \tau_s^t}, \overline{x_a : \tau_a^t}\}, A_D\,\langle \overline{X_p} \rangle\end{array}}{\Gamma \mid \Theta \vdash \Gamma', (D\,\langle \overline{X_i} \rangle \text{ where } \overline{\pi_h}, \mathit{Self} : \overline{\beta_s \rightsquigarrow x_s}, A : \overline{\beta_a \rightsquigarrow x_a}, f, \mathit{obj}) \rightsquigarrow \langle \Omega^t, \Gamma^t \rangle}$$

$\boxed{\Gamma \mid \Theta \vdash \Theta \rightsquigarrow \langle \Omega^t, \Gamma^t \rangle}$ (Well-formed constraint environments)

$$(\Theta\emptyset)\frac{}{\Gamma \mid \Theta \vdash \emptyset \rightsquigarrow \langle \emptyset, \emptyset \rangle} \qquad (\Theta\theta)\frac{\Gamma \mid \Theta \vdash \Theta' \rightsquigarrow \langle \Omega^t, \Gamma^t \rangle \qquad \Gamma \mid \Theta \vdash_{\text{WF}} \theta \rightsquigarrow \langle \sigma^t, \vartheta \rangle}{\Gamma \mid \Theta \vdash \Theta', (\theta \rightsquigarrow \langle x, c \rangle) \rightsquigarrow \langle (\Omega^t, c : \vartheta), (\Gamma^t, x : \sigma^t) \rangle}$$

$\boxed{\langle \Gamma, \Theta \rangle \rightsquigarrow \langle \Omega^t, \Theta^t \rangle}$ (Well-formed environments)

$$(\text{tr-envs})\frac{\begin{array}{cc}\Gamma \mid \Theta \vdash \Gamma \rightsquigarrow \langle \Omega_1^t, \Gamma_1^t \rangle & \Gamma \mid \Theta \vdash \Theta \rightsquigarrow \langle \Omega_2^t, \Gamma_2^t \rangle \\ \Omega^t = \Omega_1^t, \Omega_2^t & \Gamma^t = \Gamma_1^t, \Gamma_2^t\end{array}}{\langle \Gamma, \Theta \rangle \rightsquigarrow \langle \Omega^t, \Theta^t \rangle}$$

Figure 7.8: MiniRust: well-formed environments with translation

Theorem 7.2 (Translation of items preserves well-typedness)**.** *Suppose* $\Gamma' \subseteq \Gamma$ *and* $\Theta' \subseteq \Theta$.
If $\Gamma \mid \Theta \vdash \mathit{item} : \Gamma' \mid \Theta' \rightsquigarrow \overline{\mathit{item}_i^t}$ *and* $\langle \Gamma, \Theta \rangle \rightsquigarrow \langle \Omega^t, \Gamma^t \rangle$
then there are some $\Omega_1^t, \Omega_2^t, \Gamma_1^t, \Gamma_2^t, \overline{\Omega_i^t}, \overline{\Gamma_i^t}$ *such that :*

- $\overline{\Gamma \mid \Theta \vdash \Gamma' \rightsquigarrow \langle \Omega_1^t, \Gamma_1^t \rangle}$,
- $\overline{\Gamma \mid \Theta \vdash \Theta' \rightsquigarrow \langle \Omega_2^t, \Omega_2^t \rangle}$,
- $\overline{\Omega^t \mid \Gamma^t \vdash_T \mathit{item}_i^t : \Omega_i^t \mid \Gamma_i^t}$,
- $\overline{\Omega_i^t} = \Omega_1^t, \Omega_2^t$ *and*
- $\overline{\Gamma_i^t} = \Gamma_1^t, \Gamma_2^t$.

Finally, we show that well-typed MiniRust programs translate to well-typed RustIn programs and that their resulting environments are related by the translation rules in figure 7.8.

Theorem 7.3 (Translation of programs preserves well-typedness)**.** *If* $\vdash_P pgm \rightsquigarrow pgm^t : \Gamma \mid \Theta$ *and then there are some* Ω^t, Γ^t *such that* $\langle \Gamma, \Theta \rangle \rightsquigarrow \langle \Omega^t, \Gamma^t \rangle$ *and* $\vdash_T pgm^t : \Omega^t \mid \Gamma^t$.

Proofs of the above theorems can be found in appendix B.

7.8 Discussion

7.8.1 Type safety of MiniRust

To prove that well-typed MiniRust programs are type-safe, we must show that their RustIn translations satisfy the requirements for type safety, as defined in theorem 6.3. One of those requirements is that the top-level items must include a function main of type $\mathsf{fn}() \to ()$, which we already enforce in MiniRust. Another requirement is that the top-level environment Ω^t produced in $\vdash_T pgm^t : \Omega^t \mid \Gamma^t$ must be consistent, as specified in definition 6.4. One necessary condition for consistency is that source programs should not have overlapping impls. We discuss impl overlap and how it may affect consistency in the next chapter.

7.8.2 Monomorphization

As mentioned in chapter 2, to prevent traits and generics-based abstractions from introducing runtime overhead in Rust, generic items are monomorphized at compile-time, i.e., the compiler performs a translation pass on the program where generic items are translated to their monomorphic versions based on types with which they are used in the program. Our formalization of the runtime semantics of MiniRust, through a translation to an internal language where dictionaries corresponding to traits are passed around at runtime, is quite different from the Rust compilation model. While our specification of the operational semantics of MiniRust programs does not reflect Rust's runtime cost model, through evidence-based translation we are able to reason more clearly about the modular qualities of traits and about type safety of programs with traits. In particular, the trait dictionary structure gives us a clear insight into what is required for a type to implement a trait. We conjecture, however, that trait dictionaries in RustIn programs can still be compiled away and that RustIn generic items can be monomorphized similarly to the monomorphization process in Rust. As such, RustIn can serve as an intermediate language between MiniRust and some other monomorphic target language.

Chapter 8

Coherence

As we discussed in the previous chapter, in languages with type-based overloading, such as Rust, the operational meaning of a term may depend on its type. It is common to define the operational semantics of such languages in terms of a type-based translation to a target language where the overloads have already been resolved, as we have done in this thesis. However, if the type system is non-deterministic, in the sense that it allows a term to have different types, then such a term may produce different translations.

Coherence is a property of a type system where the meaning of a term does not depend on the way it was type-checked [10]. Therefore, in a coherent type system, a program with two different typing derivations, once evaluated, should produce the same observable result. It is a desirable property, especially in a system focused on predictability.

The type system of MiniRust, in the absence of restrictions, is in fact incoherent. For instance, an ambiguous trait function call, where the arguments provided to the trait function do not provide enough information to determine the trait parameter instantiations used in the call, can result in not knowing which impl to use to resolve it. For example, consider the following two traits (using Rust syntax):

```
trait FromString {
    fn from_str(String) -> Self
}

trait ToString {
    fn to_str(self) -> String
}
```

Suppose that both `FromString` and `ToString` have implementations for floats `f64` and integers `i32`. Then, in the variable declaration

```
let x = to_str(from_str("5"));
```

we know that `x` has type `String` but there is no way to know which implementations of `from_str` and `to_str` are to be used.

Ambiguity problems like the above can be solved by simply requiring the programmer to provide type annotations (in Rust, `from_str` has to be called using the explicit UFCS, since it is not a method). However, if there is ambiguity in impl matching that is due to overlapping impl declarations, type annotations cannot solve the problem. Consider the following two impls of `FromString`:

```
impl<T> FromString for T { ... }
impl FromString for f64 { ... }
```

The first impl is a *blanket impl*: it implements `FromString` for all types. Thus, the following call to `from_str`:

```
let y: String = ...
let x: f64 = from_str(y);
```

is ambiguous as both implementations of `from_str` can be used to resolve it. Therefore, Rust includes rules that ensure *trait coherence*: that for every concrete instantiation of trait parameters there is at most one matching impl.

In the context of the subset of Rust formalized in MiniRust, where programs do not include external library dependencies, this simply means that we must not allow overlap in impl declarations. Specifically, for any given two impls that produce constraint schemes $\theta_1 = \forall \overline{X_i}.\overline{\pi_j} \Rightarrow \pi_1$ and $\theta_2 = \forall \overline{X_k}.\overline{\pi_l} \Rightarrow \pi_2$ we must ensure that there is no type substitution $\phi = [\overline{u/X}]$ that maps type variables to types such that $\phi(\pi_1) = \phi(\pi_2)$ and $\Gamma \mid \Theta \Vdash \phi(\pi)$ for all $\pi \in \{\overline{\pi_j}, \overline{\pi_l}\}$, where Γ and Θ are the top-level typing and constraint environments respectively.

However, in the presence of library dependencies, traits and impls from external *crates*—Rust compilation units, which can be libraries—come into scope. If we want library writers to safely extend their crates without introducing incoherence in their user crates, while allowing users to implement external traits, more care must be taken to ensure trait coherence. In this chapter, we consider the current rules that are meant to ensure trait coherence in Rust in the presence of external crate dependencies. We extend the type system of NanoRust, presented in chapter 3, with crates and we adapt the Rust trait coherence rules to our model. We then show that the rules satisfy the guarantees that they are set to satisfy. Finally, we briefly discuss the pieces needed to extend our theorems and proofs to MiniRust and we discuss the role of trait coherence in proving type safety of MiniRust. Note that in our formal development in this chapter we disregard supertrait constraint schemes generated by trait declarations. We discuss their implications on trait coherence in section 8.4.

8.1 Crates and trait coherence

Crates are the compilation units in Rust. A crate is usually either a *library crate* or an *executable crate* that is compiled to a binary. The programmer can declare a dependency on an external crate by declaring

```
extern crate my_lib;
```

at the top of the program, where `my_lib` is the name of the crate used.

To access an item of an external crate, we specify the *path* to the item that includes the crate name. For example, we write

```
let x = my_lib::lib_function();
```

to call the function `lib_function` declared in the `my_lib` crate.

When we declare a dependency on an external crate, all traits and impls from that crate come into scope. This means that, even though the imported crate does not contain overlapping impls, the impls declared in that crate could overlap with impls from other crates that are in scope. Suppose the trait `ToString` is declared in the standard library crate but it doesn't have any impls. Then suppose that we have two library crates `lib1` and `lib2`, both dependent on the standard library:

`lib1`	`lib2`
`fn lib1_function() { ... }` `impl ToString for i32 { ... }`	`fn lib2_function() { ... }` `impl ToString for i32 { ... }`

If we want to use both `lib1_function` and `lib2_function` in our code, we have to import both `lib1` and `lib2` and thus we would have two conflicting implementations of `ToString` for `i32`.

As such, Rust enforces trait coherence rules, in the form of an overlap check and an *orphan rule*. The orphan rule effectively restricts the impls that can be declared in a crate. The general idea behind the rule is that an *orphan impl*—an impl that implements an external trait—can only be implemented for some local type (a type declared in the impl's local crate). The goal of the orphan rule, along with the overlap check, is to make sure that crate dependencies can be added safely, without causing impl overlap. Moreover, the trait coherence rules are designed in a way that is meant to give library writers the freedom to add non-blanket impls without breaking any *downstream crates*—crates that depend on the library. The trait coherence rules in Rust, including the orphan rule, are documented in [2].

8.2 NanoRust with crates

We now introduce crates to the type system of NanoRust. $K \in \textsc{Crate}$ ranges over crate names. We introduce a new *crate environment* Π defined as follows:

$$\Pi ::= \Pi, K : \langle \{\overline{K_i}\}, \Gamma, \Theta \rangle \mid \emptyset$$

The tuple $\langle \{\overline{K_i}\}, \Gamma, \Theta \rangle$ represents the contents of a crate. Specifically, it includes a set of crate identifiers $\overline{K_i}$ that are imported by the crate, and it includes local environments Γ, Θ representing the impls and typings of the items declared in the crate. The crate environment Π is then a set of bindings of crate identifiers to their tuples.

To facilitate reasoning about typing and constraint environments that span multiple crates we will use the following metafunctions:

$$\begin{aligned}
\mathbf{tenv}(\Pi, K) &= \overline{\mathbf{tenv}(\Pi, K_i)}^{\,i}, \Gamma && \text{where } \Pi(K) = \langle \{\overline{K_i}\}, \Gamma, \Theta \rangle \\
\mathbf{cenv}(\Pi, K) &= \overline{\mathbf{cenv}(\Pi, K_i)}^{\,i}, \Theta && \text{where } \Pi(K) = \langle \{\overline{K_i}\}, \Gamma, \Theta \rangle \\
\mathbf{ltenv}(\Pi, K) &= \Gamma && \text{where } \Pi(K) = \langle \{\overline{K_i}\}, \Gamma, \Theta \rangle \\
\mathbf{lcenv}(\Pi, K) &= \Theta && \text{where } \Pi(K) = \langle \{\overline{K_i}\}, \Gamma, \Theta \rangle
\end{aligned}$$

The functions $\mathbf{tenv}(\Pi, K)$ and $\mathbf{cenv}(\Pi, K)$ respectively output the typing and constraint environments that are in scope in crate K, including the environments of K's ancestor crates. The functions $\mathbf{ltenv}(\Pi, K)$ and $\mathbf{lcenv}(\Pi, K)$, on the other hand, respectively output the typing and constraint environments *local* to the trait K—environments that describe items declared directly in the crate K.

Note that the metafunctions $\mathbf{tenv}(\Pi, K)$ and $\mathbf{cenv}(\Pi, K)$ are recursive. To ensure that they are well-founded or terminating, we restrict Π to not have any circular crate dependencies. We will maintain this restriction on crate environments throughout this chapter.

To help us reason about crate dependencies, we introduce a crate dependency relation $\Pi \vdash K_1 \sqsubseteq K_2$, which we read as "crate K_1 is dependent on crate K_2 under crate environment Π". We define the relation formally as follows:

$$\frac{(K_1 : \langle \{\overline{K}, K_2\}, \Gamma, \Theta \rangle) \in \Pi}{\Pi \vdash K_1 \sqsubseteq K_2} \qquad \frac{K \in dom(\Pi)}{\Pi \vdash K \sqsubseteq K} \qquad \frac{\Pi \vdash K_1 \sqsubseteq K_3 \quad \Pi \vdash K_3 \sqsubseteq K_2}{\Pi \vdash K_1 \sqsubseteq K_2}$$

The restriction on circular crate dependencies can then be stated formally as follows: "if $\Pi \vdash K_1 \sqsubseteq K_2$ and $\Pi \vdash K_2 \sqsubseteq K_1$ then $K_1 = K_2$".

We can extend the syntax of programs in NanoRust to include crate importing statements:

$$pgm ::= \overline{\mathsf{extern}\ K;}\ \overline{item}$$

$$\frac{S\,\langle \_ \rangle\,\{\overline{x:\tau}\} \in \Gamma}{\Gamma \vdash S} \qquad \frac{(D\,\langle \_ \rangle \text{ where } \_,\_,\_) \in \Gamma}{\Gamma \vdash D}$$

Figure 8.1: Helper relations

A program, which is also a crate, then consists of *extern* statements, which declare dependencies on some external crates $\overline{K}$, and a collection of items. To reflect the change in the program syntax, we adapt the NanoRust program typing relation $\vdash pgm : \Gamma \mid \Theta$ to use the crate environment Π. We can read the new relation, $\Pi \vdash K \triangleright pgm : \Pi'$, as "under crate environment Π, the crate K corresponding to program pgm produces a crate environment Π'". We use a single rule to define the relation:

$$\text{(pgm)}\frac{\overline{\Gamma \mid \Theta \vdash item_i : \Gamma_i \mid \Theta_i}^{\,i} \qquad \Gamma = \overline{\mathbf{tenv}\,(\Pi, K_j)}^{\,j}, \overline{\Gamma_i} \qquad \Theta = \overline{\mathbf{cenv}\,(\Pi, K_j)}^{\,j}, \overline{\Theta_i}}{\Pi \vdash K \triangleright \overline{\text{extern } K_j}; \overline{item_i} : [K : \langle \{\overline{K_j}\}, \overline{\Gamma_i}, \overline{\Theta_i} \rangle]}$$

The rule is similar to the original NanoRust program typing rule in figure 3.4. The typing and constraint environments Γ, Θ, used in the item typing premises, include the environments from the imported crates $\overline{K_j}$. The output of the (pgm) rule is a binding of the current crate identifier K to its tuple, which contains the identifiers of the imported crates and the local environments corresponding to the items in K.

Our extension to NanoRust does not include paths that qualify references to items from external crates (e.g. `crate_name::function()`). We simply assume that all items in all crates in Π have unique identifiers.

8.3 Trait coherence in NanoRust with crates

The mechanism that ensures trait coherence in Rust consists of two parts: the orphan rule and the overlap check. Roughly speaking, the orphan rule ensures that an impl doesn't implement a non-local trait for a non-local type.

We first define a *local type* as a type of a struct declared in the local crate or a reference to a local type. We present the definition in the relation $\Pi \vdash_{\text{LT}} K \triangleright \tau$, which we read as "under crate environment Π, the type τ is local to crate K". The relation is defined as follows:

$$\text{(l-struct)}\frac{\mathbf{ltenv}\,(\Pi, K) \vdash S}{\Pi \vdash_{\text{LT}} K \triangleright S\,\langle \overline{\tau_i} \rangle} \qquad \text{(l-ref)}\frac{\Pi \vdash_{\text{LT}} K \triangleright \tau}{\Pi \vdash_{\text{LT}} K \triangleright \&\tau}$$

The rule (l-struct) uses in its premise the helper relation $\Gamma \vdash S$, defined in figure 8.1, which proves that the struct S is in scope of the typing environment Γ.

We can now state the orphan rule for NanoRust with crates:

Definition 8.1 (Orphan rule)**.** *Under crate environment* Π*, a constraint* $\tau_0 : D\,\langle\tau_1, \ldots, \tau_n\rangle$ *obeys the orphan rule with respect to crate* K *if either:*

- *trait* D *is local to crate* K*, or*
- *there is a type* τ_L *in* $[\tau_0, \tau_1, \ldots, \tau_n]$ *such that* $\Pi \vdash_{\text{LT}} K \rhd \tau_L$ *and, for all types* $\overline{\tau_i}$ *that precede* τ_L *in the above sequence,* $FV(\overline{\tau_i}) = \emptyset$*.*

We can restate the above definition as a relation $\Pi \vdash_{\text{OR}} K \rhd \pi$ defined below:

$$\text{(orph1)}\frac{\textbf{ltenv}\,(\Pi, K) \vdash D}{\Pi \vdash_{\text{OR}} K \rhd \tau : D\,\langle\overline{\tau_i}\rangle} \qquad \text{(orph2)}\frac{\Pi \vdash_{\text{LT}} K \rhd \tau}{\Pi \vdash_{\text{OR}} K \rhd \tau : D\,\langle\overline{\tau_i}\rangle}$$

$$\text{(orph3)}\frac{\Pi \vdash_{\text{LT}} K \rhd \tau \qquad FV(\tau_0, \overline{\tau_i}) = \emptyset}{\Pi \vdash_{\text{OR}} K \rhd \tau_0 : D\,\langle\overline{\tau_i}, \tau, \overline{\tau_j}\rangle}$$

The side condition in rule (orph1) uses a helper relation, defined in figure 8.1, that ensures that the trait D is in scope of the local typing environment of crate K.

We say that a constraint scheme $\forall\overline{T_i}.\overline{\pi_i} \Rightarrow \pi$ obeys the orphan rule with respect to some crate K if and only if its end constraint π obeys the orphan rule with respect to K. It is also worth noting that we are not keeping track of free type variables in scope—we simply don't need to.

The other component of the trait coherence system in Rust is the overlap check, which ensures that two impls don't overlap. In Rust, whenever an impl is declared in some crate K, the overlap check verifies that the impl doesn't overlap with any other impls that are in scope. When checking for overlap, if there is a trait parameter instantiation that unifies the two impls, we must verify the constraints in both impls' where-clauses to ensure that at least one of them is not satisfied. The constraint satisfaction check used to do so is strictly more permissive than the one used for resolving trait method calls (whose corresponding constraint entailment relation in NanoRust is presented in figure 3.2). In particular, we assume that constraints that do not satisfy the orphan rule for the current crate are satisfied, because the crate in which the constraint satisfies the orphan rule, could add a corresponding impl in the future. Similarly, *blanket constraints*—which are, roughly speaking, trait constraints where at least one trait parameter is a type variable—are considered satisfied as well. Consider the following pair of impls:

```
impl<T> MyTrait for T where T: Foo { ... }
impl<T> MyTrait for T where T: Bar { ... }
```

To conclude that the above two impls don't overlap, we would have to prove that the traits `Foo` and `Bar` are mutually exclusive—that there is no type τ such that both τ : `Foo` and τ : `Bar` are satisfied. There is no way to guarantee that no external crate will implement both `Foo` and `Bar` for the same type, so the above two impls are considered overlapping.

We formalize constraint satisfaction in the context of an overlap check in a new crate-aware constraint entailment relation. Before we present the new constraint entailment rules, we introduce a new syntactic set $B \in \text{BLANKETTYPE}$ defined as follows:

$$B ::= T \mid \&B$$

A *blanket type* is thus a type variable or a reference to a blanket type. We then define the new crate-aware constraint entailment relation $\Pi \mid \Theta \Vdash K \rhd \pi$ below:

$$\text{(cons)}\frac{\forall \overline{T_i}.\overline{\pi_j} \Rightarrow \pi \in \Theta \qquad \overline{\Pi \mid \Theta \Vdash K \rhd [\overline{\tau_i/T_i}]\pi_j}^{\,j}}{\Pi \mid \Theta \Vdash K \rhd [\overline{\tau_i/T_i}]\pi} \qquad \text{(blanket1)}\frac{}{\Pi \mid \Theta \Vdash K \rhd B : D\,\langle \overline{\tau} \rangle}$$

$$\text{(blanket2)}\frac{}{\Pi \mid \Theta \Vdash K \rhd \tau : D\,\langle \overline{\tau_i}, B, \overline{\tau_j} \rangle} \qquad \text{(orphan)}\frac{\Pi \nvdash_{\text{OR}} K \rhd \pi}{\Pi \mid \Theta \Vdash K \rhd \pi}$$

The relation is parameterized by the local crate K. In rule (orphan), $\nvdash_{\text{OR}}$ denotes the complement of $\vdash_{\text{OR}}$. Note that the set of constraints entailed in this relation, with respect to some constraint environment Θ, is a strict superset of the set of constraints entailed in the NanoRust constraint entailment relation from figure 3.2:

Theorem 8.1. *If* $\Gamma \mid \Theta \Vdash \pi$ *then* $\Pi \mid \Theta \Vdash K \rhd \pi$ *for any crate environment* Π *and crate* K.

Proof. Straightforward induction on derivations $\Gamma \mid \Theta \Vdash \pi$. □

We can now formally state the definition of non-overlapping impls:

Definition 8.2 (Non-overlapping impls)**.**
Given two impls with corresponding constraint schemes:
$\theta_1 = \forall \overline{T_i}.\overline{\pi_j} \Rightarrow \pi_1 \in \mathbf{lcenv}(\Pi, K_1)$ *and*
$\theta_2 = \forall \overline{T_k}.\overline{\pi_l} \Rightarrow \pi_2 \in \mathbf{lcenv}(\Pi, K_2)$,
we say that θ_1 *and* θ_2 *don't overlap under* Π, Θ *iff for each type substitution* $\phi = [\overline{\tau/T}]$ *such that* $\phi(\pi_1) = \phi(\pi_2)$*, there is a constraint* $\pi \in \{\overline{\pi_j}, \overline{\pi_l}\}$ *such that* $\Pi \mid \Theta \nVdash K \rhd \phi(\pi)$ *where*

$$K = \begin{cases} K_1 & \text{if } \Pi \vdash K_1 \sqsubseteq K_2 \\ K_2 & \text{if } \Pi \vdash K_2 \sqsubseteq K_1 \\ \text{any } K \in dom(\Pi) & \text{otherwise} \end{cases}$$

and $\nVdash$ *is the complement of* $\Vdash$.

The above definition of non-overlapping impls aims to capture what the overlap check in Rust is doing. Of particular note is that the overlap check in Rust only considers pairs of impls declared in crates that are related to one another: the overlap check compares an impl in crate K with impls that are in crates that K depends on. Our definition, however, also lets us reason about impls declared in unrelated crates (crates that do not depend on

one another). In such cases, there is no clear 'local' crate K to consider when invoking the constraint entailment relation $\Pi \mid \Theta \Vdash K \rhd \pi$. It turns out, as we will show in the proof of theorem 8.2, that we do not need to decide which K to use as if impls represented by $\forall \overline{T_i}.\overline{\pi_j} \Rightarrow \pi_1$ and $\forall \overline{T_k}.\overline{\pi_l} \Rightarrow \pi_2$ come from unrelated crates then there is no substitution ϕ such that $\phi(\pi_1) = \phi(\pi_2)$.

Finally, we define a *consistent crate* below.

Definition 8.3 (Consistent crates)**.** *Suppose* $(K : \langle \{\overline{K_i}\}, \Gamma_K, \Theta_K \rangle) \in \Pi$.
Let $\Theta = \mathbf{cenv}(\Pi, K)$ *and* $\Gamma = \mathbf{tenv}(\Pi, K)$*. Crate* K *is consistent in the crate environment* Π *iff*

1. *each crate* $K' \in \overline{K_i}$ *is consistent in* Π*,*
2. $\langle \Gamma, \Theta \rangle$ *are well-formed,*
3. *every impl in* Θ_K *obeys the orphan rule with respect to crate* K*,*
   i.e., if $(\forall \overline{T}.\overline{\pi} \Rightarrow \pi) \in \Theta_K$ *then* $\Pi \vdash_{\mathrm{OR}} K \rhd \pi$*, and*
4. *for all pairs of impl constraint schemes in* Θ*, there is no overlap under* Π, Θ *according to definition 8.2.*

By saying that $\langle \Gamma, \Theta \rangle$ are well-formed, we mean that each member of the environments is well-formed with respect to Γ (similarly to the well-formed environments judgment in MiniRust presented in chapter 7). An important insight from the above definition is that there are no overlapping impls in the scope of a consistent crate.

One purpose of the trait coherence rules is to ensure that importing one or more crate will not cause overlapping impls. The following theorem formalizes that guarantee:

Theorem 8.2 (Trait coherence)**.** *Suppose:*

- $(K : \langle \{\overline{K_r}\}, \Gamma_K, \Theta_K \rangle) \in \Pi$*,*
- *each crate* $K' \in \overline{K_r}$ *is consistent in* Π*,*
- $\langle \mathbf{tenv}(\Pi, K), \mathbf{cenv}(\Pi, K) \rangle$ *well-formed,*
- *every impl in* Θ_K *obeys the orphan rule w.r.t. crate* K*, and*
- *for every pair of impl constraints schemes* $\theta_1 \in \Theta_K$ *and* $\theta_2 \in \mathbf{cenv}(\Pi, K)$*,*
  θ_1 *and* θ_2 *don't overlap under* Π *and* $\mathbf{cenv}(\Pi, K)$*.*

Then K *is consistent in* Π*.*

The above theorem tells us that, given a crate K that depends on crates $\overline{K_r}$, if each crate in $\overline{K_r}$ is consistent and if the impls in K don't overlap with any impls in $\overline{K_r}$, then K is consistent. Specifically, the theorem guarantees that linking to consistent crates from a consistent crate maintains trait coherence.

Another purpose of the trait coherence rules is to give some flexibility for library crates to add impls without introducing overlap in their user crates. The only additional restriction on the new impls (in addition to not causing overlap in the scope of K and satisfying the orphan rule) is that they cannot be blanket impls, that is, none of their trait parameter instantiations can be blanket types. The following theorem states that guarantee formally:

Theorem 8.3 (Crate extensibility)**.** *Let K be a consistent crate under Π where $\Pi = \Pi'' \cup \{K : \langle\{\overline{K_a}\}, \Gamma_K, \Theta_K\rangle\}$. Then let $\Pi' = [K'/K]\Pi'' \cup \{K' : \langle\{\overline{K_a}\}, \Gamma_K, \Theta_K \cup \{\theta_1\}\rangle\}$ where K' is consistent in Π' and $\theta_1 = \forall \overline{T_k}.\overline{\pi_j} \Rightarrow \tau_1 : D\ \langle\overline{\tau_{1n}}\rangle$ with $\tau \notin$ BLANKETTYPE for each $\tau \in \{\tau_1, \overline{\tau_{1n}}\}$. Then, for each crate K_1 such that $\Pi' \vdash K_1 \sqsubseteq K'$, if K_1 is consistent under Π, then it is also consistent under Π'.*

The proofs of both of the above theorems can be found in the appendix.

8.4 Coherence and type safety of MiniRust

Our development of the trait coherence system in this chapter is based on a fairly simple type system. In particular, the system does not include associated types, trait objects or propagated trait constraints (supertrait constraints and associated type constraints). To ensure that the properties proven in this chapter hold in MiniRust, the work should be extended to account for those additional features. In this section we present a brief discussion of what such extensions might entail.

Associated types

One way to handle associated types in an impl header is to treat them as type variables. An associated type, whose type parameters do not provide sufficient information so that it can be matched to a specific impl, could potentially be normalized to any type, local or otherwise. That is also the approach taken in the current Rust implementation. However, one could consider ways to relax the coherence rules in relation to associated types. For instance, it would seem reasonable to allow the following two impls to coexist:

```
impl<T> IteratorWrapper for T where T: Iterator<Item = i32> { ... }
impl<T> IteratorWrapper for T where T: Iterator<Item = char> { ... }
```

The current coherence rules do not allow these impls to coexist as they are both constrained by blanket impls, so they are deemed to overlap. However, as long as `Iterator` does not

have overlapping impls, then there is no type τ such that both τ : `Iterator<Item = i32>` and τ : `Iterator<Item = char>` are true.

Also, the formalization would need to account for type equalities introduced in MicroRust. For example, in determining whether two constraints π_1, π_2 are unifiable, i.e., that there exists a substitution ϕ such that $\phi(\pi_1) = \phi(\pi_2)$, we would rather have to use type equality of the two constraints. Concretely, we would state that the two constraints π_1 and π_2 are unifiable if and only if there is some substitution ϕ such that $\phi(\pi_1)$ is type parameter-wise equivalent to $\phi(\pi_2)$.

Trait objects

Adding support for trait objects should be relatively straightforward. A trait descriptor $\exists T.D\,\langle \overline{u}, \overline{u_{1j} \sim u_{2j}} \rangle$ would simply be considered a local type in the crate in which the trait D is declared (as is the case in Rust [2]). The trait object constraint schemes generated by the object-safe trait declarations should also be included in the overlap check to ensure that the programmer doesn't implement an object-safe trait for its trait descriptor type manually.

Coherence in MiniRust

So far, in our presentation in this chapter, we have assumed that the top-level environment Θ only contains constraint schemes generated by impl declarations. We have effectively ignored propagated constraint schemes—which represent supertrait constraints and associated type constraints—generated by trait declarations. Clearly, if we take such constraints into consideration, we get overlapping constraints. For example, consider the following traits and impls:

```
trait Eq { ... }
trait Ord: Eq { ... }

impl Eq for i32 { ... }
impl Ord for i32 { ... }
```

We have two traits: `Eq` and `Ord`, where `Eq` is a supertrait of `Ord`. We also have implementations of both traits for the type `i32`. Typing the above items in NanoRust would populate the constraint environment Θ with the following constraint schemes:

- $\forall T.T$: `Ord` $\Rightarrow T$: `Eq` from the trait declaration of `Ord`,
- `i32` : `Eq` from the impl of `Eq` for `i32`, and
- `i32` : `Ord` from the impl of `Ord` for `i32`.

It is clear that the constraint scheme $\forall T.T : \texttt{Ord} \Rightarrow T : \texttt{Eq}$ overlaps with $\texttt{i32} : \texttt{Eq}$ when T is instantiated with `i32`. When we consider the dictionary translation of those constraints to RustIn, we realize that the supertrait constraint scheme $\forall T.T : \texttt{Ord} \Rightarrow T : \texttt{Eq}$ is a function that takes a dictionary of `Ord` and returns the dictionary of `Eq` contained within. The dictionary of `Eq` is however the same as the dictionary corresponding to the constraint $\texttt{i32} : \texttt{Eq}$. Therefore, even though there are two ways to solve the constraint $\texttt{i32} : \texttt{Eq}$, both ways result in the same runtime semantics.

Thus, to prove trait coherence in MiniRust in the presence of supertrait and associated type constraints, we would have to show that different translations of any given constraint π produce the same result at runtime. In general, to prove type system coherence, we would prove that all possible translations of a well-typed MiniRust term e to RustIn produce the same result at runtime. Note, however, that in the presence of coercions, it is not sufficient to show that both translations evaluate to the same value. For example, given a constraint translation environment Θ that contains the entry $\texttt{i32} : \texttt{MyTrait}\langle A \mapsto \texttt{bool}\rangle \rightsquigarrow \langle x, c\rangle$, even a very simple term like `true` can be translated to `true` or $\texttt{true} \blacktriangleright \texttt{sym}\ c \circ c$. The only difference between the two resulting terms is the coercion $\texttt{sym}\ c \circ c$ in the second term, however, as we established in the presentation of RustIn in chapter 6, coercions do not have any computation associated with them. Therefore, to assess equivalence of runtime semantics of two terms, we would have to 'erase' coercions from their resulting values.

Type safety of MiniRust

Recall that the proof of type safety of RustIn in chapter 6 relies on the consistency of top-level type equality axioms—meaning that no nonsensical type equalities such as $\texttt{i32} \sim \texttt{bool}$ can be derived from them. We believe that ensuring trait coherence in a MiniRust program would be sufficient to conclude that the equality axioms, resulting from the translation of the program, are *used* consistently in the resulting RustIn program, i.e., even if the axioms are inconsistent by themselves, their corresponding coercion constants are only instantiated in a way that does not introduce inconsistency (so we would never actually use an equality like $\texttt{i32} \sim \texttt{bool}$ in the program). We motivate our intuition by the fact that associated type instantiations, which are the source of RustIn type equality axioms, come from impl declarations, which we can prove beforehand to be non-overlapping. To complete the proof of type safety of MiniRust we would need to prove that claim.

For instance, consider the two non-overlapping impls below:

```
impl Foo for bool { type A = char; }
impl<T> Foo for T where MyStruct<T>: Bar { type A = i32; }
```

Their translation produces the following type equality axioms in RustIn:

$$\mathsf{axiom}\ c_1 : A\,\langle \mathtt{bool} \rangle \mapsto \mathsf{char};$$
$$\mathsf{axiom}\ c_2 : \forall T.A\,\langle T \rangle \mapsto \mathtt{i32};$$

we would need to show that, in the RustIn program, the coercion constant c_2 is never instantiated with `bool` and that this is sufficient to maintain type safety in RustIn. We leave such development for future work.

Chapter 9

Limitations & future work

In this chapter we discuss limitations of the work presented in this thesis and possible future work.

9.1 Limitations

9.1.1 Trait features

As discussed in sections 3.2.1 and 4.1.1, each MiniRust trait has exactly one trait function and one associated type, while Rust allows an arbitrary number of trait methods and associated types. We also do not allow trait constraints to include instantiations of supertrait associated types (as mentioned in section 4.1.2). As discussed in section 5.1.2, in a MiniRust trait parameterized by $\mathsf{Self}_{\mathsf{U}}$, a trait function cannot constrain its `Self` type variable to be an s-type (or `Sized`). Compared to Rust, this restricts the set of traits that MiniRust accepts as object-safe. In fact, Rust allows object-safe traits to include methods whose receiver is `self` (and not a pointer to `self`). Such methods are implicitly qualified by the constraint `Self: Sized` and they cannot be called on trait types, since trait types (descriptors in MiniRust) are dynamically sized and don't satisfy the constraint (the constraint `Self: Sized` may also be explicitly added by the programmer to a trait method).

9.1.2 Type safety

There is more work to be done to fully ensure type safety of MiniRust. Specifically, we need to show that well-typed MiniRust programs translate to RustIn programs that have consistent top-level type equality axioms, as we discussed in section 8.4. Moreover, our type safety proof relies on a translation to a hypothetical internal language, where trait constraints are translated to dictionaries, while in the Rust compilation model, trait constraints are compiled away through monomorphization. We believe that MiniRust programs translated to RustIn

can still be monomorphized while maintaining their runtime semantics, as the dictionaries can be optimized away [21]. As future work we would formally verify this claim.

9.1.3 Constraint entailment

Furthermore, constraints are resolved differently in Rust than in MiniRust. For instance, given an impl:

```
impl<T> MyTrait for T where T: MyTrait { ... }
```

if we try to solve the constraint `i32: MyTrait` in MiniRust, we would end up with an infinite derivation tree and type-checking would not terminate. On the other hand, Rust accepts this constraint (and any other constraint `T: MyTrait` for any type `T`) because the compiler can determine that the constraint `i32: MyTrait` is "self-supporting". A *coinductive* definition of the constraint entailment judgment would permit such cyclic derivation trees, which can prove satisfaction of the constraint. As future work we can explore the implications of resolving constraints coinductively on the semantics of MiniRust and on the metatheory presented in this thesis.

9.2 Towards an implementation of MiniRust

The type system of MiniRust presented in this thesis is a formal declaration of what constitutes a well-typed term or program. As we've discussed in the preceding chapter, it may allow multiple ways to type a program: there may be more than one valid type for a given term and there may be multiple typing derivations for a particular type assignment. The typing rules also involve a fair amount of guesswork, where types and type parameter instantiations have to be 'guessed' at certain points.

To implement the type system, we need an algorithm that provides a decidable and complete procedure for typing MiniRust programs. Such an algorithmic specification of the type system must be able to infer the correct types in the case of missing type annotations. For instance, we might adapt the type inference algorithm developed by Jones [20], which extends Algorithm W—a unification-based type inference procedure for the Hindley-Milner type system [14]—with predicates. To infer types of terms, the algorithm generates and solves type equality constraints involving new type variables that stand in for unknown types. Chakravarty et al. [12] extend the algorithm to incorporate type equality constraints arising from associated type synonyms in Haskell. Additionally, we can incorporate bidirectional type-checking to make use of the type annotations in programs and combine type-checking with inference [30]. In bidirectional type systems, typing rules are stratified into checking and synthesizing rules: checking rules verify that a term can be given a particular type and

synthesizing rules infer the type of a term. As such they provide a direction that a type-checker can take in typing a program. Bidirectional type systems have also been shown to be easy to implement and provide better error messages than purely synthesis-based algorithms [26].

An algorithmic specification of the type system should ideally be sound with respect to the declarative system, meaning that the type assignments determined by the algorithm should also correspond to valid typings in the declarative type system. Another desirable property of a type-checking or inference algorithm is its completeness with respect to the type system, meaning that if a term has a valid type under the type system, then the algorithm infers it. However, this property is more difficult to achieve in cases where the type system allows different types for a term. If the type system is coherent, it might be possible to set some restrictions on source programs (e.g. require type annotations on certain terms), for which a complete type-checking algorithm would be possible.[1]

A sound, complete and decidable algorithm can thus be used to implement the type system. A major advantage of an implementation is that, while the type system of MiniRust and the related formalization presented in this thesis help us reason about well-typed programs, an implementation of the type system enables us to empirically investigate whether the type system accepts the programs that the language designers intended to accept.

9.3 Possible extensions

There are a number of extensions that we can include in MiniRust to bring it closer to the actual Rust language.

More trait features

Our presentation of traits does not cover the entire feature set of Rust traits. In particular, we do not model *default trait methods* and *associated constants*.

A default trait method provides an implementation of a trait method inside a trait declaration. Impls of the trait can use that default implementation unless they explicitly override it. To add support for default methods to our formalization, we can simply copy the default method's implementation from the trait declaration to an impl that does not override it.

Associated constants should be straightforward to include as well: they would simply involve including a typed variable declaration in the trait declaration, which would be assigned to some value in an impl.

[1] Completeness of a type inference algorithm may also mean that the algorithm infers a *principal type* for each term: a type that generalizes all possible types that a term can be given under the declaration of the type system [14, 20].

Lifetimes

The subset of Rust examined in this thesis does not include any of the type system features that help ensure memory safety. For instance, we do not have mutability qualifiers for variables and references, and we do not adapt the ownership model. Our model also does not include *lifetimes*, based on research in region-based memory-management [19, 39]. Lifetimes are of particular interest in the study of traits because in Rust, traits and impls can also be parameterized by lifetimes.

In Rust, every piece of data on the stack has a lifetime. Also, the type of a reference to a value is parameterized by that value's lifetime. Consider the following expression:

```
y = &x;
```

Suppose that `x` has lifetime `'a` and `y` has lifetime `'b`. For `y` to be well-typed, the lifetime `'b` must be at least as long as `'a`. If `x` is an `i32`, then the type of `y` is parameterized by its lifetime: `&'a i32`.

In Rust, all items that can be generic—parameterized over types—can also be parameterized over lifetimes. For example, the following struct is parameterized by the lifetime `'a` of its enclosed reference:

```
struct IntPointer<'a> {
    x: &'a i32
}
```

We can then have impls parameterized by lifetimes:

```
impl<'a> Eq for IntPointer<'a> {
    fn eq(&self, &IntPointer<'a>) { ... }
}
```

To add lifetimes to MiniRust, we can extend the meta-variable X, which ranges over type variables, to also range over lifetime variables R, as done in the work on Cyclone [19]. In that sense, lifetimes are treated similarly to types. Rust also includes *lifetime bounds* in where-clauses. They take the form `'a: 'b`, which we can read as "the lifetime `'a` is at least as long as the lifetime `'b`". This gives rise to a subtyping relation on types. For example, if `'a` outlives `'b` then we can say that `&'a i32` is a subtype of `&'b i32`.

We expect interesting questions to arise from the formalization of the lifetime system, especially with regards to associated types, trait objects and *higher-rank trait bounds* [3]—trait bounds universally quantified over lifetimes. Moreover, we can explore how our development of traits can be combined with other existing work on the Rust type system focusing on the memory safety features [31].

Closures

A closure in Rust is an anonymous function that captures variables from its surrounding environment. Closures are implemented in the Rust compiler as structs that contain the variables from the enclosing scope. Each such closure implements one or more traits from the `Fn` family of traits: `Fn`, `FnMut` or `FnOnce`, depending on whether a closure has mutable access to its environment and if it can be called more than once. Each closure has a *singleton* type: a type that classifies only a single term. The name of the type of a closure is not available to the programmer—the only thing we know about such a type is that it implements some `Fn*` trait(s). As such, generics and traits are a fundamental part of programming with closures.

Consider the following example:

```
fn apply<T, F>(x: T, f: F) -> T
    where F: Fn(T) -> T {
    f(x)
}
```

In the above example, `F` is a type variable representing the type of the closure `F`. The trait bound `Fn(T) -> T` tells us that the closure takes a value of type `T` and returns a value of type `T`. In the body of the function, the call `f(x)` is actually syntactic sugar for a trait method call, `f.call(x)`. Explicitly named functions, which we have seen throughout the thesis, also implement the `Fn` trait and can be used in place of closures in a generic function.

As closures are implemented using traits, they can also take advantage of dynamic dispatch through trait objects. In particular, trait objects allow functions to return closures:

```
fn add_closure(x: i32) -> Box<Fn(i32) -> i32> {
    Box::new(move |y| x + y)
}
```

In the above example, the function `add_closure` takes an argument `x` and returns a closure that takes an argument `y` and returns the sum of `x` and `y`. The return value of `add_closure` is a trait object (`Box` is a heap pointer in Rust and can also be used to create trait objects, whose enclosed values are stored on the heap). The keyword `move` in the body of the function states that the closure `|y| x + y` *moves* the variables from its surrounding environment into the closure, instead of borrowing them (capturing the surrounding variables by reference).

9.4 Rust language extensions

Rust is a language under constant development. The designers of the language are continuously looking to improve the language and to add new features that can increase expressiveness.

For instance, a proposed feature related to traits is *impl specialization* [4]. Impl specialization would allow users to declare overlapping impls, as long as one impl is more *specific* than another, given some definition of a 'more specific than' relation on constraints. Implementing such a feature in MiniRust would require a significant change to the constraint entailment relation: in the rules that define the constraint entailment relation we would need to reason about constraints that are *not* entailed. For example, to show that a constraint π is satisfied by an impl A, we must prove that there is no impl B that also satisfies π and is more specific than A. This means that, to prove that A satisfies π, we must enumerate all ways to satisfy π. In consequence, while defining the constraint entailment relation, we would need to reason about *all* the members of that same relation that we are defining. Unfortunately, such a relation is not well-defined. One way to resolve this could be to use iterated inductive definitions [23], which let us stratify the constraint entailment relation into multiple levels corresponding to the level of specialization of the impl used to resolve the constraint. This way, in the definition of a relation at level n, we only need to know about members of relations at levels m where $m < n$.

Chapter 10

Related work

There has already been work on formalizing a subset of the type system of Rust's type system focusing on the features that ensure memory safety, such as lifetimes, and the ownership and borrowing system [31]. The formalization, however, does not include traits. It would be interesting to combine the two models, particularly to examine the relationship between the memory safety-related features and traits.

The term *trait* has generally been used in object-oriented programming languages such as Smalltalk [32] and Scala [25]. In those cases, a trait is a composable unit of behavior that provides and requires functionality for/from classes that implement the trait. Rust traits are similar in nature, however, they are not associated with objects or classes and they can be implemented for any type. As such, Rust traits are more closely related to *type classes* in Haskell [40].

The formal presentation of Rust's type system in this thesis is also based on work on Haskell, particularly on the qualified types framework developed by Jones [20]. In [24], Odersky et al. provide another framework for Hindley-Milner type systems with constraints, called HM(X), along with a type inference algorithm. Stuckey and Sulzmann [37] combine HM(X) with constraint-handling rules [17] to provide a formal basis for type systems with ad-hoc polymorphism (or overloading).

Associated types in Rust are also similar to *associated type synonyms* in Haskell, first proposed by Chakravarty et al. in [12]. Both represent type-level functions and introduce a notion of type equality to their respective language. In one notable difference, Haskell functions can be constrained by equality constraints, while such constraints have to be expressed in extended trait bounds in Rust. Rust associated types are often viewed as output parameters of a trait and in that sense they bear similarities to Haskell's *functional dependencies* [22], which let the programmer declare dependency relationships between parameters of a type class.

In our formalization, we also follow the Haskell implementation model for type classes.

In particular, we use the dictionary-passing style for translating trait constraints. Our translation of associated types and equality constraints also mirrors the Haskell implementation, based on System FC [38, 41]. This is in contrast with the Rust language implementation where all type schemes are monomorphised. However, work by Jones [21] illustrates how type class constraint dictionaries in the internal language can be optimized away, thus removing the runtime overhead due to passing dictionaries around.

C++ templates bear some similarities to Rust generics. In particular, Rust's way of monomorphizing generics is inspired by the C++ implementation of templates, which are also monomorphised. Unlike Rust generic functions, C++ function templates are not type-checked before they are used—only the result of their instantiation is type-checked. Also, templates rely on documentation and special encodings [1] to be constrained. There have been proposals to include *concepts* in C++ [18], which would serve as a type system for templates and provide better error messages, bringing them closer to Rust's trait system.

The Swift programming language [9] features *protocols*, which are also somewhat similar to traits. A protocol specifies an interface for user-defined types (structs) and classes. The interface can be implemented directly in the definition of the struct or class, or in an *extension*, which then acts similarly to a Rust impl. Protocols can be used to constrain generic type variables, and they can be used as types, similarly to existential types and Rust's trait objects.

To ensure coherence and decidability of type inference, Haskell imposes more restrictions than Rust on its type class instance declarations. However, the Haskell standard does not prevent declaring orphan instances, which can cause coherence errors when linking to other modules. There has been work that proposes to let the programmer explicitly determine the scope of type class instances to ensure coherence, such as work by Dreyer et al. [15], which attempts to unify ML modules with Haskell type classes.

In [35], Siek and Lumsdaine present a new language called G, focused on generic programming. G features *concepts*, which are similar to Haskell type classes (and Rust traits), and *models*, similar to type class instances (and Rust impls). G concepts also include associated types. Models in G are lexically scoped, so two overlapping models can coexist in the same program as long as their scopes don't overlap. The core of the language, called F^G, which extends System F, is formalized in [34]. The work on G and F^G is also featured in Siek's PhD thesis [33].

Chapter 11

Conclusion

In this thesis, we studied a subset of Rust's type system, focusing on traits, which are the cornerstone of Rust's abstraction mechanism. Specifically, we developed a formal language, MiniRust, which models a subset of Rust including features relevant to traits. The features modeled include some advanced features of the trait mechanism such as supertraits, associated types and trait objects. Through the formalization process we have discovered corner cases in the design of the language with regards to object safety of traits (the ability to create trait objects). The formal model allowed us to determine very general conditions under which object safety can be guaranteed. We also developed runtime semantics for MiniRust through type-directed translation to an internal language, RustIn, which we proved type-safe. We proved that the translation of MiniRust programs to RustIn preserves types, which motivates our confidence in type safety of MiniRust. We also tackled the question of trait coherence on a subset of MiniRust, proving that the coherence rules in Rust guarantee safe linking to external libraries without breaking coherence and that they give programmers the ability to safely extend their libraries without inducing impl overlap in user code.

We hope that our work proves useful in shining new light on Rust and in providing a better understanding of Rust's trait system. In particular, our model can be a useful tool for reasoning about advanced features of traits (such as associated types and trait objects) as it abstracts away the implementation details of the language and it elides other features not covered in this thesis, both of which increase the language's complexity. In particular, our formalization of trait objects and object safety can help language designers to clarify the object safety requirements in Rust. As such, our work can inform the design and implementation of the language. Moreover, the declarative specification of MiniRust developed in this thesis can give rise to a corresponding type-checking algorithm. The algorithm, along with its prototype implementation, focusing on the core parts of the language, can serve as a reference point for the implementation of the Rust compiler.

Our model of Rust traits can also serve as a framework for exploring additional fea-

tures, whether they are already present in the Rust (like closures and lifetimes) or they are proposed future extensions (like higher-kinded types and impl specialization). We can thus explore how those features interact with the trait system in the context of MiniRust. Moreover, the proofs presented in this thesis provide some confidence in the guarantees that Rust strives to provide, such as type safety and trait coherence. Extending our model can thus help us determine if those guarantees can also be satisfied in the presence of the proposed extensions.

Bibliography

[1] The Boost concept check library. http://www.boost.org/doc/libs/master/libs/concept_check/concept_check.htm. Retrieved on 2015-11-29. → pages 109

[2] Rebalancing coherence. https://github.com/rust-lang/rfcs/blob/master/text/1023-rebalancing-coherence.md. Retrieved on 2015-11-23. → pages 92, 99

[3] Higher-ranked trait bounds (HRTBs). https://doc.rust-lang.org/nomicon/hrtb.html. Retrieved on 2015-11-23. → pages 105

[4] RFC: impl specialization. https://github.com/rust-lang/rfcs/pull/1210. Retrieved on 2015-11-23. → pages 107

[5] Object safety. https://github.com/rust-lang/rfcs/blob/master/text/0255-object-safety.md. Retrieved on 2015-11-23. → pages 47

[6] The Rust programming language. http://www.rust-lang.org, . Retrieved on 2015-11-23. → pages 2

[7] The Rust programming language. http://doc.rust-lang.org/nightly/book/README.html, . Retrieved on 2015-11-23. → pages 5

[8] The Servo browser engine. http://servo.org. Retrieved on 2015-11-23. → pages 2

[9] The Swift programming language. https://developer.apple.com/swift/. Retrieved on 2015-11-29. → pages 109

[10] V. Breazu-Tannen, T. Coquand, C. A. Gunter, and A. Scedrov. Inheritance as implicit coercion. *Inf. Comput.*, 93(1):172–221, July 1991. ISSN 0890-5401. doi:10.1016/0890-5401(91)90055-7. URL http://dx.doi.org/10.1016/0890-5401(91)90055-7. → pages 90

[11] K. Bruce, L. Cardelli, G. Castagna, G. T. Leavens, and B. Pierce. On binary methods. *Theor. Pract. Object Syst.*, 1(3):221–242, Dec. 1995. ISSN 1074-3227. URL http://dl.acm.org/citation.cfm?id=230849.230854. → pages 47

[12] M. M. T. Chakravarty, G. Keller, and S. P. Jones. Associated type synonyms. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, ICFP '05, pages 241–253, New York, NY, USA, 2005. ACM. ISBN 1-59593-064-7. doi:10.1145/1086365.1086397. URL http://doi.acm.org/10.1145/1086365.1086397. → pages 3, 29, 103, 108

[13] D. Clarke and T. Wrigstad. External uniqueness is unique enough. Technical Report UU-CS-2002-048, Department of Information and Computing Sciences, Utrecht University, 2002. → pages 2

[14] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pages 207–212, New York, NY, USA, 1982. ACM. ISBN 0-89791-065-6. doi:10.1145/582153.582176. URL http://doi.acm.org/10.1145/582153.582176. → pages 103, 104

[15] D. Dreyer, R. Harper, M. M. T. Chakravarty, and G. Keller. Modular type classes. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 63–70, New York, NY, USA, 2007. ACM. ISBN 1-59593-575-4. doi:10.1145/1190216.1190229. URL http://doi.acm.org/10.1145/1190216.1190229. → pages 109

[16] K.-F. Faxén. A static semantics for Haskell. *Journal of Functional Programming*, 12: 295–357, 2002. ISSN 1469-7653. doi:10.1017/S0956796802004380. URL http://journals.cambridge.org/article_S0956796802004380. → pages 75

[17] T. Frühwirth. Constraint handling rules. In *Constraint programming: Basics and trends*, pages 90–107. Springer, 1995. → pages 108

[18] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, and A. Lumsdaine. Concepts: Linguistic support for generic programming in C++. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 291–310, New York, NY, USA, 2006. ACM. ISBN 1-59593-348-4. doi:10.1145/1167473.1167499. URL http://doi.acm.org/10.1145/1167473.1167499. → pages 109

[19] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 282–293, New York, NY, USA, 2002. ACM. ISBN 1-58113-463-0. doi:10.1145/512529.512563. URL http://doi.acm.org/10.1145/512529.512563. → pages 2, 65, 105

[20] M. P. Jones. A theory of qualified types. In *ESOP'92*, pages 287–306. Springer, 1992. → pages 21, 75, 103, 104, 108

[21] M. P. Jones. Dictionary-free overloading by partial evaluation. In *In ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 107–117, 1994. → pages 103, 109

[22] M. P. Jones. Type classes with functional dependencies. In *Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 230–244. Springer-Verlag, 2000. → pages 108

[23] P. Martin-Löf. Haupstatz for the intuitionistic theory of iterated inductive definitions. In *Proc. Second Scandinavian Logic Symposium*, pages 179–216, 1971. → pages 107

[24] M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *Theor. Pract. Object Syst.*, 5(1):35–55, Jan. 1999. ISSN 1074-3227. doi:10.1002/(SICI)1096-9942(199901/03)5:1<35::AID-TAPO4>3.0.CO;2-4. URL http://dx.doi.org/10.1002/(SICI)1096-9942(199901/03)5:1<35::AID-TAPO4>3.0.CO;2-4. → pages 108

[25] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the scala programming language. Technical report, 2004. → pages 108

[26] S. Peyton Jones, D. Vytiniotis, S. Weirich, and M. Shields. Practical type inference for arbitrary-rank types. *J. Funct. Program.*, 17(1):1–82, Jan. 2007. ISSN 0956-7968. doi:10.1017/S0956796806006034. URL http://dx.doi.org/10.1017/S0956796806006034. → pages 104

[27] B. C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002. ISBN 0-262-16209-1. → pages 1

[28] B. C. Pierce. Existential types. In *Types and Programming Languages*, chapter 24, pages 363–380. MIT Press, Cambridge, MA, USA, 2002. ISBN 0-262-16209-1. → pages 13, 37, 41

[29] B. C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004. ISBN 0262162288. → pages 2

[30] B. C. Pierce and D. N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, Jan. 2000. ISSN 0164-0925. doi:10.1145/345099.345100. URL http://doi.acm.org/10.1145/345099.345100. → pages 103

[31] E. Reed. Patina: A formalization of the Rust programming language. ftp://ftp.cs.washington.edu/tr/2015/03/UW-CSE-15-03-02.pdf, 2015. Retrieved on 2015-11-23. → pages 105, 108

[32] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behaviour. In L. Cardelli, editor, *ECOOP 2003 - Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 248–274. Springer Berlin Heidelberg, 2003. ISBN 978-3-540-40531-3. doi:10.1007/978-3-540-45070-2_12. URL http://dx.doi.org/10.1007/978-3-540-45070-2_12. → pages 108

[33] J. G. Siek. *A Language for Generic Programming*. PhD thesis, Indiana University, August 2005. → pages 109

[34] J. G. Siek and A. Lumsdaine. Essential language support for generic programming. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 73–84, New York, NY, USA, 2005. ACM. ISBN 1-59593-056-6. doi:10.1145/1065010.1065021. URL http://doi.acm.org/10.1145/1065010.1065021. → pages 109

[35] J. G. Siek and A. Lumsdaine. Language requirements for large-scale generic libraries. In *GPCE '05: Proceedings of the fourth international conference on Generative Programming and Component Engineering*, September 2005. → pages 109

[36] C. Strachey. Fundamental concepts in programming languages. *Higher Order Symbol. Comput.*, 13(1-2):11–49, Apr. 2000. ISSN 1388-3690. doi:10.1023/A:1010000313106. URL http://dx.doi.org/10.1023/A:1010000313106. → pages 3

[37] P. J. Stuckey and M. Sulzmann. A theory of overloading. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 167–178, New York, NY, USA, 2002. ACM. ISBN 1-58113-487-8. doi:10.1145/581478.581495. URL http://doi.acm.org/10.1145/581478.581495. → pages 108

[38] M. Sulzmann, M. M. T. Chakravarty, S. P. Jones, and K. Donnelly. System F with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '07, pages 53–66, New York, NY, USA, 2007. ACM. ISBN 1-59593-393-X. doi:10.1145/1190315.1190324. URL http://doi.acm.org/10.1145/1190315.1190324. → pages 55, 72, 109

[39] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109 – 176, 1997. ISSN 0890-5401. doi:10.1006/inco.1996.2613. URL http://www.sciencedirect.com/science/article/pii/S0890540196926139. → pages 2, 105

[40] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 60–76, New York, NY, USA, 1989. ACM. ISBN 0-89791-294-2. doi:10.1145/75277.75283. URL http://doi.acm.org/10.1145/75277.75283. → pages 3, 18, 108

[41] S. Weirich, D. Vytiniotis, S. Peyton Jones, and S. Zdancewic. Generative type abstraction and type-level computation. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 227–240, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0. doi:10.1145/1926385.1926411. URL http://doi.acm.org/10.1145/1926385.1926411. → pages 109

Appendix A

Proof of type safety of RustIn

Definition 6.1. *Let $A_\mu = \{w \mid \exists l.\mu(l) = w\}$. In other words, A_μ is the image of μ.*
Define $\sqsubset_\mu \subseteq A_\mu \times A_\mu$ as follows:

- $\mu(l) \sqsubset_\mu S\,\langle \overline{u} \rangle\,\{\overline{x_m : l_n}, x : l, \overline{x_n : l_n}\}$
- $\mu(l) \sqsubset_\mu S\,\langle \overline{u} \rangle\,\{\overline{x_m : l_n}, x : l, \overline{x_n : l_n}\} \blacktriangleright \gamma$

Definition 6.2 (Well-typed store)**.** *We say that $\Omega \mid \Gamma \mid \Sigma \vdash \mu$, meaning that the store μ is well typed with respect to the environments Ω, Γ and the store typing Σ if*

1. *$dom(\mu) = dom(\Sigma)$ and*
2. *$\Omega \mid \Gamma \mid \Sigma \vdash \mu(l) : \Sigma(l)$ for every $l \in dom(\mu)$*
3. *$\sqsubset_\mu$ is a well-founded relation on the set $A_\mu = \{w \mid \exists l.\mu(l) = w\}$, i.e., it has no infinite descending chains.*

Definition 6.3 (Constructed type)**.** *A **constructed type** is a type η that is of the form $(), S\,\langle \overline{u} \rangle, fn(\overline{\tau}) \to \tau, \&u, (\exists T, \overline{\tau}, \overline{\omega})$, $\forall \overline{X_i}.\eta$ or $\forall \overline{\omega_i}.\eta$.*

Definition 6.4 (Consistency of coercion environment Ω)**.** *We say that Ω is **consistent** if the following hold:*

1. *If $\Omega \vdash \gamma : S\,\langle \overline{u} \rangle \sim \eta$ and η is a constructed type then $\eta = S\,\langle \overline{u'} \rangle$ for some $\overline{u'}$.*
2. *If $\Omega \vdash \gamma : fn(\overline{\tau_i}) \to \tau \sim \eta$ and η is a constructed type then $\eta = fn(\overline{\tau'_i}) \to \tau'$ for some $\overline{\tau'_i}, \tau'$.*
3. *If $\Omega \vdash \gamma : \&u_1 \sim \eta$ and η is a constructed type then $\eta = \&u'_1$ for some u'_1.*
4. *If $\Omega \vdash \gamma : (\exists T, \overline{\tau_i}, \overline{\omega_j}) \sim \eta$ and η is a constructed type then $\eta = (\exists T, \overline{\tau'_i}, \overline{\omega'_j})$ for some $\overline{\tau'_i}, \overline{\omega'_j}$.*
5. *If $\Omega \vdash \gamma : \forall \overline{X_i}.\sigma \sim \eta'$ where η' is any type, then $\eta' = \forall \overline{X_i}.\sigma'$ for some σ'.*

6. *If* $\Omega \vdash \gamma : \forall \overline{\omega_i}.\sigma \sim \eta'$ *where* η' *is any type, then* $\eta' = \forall \overline{\omega'_i}.\sigma'$ *for some* $\overline{\omega'_i}, \sigma'$.

Corollary A.1. *Suppose consistent* Ω. *If* $\Omega \vdash \gamma : \tau \sim \sigma$ *then* $\sigma \in \text{STYPE}$.

Lemma A.1 (Inversion lemma for typing terms)**.** *The following are all true:*

1. *If* $\Omega \mid \Gamma \mid \Sigma \vdash \mathit{let}\ x : \tau = e_1\ \mathit{in}\ e_2 : \sigma$ *then there is some type* τ_2 *such that* $\sigma = \tau_2$ *and* $\Omega \mid \Gamma \mid \Sigma \vdash e_1 : \tau_1$ *and* $\Omega \mid \Gamma, x : \tau \mid \Sigma \vdash e_2 : \tau_2$.
2. *If* $\Omega \mid \Gamma \mid \Sigma \vdash (lv := e) : \sigma$ *then* $\sigma = ()$ *and* $\Omega \mid \Gamma \mid \Sigma \vdash lv : \tau'$ *and* $\Omega \mid \Gamma \mid \Sigma \vdash e : \tau'$ *for some* τ'.
3. *If* $\Omega \mid \Gamma \mid \Sigma \vdash e(\overline{e_i}) : \sigma$ *then* $\Omega \mid \Gamma \mid \Sigma \vdash e : \mathit{fn}(\overline{\tau_i}) \to \tau$ *and* $\overline{\Omega \mid \Gamma \mid \Sigma \vdash e_i : \tau_i}$ *and* $\sigma = \tau$ *for some* τ *and* $\overline{\tau_i}$.
4. *If* $\Omega \mid \Gamma \mid \Sigma \vdash \mathit{fn}\ (\overline{x_i : \tau_1})\{e\} : \sigma$ *then* $\Omega \mid \Gamma, \overline{x_i : \tau_i} \mid \Sigma \vdash e : \tau$ *and* $\sigma = \mathit{fn}(\overline{\tau_i}) \to \tau$ *for some* τ.
5. *If* $\Omega \mid \Gamma \mid \Sigma \vdash x : \sigma$ *then* $(x : \sigma) \in \Gamma$.
6. *If* $\Omega \mid \Gamma \mid \Sigma \vdash l : \sigma$ *then* $(l : \sigma) \in \Sigma$.
7. *If* $\Omega \mid \Gamma \mid \Sigma \vdash \&lv : \sigma$, *then there is some type* τ *such that* $\sigma = \&\tau$ *and* $\Omega \mid \Gamma \mid \Sigma \vdash lv : \tau$.
8. *If* $\Omega \mid \Gamma \mid \Sigma \vdash *e : \sigma$ *then there is some type* τ *such that* $\sigma = \tau$ *and* $\Omega \mid \Gamma \mid \Sigma \vdash e : \&\tau$.
9. *If* $\Omega \mid \Gamma \mid \Sigma \vdash e.x : \sigma$ *then* $\Omega \mid \Gamma \mid \Sigma \vdash e : S\ \langle \overline{u_j} \rangle$ *for some* S *and* $\overline{u_j}$ *and* $S\ \langle \overline{X_j} \rangle\ \{\overline{x_i : \sigma_i}, x : \sigma', \overline{x_k : \sigma_k}\} \in \Omega$ *for some* $\overline{X_j}, \overline{x_i}, \overline{x_k}, \overline{\sigma_i}, \sigma', \overline{\sigma_k}$ *and* $\sigma = [\overline{u_j / X_j}]\sigma'$.
10. *If* $\Omega \mid \Gamma \mid \Sigma \vdash S\ \langle \overline{u_j} \rangle\ \{\overline{x_i : e_i}\} : \sigma$ *then* $S\ \langle \overline{X_j} \rangle\ \{\overline{x_i : \sigma_i}\} \in \Omega$, $\overline{\Omega \mid \Gamma \mid \Sigma \vdash e_i : [\overline{u_j / X_j}]\sigma_i}$ *and* $\sigma = S\ \langle \overline{u_j} \rangle$ *for some* $\overline{X_j}, \overline{\sigma_i}$.
11. *If* $\Omega \mid \Gamma \mid \Sigma \vdash e \blacktriangleright \gamma : \sigma$ *then* $\Omega \vdash \gamma : \sigma' \sim \sigma$ *for some* σ' *and* $\Omega \mid \Gamma \mid \Sigma \vdash e : \sigma'$.
12. *If* $\Omega \mid \Gamma \mid \Sigma \vdash e_1; e_2 : \sigma$ *then there are some types* τ_1, τ_2 *such that* $\Omega \mid \Gamma \mid \Sigma \vdash e_1 : \tau_1$, *and* $\Omega \mid \Gamma \mid \Sigma \vdash e_2 : \tau_2$, *and* $\sigma = \tau_2$.
13. *If* $\Omega \mid \Gamma \mid \Sigma \vdash \mathit{let}\ (T, \overline{x_i}) = \mathit{unpack}\ e_1\ \mathit{in}\ e_2 : \sigma$ *then there are some* $\tau, \overline{\tau_i}, \overline{\omega}$ *such that* $\Omega \mid \Gamma \mid \Sigma \vdash e_1 : \&(\exists T, \overline{\tau_i}, \overline{\omega})$ *and* $\Omega \mid \Gamma, \overline{x_i : \tau_i} \mid \Sigma \vdash e_2 : \tau$ *and* $\sigma = \tau$ *with* $T \notin FV(\tau)$.
14. *If* $\Omega \mid \Gamma \mid \Sigma \vdash \mathit{pack}\ (\tau, \overline{e_i}, \overline{\gamma_j})\ \mathit{as}\ \&(\exists T, \overline{\tau_i}, \overline{\omega_j}) : \sigma$ *then* $\overline{\Omega \mid \Gamma \mid \Sigma \vdash e_i : [\tau/T]\tau_i}$, *and* $\overline{\Omega \vdash \gamma_j : [\tau/T]\omega_j}$, *and* $\sigma = \&(\exists T, \overline{\tau_i}, \overline{\omega_j})$.
15. *If* $\Omega \mid \Gamma \mid \Sigma \vdash e[\overline{u_i}] : \sigma$ *then there are some* $\overline{X_i}, \sigma'$ *such that* $\Omega \mid \Gamma \mid \Sigma \vdash e : \forall \overline{X_i}.\sigma'$ *and* $\sigma = [\overline{u_i / X_i}]\sigma'$.
16. *If* $\Omega \mid \Gamma \mid \Sigma \vdash \Lambda \overline{X_i}.e : \sigma$ *then there is some* σ' *such that* $\Omega \mid \Gamma, \overline{X_i} \mid \Sigma \vdash e : \sigma'$ *and* $\sigma = \forall \overline{X_i}.\sigma'$.

17. *If* $\Omega \mid \Gamma \mid \Sigma \vdash e[\![\overline{\gamma_i}]\!] : \sigma$ *then there are some* $\overline{\omega_i}$ *such that* $\Omega \mid \Gamma \mid \Sigma \vdash e : \forall \overline{\omega_i}.\sigma$ *and* $\overline{\Omega \vdash \gamma_i : \omega_i}$.

18. *If* $\Omega \mid \Gamma \mid \Sigma \vdash \Lambda \overline{c_i : \omega_i}.e : \sigma$ *then there is some* σ' *such that* $\Omega, \overline{c_i : \omega_i} \mid \Gamma \mid \Sigma \vdash e : \sigma'$ *and* $\sigma = \forall \overline{\omega_i}.\sigma'$.

19. *If* $\Omega \mid \Gamma \mid \Sigma \vdash () : \sigma$ *then* $\sigma = ()$.

Proof. Immediate from the typing relation $\Omega \mid \Gamma \mid \Sigma \vdash e : \sigma$. □

Corollary A.2. *If* $\Omega \mid \Gamma \mid \Sigma \vdash pv : \sigma$ *then* σ *is a constructed type.*

Corollary A.3. *If* $\Omega \mid \Gamma \mid \Sigma \vdash pw : \sigma$ *then* σ *is a constructed type.*

Lemma A.2 (Canonical forms)**.** *Assume consistent* Ω.

1. *If* $\Omega \mid \Gamma \mid \Sigma \vdash v : \&\tau$ *then either:*

   - $v = \&l$ *for some* l, *or*
   - $v = \&l \blacktriangleright \gamma$ *for some* l *and* γ *where* $\Omega \vdash \gamma : \&\tau' \sim \&\tau$ *for some type* τ'.

2. *If* $\Omega \mid \Gamma \mid \Sigma \vdash v : S\ \langle \overline{u_j} \rangle$ *then either:*

   - $v = S\ \langle \overline{u_j} \rangle\ \{\overline{x_i : v_i}\}$ *for some* $\overline{x_i}$ *and* $\overline{v_i}$, *or*
   - $v = S\ \langle \overline{u'_j} \rangle\ \{\overline{x_i : v_i}\} \blacktriangleright \gamma$ *for some* $\gamma, \overline{u'_j}, \overline{x_i}$ *and* $\overline{v_i}$ *where* $\Omega \vdash \gamma : S\ \langle \overline{u'_j} \rangle \sim S\ \langle \overline{u_j} \rangle$

3. *If* $\Omega \mid \Gamma \mid \Sigma \vdash v : \mathit{fn}(\overline{\tau_i}) \to \tau$ *then either:*

   - $v = \mathit{fn}\ (\overline{x_i : \tau_i})\{e\}$ *for some* $\overline{x_i}, e$
   - $v = \mathit{fn}\ (\overline{x_i : \tau'_i})\{e\} \blacktriangleright \gamma$ *for some* $\overline{x_i}, e, \gamma$ *where* $\Omega \vdash \gamma : \mathit{fn}(\overline{\tau'_i}) \to \tau' \sim \mathit{fn}(\overline{\tau_i}) \to \tau$ *for some* τ'.

4. *If* $\Omega \mid \Gamma \mid \Sigma \vdash v : \&(\exists T, \overline{\tau_i}, \overline{\omega_j})$ *then either:*

   - $v = \mathit{pack}\ (\tau, \overline{v_i}, \overline{\gamma_j})$ *as* $\&(\exists T, \overline{\tau_i}, \overline{\omega_j})$ *for some* $\tau, \overline{v_i}, \overline{\gamma_j}$ *or*
   - $v = (\mathit{pack}\ (\tau, \overline{v_i}, \overline{\gamma_j})$ *as* $\&(\exists T, \overline{\tau'_i}, \overline{\omega'_j})) \blacktriangleright \gamma$ *where* $\Omega \vdash \gamma : \&(\exists T, \overline{\tau'_i}, \overline{\omega'_j})) \sim \&(\exists T, \overline{\tau_i}, \overline{\omega_j})$ *for some* $\gamma, \tau, \overline{v_i}, \overline{\gamma_j}, \overline{\omega'_j}, \overline{\tau'_i}$.

5. *If* $\Omega \mid \Gamma \mid \Sigma \vdash v : \forall \overline{X_i}.\sigma$ *then either:*

   - $v = \Lambda \overline{X_i}.e$ *for some* e, *or*
   - $v = (\Lambda \overline{X_i}.e) \blacktriangleright \gamma$ *where* $\Omega \vdash \gamma : \forall \overline{X_i}.\sigma' \sim \forall \overline{X_i}.\sigma$ *for some* e, γ, σ'.

6. *If* $\Omega \mid \Gamma \mid \Sigma \vdash v : \forall \overline{\omega_i}.\sigma$ *then either:*

- $v = \Lambda\overline{c_i : \omega_i}.e$ *for some* $\overline{c_i}, e$, *or*
- $v = (\Lambda\overline{c_i : \omega'_i}.e) \blacktriangleright \gamma$ *where* $\Omega \vdash \gamma : \forall\overline{\omega'_i}.\sigma' \sim \forall\overline{\omega_i}.\sigma$ *for some* $\overline{c_i}, e, \gamma, \overline{\omega'_i}, \sigma'$.

Proof. If $\Omega \mid \Gamma \mid \Sigma \vdash v : \sigma$ then the following rules of the judgment $\Omega \mid \Gamma \mid \Sigma \vdash e : \sigma$ could have been used as the last rule in the derivation where the structure of e can match some value v: (t-fabs), (t-ref), (t-unit), (t-newstruct), (t-pack), (t-tabs), (t-cabs), (t-coerce).

1. The only rules matching are (t-ref) for $v = \&l$ and (t-coerce) for $v = pv \blacktriangleright \gamma$. By lemma A.1 there is some σ such that $\Omega \vdash \gamma : \sigma \sim \&\tau$ and $\Omega \mid \Gamma \mid \Sigma \vdash pv : \sigma$. By corollary A.2 and consistency of Ω, σ must be of the form $\&\tau'$ for some τ'. Then the last rule used to get $\Omega \mid \Gamma \mid \Sigma \vdash pv : \&\tau'$ can only be (t-ref) and $pv = \&l$.
2. Analogous to case 1. with rules (t-newstruct) and (t-coerce).
3. Analogous to case 1. with rules (t-fabs) and (t-coerce).
4. Analogous to case 1. with rules (t-pack) and (t-coerce).
5. Analogous to case 1. with rules (t-tabs) and (t-coerce).
6. Analogous to case 1. with rules (t-cabs) and (t-coerce).

□

Lemma A.3 (Canonical forms for store values)**.** *Assume consistent* Ω.

1. *If* $\Omega \mid \Gamma \mid \Sigma \vdash w : S\,\langle\overline{u_j}\rangle$ *and* $S\,\langle\overline{X_j}\rangle\,\{\overline{x_i : \sigma_i}\} \in \Omega$ *then either:*
   - $w = S\,\langle\overline{u_j}\rangle\,\{\overline{x_i : l_i}\}$ *for some* $\overline{l_i}$ *or*
   - $w = S\,\langle\overline{u'_j}\rangle\,\{\overline{x_i : l_i}\} \blacktriangleright \gamma$ *for some* $\gamma, \overline{l_i}, \overline{u'_j}$ *where* $\Omega \vdash \gamma : S\,\langle\overline{u'_j}\rangle \sim S\,\langle\overline{u_j}\rangle$.
2. *If* $\Omega \mid \Gamma \mid \Sigma \vdash w : \tau$ *and* $\Omega \vdash \gamma : \tau \sim S\,\langle\overline{u_j}\rangle$ *with* $S\,\langle\overline{X_j}\rangle\,\{\overline{x_i : \sigma_i}\} \in \Omega$ *then either:*
   - $w = S\,\langle\overline{u'_j}\rangle\,\{\overline{x_i : l_i}\}$ *and* $\tau = S\,\langle\overline{u'_j}\rangle$, *or*
   - $w = S\,\langle\overline{u'_j}\rangle\,\{\overline{x_i : l_i}\} \blacktriangleright \gamma'$ *where* $\Omega \vdash \gamma' : S\,\langle\overline{u'_j}\rangle \sim \tau$

Proof.

1. The last typing rule used in the derivation $\Omega \mid \Gamma \mid \Sigma \vdash w : S\,\langle\overline{u_j}\rangle$ must be either (t-newstruct) or (t-coerce). If the last rule used is (t-newstruct) then w must have form $S\,\langle\overline{u_j}\rangle\,\{\overline{x_i : l_i}\}$ as required.
   If the last rule used is (t-coerce), then $w = pw \blacktriangleright \gamma$ for some pw and γ.
   Furthermore, there is some σ such that

$\Omega \vdash \gamma : \sigma \sim S\,\langle \overline{u_j} \rangle$ and $\Omega \mid \Gamma \mid \Sigma \vdash pw : \sigma$.
By corollary A.3 and consistency of Ω, σ must be of the form $S\,\langle \overline{u'_j} \rangle$ for some $\overline{u'_j}$. Then the last rule used in the derivation $\Omega \mid \Gamma \mid \Sigma \vdash pv : S\,\langle \overline{u'_j} \rangle$ must be (t-newstruct) and pw must have form $S\,\langle \overline{u'_j} \rangle\,\{\overline{x_i : l_i}\}$.

2. Either τ is a constructed type or not. If it is a constructed type then by consistency of Ω it is of the form $S\,\langle \overline{u'_j} \rangle$ and point 1 of this lemma applies.
If τ is not a constructed type, then from examining all the typing rules only rule (t-coerce) can apply as the last rule used in the derivation $\Omega \mid \Gamma \mid \Sigma \vdash w : \tau$.
In consequence, there are some pw, σ', γ' such that
$w = pw \blacktriangleright \gamma'$, and $\Omega \vdash \gamma' : \sigma' \sim \tau$, and $\Omega \mid \Gamma \mid \Sigma \vdash pw : \sigma'$.
By (co-trans), $\Omega \vdash \gamma' \circ \gamma : \sigma' \sim S\,\langle \overline{u_j} \rangle$
and by corollary A.3 and consistency, $\sigma' = S\,\langle \overline{u'_j} \rangle$ for some $\overline{u'_j}$.
We can then use point 1 of this lemma to get $pw = S\,\langle \overline{u'_j} \rangle\,\{\overline{x_i : l_i}\}$ for some $\overline{l_i}$.

□

Lemma A.4 (Weakening for coercing typing)**.** *If $\Omega \vdash \gamma : \omega$ then $\Omega, \Omega' \vdash \gamma : \omega$ for any Ω' such that the domains of Ω and Ω' don't overlap.*

Proof. Straightforward by inductions on derivations $\Omega \vdash \gamma : \omega$. □

Lemma A.5 (Weakening for term typing)**.** *If $\Omega \mid \Gamma \mid \Sigma \vdash e : \sigma$ then $\Omega, \Omega' \mid \Gamma, \Gamma' \mid \Sigma, \Sigma' \vdash e : \sigma$ for any $\Omega', \Gamma', \Sigma'$ with no overlapping domains with Ω, Γ, Σ respectively.*

Proof. By straightforward rule induction on $\Omega \mid \Gamma \mid \Sigma \vdash e : \sigma$ using lemma A.4 when required. □

Lemma A.6 (Term substitution)**.** *If $\Omega \mid \Gamma \mid \Sigma \vdash e' : \sigma'$ and $\Omega \mid \Gamma, x' : \sigma' \mid \Sigma \vdash e'' : \sigma''$ then $\Omega \mid \Gamma \mid \Sigma \vdash [e'/x']e'' : \sigma''$*

Proof. By induction on derivations $\Omega \mid \Gamma, x' : \sigma' \mid \Sigma \vdash e'' : \sigma''$:

- Case (t-let): $e'' = \mathsf{let}\ x : \tau_1 = e_1\ \mathsf{in}\ e_2$ and $\sigma'' = \tau_2$.

  By bound variable renaming convention, we assume $x \neq x'$ and $x \notin FV(e')$,

  Then $\Omega \mid \Gamma, x' : \sigma' \mid \Sigma \vdash e_1 : \tau_1$ and $\Omega \mid \Gamma, x' : \sigma', x : \tau_1 \mid \Sigma \vdash e_2 : \tau_2$.
  By ind. hyp. $\Omega \mid \Gamma \mid \Sigma \vdash [e'/x']e_1 : \tau_1$ and $\Omega \mid \Gamma, x : \tau_1 \mid \Sigma \vdash [e'/x']e_2 : \tau_2$.
  By (t-let) $\Omega \mid \Gamma \mid \Sigma \vdash \mathsf{let}\ x : \tau_1 = [e'/x']e_1\ \mathsf{in}\ [e'/x']e_2 : \tau_2$ where by definition of substitution $(\mathsf{let}\ x : \tau_1 = [e'/x']e_1\ \mathsf{in}\ [e'/x']e_2) \equiv [e'/x']e''$ as required.

- Case (t-upd): Straightforward use of ind. hyp.

- Case (t-fapp): Straightforward use of ind. hyp.
- Case (t-fabs): $e'' = \mathsf{fn}\ (\overline{x_i : \tau_i})\{e\}$ and $\sigma'' = \mathsf{fn}(\overline{\tau_i}) \to \tau$.

  By bound variable renaming convention, we assume $x' \notin \overline{x_i}$ and $\overline{x_i} \notin FV(e')$.

  Then $\Omega \mid \Gamma, x' : \sigma', \overline{x_i : \tau_i} \mid \Sigma \vdash e : \tau$.

  By ind. hyp. $\Omega \mid \Gamma, \overline{x_i : \tau_i} \mid \Sigma \vdash [e'/x']e : \tau$ which lets us prove $\Omega \mid \Gamma \mid \Sigma \vdash [e'/x']e'' : \sigma''$ as required.
- Case (t-ref): Straightforward use of ind. hyp.
- Case (t-deref): Straightforward use of ind. hyp.
- Case (t-var): $\Omega \mid \Gamma, x' : \sigma' \vdash x : \sigma$ with $(x : \sigma) \in (\Omega \mid \Gamma, x' : \sigma')$.
  If $x \neq x'$ then $\Omega \mid \Gamma \vdash x : \sigma$ is immediate.
  If $x = x'$ then $\sigma = \sigma'$ as $(x : \sigma) \in (\Omega \mid \Gamma, x' : \tau')$ and $\Omega \mid \Gamma \mid \Sigma \vdash [e'/x']x : \sigma'$ from the lemma's assumption.
- Case (t-unit): Vacuous.
- Case (t-newstruct): Straightforward use of ind. hyp on $\overline{e_i}$ with the fact
  $[e'/x'](S\ \langle \overline{u_j} \rangle\ \{\overline{x_i : e_i}\}) \equiv S\ \langle \overline{u_j} \rangle\ \{\overline{x_i : [e'/x']e_i}\}$
- Case (t-proj): Straightforward use of ind. hyp on e with the fact
  $[e'/x'](e.x) \equiv ([e'/x']e).x$
- Case (t-seq): Straightforward use of ind. hyp.
- Case (t-unpack): Similar to (t-let) assuming $x' \notin \overline{x_i}$ and $\overline{x_i} \notin FV(e')$.
- Case (t-pack): $e'' = \mathsf{pack}\ (\tau, \overline{e_i}, \overline{\gamma_j})\ \mathsf{as}\ \&(\exists T, \overline{\tau_i}, \overline{\omega_j})$ and $\tau'' = \&(\exists T, \overline{\tau_i}, \overline{\omega_j})$
  $\overline{\Omega \mid \Gamma \mid \Sigma \vdash [e'/x']e_i : [\tau/T]\tau_i}^{\,i}$ by ind. hyp.
  $\Omega \mid \Gamma \mid \Sigma \vdash [e'/x']e'' : \sigma''$ by (t-pack).
- Case (t-tapp): Straightforward use of ind. hyp.
- Case (t-tabs): Straightforward use of ind. hyp.
- Case (t-capp): Straightforward use of ind. hyp.
- Case (t-cabs): Straightforward use of ind. hyp.
- Case (t-coerce): Straightforward use of ind. hyp.
- Case (t-loc): Trivial.

□

Lemma A.7 (Types are coercions)**.** $\Omega \vdash \eta : \eta \sim \eta$ *for any* η *and* Ω.

Proof. By structural induction on η:

- Case $\eta = ()$:

  $\Omega \vdash () : () \sim ()$ by (co-unit).

- Case $\eta = X$:

  $\Omega \vdash X : X \sim X$ by (co-tvar).

- Case $\eta = A\,\langle \overline{u_i} \rangle$:

  By ind. hyp. $\overline{\Omega \vdash u_i : u_i \sim u_i}$ for any Ω.

  Then $\Omega \vdash A\,\langle \overline{u_i} \rangle \sim A\,\langle \overline{u_i} \rangle$ by (co-atype).

- Case $\eta = (\exists T, \overline{\tau_i}, \overline{\omega_j})$: Let $\overline{\omega_j = u_j \sim u'_j}$.

  By ind. hyp. $\overline{\Omega \vdash \tau_i \sim \tau_i}$ and $\overline{\Omega \vdash u_j : u_j \sim u_j}$ and $\overline{\Omega \vdash u'_j : u'_j \sim u'_j}$ for any Ω.

  Then $\Omega \vdash (\exists T, \overline{\tau_i}, \overline{\omega_j}) : (\exists T, \overline{\tau_i}, \overline{\omega_j}) \sim (\exists T, \overline{\tau_i}, \overline{\omega_j})$ by (co-obj).

- Case $\eta = \mathsf{fn}(\overline{\tau_i}) \to \tau$

  By ind. hyp. $\overline{\Omega \vdash \tau_i : \tau_i \sim \tau_i}$ and $\Omega \vdash \tau : \tau \sim \tau$ for any Ω.

  Then by (co-fun) $\Omega \vdash \mathsf{fn}(\overline{\tau_i}) \to \tau : \mathsf{fn}(\overline{\tau_i}) \to \tau \sim \mathsf{fn}(\overline{\tau_i}) \to \tau$

- Case $\eta = \&u'$:

  By ind. hyp. $\Omega \vdash u' : u' \sim u'$.

  Then by (co-ref) $\Omega \vdash \&u' : \&u' \sim \&u'$.

- Case $\eta = S\,\langle \overline{u_i} \rangle$:

  By ind. hyp. $\overline{\Omega \vdash u_i : u_i \sim u_i}$.

  Then by (co-struct) $\Omega \vdash S\,\langle \overline{u_i} \rangle : S\,\langle \overline{u_i} \rangle \sim S\,\langle \overline{u_i} \rangle$.

- Case $\eta = \forall \overline{X_i}.\eta$.

  By ind. hyp. $\Omega \vdash \eta : \eta \sim \eta$.

  Then by (co-tabs) $\Omega \vdash \forall \overline{X_i}.\eta : \overline{X_i}.\eta \sim \overline{X_i}.\eta$.

- Case $\eta = \forall\overline{\omega_i}.\eta$.

  Let $\overline{\omega_i = u_i \sim u'_i}$.

  By ind. hyp. $\Omega \vdash \eta : \eta \sim \eta$, and $\Omega \vdash u_i : u_i \sim u_i$, and $\Omega \vdash u'_i : u'_i \sim u'_i$.

  Then by (co-cabs) $\Omega \vdash \forall\overline{\omega_i}.\eta : (\forall\overline{\omega_i}.\eta) \sim (\forall\overline{\omega_i}.\eta)$.

□

Lemma A.8 (Type substitution in coercions)**.** *If* $\Omega \vdash \gamma : \vartheta$ *then* $[u/X]\Omega \vdash [u/X]\gamma : [u/X]\vartheta$.

Proof. By induction on derivations $\Omega \vdash \gamma : \vartheta$:

- Case (co-var): $\Omega \vdash c : \vartheta$.
  Then $(c : [u/X]\vartheta) \in [u/X]\Omega$.
  and $[u/X]\Omega \vdash c : [u/X]\vartheta$ as required.
- Case (co-sym): Straightforward use of ind. hyp.
- Case (co-trans): Straightforward use of ind. hyp.
- Case (co-tapp): Straightforward use of ind. hyp.
- Case (co-tabs): Straightforward use of ind. hyp.
- Case (co-cabs): Straightforward use of ind. hyp.
- Case (co-unit): $\Omega \vdash () : () \sim ()$.
  Then $[u/X]\Omega \vdash [u/X]() : [u/X]() \sim [u/X]()$ as required since $[u/X]() = ()$.
- Case (co-tvar): $\Omega \vdash X' : X' \sim X'$.
  If $X = X'$ then $[u/X]\Omega \vdash u : u \sim u$ by lemma A.7.
  Otherwise $[X/u]X' = X'$ and by (co-tvar) $[u/X]\Omega \vdash X' : X' \sim X'$ as required.
- Case (co-atype): $\Omega \vdash A\ \langle\overline{\gamma_i}\rangle : A\ \langle\overline{u_i}\rangle \sim A\ \langle\overline{u'_i}\rangle$.
  By ind. hyp. $\overline{[u/X]\Omega \vdash [u/X]\gamma_i : [u/X]u_i \sim [u/X]u'_i}$.
  Then by (co-atype) and def. of substitution
  $[u/X]\Omega \vdash [u/X]A\ \langle\overline{\gamma_i}\rangle : [u/X]A\ \langle\overline{u_i}\rangle \sim [u/X]A\ \langle\overline{u'_i}\rangle$ as required.
- Case (co-struct): Straightforward use of ind. hyp.
- Case (co-ref): Straightforward use of ind. hyp.
- Case (co-obj): Straightforward use of ind. hyp. assuming $X \neq T$ by the bound variable renaming convention.

- Case (co-fun): Straightforward use of ind. hyp.
- Case (co-deref): Straightforward use of ind. hyp.
- Case (co-spar): Straightforward use of ind. hyp.
- Case (co-farg): Straightforward use of ind. hyp.
- Case (co-fret): Straightforward use of ind. hyp.
- Case (co-objf): Straightforward use of ind. hyp. assuming $T \neq X$ by the bound variable renaming assumption.
- Case (co-objc-l): Straightforward use of ind. hyp. assuming $T \neq X$.
- Case (co-objc-r): Straightforward use of ind. hyp. assuming $T \neq X$.
- Case (co-coer): Straightforward use of ind. hyp.
- Case (co-coer-l): Straightforward use of ind. hyp.
- Case (co-coer-r): Straightforward use of ind. hyp.

□

Lemma A.9 (Type substitution)**.** *If* $\Omega \mid \Gamma, X \mid \Sigma \vdash e : \sigma$ *then* $[u/X]\Omega \mid [u/X]\Gamma \mid [u/X]\Sigma \vdash [u/X]e : [u/X]\sigma$.

Proof. By induction on derivations $\Omega \mid \Gamma, X \mid \Sigma \vdash e : \sigma$.

- Case (t-let): $\Omega \mid \Gamma, X \mid \Sigma \vdash \mathsf{let}\ x : \tau_1 = e_1\ \mathsf{in}\ e_2 : \tau_2$.

  By ind. hyp. $[u/X]\Omega \mid [u/X]\Gamma \mid [u/X]\Sigma \vdash [u/X]e_1 : [u/X]\tau_1$ and $[u/X]\Omega \mid [u/X]\Gamma, x : [u/X]\tau_1 \mid [u/X]\Sigma x : \tau_1 \vdash [u/X]e_2 : [u/X]\tau_2$.

  Then by rule (t-let) and the definition of substitution, $[u/X]\Omega \mid [u/X]\Gamma \mid [u/X]\Sigma \vdash [u/X](\mathsf{let}\ x : \tau_1 = e_1\ \mathsf{in}\ e_2 : \tau_2) : [u/X]\tau_2$
- Case (t-upd): Straightforward use of ind. hyp. as above.
- Case (t-fapp): Straightforward use of ind. hyp.
- Case (t-fabs): Straightforward use of ind. hyp.
- Case (t-ref): Straightforward use of ind. hyp.
- Case (t-deref): Straightforward use of ind. hyp.

- Case (t-var): $\Omega \mid \Gamma, X \mid \Sigma \vdash x : \sigma$.
  Then $(x : [u/X]\sigma) \in [u/X]\Gamma$ which lets us prove $[u/X]\Omega \mid [u/X]\Gamma \mid [u/X]\Sigma \vdash x : [u/X]\sigma$.

- Case (t-unit): Vacuous.

- Case (t-newstruct): Straightforward use of ind. hyp.

- Case (t-proj): Straightforward use of ind. hyp.

- Case (t-seq): Straightforward use of ind. hyp.

- Case (t-unpack): $\Omega \mid \Gamma, X \mid \Sigma \vdash \mathsf{let}\ (T, \overline{x_i}) = \mathsf{unpack}\ e_1\ \mathsf{in}\ e_2 : \tau$.
  By bound variable renaming convention, assume $T \neq X$.
  By ind. hyp. $[u/X]\Omega \mid [u/X]\Gamma \mid [u/X]\Sigma \vdash [u/X]e_1 : [u/X](\&(\exists T, \overline{\tau_i}, \overline{\omega_j}))$ and
  $[u/X]\Omega \mid [u/X]\Gamma, T, \overline{x_i : [u/X]\tau_i} \mid [u/X]\Sigma \vdash [u/X]e_2 : [u/X]\tau$.
  $[u/X](\&(\exists T, \overline{\tau_i}, \overline{\omega_j})) = \&(\exists T, \overline{[u/X]\tau_i}, \overline{[u/X]\omega_j})$ by definition of substitution.
  $[u/X]\Omega \mid [u/X]\Gamma \mid [u/X]\Sigma \vdash \mathsf{let}\ (T, \overline{x_i}) = \mathsf{unpack}\ [u/X]e_1\ \mathsf{in}\ [u/X]e_2 : [u/X]\tau$ by (t-unpack) as required.

- Case (t-pack): $\Omega \mid \Gamma, X \mid \Sigma \vdash \mathsf{pack}\ (\tau, \overline{e_i}, \overline{\gamma_i})\ \mathsf{as}\ \&(\exists T, \overline{\tau_i}, \overline{\gamma_j}) : \&(\exists T, \overline{\tau_i}, \overline{\gamma_j})$.
  Assume $T \neq X$ by bound variable renaming convention.
  $\overline{[u/X]\Omega \mid [u/X]\Gamma \mid [u/X]\Sigma \vdash [u/X]e_i : [u/X][\tau/T]\tau_i}$ by ind. hyp.
  $[u/X]\Omega \vdash [u/X]\gamma_j : [u/X][\tau/T]\omega_j$ by lemma A.8.
  $[u/X][\tau/T]\tau_i \equiv [[u/X]\tau/T][u/X]\tau_i$ and
  $[u/X][\tau/T]\omega_j \equiv [[u/X]\tau/T][u/X]\omega_j$ by definition of substitution.
  Then
  $[u/X]\Omega \mid [u/X]\Gamma \mid [u/X]\Sigma \vdash \mathsf{pack}\ ([u/X]\tau, \overline{[u/X]e_i}, \overline{[u/X]\gamma_j})\ \mathsf{as}\ \&(\exists T, \overline{[u/X]\tau_i}, \overline{[u/X]\omega_j})$
  $: \&(\exists T, \overline{[u/X]\tau_i}, \overline{[u/X]\omega_j})$
  by (t-pack) as required.

- Case (t-tapp): $\Omega \mid \Gamma, X \mid \Sigma \vdash e[\overline{u_i}] : [\overline{u_i/X_i}]\sigma$.
  Assume $X \notin \overline{X_i}$.
  By ind. hyp. $[u/X]\Omega \mid [u/X]\Gamma \mid [u/X]\Sigma \vdash [u/X]e : [u/X]\forall \overline{X_i}.\sigma$.
  $[u/X]\Omega \mid [u/X]\Gamma \mid [u/X]\Sigma \vdash [u/X]e[\overline{[u/X]u_i}] : [\overline{[u/X]u_i/X_i}][u/X]\tau$ by (t-tapp) as required.

- Case (t-tabs): $\Omega \mid \Gamma, X \mid \Sigma \vdash \Lambda \overline{X_i}.e : \forall \overline{X_i}.\tau$.
  Assume $X \notin \overline{X_i}$.
  By ind. hyp. $[u/X]\Omega \mid [u/X]\Gamma, \overline{X_i} \mid [u/X]\Sigma \vdash [u/X]e : [u/X]\tau$.
  By (t-tabs) $[u/X]\Omega \mid [u/X]\Gamma \mid [u/X]\Sigma \vdash \Lambda \overline{X_i}.[u/X]e : \forall \overline{X_i}.[u/X]\tau$.
  $\Lambda \overline{X_i}.[u/X]e \equiv [u/X](\Lambda \overline{X_i}.e)$ and $\forall \overline{X_i}.[u/X]\sigma \equiv [u/X](\forall \overline{X_i}.\tau)$ by def. of substitution as required.

- Case (t-capp): $\Omega \mid \Gamma, X \mid \Sigma \vdash e[\![\overline{\gamma_i}]\!] : \sigma$.
  $\Omega \mid \Gamma, [u/X]\Gamma' \mid \Sigma \vdash [u/X]e : [u/X](\forall \overline{\omega_i}.\sigma)$ by ind. hyp.
  $\overline{\Omega \vdash [u/X]\gamma_i : [u/X]\omega_i}$ by lemma A.8.
  $[u/X](\forall \overline{\omega_i}.\sigma) \equiv \forall \overline{[u/X]\omega_i}.[u/X]\sigma$ by def. of substitution.
  $\Omega \mid \Gamma, [u/X]\Gamma' \mid \Sigma \vdash [u/X]e[\![\overline{[u/X]\gamma_i}]\!]$ by (t-capp).
  $[u/X]e[\![\overline{[u/X]\gamma_i}]\!] \equiv [u/X](e[\![\overline{\gamma_i}]\!])$ by def. of substitution as required.

- Case (t-cabs): $\Omega \mid \Gamma, X \mid \Sigma \vdash \Lambda \overline{c_i : \omega_i}.e : \forall \overline{\omega_i}.\sigma$.
  $\Omega, \overline{c_i : [u/X]\omega_i} \mid \Gamma, [u/X]\Gamma' \mid \Sigma \vdash [u/X]e : [u/X]\sigma$ by ind. hyp.
  $\Omega \mid \Gamma, [u/X]\Gamma' \mid \Sigma \vdash \Lambda \overline{c_i : [u/X]\omega_i}.[u/X]e : \forall \overline{[u/X]\omega_i}.[u/X]\sigma$ by (t-cabs) as required.

- Case (t-coerce): $\Omega \mid \Gamma, X \mid \Sigma \vdash e \blacktriangleright \gamma : \sigma_2$.
  $[u/X]\Omega \mid [u/X]\Gamma \mid [u/X]\Sigma \vdash [u/X]e : [u/X]\sigma_1$ by ind. hyp.
  $[u/X]\Omega \vdash [u/X]\gamma : [u/X]\sigma_1 \sim [u/X]\sigma_2$ by lemma A.8.
  $[u/X]\Omega \mid [u/X]\Gamma \mid [u/X]\Sigma \vdash [u/X]e \blacktriangleright [u/X]\gamma : [u/X]\sigma_2$ by (t-coerce) as required.

- Case (t-loc): $\Omega \mid \Gamma, X \mid \Sigma \vdash l : \sigma$.
  Then $(l : [u/X]\sigma) \in [u/X]\Omega$
  and $[u/X]\Omega \mid [u/X]\Gamma \mid [u/X]\Sigma \vdash l : [u/X]\sigma$ by (t-loc) as required.

□

Lemma A.10 (Coercion substitution)**.** *If* $\Omega, c : \vartheta' \vdash \gamma : \vartheta$ *and* $\Omega \vdash \gamma' : \vartheta'$ *then* $\Omega \vdash [\gamma'/c]\gamma : \vartheta$.

Proof. Straightforward by induction on derivations $\Omega, c : \omega' \vdash \gamma : \vartheta$. □

Lemma A.11 (Coercion substitution in terms)**.** *If* $\Omega, c : \omega \mid \Gamma \mid \Sigma \vdash e : \sigma$ *and* $\Omega \vdash \gamma : \omega$ *then* $\Omega \mid \Gamma \mid \Sigma \vdash [\gamma/c]e : \sigma$.

Proof. By induction on derivations $\Omega, c : \omega \mid \Gamma \mid \Sigma \vdash e : \sigma$ assuming $\Omega \vdash \gamma : \omega$ in all cases.
Mostly straightforward use of induction hypothesis except in the following cases:

- Case (t-pack): $\Omega, c : \omega \mid \Gamma \mid \Sigma \vdash \mathsf{pack}\ (\tau, \overline{e_i}, \overline{\gamma_j})$ as $\&(\exists T, \overline{\tau_i}, \overline{\omega_j}) : \&(\exists T, \overline{\tau_i}, \overline{\omega_j})$ and $\Omega \vdash \gamma : \omega$.
  $\overline{\Omega \mid \Gamma \mid \Sigma \vdash [\gamma/c]e_i : [\tau/T]\tau_i}$ by ind. hyp.
  $\overline{\Omega \vdash [\gamma/c]\gamma_j : [\tau/T]\gamma_j}$ by lemma A.10.
  $\Omega \mid \Gamma \mid \Sigma \vdash \mathsf{pack}\ (\tau, [\gamma/c]\overline{e_i}, [\gamma/c]\overline{\gamma_j})$ as $\&(\exists T, \overline{\tau_i}, \overline{\omega_j}) : \&(\exists T, \overline{\tau_i}, \overline{\omega_j})$
  by (t-pack) as required.

- Case (t-capp): $\Omega, c : \omega \mid \Gamma \mid \Sigma \vdash e[\![\overline{\gamma_j}]\!] : \sigma$ and $\Omega \vdash \gamma : \omega$.
  $\Omega \mid \Gamma \mid \Sigma \vdash [\gamma/c]e : \forall \overline{\omega_j}.\sigma$ by ind. hyp.
  $\overline{\Omega \vdash [\gamma/c]\gamma_j : [\overline{u_i/X_i}]\omega_j}^{\,j}$ by lemma A.10.
  $\Omega \mid \Gamma \mid \Sigma \vdash [\gamma/c]e[\![\overline{[\gamma/c]\gamma_i}]\!] : \sigma$ by (t-capp) as required.

- Case (t-cabs): $\Omega, c : \omega \mid \Gamma \mid \Sigma \vdash \Lambda \overline{c_j : \omega_j}.e : \forall \overline{\omega_j}.\sigma$.
  $\Omega, \overline{c_j : \omega_j} \mid \Gamma \mid \Sigma \vdash [\gamma/c]e : \sigma$ by ind. hyp.
  We can assume $c \notin \overline{c_j}$ because of the bound variable renaming convention.
  Then $\Omega \mid \Gamma \mid \Sigma \vdash \Lambda \overline{c_j : \omega_j}.[\gamma/c]e : \forall \overline{\omega_j}.\sigma$ as required.

- Case (t-coerce): $\Omega, c : \omega \mid \Gamma \mid \Sigma \vdash e \blacktriangleright \gamma' : \sigma_2$.
  $\Omega \mid \Gamma \mid \Sigma \vdash [\gamma/c]e : \sigma_1$ by ind. hyp.
  $\Omega \vdash [\gamma/c]\gamma' : \sigma_1 \sim \sigma_2$ by lemma A.10.
  Then $\Omega \mid \Gamma \mid \Sigma \vdash [\gamma/c]e \blacktriangleright [\gamma/c]\gamma' : \sigma_2$ by (t-coerce) as required.

□

Lemma A.12 (Type soundness of alloc)**.** *Suppose* $\Omega \mid \Gamma \mid \Sigma \vdash v : \sigma$ *and* $\Omega \mid \Gamma \mid \Sigma \vdash \mu$.
If $\mathbf{alloc}(\mu, v) = \langle \mu', l \rangle$ *then there is some* $\Sigma' \supseteq \Sigma$ *such that*
$l \notin dom(\mu)$ *and* $\Omega \mid \Gamma \mid \Sigma' \vdash \mu'$ *and* $\Omega \mid \Gamma \mid \Sigma' \vdash l : \sigma$.

Proof. By structural induction on v:

- Case $v \in \textsc{StoreVal}$:
  Suppose $\Omega \mid \Gamma \mid \Sigma \vdash v : \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu$.
  By definition of alloc, $\mathbf{alloc}(\mu, v) = \langle \mu[l \mapsto v], l \rangle$ where $l \notin dom(\mu)$.
  Let $\Sigma' = \Sigma, l : \sigma$. Then $\Omega \mid \Gamma \mid \Sigma' \vdash l : \sigma$ by (t-loc) and $\Omega \mid \Gamma \mid \Sigma' \vdash \mu'$ as there is no w such that $w \sqsubset_\mu v$.

- Case $v = S \langle \overline{u_j} \rangle \{\overline{x_i : v_i}\}$:
  Suppose $\Omega \mid \Gamma \mid \Sigma \vdash S \langle \overline{u_j} \rangle \{\overline{x_i : v_i}\} : \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu_0$.
  By lemma $A.1$, there are some $\overline{X_j}, \overline{\sigma_i}$ such that
  $\overline{\Omega \mid \Gamma \mid \Sigma \vdash v_i : [\overline{u_j/X_j}]\sigma_i}$ and $S \langle \overline{X_j} \rangle \{\overline{x_i : \sigma_i}\} \in \Omega$ and $\sigma = S \langle \overline{u_j} \rangle$.
  By definition, $\mathbf{alloc}(\mu_0, S \langle \overline{u_j} \rangle \{\overline{x_i : v_i}\}) = \langle \mu', l \rangle$
  where $\overline{\langle \mu_i, l_i \rangle = \mathbf{alloc}(\mu_{i-1}, v_i)}^{\,i}$
  and $\mu' = \mu_i[l \mapsto S \langle \overline{u_j} \rangle \{\overline{x_i : l_i}\}]$ and $l \notin dom(\mu_i)$.
  Lemma A.5 gives us $\overline{\Omega \mid \Gamma \mid \Sigma' \vdash v_i : [\overline{u_j/X_j}]\sigma_i}$ for any $\Sigma' \supseteq \Sigma$.
  This allows us to use the induction hypothesis i times to get
  $\overline{l_i \notin dom(\mu_{i-1})}$ and $\overline{\Omega \mid \Gamma \mid \Sigma_i \vdash \mu_i}^{\,i}$ and $\overline{\Omega \mid \Gamma \mid \Sigma_i \vdash l_i : [\overline{u_j/X_j}]\sigma_i}^{\,i}$
  for some $\overline{\Sigma_i \supseteq \Sigma_{i-1}}^{\,i}$. Let $\Sigma' = \Sigma_i$.
  By lemma A.5, $\overline{\Omega \mid \Gamma \mid \Sigma' \vdash l_i : [\overline{u_j/X_j}]\sigma_i}$.
  Then by (t-newstruct) $\Omega \mid \Gamma \mid \Sigma' \vdash S \langle \overline{u_j} \rangle \{\overline{x_i : l_i}\} : S \langle \overline{u_j} \rangle$.
  Let $\Sigma'' = \Sigma', l : S \langle \overline{u_j} \rangle$. Then $\Omega \mid \Gamma \mid \Sigma'' \vdash l : S \langle \overline{u_j} \rangle$ by (t-loc).

$\overline{\mu'(l_i) \sqsubset_{\mu'} S\,\langle \overline{u_j} \rangle\,\{\overline{x_i : l_i}\}}^{\,i}$ by def. of $\sqsubset_{\mu'}$.
Since l is not mentioned in μ_i and $\sqsubset_{\mu_i}$ is well-founded,
we conclude that $\sqsubset_{\mu'}$ is well-founded and $\Omega \mid \Gamma \mid \Sigma'' \vdash \mu'$.

- Case $v = S\,\langle \overline{u_j} \rangle\,\{\overline{x_i : v_i}\} \blacktriangleright \gamma$:

  Suppose $\Omega \mid \Gamma \mid \Sigma \vdash S\,\langle \overline{u_j} \rangle\,\{\overline{x_i : v_i}\} \blacktriangleright \gamma : \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu_0$.

  By lemma $A.1$, there are some $\tau, \overline{X_j}, \overline{\sigma_i}$ such that:
  $\Omega \vdash \gamma : S\,\langle \overline{u_j} \rangle \sim \tau$ and
  $\overline{\Omega \mid \Gamma \mid \Sigma \vdash v_i : [\overline{u_j/X_j}]\sigma_i}$ and
  $S\,\langle \overline{X_j} \rangle\,\{\overline{x_i : \sigma_i}\} \in \Omega$ and
  $\sigma = \tau$.

  By definition, $\mathbf{alloc}(\mu_0, S\,\langle \overline{u_j} \rangle\,\{\overline{x_i : v_i}\} \blacktriangleright \gamma) = \langle \mu_i[l \mapsto S\,\langle \overline{u_j} \rangle\,\{\overline{x_i : l_i}\} \blacktriangleright \gamma], l \rangle$
  where $\overline{\langle \mu_i, l_i \rangle = \mathbf{alloc}(\mu_{i-1}, v_i)}^{\,i}$ and $l \notin dom(\mu_0)$.

  Lemma A.5 gives us $\overline{\Omega \mid \Gamma \mid \Sigma' \vdash v_i : [\overline{u_j/X_j}]\sigma_i}$ for any $\Sigma' \supseteq \Sigma$.

  This allows us to use the induction hypothesis i times to get
  $\overline{l_i \notin dom(\mu_{i-1})}$ and $\overline{\Omega \mid \Gamma \mid \Sigma_i \vdash \mu_i}^{\,i}$ and $\overline{\Omega \mid \Gamma \mid \Sigma_i \vdash l_i : [\overline{u_j/X_j}]\sigma_i}^{\,i}$
  for some $\overline{\Sigma_i \supseteq \Sigma_{i-1}}^{\,i}$. Let $\Sigma' = \Sigma_i$.

  By lemma A.5, $\overline{\Omega \mid \Gamma \mid \Sigma' \vdash l_i : [\overline{u_j/X_j}]\sigma_i}$.

  Then by (t-newstruct) $\Omega \mid \Gamma \mid \Sigma' \vdash S\,\langle \overline{u_j} \rangle\,\{\overline{x_i : l_i}\} : S\,\langle \overline{u_j} \rangle$
  and by (t-coerce) $\Omega \mid \Gamma \mid \Sigma' \vdash S\,\langle \overline{u_j} \rangle\,\{\overline{x_i : l_i}\} \blacktriangleright \gamma : S\,\langle \overline{u_j} \rangle$.

  Let $\Sigma'' = \Sigma', l : \tau$. Then $\Omega \mid \Gamma \mid \Sigma'' \vdash l : \tau$ by (t-loc)
  and $\Omega \mid \Gamma \mid \Sigma'' \vdash \mu_i[l \mapsto S\,\langle \overline{u_j} \rangle\,\{\overline{x_i : l_i}\} \blacktriangleright \gamma]$
  by the same argument as in the previous case.

□

Lemma A.13 (Lifting)**.** *If* $\overline{\Omega \vdash \gamma_i : u_i \sim u'_i}$ *then* $\Omega \vdash [\overline{\gamma_i/X_i}]\eta : [\overline{u_i/X_i}]\eta \sim [\overline{u'_i/X_i}]\eta$.

Proof. By structural induction on u supposing in all cases that $\overline{\Omega \vdash \gamma_i : u_i \sim u'_i}$:

- Case $\eta = ()$:

  $\Omega \vdash () : () \sim ()$ by (co-unit).

- Case $\eta = X$:

  If $X \notin \overline{X_i}$ then the case holds by (co-tvar).

  Otherwise, $[\gamma/X] \in [\overline{\gamma_i/X_i}]$ and $\Omega \vdash \gamma : u_1 \sim u_2$ for some γ, u_1, u_2.

  Then $\Omega \vdash [\overline{\gamma_i/X_i}]\gamma : [\overline{u_i/X_i}]u_1 \sim [\overline{u'_i/X_i}]u_2$ as required.

- Case $\eta = A\,\langle\overline{u_j}\rangle$:

  By ind. hyp. $\overline{\Omega \vdash [\overline{\gamma_i/X_i}]u_j : [\overline{u_i/X_i}]u_j \sim [\overline{u'_i/X_i}]u_j}^{\,j}$.

  Then by (co-atype) and def. of substitution
  $\Omega \vdash [\overline{\gamma_i/X_i}]A\,\langle\overline{u_j}\rangle : [\overline{\gamma_i/X_i}]A\,\langle\overline{u_j}\rangle \sim [\overline{\gamma_i/X_i}]A\,\langle\overline{u_j}\rangle$ as required.

- Case $\eta = (\exists T, \overline{\tau_k}, \overline{\omega_j})$:

  Straightforward use of the induction hypothesis as in the previous case.

- Case $\eta = \mathsf{fn}(\overline{\tau_i}) \to \tau$

  Straightforward use of the induction hypothesis as in the previous case.

- Case $\eta = \&u'$:

  Straightforward use of the induction hypothesis as in the previous case.

- Case $\eta = S\,\langle\overline{u_i}\rangle$:

  Straightforward use of the induction hypothesis as in the previous case.

- Case $\eta = \forall\overline{X_j}.\eta$:

  Straightforward use of the induction hypothesis as in the previous case.

- Case $\eta = \forall\overline{\omega_j}.\eta$:

  Straightforward use of the induction hypothesis as in the previous case.

□

Lemma A.14 (Value-preserving coercion)**.** *If* $\Omega \mid \Gamma \mid \Sigma \vdash v \blacktriangleright \gamma : \sigma$
then $v \blacktriangleright\!\blacktriangleright \gamma = v'$ *for some* v' *where* $\Omega \mid \Gamma \mid \Sigma \vdash v' : \sigma$.

Proof. By cases on v:

- Case $v = pv$:
  Then by def. $pv \blacktriangleright\!\blacktriangleright \gamma = pv \blacktriangleright \gamma$ and $pv \blacktriangleright \gamma \in \textsc{Val}$.

- Case $v = pv \blacktriangleright \gamma'$:
  Then by def. $pv \blacktriangleright \gamma' \blacktriangleright\!\blacktriangleright \gamma = pv \blacktriangleright (\gamma' \circ \gamma)$.
  and $pv \blacktriangleright (\gamma' \circ \gamma) \in \textsc{Val}$.
  From lemma A.1 we get $\Omega \vdash \gamma : \sigma' \sim \sigma$ and $\Omega \vdash \gamma' : \sigma'' \sim \sigma'$ and $\Omega \mid \Gamma \mid \Sigma \vdash pv : \sigma''$.
  Then by (co-trans) $\Omega \vdash \gamma' \circ \gamma : \sigma'' \sim \sigma$
  and by (t-coerce) $\Omega \mid \Gamma \mid \Sigma \vdash pv \blacktriangleright (\gamma' \circ \gamma) : \sigma$ as required.

□

Lemma A.15 (Type soundness of **update**). *Suppose* $\Omega \mid \Gamma \mid \Sigma \vdash v : \sigma$ *and* $\Omega \mid \Gamma \mid \Sigma \vdash l : \sigma$ *and* $\Omega \mid \Gamma \mid \Sigma \vdash \mu$.
Then $\mathbf{update}_\Omega(\mu, l, v) = \mu'$ *where* $\Omega \mid \Gamma \mid \Sigma \vdash \mu'$.

Proof. By structural induction on v:

- Case $v = S\,\langle \overline{u_j} \rangle\,\{\overline{x_i : v_i}\}$:

  Suppose $\Omega \mid \Gamma \mid \Sigma \vdash S\,\langle \overline{u_j} \rangle\,\{\overline{x_i : v_i}\} : \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash l : \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu_0$.

  By lemma A.1, there are some $\overline{X_j}, \overline{\sigma_i}$ such that
  $\overline{\Omega \mid \Gamma \mid \Sigma \vdash v_i : [\overline{u_j/X_j}]\sigma_i}$ and $S\,\langle \overline{X_j} \rangle\,\{\overline{x_i : \sigma_i}\} \in \Omega$ and $\sigma = S\,\langle \overline{u_j} \rangle$.

  From $\Omega \mid \Gamma \mid \Sigma \vdash \mu_0$ and $\Omega \mid \Gamma \mid \Sigma \vdash l : S\,\langle \overline{u_j} \rangle$ we can deduce $\Omega \mid \Gamma \mid \Sigma \vdash \mu_0(l) : S\,\langle \overline{u_j} \rangle$.

  From lemma A.3 we know that either
  $\mu_0(l) = S\,\langle \overline{u_j} \rangle\,\{\overline{x_i : l_i}\}$ for some l_i or
  $\mu_0(l) = S\,\langle \overline{u'_j} \rangle\,\{\overline{x_i : l_i}\} \blacktriangleright \gamma$ for some $\gamma, \overline{l_i}, \overline{u'_j}$ where $\Omega \vdash \gamma : S\,\langle \overline{u'_j} \rangle \sim S\,\langle \overline{u_j} \rangle$.

  We must consider both cases separately:

  - Case $\mu_0(l) = S\,\langle \overline{u_j} \rangle\,\{\overline{x_i : l_i}\}$:

    Then by definition of **update**,
    $\mathbf{update}_\Omega(\mu_0, l, S\,\langle \overline{u_j} \rangle\,\{\overline{x_i : v_i}\}) = \mu_i$ where $\overline{\mu_i = \mathbf{update}_\Omega(\mu_{i-1}, l_i, v_i)}$.
    By lemma A.1, $\overline{\Omega \mid \Gamma \mid \Sigma \vdash l_i : [u_j/X_j]\sigma_i}$
    and by ind. hyp. $\Omega \mid \Gamma \mid \Sigma \vdash \mu_i$ as required.

  - Case $\mu_0(l) = S\,\langle \overline{u'_j} \rangle\,\{\overline{x_i : l_i}\} \blacktriangleright \gamma$:

    Then, by definition of **update**,
    $\mathbf{update}_\Omega(\mu_0, l, S\,\langle \overline{u_j} \rangle\,\{\overline{x_i : v_i}\}) = \mu_i$
    where $\overline{\mu_i = \mathbf{update}_\Omega(\mu_{i-1}, l_i, v_i \blacktriangleright\!\blacktriangleright [\overline{\gamma_j/X_j}]\sigma_i)}$
    and $\overline{\gamma_j = \mathsf{spar}\ (j, \mathsf{sym}\ \gamma)}$.
    By lemma A.1, $\Omega \mid \Gamma \mid \Sigma \vdash S\,\langle \overline{u'_j} \rangle\,\{\overline{x_i : l_i}\} : S\,\langle \overline{u'_j} \rangle$
    and $\overline{\Omega \mid \Gamma \mid \Sigma \vdash l_i : [u'_j/X_j]\sigma_i}$.
    By (co-sym) and (co-spar), $\overline{\Omega \vdash \gamma_j : u_j \sim u'_j}$.
    By lemma A.13, $\overline{\Omega \mid \Gamma \mid \Sigma \vdash v_i \blacktriangleright [\overline{\gamma_j/X_j}]\sigma_i : [\overline{u'_j/X_j}]\sigma_i}$
    and by lemma A.14, $\overline{\Omega \mid \Gamma \mid \Sigma \vdash v_i \blacktriangleright\!\blacktriangleright [\overline{\gamma_j/X_j}]\sigma_i : [\overline{u'_j/X_j}]\sigma_i}$ where $\overline{v_i \blacktriangleright\!\blacktriangleright [\overline{\gamma_j/X_j}]\sigma_i \in \textsc{Val}}$.
    Then, by the induction hypothesis, $\Omega \mid \Gamma \mid \Sigma \vdash \mu_i$ as required.

- Case $v = S\,\langle \overline{u_j} \rangle\,\{\overline{x_i : v_i}\} \blacktriangleright \gamma$:

  Suppose $\Omega \mid \Gamma \mid \Sigma \vdash S\,\langle \overline{u_j} \rangle\,\{\overline{x_i : v_i}\} \blacktriangleright \gamma : \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash l : \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu_0$.

From lemma A.1 we get $\Omega \vdash \gamma : S\,\langle \overline{u_j} \rangle \sim \sigma$ as well as
$\overline{\Omega \mid \Gamma \mid \Sigma \vdash v_i : [\overline{u_j/X_j}]\sigma_i}$ and $S\,\langle \overline{X_j} \rangle\,\{\overline{x_i : \sigma_i}\} \in \Omega$
for some $\overline{X_j}, \overline{\sigma_i}$.

From $\Omega \mid \Gamma \mid \Sigma \vdash \mu_0$ and $\Omega \mid \Gamma \mid \Sigma \vdash l : \sigma$ we can deduce $\Omega \mid \Gamma \mid \Sigma \vdash \mu_0(l) : \sigma$.

By lemma A.3, since $\Omega \vdash \mathsf{sym}\ \gamma : \sigma \sim S\,\langle \overline{u_j} \rangle$ by (co-sym), either
$\mu_0(l) = S\,\langle \overline{u'_j} \rangle\,\{\overline{x_i : l_i}\}$ and $\sigma = S\,\langle \overline{u'_j} \rangle$ for some $\overline{u'_j}, \overline{l_i}$ or
$\mu_0(l) = S\,\langle \overline{u'_j} \rangle\,\{\overline{x_i : l_i}\} \blacktriangleright \gamma'$ where $\Omega \vdash \gamma' : S\,\langle \overline{u'_j} \rangle \sim \sigma$ for some $\overline{u'_j}, \overline{l_i}$.

We consider both cases separately:

- Case $\mu_0(l) = S\,\langle \overline{u'_j} \rangle\,\{\overline{x_i : l_i}\}$:
  Then by definition of update,
  $\mathbf{update}_\Omega(\mu_0, l, S\,\langle \overline{u_j} \rangle\,\{\overline{x_i : v_i}\} \blacktriangleright \gamma) = \mu_i$
  where $\overline{\mu_i = \mathbf{update}_\Omega(\mu_{i-1}, l_i, v_i \blacktriangleright\!\blacktriangleright [\overline{\gamma_j/X_j}]\sigma_i)}$
  and $\overline{\gamma_j = \mathsf{spar}\ (j, \gamma)}$.
  From lemma A.1 we get $\overline{\Omega \mid \Gamma \mid \Sigma \vdash l_i : [\overline{u'_j/X_j}]\sigma_i}$.
  Using (co-spar) we get $\overline{\Omega \vdash \gamma_j : u_j \sim u'_j}$.
  By lemma A.13, $\overline{\Omega \mid \Gamma \mid \Sigma \vdash v_i \blacktriangleright [\overline{\gamma_j/X_j}]\sigma_i : [\overline{u'_j/X_j}]\sigma_i}$
  and by lemma A.14, $\overline{\Omega \mid \Gamma \mid \Sigma \vdash v_i \blacktriangleright\!\blacktriangleright [\overline{\gamma_j/X_j}]\sigma_i : [\overline{u'_j/X_j}]\sigma_i}$ where $\overline{v_i \blacktriangleright\!\blacktriangleright [\overline{\gamma_j/X_j}]\sigma_i \in \textsc{Val}}$.
  Then, by the induction hypothesis, $\Omega \mid \Gamma \mid \Sigma \vdash \mu_i$ as required.
- Case $\mu_0(l) = S\,\langle \overline{u'_j} \rangle\,\{\overline{x_i : l_i}\} \blacktriangleright \gamma'$:
  Then by definition of update,
  $\mathbf{update}_\Omega(\mu_0, l, S\,\langle \overline{u_j} \rangle\,\{\overline{x_i : v_i}\} \blacktriangleright \gamma) = \mu_i$
  where $\overline{\mu_i = \mathbf{update}_\Omega(\mu_{i-1}, l_i, v_i \blacktriangleright\!\blacktriangleright [\overline{\gamma_j/X_j}]\sigma_i)}$
  and $\overline{\gamma_j = \mathsf{spar}\ (j, \gamma \circ \mathsf{sym}\ \gamma')}$.
  From lemma A.1 we get $\overline{\Omega \mid \Gamma \mid \Sigma \vdash l_i : [\overline{u'_j/X_j}]\sigma_i}$.
  Using (co-sym), (co-trans) and (co-spar) we get $\overline{\Omega \vdash \gamma_j : u_j \sim u'_j}$.
  By lemma A.13, $\overline{\Omega \mid \Gamma \mid \Sigma \vdash v_i \blacktriangleright [\overline{\gamma_j/X_j}]\sigma_i : [\overline{u'_j/X_j}]\sigma_i}$
  and by lemma A.14, $\overline{\Omega \mid \Gamma \mid \Sigma \vdash v_i \blacktriangleright\!\blacktriangleright [\overline{\gamma_j/X_j}]\sigma_i : [\overline{u'_j/X_j}]\sigma_i}$ where $\overline{v_i \blacktriangleright\!\blacktriangleright [\overline{\gamma_j/X_j}]\sigma_i \in \textsc{Val}}$.
  Then, by the induction hypothesis, $\Omega \mid \Gamma \mid \Sigma \vdash \mu_i$ as required.

- Case $v \in \textsc{PVal}$:
  Then $\mu' = \mu[l \mapsto v]$.
  $\Omega \mid \Gamma \mid \Sigma \vdash v : \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash l : \sigma$ by assumption
  and there is no possible $w \sqsubset_{\mu'} v$ which lets us conclude $\Omega \mid \Gamma \mid \Sigma \vdash \mu'$.

□

Lemma A.16 (Type soundness of **getValue**). *If* $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ *and* $\Sigma(l) = \sigma$ *then* $\Omega \mid \Gamma \mid \Sigma \vdash \mathbf{getValue}(\mu, l) : \sigma$.

Proof. From the def. of $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ we know that $\sqsubset_\mu$ is a well founded relation on the elements in the image of μ. We then prove the lemma by induction on l assuming the induction hypothesis for all l' where $\mu(l') \sqsubset_\mu \mu(l)$.

- Case $\mu(l) = S\,\langle\overline{u_j}\rangle\,\{\overline{x_i : l_i}\}$:

  Suppose $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and $\Omega \mid \Gamma \mid \Sigma \vdash l : \sigma$.

  Then $S\,\langle\overline{u_j}\rangle\,\{\overline{x_i : l_i}\} : \sigma$.

  From lemma A.1 we get
  $\dfrac{S\,\langle\overline{X_j}\rangle\,\{\overline{x_i : \sigma_i}\} \in \Omega \text{ and}}{\Omega \mid \Gamma \mid \Sigma \vdash l_i : [\overline{u_j/X_j}]\sigma_i}$ and
  $\sigma = S\,\langle\overline{u_j}\rangle$ for some $\overline{X_j}, \sigma_i$.

  By def., $\mathbf{getValue}(\mu, l) = S\,\langle\overline{u_j}\rangle\,\{\overline{x_i : \mathbf{getValue}(\mu, l_i)}\}$.

  By ind. hyp. $\overline{\Omega \mid \Gamma \mid \Sigma \vdash \mathbf{getValue}(\mu, l_i) : [\overline{u_j/X_j}]\sigma_i}^{\,i}$.

  Then by (t-newstruct), $\Omega \mid \Gamma \mid \Sigma \vdash S\,\langle\overline{u_j}\rangle\,\{\overline{x_i : \mathbf{getValue}(\mu, l_i)}\} : S\,\langle\overline{u_j}\rangle$ as required.

- Case $\mu(l) = S\,\langle\overline{u_j}\rangle\,\{\overline{x_i : l_i}\} \blacktriangleright \gamma$:

  Suppose $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and $\Omega \mid \Gamma \mid \Sigma \vdash l : \sigma$.

  Then $S\,\langle\overline{u_j}\rangle\,\{\overline{x_i : l_i}\} \blacktriangleright \gamma : \sigma$.

  From lemma A.1 we get
  $\Omega \vdash \gamma : S\,\langle\overline{u_j}\rangle \sim \sigma$ and
  $\Omega \mid \Gamma \mid \Sigma \vdash S\,\langle\overline{u_j}\rangle\,\{\overline{x_i : l_i}\}$ and
  $\dfrac{S\,\langle\overline{X_j}\rangle\,\{\overline{x_i : \sigma_i}\} \in \Omega \text{ and}}{\Omega \mid \Gamma \mid \Sigma \vdash l_i : [\overline{u_j/X_j}]\sigma_i}$
  for some $\overline{X_j}, \sigma_i$.

  By def. $\mathbf{getValue}(\mu, l) = S\,\langle\overline{u_j}\rangle\,\{\overline{x_i : \mathbf{getValue}(\mu, l_i)}\}$.

  By ind. hyp. $\overline{\Omega \mid \Gamma \mid \Sigma \vdash \mathbf{getValue}(\mu, l_i) : [\overline{u_j/X_j}]\sigma_i}^{\,i}$.

  Then by (t-newstruct) and (t-coerce) $\Omega \mid \Gamma \mid \Sigma \vdash S\,\langle\overline{u_j}\rangle\,\{\overline{x_i : \mathbf{getValue}(\mu, l_i)}\} \blacktriangleright \gamma : \sigma$ as required.

- Case $\mu(l) = w$ where w matches neither of the above forms:

  $\mathbf{getValue}(\mu, l) = \mu(l)$ as required.

□

Theorem 6.1 (Preservation)**.** *Suppose Ω is a consistent environment.*

1. *If $\Omega \mid \Gamma \mid \Sigma \vdash e : \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and $\langle \mu, e\rangle \xrightarrow{\Omega} \langle \mu', e'\rangle$*
   then, there is some store typing $\Sigma' \supseteq \Sigma$ such that
   $\Omega \mid \Gamma \mid \Sigma' \vdash e' : \sigma$ and $\Omega \mid \Gamma \mid \Sigma' \vdash \mu'$.

2. *If $\Omega \mid \Gamma \mid \Sigma \vdash lv : \tau$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and $\langle \mu, lv\rangle \xrightarrow{lv(\Omega)} \langle \mu', lv'\rangle$*
   then, there is some store typing $\Sigma' \supseteq \Sigma$ such that
   $\Omega \mid \Gamma \mid \Sigma' \vdash lv' : \tau$ and $\Omega \mid \Gamma \mid \Sigma' \vdash \mu'$.

Proof. By simultaneous induction on derivations $\langle \mu, e\rangle \xrightarrow{\Omega} \langle \mu', e'\rangle$ and $\langle \mu, lv\rangle \xrightarrow{lv(\Omega)} \langle \mu', lv'\rangle$.

- Case (e-let1):

  Suppose $\Omega \mid \Gamma \mid \Sigma \vdash \mathsf{let}\ x : \tau = v\ \mathsf{in}\ e_2 : \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and
  $\langle \mu, \mathsf{let}\ x : \tau = v\ \mathsf{in}\ e_2\rangle \xrightarrow{\Omega} \langle \mu', [l/x]e_2\rangle$
  where $\langle \mu', l\rangle = \mathbf{alloc}(\mu, v)$

  From lemma A.1 we know that $\Omega \mid \Gamma \mid \Sigma \vdash v : \tau$ and $\Omega \mid \Gamma, x : \tau \mid \Sigma \vdash e_2 : \tau_2$ and $\sigma = \tau_2$ for some τ_2.

  From lemma A.12 we know that $\Omega \mid \Gamma \mid \Sigma' \vdash \mu'$ and $\Omega \mid \Gamma \mid \Sigma' \vdash l : \tau$ for some $\Sigma' \supseteq \Sigma$.

  We can then use lemma A.6 to conclude that $\Omega \mid \Gamma \mid \Sigma' \vdash [l/x]e_2 : \tau_2$

- Case (e-let2):

  Suppose $\Omega \mid \Gamma \mid \Sigma \vdash \mathsf{let}\ x : \tau = e\ \mathsf{in}\ e_2 : \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and
  $\langle \mu, \mathsf{let}\ x : \tau = e\ \mathsf{in}\ e_2\rangle \xrightarrow{\Omega} \langle \mu', \mathsf{let}\ x : \tau = e'\ \mathsf{in}\ e_2\rangle$
  where $\langle \mu, e\rangle \xrightarrow{\Omega} \langle \mu', e'\rangle$.

  $\Omega \mid \Gamma \mid \Sigma \vdash e : \tau$ and $\Omega \mid \Gamma, x : \tau \mid \Sigma \vdash e_2 : \tau_2$ and $\sigma = \tau_2$ for some τ_2 by lemma A.1.

  $\Omega \mid \Gamma \mid \Sigma' \vdash e' : \tau$ and $\Omega \mid \Gamma \mid \Sigma' \vdash \mu'$. for some $\Sigma' \supseteq \Sigma$ by ind. hyp.

  $\Omega \mid \Gamma \mid \Sigma' \vdash \mathsf{let}\ x : \tau = e'\ \mathsf{in}\ e_2 : \tau_2$ by lemma A.5 and (t-let) as required.

- Case (e-upd1):

  Suppose $\Omega \mid \Gamma \mid \Sigma \vdash lv := e : \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and
  $\langle \mu, lv := e\rangle \xrightarrow{\Omega} \langle \mu', lv' := e\rangle$
  where $\langle \mu, lv\rangle \xrightarrow{lv(\Omega)} \langle \mu', lv'\rangle$.

  By lemma A.1 there is some τ such that $\Omega \mid \Gamma \mid \Sigma \vdash lv : \tau$ and $\Omega \mid \Gamma \mid \Sigma \vdash e : \tau$ and $\sigma = ()$.

  By ind. hyp. there is some $\Sigma' \supseteq \Sigma$ such that $\Omega \mid \Gamma \mid \Sigma' \vdash \mu'$ and $\Omega \mid \Gamma \mid \Sigma' \vdash lv' : \tau$.

  Then $\Omega \mid \Gamma \mid \Sigma' \vdash lv' := e : ()$ by lemma A.5 and (t-upd) as required.

- Case (e-upd2):

  Suppose $\Omega \mid \Gamma \mid \Sigma \vdash l := v : \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and
  $\langle \mu, l := v \rangle \xrightarrow{\Omega} \langle \mu', () \rangle$
  where $\mu' = \mathbf{update}_{\Omega}(\mu, l, v)$.

  By lemma A.1 there is some τ such that $\Omega \mid \Gamma \mid \Sigma \vdash l : \tau$ and $\Omega \mid \Gamma \mid \Sigma \vdash e : \tau$ and $\sigma = ()$.

  $\Omega \mid \Gamma \mid \Sigma \vdash \mu'$ by lemma A.15.

  By (t-unit) $\Omega \mid \Gamma \mid \Sigma \vdash () : ()$ as required.

- Case (e-upd3):

  Suppose $\Omega \mid \Gamma \mid \Sigma \vdash (l \blacktriangleright \gamma) := e : \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and
  $\langle \mu, (l \blacktriangleright \gamma) := e \rangle \xrightarrow{\Omega} \langle \mu, l := (e \blacktriangleright \mathsf{sym}\ \gamma) \rangle$.

  By lemma A.1 there is some τ such that $\Omega \mid \Gamma \mid \Sigma \vdash l \blacktriangleright \gamma : \tau$ and $\Omega \mid \Gamma \mid \Sigma \vdash e : \tau$ and $\sigma = ()$.

  Using lemma A.1 again with corollary A.1 we get $\Omega \mid \Gamma \mid \Sigma \vdash l : \tau'$ and $\Omega \vdash \gamma : \tau' \sim \tau$ for some type τ'.

  Using (co-sym) we get $\Omega \vdash \mathsf{sym}\ \gamma : \tau \sim \tau'$.

  Using (t-coerce) we get $\Omega \mid \Gamma \mid \Sigma \vdash e \blacktriangleright \mathsf{sym}\ \gamma : \tau'$.

  Finally, using (t-upd) we can prove $\Omega \mid \Gamma \mid \Sigma \vdash l := (e \blacktriangleright \mathsf{sym}\ \gamma) : ()$ as required.

- Case (e-upd4):

  Suppose $\Omega \mid \Gamma \mid \Sigma \vdash l := e : \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and
  $\langle \mu, l := e \rangle \xrightarrow{\Omega} \langle \mu', l := e' \rangle$
  where $\langle \mu, e \rangle \xrightarrow{\Omega} \langle \mu', e' \rangle$.

  By lemma A.1 there is some τ such that $\Omega \mid \Gamma \mid \Sigma \vdash l : \tau$ and $\Omega \mid \Gamma \mid \Sigma \vdash e : \tau$ and $\sigma = ()$.

  By ind. hyp. $\Omega \mid \Gamma \mid \Sigma' \vdash \mu'$ and $\Omega \mid \Gamma \mid \Sigma' \vdash e' : \tau$ for some $\Sigma' \supseteq \Sigma$.

  By lemma A.5 and (t-upd) $\Omega \mid \Gamma \mid \Sigma' \vdash l := e' : ()$ as required.

- Case (e-loc):

  Suppose $\Omega \mid \Gamma \mid \Sigma \vdash l : \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and
  $\langle \mu, l \rangle \xrightarrow{\Omega} \langle \mu, v \rangle$
  where $v = \mathbf{getValue}(\mu, l)$.

  From lemma A.1 we know $\Sigma(l) = \sigma$.

  From lemma A.16 we get $\Omega \mid \Gamma \mid \Sigma \vdash v : \sigma$ as required.

- Case (e-deref1):

Suppose $\Omega \mid \Gamma \mid \Sigma \vdash *\&l : \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and
$\langle \mu, *\&l \rangle \xrightarrow{\Omega} \langle \mu, l \rangle$.

Using lemma A.1 we get that $\Omega \mid \Gamma \mid \Sigma \vdash \&l : \&\tau$ and $\sigma = \tau$ for some τ.

Using lemma A.1 again we get $\Omega \mid \Gamma \mid \Sigma \vdash l : \tau$ as required.

- Case (e-deref2):

  Suppose $\Omega \mid \Gamma \mid \Sigma \vdash *(\&l \blacktriangleright \gamma) : \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and
  $\langle \mu, *(\&l \blacktriangleright \gamma) \rangle \xrightarrow{\Omega} \langle \mu', l \blacktriangleright *\gamma \rangle$.

  Using lemma A.1 we get $\Omega \mid \Gamma \mid \Sigma \vdash \&l \blacktriangleright \gamma : \&\tau$ and $\sigma = \tau$ for some τ.

  Using lemma A.1 again we get $\Omega \mid \Gamma \mid \Sigma \vdash \&l : \tau'$ and $\Omega \vdash \gamma : \tau' \sim \&\tau$ for some τ'.

  Another use of lemma A.1 gives us $\Omega \mid \Gamma \mid \Sigma \vdash l : \tau''$ and $\tau' = \&\tau''$ for some τ''.

  By (co-deref) we get $\Omega \mid \Gamma \mid \Sigma \vdash *\gamma : \tau'' \sim \tau$.

  By (t-coerce) we can prove $\Omega \mid \Gamma \mid \Sigma \vdash l \blacktriangleright *\gamma : \tau$ as required.

- Case (e-deref3):

  Suppose $\Omega \mid \Gamma \mid \Sigma \vdash *e : \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and
  $\langle \mu, *e \rangle \xrightarrow{\Omega} \langle \mu', *e' \rangle$
  where $\langle \mu, e \rangle \xrightarrow{\Omega} \langle \mu', e' \rangle$.

  From lemma A.1 we get $\Omega \mid \Gamma \mid \Sigma \vdash e : \&\tau$ and $\sigma = \tau$ for some τ.

  By ind. hyp. $\Omega \mid \Gamma \mid \Sigma' \vdash e' : \&\tau$ and $\Omega \mid \Gamma \mid \Sigma' \vdash \mu'$ for some $\Sigma' \supseteq \Sigma$.

  By (t-deref) $\Omega \mid \Gamma \mid \Sigma' \vdash *e' : \tau$ as required.

- Case (e-ref1):

  Suppose $\Omega \mid \Gamma \mid \Sigma \vdash \&lv : \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and
  $\langle \mu, \&lv \rangle \xrightarrow{\Omega} \langle \mu', \&lv' \rangle$
  where $\langle \mu, lv \rangle \xrightarrow{lv(\Omega)} \langle \mu', lv' \rangle$.

  From lemma A.1 we get $\Omega \mid \Gamma \mid \Sigma \vdash lv : \tau$ and $\sigma = \&\tau$ for some τ.

  By ind. hyp. $\Omega \mid \Gamma \mid \Sigma' \vdash lv' : \tau$ and $\Omega \mid \Gamma \mid \Sigma' \vdash \mu'$ for some $\Sigma' \supseteq \Sigma$.

  With rule (t-ref) we can prove $\Omega \mid \Gamma \mid \Sigma' \vdash \&lv' : \&\tau$ as required.

- Case (e-ref2):

  Suppose $\Omega \mid \Gamma \mid \Sigma \vdash \&(l \blacktriangleright \gamma) : \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and
  $\langle \mu, \&(l \blacktriangleright \gamma) \rangle \xrightarrow{\Omega} \langle \mu, \&l \blacktriangleright \&\gamma \rangle$.

  From lemma A.1 we get $\Omega \mid \Gamma \mid \Sigma \vdash l \blacktriangleright \gamma : \tau$ and $\sigma = \&\tau$ for some τ.

Using lemma A.1 again we get $\Omega \mid \Gamma \mid \Sigma \vdash l : \tau'$ and $\Omega \vdash \gamma : \tau' \sim \tau$ for some type τ.

By (co-ref) we get $\Omega \vdash \&\gamma : \&\tau' \sim \&\tau$.

By (t-ref) we get $\Omega \mid \Gamma \mid \Sigma \vdash \&l : \&\tau'$.

Finally, by (t-coerce) we get $\Omega \mid \Gamma \mid \Sigma \vdash \&l \blacktriangleright \&\gamma : \&\tau$ as required.

- Case (e-fapp1):

  Suppose $\Omega \mid \Gamma \mid \Sigma \vdash \mathsf{fn}\ (\overline{x_i : \tau_i})\{e\}(\overline{v_i}) : \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and
  $\langle \mu, \mathsf{fn}\ (\overline{x_i : \tau_i})\{e\}(\overline{v_i})\rangle \xrightarrow{\Omega} \langle \mu_i, [\overline{l_i/x_i}]e\rangle$
  where $\mu_0 = \mu$ and $\overline{\langle \mu_i, l_i\rangle = \mathbf{alloc}(\mu_{i-1}, v_i)}^{\,i}$.

  From lemma A.1 we get $\Omega \mid \Gamma \mid \Sigma \vdash \mathsf{fn}\ (\overline{x_i : \tau_i})\{e\} : \mathsf{fn}(\overline{\tau_i'}) \to \tau$ and $\overline{\Omega \mid \Gamma \mid \Sigma \vdash v_i : \tau_i'}$ and $\sigma = \tau$ for some τ and $\overline{\tau_i'}$.

  Using lemma A.1 again we get $\Omega \mid \Gamma, \overline{x_i : \tau_i} \mid \Sigma \vdash e : \tau$ and $\overline{\tau_i} = \overline{\tau_i'}$.

  From lemma A.12 we get $\Omega \mid \Gamma \mid \Sigma' \vdash \mu_i$ and with lemma A.5 we have $\overline{\Omega \mid \Gamma \mid \Sigma' \vdash l_i : \tau_i}$ for some $\Sigma' \supseteq \Sigma$.

  Then, using lemmas A.5 and A.6 we get $\Omega \mid \Gamma \mid \Sigma' \vdash [\overline{l_i/x_i}]e : \tau$ as required.

- Case (e-fapp2):

  Suppose $\Omega \mid \Gamma \mid \Sigma \vdash (\mathsf{fn}\ (\overline{x_i : \tau_i})\{e\} \blacktriangleright \gamma)(\overline{e_i}) : \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and
  $\langle \mu, (\mathsf{fn}\ (\overline{x_i : \tau_i})\{e\} \blacktriangleright \gamma)(\overline{e_i})\rangle \xrightarrow{\Omega} \langle \mu, \mathsf{fn}\ (\overline{x_i : \tau_i})\{e\}(\overline{e_i \blacktriangleright \gamma_i}) \blacktriangleright \gamma_R\rangle$
  where $\overline{\gamma_i = \mathsf{sym}\ (\mathsf{farg}\ (i, \gamma))}$ and $\gamma_R = \mathsf{fret}\ \gamma$.

  From lemma A.1 we get $\Omega \mid \Gamma \mid \Sigma \vdash \mathsf{fn}\ (\overline{x_i : \tau_i})\{e\} \blacktriangleright \gamma : \mathsf{fn}(\overline{\tau_i'}) \to \tau$ and $\overline{\Omega \mid \Gamma \mid \Sigma \vdash e_i : \tau_i'}$ and $\sigma = \tau$
  for some τ and $\overline{\tau_i'}$.

  Using lemma A.1 again we get $\Omega \vdash \gamma : \mathsf{fn}(\overline{\tau_i''}) \to \tau' \sim \mathsf{fn}(\overline{\tau_i'}) \to \tau$
  and $\Omega \mid \Gamma \mid \Sigma \vdash \mathsf{fn}\ (\overline{x_i : \tau_i})\{e\} : \mathsf{fn}(\overline{\tau_i''}) \to \tau'$.

  Yet another use of lemma A.1 gives us $\Omega \mid \Gamma, \overline{x_i : \tau_i} \mid \Sigma \vdash e : \tau'$ and $\overline{\tau_i = \tau_i''}$.

  Using (co-farg) and (co-sym) we get $\overline{\Omega \vdash \gamma_i : \tau_i' \sim \tau_i}$.

  Using (co-fret) we get $\Omega \vdash \gamma_R : \tau' \sim \tau$.

  By (t-coerce) $\overline{\Omega \mid \Gamma \mid \Sigma \vdash e_i \blacktriangleright \gamma_i : \tau_i}$.

  By (t-fapp) $\Omega \mid \Gamma \mid \Sigma \vdash \mathsf{fn}\ (\overline{x_i : \tau_i})\{e\}(\overline{e_i \blacktriangleright \gamma_i}) : \tau'$.

  Finally, by (t-coerce) $\Omega \mid \Gamma \mid \Sigma \vdash (\mathsf{fn}\ (\overline{x_i : \tau_i})\{e\}(\overline{e_i \blacktriangleright \gamma_i})) \blacktriangleright \gamma_R : \tau$ as required.

- Case (e-fapp3):

Suppose $\Omega \mid \Gamma \mid \Sigma \vdash e(\overline{e_i}) : \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and
$\langle \mu, e(\overline{e_i}) \rangle \xrightarrow{\Omega} \langle \mu', e'(\overline{e_i}) \rangle$
where $\langle \mu, e \rangle \xrightarrow{\Omega} \langle \mu', e' \rangle$.

From lemma A.1 we get $\Omega \mid \Gamma \mid \Sigma \vdash e : \mathsf{fn}(\overline{\tau_i}) \to \tau$ and $\overline{\Omega \mid \Gamma \mid \Sigma \vdash e_i : \tau_i}$ and $\sigma = \tau$
for some τ and $\overline{\tau_i}$.

By ind. hyp. $\Omega \mid \Gamma \mid \Sigma' \vdash e' : \mathsf{fn}(\overline{\tau_i}) \to \tau$ and $\Omega \mid \Gamma \mid \Sigma' \vdash \mu'$ for some $\Sigma' \supseteq \Sigma$.

Then, by lemma A.5 and (t-fapp) $\Omega \mid \Gamma \mid \Sigma' \vdash e'(\overline{e_i}) : \tau$ as required.

- Case (e-fapp4):

  Suppose $\Omega \mid \Gamma \mid \Sigma \vdash \mathsf{fn}\ (\overline{x_i : \tau_i})\{e_1\}(\overline{v_m}, e, \overline{e_n}) : \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and
  $\langle \mu, \mathsf{fn}\ (\overline{x_i : \tau_i})\{e_1\}(\overline{v_m}, e, \overline{e_n}) \rangle \xrightarrow{\Omega} \langle \mu', \mathsf{fn}\ (\overline{x_i : \tau_i})\{e_1\}(\overline{v_m}, e', \overline{e_n}) \rangle$
  where $\langle \mu, e \rangle \xrightarrow{\Omega} \langle \mu', e' \rangle$.

  By lemma A.1 $\Omega \mid \Gamma \mid \Sigma \vdash \mathsf{fn}\ (\overline{x_i : \tau_i})\{e_1\} : \mathsf{fn}(\overline{\tau_i}) \to \tau_1$ and $\overline{\Omega \mid \Gamma \mid \Sigma \vdash e_i : \tau_i}$ and $\sigma = \tau_1$
  for some τ_1 where $\overline{x_i} = \overline{x_m}, x, \overline{x_n}$ and $\overline{\tau_i} = \overline{\tau_m}, \tau, \overline{\tau_n}$ and $\overline{e_i} = \overline{v_m}, e, \overline{e_n}$.

  By ind. hyp. $\Omega \mid \Gamma \mid \Sigma' \vdash e' : \tau$ and $\Omega \mid \Gamma \mid \Sigma' \vdash \mu'$ for some $\Sigma' \supseteq \Sigma$.

  Then, by lemma A.5 and (t-fapp) $\Omega \mid \Gamma \mid \Sigma' \vdash \mathsf{fn}\ (\overline{x_i : \tau_i})\{e_1\}(\overline{v_m}, e', \overline{e_n}) : \tau_1$ as required.

- Case (e-seq1):

  Suppose $\Omega \mid \Gamma \mid \Sigma \vdash v; e_2 : \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and
  $\langle \mu, v; e_2 \rangle \xrightarrow{\Omega} \langle \mu, e_2 \rangle$.

  By lemma A.1 there are some types τ_1, τ_2 such that
  $\Omega \mid \Gamma \mid \Sigma \vdash v : \tau_1$ and $\Omega \mid \Gamma \mid \Sigma \vdash e_2 : \tau_2$ and $\sigma = \tau_2$ as required.

- Case (e-seq2):

  Suppose $\Omega \mid \Gamma \mid \Sigma \vdash e; e_2 : \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and
  $\langle \mu, e; e_2 \rangle \xrightarrow{\Omega} \langle \mu', e'; e_2 \rangle$
  where $\langle \mu, e \rangle \xrightarrow{\Omega} \langle \mu', e' \rangle$.

  By lemma A.1 there are some types τ_1, τ_2 such that
  $\Omega \mid \Gamma \mid \Sigma \vdash e : \tau_1$ and $\Omega \mid \Gamma \mid \Sigma \vdash e_2 : \tau_2$.

  By ind. hyp. $\Omega \mid \Gamma \mid \Sigma' \vdash e' : \tau_1$ and $\Omega \mid \Gamma \mid \Sigma' \vdash \mu'$ for some $\Sigma' \supseteq \Sigma$.

  Then, by lemma A.5 and t-seq, $\Omega \mid \Gamma \mid \Sigma' \vdash e'; e_2 : \tau_2$ as required.

- Case (e-proj1):

  Suppose $\Omega \mid \Gamma \mid \Sigma \vdash S\ \langle \overline{u_j} \rangle\ \{\overline{x_i : v_i}\}.x : \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and
  $\langle \mu, S\ \langle \overline{u_j} \rangle\ \{\overline{x_i : v_i}\}.x \rangle \xrightarrow{\Omega} \langle \mu, v \rangle$
  where $(x : v) \in \overline{x_i : v_i}$.

From lemma A.1 we know that there are some $\overline{X_j}, \overline{\sigma_i}, \sigma'$ such that
$\Omega \mid \Gamma \mid \Sigma \vdash S\,\langle \overline{u_j} \rangle\,\{\overline{x_i : v_i}\} : S\,\langle \overline{u_j} \rangle$ and $S\,\langle \overline{X_j} \rangle\,\{\overline{x_i : \sigma_i}\} \in \Omega$
where $(x : \sigma') \in \overline{x_i : \sigma_i}$ and $\sigma = [\overline{u_j / X_j}]\sigma'$.

Using lemma A.1 again we get $\overline{\Omega \mid \Gamma \mid \Sigma \vdash v_i : [u_j / X_j]\sigma_i}$
so $\Omega \mid \Gamma \mid \Sigma \vdash v : [\overline{u_j / X_j}]\sigma'$ as required.

- Case (e-proj2):

  Suppose $\Omega \mid \Gamma \mid \Sigma \vdash (S\,\langle \overline{u_j} \rangle\,\{\overline{x_i : v_i}\} \blacktriangleright \gamma).x : \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and
  $\langle \mu, (S\,\langle \overline{u_j} \rangle\,\{\overline{x_i : v_i}\} \blacktriangleright \gamma).x \rangle \xrightarrow{\Omega} \langle \mu, S\,\langle \overline{u'_j} \rangle\,\{\overline{x_i : v_i \blacktriangleright\blacktriangleright [\overline{\gamma_j / X_j}]\sigma_i}\}.x \rangle$
  where $\overline{\gamma_j = \mathsf{spar}\ (j, \gamma)}$
  and $S\,\langle \overline{X_j} \rangle\,\{\overline{x_i : \sigma_i}\} \in \Omega$
  and $\Omega \vdash \gamma : S\,\langle \overline{u_j} \rangle \sim S\,\langle \overline{u'_j} \rangle$.

  From lemma A.1 we know that for some σ':
  $\Omega \mid \Gamma \mid \Sigma \vdash S\,\langle \overline{u_j} \rangle\,\{\overline{x_i : v_i}\} : S\,\langle \overline{u_j} \rangle$ and
  $\overline{\Omega \mid \Gamma \mid \Sigma \vdash v_i : [\overline{u_j / X_j}]\sigma_i}^{\,i}$ and
  $\Omega \mid \Gamma \mid \Sigma \vdash S\,\langle \overline{u_j} \rangle\,\{\overline{x_i : v_i}\} \blacktriangleright \gamma : S\,\langle \overline{u'_j} \rangle$ and
  $(x : \sigma') \in \overline{x_i : \sigma_i}$ and
  $\sigma = [\overline{u'_j / X_j}]\sigma'$.

  By (co-spar), $\overline{\Omega \vdash \gamma_j : u_j \sim u'_j}$.

  By lemma A.13 and rule (t-coerce), $\overline{\Omega \mid \Gamma \mid \Sigma \vdash v_i \blacktriangleright [\overline{\gamma_j / X_j}]\sigma_i : [\overline{u'_j / X_j}]\sigma_i}^{\,i}$.

  Then by (new-struct) $\Omega \mid \Gamma \mid \Sigma \vdash S\,\langle \overline{u'_j} \rangle\,\{\overline{x_i : v_i \blacktriangleright [\overline{\gamma_j / X_j}]\sigma_i}\} : S\,\langle \overline{u'_j} \rangle$
  and by (t-proj) $\Omega \mid \Gamma \mid \Sigma \vdash S\,\langle \overline{u'_j} \rangle\,\{\overline{x_i : v_i \blacktriangleright [\overline{\gamma_j / X_j}]\sigma_i}\}.x : [\overline{u'_j / X_j}]\sigma'$ as required.

- Case (e-proj3):

  Suppose $\Omega \mid \Gamma \mid \Sigma \vdash e.x : \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and
  $\langle \mu, e.x \rangle \xrightarrow{\Omega} \langle \mu, e'.x \rangle$
  where $\langle \mu, e \rangle \xrightarrow{\Omega} \langle \mu', e' \rangle$.

  From lemma A.1 we know that there are some $S, \overline{u_j}, \overline{X_j}, \overline{\sigma_i}, \sigma'$ such that
  $\Omega \mid \Gamma \mid \Sigma \vdash e : S\,\langle \overline{u_j} \rangle$ and $S\,\langle \overline{X_j} \rangle\,\{\overline{x_i : \sigma_i}\} \in \Omega$
  and $(x : \sigma') \in \overline{x_i : \sigma_i}$ and $\sigma = [\overline{u_j / X_j}]\sigma'$.

  By ind. hyp. there is some $\Sigma' \supseteq \Sigma$ where
  $\Omega \mid \Gamma \mid \Sigma' \vdash \mu$ and $\Omega \mid \Gamma \mid \Sigma' \vdash e' : S\,\langle \overline{u_j} \rangle$.

  Then we can apply rule (t-proj) to get $\Omega \mid \Gamma \mid \Sigma' \vdash e'.x : [\overline{u_j / X_j}]\sigma'$ as required.

- Case (e-struct):

Suppose $\Omega \mid \Gamma \mid \Sigma \vdash S\,\langle \overline{u_j} \rangle\,\{\overline{x_m : v_n}, x : e, \overline{x_n : e_n}\} : \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and
$\langle \mu, S\,\langle \overline{u_j} \rangle\,\{\overline{x_m : v_n}, x : e, \overline{x_n : e_n}\} \rangle \xrightarrow{\Omega} \langle \mu', S\,\langle \overline{u_j} \rangle\,\{\overline{x_m : v_n}, x : e', \overline{x_n : e_n}\} \rangle$
where $\langle \mu, e \rangle \xrightarrow{\Omega} \langle \mu, e' \rangle$.

Let $\overline{x_i} = \overline{x_m}, x, \overline{x_n}$ and $\overline{e_i} = \overline{v_m}, e, \overline{e_n}$.

Fromm lemma A.1 we know that there are some $\overline{X_j}, \overline{\sigma_i}$ such that:
$S\,\langle \overline{X_j} \rangle\,\{\overline{x_i : \sigma_i}\} \in \Omega$
and $\Omega \mid \Gamma \mid \Sigma \vdash e_i : \overline{[u_j/X_j]\sigma_i}$ and $\sigma = S\,\langle \overline{u_j} \rangle$.

Then, as there is some σ' where $(x : \sigma') \in \overline{x_i : \sigma_i}$ we have $\Omega \mid \Gamma \mid \Sigma \vdash e : \overline{[u_j/X_j]}\sigma'$.

By ind. hyp. there is some $\Sigma' \supseteq \Sigma$ where
$\Omega \mid \Gamma \mid \Sigma' \vdash \mu$ and $\Omega \mid \Gamma \mid \Sigma' \vdash e' : \overline{[u_j/X_j]}\sigma'$.

We can then apply rule (t-newstruct) to get
$\Omega \mid \Gamma \mid \Sigma \vdash S\,\langle \overline{u_j} \rangle\,\{\overline{x_m : v_n}, x : e', \overline{x_n : e_n}\} : S\,\langle \overline{u_j} \rangle$
as required.

- Case (e-tapp1):

  Suppose $\Omega \mid \Gamma \mid \Sigma \vdash (\Lambda \overline{X_i}.e)[\overline{u_i}] : \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and
  $\langle \mu, (\Lambda \overline{X_i}.e)[\overline{u_i}] \rangle \xrightarrow{\Omega} \langle \mu, \overline{[u_i/X_i]}e \rangle$.

  From lemma A.1 we know that there is some σ' such that
  $\Omega \mid \Gamma \mid \Sigma \vdash \Lambda \overline{X_i}..e : \forall \overline{X_i}.\sigma'$ and $\sigma = \overline{[u_i/X_i]}\sigma'$.

  Using lemma A.1 again we get $\Omega \mid \Gamma, \overline{X_i} \mid \Sigma \vdash e : \sigma'$ with $\overline{X_i \notin FV(\Omega, \Gamma, \Sigma)}$.

  We can then use lemma A.9 to get $\Omega \mid \Gamma \mid \Sigma \vdash e : \overline{[u_i/X_i]}\sigma'$ as required.

- Case (e-tapp2):

  Suppose $\Omega \mid \Gamma \mid \Sigma \vdash ((\Lambda \overline{X_i}.e) \blacktriangleright \gamma)[\overline{u_i}]$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and
  $\langle \mu, ((\Lambda \overline{X_i}.e) \blacktriangleright \gamma)[\overline{u_i}] \rangle \xrightarrow{\Omega} \langle \mu, \overline{[u_i/X_i]}e \blacktriangleright \gamma[\overline{u_i}] \rangle$.

  From lemma A.1 we get:
  $\Omega \mid \Gamma \mid \Sigma \vdash (\Lambda \overline{X_i}.e) \blacktriangleright \gamma : \forall \overline{X_i}.\sigma'$ and
  $\Omega \vdash \gamma : \forall \overline{X_i}.\sigma'' \sim \forall \overline{X_i}.\sigma'$ and
  $\Omega \mid \Gamma \mid \Sigma \vdash \Lambda \overline{X_i}.e : \forall \overline{X_i}.\sigma''$ and
  $\Omega \mid \Gamma, \overline{X_i} \mid \Sigma \vdash e : \sigma''$ with $\overline{X_i} \notin FV(\Omega, \Gamma, \Sigma)$ and
  $\sigma = \overline{[u_i/X_i]}\sigma'$
  for some σ', σ''.

  Using lemma A.9 we get $\Omega \mid \Gamma \mid \Sigma \vdash \overline{[u_i/X_i]}e : \sigma''$.

  From (co-tapp) we get $\Omega \vdash \gamma[\overline{u_i}] : \overline{[u_i/X_i]}\sigma'' \sim \overline{[u_i/X_i]}\sigma'$.

  Then using (t-coerce) we get $\Omega \mid \Gamma \mid \Sigma \vdash \overline{[u_i/X_i]}e \blacktriangleright \gamma[\overline{u_i}] : \overline{[u_i/X_i]}\sigma'$ as required.

- Case (e-tapp3):

  Suppose $\Omega \mid \Gamma \mid \Sigma \vdash e[\overline{u_i}] : \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and
  $\langle \mu, e[\overline{u_i}] \rangle \xrightarrow{\Omega} \langle \mu', e'[\overline{u_i}] \rangle$
  where $\langle \mu, e \rangle \xrightarrow{\Omega} \langle \mu', e' \rangle$.

  From lemma A.1 we know that there are some $\overline{X_i}, \overline{\omega_j}, \sigma'$ such that
  $\Omega \mid \Gamma \mid \Sigma \vdash e : \forall \overline{X_i}.\sigma'$ and $\sigma = [\overline{u_i/X_i}]\sigma'$.

  From ind. hyp. we get $\Omega \mid \Gamma \mid \Sigma' \vdash e' : \forall \overline{X_i}.\tau$ and $\Omega \mid \Gamma \mid \Sigma' \vdash \mu'$ for some $\Sigma' \supseteq \Sigma$.

  Then, using rule (t-tapp) we get $\Omega \mid \Gamma \mid \Sigma \vdash e'[\overline{u_i}] : [\overline{u_i/X_i}]\sigma'$ as required.

- Case (e-capp1):

  Suppose $\Omega \mid \Gamma \mid \Sigma \vdash (\Lambda \overline{c_i : \omega_i}.e)[\![\overline{\gamma_i}]\!] : \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and
  $\langle \mu, (\Lambda \overline{c_i : \omega_i}.e)[\![\overline{\gamma_i}]\!] \rangle \xrightarrow{\Omega} \langle \mu, [\overline{\gamma_i/c_i}]e \rangle$.

  From lemma A.1 we know that
  $\Omega \mid \Gamma \mid \Sigma \vdash \Lambda \overline{c_i : \omega_i}.e : \forall \overline{\omega_i}.\sigma$ and $\overline{\Omega \vdash \gamma_i : \omega_i}$.

  Using lemma A.1 again we get $\Omega, \overline{c_i : \omega_i} \mid \Gamma \mid \Sigma \vdash e : \sigma$.

  Then lemma A.11 gives us $\Omega \mid \Gamma \mid \Sigma \vdash [\overline{\gamma_i/c_i}]e : \sigma$ as required.

- Case (e-capp2):

  Suppose $\Omega \mid \Gamma \mid \Sigma \vdash ((\Lambda \overline{c_i : \omega_i}.e) \blacktriangleright \gamma)[\![\overline{\gamma_i}]\!] : \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and
  $\langle \mu, ((\Lambda \overline{c_i : \omega_i}.e) \blacktriangleright \gamma)[\![\overline{\gamma_i}]\!] \rangle \xrightarrow{\Omega} \langle \mu, [\overline{\gamma'_i/c_i}]e \blacktriangleright \gamma' \rangle$
  where $\overline{\gamma'_i = \mathsf{coer\text{-}l}\ (i, \gamma) \circ \gamma_i \circ \mathsf{sym}\ (\mathsf{coer\text{-}r}\ (i, \gamma))}$ and $\gamma' = \mathsf{coer}\ \gamma$.

  From lemma A.1 we get: $\Omega \mid \Gamma \mid \Sigma \vdash (\Lambda \overline{c_i : \omega_i}.e) \blacktriangleright \gamma : \forall \overline{\omega'_i}.\sigma$ and
  $\overline{\Omega \vdash \gamma_i : \omega'_i}$ and
  $\Omega \vdash \gamma : \forall \overline{\omega_i}.\sigma' \sim \forall \overline{\omega'_i}.\sigma$ and
  $\Omega \mid \Gamma \mid \Sigma \vdash \Lambda \overline{c_i : \omega_i}.e : \forall \overline{\omega_i}.\sigma'$ and
  $\Omega, \overline{c_i : \omega_i} \mid \Gamma \mid \Sigma \vdash e : \sigma'$.

  Let $\overline{\omega_i = u_{1i} \sim u_{2i}}$ and $\overline{\omega'_i = u'_{1i} \sim u'_{2i}}$.

  Using (co-coer-l) we get $\overline{\Omega \vdash \mathsf{coer\text{-}l}\ (i, \gamma) : u_{1i} \sim u'_{1i}}$.

  Using (co-coer-r) and (co-sym) we get $\overline{\Omega \vdash \mathsf{sym}\ (\mathsf{coer\text{-}r}\ (i, \gamma)) : u'_{2i} \sim u_{2i}}$.

  Using (co-trans) twice we get $\overline{\Omega \vdash \mathsf{coer\text{-}l}\ (i, \gamma) \circ \gamma_i \circ \mathsf{sym}\ (\mathsf{coer\text{-}r}\ (i, \gamma)) : u_{1i} \sim u_{2i}}$.

  Using (co-coer) we get $\Omega \vdash \mathsf{coer}\ \gamma : \sigma' \sim \sigma$.

  From lemma A.11 we get $\Omega \mid \Gamma \mid \Sigma \vdash [\overline{\gamma'_i/c_i}]e : \sigma'$

  and using (t-coerce) we get $\Omega \mid \Gamma \mid \Sigma \vdash [\overline{\gamma'_i/c_i}]e : \sigma$ as required.

- Case (e-capp3):

  Suppose $\Omega \mid \Gamma \mid \Sigma \vdash e[\![\overline{\gamma_i}]\!] : \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and
  $\langle \mu, e[\![\overline{\gamma_i}]\!] \rangle \xrightarrow{\Omega} \langle \mu', e'[\![\overline{\gamma_i}]\!] \rangle$
  where $\langle \mu, e \rangle \xrightarrow{\Omega} \langle \mu', e' \rangle$.

  From lemma A.1 we know that $\Omega \mid \Gamma \mid \Sigma \vdash e : \forall \overline{\omega_i}.\sigma$ and $\overline{\Omega \vdash \gamma_i : \omega_i}$ for some $\overline{c_i}, \overline{\omega_i}$.

  By ind. hyp. there is some $\Sigma' \supseteq \Sigma$ where $\Omega \mid \Gamma \mid \Sigma' \vdash e' : \forall \overline{\omega_i}.\sigma$ and $\Omega \mid \Gamma \mid \Sigma' \vdash \mu'$.

  Then we can use rule (t-capp) to prove $\Omega \mid \Gamma \mid \Sigma \vdash e'[\![\overline{\gamma_i}]\!] : \sigma$ as required.

- Case (e-pack):

  Suppose $\Omega \mid \Gamma \mid \Sigma \vdash \mathsf{pack}\ (\tau_1, \overline{v_m}, e, \overline{e_n}, \overline{\gamma_j})\ \mathsf{as}\ \tau_2 : \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and
  $\langle \mu, \mathsf{pack}\ (\tau_1, \overline{v_m}, e, \overline{e_n}, \overline{\gamma_j})\ \mathsf{as}\ \tau_2 \rangle \xrightarrow{\Omega} \langle \mu', \mathsf{pack}\ (\tau_1, \overline{v_m}, e', \overline{e_n}, \overline{\gamma_j})\ \mathsf{as}\ \tau_2 \rangle$
  where $\langle \mu, e \rangle \xrightarrow{\Omega} \langle \mu', e' \rangle$.

  Let $\overline{e_i} = \overline{v_m}, e, \overline{e_n}$.

  From lemma A.1 we know that there are some $T, \overline{\tau_i}, \overline{\omega_j}$ such that
  $\overline{\Omega \mid \Gamma \mid \Sigma \vdash e_i : [\tau_1/T]\tau_i}$ and $\overline{\Omega \vdash \gamma_j : [\tau_1/T]\omega_j}$ and $\sigma = \tau_2 = \&(\exists T, \overline{\tau_i}, \overline{\omega_j})$.

  So, there is some $\tau \in \overline{\tau_i}$ such that $\Omega \mid \Gamma \mid \Sigma \vdash e : [\tau_1/T]\tau$.

  By ind. hyp. there is some $\Sigma' \supseteq \Sigma$ where $\Omega \mid \Gamma \mid \Sigma' \vdash e' : [\tau_1/T]\tau$ and $\Omega \mid \Gamma \mid \Sigma' \vdash \mu'$.

  We can then use rule (t-pack) with lemma A.5 to prove
  $\Omega \mid \Gamma \mid \Sigma' \vdash \mathsf{pack}\ (\tau, \overline{v_i}, \overline{\gamma_j})\ \mathsf{as}\ \tau_2 : \&(\exists T, \overline{\tau_i}, \overline{\omega_j})$ as required.

- Case (e-unpack1):

  Suppose $\Omega \mid \Gamma \mid \Sigma \vdash \mathsf{let}\ (T, \overline{x_i}) = \mathsf{unpack}\ (\mathsf{pack}\ (\tau_1, \overline{v_i}, \overline{\gamma_j})\ \mathsf{as}\ \tau_2)\ \mathsf{in}\ e : \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and
  $\langle \mu, \mathsf{let}\ (T, \overline{x_i}) = \mathsf{unpack}\ (\mathsf{pack}\ (\tau_1, \overline{v_i}, \overline{\gamma_j})\ \mathsf{as}\ \tau_2)\ \mathsf{in}\ e \rangle \xrightarrow{\Omega} \langle \mu_i, [\overline{l_i/x_i}][\tau_1/T]e \rangle$
  where $\mu_0 = \mu$ and $\overline{\langle \mu_i, l_i \rangle = \mathbf{alloc}(\mu_{i-1}, v_i)}$ and $\overline{l_i} \notin dom(\mu)$.

  Lemma A.1 tells us that there are some $\tau, \overline{\tau_i}, \overline{\omega_j}$ such that:
  $\Omega \mid \Gamma \mid \Sigma \vdash \mathsf{pack}\ (\tau_1, \overline{v_i}, \overline{\gamma_j})\ \mathsf{as}\ \tau_2 : \&(\exists T, \overline{\tau_i}, \overline{\omega_j})$ and
  $\Omega \mid \Gamma, T, \overline{x_i : \tau_i} \mid \Sigma \vdash e : \tau$ with $T \notin FV(\Omega, \Gamma, \Sigma, \tau)$ and
  $\overline{\Omega \mid \Gamma \mid \Sigma \vdash v_i : [\tau_1/T]\tau_i}$ and
  $\sigma = \tau$.

  Lemma A.12 tells us that there is a store typing $\Sigma' \supseteq \Sigma$ such that
  $\Omega \mid \Gamma \mid \Sigma' \vdash \mu_i$ and $\overline{\Omega \mid \Gamma \mid \Sigma' \vdash l_i : [\tau_1/T]\tau_i}$.

  We can use lemma A.9 to get $\Omega \mid \Gamma, \overline{x_i : [\tau_1/T]\tau_i} \mid \Sigma \vdash [\tau_1/T]e : [\tau_1/T]\tau$.

  As $T \notin FV(\tau)$, we know that $[\tau_1/T]\tau = \tau$.

  We can then use lemma A.6 and A.5 to get $\Omega \mid \Gamma \mid \Sigma' \vdash [\overline{l_i/x_i}][\tau_1/T]e : \tau$. as required.

- Case (e-unpack2):

  Suppose $\Omega \mid \Gamma \mid \Sigma \vdash \mathsf{let}\ (T, \overline{x_i}) = \mathsf{unpack}\ ((\mathsf{pack}\ (\tau_1, \overline{v_i}, \overline{\gamma_j})\ \mathsf{as}\ \&(\exists T, \overline{\tau'_i}, \overline{\omega'_j})) \blacktriangleright \gamma)\ \mathsf{in}\ e : \sigma$ and
  $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and
  $\langle \mu, \mathsf{let}\ (T, \overline{x_i}) = \mathsf{unpack}\ ((\mathsf{pack}\ (\tau_1, \overline{v_i}, \overline{\gamma_j})\ \mathsf{as}\ \&(\exists T, \overline{\tau'_i}, \overline{\omega'_j})) \blacktriangleright \gamma)\ \mathsf{in}\ e \rangle \xrightarrow{\Omega}$
  $\langle \mu, \mathsf{let}\ (T, \overline{x_i}) = \mathsf{unpack}\ (\mathsf{pack}\ (\tau_1, \overline{v_i \blacktriangleright \gamma_i}, \overline{\gamma'_j})\ \mathsf{as}\ \&(\exists T, \overline{\tau_i}, \overline{\omega_j}))\ \mathsf{in}\ e \rangle$
  where $\Omega \vdash \gamma : \&(\exists T, \overline{\tau'_i}, \overline{\omega'_j}) \sim \&(\exists T, \overline{\tau_i}, \overline{\omega_j})$ and
  $\overline{\gamma_i = \mathsf{objf}\ (\tau_1, i, *\gamma)}$ and $\overline{\gamma'_j = (\mathsf{sym}\ (\mathsf{objc\text{-}l}\ (\tau_1, j, *\gamma))) \circ \gamma_j \circ \mathsf{objc\text{-}r}\ (\tau_1, j, *\gamma)}^{j}$.

  Lemma A.1 tells us that:
  $\Omega \mid \Gamma \mid \Sigma \vdash (\mathsf{pack}\ (\tau_1, \overline{v_i}, \overline{\gamma_j})\ \mathsf{as}\ \&(\exists T, \overline{\tau'_i}, \overline{\omega'_j})) \blacktriangleright \gamma : \&(\exists T, \overline{\tau_i}, \overline{\omega_j})$ and
  $\Omega \mid \Gamma, T, \overline{x_i : \tau_i} \mid \Sigma \vdash e : \tau$ with $T \notin FV(\Omega, \Gamma, \Sigma, \tau)$ and
  $\Omega \mid \Gamma \mid \Sigma \vdash \mathsf{pack}\ (\tau, \overline{v_i}, \overline{\gamma_j})\ \mathsf{as}\ \&(\exists T, \overline{\tau'_i}, \overline{\omega'_j}) : \&(\exists T, \overline{\tau'_i}, \overline{\omega'_j})$ and
  $\overline{\Omega \vdash \gamma_j : [\tau_1/T]\omega'_j}$ and
  $\overline{\Omega \mid \Gamma \mid \Sigma \vdash v_i : [\tau_1/T]\tau'_i}$ and
  $\sigma = \tau$ for some τ.

  Let $\overline{\omega_j = u_{1j} \sim u_{3j}}$ and $\overline{\omega'_j = u_{2j} \sim u_{4j}}$.

  By applying rule (co-deref) and (co-objf) we get $\overline{\Omega \vdash \gamma_i : [\tau_1/T]\tau'_i \sim [\tau_1/T]\tau_i}$.

  We can show that $\overline{\Omega \vdash \gamma'_j : [\tau_1/T]\omega_j}$ with the following derivation:

$$\text{(co-comp)}\ \frac{\begin{array}{c}\Omega \vdash \mathsf{sym}\ (\mathsf{objc\text{-}l}\ (\tau_1, j, *\gamma)) : [\tau_1/T]u_{1j} \sim [\tau_1/T]u_{2j} \\ \Omega \vdash \gamma_j \circ \mathsf{objc\text{-}r}\ (\tau_1, j, *\gamma) : [\tau_1/T]u_{2j} \sim [\tau_1/T]u_{3j}\end{array}}{\Omega \vdash (\mathsf{sym}\ (\mathsf{objc\text{-}l}\ (\tau_1, j, *\gamma))) \circ \gamma_j \circ \mathsf{objc\text{-}r}\ (\tau_1, j, *\gamma) : [\tau_1/T]u_{1j} \sim [\tau_1/T]u_{3j}}$$

  where

$$\text{(co-sym)}\ \frac{\text{(co-objf)}\ \dfrac{\text{(co-deref)}\ \dfrac{\Omega \vdash \gamma : \&(\exists T, \overline{\tau'_i}, \overline{\omega'_j}) \sim \&(\exists T, \overline{\tau_i}, \overline{\omega_j})}{\Omega \vdash *\gamma : (\exists T, \overline{\tau'_i}, \overline{\omega'_j}) \sim (\exists T, \overline{\tau_i}, \overline{\omega_j})}}{\Omega \vdash \mathsf{objc\text{-}l}\ (\tau_1, j, *\gamma) : [\tau_1/T]u_{2j} \sim [\tau_1/T]u_{1j}}}{\Omega \vdash \mathsf{sym}\ (\mathsf{objc\text{-}l}\ (\tau_1, j, *\gamma)) : [\tau_1/T]u_{1j} \sim [\tau_1/T]u_{2j}}$$

  and

$$\text{(co-comp)}\ \frac{\Omega \vdash \gamma_j : [\tau_1/T]u_{2j} \sim [\tau_1/T]u_{4j} \qquad \text{(co-objc-r)}\ \dfrac{\dfrac{\Omega \vdash \gamma : \&(\exists T, \overline{\tau'_i}, \overline{\omega'_j}) \sim \&(\exists T, \overline{\tau_i}, \overline{\omega_j})}{\Omega \vdash *\gamma : (\exists T, \overline{\tau'_i}, \overline{\omega'_j}) \sim (\exists T, \overline{\tau_i}, \overline{\omega_j})}}{\mathsf{objc\text{-}r}\ (\tau_1, j, *\gamma) : [\tau_1/T]u_{4j} \sim [\tau_1/T]u_{3j}}}{\Omega \vdash \gamma_j \circ \mathsf{objc\text{-}r}\ (\tau_1, j, *\gamma) : [\tau_1/T]u_{2j} \sim [\tau_1/T]u_{3j}}$$

  We can use rule (t-coerce) to prove $\overline{\Omega \mid \Gamma \mid \Sigma \vdash v_i \blacktriangleright \gamma_i : [\tau_1/T]\tau_i}$

  and then rule (t-pack) to prove
  $\Omega \mid \Gamma \mid \Sigma \vdash \mathsf{pack}\ (\tau_1, \overline{v_i \blacktriangleright \gamma_i}, \overline{\gamma'_j})\ \mathsf{as}\ \&(\exists T, \overline{\tau_i}, \overline{\omega_j}) : \&(\exists T, \overline{\tau_i}, \overline{\omega_j})$.

  Finally we can use (t-unpack) to prove
  $\Omega \mid \Gamma \mid \Sigma \vdash \mathsf{let}\ (T, \overline{x_i}) = \mathsf{unpack}\ (\mathsf{pack}\ (\tau_1, \overline{v_i \blacktriangleright \gamma_i}, \overline{\gamma'_j})\ \mathsf{as}\ \&(\exists T, \overline{\tau_i}, \overline{\omega_j}))\ \mathsf{in}\ e : \tau$ as required.

- Case (e-unpack3):

  Suppose $\Omega \mid \Gamma \mid \Sigma \vdash \mathsf{let}\ (T, \overline{x_i}) = \mathsf{unpack}\ e\ \mathsf{in}\ e_2 : \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and
  $\langle \mu, \mathsf{let}\ (T, \overline{x_i}) = \mathsf{unpack}\ e\ \mathsf{in}\ e_2 \rangle \xrightarrow{\Omega} \langle \mu', \mathsf{let}\ (T, \overline{x_i}) = \mathsf{unpack}\ e'\ \mathsf{in}\ e_2 \rangle$
  where $\langle \mu, e \rangle \xrightarrow{\Omega} \langle \mu', e' \rangle$.

  Lemma A.1 tells us that there are some $\tau, \overline{\tau_i}, \overline{\omega_j}$ such that:
  $\Omega \mid \Gamma \mid \Sigma \vdash e : \&(\exists T, \overline{\tau_i}, \overline{\omega_j})$ and
  $\Omega \mid \Gamma, T, \overline{x_i : \tau_i} \mid \Sigma \vdash e_2 : \tau$ with $T \notin FV(\Omega, \Gamma, \Sigma, \tau)$ and $\sigma = \tau$.

  From ind. hyp. we get some $\Sigma' \supseteq \Sigma$ where
  $\Omega \mid \Gamma \mid \Sigma' \vdash e' : \&(\exists T, \overline{\tau_i}, \overline{\omega_j})$ and
  $\Omega \mid \Gamma \mid \Sigma' \vdash \mu'$.

  Then by (t-unpack) and lemma A.5 we get $\Omega \mid \Gamma \mid \Sigma' \vdash \mathsf{let}\ (T, \overline{x_i}) = \mathsf{unpack}\ e'\ \mathsf{in}\ e_2 : \tau$ as required.

- Case (e-coerce1):

  Suppose $\Omega \mid \Gamma \mid \Sigma \vdash (pv \blacktriangleright \gamma_1) \blacktriangleright \gamma_2 : \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and
  $\langle \mu, (pv \blacktriangleright \gamma_1) \blacktriangleright \gamma_2 \rangle \xrightarrow{\Omega} \langle \mu, pv \blacktriangleright (\gamma_1 \circ \gamma_2) \rangle$.

  From lemma A.1 we get $\Omega \vdash \gamma_2 : \sigma' \sim \sigma$ and
  $\Omega \mid \Gamma \mid \Sigma \vdash pv \blacktriangleright \gamma_1 : \sigma'$ and
  $\Omega \vdash \gamma_1 : \sigma'' \sim \sigma'$ and
  $\Omega \mid \Gamma \mid \Sigma \vdash pv : \sigma''$
  for some σ', σ''.

  By (co-trans) $\Omega \vdash \gamma_1 \circ \gamma_2 : \sigma'' \sim \sigma$
  and by (t-coerce) $\Omega \mid \Gamma \mid \Sigma \vdash pv \blacktriangleright (\gamma_1 \circ \gamma_2) : \sigma$ as required.

- Case (e-coerce2):

  Suppose $\Omega \mid \Gamma \mid \Sigma \vdash e \blacktriangleright \gamma : \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and
  $\langle \mu, e \blacktriangleright \gamma \rangle \xrightarrow{\Omega} \langle \mu', e' \blacktriangleright \gamma \rangle$.

  From lemma A.1 we get $\Omega \vdash \gamma : \sigma' \sim \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash e : \sigma'$ for some σ'.

  From ind. hyp. we get some $\Sigma' \supseteq \Sigma$ where
  $\Omega \mid \Gamma \mid \Sigma' \vdash e' : \sigma'$ and
  $\Omega \mid \Gamma \mid \Sigma' \vdash \mu'$.

  Then by (t-coerce) $\Omega \mid \Gamma \mid \Sigma' \vdash e' \blacktriangleright \gamma : \sigma$ as required.

- Case (el-deref1):

  Suppose $\Omega \mid \Gamma \mid \Sigma \vdash *e : \tau$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and $\langle \mu, *e \rangle \xrightarrow{lv(\Omega)} \langle \mu', *e' \rangle$
  where $\langle \mu, e \rangle \xrightarrow{\Omega} \langle \mu', e' \rangle$.

From lemma A.1 we get $\Omega \mid \Gamma \mid \Sigma \vdash e : \&\tau$.

From ind. hyp. we get some $\Sigma' \supseteq \Sigma$ where
$\Omega \mid \Gamma \mid \Sigma' \vdash e' : \&\tau$ and
$\Omega \mid \Gamma \mid \Sigma' \vdash \mu'$.

Then by (t-deref) $\Omega \mid \Gamma \mid \Sigma' \vdash *e' : \tau$ as required.

- Case (el-deref2):

  Suppose $\Omega \mid \Gamma \mid \Sigma \vdash *\&l : \tau$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and $\langle \mu, *\&l \rangle \xrightarrow{lv(\Omega)} \langle \mu', l \rangle$.

  From lemma A.1 we get $\Omega \mid \Gamma \mid \Sigma \vdash \&l : \&\tau$ and $\Omega \mid \Gamma \mid \Sigma \vdash l : \tau$ as required.

- Case (el-deref3):

  Suppose $\Omega \mid \Gamma \mid \Sigma \vdash *(\&l \blacktriangleright \gamma) : \tau$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and $\langle \mu, *(\&l \blacktriangleright \gamma) \rangle \xrightarrow{lv(\Omega)} \langle \mu', l \blacktriangleright *\gamma \rangle$.

  From lemma A.1 we get $\Omega \mid \Gamma \mid \Sigma \vdash \&l \blacktriangleright \gamma : \&\tau$ and
  $\Omega \vdash \gamma : \&\tau' \sim \&\tau$ and
  $\Omega \mid \Gamma \mid \Sigma \vdash \&l : \&\tau'$ and
  $\Omega \mid \Gamma \mid \Sigma \vdash l : \tau'$ for some τ'.

  By (co-deref) $\Omega \vdash *\gamma : \tau' \sim \tau$ and
  by (t-coerce) $\Omega \vdash l \blacktriangleright *\gamma : \tau$ as required.

- Case (el-proj1):

  Suppose $\Omega \mid \Gamma \mid \Sigma \vdash lv.x : \tau$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and $\langle \mu, lv.x \rangle \xrightarrow{lv(\Omega)} \langle \mu', lv'.x \rangle$
  where $\langle \mu, lv \rangle \xrightarrow{lv(\Omega)} \langle \mu, lv' \rangle$.

  From lemma A.1 we get $\Omega \mid \Gamma \mid \Sigma \vdash lv : S\,\langle \overline{u_j} \rangle$ and
  $S\,\langle \overline{X_j} \rangle\,\{\overline{x_m : \sigma_m}, \overline{x : \tau'}, \overline{x_n : \sigma_n}\} \in \Omega$ and
  $\tau = [\overline{u_j / X_j}]\tau'$
  for some $S, \overline{u_j}, \overline{X_j}, \overline{x_m}, \overline{x_n}, \overline{\sigma_m}, \overline{\sigma_n}, \tau'$.

  From ind. hyp. we get some $\Sigma' \supseteq \Sigma$ where
  $\Omega \mid \Gamma \mid \Sigma' \vdash lv' : S\,\langle \overline{u_j} \rangle$ and
  $\Omega \mid \Gamma \mid \Sigma' \vdash \mu'$.

  Then by (t-proj) $\Omega \mid \Gamma \mid \Sigma' \vdash lv'.x : [\overline{u_j / X_j}]\tau'$ as required.

- Case (el-proj2):

  Suppose $\Omega \mid \Gamma \mid \Sigma \vdash l.x : \tau$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and $\langle \mu, l.x \rangle \xrightarrow{lv(\Omega)} \langle \mu, l' \rangle$
  where $\mu(l) = S\,\langle \overline{u_j} \rangle\,\{\overline{x_m : l_m}, x : l', \overline{x_n : l_n}\}$.

  From the definition of $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ we get
  $\Omega \mid \Gamma \mid \Sigma \vdash \mu(l) : \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash l : \sigma$ for some σ.

From lemma A.1 we get
$S\,\langle\overline{X_j}\rangle\,\{\overline{x_m : \sigma_m}, \overline{x : \tau'}, \overline{x_n : \sigma_n}\} \in \Omega$ and
$\Omega \mid \Gamma \mid \Sigma \vdash S\,\langle\overline{u_j}\rangle\,\{\overline{x_m : l_m}, x : l', \overline{x_n : l_n}\} : S\,\langle\overline{u_j}\rangle$ and
$\Omega \mid \Gamma \mid \Sigma \vdash l' : [\overline{u_j/X_j}]\tau'$ and
$\sigma = S\,\langle\overline{u_j}\rangle$ and for some $\overline{X_j}, \overline{\sigma_m}, \overline{\sigma_n}, \tau'$.

From (t-proj) we get $\Omega \mid \Gamma \mid \Sigma \vdash l.x : [\overline{u_j/X_j}]\tau'$ as required.

- Case (el-proj3):

  Suppose $\Omega \mid \Gamma \mid \Sigma \vdash l.x : \tau$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and $\langle \mu, l.x\rangle \xrightarrow{lv(\Omega)} \langle \mu, l' \blacktriangleright [\overline{\gamma_j/X_j}]\tau'\rangle$
  where $\mu(l) = S\,\langle\overline{u_j}\rangle\,\{\overline{x_m : l_m}, x : l', \overline{x_n : l_n}\} \blacktriangleright \gamma$ and
  $S\,\langle\overline{X_j}\rangle\,\{\overline{x_m : \sigma_m}, x : \tau', \overline{x_n : \sigma_n}\} \in \Omega$ and
  $\overline{\gamma_j = \mathsf{spar}\ (j, \gamma)}$.

  From $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ we get $\Omega \mid \Gamma \mid \Sigma \vdash \mu(l) : \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash l : \sigma$ for some σ.

  From lemma A.1 we get
  $\Omega \mid \Gamma \mid \Sigma \vdash S\,\langle\overline{u_j}\rangle\,\{\overline{x_m : l_m}, x : l', \overline{x_n : l_n}\} : S\,\langle\overline{u_j}\rangle$ and
  $\Omega \mid \Gamma \mid \Sigma \vdash \gamma : S\,\langle\overline{u_j}\rangle \sim \sigma$ and
  $\Omega \mid \Gamma \mid \Sigma \vdash l' : [\overline{u_j/X_j}]\tau'$.
  Combining lemma A.1 with consistency of Ω we get $\sigma = S\,\langle\overline{u'_j}\rangle$ for some $\overline{u'_j}$.
  $\Omega \mid \Gamma \mid \Sigma \vdash \mu(l) : S\,\langle\overline{u'_j}\rangle$.
  By (t-proj) $\Omega \mid \Gamma \mid \Sigma \vdash l.x : [\overline{u'_j/X_j}]\tau'$.
  By (co-spar) $\overline{\Omega \vdash \gamma_j : u_j \sim u'_j}$.
  By lemma A.13 $\Omega \vdash [\overline{\gamma_j/X}]\tau' : [\overline{u_j/X_j}]\tau' \sim [\overline{u'_j/X_j}]\tau'$ and
  by (t-coerce) $\Omega \mid \Gamma \mid \Sigma \vdash l' \blacktriangleright [\overline{\gamma_j/X}]\tau' : [\overline{u'_j/X_j}]\tau'$ as required.

- Case (el-proj4):

  Suppose $\Omega \mid \Gamma \mid \Sigma \vdash (l \blacktriangleright \gamma).x : \tau$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and $\langle \mu, (l \blacktriangleright \gamma).x\rangle \xrightarrow{lv(\Omega)} \langle \mu, l' \blacktriangleright [\overline{\gamma_j/X_j}]\tau'\rangle$
  where $\mu(l) = S\,\langle\overline{u_j}\rangle\,\{\overline{x_m : l_m}, x : l', \overline{x_n : l_n}\}$ and
  $S\,\langle\overline{X_j}\rangle\,\{\overline{x_m : \sigma_m}, x : \tau', \overline{x_n : \sigma_n}\} \in \Omega$ and
  $\overline{\gamma_j = \mathsf{spar}\ (j, \gamma)}$.

  From $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ we get $\Omega \mid \Gamma \mid \Sigma \vdash \mu(l) : \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash l : \sigma$ for some σ.

  From lemma A.1 we get
  $\Omega \mid \Gamma \mid \Sigma \vdash \mu(l) : S\,\langle\overline{u_j}\rangle$ and $\sigma = S\,\langle\overline{u_j}\rangle$, and
  $\Omega \mid \Gamma \mid \Sigma \vdash l' : [\overline{u_j/X_j}]\tau'$.

  Combining lemma A.1 with consistency of Ω we get
  $\Omega \vdash \gamma : S\,\langle\overline{u_j}\rangle \sim S\,\langle\overline{u'_j}\rangle$ and
  $\Omega \mid \Gamma \mid \Sigma \vdash l \blacktriangleright \gamma : S\,\langle\overline{u'_j}\rangle$ and

$\Omega \mid \Gamma \mid \Sigma \vdash (l \blacktriangleright \gamma).x : [\overline{u'_j/X_j}]\tau'$
for some $\overline{u'_j}$.

By (co-spar) $\overline{\Gamma \vdash \gamma_j : u_j \sim u'_j}$.
By lemma A.13 $\Omega \vdash [\overline{\gamma_j/X_j}]\tau' : [\overline{u_j/X_j}]\tau' \sim [\overline{u'_j/X_j}]\tau'$.
By (t-coerce) $\Omega \mid \Gamma \mid \Sigma \vdash l' \blacktriangleright [\overline{\gamma_j/X_j}]\tau' : [\overline{u'_j/X_j}]\tau'$ as required.

- Case (el-proj5):

  Suppose $\Omega \mid \Gamma \mid \Sigma \vdash (l \blacktriangleright \gamma_1).x : \tau$ and $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ and
  $\langle \mu, (l \blacktriangleright \gamma_1).x \rangle \xrightarrow{lv(\Omega)} \langle \mu, l' \blacktriangleright [\overline{\gamma_j/X_j}]\tau' \rangle$
  where $\mu(l) = S\,\langle \overline{u_j} \rangle\,\{\overline{x_m : l_m}, x : l', \overline{x_n : l_n}\} \blacktriangleright \gamma_2$ and
  $S\,\langle \overline{X_j} \rangle\,\{\overline{x_m : \sigma_m}, x : \tau', \overline{x_n : \sigma_n}\} \in \Omega$ and
  $\overline{\gamma_j = \mathsf{spar}\ (j, \gamma_2 \circ \gamma_1)}$.

  From $\Omega \mid \Gamma \mid \Sigma \vdash \mu$ we get $\Omega \mid \Gamma \mid \Sigma \vdash \mu(l) : \sigma$ and $\Omega \mid \Gamma \mid \Sigma \vdash l : \sigma$ for some σ.

  From lemma A.1 and corollary A.1we get
  $\Omega \mid \Gamma \mid \Sigma \vdash \gamma_2 : S\,\langle \overline{u_j} \rangle \sim \tau''$ and
  $\Omega \mid \Gamma \mid \Sigma \vdash S\,\langle \overline{u_j} \rangle\,\{\overline{x_m : l_m}, x : l', \overline{x_n : l_n}\} : S\,\langle \overline{u_j} \rangle$ and
  $\Omega \mid \Gamma \mid \Sigma \vdash l' : [\overline{u_j/X_j}]\tau'$.

  Then, using lemma A.1 again we get $\Omega \vdash \gamma_1 : \tau'' \sim \sigma'$
  where σ' is a type of the form $S'\,\langle \overline{u_k} \rangle$ for some $S', \overline{u_k}$.

  By (co-trans) $\Omega \vdash \gamma_2 \circ \gamma_1 : S\,\langle \overline{u_j} \rangle \sim \sigma'$
  and from consistency of Ω' it follows that $\sigma' = S\,\langle \overline{u'_j} \rangle$ for some $\overline{u'_j}$.
  By (t-coerce) $\Omega \mid \Gamma \mid \Sigma \vdash l : \tau''$ and $\Omega \mid \Gamma \mid \Sigma \vdash l \blacktriangleright \gamma_1 : S\,\langle \overline{u'_j} \rangle$.
  By (t-proj) $\Omega \mid \Gamma \mid \Sigma \vdash (l \blacktriangleright \gamma_1).x : [\overline{u'_j/X_j}]\tau'$.
  By (co-spar) $\overline{\Omega \vdash \gamma_j : u_j \sim u'_j}$.
  Then by lemma A.13 $\Omega \vdash [\overline{\gamma_j/X}]\tau' : [\overline{u_j/X_j}]\tau' \sim [\overline{u'_j/X_j}]\tau'$ and
  by (t-coerce) $\Omega \mid \Gamma \mid \Sigma \vdash l' \blacktriangleright [\overline{\gamma_j/X}]\tau' : [\overline{u'_j/X_j}]\tau'$ as required.

□

Theorem 6.2 (Progress)**.** *Suppose Ω is a consistent environment.*

1. *If $\Omega \mid \emptyset \mid \Sigma \vdash e : \sigma$ then either $e \in$ VAL or, for any store μ such that $\Omega \mid \emptyset \mid \Sigma \vdash \mu$, there is some term e' and store μ' with $\langle \mu, e \rangle \xrightarrow{\Omega} \langle \mu', e' \rangle$*

2. *If $\Omega \mid \emptyset \mid \Sigma \vdash lv : \tau$ then either $lv \in$ CLOC or, for any store μ such that $\Omega \mid \emptyset \mid \Sigma \vdash \mu$, there is some lvalue lv' and store μ' with $\langle \mu, lv \rangle \xrightarrow{lv(\Omega)} \langle \mu', lv' \rangle$*

Proof. By simultaneous induction on derivations $\Omega \mid \emptyset \mid \Sigma \vdash e : \sigma$ assuming $\Omega \mid \emptyset \mid \Sigma \vdash \mu$ in all cases:

- Case (t-let):

  $\Omega \mid \emptyset \mid \Sigma \vdash \mathsf{let}\ x : \tau_1 = e_1\ \mathsf{in}\ e_2 : \tau_2$ and
  $\Omega \mid \emptyset \mid \Sigma \vdash e_1 : \tau_1$ and
  $\Omega \mid x : \tau_1 \mid \Sigma \vdash e_2 : \tau_2$.

  We consider two cases for e_1:

  - Case $e_1 = v$:

    We can take a step using (e-let1).

  - Case $e_1 \notin \textsc{Val}$:

    From ind. hyp. we know that there are some e_1', μ' such that $\langle \mu, e_1 \rangle \xrightarrow{\Omega} \langle \mu', e_1' \rangle$
    We can then take a step using (e-let2).

- Case (t-upd):

  $\Omega \mid \emptyset \mid \Sigma \vdash l v := e : ()$ and
  $\Omega \mid \emptyset \mid \Sigma \vdash l v : \tau$ and
  $\Omega \mid \emptyset \mid \Sigma \vdash e : \tau$.

  We consider the following cases for $l v$ and e:

  - Case $l v = l$ and $e \notin \textsc{Val}$:

    From ind. hyp. we know that there are some e', μ' such that $\langle \mu, e \rangle \xrightarrow{\Omega} \langle \mu', e' \rangle$
    We can then take a step with rule (e-upd4).

  - Case $l v = l$ and $e = v$:

    We can take a step with rule (e-upd2).

  - Case $l v = l \blacktriangleright \gamma$:

    We can take a step with rule (e-upd3).

  - Case $l v \notin \textsc{Cloc}$:

    From ind. hyp. we know that there are some $l v', \mu'$ such that
    $\langle \mu, l v \rangle \xrightarrow{lv(\Omega)} \langle \mu', l v' \rangle$.

    We can then use (e-upd1) to take a step.

- Case (t-fapp) :

  Suppose $\Omega \mid \emptyset \mid \Sigma \vdash e(\overline{e_i}) : \tau$ and
  $\Omega \mid \emptyset \mid \Sigma \vdash e : \mathsf{fn}(\overline{\tau_i}) \to \tau$ and
  $\overline{\Omega \mid \emptyset \mid \Sigma \vdash e_i : \tau_i}$.

From lemma A.2 we know that if $v \in \text{VAL}$ then either
$v = \mathsf{fn}\ (\overline{x_i : \tau_i})\{e_1\}$ or
$v = \mathsf{fn}\ (\overline{x_i : \tau'_i})\{e_1\} \blacktriangleright \gamma$
for some $\overline{x_i}, \overline{\tau'_i}, e_1, \gamma$.

We consider the following cases for e and $\overline{e_i}$:

- Case $e = \mathsf{fn}\ (\overline{x_i : \tau_i})\{e_1\}$ and $\overline{e_i \in \text{VAL}}$:

  We can use rule (e-fapp1) to take a step.

- Case $e = \mathsf{fn}\ (\overline{x_i : \tau_i})\{e_1\}$ where there is a term $e' \in \overline{e_i}$ that is not a value:

  Let $\overline{e_i} = \overline{v_j}, e', \overline{e_k}$ and $\overline{\tau_i} = \overline{\tau_j}, \tau', \overline{\tau_k}$.

  Then, since $\Omega \mid \emptyset \mid \Sigma \vdash e' : \tau'$, from ind. hyp. we get
  $\langle \mu, e' \rangle \xrightarrow{\Omega} \langle \mu', e'' \rangle$ and μ', e''.

  We can then use rule (e-fapp4) to take a step.

- Case $e = \mathsf{fn}\ (\overline{x_i : \tau'_i})\{e_1\} \blacktriangleright \gamma$:

  We can use rule (e-fapp2) to take a step.

- Case $e \notin \text{VAL}$:

  From ind. hyp. we get $\langle \mu, e \rangle \xrightarrow{\Omega} \langle \mu', e' \rangle$
  for some e', μ' and $\Sigma' \supseteq \Sigma$.

  Then we can take a step using (e-fapp3).

- Case (t-fabs) :

  $\Omega \mid \Gamma \mid \Sigma \vdash \mathsf{fn}\ (\overline{x_i : \tau_i})\{e\} : \mathsf{fn}(\overline{\tau_i}) \to \tau$

  $\mathsf{fn}\ (\overline{x_i : \tau_i})\{e\} \in \text{VAL}$ as required.

- Case (t-ref) :

  $\Omega \mid \emptyset \mid \Sigma \vdash \&lv : \&\tau$ and
  $\Omega \mid \emptyset \mid \Sigma \vdash lv : \tau$.

  - Case $lv = l$:

    $\&l \in \text{VAL}$.

  - Case $lv = l \blacktriangleright \gamma$:

    We can take a step with (e-ref2).

  - Case $lv \notin \text{CLOC}$:

    From ind. hyp. we know that there some μ', lv' where $\langle \mu, lv \rangle \xrightarrow{lv(\Omega)} \langle \mu', lv' \rangle$.

    We can then use rule (e-ref1) to take a step.

- Case (t-deref):

  $\Omega \mid \emptyset \mid \Sigma \vdash *e : \tau$ and
  $\Omega \mid \emptyset \mid \Sigma \vdash e : \&\tau$.

  If $e \in \text{VAL}$ then by lemma A.2, either $e = \&l$ for some l or $e = \&l \blacktriangleright \gamma$ for some l and γ.

  We then consider the following cases for e, proving both points 1. and 2. as $*e$ is an lvalue:

  - Case $e = \&l$:
    1. We can use rule (e-deref1) to take a step.
    2. We can take a step with (el-deref2).
  - Case $e = \&l \blacktriangleright \gamma$:
    1. We can use rule (e-deref2) to take a step.
    2. We can take a step with (el-deref3).
  - Case $e \notin \text{VAL}$:

    From ind. hyp. we know that there are some μ', e' such that
    $\langle \mu, e \rangle \xrightarrow{\Omega} \langle \mu', e' \rangle$.

    Then:
    1. We can take a step with (e-deref3).
    2. We can take a step with (e-deref1).

- Case (t-var):

  $\Omega \mid \emptyset \mid \Sigma \vdash x : \sigma$ and
  $(x : \sigma) \in \emptyset$.

  Vacuous as $(x : \sigma) \in \emptyset$ is impossible.

- Case (t-loc):

  Suppose $\Omega \mid \emptyset \mid \Sigma \vdash l : \sigma$ and $\Gamma_T \mid \Sigma \vdash \mu$.

  $l \in \text{LVAL}$ so we must show the case holds for both cases:
  1. We can take a step with (e-loc) as by lemma A.16 the call to getValue will succeed.
  2. $l \in \text{CLOC}$ as required.

  From lemma A.1 we know $\Sigma(l) = \sigma$.

- Case (t-unit):

  $\Omega \mid \emptyset \mid \Sigma \vdash () : ()$
  $() \in \text{VAL}$ as required.

- Case (t-seq):

  $\Omega \mid \emptyset \mid \Sigma \vdash e_1; e_2 : \tau_2$ and $\Omega \mid \emptyset \mid \Sigma \vdash e_1 : \tau_1$ and $\Omega \mid \emptyset \mid \Sigma \vdash e_2 : \tau_2$.

  We consider two cases of e_1:

  - Case $e_1 = v$:

    We can use (e-seq1) to take a step.

  - Case $e_1 \notin \text{VAL}$:

    From ind. hyp. we know that there are some μ', e_1' such that $\langle \mu, e_1 \rangle \xrightarrow{\Omega} \langle \mu', e_1' \rangle$.

    We can then make a step using rule (e-seq2).

- Case (t-newstruct):

  $\Omega \mid \emptyset \mid \Sigma \vdash S\,\langle \overline{u_j} \rangle\,\{\overline{x_i : e_i}\} : S\,\langle \overline{u_j} \rangle$ and
  $S\,\langle \overline{X_j} \rangle\,\{\overline{x_i : \sigma_i}\} \in \Omega$ and
  $\Omega \mid \emptyset \mid \Sigma \vdash e_i : [\overline{u_j / X_j}]\sigma_i$.

  We consider two cases of $\overline{e_i}$:

  - Case $\overline{e_i = v_i}$:

    Then $S\,\langle \overline{u_j} \rangle\,\{\overline{x_i : v_i}\} \in \text{VAL}$.

  - Case $\overline{e_i} = \overline{v_m}, e, \overline{e_n}$ where $e \notin \text{VAL}$:

    Let $\overline{x_i} = \overline{x_m}, x, \overline{x_n}$ and $\overline{\sigma_i} = \overline{\sigma_m}, \sigma, \overline{\sigma_n}$.

    From ind. hyp. we know there are some μ', e_1' such that $\langle \mu, e_1 \rangle \xrightarrow{\Omega} \langle \mu', e_1' \rangle$.

    We can then use rule (e-struct) to take a step as required.

- Case (t-proj) :

  $\Omega \mid \emptyset \mid \Sigma \vdash e.x : [\overline{u_j / X_j}]\sigma$ and
  $\Omega \mid \emptyset \mid \Sigma \vdash e : S\,\langle \overline{u_j} \rangle$ and
  $S\,\langle \overline{X_j} \rangle\,\{\overline{x_i : \sigma_i}\} \in \Omega$
  where $(x : \sigma) \in \overline{x_i : \sigma_i}$.

  Since $e.x$ can be an lvalue, we must prove that both statements of the lemma hold.

  1. If $e \in \text{VAL}$ then from lemma A.2 we know that either
  $e = S\,\langle \overline{u_j} \rangle\,\{\overline{x_i : v_i}\}$ for some $\overline{x_i}, \overline{v_i}$ or
  $e = S\,\langle \overline{u_j'} \rangle\,\{\overline{x_i : v_i}\} \blacktriangleright \gamma$ for some $\overline{u_j'}, \overline{x_i}, \overline{v_i}$ and γ where $\Gamma_T \mid \Sigma \vdash \gamma : S\,\langle \overline{u_j'} \rangle \sim S\,\langle \overline{u_j} \rangle$.

  We then proceed by cases on e:

– Case $e = S\,\langle \overline{u_j} \rangle\,\{\overline{x_i : v_i}\}$:

We can take a step with (e-proj1).

– Case $e = S\,\langle \overline{u'_j} \rangle\,\{\overline{x_i : v_i}\} \blacktriangleright \gamma$:

We can take a step with (e-proj2).

– Case $e \notin \text{VAL}$:

Using ind. hyp. we know that there are some μ', e' such that
$\langle \mu, e \rangle \xrightarrow{\Omega} \langle \mu', e' \rangle$.

We can then take a step using rule (e-proj3).

2. Suppose $e = l\,v$ and $\sigma = \tau$ for some $l\,v$ and τ.
We then proceed by cases on $l\,v$:

– $l\,v = l$:

From $\Omega \mid \emptyset \mid \Sigma \vdash \mu$ it follows that $\Omega \mid \emptyset \mid \Sigma \vdash \mu(l) : S\,\langle \overline{u_j} \rangle$.

From lemma A.3 we know that either
$\mu(l) = S\,\langle \overline{u_j} \rangle\,\{\overline{x_i : l_i}\}$ for some $\overline{l_i}$ or
$\mu(l) = S\,\langle \overline{u'_j} \rangle\,\{\overline{x_i : l_i}\} \blacktriangleright \gamma$ for some $\overline{u'_j}, \overline{l_i}, \gamma$ where $\Omega \vdash \gamma : S\,\langle \overline{u'_j} \rangle \sim S\,\langle \overline{u_j} \rangle$.

In the first case we can take a step with (el-proj2)
and in the second case we can take a step with (el-proj3).

– $l\,v = l \blacktriangleright \gamma$:

We know that $\Omega \mid \emptyset \mid \Sigma \vdash l \blacktriangleright \gamma : S\,\langle \overline{u_j} \rangle$.

From lemma A.1 and corollary A.1 we know that there is some τ' such that
$\Omega \vdash \gamma : \tau' \sim S\,\langle \overline{u_j} \rangle$ and
$\Omega \mid \emptyset \mid \Sigma \vdash l : \tau'$.

From $\Omega \mid \emptyset \mid \Sigma \vdash \mu$ it follows that $\Omega \mid \emptyset \mid \Sigma \vdash \mu(l) : \tau'$.

From lemma A.3 we know that either
$\mu(l) = S\,\langle \overline{u'_j} \rangle\,\{\overline{x_i : l_i}\}$ where $\tau' = S\,\langle \overline{u'_j} \rangle$ for some $\overline{l_i}, \overline{u'_j}$ or
$\mu(l) = S\,\langle \overline{u'_j} \rangle\,\{\overline{x_i : l_i}\} \blacktriangleright \gamma'$ for some $\overline{u'_j}, \overline{l_i}, \gamma$ where $\Omega \vdash \gamma' : S\,\langle \overline{u'_j} \rangle \sim \tau'$.

In the first case we can take a step with (el-proj4)
and in the second case we can take a step with (el-proj5).

– $l\,v \notin \text{CLOC}$:

By ind. hyp. there are some $l\,v', \mu'$ where $\langle \mu, l\,v \rangle \xrightarrow{l\,v(\Omega)} \langle \mu', l\,v' \rangle$.

We can then take a step with (el-proj1) as required.

• Case (t-unpack):

$\Omega \mid \emptyset \mid \Sigma \vdash \mathsf{let}\ (T, \overline{x_i}) = \mathsf{unpack}\ e_1\ \mathsf{in}\ e_2 : \tau$ and
$\Omega \mid \emptyset \mid \Sigma \vdash e_1 : \&(\exists T, \overline{\tau_i}, \overline{\omega_j})$ and
$\Omega \mid T, \overline{x_i : \tau_i} \mid \Sigma \vdash e_2 : \tau$.

From lemma A.2 we know that if e_1 is a value then either
$e_1 = \mathsf{pack}\ (\tau, \overline{v_i}, \overline{\gamma_j})$ as $\&(\exists T, \overline{\tau_i}, \overline{\omega_j})$, or
$e_1 = (\mathsf{pack}\ (\tau, \overline{v_i}, \overline{\gamma_j})$ as $\&(\exists T, \overline{\tau_i'}, \overline{\omega_j'})) \blacktriangleright \gamma$ for some $\overline{\tau_i'}, \overline{\omega_j'}$ where $\Gamma \mid \Sigma \vdash \gamma : \&(\exists T, \overline{\tau_i'}, \overline{\omega_j'}) \sim \&(\exists T, \overline{\tau_i}, \overline{\omega_j})$.

We proceed by cases of e_1:

- $e_1 = \mathsf{pack}\ (\tau, \overline{v_i}, \overline{\gamma_j})$ as $\&(\exists T, \overline{\tau_i}, \overline{\omega_j})$:
  We can take a step with (e-unpack1).
- $e_1 = (\mathsf{pack}\ (\tau, \overline{v_i}, \overline{\gamma_j})$ as $\&(\exists T, \overline{\tau_i'}, \overline{\omega_j'})) \blacktriangleright \gamma$:
  We can take a step with (e-unpack2).
- $e_1 \notin \textsc{Val}$:
  By the ind. hyp. there are μ', e_1' such that $\langle \mu, e_1 \rangle \xrightarrow{\Omega} \langle \mu', e_1' \rangle$.
  We can then take a step using (e-unpack3) as required.

- Case (t-pack):

  $\Gamma_T \mid \Sigma \vdash \mathsf{pack}\ (\tau, \overline{e_i}, \overline{\gamma_j})$ as $\&(\exists T, \overline{\tau_i}, \overline{\omega_j}) : \&(\exists T, \overline{\tau_i}, \overline{\omega_j})$ and
  $\overline{\Omega \mid \emptyset \mid \Sigma \vdash e_i : [\tau/T]\tau_i}$ and
  $\overline{\Omega \vdash \gamma_j : [\tau/T]\omega_j}$.

  We proceed by cases of $\overline{e_i}$:

  - Case $\overline{e_i} = \overline{v_i}$:
    $(\mathsf{pack}\ (\tau, \overline{v_i}, \overline{\gamma_j})$ as $\&(\exists T, \overline{\tau_i}, \overline{\omega_j})) \in \textsc{Val}$.
  - Case $\overline{e_i} = \overline{v_m}, e, \overline{e_n}$ where $e \notin \textsc{Val}$:
    From ind. hyp. there are some μ', e' such that
    $\langle \mu, e \rangle \xrightarrow{\Omega} \langle \mu, e' \rangle$.
    We can then use rule (e-pack) to make a step.

- Case (t-tapp) :

  $\Omega \mid \emptyset \mid \Sigma \vdash e[\overline{u_i}] : [\overline{u_i/X_i}]\sigma$ and
  $\Omega \mid \emptyset \mid \Sigma \vdash e : \forall \overline{X_i}.\sigma$.

  From lemma A.2 we know that if $e \in \textsc{Val}$ then either:
  $e = \forall \overline{X_i}.e'$ for some e' or
  $e = (\forall \overline{X_i}.e') \blacktriangleright \gamma$ for some e', γ.

  We proceed by cases of e:

- Case $e = \forall\overline{X_i}.e'$:

  We can take a step with rule (e-tapp1).

- Case $e = (\forall\overline{X_i}.e') \blacktriangleright \gamma$:

  We can take a step with rule (e-tapp2).

- Case $e \notin \text{VAL}$:

  By ind. hyp. there are some e', μ' such that
  $\langle \mu, e\rangle \xrightarrow{\Omega} \langle \mu', e'\rangle$.

  We can then use rule (e-tapp3) to take a step.

- Case (t-tabs) :

  $\Omega \mid \emptyset \mid \Sigma \vdash \Lambda\overline{X_i}.e : \forall\overline{X_i}.\sigma$.
  $\Lambda\overline{X_i}.e \in \text{VAL}$.

- Case (t-capp) :

  $\Omega \mid \emptyset \mid \Sigma \vdash e[\![\overline{\gamma_i}]\!] : \sigma$ and
  $\Omega \mid \emptyset \mid \Sigma \vdash e : \forall\overline{\omega_i}.\sigma$ and
  $\overline{\Omega \vdash \gamma_i : \omega_i}$.

  From lemma A.2 we know that if $e \in \text{VAL}$ then either:
  $e = \forall\overline{\omega_i}.e'$ for some e' or
  $e = (\forall\overline{\omega'_i}.e') \blacktriangleright \gamma$ for some $e', \overline{\omega'_i}, \gamma$.

  We proceed on cases of e:

  - Case $e = \forall\overline{\omega_i}.e'$:

    We can use rule (e-capp1) to take a step.

  - Case $e = (\forall\overline{\omega'_i}.e') \blacktriangleright \gamma$

    We can use rule (e-capp2) to take a step.

  - Case $e \notin \text{VAL}$:

    By ind. hyp. there are some e', μ' such that
    $\langle \mu, e\rangle \xrightarrow{\Omega} \langle \mu', e'\rangle$.

    Then we can use rule (e-capp3) to take a step.

- Case (t-cabs):

  $(\Lambda\overline{c_i : \omega_i}.e) \in \text{VAL}$.

- Case (t-coerce):

$\Gamma_T \mid \Sigma \vdash e \blacktriangleright \gamma : \sigma_2$ and
$\Gamma \vdash \gamma : \sigma_1 \sim \sigma_2$ and
$\Gamma \mid \Sigma \vdash e : \sigma_1$.

Since $e \blacktriangleright \gamma$ could be an lvalue, we must show that the case holds for both statements of the theorem.

1. We proceed by cases of e:

   - Case $e = pv$:

     Then $(pv \blacktriangleright \gamma) \in \text{VAL}$.

   - Case $e = pv \blacktriangleright \gamma'$:

     Then we can take a step with (e-coerce1).

   - Case $e \notin \text{VAL}$:

     By ind. hyp. there are some e', μ' such that
     $\langle \mu, e \rangle \xrightarrow{\Omega} \langle \mu', e' \rangle$.

     Then we can use rule (e-coerce2) to take a step.

2. If $e \blacktriangleright \gamma$ is an lvalue it must be of the form $l \blacktriangleright \gamma$ for some l.

Then $l \blacktriangleright \gamma \in \text{CLOC}$ as required.

□

Theorem 6.3. *Suppose* $\vdash pgm : \Omega \mid \Gamma$ *where* Ω *is consistent and* $\Gamma = \{\overline{x_i : \sigma_i}\}$.
If $(\mathsf{main} : \mathsf{fn}() \to ()) \in \Gamma$, *then there are some* μ, e, Σ *for which*
$\langle pgm \rangle \xrightarrow{\Omega} \langle \mu, e \rangle$, *and* $\Omega \mid \emptyset \mid \Sigma \vdash \mu$, *and* $\Omega \mid \emptyset \mid \Sigma \vdash e : ()$.

Proof. Wlog, let $pgm = \{\overline{type}, \overline{axiom}, \overline{struct}, \overline{decl}\}$ where $\overline{decl} = \overline{x_i : \sigma_i = v_i}$ and $\mathsf{main} \in \overline{x_i}$.
By rule (t-decl), $\overline{\Omega \mid \Gamma \mid \emptyset \vdash v_i : \sigma_i}$.
Let $\mu_0 = \emptyset$. Then by lemma A.12 $\overline{\mathbf{alloc}(\mu_{i-1}, v_i) = \langle \mu_i, l_i \rangle}^{i}$
where $\overline{l_i} \notin dom(\mu_0)$ and there is some Σ where
$\Omega \mid \Gamma \mid \Sigma \vdash \mu_i$ and $\overline{\Omega \mid \Gamma \mid \Sigma \vdash l_i : \sigma_i}$.
By lemma A.1 $\overline{(l_i : \sigma_i) \in \Sigma}$ and
we can use rule (t-loc) to conclude $\overline{\Omega \mid \emptyset \mid \Sigma \vdash l_i : \sigma_i}$.
We can use the program evaluation rule to take a step
$\langle pgm \rangle \xrightarrow{\Omega} \langle [\overline{l_i/x_i}]\mu_i, [\overline{l_i/x_i}] f_{main}() \rangle$.
Wlog, let $\mu_i = \{\overline{l_j : w_j}\}$.
By definition of substitution, $[\overline{l_i/x_i}]\mu_i = \{\overline{l_j : [\overline{l_i/x_i}]w_j}\}$.
From $\Omega \mid \Gamma \mid \Sigma \vdash \mu_i$ we get $\overline{\Omega \mid \Gamma \mid \Sigma \vdash l_j : \sigma_j}^{j}$ and $\overline{\Omega \mid \Gamma \mid \Sigma \vdash \mu_i(l_j) : \sigma_j}^{j}$ for some $\overline{\sigma_j}$.
By lemma A.6 we get $\overline{\Omega \mid \emptyset \mid \Sigma \vdash l_j : \sigma_j}$ and $\overline{\Omega \mid \emptyset \mid \Sigma \vdash [\overline{l_i/x_i}]\mu_i(l_j) : \sigma_j}^{j}$

so $\Omega \mid \emptyset \mid \Sigma \vdash [\overline{l_i/x_i}]\mu_i$.

Finally, let $l_{\mathsf{main}} = [\overline{l_i/x_i}]\mathsf{main}$.

Then $\Omega \mid \emptyset \mid \Sigma \vdash l_{\mathsf{main}} : \mathsf{fn}() \to ()$ by (t-loc) and $\Omega \mid \emptyset \mid \Sigma \vdash l_{\mathsf{main}}() : ()$ by (t-app) as required.

□

Appendix B

MiniRust translation proofs

Lemma B.1 (Inversion of well-formed environment rules)**.** *Suppose* $\langle \Gamma, \Theta \rangle \rightsquigarrow \langle \Omega^t, \Gamma^t \rangle$.

1. *If* $X \in \Gamma$ *then* $X \in \Gamma^t$.
2. *If* $(x : \sigma) \in \Gamma$ *then there is some* σ^t *such that* $(x : \sigma^t) \in \Gamma^t$ *and* $\Gamma \mid \Theta \vdash_{\text{WF}} \sigma \rightsquigarrow \sigma^t$.
3. *If* $S\,\langle \overline{X_j} \rangle\,\{\overline{x_i : \tau_i}\} \in \Gamma$ *then there are some* $\overline{\tau_i^t}$ *such that* $\overline{\Gamma, \overline{X_j} \mid \Theta \vdash_{\text{WF}} \tau_i \rightsquigarrow \tau_i^t}$ *and* $S\,\langle \overline{X_j} \rangle\,\{\overline{x_i : \tau_i^t}\} \in \Omega^t$.
4. *If* $(D\,\langle \overline{X_i} \rangle \text{ where } \overline{\pi_h}, \mathit{Self} : \overline{\beta_s \rightsquigarrow x_s}, A : \overline{\beta_a \rightsquigarrow x_a}, f, \mathit{obj}) \in \Gamma$ *then*
   $\overline{\Gamma, \overline{X_p} \mid \Theta, \mathit{Self} : D\,\langle \overline{X_i} \rangle \rightsquigarrow \_ \vdash_{\text{WF}} \mathit{Self} : \beta_s \rightsquigarrow \langle \tau_s^t, \omega_s \rangle}^{s}$,
   $\overline{\Gamma, \overline{X_p} \mid \Theta, \mathit{Self} : D\,\langle \overline{X_i} \rangle \rightsquigarrow \_ \vdash_{\text{WF}} A_D\,\langle \overline{X_p} \rangle : \beta_a \rightsquigarrow \langle \tau_a^t, \omega_a \rangle}^{a}$,
   $(f : \forall \overline{X_p}, \overline{X_k}.(\mathit{Self} : D\,\langle \overline{X_i} \rangle, \overline{\pi_l}) \Rightarrow \tau) \in \Gamma$,
   $\Gamma, \overline{X_p} \mid \Theta, \mathit{Self} : D\,\langle \overline{X_i} \rangle \rightsquigarrow \_ \vdash_{\text{WF}} \forall \overline{X_k}.\overline{\pi_l} \Rightarrow \tau \rightsquigarrow \sigma^t$,
   $\{S\,\langle \overline{X_p} \rangle\,\{f : \sigma^t, \overline{x_s : \tau_s^t}, \overline{x_a : \tau_a^t}\}, A_D\,\langle \overline{X_p} \rangle\} \subseteq \Omega^t$
   where $\overline{X_p} = \mathit{Self}, \overline{X_i}$ *for some* $\overline{\tau_s^t}, \overline{\omega_s}, \overline{\tau_a^t}, \overline{\omega_a}, \overline{X_k}, \overline{\pi_l}, \tau$.
5. *If* $(\theta \rightsquigarrow \langle x, c \rangle) \in \Theta$ *then there are some* σ^t *and* ϑ *such that* $(x : \sigma^t) \in \Gamma^t$, *and* $(c : \vartheta) \in \Omega^t$ *and* $\Gamma \mid \Theta \vdash_{\text{WF}} \theta \rightsquigarrow \langle \sigma^t, \vartheta \rangle$.

Proof. Immediate from the well-formed environments judgments with translation. □

Lemma B.2 (Inversion of struct declaration typing rule)**.**
If $\Gamma \mid \Theta \vdash \text{struct } S\,\langle \overline{X_j} \rangle\,\{\overline{x_i : \tau_i}\} : \Gamma' \mid \Theta' \rightsquigarrow pgm^t$
then there are some $\overline{\tau_i^t}$ *where* $\overline{\Gamma, \overline{X_j} \mid \Theta \vdash_{\text{WF}} \tau_i \rightsquigarrow \tau_i^t}^{i}$,
$\Gamma' = [S\,\langle \overline{X_j} \rangle\,\{\overline{x_i : \tau_i}\}]$, *and* $\Theta' = \emptyset$, *and* $pgm^t = \text{struct } S\,\langle \overline{X_j} \rangle\,\{\overline{x_i : \tau_i^t}\}$.

Proof. Immediate from the well-typed items relation. □

Lemma B.3 (Inversion of well-formedness rules)**.**

1. *If* $\Gamma \mid \Theta \vdash_{\mathrm{WF}} X \rightsquigarrow u^t$ *then* $X \in \Gamma$ *and* $u^t = X$.

2. *If* $\Gamma \mid \Theta \vdash_{\mathrm{WF}} \mathsf{fn}(\overline{\tau_i}) \to \tau \rightsquigarrow u^t$ *then*
$\overline{\Gamma \mid \Theta \vdash_{\mathrm{WF}} \tau_i \rightsquigarrow \tau_i^t}$, *and* $\Gamma \mid \Theta \vdash_{\mathrm{WF}} \tau \rightsquigarrow \tau^t$, *and* $u^t = \mathsf{fn}(\overline{\tau_i^t}) \to \tau^t$ *for some* τ^t, $\overline{\tau_i^t}$.

3. *If* $\Gamma \mid \Theta \vdash_{\mathrm{WF}} \&u \rightsquigarrow u^t$ *then* $\Gamma \mid \Theta \vdash_{\mathrm{WF}} u \rightsquigarrow u_1^t$ *and* $u^t = \&u_1^t$ *for some* u_1^t.

4. *If* $\Gamma \mid \Theta \vdash_{\mathrm{WF}} S\,\langle \overline{u_i} \rangle \rightsquigarrow u^t$ *then*
$\overline{\Gamma \mid \Theta \vdash_{\mathrm{WF}} u_i \rightsquigarrow u_i^t}$, *and* $S\,\langle \overline{X_i} \rangle\,\{\overline{x_j : \tau_j}\} \in \Gamma$, *and* $[\overline{u_i/X_i}]$ *defined, and* $u^t = S\,\langle \overline{u_i^t} \rangle$
for some $\overline{X_i}, \overline{u_i^t}, \overline{x_j}, \overline{\tau_j}$.

5. *If* $\Gamma \mid \Theta \vdash_{\mathrm{WF}} A_D\,\langle u, \overline{u_i} \rangle \rightsquigarrow u^t$ *then*
$\Gamma \mid \Theta \Vdash u : D\,\langle \overline{u_i}, A \mapsto \star \rangle$, *and* $\overline{\Gamma \mid \Theta \vdash_{\mathrm{WF}} u \rightsquigarrow u_1^t}$, *and*
$\overline{\Gamma \mid \Theta \vdash_{\mathrm{WF}} u_i \rightsquigarrow u_i^t}$, *and* $u^t = A_D\,\langle u_1^t, \overline{u_i^t} \rangle$
for some u_1^t, $\overline{u_i^t}$.

6. *If* $\Gamma \mid \Theta \vdash_{\mathrm{WF}} \exists T : D\,\langle \overline{u_i}, \overline{u_{1j} \sim u_{2j}} \rangle \rightsquigarrow u^t$ *then*
there are some $\overline{A_{Dj}}, \overline{\overline{u_{sj}}}, \overline{u_j}, \overline{u_i^t}, \overline{\overline{u_{sj}^t}}, \overline{u_j^t}$ *such that*
$\overline{u_{1j} = A_{Dj}\,\langle T, \overline{u_{sj}} \rangle}$, *and* $\Gamma \vdash D$ *obj-safe, and* $\Gamma \vdash \mathit{Self}_{\mathbf{U}} : D\,\langle \overline{u_i} \rangle$, *and*
$\overline{\Gamma \mid \Theta \vdash_{\mathrm{WF}} u_i \rightsquigarrow u_i^t}$, *and* $\overline{\Gamma, T \mid \Theta, T : D\,\langle \overline{u_i} \rangle \vdash_{\mathrm{WF}} u_{1j} \rightsquigarrow A_{Dj}\,\langle T, \overline{u_{sj}^t} \rangle}$,
and $\overline{\Gamma \mid \Theta \vdash_{\mathrm{WF}} u_j \rightsquigarrow u_j^t}$, *and* $u^t = \left(\exists T, \&\tau, S_D\,\langle T, \overline{u_i^t} \rangle, \overline{A_{Dj}\,\langle T, \overline{u_{sj}^t} \rangle \sim u_j^t}\right)$

7. *If* $\Gamma \mid \Theta \vdash_{\mathrm{WF}} \forall \overline{X_i}.\overline{\pi_j} \Rightarrow \tau \rightsquigarrow \sigma^t$ *then* $\overline{X_i \notin \Gamma}$ *and* $\overline{\Gamma, \overline{X_i} \mid \Theta, \overline{\pi_j} \vdash_{\mathrm{WF}} \pi_j \rightsquigarrow \langle \tau_j^t, \omega_j \rangle}^{\,j}$
and $\Gamma, \overline{X_i} \mid \Theta, \overline{\pi_j} \vdash_{\mathrm{WF}} \tau \rightsquigarrow \tau^t$ *and* $\sigma^t = \forall \overline{X_i}.\forall \overline{\omega_j}.\mathsf{fn}(\overline{\tau_j^t}) \to \tau^t$ *for some* $\overline{\tau_j^t}, \overline{\omega_j}$ *and* τ^t.

8. *If* $\Gamma \mid \Theta \vdash_{\mathrm{WF}} \forall \overline{X_i}.\overline{\pi_j} \Rightarrow \pi \rightsquigarrow \langle \sigma^t, \vartheta \rangle$ *then* $\overline{\Gamma, \overline{X_i} \mid \Theta, \overline{\pi_j} \vdash_{\mathrm{WF}} \pi_j \rightsquigarrow \langle \tau_j^t, \omega_j \rangle}^{\,j}$,
$\Gamma, \overline{X_i} \mid \Theta, \overline{\pi_j} \vdash_{\mathrm{WF}} \pi \rightsquigarrow \langle \tau^t, u_1^t \sim u_2^t \rangle$,
$\sigma^t = \forall \overline{X_i}.\forall \overline{\omega_j}.fn(\overline{\tau_j^t}) \to \tau^t$, *and* $\vartheta = \forall \overline{X_i}.u_1^t \sim \forall \overline{X_i}.u_2^t$ *for some* $\overline{\tau_j^t}, \overline{\omega_j}, u_1^t, u_2^t, \tau^t$.

9. *If* $\Gamma \mid \Theta \vdash_{\mathrm{WF}} u : D\,\langle \overline{u_i}, A \mapsto u_A \rangle \rightsquigarrow \sigma^t, \vartheta$ *then there are some* $u^t, \overline{u_i^t}, u_A^t$ *where*
$\Gamma \vdash u : D\,\langle \overline{u_i} \rangle$, *and* $\Gamma \vdash A_D$, *and*
$\Gamma \mid \Theta \vdash_{\mathrm{WF}} u \rightsquigarrow u^t$, *and* $\overline{\Gamma \mid \Theta \vdash_{\mathrm{WF}} u_i \rightsquigarrow u_i^t}$, *and* $\Gamma \mid \Theta \vdash_{\mathrm{WF}} u_A \rightsquigarrow u_A^t$, *and*
$\sigma^t = S_D\,\langle u^t, \overline{u_i^t} \rangle$, *and* $\vartheta = A_D\,\langle u^t, \overline{u_i^t} \rangle \sim u_A^t$.

10. *If* $\Gamma \mid \Theta \vdash_{\mathrm{WF}} u : D\,\langle \overline{u_i}, A \mapsto \star \rangle \rightsquigarrow \sigma^t, \vartheta$ *then there are some* $u^t, \overline{u_i^t}$ *where*
$\Gamma \vdash u : D\,\langle \overline{u_i} \rangle$, *and* $\Gamma \vdash A_D$, *and*
$\Gamma \mid \Theta \vdash_{\mathrm{WF}} u \rightsquigarrow u^t$, *and* $\overline{\Gamma \mid \Theta \vdash_{\mathrm{WF}} u_i \rightsquigarrow u_i^t}$, *and*
$\sigma^t = S_D\,\langle u^t, \overline{u_i^t} \rangle$, *and* $\vartheta = A_D\,\langle u^t, \overline{u_i^t} \rangle \sim A_D\,\langle u^t, \overline{u_i^t} \rangle$.

Proof. Immediate from the well-formedness judgments. □

Lemma B.4 (Contraction of the constraint environment I)**.**

1. *If* $\Gamma \mid \Theta, \pi' \rightsquigarrow \langle x, c\rangle \vdash_{\mathrm{WF}} u \rightsquigarrow u^t$ *and* $\Gamma \mid \Theta \Vdash \pi' \rightsquigarrow \langle e'^t, \gamma'\rangle$ *then* $\Gamma \mid \Theta \vdash_{\mathrm{WF}} u \rightsquigarrow u^t$.

2. *If* $\Gamma \mid \Theta, \pi' \rightsquigarrow \langle x, c\rangle \Vdash \pi \rightsquigarrow \langle e^t, \gamma\rangle$ *and* $\Gamma \mid \Theta \Vdash \pi' \rightsquigarrow \langle e'^t, \gamma'\rangle$ *then* $\Gamma \mid \Theta \Vdash \pi \rightsquigarrow [e'^t/x][\gamma'/c]e^t$.

3. *If* $\Gamma \mid \Theta, \pi' \rightsquigarrow \langle x, c\rangle \Vdash u_1 \sim u_2 \rightsquigarrow \gamma$ *and* $\Gamma \mid \Theta \Vdash \pi' \rightsquigarrow \langle e'^t, \gamma'\rangle$ *then* $\Gamma \mid \Theta \Vdash u_1 \sim u_2 \rightsquigarrow [\gamma'/c]\gamma$.

Proof. By simultaneous induction on derivations
$\Gamma \mid \Theta, \pi' \rightsquigarrow \langle x, c\rangle \vdash_{\mathrm{WF}} u \rightsquigarrow u^t$,
$\Gamma \mid \Theta, \pi' \rightsquigarrow \langle x, c\rangle \Vdash \pi \rightsquigarrow \langle e^t, \gamma\rangle$ and
$\Gamma \mid \Theta, \pi' \rightsquigarrow \langle x, c\rangle \Vdash u_1 \sim u_2 \rightsquigarrow \gamma$.
In most cases it is a straightforward use of the induction hypothesis and the definition of substitution.
We show the proof for the only interesting case:

- **Case** (c-ext): $\Gamma \mid \Theta, \pi' \rightsquigarrow \langle x', c'\rangle \Vdash [\overline{u_i/X_i}]\pi \rightsquigarrow \langle x[\overline{u_i^t}][\![\overline{\gamma_j}]\!](\overline{e_j^t}), c[\overline{u_i^t}]\rangle$ where
  $(\forall \overline{X_i}.\overline{\pi_j} \Rightarrow \pi \rightsquigarrow \langle x, c\rangle) \in (\Theta, \pi' \rightsquigarrow \langle x', c'\rangle)$,
  $\dfrac{}{\Gamma \mid \Theta, \pi' \rightsquigarrow \langle x', c'\rangle \vdash_{\mathrm{WF}} u_i \rightsquigarrow u_i^t}$ and
  $\dfrac{}{\Gamma \mid \Theta, \pi' \rightsquigarrow \langle x', c'\rangle \Vdash [\overline{u_i/X_i}]\pi_j \rightsquigarrow \langle e_j^t, \gamma_j\rangle}$.

  Assume $\Gamma \mid \Theta \Vdash \pi' \rightsquigarrow \langle e'^t, \gamma'\rangle$.

  If $(\forall \overline{X_i}.\overline{\pi_j} \Rightarrow \pi) \in \Theta$ we can assume $x \neq x'$, $c \neq c'$ by the bound variable renaming assumption.
  We can use IH to prove $\Gamma \mid \Theta \Vdash [\overline{u_i/X_i}]\pi \rightsquigarrow \langle x[\overline{u_i^t}][\![\overline{[\gamma'/c']\gamma_j}]\!](\overline{[e'^t/x'][\gamma'/c']e_j^t}), c[\overline{u_i^t}]\rangle$.

  Otherwise, $(\forall \overline{X_i}.\overline{\pi_j} \Rightarrow \pi \rightsquigarrow \langle x, c\rangle) = (\pi' \rightsquigarrow \langle x', c'\rangle)$.

  Then, as $\overline{u_i}, \overline{X_i}, \overline{e_j^t}$ and $\overline{\gamma_j}$ are all empty sequences, we need to show
  $\Gamma \mid \Theta \Vdash \pi' \rightsquigarrow \langle e'^t, \gamma'\rangle$ which holds from the assumption as required.

□

Lemma B.5 (Contraction of the constraint environment II)**.** *If* $\Gamma \mid \Theta, \pi' \rightsquigarrow \langle x, c\rangle \vdash_{\mathrm{WF}} \theta \rightsquigarrow \langle \sigma^t, \vartheta\rangle$ *and* $\Gamma \mid \Theta \Vdash \pi' \rightsquigarrow \langle e'^t, \gamma'\rangle$ *then* $\Gamma \mid \Theta \vdash_{\mathrm{WF}} \theta \rightsquigarrow \langle \sigma^t, \vartheta\rangle$.

Proof. Straightforward by induction on derivations $\Gamma \mid \Theta, \pi' \rightsquigarrow \langle x, c\rangle \vdash_{\mathrm{WF}} \theta \rightsquigarrow \langle \sigma^t, \vartheta\rangle$
using lemma B.4 and the IH. □

Lemma B.6 (Contraction of the constraint environment III)**.** *If* $\Gamma \mid \Theta, \pi' \rightsquigarrow \langle x, c\rangle \vdash_{\mathrm{WF}} \sigma \rightsquigarrow \sigma^t$ *and* $\Gamma \mid \Theta \Vdash \pi' \rightsquigarrow \langle e'^t, \gamma'\rangle$ *then* $\Gamma \mid \Theta \vdash_{\mathrm{WF}} \sigma \rightsquigarrow \sigma^t$.

Proof. If $\sigma \in \text{STYPE}$ then the proof is immediate from lemma B.4.
Otherwise $\sigma = \forall \overline{X_i}.\overline{\pi_j} \Rightarrow \tau$ for some $\overline{X_i}, \overline{\pi_j}, \tau$.
From lemma B.3 we get
$$\frac{}{\Gamma, \overline{X_i} \mid \Theta, \pi' \rightsquigarrow \langle x, c \rangle, \overline{\pi_j} \vdash_{\text{WF}} \pi_j \rightsquigarrow \langle \tau_j^t, \omega_j \rangle}\, j$$
and $\Gamma, \overline{X_i} \mid \Theta, \pi' \rightsquigarrow \langle x, c \rangle, \overline{\pi_j} \vdash_{\text{WF}} \tau \rightsquigarrow \tau^t$ and $\sigma^t = \forall \overline{X_i}.\forall \overline{\omega_j}.\text{fn}(\overline{\tau_j^t}) \to \tau^t$ for some $\overline{\tau_j^t}, \overline{\omega_j}$ and τ^t.
By lemmas B.5 and B.4 we have
$$\frac{}{\Gamma, \overline{X_i} \mid \Theta, \overline{\pi_j} \vdash_{\text{WF}} \pi_j \rightsquigarrow \langle \tau_j^t, \omega_j \rangle}\, j$$
and $\Gamma, \overline{X_i} \mid \Theta, \overline{\pi_j} \vdash_{\text{WF}} \tau \rightsquigarrow \tau^t$.
We can then use (wf-tscheme) to prove $\Gamma \mid \Theta \vdash_{\text{WF}} \sigma \rightsquigarrow \sigma^t$ as required. □

Lemma B.7 (Type substitution I)**.**

1. *If* $\Gamma, X \mid \Theta \vdash_{\text{WF}} u' \rightsquigarrow u'^t$ *and* $[u/X]\Gamma \mid [u/X]\Theta \vdash_{\text{WF}} u \rightsquigarrow u^t$
   then $[u/X]\Gamma \mid [u/X]\Theta \vdash_{\text{WF}} [u/X]u' \rightsquigarrow [u^t/X]u'^t$.

2. *If* $\Gamma, X \mid \Theta \Vdash \pi \rightsquigarrow \langle e^t, \gamma \rangle$ *and* $[u/X]\Gamma \mid [u/X]\Theta \vdash_{\text{WF}} u \rightsquigarrow u^t$
   then $[u/X]\Gamma \mid [u/X]\Theta \Vdash [u/X]\pi \rightsquigarrow \langle [u/X]e^t, [u/X]\gamma \rangle$

3. *If* $\Gamma, X \mid \Theta \Vdash u_1 \sim u_2 \rightsquigarrow \gamma$ *and* $[u/X]\Gamma \mid [u/X]\Theta \vdash_{\text{WF}} u \rightsquigarrow u^t$
   then $[u/X]\Gamma \mid [u/X]\Theta \Vdash [u/X](u_1 \sim u_2) \rightsquigarrow [u/X]\gamma$.

Proof. By simultaneous induction on derivations
$\Gamma, X \mid \Theta \vdash_{\text{WF}} u' \rightsquigarrow u'^t$,
$\Gamma, X \mid \Theta \Vdash \pi \rightsquigarrow \langle e^t, \gamma \rangle$ and
$\Gamma, X \mid \Theta \Vdash u_1 \sim u_2 \rightsquigarrow \gamma$.
In most cases it is a straightforward use of the induction hypothesis.
The only special case is (wf-var): $\Gamma, X \mid \Theta \vdash_{\text{WF}} X' \rightsquigarrow X'$ where $X' \in (\Gamma, X)$.
If $X = X'$ then $[u/X]\Gamma \mid [u/X]\Theta \vdash_{\text{WF}} u \rightsquigarrow u^t$ from assumption as required. □

Lemma B.8 (Type substitution II)**.** *If* $\Gamma, X \mid \Theta \vdash_{\text{WF}} \theta \rightsquigarrow \langle \sigma^t, \gamma \rangle$ *and* $[u/X]\Gamma \mid [u/X]\Theta \vdash_{\text{WF}} u \rightsquigarrow u^t$ *then* $[u/X]\Gamma \mid [u/X]\Theta \vdash_{\text{WF}} [u/X]\theta \rightsquigarrow \langle [u^t/X]\tau^t, [u^t/X]\gamma \rangle$.

Proof. Straightforward by induction on derivations $\Gamma, X \mid \Theta \vdash_{\text{WF}} \theta \rightsquigarrow \langle \sigma^t, \gamma \rangle$ using IH and lemma B.7.
In case (wf-cscheme) we can assume that $X \notin \overline{X_i}$ using the bound variable renaming convention. □

Lemma B.9 (Type substitution III)**.** *If* $\Gamma, X \mid \Theta \vdash_{\text{WF}} \sigma \rightsquigarrow \sigma^t$ *and* $[u/X]\Gamma \mid [u/X]\Theta \vdash_{\text{WF}} u \rightsquigarrow u^t$ *then* $[u/X]\Gamma \mid [u/X]\Theta \vdash_{\text{WF}} [u/X]\sigma \rightsquigarrow [u^t/X]\sigma^t$.

Proof. If $\sigma \in \text{STYPE}$ then the proof is immediate from lemma B.7.
Otherwise $\sigma = \forall\overline{X_i}.\overline{\pi_j} \Rightarrow \tau$ for some $\overline{X_i}, \overline{\pi_j}, \tau$ where we can assume $X \notin \overline{X_i}$.
From lemma B.3 we get
$$\frac{}{\Gamma, X, \overline{X_i} \mid \Theta, \overline{\pi_j} \vdash_{\text{WF}} \pi_j \rightsquigarrow \langle \tau_j^t, \omega_j \rangle}\, j$$
and $\Gamma, X, \overline{X_i} \mid \Theta, \overline{\pi_j} \vdash_{\text{WF}} \tau \rightsquigarrow \tau^t$ and $\sigma^t = \forall\overline{X_i}.\forall\overline{\omega_j}.\mathsf{fn}(\overline{\tau_j^t}) \to \tau^t$ for some $\overline{\tau_j^t}, \overline{\omega_j}$ and τ^t.
By lemmas B.8 and B.7 we have
$$\frac{}{[u/X]\Gamma, \overline{X_i} \mid [u/X]\Theta, \overline{[u/X]\pi_j} \vdash_{\text{WF}} [u/X]\pi_j \rightsquigarrow \langle [u/X]\tau_j^t, [u/X]\omega_j \rangle}\, j$$
and $[u/X]\Gamma, \overline{X_i} \mid [u/X]\Theta, \overline{[u/X]\pi_j} \vdash_{\text{WF}} [u/X]\tau \rightsquigarrow [u/X]\tau^t$.
We can then use (wf-tscheme) to prove $[u/X]\Gamma \mid [u/X]\Theta \vdash_{\text{WF}} [u/X]\sigma \rightsquigarrow [u/X]\sigma^t$ as required. □

Lemma B.10 (Weakening I)**.**

1. *If* $\Gamma \mid \Theta \vdash_{\text{WF}} u \rightsquigarrow u^t$ *then* $\Gamma' \mid \Theta' \vdash_{\text{WF}} u \rightsquigarrow u^t$ *for any* $\Gamma' \supseteq \Gamma$ *and* $\Theta' \supseteq \Theta$.
2. *If* $\Gamma \mid \Theta \Vdash \pi \rightsquigarrow \langle e^t, \gamma \rangle$ *then* $\Gamma' \mid \Theta' \Vdash \pi \rightsquigarrow \langle e^t, \gamma \rangle$ *for any* $\Gamma' \supseteq \Gamma$ *and* $\Theta' \supseteq \Theta$.
3. *If* $\Gamma \mid \Theta \Vdash u_1 \sim u_2 \rightsquigarrow \gamma$ *then* $\Gamma' \mid \Theta' \Vdash u_1 \sim u_2 \rightsquigarrow \gamma$ *for any* $\Gamma' \supseteq \Gamma$ *and* $\Theta' \supseteq \Theta$.

Proof. Straightforward by simultaneous induction on derivations
$\Gamma \mid \Theta \vdash_{\text{WF}} u' \rightsquigarrow u'^t$,
$\Gamma \mid \Theta \Vdash \pi \rightsquigarrow \langle e^t, \gamma \rangle$ and
$\Gamma \mid \Theta \Vdash u_1 \sim u_2 \rightsquigarrow \gamma$. □

Lemma B.11 (Weakening II)**.** *If* $\Gamma \mid \Theta \vdash_{\text{WF}} \theta \rightsquigarrow \langle \sigma^t, \vartheta \rangle$ *then* $\Gamma' \mid \Theta' \vdash_{\text{WF}} \theta \rightsquigarrow \langle \sigma^t, \vartheta \rangle$ *for any* $\Gamma' \supseteq \Gamma$ *and* $\Theta' \supseteq \Theta$.

Proof. Straightforward by induction on derivations $\Gamma \mid \Theta \vdash_{\text{WF}} \theta \rightsquigarrow \langle \sigma^t, \vartheta \rangle$ using IH and lemma B.10. □

Lemma B.12 (Weakening III)**.** *If* $\Gamma \mid \Theta \vdash_{\text{WF}} \sigma \rightsquigarrow \sigma^t$ *then* $\Gamma' \mid \Theta' \vdash_{\text{WF}} \sigma \rightsquigarrow \sigma^t$ *for any* $\Gamma' \supseteq \Gamma$ *and* $\Theta' \supseteq \Theta$.

Proof. If $\sigma \in \text{STYPE}$ then the result is immediate from lemma B.10.
Otherwise, the only rule that could apply as the last rule in the derivation $\Gamma \mid \Theta \vdash_{\text{WF}} \sigma \rightsquigarrow \sigma^t$ is (wf-tscheme).
Using lemmas B.10 and B.11 on the rule's side conditions we get
$\Gamma' \mid \Theta' \vdash_{\text{WF}} \sigma \rightsquigarrow \sigma^t$ as required. □

Lemma B.13 (Weakening IV)**.** *If* $\Gamma \mid \Theta \vdash e : \tau \rightsquigarrow e^t$ *then* $\Gamma' \mid \Theta' \vdash e : \tau \rightsquigarrow e^t$ *for any* $\Gamma' \supseteq \Gamma$ *and* $\Theta' \supseteq \Theta$.

Proof. Straightforward induction on derivations $\Gamma \mid \Theta \vdash e : \tau \rightsquigarrow e^t$ using IH and lemma B.10 when required. □

Lemma B.14 (Entailed constraints are well-formed)**.** *Suppose* $\langle \Gamma, \Theta \rangle \rightsquigarrow \langle \_, \_ \rangle$.

1. *If* $\Gamma \mid \Theta \Vdash \pi \rightsquigarrow \langle e^t, \gamma \rangle$ *then* $\Gamma \mid \Theta \vdash_{\text{WF}} \pi \rightsquigarrow \langle \tau^t, \omega \rangle$ *for some* τ^t, ω.
2. *If* $\Gamma \mid \Theta \Vdash u_1 \sim u_2 \rightsquigarrow \gamma$ *and* $\Gamma \mid \Theta \vdash_{\text{WF}} u_1 \rightsquigarrow u_1^t$ *then then there is some type* u_2^t *such that* $\Gamma \mid \Theta \vdash_{\text{WF}} u_2 \rightsquigarrow u_2^t$.
3. *If* $\Gamma \mid \Theta \Vdash u_1 \sim u_2 \rightsquigarrow \gamma$ *and* $\Gamma \mid \Theta \vdash_{\text{WF}} u_2 \rightsquigarrow u_2^t$*, then there is some type* u_1^t *such that* $\Gamma \mid \Theta \vdash_{\text{WF}} u_1 \rightsquigarrow u_1^t$ *and.*

Proof. Proof by simultaneous induction on derivations
$\Gamma \mid \Theta \Vdash u_1 \sim u_2 \rightsquigarrow \gamma$ and $\Gamma \mid \Theta \Vdash \pi \rightsquigarrow \langle e^t, \gamma \rangle$:

- **Case** (c-ext): $\Gamma \mid \Theta \Vdash [\overline{u_i/X_i}]\pi \rightsquigarrow \langle x[\overline{u_i^t}][\![\overline{\gamma_j}]\!](\overline{e_j^t}), c[\overline{u_i^t}] \rangle$ and
  $((x,c) : \forall \overline{X_i}.\overline{\pi_j} \Rightarrow \pi) \in \Theta$,
  $\dfrac{\overline{\Gamma \mid \Theta \vdash_{\text{WF}} u_i \rightsquigarrow u_i^t},}{\Gamma \mid \Theta \Vdash [\overline{u_i/X_i}]\pi_j \rightsquigarrow (e_j^t, \gamma_j)}$.

  From well-formedness of Θ we know $\Gamma \mid \Theta \vdash_{\text{WF}} \forall \overline{X_i}.\overline{\pi_j} \Rightarrow \pi \rightsquigarrow \langle \sigma^t, \vartheta \rangle$ for some σ^t and ϑ.
  From lemma B.3 we get $\overline{\Gamma, \overline{X_i} \mid \Theta \vdash_{\text{WF}} \pi_j \rightsquigarrow \langle \tau_j^t, \omega_j \rangle}^{j}$
  and $\Gamma, \overline{X_i} \mid \Theta, \overline{\pi_j} \vdash_{\text{WF}} \pi \rightsquigarrow \langle \tau^t, u_1^t \sim u_2^t \rangle$
  and $\sigma^t = \forall \overline{X_i}.\forall \overline{\omega_j}.\mathsf{fn}(\overline{\tau_j^t}) \to \tau^t$
  and $\vartheta = \forall \overline{X_i}.u_1^t \sim \forall \overline{X_i}.u_2^t$ for some $\overline{\tau_j^t}, \overline{\omega_j}, \overline{c_j}, u_1^t, u_2^t, \tau^t$.
  By the assumption that $\overline{X_i} \notin \Gamma, \Theta$ and lemma B.8,
  $\Gamma \mid \Theta, \overline{[\overline{u_i/X_i}]\pi_j} \vdash_{\text{WF}} [\overline{u_i/X_i}]\pi \rightsquigarrow \langle [\overline{u_i^t/X_i}]\tau^t, [\overline{u_i^t/X_i}]u_1^t \sim [\overline{u_i^t/X_i}]u_2^t \rangle$.
  From the ind. hyp. we get $\overline{\Gamma \mid \Theta \vdash_{\text{WF}} [\overline{u_i/X_i}]\pi_j \rightsquigarrow \langle \tau_j^t, \omega_j \rangle}$ for some $\overline{\tau_j^t}$ and $\overline{\omega_j}$.
  We can then use that with lemma B.5 to show
  $\Gamma \mid \Theta \vdash_{\text{WF}} [\overline{u_i/X_i}]\pi \rightsquigarrow \langle [\overline{u_i^t/X_i}]\tau^t, [\overline{u_i^t/X_i}]u_1^t \sim [\overline{u_i^t/X_i}]u_2^t \rangle$ as required.

- **Case** (c-treq1):
  $\Gamma \mid \Theta \Vdash u_2 : D\,\langle \overline{u_{2i}}, A \mapsto u_4 \rangle \rightsquigarrow \langle e^t \blacktriangleright S_D\,\langle \gamma_2, \overline{\gamma_i} \rangle, A_D\,\langle \mathsf{sym}\,(\gamma_2), \overline{\mathsf{sym}\,(\gamma_i)} \rangle \circ \gamma_1 \circ \gamma_3 \rangle$ where
  $\Gamma \vdash u_2 : D\,\langle \overline{u_{2i}} \rangle$,
  $\Gamma \mid \Theta \Vdash u_1 : D\,\langle \overline{u_{1i}}, A \mapsto u_3 \rangle \rightsquigarrow \langle e^t, \gamma_1 \rangle$,
  $\dfrac{\Gamma \mid \Theta \Vdash u_1 \sim u_2 \rightsquigarrow \gamma_2,}{\overline{\Gamma \mid \Theta \Vdash u_{1i} \sim u_{2i} \rightsquigarrow \gamma_i}^{i}}$ and
  $\Gamma \mid \Theta \Vdash u_3 \sim u_4 \rightsquigarrow \gamma_3$.
  By IH, $\Gamma \mid \Theta \vdash_{\text{WF}} u_1 : D\,\langle \overline{u_{1i}}, A \mapsto u_3 \rangle \rightsquigarrow \langle \tau^t, \omega \rangle$ for some τ^t, ω.
  By lemma B.3, $\Gamma \mid \Theta \vdash_{\text{WF}} u_1 \rightsquigarrow u_1^t$, and $\overline{\Gamma \mid \Theta \vdash_{\text{WF}} u_{1i} \rightsquigarrow u_{1i}^t}$,
  and $\Gamma \mid \Theta, u : D\,\langle \overline{u_{1i}} \rangle \vdash_{\text{WF}} u_3 \rightsquigarrow u_3^t$, and $\Gamma \vdash A_D$
  for some types u_1^t, $\overline{u_{1i}^t}$, u_3^t.

By lemma B.5, $\Gamma \mid \Theta \vdash_{\text{WF}} u_3 \rightsquigarrow u_3^t$.

We can then use the IH get
$\Gamma \mid \Theta \vdash_{\text{WF}} u_2 \rightsquigarrow u_2^t$,
$\overline{\Gamma \mid \Theta \vdash_{\text{WF}} u_{2i} \rightsquigarrow u_{2i}^t}$, and
$\Gamma \mid \Theta \vdash_{\text{WF}} u_4 \rightsquigarrow u_4^t$ for some types u_2^t, $\overline{u_2^t}$ and u_4^t.

By lemma B.10, $\Gamma \mid \Theta, u_2 : D\,\langle \overline{u_{2i}} \rangle \vdash_{\text{WF}} u_4 \rightsquigarrow u_4^t$.

Then, by rule (wf-treqcons),
$\Gamma \mid \Theta \vdash_{\text{WF}} u_2 : D\,\langle \overline{u_{2i}}, A \mapsto u_4 \rangle \rightsquigarrow \langle S_D\,\langle u_2^t, \overline{u_{2i}^t} \rangle, A_D\,\langle u_2^t, \overline{u_{2i}^t} \rangle \sim u_4^t \rangle$ as required.

- **Case** (c-treq2): $\Gamma \mid \Theta \Vdash u_2 : D\,\langle \overline{u_{2i}}, A \mapsto \star \rangle \rightsquigarrow \langle e^t \blacktriangleright S_D\,\langle \gamma_2, \overline{\gamma_i} \rangle, A_D\,\langle u^t, \overline{u_i^t} \rangle \rangle$ where
  $\Gamma \vdash u_2 : D\,\langle \overline{u_{2i}} \rangle$,
  $\Gamma \mid \Theta \Vdash u_1 : D\,\langle \overline{u_{1i}}, A \mapsto \star \rangle \rightsquigarrow \langle e^t, \gamma_1 \rangle$,
  $\Gamma \mid \Theta \Vdash u_1 \sim u_2 \rightsquigarrow \gamma_2$,
  $\overline{\Gamma \mid \Theta \Vdash u_{1i} \sim u_{2i} \rightsquigarrow \gamma_i}^{\,i}$,
  $\Gamma \mid \Theta \vdash_{\text{WF}} u_2 \rightsquigarrow u_2^t$,
  $\overline{\Gamma \mid \Theta \vdash_{\text{WF}} u_{2i} \rightsquigarrow u_{2i}^t}$.

  By IH, $\Gamma \mid \Theta \vdash_{\text{WF}} u_1 : D\,\langle \overline{u_{1i}}, A \mapsto u_3 \rangle \rightsquigarrow \langle \tau^t, \omega \rangle$ for some τ^t, ω.

  By lemma B.3, $\Gamma \vdash A_D$.

  Then, by rule (wf-trcons),
  $\Gamma \mid \Theta \vdash_{\text{WF}} u_2 : D\,\langle \overline{u_{2i}}, A \mapsto \star \rangle \rightsquigarrow \langle S_D\,\langle u_2^t, \overline{u_{2i}^t} \rangle, A_D\,\langle u_2^t, \overline{u_{2i}^t} \rangle \sim A_D\,\langle u_2^t, \overline{u_{2i}^t} \rangle \rangle$ as required.

- **Case** (c-astar): $\Gamma \mid \Theta \Vdash u : D\,\langle \overline{u_i}, A \mapsto \star \rangle \rightsquigarrow \langle e^t, A_D\,\langle u^t, \overline{u_i^t} \rangle \rangle$ where
  $\Gamma \mid \Theta \Vdash u : D\,\langle \overline{u_i}, A \mapsto u_A \rangle \rightsquigarrow \langle e^t, \gamma \rangle$,
  $\Gamma \mid \Theta \vdash_{\text{WF}} u \rightsquigarrow u^t$ and
  $\overline{\Gamma \mid \Theta \vdash_{\text{WF}} u_i \rightsquigarrow u_i^t}$.

  By IH, there are some τ^t, ω such that $\Gamma \mid \Theta \vdash_{\text{WF}} u : D\,\langle \overline{u_i}, A \mapsto \star \rangle \rightsquigarrow \langle \tau^t, \omega \rangle$.

  By lemma B.3, $\Gamma \vdash u : D\,\langle \overline{u_i} \rangle$,
  $\Gamma \vdash A_D$ and $\tau^t = S_D\,\langle u^t, \overline{u_i^t} \rangle$.

  We can then use (wf-trcons) to prove
  $\Gamma \mid \Theta \vdash_{\text{WF}} u : D\,\langle \overline{u_i}, A \mapsto \star \rangle \rightsquigarrow \langle \tau^t, A_D\,\langle u^t, \overline{u_i^t} \rangle \sim A_D\,\langle u^t, \overline{u_i^t} \rangle \rangle$.

- **Case** (eq-sep): $\Gamma \mid \Theta \Vdash A_D\,\langle u, \overline{u_i} \rangle \sim u_A \rightsquigarrow \gamma$ and
  $\Gamma \mid \Theta \Vdash u : D\,\langle \overline{u_i}, A \sim u_A \rangle \rightsquigarrow \langle e^t, \gamma \rangle$

  By IH $\Gamma \mid \Theta \vdash_{\text{WF}} u : D\,\langle \overline{u_i}, A \sim u_A \rangle \rightsquigarrow \langle \tau^t, \omega \rangle$ for some τ^t, ω.

  Lemma B.3 gives us $\Gamma \mid \Theta \vdash_{\text{WF}} u \rightsquigarrow u^t$,
  $\overline{\Gamma \mid \Theta \vdash_{\text{WF}} u_i \rightsquigarrow u_i^t}$,

$\Gamma \mid \Theta, u : D\,\langle \overline{u_i} \rangle \vdash_{\mathrm{WF}} u_A \rightsquigarrow u_A^t$,
and $\Gamma \vdash A_D$ for some types $u^t, \overline{u_i^t}$ and u_A^t.

Using (wf-atype) we get $\Gamma \mid \Theta \vdash_{\mathrm{WF}} A_D\,\langle u, \overline{u_i^t} \rangle \rightsquigarrow A_D\,\langle u^t, \overline{u_i^t} \rangle$.

Using lemma B.4 we get
$\Gamma \mid \Theta \vdash_{\mathrm{WF}} u_A \rightsquigarrow u_A^t$ as required.

- **Case** (eq-refl): $\Gamma \mid \Theta \Vdash u \sim u \rightsquigarrow u^t$
  Trivial.

- **Case** (eq-trans): $\Gamma \mid \Theta \Vdash u_1 \sim u_2 \rightsquigarrow \gamma_1 \circ \gamma_2$ and
  $\Gamma \mid \Theta \Vdash u_1 \sim u_3 \rightsquigarrow \gamma_1$ and
  $\Gamma \mid \Theta \Vdash u_3 \sim u_2 \rightsquigarrow \gamma_2$.

  Suppose $\Gamma \mid \Theta \vdash_{\mathrm{WF}} u_1 \rightsquigarrow u_1^t$.

  By IH $\Gamma \mid \Theta \vdash_{\mathrm{WF}} u_3 \rightsquigarrow u_3^t$ and then $\Gamma \mid \Theta \vdash_{\mathrm{WF}} u_2 \rightsquigarrow u_2^t$.

  If, on the other hand, $\Gamma \mid \Theta \vdash_{\mathrm{WF}} u_2 \rightsquigarrow u_2^t$,
  then by IH $\Gamma \mid \Theta \vdash_{\mathrm{WF}} u_3 \rightsquigarrow u_3^t$ for some u_3^t
  and $\Gamma \mid \Theta \vdash_{\mathrm{WF}} u_1 \rightsquigarrow u_1^t$ for some u_1^t as required.

- **Case** (eq-sym): $\Gamma \mid \Theta \Vdash u_1 \sim u_2 \rightsquigarrow \mathrm{sym}\ \gamma$ and
  $\Gamma \mid \Theta \Vdash u_2 \sim u_1 \rightsquigarrow \gamma$.

  If $\Gamma \mid \Theta \vdash_{\mathrm{WF}} u_1 \rightsquigarrow u_1^t$ then we can use the IH to get $\Gamma \mid \Theta \vdash_{\mathrm{WF}} u_2 \rightsquigarrow u_2^t$ as required.

  The symmetric case is analogous.

- **Case** (eq-struct): $\Gamma \mid \Theta \Vdash S\,\langle \overline{u_i} \rangle \sim S\,\langle \overline{u_i'} \rangle \rightsquigarrow S\,\langle \overline{\gamma_i} \rangle$ where
  $S\,\langle \overline{X_i} \rangle\,\{\ldots\} \in \Gamma$, and $[\overline{u_i/X_i}][\overline{u_i'/X_i}]$ *defined*, and $\overline{\Gamma \mid \Theta \Vdash u_i \sim u_i' \rightsquigarrow \gamma_i}$.

  Suppose $\Gamma \mid \Theta \vdash_{\mathrm{WF}} S\,\langle \overline{u_i} \rangle \rightsquigarrow u^t$.

  From lemma B.3 we get $\overline{\Gamma \mid \Theta \vdash_{\mathrm{WF}} u_i \rightsquigarrow u_i^t}$ and $u^t = S\,\langle \overline{u_i^t} \rangle$ and $S\,\langle \_ \rangle\,\{\ldots\} \in \Gamma$).

  From IH we get $\overline{\Gamma \mid \Theta \vdash_{\mathrm{WF}} u_i' \rightsquigarrow u_i^t}$ for some $\overline{u_i^t}$.

  Then by (wf-struct) $\Gamma \mid \Theta \vdash_{\mathrm{WF}} S\,\langle \overline{u_i'} \rangle \rightsquigarrow S\,\langle \overline{u_i'^t} \rangle$ as required.

  If we assume instead $\Gamma \mid \Theta \vdash_{\mathrm{WF}} S\,\langle \overline{u_i'} \rangle$ then the proof is analogous.

- **Case** (eq-ref): Straightforward use of IH, lemma B.3 and rule (wf-ref) as in the case for (eq-struct).

- **Case** (eq-obj): $\Gamma \mid \Theta \Vdash \exists T : D\,\langle \overline{u_{1i}}, \overline{A_{Dj}\,\langle T, \overline{u_{1sj}} \rangle \sim u_{1j}} \rangle \sim \exists T : D\,\langle \overline{u_{2i}}, \overline{A_{Dj}\,\langle T, \overline{u_{2sj}} \rangle \sim u_{2j}} \rangle$
  $\rightsquigarrow (\exists T, \& T, S_D\,\langle T, \overline{\gamma_i} \rangle, \overline{A_{Dj}\,\langle T, \overline{\gamma_{sj}} \rangle \sim \gamma_j})$ where
  $\mathsf{Self}_{\mathsf{U}} : D\,\langle \overline{X_i} \rangle$,

$[\overline{u_{1i}/X_i}][\overline{u_{2i}/X_i}]$ *defined*,
$\overline{\mathsf{Self}_{\mathbf{U}} : D_j \langle \overline{X_{sj}} \rangle}^{j}$,
$\overline{[\overline{u_{1sj}/X_{sj}}][\overline{u_{2sj}/X_{sj}}] \textit{ defined}}^{j}$,
$\overline{\Gamma \mid \Theta \Vdash u_{1i} \sim u_{2i} \rightsquigarrow \gamma_i}$,
$\overline{\Gamma, T \mid \Theta, T : D \langle \overline{u_{1j}}, A \mapsto \_ \rangle \Vdash u_{1sj} \sim u_{2sj} \rightsquigarrow \gamma_{sj}}$ and
$\overline{\Gamma \mid \Theta \Vdash u_{1j} \sim u_{2j} \rightsquigarrow \gamma_j}$

Suppose $\Gamma \mid \Theta \vdash_{\text{WF}} \exists T : D \langle \overline{u_{1i}}, \overline{A_{Dj} \langle T, \overline{u_{1sj}} \rangle \sim u_{1j}} \rangle \rightsquigarrow u_1^t$.

From lemma B.3 we know that there are some $\overline{u_{1i}^t}, \overline{\overline{u_{1sj}^t}}, \overline{u_{1j}^t}$ such that
$\Gamma \vdash D$ *obj-safe*
$\overline{\Gamma \mid \Theta \vdash_{\text{WF}} u_{1i} \rightsquigarrow u_{1i}^t}$
$\overline{\overline{\Gamma, T \mid \Theta, T : D \langle \overline{u_{1i}}, A \mapsto \_ \rangle \vdash_{\text{WF}} u_{1sj} \rightsquigarrow u_{1sj}^t}^{s}}^{j}$ ahd
$\overline{\Gamma \mid \Theta \vdash_{\text{WF}} u_{1j} \rightsquigarrow u_{1j}^t}$.

Then, by IH, there are some $\overline{u_{2i}^t}, \overline{\overline{u_{2sj}^t}}, \overline{u_{2j}^t}$ such that
$\overline{\Gamma \mid \Theta \vdash_{\text{WF}} u_{2i} \rightsquigarrow u_{2i}^t}$,
$\overline{\overline{\Gamma, T \mid \Theta, T : D \langle \overline{u_{1j}}, A \mapsto \_ \rangle \vdash_{\text{WF}} u_{2sj} \rightsquigarrow u_{2sj}^t}}$ and
$\overline{\Gamma \mid \Theta \vdash_{\text{WF}} u_{2j} \rightsquigarrow u_{2j}^t}$.

By lemma B.10 we get
$\overline{\overline{\Gamma, T \mid \Theta, T : D \langle \overline{u_{1j}}, A \mapsto \_ \rangle, T : D \langle \overline{u_{2j}}, A \mapsto \_ \rangle \vdash_{\text{WF}} u_{2sj} \rightsquigarrow u_{2sj}^t}}$ and
$\overline{\Gamma, T \mid \Theta, T : D \langle \overline{u_{1j}}, A \mapsto \_ \rangle \Vdash u_{1j} \sim u_{2j}}$.

Using (eq-sym) we get $\overline{\Gamma, T \mid \Theta, T : D \langle \overline{u_{2j}}, A \mapsto \_ \rangle \Vdash u_{2j} \sim u_{1j}}$.

Then using (c-ext) and (c-treq) we get $\Gamma, T \mid \Theta, T : D \langle \overline{u_{2j}}, A \mapsto \_ \rangle \Vdash T : D \langle \overline{u_{1j}}, A \mapsto \_ \rangle$.

We can then use lemma B.4 to prove
$\overline{\overline{\Gamma, T \mid \Theta, T : D \langle \overline{u_{2j}}, A \mapsto \_ \rangle \vdash_{\text{WF}} u_{2sj} \rightsquigarrow u_{2sj}^t}}$

Finally, using rules (wf-atype) and (wf-desc) we get
$\Gamma \mid \Theta \vdash_{\text{WF}} \exists T : D \langle \overline{u_{2i}}, \overline{A_{Dj} \langle T, \overline{u_{2sj}} \rangle \sim u_{2j}} \rangle \rightsquigarrow u_2^t$ for some u_2^t as required.

The proof of point 2. is similar.

- **Case** (eq-fun): Straightforward use of IH, lemma B.3 and rule (wf-fun) as in the case for (eq-struct).

- **Case** (eq-atype): $\Gamma \mid \Theta \Vdash A_D \langle u_1, \overline{u_{1i}} \rangle \sim A_D \langle u_2, \overline{u_{2i}} \rangle \rightsquigarrow A_D \langle \gamma, \overline{\gamma_i} \rangle$ where
$\Gamma \vdash \mathsf{Self}_{\mathbf{U}} : D \langle \overline{X_i} \rangle$,
$[u_1/\mathsf{Self}_{\mathbf{U}}][u_2/\mathsf{Self}_{\mathbf{U}}][\overline{u_{1i}/X_i}][\overline{u_{2i}/X_i}]$ *defined*,

$\Gamma \mid \Theta \Vdash u_1 \sim u_2 \rightsquigarrow \gamma$, and
$\overline{\Gamma \mid \Theta \Vdash u_{1i} \sim u_{2i} \rightsquigarrow \gamma_i}$.

Suppose $\Gamma \mid \Theta \vdash_{\text{WF}} A_D \langle u_1, \overline{u_{1i}} \rangle \rightsquigarrow u^t$.

From lemma B.3 we get $\Gamma \mid \Theta \Vdash u_1 : D \langle \overline{u_{1i}}, A \mapsto \_ \rangle$,
$\Gamma \mid \Theta \vdash_{\text{WF}} u_1 \rightsquigarrow {u_1}^t$ and
$\overline{\Gamma \mid \Theta \vdash_{\text{WF}} u_i \rightsquigarrow u_1{}_i^t}$
for some $u^t, \overline{u_1{}_i^t}$.

From the IH we get $\Gamma \mid \Theta \vdash_{\text{WF}} u_2 \rightsquigarrow u_2^t$ and $\overline{\Gamma \mid \Theta \vdash_{\text{WF}} u_{2i} \rightsquigarrow u_{2i}^t}$.

Using (c-treq2) we get $\Gamma \mid \Theta \Vdash u_2 : D \langle \overline{u_{2i}}, A \mapsto \star \rangle$.

Then we can use (wf-atype) to conclude
$\Gamma \mid \Theta \vdash_{\text{WF}} A_D \langle u_2, \overline{u_{2i}} \rangle \rightsquigarrow A_D \langle u_2^t, \overline{u_{2i}^t} \rangle$ as required.

If we suppose $\Gamma \mid \Theta \vdash_{\text{WF}} A_D \langle u_2, \overline{u_{2i}} \rangle \rightsquigarrow u'^t$ instead, we can use an analogous argument.

□

Lemma B.15 (Type environment contraction)**.**

1. *If* $\Gamma, x : \tau \mid \Theta \vdash_{\text{WF}} u \rightsquigarrow u^t$ *then* $\Gamma \mid \Theta \vdash_{\text{WF}} u \rightsquigarrow u^t$.
2. *If* $\Gamma, x : \tau \mid \Theta \Vdash \pi \rightsquigarrow \langle e^t, \gamma \rangle$ *then* $\Gamma \mid \Theta \Vdash \pi \rightsquigarrow \langle e^t, \gamma \rangle$
3. *If* $\Gamma, x : \tau \mid \Theta \Vdash u_1 \sim u_2 \rightsquigarrow \gamma$ *then* $\Gamma \mid \Theta \Vdash u_1 \sim u_2 \rightsquigarrow \gamma$.

Proof. Immediate from the well-formedness and constraint entailment judgments as type assignments in the typing environment Γ are never used in the rules. □

Lemma B.16 (Translation of types preserves kinds)**.** *If* $\Gamma \mid \Theta \vdash_{\text{WF}} \tau \rightsquigarrow u^t$ *then* $u^t \in \text{SType}^t$.

Proof. Straightforward induction on derivations $\Gamma \mid \Theta \vdash_{\text{WF}} \tau \rightsquigarrow u^t$. □

Lemma B.17 (Well-typed terms have well-formed types)**.** *Suppose* $\langle \Gamma, \Theta \rangle \rightsquigarrow \langle \_, \_ \rangle$.
If $\Gamma \mid \Theta \vdash e : \tau \rightsquigarrow e^t$ *then there is some* τ^t *such that* $\Gamma \mid \Theta \vdash_{\text{WF}} \tau \rightsquigarrow \tau^t$.

Proof. By induction on derivations $\Gamma \mid \Theta \vdash e : \tau \rightsquigarrow e^t$:

- **Case** (sub): $\Gamma \mid \Theta \vdash e : \tau_2 \rightsquigarrow e^t \blacktriangleright \gamma$ where
  $\Gamma \mid \Theta \vdash e : \tau_1 \rightsquigarrow e^t$ and
  $\Gamma \mid \Theta \Vdash \tau_1 \sim \tau_2 \rightsquigarrow \gamma$.

  By IH, $\Gamma \mid \Theta \vdash_{\text{WF}} \tau_1 \rightsquigarrow \tau_1^t$ for some τ_1^t.

  From lemma B.14 we get $\Gamma \mid \Theta \vdash_{\text{WF}} \tau_1 \sim \tau_2 \rightsquigarrow \omega$ for some ω
  and from lemma B.3 we get $\Gamma \mid \Theta \vdash_{\text{WF}} \tau_2 \rightsquigarrow \tau_2^t$ for some τ_2^t as required.

- **Case** (as): Straightforward use of IH.
- **Case** (unit): Holds by (wf-unit).
- **Case** (let-un): $\Gamma \mid \Theta \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2 \rightsquigarrow \mathsf{let}\ x : \tau^t = e_1^t\ \mathsf{in}\ e_2^t$ where
  $\Gamma \mid \Theta \vdash e_1 : \tau_1 \rightsquigarrow e_1^t$,
  $\Gamma, x : \tau_1 \mid \Theta \vdash e_2 : \tau_2 \rightsquigarrow e_2^t$ and
  $\Gamma \mid \Theta \vdash_{\mathrm{WF}} \tau_1 \rightsquigarrow \tau^t$.

  By IH, $\Gamma, x : \tau_1 \mid \Theta \vdash_{\mathrm{WF}} \tau_2 \rightsquigarrow \tau_2^t$ for some τ_2^t
  and by lemma B.15 we have $\Gamma \mid \Theta \vdash_{\mathrm{WF}} \tau_2 \rightsquigarrow \tau_2^t$ as required.
- **Case** (seq): Straightforward use of IH.
- **Case** (var): $\Gamma \mid \Theta \vdash x : [\overline{u_i/X_i}]\tau \rightsquigarrow x[\overline{u_i^t}][\![\overline{\gamma_j}]\!](\overline{e_j^t})$ where
  $(x : \forall \overline{X_i}.\overline{\pi_j} \Rightarrow \tau) \in \Gamma$ and
  $\overline{\Gamma \mid \Theta \vdash_{\mathrm{WF}} u_i \rightsquigarrow u_i^t}$ and
  $\overline{\Gamma \mid \Theta \Vdash [\overline{u_i/X_i}]\pi_j \rightsquigarrow \langle e_j^t, \gamma_j \rangle}^{\,j}$.

  From well-formedness of Γ, Θ we get $\Gamma \mid \Theta \vdash \forall \overline{X_i}.\overline{\pi_j} \Rightarrow \tau \rightsquigarrow \sigma^t$ for some σ^t.

  From lemma B.3 we get $\Gamma, \overline{X_i} \mid \Theta, \overline{\pi_j \rightsquigarrow \langle x_j, c_j \rangle} \vdash_{\mathrm{WF}} \tau \rightsquigarrow \tau^t$ for some τ^t.

  From lemma B.7 we get $\Gamma \mid \Theta, \overline{[\overline{u_i/X_i}]\pi_j \rightsquigarrow \langle x_j, c_j \rangle} \vdash_{\mathrm{WF}} [\overline{u_i/X_i}]\tau \rightsquigarrow [\overline{u_i/X_i}]\tau^t$.

  Then we can use lemma B.4 to get $\Gamma \mid \Theta \vdash_{\mathrm{WF}} [\overline{u_i/X_i}]\tau \rightsquigarrow [\overline{u_i/X_i}]\tau^t$ as required.
- **Case** (app): $\Gamma \mid \Theta \vdash e(\overline{e_i}) : \tau \rightsquigarrow e^t(\overline{e_i^t})$ where
  $\Gamma \mid \Theta \vdash e : \mathsf{fn}(\overline{\tau_i}) \to \tau \rightsquigarrow e^t$ and
  $\overline{\Gamma \mid \Theta \vdash e_i : \tau_i \rightsquigarrow e_i^t}$.

  By IH there is some τ'^t where $\Gamma \mid \Theta \vdash_{\mathrm{WF}} \mathsf{fn}(\overline{\tau_i}) \to \tau \rightsquigarrow \tau'^t$.

  By lemma B.3, there is some type τ^t such that $\Gamma \mid \Theta \vdash_{\mathrm{WF}} \tau \rightsquigarrow \tau^t$ as required.
- **Case** (ref): $\Gamma \mid \Theta \vdash \&lv : \&\tau \rightsquigarrow \&lv^t$ where
  $\Gamma \mid \Theta \vdash lv : \tau \rightsquigarrow lv^t$.

  By IH $\Gamma \mid \Theta \vdash_{\mathrm{WF}} \tau \rightsquigarrow \tau^t$ for some τ^t.

  We can then use (wf-ref) to get $\Gamma \mid \Theta \vdash_{\mathrm{WF}} \&\tau \rightsquigarrow \&\tau^t$ as required.
- **Case** (deref): $\Gamma \mid \Theta \vdash *e : \tau \rightsquigarrow *e^t$ where
  $\Gamma \mid \Theta \vdash e : \&\tau \rightsquigarrow e^t$.

  From IH there is some τ'^t where $\Gamma \mid \Theta \vdash_{\mathrm{WF}} \&\tau \rightsquigarrow \tau'^t$.

  From lemma B.3 we get $\tau'^t = \&u^t$ and $\Gamma \mid \Theta \vdash_{\mathrm{WF}} \tau \rightsquigarrow u^t$.

  From lemma B.16 we know that $u^t \in \textsc{Type}^t$ as required.

- **Case** (asgn): Immediate from (wf-unit).
- **Case** (new-struct): Immediate from (wf-struct).
- **Case** (proj): $\Gamma \mid \Theta \vdash e.x : [\overline{u_j/X_j}]\tau \rightsquigarrow e^t.x$ where
  $\Gamma \mid \Theta \vdash e : S\,\langle \overline{u_j} \rangle \rightsquigarrow e^t$ and
  $\Gamma(S) = \mathsf{struct}\ S\,\langle \overline{X_j} \rangle\,\{\overline{x_m : \tau_m}, x : \tau, \overline{x_n : \tau_n}\}$.

  From Γ's well-formedness and lemmas B.1 and B.2 we know that $\Gamma, \overline{X_j} \mid \Theta \vdash_{\mathrm{WF}} \tau \rightsquigarrow \tau^t$ for some τ^t.

  From IH we get $\Gamma \mid \Theta \vdash_{\mathrm{WF}} S\,\langle \overline{u_j} \rangle \rightsquigarrow \tau^t$ for some type τ^t.

  From lemma B.3 we get $\overline{\Gamma \mid \Theta \vdash_{\mathrm{WF}} u_j \rightsquigarrow u_j^t}$ for some types $\overline{u_j^t}$.

  We can then use lemma B.7 to get $\Gamma \mid \Theta \vdash_{\mathrm{WF}} [\overline{u_j/X_j}]\tau \rightsquigarrow [\overline{u_j^t/X_j}]\tau^t$ as required.
- **Case** (obj-cast): Immediate from the rule's side condition.

□

Lemma B.18 (Translation of constraints preserves types)**.** *Suppose* $\langle \Gamma, \Theta \rangle \rightsquigarrow \langle \Omega^t, \Gamma^t \rangle$.

1. *If* $\Gamma \mid \Theta \Vdash \pi \rightsquigarrow \langle e^t, \gamma \rangle$ *and* $\Gamma \mid \Theta \vdash_{\mathrm{WF}} \pi \rightsquigarrow \langle \tau^t, \omega \rangle$
   then $\Omega^t \mid \Gamma^t \mid \emptyset \vdash_T e^t : \tau^t$ *and* $\Omega^t \vdash_T \gamma : \omega$
2. *If* $\Gamma \mid \Theta \Vdash u_1 \sim u_2 \rightsquigarrow \gamma$, *and* $\Gamma \mid \Theta \vdash_{\mathrm{WF}} u_1 \rightsquigarrow u_1^t$ *and* $\Gamma \mid \Theta \vdash_{\mathrm{WF}} u_2 \rightsquigarrow u_2^t$
   then $\Omega^t \vdash_T \gamma : u_1^t \sim u_2^t$

Proof. By simultaneous induction on derivations $\Gamma \mid \Theta \Vdash \pi \rightsquigarrow \langle e^t, \gamma \rangle$ and $\Gamma \mid \Theta \Vdash u_1 \sim u_2 \rightsquigarrow \gamma$:

- **Case** (c-ext): $\Gamma \mid \Theta \Vdash [\overline{u_i/X_i}]\pi \rightsquigarrow \langle x[\overline{u_i^t}][\![\overline{\gamma_j}]\!](\overline{e_j^t}), c[\overline{u_i^t}] \rangle$ where
  $(\forall \overline{X_i}.\overline{\pi_j} \Rightarrow \pi \rightsquigarrow \langle x, c \rangle) \in \Theta$,
  $\overline{\Gamma \mid \Theta \vdash_{\mathrm{WF}} u_i \rightsquigarrow u_i^t}$ and
  $\overline{\Gamma \mid \Theta \Vdash [\overline{u_i/X_i}]\pi_j \rightsquigarrow (e_j^t, \gamma_j)}$

  From well-formedness of Θ we know $\Gamma \mid \Theta \vdash_{\mathrm{WF}} \forall \overline{X_i}.\overline{\pi_j} \Rightarrow \pi \rightsquigarrow \langle \sigma^t, \vartheta \rangle$ for some σ^t and ϑ.

  From lemma B.3 we get $\overline{\Gamma, \overline{X_i} \mid \Theta \vdash_{\mathrm{WF}} \pi_j \rightsquigarrow \langle \tau_j^t, \omega_j \rangle}^{\,j}$
  and $\Gamma, \overline{X_i} \mid \Theta, \overline{\pi_j} \vdash_{\mathrm{WF}} \pi \rightsquigarrow \langle \tau^t, u_1^t \sim u_2^t \rangle$
  and $\sigma^t = \forall \overline{X_i}.\forall \overline{\omega_j}.\mathsf{fn}(\overline{\tau_j^t}) \to \tau^t$
  and $\vartheta = \forall \overline{X_i}.u_1^t \sim \forall \overline{X_i}.u_2^t$ for some $\overline{\tau_j^t}, \overline{\omega_j}, u_1^t, u_2^t, \tau^t$.

  We can use lemma B.1 to conclude that $(x : \sigma^t) \in \Gamma^t$ and $(c : \vartheta) \in \Omega^t$.

  From the assumption $\overline{X_i} \notin \Gamma, \Theta$ and lemma B.8,
  $\overline{\Gamma \mid \Theta \vdash_{\mathrm{WF}} [\overline{u_i/X_i}]\pi_j \rightsquigarrow \langle [\overline{u_i^t/X_i}]\tau_j^t, [\overline{u_i^t/X_i}]\omega_j \rangle}$ and
  $\Gamma \mid \Theta, \overline{[\overline{u_i/X_i}]\pi_j} \vdash_{\mathrm{WF}} [\overline{u_i/X_i}]\pi \rightsquigarrow \langle [\overline{u_i^t/X_i}]\tau^t, [\overline{u_i^t/X_i}]u_1^t \sim [\overline{u_i^t/X_i}]u_2^t \rangle$.

Then, using lemma B.5 we get
$\Gamma \mid \Theta \vdash_{\text{WF}} [\overline{u_i/X_i}]\pi \rightsquigarrow \langle [\overline{u_i^t/X_i}]\tau^t, [\overline{u_i^t/X_i}]u_1^t \sim [\overline{u_i^t/X_i}]u_2^t \rangle$.

From IH we get $\overline{\Omega^t \mid \Gamma^t \mid \emptyset \vdash_T e_j^t : [\overline{u_i^t/X_i}]\tau_j^t}^{\,j}$ and $\overline{\Omega^t \vdash_T \gamma_j : [\overline{u_i^t/X_i}]\omega_j}$.

We can then use the typing rules of the target language as follows:
$\Omega^t \mid \Gamma^t \mid \emptyset \vdash_T x : \forall \overline{X_i}.\forall \overline{\omega_j}.\mathsf{fn}(\overline{\tau_j^t}) \to \tau^t$ by (t-var).
$\Omega^t \mid \Gamma^t \mid \emptyset \vdash_T x[\overline{u_i^t}] : \forall \overline{[\overline{u_i^t/X_i}]\omega_j}.\mathsf{fn}(\overline{[\overline{u_i^t/X_i}]\tau_j^t}) \to [\overline{u_i^t/X_i}]\tau^t$ by (t-tapp).
$\Omega^t \mid \Gamma^t \mid \emptyset \vdash_T x[\overline{u_i^t}][\![\overline{\gamma_j}]\!] : \mathsf{fn}(\overline{[\overline{u_i^t/X_i}]\tau_j^t}) \to [\overline{u_i^t/X_i}]\tau^t$ by (t-capp).
$\Omega^t \mid \Gamma^t \mid \emptyset \vdash_T x[\overline{u_i^t}][\![\overline{\gamma_j}]\!](\overline{e_j^t}) : [\overline{u_i^t/X_i}]\tau^t$ by (t-fapp) as required.

$\Omega^t \vdash_T c : \forall \overline{X_i}.u_1^t \sim \forall \overline{X_i}.u_2^t$ by (co-var).
$\Omega^t \vdash_T c[\overline{u_i^t}] : [\overline{u_i^t/X_i}]u_1^t \sim [\overline{u_i^t/X_i}]u_2^t$ by (co-tapp) as required.

- **Case** (c-treq1):
  $\Gamma \mid \Theta \Vdash u_2 : D\,\langle \overline{u_{2i}}, A \mapsto u_4 \rangle \rightsquigarrow \left\langle e^t \blacktriangleright S_D\,\langle \gamma_2, \overline{\gamma_i} \rangle, A_D\,\langle \mathsf{sym}\,(\gamma_2), \overline{\mathsf{sym}\,(\gamma_i)} \rangle \circ \gamma_1 \circ \gamma_3 \right\rangle$ where
  $\Gamma \mid \Theta \Vdash u_1 : D\,\langle \overline{u_{1i}}, A \mapsto u_3 \rangle \rightsquigarrow \langle e^t, \gamma_1 \rangle$,
  $\Gamma \vdash u_2 : D\,\langle \overline{u_{2i}} \rangle$,
  $\Gamma \mid \Theta \Vdash u_1 \sim u_2 \rightsquigarrow \gamma$,
  $\overline{\Gamma \mid \Theta \Vdash u_{1i} \sim u_{2i} \rightsquigarrow \gamma_i}^{\,i}$ and
  $\Gamma \mid \Theta \Vdash u_3 \sim u_4 \rightsquigarrow \gamma_3$.

  Suppose $\Gamma \mid \Theta \Vdash u_2 : D\,\langle \overline{u_{2i}}, A \mapsto u_4 \rangle \rightsquigarrow \langle \tau_2^t, \omega_2 \rangle$.
  By lemma B.3, there are some $u_2^t, \overline{u_{2i}^t}, u_4^t$ such that
  $\dfrac{\Gamma \mid \Theta \vdash_{\text{WF}} u_2 \rightsquigarrow u_2^t,}{\Gamma \mid \Theta \vdash_{\text{WF}} u_{2i} \rightsquigarrow u_{2i}^t,}$
  $\Gamma \mid \Theta, u_2 : D\,\langle \overline{u_{2i}}, A \mapsto \star \rangle \vdash_{\text{WF}} u_4 \rightsquigarrow u_4^t$,
  $\tau_2^t = S_D\,\langle u_2^t, \overline{u_{2i}^t} \rangle$ and
  $\omega_2 = A_D\,\langle u_2^t, \overline{u_{2i}^t} \rangle \sim u_4^t$.

  By lemma B.14 $\Gamma \mid \Theta \vdash_{\text{WF}} u_1 : D\,\langle \overline{u_{1i}}, A \mapsto u_3 \rangle \rightsquigarrow \langle \tau_1^t, \omega_1 \rangle$ for some τ_1^t, ω_1.

  By lemma B.3, there are some $u_1^t, \overline{u_{1i}^t}, u_3^t$ such that
  $\dfrac{\Gamma \mid \Theta \vdash_{\text{WF}} u_1 \rightsquigarrow u_1^t,}{\Gamma \mid \Theta \vdash_{\text{WF}} u_{1i} \rightsquigarrow u_{1i}^t,}$
  $\Gamma \mid \Theta, u_1 : D\,\langle \overline{u_{1i}}, A \mapsto \star \rangle \vdash_{\text{WF}} u_3 \rightsquigarrow u_3^t$,
  $\tau_1^t = S_D\,\langle u_1^t, \overline{u_{1i}^t} \rangle$ and
  $\omega_1 = A_D\,\langle u_1^t, \overline{u_{1i}^t} \rangle \sim u_3^t$.

  Using lemma B.4 we get $\Gamma \mid \Theta \vdash_{\text{WF}} u_3 \rightsquigarrow u_3^t$ and $\Gamma \mid \Theta \vdash_{\text{WF}} u_4 \rightsquigarrow u_4^t$.

  Then, by IH we get:
  $\Omega^t \vdash_T \gamma_1 : A_D\,\langle u_1^t, \overline{u_{1i}^t} \rangle \sim u_3^t$,

$\Omega^t \vdash_T \gamma_2 : u_1^t \sim u_2^t$,
$\overline{\Omega^t \vdash_T \gamma_i : u_{1i}^t \sim u_{2i}^t}$, and
$\Omega^t \vdash_T \gamma_3 : u_3^t \sim u_4^t$.

Using rule (co-struct) we get $\Omega^t \vdash_T S_D \langle \gamma, \overline{\gamma_i} \rangle : S_D \langle u_1^t, \overline{u_{1i}^t} \rangle \sim S_D \langle u_2^t, \overline{u_{2i}^t} \rangle$
and using (t-coerce) we get $\Omega^t \mid \Gamma^t \mid \emptyset \vdash_T e^t \blacktriangleright S_D \langle \gamma, \overline{\gamma_i} \rangle : S_D \langle u_2^t, \overline{u_{2i}^t} \rangle$.

Using (co-sym) and (co-atype) we get
$\Omega^t \vdash_T A_D \langle \mathsf{sym}\ (\gamma_2), \overline{\mathsf{sym}\ (\gamma_i)} \rangle : A_D \langle u_2^t, \overline{u_{2i}^t} \rangle \sim A_D \langle u_1^t, \overline{u_{1i}^t} \rangle$.
Then using (co-sym) twice again we get
$\Omega^t \vdash_T A_D \langle \mathsf{sym}\ (\gamma_2), \overline{\mathsf{sym}\ (\gamma_i)} \rangle \circ \gamma_1 \circ \gamma_3 : A_D \langle u_2^t, \overline{u_{2i}^t} \rangle \sim u_4^t$ as required.

- **Case** (c-treq2):
  $\Gamma \mid \Theta \Vdash u_2 : D \langle \overline{u_{2i}}, A \mapsto \star \rangle \rightsquigarrow \left\langle e^t \blacktriangleright S_D \langle \gamma_2, \overline{\gamma_i} \rangle, A_D \langle u^t, \overline{u_{2i}} \rangle \right\rangle$ where
  $\Gamma \mid \Theta \Vdash u_1 : D \langle \overline{u_{1i}}, A \mapsto \star \rangle \rightsquigarrow \langle e^t, \gamma_1 \rangle$,
  $\Gamma \vdash u_2 : D \langle \overline{u_{2i}} \rangle$,
  $\Gamma \mid \Theta \Vdash u_1 \sim u_2 \rightsquigarrow \gamma$,
  $\overline{\Gamma \mid \Theta \Vdash u_{1i} \sim u_{2i} \rightsquigarrow \gamma_i}^{\,i}$,
  $\Gamma \mid \Theta \Vdash u_2 \rightsquigarrow u_2^t$ and
  $\overline{\Gamma \mid \Theta \Vdash u_{2i} \rightsquigarrow u_{2i}^t}$.

  Suppose $\Gamma \mid \Theta \Vdash u_2 : D \langle \overline{u_{2i}}, A \mapsto \star \rangle \rightsquigarrow \langle \tau_2^t, \omega_2 \rangle$.
  By lemma B.3, there are some $u_2^t, \overline{u_{2i}^t}$ such that
  $\Gamma \mid \Theta \vdash_{\mathrm{WF}} u_2 \rightsquigarrow u_2^t$,
  $\overline{\Gamma \mid \Theta \vdash_{\mathrm{WF}} u_{2i} \rightsquigarrow u_{2i}^t}$,
  $\tau_2^t = S_D \langle u_2^t, \overline{u_{2i}^t} \rangle$ and
  $\omega_2 = A_D \langle u_2^t, \overline{u_{2i}^t} \rangle \sim A_D \langle u_2^t, \overline{u_{2i}^t} \rangle$.

  By lemma B.14 $\Gamma \mid \Theta \vdash_{\mathrm{WF}} u_1 : D \langle \overline{u_{1i}}, A \mapsto \star \rangle \rightsquigarrow \langle \tau_1^t, \omega_1 \rangle$ for some τ_1^t, ω_1.

  By lemma B.3, there are some $u_1^t, \overline{u_{1i}^t}$ such that
  $\Gamma \mid \Theta \vdash_{\mathrm{WF}} u_1 \rightsquigarrow u_1^t$,
  $\overline{\Gamma \mid \Theta \vdash_{\mathrm{WF}} u_{1i} \rightsquigarrow u_{1i}^t}$ and $\tau_1^t = S_D \langle u_1^t, \overline{u_{1i}^t} \rangle$.

  Then, by IH we get:
  $\Omega^t \vdash_T \gamma_2 : u_1^t \sim u_2^t$ and
  $\overline{\Omega^t \vdash_T \gamma_i : u_{1i}^t \sim u_{2i}^t}$.

  Using rule (co-struct) we get $\Omega^t \vdash_T S_D \langle \gamma, \overline{\gamma_i} \rangle : S_D \langle u_1^t, \overline{u_{1i}^t} \rangle \sim S_D \langle u_2^t, \overline{u_{2i}^t} \rangle$
  and using (t-coerce) we get $\Omega^t \mid \Gamma^t \mid \emptyset \vdash_T e^t \blacktriangleright S_D \langle \gamma, \overline{\gamma_i} \rangle : S_D \langle u_2^t, \overline{u_{2i}^t} \rangle$.

  Then, using lemma A.13 we get
  $\Omega^t \vdash_T A_D \langle u_2^t, \overline{u_{2i}^t} \rangle : A_D \langle u_2^t, \overline{u_{2i}^t} \rangle \sim A_D \langle u_2^t, \overline{u_{2i}^t} \rangle$ as required.

- **Case** (c-astar): $\Gamma \mid \Theta \Vdash u : D\,\langle \overline{u_i}, A \mapsto \star\rangle \rightsquigarrow \langle e^t, A_D\,\langle u^t, \overline{u_i^t}\rangle\rangle$ where
  $\Gamma \mid \Theta \Vdash u : D\,\langle \overline{u_i}, A \mapsto u_A\rangle \rightsquigarrow \langle e^t, \gamma\rangle$,
  $\Gamma \mid \Theta \vdash_{\text{WF}} u \rightsquigarrow u^t$ and $\overline{\Gamma \mid \Theta \vdash_{\text{WF}} u_i \rightsquigarrow u_i^t}$.

  Suppose $\Gamma \mid \Theta \vdash_{\text{WF}} u : D\,\langle \overline{u_i}, A \mapsto \star\rangle \rightsquigarrow \langle \tau^t, \omega\rangle$.

  By lemma B.3, $\Gamma \vdash u : D\,\langle \overline{u_i}\rangle$,
  $\tau^t = S_D\,\langle u^t, \overline{u_i^t}\rangle$ and
  $\omega = A_D\,\langle u^t, \overline{u_i^t}\rangle \sim A_D\,\langle u^t, \overline{u_i^t}\rangle$.

  By lemma B.14, $\Gamma \mid \Theta \vdash_{\text{WF}} u : D\,\langle \overline{u_i}, A \mapsto u_A\rangle \rightsquigarrow \langle \tau^t, \omega\rangle$ for some τ_1^t, ω_1.

  By lemma B.3, $\tau_1^t = \tau^t$.

  Then, using IH, we get $\Omega^t \mid \Gamma^t \mid \emptyset \vdash_T e^t : \tau^t$.

  Using lemma A.13 we get $\Omega^t \vdash_T A_D\,\langle u^t, \overline{u_i^t}\rangle : A_D\,\langle u^t, \overline{u_i^t}\rangle \sim A_D\,\langle u^t, \overline{u_i^t}\rangle$ as required.

- **Case** (eq-sep): $\Gamma \mid \Theta \Vdash A_D\,\langle u, \overline{u_i}\rangle \sim u_A \rightsquigarrow \gamma$ where
  $\Gamma \mid \Theta \Vdash u : D\,\langle \overline{u_i}, A \mapsto u_A\rangle \rightsquigarrow \langle e^t, \gamma\rangle$.

  Suppose $\Gamma \mid \Theta \vdash_{\text{WF}} A_D\,\langle u, \overline{u_i}\rangle \rightsquigarrow u_1^t$ and $\Gamma \mid \Theta \vdash_{\text{WF}} u_A \rightsquigarrow u_A^t$.

  By lemma B.3, there are some u^t, $\overline{u_i^t}$ such that
  $\Gamma \mid \Theta \vdash u \rightsquigarrow u^t$,
  $\overline{\Gamma \mid \Theta \vdash u_i \rightsquigarrow u_i^t}$ and
  $u_1^t = A_D\,\langle u^t, \overline{u_i^t}\rangle$.

  By lemma B.14, there are some τ^t, ω such that
  $\Gamma \mid \Theta \vdash_{\text{WF}} u : D\,\langle \overline{u_i}, A \mapsto u_A\rangle \rightsquigarrow \langle \tau^t, \omega\rangle$.

  By lemma B.3, $\omega = A_D\,\langle u^t, \overline{u_i^t}\rangle \sim u_A^t$.

  Then, by IH, $\Omega \vdash_T \gamma : \omega$ as required.

- **Case** (eq-refl): $\Gamma \mid \Theta \Vdash u \sim u \rightsquigarrow u^t$ where
  $\Gamma \mid \Theta \vdash_{\text{WF}} u \rightsquigarrow u^t$.

  By lemma A.7, $\Omega^t \vdash_T u^t : u^t \sim u^t$ as required.

- **Case** (eq-trans): $\Gamma \mid \Theta \Vdash u_1 \sim u_2 \rightsquigarrow \gamma_1 \circ \gamma_2$ where
  $\Gamma \mid \Theta \Vdash u_1 \sim u_3 \rightsquigarrow \gamma_1$ and
  $\Gamma \mid \Theta \Vdash u_3 \sim u_2 \rightsquigarrow \gamma_2$.

  Suppose $\Gamma \mid \Theta \vdash_{\text{WF}} u_1 \rightsquigarrow u_1^t$ and $\Gamma \mid \Theta \vdash_{\text{WF}} u_2 \rightsquigarrow u_2^t$.

  From lemma B.14 we get $\Gamma \mid \Theta \vdash_{\text{WF}} u_3 \rightsquigarrow u_3^t$ for some u_3^t.

  From IH we get $\Omega^t \vdash_T \gamma_1 : u_1^t \sim u_3^t$ and $\Omega^t \vdash_T \gamma_2 : u_3^t \sim u_2^t$.

  We can then use rule (co-trans) to get $\Omega^t \vdash_T \gamma_1 \circ \gamma_2 : u_1^t \sim u_2^t$ as required.

- **Case** (eq-sym): $\Gamma \mid \Theta \Vdash u_1 \sim u_2 \rightsquigarrow \mathrm{sym}\ (\gamma)$ where
  $\Gamma \mid \Theta \Vdash u_2 \sim u_1 \rightsquigarrow \gamma$.

  Suppose $\Gamma \mid \Theta \vdash_{\mathrm{WF}} u_1 \rightsquigarrow u_1^t$ and $\Gamma \mid \Theta \vdash_{\mathrm{WF}} u_2 \rightsquigarrow u_2^t$ for some u_1^t, u_2^t.

  By IH $\Omega^t \vdash_T \gamma : u_2^t \sim u_1^t$.

  We can then use (co-sym) to prove $\Omega^t \vdash_T \mathrm{sym}\ \gamma : u_1^t \sim u_2^t$ as required.

- **Case** (eq-struct): $\Gamma \mid \Theta \Vdash S\ \langle \overline{u_{1i}} \rangle \sim S\ \langle \overline{u_{2i}} \rangle \rightsquigarrow S\ \langle \overline{\gamma_i} \rangle$ where
  $\overline{\Gamma \mid \Theta \Vdash u_{1i} \sim u_{2i} \rightsquigarrow \gamma_i}^{\,i}$.

  Suppose $\Gamma \mid \Theta \vdash_{\mathrm{WF}} S\ \langle \overline{u_{1i}} \rangle \rightsquigarrow u_1^t$ and $\Gamma \mid \Theta \vdash_{\mathrm{WF}} S\ \langle \overline{u_{2i}} \rangle \rightsquigarrow u_2^t$.

  From lemma B.3 we get
  $\overline{\Gamma \mid \Theta \vdash u_{1i} \rightsquigarrow u_{1i}^t}$,
  $\overline{\Gamma \mid \Theta \vdash u_{2i} \rightsquigarrow u_{2i}^t}$,
  $u_1^t = S\ \langle \overline{u_{1i}^t} \rangle$, and
  $u_2^t = S\ \langle \overline{u_{2i}^t} \rangle$
  for some $\overline{u_{1i}^t}, \overline{u_{2i}^t}$.

  By IH, $\overline{\Omega^t \vdash_T \gamma_i : u_{1i}^t \sim u_{2i}^t}$.

  We can then use (co-struct) to get $\Omega^t \vdash_T S\ \langle \overline{\gamma_i} \rangle : S\ \langle \overline{u_{1i}^t} \rangle \sim S\ \langle \overline{u_{2i}^t} \rangle$ as required.

- **Case** (eq-ref): $\Gamma \mid \Theta \Vdash \&u_1 \sim \&u_2 \rightsquigarrow \&\gamma$ where $\Gamma \mid \Theta \Vdash u_1 \sim u_2 \rightsquigarrow \gamma$.

  Suppose $\Gamma \mid \Theta \vdash_{\mathrm{WF}} \&u_1 \rightsquigarrow u_1'^t$ and $\Gamma \mid \Theta \vdash_{\mathrm{WF}} \&u_2 \rightsquigarrow u_2'^t$.

  From lemma B.3 we get
  $\Gamma \mid \Theta \vdash_{\mathrm{WF}} u_1 \rightsquigarrow u_1^t$,
  $\Gamma \mid \Theta \vdash_{\mathrm{WF}} u_2 \rightsquigarrow u_2^t$,
  $u_1'^t = \&u_1^t$ and $u_2'^t = \&u_2^t$ for some u_1^t and u_2^t.

  Using the IH we get $\Omega^t \vdash_T \gamma : u_1^t \sim u_2^t$.

  Then, using (co-ref) we get $\Omega^t \vdash_T \&\gamma : \&u_1^t \sim \&u_2^t$ as required.

- **Case** (eq-obj): $\Gamma \mid \Theta \Vdash \exists T : D\ \langle \overline{u_{1i}}, \overline{A_{Dj}\ \langle T, \overline{u_{1s_j}} \rangle \sim u_{1j}} \rangle \sim \exists T : D\ \langle \overline{u_{2i}}, \overline{A_{Dj}\ \langle T, \overline{u_{2s_j}} \rangle \sim u_{2j}} \rangle$
  $\rightsquigarrow (\exists T, \&T, S_D\ \langle T, \overline{\gamma_i} \rangle, \overline{A_{Dj}\ \langle T, \overline{\gamma_{s_j}} \rangle \sim \gamma_j})$ where
  $\overline{\Gamma \mid \Theta \Vdash u_{1i} \sim u_{2i} \rightsquigarrow \gamma_i}$,
  $\overline{\overline{\Gamma, T \mid \Theta, T : D\ \langle \overline{u_{1i}}, A \mapsto \_ \rangle \Vdash u_{1s_j} \sim u_{2s_j} \rightsquigarrow \gamma_{s_j}}}$ and
  $\overline{\Gamma \mid \Theta \Vdash u_{1j} \sim u_{2j} \rightsquigarrow \gamma_j}$.

  Suppose $\Gamma \mid \Theta \vdash_{\mathrm{WF}} \exists T : D\ \langle \overline{u_{1i}}, \overline{A_{Dj}\ \langle T, \overline{u_{1s_j}} \rangle \sim u_{1j}} \rangle \rightsquigarrow u_1^t$ and
  $\Gamma \mid \Theta \vdash_{\mathrm{WF}} \exists T : D\ \langle \overline{u_{2i}}, \overline{A_{Dj}\ \langle T, \overline{u_{2s_j}} \rangle \sim u_{2j}} \rangle \rightsquigarrow u_2^t$.

Then from lemma B.3 there are some $\overline{u^t_{1i}}, \overline{\overline{u^t_{1sj}}}, \overline{u^t_{1j}}, \overline{u^t_{2i}}, \overline{\overline{u^t_{2sj}}}, \overline{u^t_{2j}}$ such that
$\overline{\Gamma \mid \Theta \vdash_{\mathrm{WF}} u_{1i} \rightsquigarrow u^t_{1i}}$, $\overline{\overline{\Gamma, T \mid \Theta, T : D\ \langle \overline{u_{1i}}, A \mapsto \_ \rangle \vdash_{\mathrm{WF}} u_{1sj} \rightsquigarrow u^t_{1i}}}$, $\overline{\Gamma \mid \Theta \vdash_{\mathrm{WF}} u_{1j} \rightsquigarrow u^t_{1j}}$,
$\overline{\Gamma \mid \Theta \vdash_{\mathrm{WF}} u_{2i} \rightsquigarrow u^t_{2i}}$, $\overline{\overline{\Gamma, T \mid \Theta, T : D\ \langle \overline{u_{2i}}, A \mapsto \_ \rangle \vdash_{\mathrm{WF}} u_{2sj} \rightsquigarrow u^t_{2i}}}$, $\overline{\Gamma \mid \Theta \vdash_{\mathrm{WF}} u_{2j} \rightsquigarrow u^t_{2j}}$,
$u^t_1 = (\exists T, \& T, S_D\ \langle T, \overline{u^t_{1i}} \rangle, \overline{A_{Dj}\ \langle T, \overline{u^t_{1sj}} \rangle \sim u^t_{1j}})$ and
$u^t_2 = (\exists T, \& T, S_D\ \langle T, \overline{u^t_{2i}} \rangle, \overline{A_{Dj}\ \langle T, \overline{u^t_{2sj}} \rangle \sim u^t_{2j}})$.
Then by IH $\overline{\Omega^t \vdash_T \gamma_i : u^t_{1i} \sim u^t_{2i}}$,
$\overline{\overline{\Omega^t \vdash_T \gamma_{sj} : u^t_{1sj} \sim u^t_{2sj}}}$ and
$\overline{\Omega^t \vdash_T \gamma_j : u^t_{1j} \sim u^t_{2j}}$.
By (co-tvar) $\Omega^t \vdash_T T : T \sim T$.
By (co-ref) $\Omega^t \vdash_T \& T : \& T \sim \& T$.
By (co-struct) $\overline{\Omega^t \vdash_T S_D\ \langle T, \overline{\gamma_i} \rangle : S_D\ \langle T, \overline{u^t_{1i}} \rangle \sim S_D\ \langle T, \overline{u^t_{2i}} \rangle}$.
By (co-atype) $\overline{\Omega^t \vdash_T A_{Dj}\ \langle T, \overline{\gamma_{sj}} \rangle : A_{Dj}\ \langle T, \overline{u^t_{1sj}} \rangle \sim A_{Dj}\ \langle T, \overline{u^t_{2sj}} \rangle}$.
Then we can apply (co-obj) to get
$\Omega^t \vdash_T (\exists T, \& T, S_D\ \langle T, \overline{\gamma_i} \rangle, \overline{A_{Dj}\ \langle T, \overline{\gamma_{sj}} \rangle \sim \gamma_j}) :$
$(\exists T, \& T, S_D\ \langle T, \overline{u^t_{1i}} \rangle, \overline{A_{Dj}\ \langle T, \overline{u^t_{1sj}} \rangle \sim u^t_{1j}}) \sim (\exists T, \& T, S_D\ \langle T, \overline{u^t_{2i}} \rangle, \overline{A_{Dj}\ \langle T, \overline{u^t_{2sj}} \rangle \sim u^t_{2j}})$
as required.

- **Case** (eq-fun): $\Gamma \mid \Theta \Vdash \mathsf{fn}(\overline{\tau_{1i}}) \to \tau_1 \sim \mathsf{fn}(\overline{\tau_{2i}}) \to \tau_2 \rightsquigarrow \mathsf{fn}(\overline{\gamma_i}) \to \gamma$ where
$\overline{\Gamma \mid \Theta \Vdash \tau_{1i} \sim \tau_{2i} \rightsquigarrow \gamma_i}^{\,i}$ and
$\Gamma \mid \Theta \Vdash \tau_1 \sim \tau_2 \rightsquigarrow \gamma$.

  Suppose $\Gamma \mid \Theta \vdash_{\mathrm{WF}} \mathsf{fn}(\overline{\tau_{1i}}) \to \tau_1 \rightsquigarrow u^t_1$ and $\Gamma \mid \Theta \vdash_{\mathrm{WF}} \mathsf{fn}(\overline{\tau_{2i}}) \to \tau_2 \rightsquigarrow u^t_2$.

  From lemma B.3 we get
  $\overline{\Gamma \mid \Theta \vdash_{\mathrm{WF}} \tau_{1i} \rightsquigarrow \tau^t_{1i}}$,
  $\overline{\Gamma \mid \Theta \vdash_{\mathrm{WF}} \tau_{2i} \rightsquigarrow \tau^t_{2i}}$,
  $\Gamma \mid \Theta \vdash_{\mathrm{WF}} \tau_1 \rightsquigarrow \tau^t_1$,
  $\Gamma \mid \Theta \vdash_{\mathrm{WF}} \tau_2 \rightsquigarrow \tau^t_2$,
  $u^t_1 = \mathsf{fn}(\overline{\tau^t_{1i}}) \to \tau^t_1$, and
  $u^t_2 = \mathsf{fn}(\overline{\tau^t_{2i}}) \to \tau^t_2$
  for some $\overline{\tau^t_{1i}}, \overline{\tau^t_2}, \tau^t_1, \tau^t_2$.

  Then from IH we get
  $\overline{\Omega^t \vdash_T \gamma_i : \tau^t_{1i} \sim \tau^t_{2i}}$ and
  $\Omega^t \vdash_T \gamma : \tau^t_1 \sim \tau^t_2$.

  We can then use rule (co-fun) to get
  $\Omega^t \vdash_T \mathsf{fn}(\overline{\gamma_i}) \to \gamma : \mathsf{fn}(\overline{\tau^t_{1i}}) \to \tau^t_1 \sim \mathsf{fn}(\overline{\tau^t_{2i}}) \to \tau^t_2$ as required.

- **Case** (eq-atype): $\Gamma \mid \Theta \Vdash A_D\,\langle u_1, \overline{u_{1i}}\rangle \sim A_D\,\langle u_2, \overline{u_{2i}}\rangle \rightsquigarrow A_D\,\langle \gamma, \overline{\gamma_i}\rangle$ where
  $\Gamma \mid \Theta \Vdash u_1 \sim u_2 \rightsquigarrow \gamma$ and
  $\overline{\Gamma \mid \Theta \Vdash u_{1i} \sim u_{2i} \rightsquigarrow \gamma_i}$.

  Suppose $\Gamma \mid \Theta \vdash_{\text{WF}} A_D\,\langle u_1, \overline{u_{1i}}\rangle \rightsquigarrow u_1'^t$ and $\Gamma \mid \Theta \vdash_{\text{WF}} A_D\,\langle u_2, \overline{u_{2i}}\rangle \rightsquigarrow u_2'^t$.

  From lemma B.3 we get
  $\Gamma \mid \Theta \vdash u_1 \rightsquigarrow u_1^t$,
  $\Gamma \mid \Theta \vdash u_2 \rightsquigarrow u_2^t$,
  $\overline{\Gamma \mid \Theta \vdash u_{1i} \rightsquigarrow u_{1i}^t}$,
  $\overline{\Gamma \mid \Theta \vdash u_{2i} \rightsquigarrow u_{2i}^t}$,
  $u_1'^t = A_D\,\langle u_1^t, \overline{u_{1i}^t}\rangle$ and $u_2'^t = A_D\,\langle u_2^t, \overline{u_{2i}^t}\rangle$
  for some $u_1^t, u_2^t, \overline{u_{1i}^t}, \overline{u_{2i}^t}$.

  We can then use IH to get
  $\Omega^t \vdash_T \gamma : u_1^t \sim u_2^t$ and
  $\overline{\Omega^t \vdash_T \gamma_i : u_{1i}^t \sim u_{2i}^t}$
  and we can use rule (co-atype) to get
  $\Omega^t \vdash_T A_D\,\langle \gamma, \overline{\gamma_i}\rangle : A_D\,\langle u_1^t, \overline{u_{1i}^t}\rangle \sim A_D\,\langle u_2^t, \overline{u_{2i}^t}\rangle$ as required.

□

Lemma B.19. *If $X \notin \Gamma$ and $\Gamma \mid \Theta \vdash_{\text{WF}} u \rightsquigarrow u^t$ then $X \notin FV(u, u^t)$.*

Proof. Straightforward induction on derivations $\Gamma \mid \Theta \vdash_{\text{WF}} u$. □

Theorem B.1 (Translation of terms preserves well-typedness)**.** *If:*

- $\Gamma \mid \Theta \vdash e : \tau \rightsquigarrow e^t$ *and*
- $\Gamma \mid \Theta \vdash_{\text{WF}} \tau \rightsquigarrow \tau^t$ *and*
- $\langle \Gamma, \Theta\rangle \rightsquigarrow \langle \Omega^t, \Gamma^t\rangle$,

then $\Omega^t \mid \Gamma^t \mid \emptyset \vdash_T e^t : \tau^t$.

Proof. By induction on derivations $\Gamma \mid \Theta \vdash e : \tau \rightsquigarrow e^t$:

- **Case** (sub): $\Gamma \mid \Theta \vdash e : \tau_2 \rightsquigarrow e^t \blacktriangleright \gamma$ where
  $\Gamma \mid \Theta \vdash e : \tau_1 \rightsquigarrow e^t$ and
  $\Gamma \mid \Theta \Vdash \tau_1 \sim \tau_2 \rightsquigarrow \gamma$.

  Suppose $\Gamma \mid \Theta \vdash_{\text{WF}} \tau_2 \rightsquigarrow \tau_2^t$.

  From lemma B.17 there is some τ_1^t such that $\Gamma \mid \Theta \vdash_{\text{WF}} \tau_1 \rightsquigarrow \tau_1^t$.

From IH we have $\Omega^t \mid \Gamma^t \mid \emptyset \vdash_T e^t : \tau_1^t$.

From lemma B.14 and lemma B.16 we get
$\Gamma \mid \Theta \vdash_{\text{WF}} \tau_1 \rightsquigarrow \tau_1^t$ and $\Gamma \mid \Theta \vdash_{\text{WF}} \tau_2 \rightsquigarrow \tau_2^t$
for some s-types τ_1^t, τ_2^t.

Then, we can use lemma B.18 to get $\Omega^t \vdash_T \gamma : \tau_1^t \sim \tau_2^t$.

We can then apply rule (t-coerce) to conclude
$\Omega^t \mid \Gamma^t \mid \emptyset \vdash_T e^t \blacktriangleright \gamma : \tau_2^t$ as required.

- **Case** (as): $\Gamma \mid \Theta \vdash e$ as $\tau : \tau \rightsquigarrow e^t$ where
$\Gamma \mid \Theta \vdash e : \tau \rightsquigarrow e^t$.

  Suppose $\Gamma \mid \Theta \vdash \tau \rightsquigarrow \tau^t$.

  From IH we get $\Omega^t \mid \Gamma^t \mid \emptyset \vdash_T e^t \vdash \tau^t$ as required.

- **Case** (unit): $\Gamma \mid \Theta \vdash () : () \rightsquigarrow ()$

  By (wf-unit) $\Gamma \mid \Theta \vdash () \rightsquigarrow ()$ and
  by (t-unit) $\Omega^t \mid \Gamma^t \mid \emptyset \vdash_T () : ()$ as required.

- **Case** (let-un): $\Gamma \mid \Theta \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2 \rightsquigarrow \mathsf{let}\ x : \tau_1^t = e_1^t\ \mathsf{in}\ e_2^t$ where
$\Gamma \mid \Theta \vdash e_1 : \tau_1 \rightsquigarrow e_1^t$,
$\Gamma, x : \tau_1 \mid \Theta \vdash e_2 : \tau_2 \rightsquigarrow e_2^t$ and
$\Gamma \mid \Theta \vdash_{\text{WF}} \tau_1 \rightsquigarrow \tau_1^t$.

  Suppose $\Gamma \mid \Theta \vdash \tau_2 \rightsquigarrow \tau_2^t$.

  By lemma B.10 $\Gamma, x : \tau_1 \mid \Theta \vdash \tau_2 \rightsquigarrow \tau_2^t$.

  $\langle (\Gamma, x : \tau_1), \Theta \rangle \rightsquigarrow \langle \Omega^t, (\Gamma^t, x : \tau_1^t) \rangle$ by definition of environment translation.

  Then by IH $\Omega^t \mid \Gamma^t \mid \emptyset \vdash_T e_1^t : \tau_1^t$ and
  $\Omega^t \mid \Gamma^t, x : \tau_1^t \mid \emptyset \vdash e_2^t : \tau_2^t$.

  We can then apply (t-let) to get $\Omega^t \mid \Gamma^t \mid \emptyset \vdash_T \mathsf{let}\ x : \tau_1^t = e_1^t\ \mathsf{in}\ e_2^t : \tau_2^t$ as required.

- **Case** (seq): $\Gamma \mid \Theta \vdash e_1; e_2 : \tau_2 \rightsquigarrow e_1^t; e_2^t$ where
$\Gamma \mid \Theta \vdash e_1 : \tau_1 \rightsquigarrow e_1^t$ and
$\Gamma \mid \Theta \vdash e_2 : \tau_2 \rightsquigarrow e_2^t$.

  Suppose $\Gamma \mid \Theta \vdash \tau_2 \rightsquigarrow \tau_2^t$.

  By lemma B.17 $\Gamma \mid \Theta \vdash \tau_1 \rightsquigarrow \tau_1^t$ for some τ_1^t.

  Then, by IH $\Omega^t \mid \Gamma^t \mid \emptyset \vdash_T e_1^t : \tau_1^t$ and $\Omega^t \mid \Gamma^t \mid \emptyset \vdash_T e_2^t : \tau_2^t$.

  We can then apply rule (t-seq) to get $\Omega^t \mid \Gamma^t \mid \emptyset \vdash_T e_1^t; e_2^t : \tau_2^t$ as required.

- **Case** (var): $\Gamma \mid \Theta \vdash x : [\overline{u_i/X_i}]\tau \rightsquigarrow x[\overline{u_i^t}][\![\overline{\gamma_j}]\!](\overline{e_j^t})$ where
  $(x : \forall \overline{X_i}.\overline{\pi_j} \Rightarrow \tau) \in \Gamma$,
  $\overline{\Gamma \mid \Theta \vdash_{\text{WF}} u_i \rightsquigarrow u_i^t}$ and
  $\overline{\Gamma \mid \Theta \Vdash [\overline{u_i/X_i}]\pi_j \rightsquigarrow \langle e_j^t, \gamma_j \rangle}^{j}$.

  Suppose $\Gamma \mid \Theta \vdash_{\text{WF}} [\overline{u_i/X_i}]\tau \rightsquigarrow \tau_1^t$.

  By lemma B.1 there is some type scheme σ^t where $(x : \sigma^t) \in \Gamma^t$ and
  $\Gamma \mid \Theta \vdash_{\text{WF}} \forall \overline{X_i}.\overline{\pi_j} \Rightarrow \tau \rightsquigarrow \sigma^t$.

  From lemma B.3 we know that $\sigma^t = \forall \overline{X_i}.\forall \overline{\omega_j}.\mathsf{fn}(\overline{\tau_j^t}) \to \tau^t$
  where $\overline{\Gamma, \overline{X_i} \mid \Theta \vdash_{\text{WF}} \pi_j \rightsquigarrow \langle \tau_j^t, \omega_j \rangle}^{j}$
  and $\Gamma, \overline{X_i} \mid \Theta, \overline{\pi_j \rightsquigarrow \langle x_j, c_j \rangle} \vdash_{\text{WF}} \tau \rightsquigarrow \tau^t$
  for some $\overline{\tau_j^t}, \overline{\omega_j}, \overline{c_j}, \tau^t$.

  From lemma B.7 we get $\overline{\Gamma \mid \Theta \vdash_{\text{WF}} [\overline{u_i/X_i}]\pi_j \rightsquigarrow \langle [\overline{u_i^t/X_i}]\tau_j^t, [\overline{u_i^t/X_i}]\omega_j \rangle}^{j}$
  and $\Gamma \mid \Theta, \overline{[\overline{u_i/X_i}]\pi_j \rightsquigarrow \langle x_j, c_j \rangle} \vdash_{\text{WF}} [\overline{u_i/X_i}]\tau \rightsquigarrow [\overline{u_i^t/X_i}]\tau^t$.
  So $\tau_1^t = [\overline{u_i^t/X_i}]\tau^t$.

  From lemma B.18 we get
  $\overline{\Omega^t \mid \Gamma^t \mid \emptyset \vdash_T e_j^t : [\overline{u_i^t/X_i}]\tau_j^t}^{j}$ and
  $\overline{\Omega^t \vdash_T \gamma_j^t : [\overline{u_i^t/X_i}]\omega_j}^{j}$.

  We can then apply rule(t-var) to get $\Omega^t \mid \Gamma^t \mid \emptyset \vdash_T x : \forall \overline{X_i}.\forall \overline{\omega_j}.\mathsf{fn}(\overline{\tau_j^t}) \to \tau^t$,
  rule (t-tapp) to get $\Omega^t \mid \Gamma^t \mid \emptyset \vdash_T x[\overline{u_i^t}] : \forall \overline{[\overline{u_i^t/X_i}]\omega_j}.\mathsf{fn}(\overline{[\overline{u_i^t/X_i}]\tau_j^t}) \to [\overline{u_i^t/X_i}]\tau^t$,
  rule (t-capp) to get $\Omega^t \mid \Gamma^t \mid \emptyset \vdash_T x[\overline{u_i^t}][\![\overline{\gamma_j}]\!] : \mathsf{fn}(\overline{[\overline{u_i^t/X_i}]\tau_j^t}) \to [\overline{u_i^t/X_i}]\tau^t$ and
  rule (t-fapp) to get $\Omega^t \mid \Gamma^t \mid \emptyset \vdash_T x[\overline{u_i^t}][\![\overline{\gamma_j}]\!](\overline{e_j^t}) : [\overline{u_i^t/X_i}]\tau^t$ as required.

- **Case** (app): $\Gamma \mid \Theta \vdash e(\overline{e_i}) : \tau \rightsquigarrow e^t(\overline{e_i^t})$ where
  $\Gamma \mid \Theta \vdash e : \mathsf{fn}(\overline{\tau_i}) \to \tau \rightsquigarrow e^t$ and
  $\overline{\Gamma \mid \Theta \vdash e_i : \tau_i \rightsquigarrow e_i^t}$.

  Suppose $\Gamma \mid \Theta \vdash_{\text{WF}} \tau \rightsquigarrow \tau^t$.

  From lemma B.17 there are some $\overline{\tau_i^t}$ where $\overline{\Gamma \mid \Theta \vdash_{\text{WF}} \tau_i \rightsquigarrow \tau_i^t}$.

  We can use rule (wf-fun) to get $\Gamma \mid \Theta \vdash \mathsf{fn}(\overline{\tau_i}) \to \tau \rightsquigarrow \mathsf{fn}(\overline{\tau_i^t}) \to \tau^t$.

  Then from IH we get $\Omega^t \mid \Gamma^t \mid \emptyset \vdash_T e^t : \mathsf{fn}(\overline{\tau_i^t}) \to \tau^t$ and $\overline{\Omega^t \mid \Gamma^t \mid \emptyset \vdash_T e_i^t : \tau_i^t}$.

  We can then apply rule (t-fapp) to get $\Omega^t \mid \Gamma^t \mid \emptyset \vdash_T e^t(\overline{e_i^t}) : \tau^t$ as required.

- **Case** (ref): $\Gamma \mid \Theta \vdash \&lv : \&\tau \rightsquigarrow \&lv^t$ where
  $\Gamma \mid \Theta \vdash lv : \tau \rightsquigarrow lv^t$.

Suppose $\Gamma \mid \Theta \vdash_{\text{WF}} \&\tau \rightsquigarrow \tau_1^t$.

By lemma B.17 there is some τ^t such that $\Gamma \mid \Theta \vdash_{\text{WF}} \tau \rightsquigarrow \tau^t$.

Then by (wf-ref) $\Gamma \mid \Theta \vdash_{\text{WF}} \&\tau \rightsquigarrow \&\tau^t$
so $\tau_1^t = \&\tau^t$.

By IH $\Omega^t \mid \Gamma^t \mid \emptyset \vdash_T l v^t : \tau^t$.

We can then apply rule (t-ref) to get $\Omega^t \mid \Gamma^t \mid \emptyset \vdash_T \&l v^t : \&\tau^t$ as required.

- **Case** (deref): $\Gamma \mid \Theta \vdash *e : \tau \rightsquigarrow *e^t$ where
$\Gamma \mid \Theta \vdash e : \&\tau \rightsquigarrow e^t$.

  Suppose $\Gamma \mid \Theta \vdash_{\text{WF}} \tau \rightsquigarrow \tau^t$.

  We can use rule (wf-ref) to get $\Gamma \mid \Theta \vdash_{\text{WF}} \&\tau \rightsquigarrow \&\tau^t$.

  Then by IH $\Omega^t \mid \Gamma^t \mid \emptyset \vdash_T e^t : \&\tau^t$.

  We can then use rule (t-deref) to get $\Omega^t \mid \Gamma^t \mid \emptyset \vdash_T *e^t \rightsquigarrow \tau^t$ as required.

- **Case** (asgn): $\Gamma \mid \Theta \vdash l v := e : () \rightsquigarrow l v^t := e^t$ where
$\Gamma \mid \Theta \vdash l v : \tau \rightsquigarrow l v^t$ and
$\Gamma \mid \Theta \vdash e : \tau \rightsquigarrow e^t$.

  Suppose $\Gamma \mid \Theta \vdash_{\text{WF}} () \rightsquigarrow \tau_1^t$.

  From lemma B.3 we know that $\tau_1^t = ()$.

  From lemma B.17 we know that there is some τ^t such that $\Gamma \mid \Theta \vdash_{\text{WF}} \tau \rightsquigarrow \tau^t$.

  We can then use IH to get $\Omega^t \mid \Gamma^t \mid \emptyset \vdash_T l v^t : \tau^t$ and $\Omega^t \mid \Gamma^t \mid \emptyset \vdash_T e^t : \tau^t$.

  We can then use rule (t-upd) to get $\Omega^t \mid \Gamma^t \mid \emptyset \vdash_T l v^t := e^t : ()$ as required.

- **Case** (new-struct): $\Gamma \mid \Theta \vdash S\{\overline{x_i : e_i}\} : S\,\langle \overline{u_k} \rangle \rightsquigarrow S\,\langle \overline{u_k^t} \rangle \{\overline{x_i : e_i^t}\}$ where
$S\,\langle \overline{X_k} \rangle \{\overline{x_i : \tau_i}\} \in \Gamma$,
$\overline{\Gamma \mid \Theta \vdash e_i : [\overline{u_k/X_k}]\tau_i \rightsquigarrow e_i^t}$ and
$\overline{\Gamma \mid \Theta \vdash_{\text{WF}} u_k \rightsquigarrow u_k^t}$.

  Suppose $\Gamma \mid \Theta \vdash_{\text{WF}} S\,\langle \overline{u_k} \rangle \rightsquigarrow \tau^t$.

  From lemmas B.1 and B.2 we know that $S\,\langle \overline{X_k} \rangle \{\overline{x_i : \tau_i^t}\} \in \Omega^t$
  for some $\overline{\tau_i^t}$ where $\overline{\Gamma, \overline{X_k} \mid \Theta \vdash_{\text{WF}} \tau_i \rightsquigarrow \tau_i^t}$.

  From lemma B.3 we get $\tau^t = S\,\langle \overline{u_k^t} \rangle$.

  By lemma B.7, $\overline{\Gamma \mid \Theta \vdash_{\text{WF}} [\overline{u_k/X_k}]\tau_i \rightsquigarrow [\overline{u_k^t/X_k}]\tau_i^t}$.

  We can use the IH to get $\overline{\Omega^t \mid \Gamma^t \mid \emptyset \vdash_T e_i^t : [\overline{u_k^t/X_k}]\tau_i^t}$.

We can then apply rule (t-newstruct) to get $\Omega^t \mid \Gamma^t \mid \emptyset \vdash_T S\,\langle\overline{u_k^t}\rangle\,\{\overline{x_i : e_i^t}\} : S\,\langle\overline{u_k^t}\rangle$ as required.

- **Case** (proj): $\Gamma \mid \Theta \vdash e.x : [\overline{u_j/X_j}]\tau \rightsquigarrow e^t.x$ where
  $\Gamma \mid \Theta \vdash e : S\,\langle\overline{u_j}\rangle \rightsquigarrow e^t$ and
  $\Gamma(S) = \mathsf{struct}\ S\,\langle\overline{X_j}\rangle\,\{\overline{x_m : \tau_m}, x : \tau, \overline{x_n : \tau_n}\}$.

  Suppose $\Gamma \mid \Theta \vdash_{\mathrm{WF}} [\overline{u_j/X_j}]\tau \rightsquigarrow \tau_1^t$.

  From lemmas B.1 and B.2 we know that $S\,\langle\overline{X_k}\rangle\,\{\overline{x_m : \tau_m^t}, x : \tau^t, \overline{x_n : \tau_n^t}\} \in \Omega^t$
  for some $\overline{\tau_m^t}, \tau^t, \overline{\tau_n^t}$ where $\overline{\Gamma, \overline{X_k} \mid \Theta \vdash_{\mathrm{WF}} \tau \rightsquigarrow \tau^t}$.

  From lemmas B.17 and B.3 we get $\Gamma \mid \Theta \vdash_{\mathrm{WF}} S\,\langle\overline{u_j}\rangle \rightsquigarrow S\,\langle\overline{u_j^t}\rangle$
  where $\overline{\Gamma \mid \Theta \vdash_{\mathrm{WF}} u_j \rightsquigarrow u_j^t}$ for some $\overline{u_j^t}$.

  By IH $\Omega^t \mid \Gamma^t \mid \emptyset \vdash_T e^t : S\,\langle\overline{u_j^t}\rangle$.

  By lemma B.7, $\Gamma \mid \Theta \vdash_{\mathrm{WF}} [\overline{u_j/X_j}]\tau \rightsquigarrow [\overline{u_j^t/X_j}]\tau^t$ so $\tau_1^t = [\overline{u_j^t/X_j}]\tau^t$.

  Then we can use rule (t-proj) to get $\Omega^t \mid \Gamma^t \mid \emptyset \vdash_T e^t.x : [\overline{u_j^t/X_j}]\tau^t$ as required.

- **Case** (obj-cast): $\Gamma \mid \Theta \vdash e : \&(\exists T : D\,\langle\overline{u_i}, \overline{u_{1j} \sim u_{2j}}\rangle) \rightsquigarrow \mathsf{pack}\ (\tau_1^t, e^t, e_D^t, \overline{\gamma_j})\ \mathsf{as}\ \tau_2^t$ where
  $\Gamma \mid \Theta \vdash e : \&\tau \rightsquigarrow e^t$,
  $\Gamma \mid \Theta \vdash_{\mathrm{WF}} \tau \rightsquigarrow \tau_1^t$,
  $\Gamma \mid \Theta \Vdash \tau : D\,\langle\overline{u_i}\rangle \rightsquigarrow e_D^t$,
  $\overline{\Gamma \mid \Theta \Vdash [\tau/T]u_{1j} \sim u_{2j} \rightsquigarrow \gamma_j}$,
  $\Gamma \mid \Theta \Vdash (\exists T : D\,\langle\overline{u_i}, \overline{u_{1j} \sim u_{2j}}\rangle) : D\,\langle\overline{u_i}\rangle$ and
  $\Gamma \mid \Theta \vdash_{\mathrm{WF}} \&(\exists T : D\,\langle\overline{u_i}, \overline{u_{1j} \sim u_{2j}}\rangle) \rightsquigarrow \tau_2^t$.

  From lemma B.3 we know that there are some $\overline{A_{Dj}}, \overline{\overline{u_{sj}}}, \overline{u_j}, \overline{u_i^t}, \overline{\overline{u_{sj}^t}}, \overline{u_j^t}$ such that
  $\overline{u_{1j} = A_{Dj}\,\langle T, \overline{u_{sj}}\rangle}$,
  $\overline{\Gamma \mid \Theta \vdash_{\mathrm{WF}} u_i \rightsquigarrow u_i^t}$,
  $\overline{\overline{\Gamma, T \mid \Theta, T : D\,\langle\overline{u_i}, A \mapsto \_\rangle \vdash_{\mathrm{WF}} u_{sj} \rightsquigarrow u_{sj}^t}}$,
  $\overline{\Gamma \mid \Theta \vdash_{\mathrm{WF}} u_{2j} \rightsquigarrow u_{2j}^t}$ and
  $\tau_2^t = (\exists T, \&T, S_D\,\langle T, \overline{u_i^t}\rangle, \overline{A_{Dj}\,\langle T, \overline{u_{sj}^t}\rangle \sim u_j^t})$.

  By rule (wf-ref) $\Gamma \mid \Theta \vdash_{\mathrm{WF}} \&\tau \rightsquigarrow \&\tau_1^t$
  and by IH $\Omega^t \mid \Gamma^t \mid \emptyset \vdash_T e^t : \&\tau_1^t$.

  By lemmas B.14 and B.3, $\Gamma \mid \Theta \vdash_{\mathrm{WF}} \tau : D\,\langle\overline{u_i}\rangle \rightsquigarrow S_D\,\langle\tau_1^t, \overline{u_i^t}\rangle$
  and by lemma B.18, $\Omega^t \mid \Gamma^t \mid \emptyset \vdash_T e_D^t : S_D\,\langle\tau_1^t, \overline{u_i^t}\rangle$.

  By (wf-atype), $\Gamma, T \mid \Theta, T : D\,\langle\overline{u_i}, A \mapsto \_\rangle \vdash_{\mathrm{WF}} A_{Dj}\,\langle T, \overline{u_{sj}}\rangle \rightsquigarrow A_{Dj}\,\langle T, \overline{u_{sj}^t}\rangle$.

  Using lemma B.7, we get
  $\Gamma \mid \Theta \vdash_{\mathrm{WF}} [\tau/T]A_{Dj}\,\langle T, \overline{u_{sj}}\rangle \rightsquigarrow A_{Dj}\,\langle \tau_1^t, \overline{[\tau_1^t/T]u_{sj}^t}\rangle$.

As $T \notin \Gamma$, lemma B.19 gives us $T \notin FV(\overline{u_i}, \overline{u_{2j}}, \overline{u_i^t}, \overline{u_{2_j}^t})$.
Thus $[\tau_1^t/T]S_D\,\langle T, u_i^t\rangle = S_D\,\langle \tau_i^t, \overline{u_i^t}\rangle$ and $\overline{[\tau/T]u_{2_j}^t = u_{2_j}^t}$.
Then by lemma B.18 $\overline{\Omega^t \vdash_T \gamma_j : A_{Dj}\,\langle \tau_1^t, \overline{u_{sj}^t}\rangle \sim u_{2_j}^t}$.
We can then apply rule (t-pack) to prove
$\Omega^t \mid \Gamma^t \mid \emptyset \vdash_T \mathsf{pack}\ (\tau_1^t, e^t, e_D^t, \overline{\gamma_j})\ \mathsf{as}\ \tau_2^t : \tau_2^t$ as required.

□

Lemma B.20 (Environment extension)**.** *If*

- $\langle \Gamma, \Theta\rangle \rightsquigarrow \langle \Omega^t, \Gamma^t\rangle$,
- $\Gamma, \Gamma' \mid \Theta, \Theta' \vdash \Gamma' \rightsquigarrow \langle \Omega_1^t, \Gamma_1^t\rangle$ *and*
- $\Gamma, \Gamma' \mid \Theta, \Theta' \vdash \Theta' \rightsquigarrow \langle \Omega_2^t, \Gamma_2^t\rangle$

where the domains of Γ, Γ' *and* Θ, Θ' *respectively do not overlap, then*
$\langle (\Gamma, \Gamma'), (\Theta, \Theta')\rangle \rightsquigarrow \langle (\Omega^t, \Omega_1^t, \Omega_2^t), (\Gamma^t, \Gamma_1^t, \Gamma_2^t)\rangle$.

Proof. From rule (tr-envs) we know that there are some $\Omega_3^t, \Omega_4^t, \Gamma_3^t, \Gamma_4^t$ such that
$\{\Omega_3^t, \Omega_4^t\} = \Omega^t$,
$\{\Gamma_3^t, \Gamma_4^t\} = \Gamma^t$,
$\Gamma \mid \Theta \vdash \Gamma \rightsquigarrow \langle \Omega_3^t, \Gamma_3^t\rangle$ and
$\Gamma \mid \Theta \vdash \Theta \rightsquigarrow \langle \Omega_4^t, \Gamma_4^t\rangle$.
A simple simultaneous induction on derivations $\Gamma \mid \Theta \vdash \Gamma \rightsquigarrow \langle \Omega_3^t, \Gamma_3^t\rangle$ and $\Gamma \mid \Theta \vdash \Theta \rightsquigarrow \langle \Omega_4^t, \Gamma_4^t\rangle$ using lemmas B.10, B.11 and B.12 lets us show that
$\Gamma, \Gamma' \mid \Theta, \Theta' \vdash \Gamma \rightsquigarrow \langle \Omega_3^t, \Gamma_3^t\rangle$ and $\Gamma, \Gamma' \mid \Theta, \Theta' \vdash \Theta \rightsquigarrow \langle \Omega_4^t, \Gamma_4^t\rangle$.
Then, applying lemma B.1 and the environment translation rules we get
$\Gamma, \Gamma' \mid \Theta, \Theta' \vdash \Gamma, \Gamma' \rightsquigarrow \langle (\Omega_3^t, \Omega_1^t), (\Gamma_3^t, \Gamma_1^t)\rangle$ and $\Gamma, \Gamma' \mid \Theta, \Theta' \vdash \Theta, \Theta' \rightsquigarrow \langle (\Omega_4^t, \Omega_2^t), (\Gamma_4^t, \Gamma_2^t)\rangle$.
Applying (tr-envs) we then get $\langle (\Gamma, \Gamma'), (\Theta, \Theta')\rangle \rightsquigarrow \langle (\Omega^t, \Omega_1^t, \Omega_2^t), (\Gamma^t, \Gamma_1^t, \Gamma_2^t)\rangle$ as required. □

Lemma B.21 (Soundness of supertrait hierarchy relation)**.** *Suppose* $\langle \Gamma, \Theta\rangle \rightsquigarrow \langle \Omega^t, \Gamma^t\rangle$.
If $\Gamma \vdash D\,\langle \overline{u_i}\rangle \Uparrow \{\overline{\beta_j}\}$ *and* $\Gamma \mid \Theta \Vdash \mathit{Self}_{\mathbf{U}} : D\,\langle \overline{u_i}\rangle$ *then* $\overline{\Gamma \mid \Theta \Vdash \mathit{Self}_{\mathbf{U}} : \beta_j}$.

Proof. From the relation $\Uparrow$'s only rule we get
$\Gamma \vdash D\,\langle \overline{u_i}\rangle \Uparrow \left\{\overline{\overline{\beta_{j_s}}},\ D\,\langle \overline{u_i}, A \mapsto \star\rangle\right\}$ where
$\dfrac{(D\,\langle \overline{X_i}\rangle\ \mathsf{where}\ \_, \mathsf{Self}_{\mathbf{U}} : \overline{D_s\,\langle \overline{u_{ns}}, A_s \mapsto u_s\rangle \rightsquigarrow \_}, A : \_, \_, \_) \in \Gamma \text{ and } \Gamma \vdash D_s\,\langle \overline{[\overline{u_i/X_i}]u_{ns}}\rangle \Uparrow \{\overline{\beta_{j_s}}\}}^{s}$.

$\Gamma \mid \Theta \Vdash u : D\,\langle \overline{u_i, A \mapsto \star}\rangle$ by assumption.
Using lemma B.1 we get $(\forall \mathsf{Self}_{\mathbf{U}}, \overline{X_i}.(\mathsf{Self}_{\mathbf{U}} : D\,\langle \overline{X_i}\rangle) \Rightarrow \overline{\mathsf{Self}_{\mathbf{U}} : D\,\langle \overline{u_{ns}}, A_s \mapsto u_s\rangle \rightsquigarrow \_}^{s}) \in \Theta$.

Then using (c-ext) and (c-astar) we get $\Gamma \mid \Theta \Vdash \mathsf{Self}_{\mathbf{U}} : D_s \langle \overline{\overline{[u_i/X_i]}u_{n_s}}, A_s \mapsto \star \rangle$.
Then by IH we get $\overline{\overline{\Gamma \mid \Theta \Vdash \mathsf{Self}_{\mathbf{U}} : \beta_{j_s}}}$ as required.

□

Lemma B.22. *Suppose* $\langle \Gamma, (\Theta_1, \Theta_2) \rangle \rightsquigarrow \langle \Omega^t, \Gamma^t \rangle$.
If $\Gamma \mid \Theta_1 \mid \Theta_2 \vdash \pi_1 \Rrightarrow \pi_2 \rightsquigarrow \gamma$ *and* $\Gamma \mid \Theta_1, \Theta_2 \vdash_{\mathrm{WF}} \pi_1 \rightsquigarrow \tau_1^t$ *then*
$\Gamma \mid \Theta_1 \vdash_{\mathrm{WF}} \pi_2 \rightsquigarrow \tau_2^t$ *and* $\Omega^t \vdash_T \gamma : \tau_1^t \sim \tau_2^t$.

Proof. By rule induction on $\Gamma \mid \Theta_1 \mid \Theta_2 \vdash \pi_1 \Rrightarrow \pi_2 \rightsquigarrow \gamma$.

- Let $\pi_1 = u_1 : D \langle \overline{u_{1i}}, A \mapsto u_3 \rangle$ and $\pi_2 = u_2 : D \langle \overline{u_{2i}}, A \mapsto u_4 \rangle$.
  Then $\Gamma \mid \Theta_1 \mid \Theta_2 \vdash u_1 : D \langle \overline{u_{1i}}, A \mapsto u_3 \rangle \Rrightarrow u_2 : D \langle \overline{u_{2i}}, A \mapsto u_4 \rangle \rightsquigarrow S_D \langle \gamma_1, \overline{\gamma_i} \rangle$ and
  $\Gamma \mid \Theta_1, \Theta_2 \Vdash u_1 \sim u_2 \rightsquigarrow \gamma_1$,
  $\overline{\Gamma \mid \Theta_1, \Theta_2 \Vdash u_{1i} \sim u_{2i} \rightsquigarrow \gamma_i}^{\,i}$,
  $\Gamma \mid \Theta_1, \Theta_2 \Vdash u_3 \sim u_4$,
  $\Gamma \mid \Theta_1 \vdash_{\mathrm{WF}} u_2 : D \langle \overline{u_{2i}}, A \mapsto u_2 \rangle$.
  By lemma B.3, $\Gamma \mid \Theta_1, \Theta_2 \vdash_{\mathrm{WF}} u_1 \rightsquigarrow u_1^t$ and $\overline{\Gamma \mid \Theta_2, \Theta_2 \vdash_{\mathrm{WF}} u_{1i} \rightsquigarrow u_{1i}^t}$.
  By lemma B.3 and lemma B.10,
  $\Gamma \mid \Theta_1, \Theta_2 \vdash_{\mathrm{WF}} u_2 \rightsquigarrow u_2^t$, and $\overline{\Gamma \mid \Theta_1, \Theta_2 \vdash_{\mathrm{WF}} u_{2i} \rightsquigarrow u_{2i}^t}$, and
  $\tau_1^t = S_D \langle u_1^t, \overline{u_{1i}^t} \rangle$ and $\tau_2^t = S_D \langle u_2^t, \overline{u_{2i}^t} \rangle$
  By theorem B.18, $\Omega^t \vdash_T \gamma_1 : u_1^t \sim u_2^t$ and $\Omega^t \vdash_T \gamma_1 : u_{1i}^t \sim u_{1i}^t$.
  Then using (co-struct), $\Omega^t \vdash_T S_D \langle \gamma_1, \overline{\gamma_i} \rangle : S_D \langle u_1^t, \overline{u_{1i}^t} \rangle \sim S_D \langle u_2^t, \overline{u_{2i}^t} \rangle$ as required.
- Let $\pi_1 = u_1 : D \langle \overline{u_{1i}}, A \mapsto \star \rangle$ and $\pi_2 = u_2 : D \langle \overline{u_{2i}}, A \mapsto \star \rangle$.
  Then $\Gamma \mid \Theta_1 \mid \Theta_2 \vdash u_1 : D \langle \overline{u_{1i}}, A \mapsto \star \rangle \Rrightarrow u_2 : D \langle \overline{u_{2i}}, A \mapsto \star \rangle \rightsquigarrow S_D \langle \gamma_1, \overline{\gamma_i} \rangle$ and
  $\Gamma \mid \Theta_1, \Theta_2 \Vdash u_1 \sim u_2 \rightsquigarrow \gamma_1$,
  $\overline{\Gamma \mid \Theta_1, \Theta_2 \Vdash u_{1i} \sim u_{2i} \rightsquigarrow \gamma_i}^{\,i}$,
  $\Gamma \mid \Theta_1 \vdash_{\mathrm{WF}} u_2 : D \langle \overline{u_{2i}}, A \mapsto u_2 \rangle$.
  By lemma B.3, $\Gamma \mid \Theta_1, \Theta_2 \vdash_{\mathrm{WF}} u_1 \rightsquigarrow u_1^t$ and $\overline{\Gamma \mid \Theta_2, \Theta_2 \vdash_{\mathrm{WF}} u_{1i} \rightsquigarrow u_{1i}^t}$.
  By lemma B.3 and lemma B.10,
  $\Gamma \mid \Theta_1, \Theta_2 \vdash_{\mathrm{WF}} u_2 \rightsquigarrow u_2^t$, and $\overline{\Gamma \mid \Theta_1, \Theta_2 \vdash_{\mathrm{WF}} u_{2i} \rightsquigarrow u_{2i}^t}$, and
  $\tau_1^t = S_D \langle u_1^t, \overline{u_{1i}^t} \rangle$ and $\tau_2^t = S_D \langle u_2^t, \overline{u_{2i}^t} \rangle$
  By theorem B.18, $\Omega^t \vdash_T \gamma_1 : u_1^t \sim u_2^t$ and $\Omega^t \vdash_T \gamma_1 : u_{1i}^t \sim u_{1i}^t$.
  Then using (co-struct), $\Omega^t \vdash_T S_D \langle \gamma_1, \overline{\gamma_i} \rangle : S_D \langle u_1^t, \overline{u_{1i}^t} \rangle \sim S_D \langle u_2^t, \overline{u_{2i}^t} \rangle$ as required.

□

Lemma B.23 (Soundness of dictionary path relation)**.** *If*

- $\langle \Gamma, \Theta \rangle \rightsquigarrow \langle \Omega^t, \Gamma^t \rangle$,

- $\Gamma \vdash u : D_1\,\langle \overline{u_i} \rangle \mapsto D_2\,\langle \overline{u_j} \rangle \rightsquigarrow e_1^t$,
- $\overline{\Gamma \mid \Theta \vdash_{\text{WF}} u_j \rightsquigarrow u_j^t}$,
- $\Gamma \mid \Theta \vdash_{\text{WF}} u : D_1\,\langle \overline{u_i} \rangle \rightsquigarrow S_{D1}\,\langle u^t, \overline{u_i^t} \rangle$,
- $\Gamma \mid \Theta \Vdash u : D_1\,\langle \overline{u_i} \rangle$
- $(D\,\langle \overline{X_j} \rangle \textit{ where } \_, \textit{Self}_{\mathbf{U}} : \_, \_, f, \_)$,
- $(f : \forall \textit{Self}_{\mathbf{U}}, \overline{X_j}.(\textit{Self}_{\mathbf{U}} : D_2\,\langle \overline{X_j} \rangle) \Rightarrow \sigma) \in \Gamma$, *and*
- $\Gamma \mid \Theta \vdash_{\text{WF}} \sigma \rightsquigarrow \sigma^t$

then $\Omega^t \mid \Gamma^t, x_{dict} : S_{D1}\,\langle u^t, \overline{u_i^t} \rangle \mid \emptyset \vdash_T x_{dict}.e_1^t : [u^t/\textit{Self}_{\mathbf{U}}][\overline{u_j^t/X_j}]\sigma^t$.

Proof. By induction on derivations $\Gamma \mid \Theta \vdash_{\text{WF}} u : D_1\,\langle \overline{u_i} \rangle \mapsto D_2\,\langle \overline{u_j} \rangle \rightsquigarrow e_1^t$.

- **Case** (path1): $\Gamma \vdash u : D_1\,\langle \overline{u_i} \rangle \mapsto D_2\,\langle \overline{u_j} \rangle \rightsquigarrow x.e^t$ where
  $(D_1\,\langle \overline{X_i} \rangle \text{ where } \_, \mathsf{Self}_{\mathbf{U}} : \overline{\beta_s \rightsquigarrow x_s}, \_, \_)$,
  $(D_3\,\langle \overline{u_k}, A \mapsto \_ \rangle \rightsquigarrow x) \in \{[u/\mathsf{Self}_{\mathbf{U}}][\overline{u_i/X_i}]\beta_s \rightsquigarrow x_s\}$ and
  $\Gamma \vdash u : D_3\,\langle \overline{u_k} \rangle \mapsto D_2\,\langle \overline{u_j} \rangle \rightsquigarrow e^t$.

  From lemma B.1 we get
  $(\forall \overline{X_p}.(\mathsf{Self}_{\mathbf{U}} : D_1\,\langle \overline{X_i} \rangle) \Rightarrow \mathsf{Self}_{\mathbf{U}} : D_3\,\langle \overline{u'_k} \rangle \rightsquigarrow \langle x, c \rangle) \in \Theta$,
  $\Gamma, \overline{X_p} \mid \Theta, \mathsf{Self}_{\mathbf{U}} : D_1\,\langle \overline{X_i} \rangle \vdash_{\text{WF}} \mathsf{Self}_{\mathbf{U}} : D_3\,\langle \overline{u'_k} \rangle \rightsquigarrow \langle \tau_3^t, \omega_3 \rangle$ and
  $S_{D1}\,\langle \overline{X_p} \rangle\,\{\ldots, x : \tau_3^t, \ldots\} \in \Omega^t$
  where $\overline{u_k = [u/\mathsf{Self}_{\mathbf{U}}][\overline{u_i/X_i}]u'_k}$.

  By lemma B.3, $\tau_3^t = S_{D3}\,\langle \mathsf{Self}_{\mathbf{U}}, \overline{u_k'^t} \rangle$ for some $\overline{u_k'^t}$.

  Then, using rule (c-ext) we get $\Gamma \mid \Theta \Vdash u : D_3\,\langle \overline{u_k} \rangle$.

  By lemma B.3, $\Gamma \mid \Theta \vdash_{\text{WF}} u \rightsquigarrow u^t$ and $\overline{\Gamma \mid \Theta \vdash_{\text{WF}} u_i \rightsquigarrow u_i^t}$.
  Then, by lemma B.5 and lemma B.3,
  $\Gamma \mid \Theta \vdash_{\text{WF}} u : D_3\,\langle \overline{u_k} \rangle \rightsquigarrow \langle S_{D3}\,\langle u^t, \overline{u_k^t} \rangle, \_ \rangle$ where $\overline{u_k^t = [u^t/\mathsf{Self}_{\mathbf{U}}][\overline{u_i^t/X_i}]u_k'^t}$.

  Then, by IH, we get $\Omega^t \mid \Gamma^t, x_3 : S_{D3}\,\langle u^t, \overline{u_k^t} \rangle \mid \emptyset \vdash_T x_3.e^t : [u^t/\mathsf{Self}_{\mathbf{U}}][\overline{u_j^t/X_j}]\sigma^t$.

  Using rules (t-var) and (t-proj) we get
  $\Omega^t \mid \Gamma^t, x_{\mathsf{dict}} : S_{D1}\,\langle u^t, \overline{u_i^t} \rangle \mid \emptyset \vdash_T x_{\mathsf{dict}}.x : S_{D3}\,\langle u^t, \overline{u_k^t} \rangle$.

  Then, using lemma A.5 and A.6, we get
  $\Omega^t \mid \Gamma^t, x_{\mathsf{dict}} : S_{D1}\,\langle u^t, \overline{u_i^t} \rangle \mid \emptyset \vdash_T x_{\mathsf{dict}}.x.e^t : [u^t/\mathsf{Self}_{\mathbf{U}}][\overline{u_j^t/X_j}]\sigma^t$ as required.

- **Case** (path2): $\Gamma \vdash u : D_1\,\langle \overline{u_i} \rangle \mapsto D_1\,\langle \overline{u_i} \rangle \rightsquigarrow f$
  where $(D_1\,\langle \overline{X_i} \rangle \text{ where } \_, \_, \_, f, \_)$.

By lemma B.1, $S_{D1}\,\langle \mathsf{Self}_{\mathbf{U}}, \overline{X_i}\rangle\,\{f : \sigma^t, \ldots\} \in \Omega^t$.

Note that $\overline{u_i^t} = \overline{u_j^t}$ and $\overline{X_i} = \overline{X_j}$.

Then, using (t-var) and (t-proj) we get
$\Omega^t \mid \Gamma^t, x_{\mathsf{dict}} : S_{D1}\,\langle u^t, \overline{u_i^t}\rangle \mid \emptyset \vdash_T x_{\mathsf{dict}}.f : [u^t/\mathsf{Self}_{\mathbf{U}}][\overline{u_j^t/X_j}]\sigma^t$ as required.

□

Lemma B.24 (Soundness of auxiliary object-safe trait relation)**.** *If*

1. $\Gamma \mid \Theta \mid \overline{\theta_d \rightsquigarrow \langle \_, c'_d\rangle} \vdash_{os} u_{obj} : D\,\langle \overline{u_i}, A \mapsto X\rangle \rightsquigarrow \langle e_D^t, \overline{\pi_s \rightsquigarrow x'_s}, \overline{\pi_a \rightsquigarrow x'_a}\rangle$,
2. $\langle \Gamma, \Theta\rangle \rightsquigarrow \langle \Omega^t, \Gamma^t\rangle$,
3. $\langle \Gamma, (\Theta, \overline{\theta_d \rightsquigarrow \langle \_, c'_d\rangle})\rangle \rightsquigarrow \langle (\Omega^t, \overline{c'_d : \vartheta'_d}), (\Gamma^t, \_)\rangle$
4. $\Gamma \mid \Theta \vdash_{\mathrm{WF}} u_{obj} \rightsquigarrow u^t_{obj}$
5. $\Gamma \mid \Theta, \overline{\theta_d \rightsquigarrow \langle \_, c'_d\rangle} \vdash_{\mathrm{WF}} u_{obj} : D\,\langle \overline{u_i}\rangle \rightsquigarrow \tau_D^t$.

then

1. $\overline{\Gamma \mid \Theta \vdash_{\mathrm{WF}} \pi_s \rightsquigarrow \tau'^t_s}$,
2. $\overline{\Gamma \mid \Theta \vdash_{\mathrm{WF}} \pi_a \rightsquigarrow \tau'^t_a}$,
3. $\Omega^t, \overline{c'_d : \vartheta'_d} \mid \Gamma^t, \overline{x'_s : \tau'^t_s}, \overline{x'_a : \tau'^t_a} \mid \emptyset \vdash_T e_D^t : \tau_D^t$.

Proof. By lemma B.3,
$\Gamma \mid \Theta, \overline{\theta_d \rightsquigarrow \langle \_, c'_d\rangle} \vdash_{\mathrm{WF}} u_{\mathsf{obj}} \rightsquigarrow u^t_{\mathsf{obj}}$,
$\overline{\Gamma \mid \Theta, \overline{\theta_d \rightsquigarrow \langle \_, c'_d\rangle} \vdash_{\mathrm{WF}} u_i \rightsquigarrow u_i^t}$,
and $\tau_D^t = S_D\,\langle u^t_{\mathsf{obj}}, \overline{u_i^t}\rangle$ for some $u^t_{\mathsf{obj}}, \overline{u_i^t}$.

Let $\overline{X_p} = \mathsf{Self}_{\mathbf{U}}, \overline{X_i}$ and $\overline{u_p} = u_{\mathsf{obj}}, \overline{u_i}$ and $\overline{u_p^t} = u^t_{\mathsf{obj}}, \overline{u_i^t}$.

From the relation $\vdash_{\mathsf{os}}$'s side conditions,
$(D\,\langle \overline{X_i}\rangle$ where $\_, \mathsf{Self}_{\mathbf{U}} : \overline{\beta_s \rightsquigarrow x_s}, A : \overline{\beta_a \rightsquigarrow x_a}, f, \mathsf{obj\text{-}safe}) \in \Gamma$.

By lemma B.1, $S_D\,\langle \overline{X_p}\rangle\,\{f : \sigma^t, \overline{x_s : \tau_s^t}, \overline{x_a : \tau_a^t}\} \in \Omega^t$ where
$\sigma^t = \mathsf{fn}(\&\mathsf{Self}_{\mathbf{U}}, \overline{\tau_m^t}) \to \tau_R^t$,
$\dfrac{}{\Gamma, \overline{X_p} \mid \Theta, \mathsf{Self}_{\mathbf{U}} : D\,\langle \overline{X_i}\rangle \vdash_{\mathrm{WF}} \mathsf{Self}_{\mathbf{U}} : \beta_s \rightsquigarrow \langle \tau_s^t, \omega_s\rangle}\,{}^{s}$ and
$\dfrac{}{\Gamma, \overline{X_p} \mid \Theta, \mathsf{Self}_{\mathbf{U}} : D\,\langle \overline{X_i}\rangle \vdash_{\mathrm{WF}} A_D\,\langle \overline{X_p}\rangle : \beta_a \rightsquigarrow \langle \tau_a^t, \omega_a\rangle}\,{}^{a}$.

By lemma B.8,
$\dfrac{}{\Gamma \mid \Theta, \overline{\theta_d \rightsquigarrow \langle \_, c'_d \rangle} \vdash_{\mathrm{WF}} u_{\mathsf{obj}} : [\overline{u_p/X_p}]\beta_s \rightsquigarrow \langle [\overline{u^t_p/X_p}]\tau^t_s, \_ \rangle}^{s}$ and
$\dfrac{}{\Gamma \mid \Theta, \overline{\theta_d \rightsquigarrow \langle \_, c'_d \rangle} \vdash_{\mathrm{WF}} A_D \langle \overline{u_p} \rangle : [\overline{u_p/X_p}]\beta_a \rightsquigarrow \langle [\overline{u_p/X_p}]\tau^t_a, \_ \rangle}^{a}$

By lemma B.22, there are some $\overline{\tau'^t_s}, \overline{\tau'^t_a}$ such that
$\overline{\Gamma \mid \Theta \vdash_{\mathrm{WF}} \pi_s \rightsquigarrow \tau'^t_s}$,
$\overline{\Gamma \mid \Theta \vdash_{\mathrm{WF}} \pi_a \rightsquigarrow \tau'^t_a}$,
$\dfrac{}{\Omega^t, \overline{c'_d : \vartheta'_d} \vdash_T \gamma_s : [\overline{u^t_p/X_p}]\tau^t_s \sim \tau'^t_s}^{s}$ and
$\dfrac{}{\Omega^t, \overline{c'_d : \vartheta'_d} \vdash_T \gamma_a : [\overline{u^t_p/X_p}]\tau^t_a \sim \tau'^t_a}^{a}$.

Let $\Theta'' = \Theta, T : D_{\mathsf{obj}} \langle \overline{X_h} \rangle \rightsquigarrow x$.
Using lemma B.3 on u_{obj} and (wf-trcons) we get
$\Gamma' \mid \Theta \vdash_{\mathrm{WF}} T : D_{\mathsf{obj}} \langle \overline{X_h} \rangle \rightsquigarrow S_{D\,\mathsf{obj}} \langle T, \overline{X_h} \rangle$.
Using lemma B.20 we get $\langle \Gamma', \Theta'' \rangle \rightsquigarrow \langle \Omega^t, (\Gamma^t, T, x : S_{D\,\mathsf{obj}} \langle T, \overline{X_h} \rangle) \rangle$.
Using lemma B.3 we know that there are some types $\overline{u'^t_i}$ such that
$\overline{\Gamma' \mid \Theta'' \vdash_{\mathrm{WF}} u'_i \rightsquigarrow u'^t_i}$.
By (c-ext) we get $\Gamma' \mid \Theta'' \Vdash T : D_{\mathsf{obj}} \langle \overline{X_h} \rangle$.
We can then use lemma B.23 to get
$\Omega^t \mid \Gamma^t, T, x : S_{D\,\mathsf{obj}} \langle T, \overline{X_h} \rangle \mid \emptyset \vdash_T x_{\mathsf{dict}}.e^t_P : \mathsf{fn}(\&T, \overline{[T/\mathsf{Self}_{\mathsf{U}}][\overline{u'^t_i/X_i}]\tau^t_m}) \to [T/\mathsf{Self}_{\mathsf{U}}][\overline{u'^t_i/X_i}]\tau^t_R$.

From the side conditions we have
$\Gamma' = \Gamma, T$ and
$\Theta' = \Theta, T : D_{\mathsf{obj}} \langle \overline{X_h} \rangle, \overline{T : D_d \langle \overline{u_{nd}}, A \mapsto X_d \rangle} \rightsquigarrow \langle \_, c_d \rangle$

Using a combination of lemma B.3 and well-formedness rules we have $\langle \Gamma', \Theta' \rangle \rightsquigarrow \langle \Omega^t_0, \Gamma^t_3 \rangle$ for some Ω^t_0, Γ^t_0 where $\Omega^t_0 = \Omega^t$, $\overline{c'_d : \vartheta'_d}$, $\overline{c_d : A_{D\,d} \langle T, \overline{u^t_{nd}} \rangle \sim X_d}$.
Using lemma B.3 we get
$\Gamma, \mathsf{Self}_{\mathsf{U}}, \overline{X_i} \mid \Theta, \mathsf{Self}_{\mathsf{U}} : D \langle \overline{X_i} \rangle \vdash_{\mathrm{WF}} \tau_R \rightsquigarrow \tau^t_R$ and
$\dfrac{}{\Gamma, \mathsf{Self}_{\mathsf{U}}, \overline{X_i} \mid \Theta, \mathsf{Self}_{\mathsf{U}} : D \langle \overline{X_i} \rangle \vdash_{\mathrm{WF}} \tau_m \rightsquigarrow \tau^t_m}^{m}$.
Using lemma B.10 we get
$\dfrac{\Gamma', \mathsf{Self}_{\mathsf{U}}, \overline{X_i} \mid \Theta', \mathsf{Self}_{\mathsf{U}} : D \langle \overline{X_i} \rangle \vdash_{\mathrm{WF}} \tau_R \rightsquigarrow \tau^t_R,}{\Gamma', \mathsf{Self}_{\mathsf{U}}, \overline{X_i} \mid \Theta', \mathsf{Self}_{\mathsf{U}} : D \langle \overline{X_i} \rangle \vdash_{\mathrm{WF}} \tau_m \rightsquigarrow \tau^t_m}^{m}$.
$\dfrac{}{\Gamma' \mid \Theta', u_{\mathsf{obj}} : D \langle \overline{u_i} \rangle \vdash_{\mathrm{WF}} u_p \rightsquigarrow u^t_p}^{p}$, and
$\dfrac{}{\Gamma' \mid \Theta', T : D \langle \overline{u'_i} \rangle \vdash_{\mathrm{WF}} u'_i \rightsquigarrow u^t_i}^{i}$.
Using lemma B.7 we get
$\Gamma' \mid \Theta', u_{\mathsf{obj}} : D \langle \overline{u_i} \rangle \vdash_{\mathrm{WF}} [\overline{u_p/X_p}]\tau_R \rightsquigarrow [\overline{u^t_p/X_p}]\tau^t_R$,

$$\frac{}{\Gamma' \mid \Theta', u_{\mathsf{obj}} : D\,\langle \overline{u_i} \rangle \vdash_{\mathrm{WF}} [\overline{u_p/X_p}]\tau_m \rightsquigarrow [\overline{u_i^t/X_i}]\tau_m^t}\,m$$

$$\frac{\Gamma' \mid \Theta', T : D\,\langle \overline{u_i'} \rangle \vdash_{\mathrm{WF}} [T/\mathsf{Self}_{\mathbf{U}}][\overline{u_i'/X_i}]\tau_R \rightsquigarrow [T/\mathsf{Self}_{\mathbf{U}}][\overline{u_i'^t/X_i}]\tau_R^t}{\Gamma' \mid \Theta', T : D\,\langle \overline{u_i'} \rangle \vdash_{\mathrm{WF}} [T/\mathsf{Self}_{\mathbf{U}}][\overline{u_i'/X_i}]\tau_m \rightsquigarrow [T/\mathsf{Self}_{\mathbf{U}}][\overline{u_i'^t/X_i}]\tau_m^t}\,m$$

From lemma's assumptions and lemma B.11 we get $\Gamma' \mid \Theta' \vdash_{\mathrm{WF}} u_{\mathsf{obj}} : D\,\langle \overline{u_i} \rangle \rightsquigarrow \tau_D^t$.

We can then use lemma B.4 to get

$$\frac{\Gamma' \mid \Theta' \vdash_{\mathrm{WF}} [\overline{u_p/X_p}]\tau_R \rightsquigarrow [\overline{u_p^t/X_p}]\tau_R^t}{\Gamma' \mid \Theta' \vdash_{\mathrm{WF}} [\overline{u_p/X_p}]\tau_m \rightsquigarrow [\overline{u_i^t/X_i}]\tau_m^t}\,m$$

Since $(T : D\,\langle \overline{u_i'} \rangle) \in \{T : D_d\,\langle \overline{u_{nd}} \rangle\}$, we can use lemma B.4 to get

$$\frac{\Gamma' \mid \Theta' \vdash_{\mathrm{WF}} [T/\mathsf{Self}_{\mathbf{U}}][\overline{u_i'/X_i}]\tau_R \rightsquigarrow [T/\mathsf{Self}_{\mathbf{U}}][\overline{u_i'^t/X_i}]\tau_R^t}{\Gamma' \mid \Theta' \vdash_{\mathrm{WF}} [T/\mathsf{Self}_{\mathbf{U}}][\overline{u_i'/X_i}]\tau_m \rightsquigarrow [T/\mathsf{Self}_{\mathbf{U}}][\overline{u_i'^t/X_i}]\tau_m^t}\,m \text{ and}$$

Then by lemma B.18 we have

$$\frac{\Omega_0^t \vdash \gamma : [T/\mathsf{Self}_{\mathbf{U}}][\overline{u_i^t/X_i}]\tau_R^t \sim [\overline{u_p^t/X_p}]\tau_R^t}{\Omega_0^t \vdash \gamma_m : [\overline{u_p^t/X_p}]\tau_m^t \sim [T/\mathsf{Self}_{\mathbf{U}}][\overline{u_i^t/X_i}]\tau_m^t}\,m \text{ and}$$

Using lemma B.3 we get $\tau_D^t = S_D\,\langle u_{\mathsf{obj}}^t, \overline{u_i^t} \rangle$.

Let $\overline{u_p'^t} = T, \overline{u_i'^t}$.

Let $e_1^t = \mathsf{let}\ (T, x_{\mathsf{val}}, x_{\mathsf{dict}}, \overline{c_d}) = \mathsf{unpack}\ x_{\mathsf{obj}}\ \mathsf{in}\ e_2^t$,
$e_2^t = x_{\mathsf{dict}}.e_P^t.f(x_{\mathsf{val}}, \overline{x_m \blacktriangleright \gamma_m}) \blacktriangleright \gamma$,
$\Gamma_0^t = \Gamma^t, \overline{x_s' : \tau_s'^t}, \overline{x_a' : \tau_a'^t}$,
$\Gamma_1^t = \Gamma_0^t, x_{\mathsf{obj}} : \&u_{\mathsf{obj}}^t, \overline{x_m : [\overline{u_p^t/X_p}]\tau_m^t}$ and
$\Gamma_2^t = \Gamma_1^t, T, x_{\mathsf{val}} : \&T, x_{\mathsf{dict}} : S_D\,\langle T, \overline{X_h} \rangle$.

By lemma A.5, $\Omega_0^t \mid \Gamma_2^t \vdash x_{\mathsf{dict}}.e_P^t : S_D\,\langle T, \overline{u_i'^t} \rangle$.

We can then obtain the derivation

$$\dfrac{\dfrac{\dfrac{\mathscr{D}_1 \quad \mathscr{D}_2 \quad \overline{\mathscr{D}_m}}{\Omega_0^t \mid \Gamma_2^t \vdash x_{\mathsf{dict}}.e_P^t.f(x_{\mathsf{val}}, \overline{x_m \blacktriangleright \gamma_m}) : [\overline{u_p'^t/X_p}]\tau_R^t}\,\text{(t-fapp)} \qquad \Omega_0^t \vdash \gamma : [\overline{u_p'^t/X_p}]\tau_R^t \sim [\overline{u_p^t/X_p}]\tau_R^t}{\Omega_0^t \mid \Gamma_2^t \vdash x_{\mathsf{dict}}.e_P^t.f(x_{\mathsf{val}}, \overline{x_m \blacktriangleright \gamma_m}) \blacktriangleright \gamma : [\overline{u_p^t/X_p}]\tau_R^t}\,\text{(t-coerce)}}{\Omega^t, \overline{c_d' : \vartheta_d'} \mid \Gamma_1^t \vdash \mathsf{let}\ (T, x_{\mathsf{val}}, x_{\mathsf{dict}}, \overline{c_d}) = \mathsf{unpack}\ x_{\mathsf{obj}}\ \mathsf{in}\ e_2^t : [\overline{u_p^t/X_p}]\tau_R^t}\,\text{(t-unpack)}}{\Omega^t, \overline{c_d' : \vartheta_d'} \mid \Gamma_0^t \vdash \mathsf{fn}\ (x_{\mathsf{obj}} : \&u_{\mathsf{obj}}^t, \overline{x_m : [\overline{u_p^t/X_p}]\tau_m^t})\,\{\,e_1^t\,\} : \mathsf{fn}(\&u_{\mathsf{obj}}^t, \overline{[\overline{u_p^t/X_p}]\tau_m^t}) \to [\overline{u_p^t/X_p}]\tau_R^t}\,\text{(t-fabs)}$$

where

$$\mathscr{D}_1 = \dfrac{\Omega_0^t \mid \Gamma_2^t \vdash x_{\mathsf{dict}}.e_P^t : S_D\,\langle T, \overline{u_i'^t}\rangle}{\Omega_0^t \mid \Gamma_2^t \vdash x_{\mathsf{dict}}.e_P^t.f : \mathsf{fn}(\&T, \overline{[\overline{u_p'^t/X_p}]\tau_m^t}) \to [\overline{u_p'^t/X_p}]\tau_R^t}\ \text{(t-proj)}$$

$$\mathscr{D}_2 = \dfrac{}{\Omega_2^t \mid \Gamma_2^t \vdash x_{\mathsf{val}} : \&T}\ \text{(t-var)}$$

$$\mathscr{D}_m = \dfrac{\dfrac{}{\Omega_0^t \mid \Gamma_2^t \vdash x_m : [\overline{u_p^t/X_p}]\tau_m^t}\ \text{(t-var)} \qquad \Omega_0^t \vdash \gamma_m : [\overline{u_p^t/X_p}]\tau_m^t \sim [\overline{u_{;p}^t/X_p}]\tau_m^t}{\Omega_0^t \mid \Gamma_2^t \vdash x_m \blacktriangleright \gamma_m : [\overline{u_{;p}^t/X_p}]\tau_m^t}\ \text{(t-coerce)}$$

In all above derivations we assume an empty store typing environment Σ.

We have proved earlier that
$\dfrac{}{\Omega^t, \overline{c_d' : \vartheta_d'} \vdash_T \gamma_s : [\overline{u_p^t/X_p}]\tau_s^t \sim \tau_s'^t}{}^{s}$ and
$\dfrac{}{\Omega^t, \overline{c_d' : \vartheta_d'} \vdash_T \gamma_a : [\overline{u_p^t/X_p}]\tau_a^t \sim \tau_a'^t}{}^{a}$.
Then, using (co-sym) and (t-coerce), we get
$\dfrac{}{\Omega^t, \overline{c_d' : \vartheta_d'} \mid \Gamma_0^t \mid \emptyset \vdash_T x_s' \blacktriangleright \mathsf{sym}\ \gamma_s : [\overline{u_p^t/X_p}]\tau_s^t}{}^{s}$ and
$\dfrac{}{\Omega^t, \overline{c_d' : \vartheta_d'} \mid \Gamma_0^t \mid \emptyset \vdash_T x_a' \blacktriangleright \mathsf{sym}\ \gamma_s : [\overline{u_p^t/X_p}]\tau_a^t}{}^{a}$.

We can then use (t-newstruct) to prove $\Omega^t, \overline{c_d' : \vartheta_d'} \mid \Gamma_0^t \mid \emptyset \vdash_T e_D^t : S_D\,\langle \overline{u_p^t}\rangle$ as required. □

Lemma B.25 (Well-typed items generate well-formed environments)**.**
Suppose $\Gamma' \subseteq \Gamma$ *and* $\Theta' \subseteq \Theta$. *If* $\Gamma \mid \Theta \vdash item : \Gamma' \mid \Theta' \rightsquigarrow pgm^t$ *then there are some* $\Omega_1^t, \Omega_2^t, \Gamma_1^t, \Gamma_2^t$ *such that* $\Gamma \mid \Theta \vdash \Gamma' \rightsquigarrow \langle \Omega_1^t, \Gamma_1^t\rangle$ *and* $\Gamma \mid \Theta \vdash \Theta' \rightsquigarrow \langle \Omega_2^t, \Gamma_2^t\rangle$.

Proof. By induction on derivations $\Gamma \mid \Theta \vdash item : \Gamma' \mid \Theta' \rightsquigarrow pgm^t$:

- **Case** (struct): $\Gamma' = \{S\,\langle \overline{X_j}\rangle\ \langle \overline{x_i : \tau_i}\rangle\}$ and $\Theta' = \emptyset$
  where $\overline{\Gamma, \overline{X_j} \mid \Theta \vdash_{\mathrm{WF}} \tau_i \rightsquigarrow \tau_i^t}$.

  By $(\Theta\emptyset)$, $\Gamma \mid \Theta \vdash \emptyset \rightsquigarrow \langle \emptyset, \emptyset\rangle$.

  By $(\Gamma\mathrm{sct})$, $\Gamma \mid \Theta \vdash S\,\langle \overline{X_j}\rangle\,\{\overline{x_i : \tau_i}\} \rightsquigarrow \langle \{S\,\langle \overline{x_j}\rangle\}, \emptyset\rangle$ as required.

- **Case** (fun): $\Gamma' = \{f : \sigma\}$ and $\Theta' = \emptyset$
  where $\Gamma \mid \Theta \vdash_{\mathrm{WF}} \sigma \rightsquigarrow \sigma^t$.

  By $(\Theta\emptyset)$, $\Gamma \mid \Theta \vdash \emptyset \rightsquigarrow \langle \emptyset, \emptyset\rangle$.

  By $(\Gamma\mathrm{var})$, $\Gamma \mid \Theta \vdash \{f : \sigma\} \rightsquigarrow \langle \emptyset, \{f : \sigma^t\}\rangle$.

- **Case** (trait): $\Gamma' = \{(D\,\langle \overline{X_i}\rangle\ \mathsf{where}\ \overline{\pi_h}, \mathit{Self} : \overline{\beta_s \rightsquigarrow x_s}^{s}, A : \overline{\beta_a \rightsquigarrow x_a}^{a}, \mathsf{obj\text{-}unsafe}), f : \sigma'\}$ and
  $\Theta' = \{\overline{\forall X_p.(\mathit{Self} : D\,\langle \overline{X_i}\rangle) \Rightarrow \mathit{Self} : \beta_s \rightsquigarrow \langle x_{D,s}, c_{D,s}\rangle}^{s},$
  $\overline{\forall X_p.(\mathit{Self} : D\,\langle \overline{X_i}\rangle) \Rightarrow A_D\,\langle \overline{X_p}\rangle : \beta_a \rightsquigarrow \langle x_{D,a}, c_{D,a}\rangle}^{a}\}$.

The rule's side conditions are sufficient to prove with (Γvar) and (Γtrt) that
$\Gamma \mid \Theta \vdash \Gamma' \rightsquigarrow \langle \{S\,\langle \overline{X_p}\rangle\,\{f:\sigma^t, \overline{x_s:\tau_s^t}, \overline{x_a:\tau_a^t}\}, A_D\,\langle \overline{X_p}\rangle\}, \{f:\sigma'^t\}\rangle$.

With the rule's side conditions we can also use $\Theta\theta$ twice to prove
$\Gamma \mid \Theta \vdash \Theta' \rightsquigarrow$
$\langle \{\overline{c_{D,s}:\vartheta_s}, \overline{c_{D,a}:\vartheta_a}\}, \{\overline{x_{D,s}:\forall \overline{X_p}.\mathsf{fn}(S_D\,\langle \overline{X_p}\rangle) \to \tau_s^t}, \overline{x_{D,a}:\forall \overline{X_p}.\mathsf{fn}(S_D\,\langle \overline{X_p}\rangle) \to \tau_a^t}\}\rangle$
as required.

- **Case** (impl): $\Gamma' = \emptyset$ and $\Theta' = \{\forall \overline{X_k}.\overline{\pi_j} \Rightarrow u : D\,\langle \overline{u_i}, A \mapsto u_A\rangle \rightsquigarrow \langle x_{u:D\langle \overline{u_i}\rangle}, c_{u:D\langle \overline{u_i}\rangle}\rangle\}$.

  Using the rule's side conditions with well-formedness judgments and lemma B.3, we can deduce $\Gamma \mid \Theta \vdash_{\mathrm{WF}} \forall \overline{X_k}.\overline{\pi_j} \Rightarrow u : D\,\langle \overline{u_i}, A \mapsto u_A\rangle \rightsquigarrow \langle \sigma_D^t, \overline{\forall X_k.A_D\,\langle \overline{u_p^t}\rangle \sim u_A^t}\rangle$.

  Then using $(\Theta\theta)$ we get
  $\Gamma \mid \Theta \vdash \Gamma' \rightsquigarrow \langle \{c_{u:D\langle \overline{u_i}\rangle} : \forall \overline{X_k}.A_D\,\langle \overline{u_p^t}\rangle \sim \forall \overline{X_k}.u_A^t\}, \{x_{u:D\langle \overline{u_i}\rangle} : \sigma_D^t\}\rangle$
  as required.

- **Case** (obj-trt): $\Gamma' = \{(D\,\langle \overline{X_i}\rangle \text{ where } \overline{\pi_h},\ \mathit{Self}:\overline{\beta_s \rightsquigarrow x_s},\ A:\overline{\beta_a \rightsquigarrow x_a},\ \text{obj-unsafe}), f:\sigma'\}$ and
  $\Theta' = \{\overline{\forall \overline{X_p}.(\mathit{Self}:D\,\langle \overline{X_i}\rangle) \Rightarrow \mathit{Self}:\beta_s \rightsquigarrow \langle x_{D,s}, c_{D,s}\rangle}^{s},$
  $\overline{\forall \overline{X_p}.(\mathit{Self}:D\,\langle \overline{X_i}\rangle) \Rightarrow A_D\,\langle \overline{X_p}\rangle : \beta_a \rightsquigarrow \langle x_{D,a}, c_{D,a}\rangle}^{a},$
  $\overline{\forall \overline{X_i}, \overline{X_d}.\overline{\pi_{dm}}^{m} \Rightarrow \pi_d \rightsquigarrow \langle x_d, c_d\rangle}^{d}\ \}$.

  The rule's side conditions are sufficient to prove with (Γvar) and (Γtrt) that
  $\Gamma \mid \Theta \vdash \Gamma' \rightsquigarrow \langle \{S\,\langle \overline{X_p}\rangle\,\{f:\sigma^t, \overline{x_s:\tau_s^t}, \overline{x_a:\tau_a^t}\}, A_D\,\langle \overline{X_p}\rangle\}, \{f:\sigma'^t\}\rangle$.

  With the rule's side conditions we can also use $\Theta\theta$ to prove
  $\Gamma \mid \Theta \vdash \Theta' \rightsquigarrow \langle \{\overline{c_{D,s}:\vartheta_s}, \overline{c_{D,a}:\vartheta_a}, \overline{c_d : \forall \overline{X_i}.\overline{X_d}.u_d^t \sim \forall \overline{X_i}.\overline{X_d}.X_d}\},$
  $\{\overline{x_{D,s}:\forall \overline{X_p}.\mathsf{fn}(S_D\,\langle \overline{X_p}\rangle) \to \tau_s^t}, \overline{x_{D,a}:\forall \overline{X_p}.\mathsf{fn}(S_D\,\langle \overline{X_p}\rangle) \to \tau_a^t},$
  $x_d : \forall \overline{X_i}.\overline{X_d}.\forall \overline{\omega_{dm}}.\mathsf{fn}(\overline{\tau_{dm}^t}) \to \tau_d^t\}\rangle$
  as required.

□

Theorem B.2 (Translation of items preserves well-typedness)**.** *Suppose* $\Gamma' \subseteq \Gamma$ *and* $\Theta' \subseteq \Theta$.
If $\Gamma \mid \Theta \vdash \mathit{item} : \Gamma' \mid \Theta' \rightsquigarrow \overline{\mathit{item}_i^t}$ *and* $\langle \Gamma, \Theta\rangle \rightsquigarrow \langle \Omega^t, \Gamma^t\rangle$
then there are some $\Omega_1^t, \Omega_2^t, \Gamma_1^t, \Gamma_2^t, \overline{\Omega_i^t}, \overline{\Gamma_i^t}$ *such that :*

- $\overline{\Gamma \mid \Theta \vdash \Gamma' \rightsquigarrow \langle \Omega_1^t, \Gamma_1^t\rangle}$,
- $\overline{\Gamma \mid \Theta \vdash \Theta' \rightsquigarrow \langle \Omega_2^t, \Omega_2^t\rangle}$,
- $\overline{\Omega^t \mid \Gamma^t \vdash_T \mathit{item}_i^t : \Omega_i^t \mid \Gamma_i^t}$,
- $\overline{\Omega_i^t} = \Omega_1^t, \Omega_2^t$ *and*

- $\overline{\Gamma_i^t} = \Gamma_1^t, \Gamma_2^t$.

Proof. By induction on derivations $\Gamma \mid \Theta \vdash item : \Gamma' \mid \Theta' \rightsquigarrow \overline{item_i^t}$.

- **Case** (struct):
  $\Gamma \mid \Theta \vdash \mathsf{struct}\ S\ \langle\overline{X_j}\rangle\ \{\overline{x_i : \tau_i}\} : [S\ \langle\overline{X_j}\rangle\ \{\overline{x_i : \tau_i}\}] \mid \emptyset \rightsquigarrow \mathsf{struct}\ S\ \langle\overline{X_j}\rangle\ \{\overline{x_i : \tau_i^t}\}$
  where $\overline{\Gamma, \overline{X_j} \mid \Theta \vdash_{\mathrm{WF}} \tau_i \rightsquigarrow \tau_i^t}^{\,i}$.

  Suppose $\langle \Gamma, \Theta \rangle \rightsquigarrow \langle \Omega^t, \Gamma^t \rangle$.

  Then, using the environment translation judgment we get
  $\{\Omega_1^t, \Omega_2^t\} = [S : S\ \langle\overline{X_j}\rangle\ \{\overline{x_i : \tau_i^t}\}]$ and $\{\Gamma_1^t, \Gamma_2^t\} = \emptyset$.

  By rule (t-struct) $\Omega^t \mid \Gamma^t \vdash \mathsf{struct}\ S\ \langle\overline{X_j}\rangle\ \{\overline{x_i : \tau_i^t}\} : [S\ \langle\overline{X_j}\rangle\ \{\overline{x_i : \tau_i^t}\}] \mid \emptyset$ as required.

- **Case** (fun): $\Gamma \mid \Theta \vdash \mathsf{fn}\ f\ \langle\overline{X_k}\rangle\ (\overline{x_i : \tau_i}) \to \tau\ \mathsf{where}\ \overline{\pi_j}\ \{\, e \,\} : [f : \sigma] \mid \emptyset$
  $\rightsquigarrow f : \sigma^t = \Lambda\overline{X_k}.\Lambda\overline{c_j : \omega_j}.\mathsf{fn}\ (\overline{x_j : \tau_j^t})\{\, \mathsf{fn}\ (\overline{x_i : \tau_i^t})\{\, e^t \,\}\,\}$ where
  $\sigma = \forall\overline{X_k}.\overline{\pi_j} \Rightarrow \mathsf{fn}(\overline{\tau_i}) \to \tau$,
  $\Gamma \mid \Theta \vdash_{\mathrm{WF}} \sigma \rightsquigarrow \sigma^t$,
  $\sigma^t = \forall\overline{X_k}.\forall\overline{\omega_j}.\mathsf{fn}(\overline{\tau_j^t}) \to \mathsf{fn}(\overline{\tau_i^t}) \to \tau^t$,
  $\Gamma'' = \Gamma, \overline{X_k}, \overline{x_i : \tau_i}$,
  $\Theta'' = \Theta, \overline{\pi_j \rightsquigarrow \langle x_j, c_j \rangle}$ and
  $\Gamma'' \mid \Theta'' \vdash e : \tau \rightsquigarrow e^t$

  Suppose $\langle \Gamma, \Theta \rangle \rightsquigarrow \langle \Omega^t, \Gamma^t \rangle$.

  From the environment translation and from $\Gamma \mid \Theta \vdash_{\mathrm{WF}} \sigma \rightsquigarrow \sigma^t$ we get
  $\{\Omega_1^t, \Omega_2^t\} = \emptyset$ and $\{\Gamma_1^t, \Gamma_2^t\} = [f : \sigma^t]$.

  From lemma B.3 we get
  $\overline{\Gamma, \overline{X_k} \mid \Theta \vdash_{\mathrm{WF}} \pi_j \rightsquigarrow \langle \tau_j^t, \omega_j \rangle}^{\,j}$,
  $\Gamma, \overline{X_k} \mid \Theta, \overline{\pi_j \rightsquigarrow \langle x_j, c_j \rangle} \vdash_{\mathrm{WF}} \mathsf{fn}(\overline{\tau_i}) \to \tau \rightsquigarrow \mathsf{fn}(\overline{\tau_i^t}) \to \tau^t$,
  $\Gamma, \overline{X_k} \mid \Theta, \overline{\pi_j \rightsquigarrow \langle x_j, c_j \rangle} \vdash_{\mathrm{WF}} \tau_i \rightsquigarrow \tau_i^t$ and
  $\Gamma, \overline{X_k} \mid \Theta, \overline{\pi_j \rightsquigarrow \langle x_j, c_j \rangle} \vdash_{\mathrm{WF}} \tau \rightsquigarrow \tau^t$.

  Using lemmas B.10 and B.11 we get
  $\overline{\Gamma'' \mid \Theta'' \vdash_{\mathrm{WF}} \pi_j \rightsquigarrow \langle \tau_j^t, \omega_j \rangle}^{\,j}$,
  $\Gamma'' \mid \Theta'' \vdash_{\mathrm{WF}} \tau_i \rightsquigarrow \tau_i^t$ and
  $\Gamma'' \mid \Theta'' \vdash_{\mathrm{WF}} \tau \rightsquigarrow \tau^t$.

  From the environment translation judgment we get
  $\langle \Gamma'' \mid \Theta'' \rangle \rightsquigarrow \langle (\Omega^t, \overline{c_j : \omega_j}), (\Gamma^t, \overline{X_k}, \overline{x_i : \tau_i^t}, \overline{x_j : \tau_j^t}) \rangle$.

  From theorem B.1 we get $\Omega^t, \overline{c_j : \omega_j} \mid \Gamma^t, \overline{X_k}, \overline{x_i : \tau_i^t}, \overline{x_j : \tau_j^t} \vdash e^t : \tau^t$.

We can then obtain the following derivation:

$$\text{(t-tabs)}\ \frac{\text{(t-cabs)}\ \frac{\text{(t-fabs)}\ \frac{\text{(t-fabs)}\ \frac{\Omega^t, \overline{c_j : \omega_j} \mid \Gamma^t, \overline{X_k}, \overline{x_j : \tau_j^t}, \overline{x_i : \tau_i^t} \mid \emptyset \vdash e^t : \tau^t}{\Omega^t, \overline{c_j : \omega_j} \mid \Gamma^t, \overline{X_k}, \overline{x_j : \tau_j^t} \mid \emptyset \vdash \mathsf{fn}\ (\overline{x_i : \tau_i^t})\{\, e^t \,\} : \mathsf{fn}(\overline{\tau_i^t}) \to \tau^t}}{\Omega^t, \overline{c_j : \omega_j} \mid \Gamma^t, \overline{X_k} \mid \emptyset \vdash \mathsf{fn}\ (\overline{x_j : \tau_j^t})\{\, \mathsf{fn}\ (\overline{x_i : \tau_i^t})\{\, e^t \,\} \,\} : \mathsf{fn}(\overline{\tau_j^t}) \to \mathsf{fn}(\overline{\tau_i^t}) \to \tau^t}}{\Omega^t \mid \Gamma^t, \overline{X_k} \mid \emptyset \vdash \Lambda \overline{c_j : \omega_j}.\mathsf{fn}\ (\overline{x_j : \tau_j^t})\{\, \mathsf{fn}\ (\overline{x_i : \tau_i^t})\{\, e^t \,\} \,\} : \forall \overline{\omega_j}.\mathsf{fn}(\overline{\tau_j^t}) \to \mathsf{fn}(\overline{\tau_i^t}) \to \tau^t}}{\Omega^t \mid \Gamma^t \mid \emptyset \vdash \Lambda \overline{X_k}.\Lambda \overline{c_j : \omega_j}.\mathsf{fn}\ (\overline{x_j : \tau_j^t})\{\, \mathsf{fn}\ (\overline{x_i : \tau_i^t})\{\, e^t \,\} \,\} : \forall \overline{X_k}.\forall \overline{\omega_j}.\mathsf{fn}(\overline{\tau_j^t}) \to \mathsf{fn}(\overline{\tau_i^t}) \to \tau^t}$$

We can then apply rule (t-decl) to get
$\Omega^t \mid \Gamma^t \vdash f : \sigma^t = \Lambda \overline{X_k}.\Lambda \overline{c_j : \omega_j}.\mathsf{fn}\ (\overline{x_j : \tau_j^t})\{\, \mathsf{fn}\ (\overline{x_i : \tau_i^t})\{\, e^t \,\} \,\} : \emptyset \mid [f : \sigma^t]$ as required.

- **Case** (trait):

  $item = \mathsf{trait}\ D\ \langle \overline{X_i} \rangle\ \mathsf{where}\ \overline{\pi_j}\ \{\mathsf{type}\ A : \overline{\beta_a};\ f : \sigma;\}$

  $\Gamma' = [(D\ \langle \overline{X_i} \rangle\ \mathsf{where}\ \overline{\pi_h},\ Self : \overline{\beta_s \rightsquigarrow x_s},\ A : \overline{\beta_a \rightsquigarrow x_a},\ f, \mathsf{obj\text{-}unsafe}),\ f : \sigma']$

  $\Theta' = [\overline{\forall \overline{X_p}.(Self : D\ \langle \overline{X_i} \rangle) \Rightarrow Self : \beta_s \rightsquigarrow \langle x_{D,s}, c_{D,s} \rangle}^{s},$
  $\overline{\forall \overline{X_p}.(Self : D\ \langle \overline{X_i} \rangle) \Rightarrow A_D\ \langle \overline{X_p} \rangle : \beta_a \rightsquigarrow \langle x_{D,a}, c_{D,a} \rangle}^{a}\ ]$

  $\overline{item_i^t} =$
  $\quad \mathsf{type}\ A_D\ \langle \overline{X_p} \rangle;$
  $\quad \overline{\mathsf{axiom}\ c_{D,a} : \vartheta_a}^{a};$
  $\quad \overline{\mathsf{axiom}\ c_{D,s} : \vartheta_s}^{s};$
  $\quad \mathsf{struct}\ S_D\ \langle \overline{X_p} \rangle\ \{f : \sigma^t, \overline{x_s : \tau_s^t}, \overline{x_a : \tau_a^t}\}$
  $\quad \overline{f : \sigma'^t = \Lambda \overline{X_p}, \overline{X_k}.\Lambda \overline{c_l : \omega_l}.\mathsf{fn}\ (x_D : S_D\ \langle \overline{X_p} \rangle, \overline{x_l : \tau_l^t})\{\, (x_D.f)[\overline{X_k}][\![\overline{c_l}]\!](\overline{x_l}) \,\}};$
  $\quad \overline{x_{D,s} : \forall \overline{X_p}.\mathsf{fn}(S_D\ \langle \overline{X_p} \rangle) \to \tau_s^t = \Lambda \overline{X_p}.\mathsf{fn}\ (x_D : S_D\ \langle \overline{X_p} \rangle)\{x_D.x_s\}}^{s};$
  $\quad \overline{x_{D,a} : \forall \overline{X_p}.\mathsf{fn}(S_D\ \langle \overline{X_p} \rangle) \to \tau_a^t = \Lambda \overline{X_p}.\mathsf{fn}\ (x_D : S_D\ \langle \overline{X_p} \rangle)\{x_D.x_a\}}^{a};$

  Side conditions:

  $\{\overline{\pi_j}\} \equiv \{(Self : \overline{\beta_s}), \overline{\pi_h}\}$

  $\overline{X_p} = Self, \overline{X_i}$

  $\overline{\Gamma \mid \Theta \vdash_{\mathrm{WF}} \forall \overline{X_p}.(Self : D\ \langle \overline{X_i} \rangle) \Rightarrow \pi_h \rightsquigarrow \_}^{h}$

  $\sigma = \forall \overline{X_k}.\overline{\pi_l} \Rightarrow \mathsf{fn}(\overline{\tau_m}) \to \tau_R$

  $\sigma' = \forall \overline{X_p}, \overline{X_k}.(Self : D\ \langle \overline{X_i} \rangle, \overline{\pi_l}) \Rightarrow \mathsf{fn}(\overline{\tau_m}) \to \tau_R$

  $\Gamma \mid \Theta \vdash_{\mathrm{WF}} \sigma' \rightsquigarrow \sigma'^t$

  $\Gamma, \overline{X_p} \mid \Theta, Self : D\ \langle \overline{X_i} \rangle \vdash_{\mathrm{WF}} \sigma \rightsquigarrow \sigma^t$

  $\sigma^t = \forall \overline{X_k}.\forall \overline{\omega_l}.\mathsf{fn}(\overline{\tau_l^t}) \to \mathsf{fn}(\overline{\tau_m^t}) \to \tau_R^t$

  $$\frac{\overline{\Gamma \mid \Theta \vdash_{\mathrm{WF}} \forall \overline{X_p}.(Self : D\ \langle \overline{X_i} \rangle) \Rightarrow Self : \beta_s \rightsquigarrow \langle \forall \overline{X_p}.\mathsf{fn}(S_D\ \langle \overline{X_p} \rangle) \to \tau_s^t, \vartheta_s \rangle}^{s}}{\overline{\Gamma \mid \Theta \vdash_{\mathrm{WF}} \forall \overline{X_p}.(Self : D\ \langle \overline{X_i} \rangle) \Rightarrow A_D\ \langle \overline{X_p} \rangle : \beta_a \rightsquigarrow \langle \forall \overline{X_p}.\mathsf{fn}(S_D\ \langle \overline{X_p} \rangle) \to \tau_a^t, \vartheta_a \rangle}^{a}}$$

  Suppose $\langle \Gamma, \Theta \rangle \rightsquigarrow \langle \Omega^t, \Gamma^t \rangle$.

  Using the rule's side conditions and the environment translation rules we get

$\Gamma_1^t = [f : \sigma'^t]$,
$\Omega_1^t = [A_D\ \langle \overline{X_p} \rangle\ ,\ S\ \langle \overline{X_p} \rangle\ \{f : \sigma^t, \overline{x_s : \tau_s^t}, \overline{x_a : \tau_a^t}\}]$,
$\Gamma_2^t = [\overline{x_{D,s} : \forall \overline{X_p}.\mathsf{fn}(S_D\ \langle \overline{X_p} \rangle) \to \tau_s^t}^{\,s}\ , \overline{x_{D,a} : \forall \overline{X_p}.\mathsf{fn}(S_D\ \langle \overline{X_p} \rangle) \to \tau_a^t}^{\,a}\ ]$ and
$\Omega_2^t = [\overline{c_{D,s} : \vartheta_s}, \overline{c_{D,a} : \vartheta_a}]$.

It follows that $\{\Omega_1^t, \Omega_2^t\} \subseteq \Omega^t$ and $\{\Gamma_1^t, \Gamma_2^t\} \subseteq \Gamma^t$.

Then by (t-type) $\Omega^t \mid \Gamma^t \vdash \mathsf{type}\ A_D\ \langle \overline{X_p} \rangle : [A_D\ \langle \overline{X_p} \rangle] \mid \emptyset$.

From lemma B.3 we get $\overline{\vartheta_s = \forall \overline{X_p}.A_{Ds}\ \langle u_s, \overline{u_{ns}} \rangle \sim \forall \overline{X_p}.u_{As}}$,
$\overline{(D_s\ \langle \overline{X_{ns}} \rangle \mathsf{\ where\ } \_, X_s : \_, A : \_, \_, \_)}^{\,s} \in \Gamma$ and
$[\overline{u_s/X_s}][\overline{\overline{u_{sn}}/X_{sn}}]$ defined
for some $\overline{A_{Ds}}, \overline{u_s}, \overline{\overline{u_{sn}}}$ and $\overline{A_{Ds}\ \langle \overline{X_{sn}} \rangle}$. Then by lemma B.1, $\overline{A_{Ds}\ \langle X_s, \overline{X_{sn}} \rangle} \in \Omega^t$.
Then by (t-type) $\overline{\Omega^t \mid \Gamma^t \vdash \mathsf{axiom}\ c_{D,s} : \vartheta_s : [c_{D,s} : \vartheta_s] \mid \emptyset}^{\,s}$.
Following similar reasoning we get $\overline{\Omega^t \mid \Gamma^t \vdash \mathsf{axiom}\ c_{D,a} : \vartheta_a : [c_{D,a} : \vartheta_a] \mid \emptyset}^{\,a}$.
By (t-struct),
$\Omega^t \mid \Gamma^t \vdash \mathsf{struct}\ S\ \langle \overline{X_p} \rangle\ \{f : \sigma^t, \overline{x_s : \tau_s^t}, \overline{x_a : \tau_a^t}\} : [S\ \langle \overline{X_p} \rangle\ \{f : \sigma^t, \overline{x_s : \tau_s^t}, \overline{x_a : \tau_a^t}\}] \mid \emptyset$.

Using the well-formedness judgments we get
$\Gamma \mid \Theta \vdash \forall \overline{X_p}, \overline{X_k}.(Self : D\ \langle \overline{X_i} \rangle\ , \overline{\pi_l}) \Rightarrow \mathsf{fn}(\overline{\tau_m}) \to \tau_R \rightsquigarrow$
$\forall \overline{X_p}, \overline{X_k}.\forall \overline{\omega_l}.\mathsf{fn}(S_D\ \langle \overline{X_p} \rangle\ , \overline{\tau_l^t}) \to \mathsf{fn}(\overline{\tau_m^t}) \to \tau_R^t$
so $\sigma'^t = \forall \overline{X_p}, \overline{X_k}.\forall \overline{\omega_l}.\mathsf{fn}(S_D\ \langle \overline{X_p} \rangle\ , \overline{\tau_l^t}) \to \mathsf{fn}(\overline{\tau_m^t}) \to \tau_R^t$.

Using the target language typing rules (and combining some steps for conciseness) we get the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Omega_1^t \mid \Gamma_1^t \mid \emptyset \vdash x_D.f : \forall \overline{X_k}.\forall \overline{\omega_l}.\mathsf{fn}(\overline{\tau_l^t}) \to \mathsf{fn}(\overline{\tau_m^t}) \to \tau_R^t}{\Omega_1^t \mid \Gamma_1^t \mid \emptyset \vdash (x_D.f)[\overline{X_k}][\![\overline{c_l}]\!] : \mathsf{fn}(\overline{\tau_l^t}) \to \mathsf{fn}(\overline{\tau_m^t}) \to \tau_R^t} \quad \overline{\Omega_1^t \mid \Gamma_1^t \mid \emptyset \vdash x_l : \tau_l^t}^{\,l}}{\Omega_1^t \mid \Gamma_1^t \mid \emptyset \vdash (x_D.f)[\overline{X_k}][\![\overline{c_l}]\!](\overline{x_l}) : \mathsf{fn}(\overline{\tau_m^t}) \to \tau_R^t}}{\Omega_1^t \mid \Gamma^t, \overline{X_p}, \overline{X_k} \mid \emptyset \vdash \mathsf{fn}\ (x_D : S_D\ \langle \overline{X_p} \rangle\ , \overline{x_l : \tau_l^t})\{(x_D.f)[\overline{X_k}][\![\overline{c_l}]\!](\overline{x_l})\} : \mathsf{fn}(S_D\ \langle \overline{X_p} \rangle\ , \overline{\tau_l^t}) \to \mathsf{fn}(\overline{\tau_m^t}) \to \tau_R^t}}{\begin{array}{l}\Omega^t \mid \Gamma^t \mid \emptyset \vdash \Lambda \overline{X_p}, \overline{X_k}.\Lambda \overline{c_l : \omega_l}.\mathsf{fn}\ (x_D : S_D\ \langle \overline{X_p} \rangle\ , \overline{x_l : \tau_l^t})\{(x_D.f)[\overline{X_k}][\![\overline{c_l}]\!](\overline{x_l})\} \\ : \forall \overline{X_p}, \overline{X_k}.\forall \overline{\omega_l}.\mathsf{fn}(S_D\ \langle \overline{X_p} \rangle\ , \overline{\tau_l^t}) \to \mathsf{fn}(\overline{\tau_m^t}) \to \tau_R^t\end{array}}$$

where $\Omega_1^t = \Omega^t, \overline{c_l : \omega_l}$ and $\Gamma_1^t = \Gamma^t, \overline{X_p}, \overline{X_k}, x_D : S_D\ \langle \overline{X_p} \rangle\ , \overline{x_l : \tau_l^t}$.

We can then use (t-decl) to get
$\Omega^t \mid \Gamma^t \vdash f : \sigma'^t = \Lambda \overline{X_p}, \overline{X_k}.\Lambda \overline{c_l : \omega_l}.\mathsf{fn}\ (x_D : S_D\ \langle \overline{X_p} \rangle\ , \overline{x_l : \tau_l^t})\{(x_D.f)[\overline{X_k}][\![\overline{c_l}]\!](\overline{x_l})\}$
$: \emptyset \mid [f : \sigma'^t]$.

For each $x_{D,s}$ we get the following derivation:

$$\frac{\frac{\Omega^t \mid \Gamma^t, \overline{X_p}, (x_D : S_D\ \langle \overline{X_p} \rangle) \mid \emptyset \vdash x_D.x_s : \tau_s^t}{\Omega^t \mid \Gamma^t, \overline{X_p} \mid \emptyset \vdash \mathsf{fn}\ (x_D : S_D\ \langle \overline{X_p} \rangle)\{x_D.x_s\} : \mathsf{fn}(S_D\ \langle \overline{X_p} \rangle) \to \tau_s^t}}{\Omega^t \mid \Gamma^t \mid \emptyset \vdash \Lambda \overline{X_p}.\mathsf{fn}\ (x_D : S_D\ \langle \overline{X_p} \rangle)\{x_D.x_s\} : \forall \overline{X_p}.\mathsf{fn}(S_D\ \langle \overline{X_p} \rangle) \to \tau_s^t}$$

Using (t-decl) we then get

$$\frac{\Omega^t \mid \Gamma^t \vdash x_{D,s} : \forall \overline{X_p}.\mathsf{fn}(S_D\ \langle \overline{X_p} \rangle) \to \tau_s^t = \Lambda \overline{X_p}.\mathsf{fn}\ (x_D : S_D\ \langle \overline{X_p} \rangle)\{x_D.x_s\}}{: \emptyset, [x_{D,s} : \forall \overline{X_p}.\mathsf{fn}(S_D\ \langle \overline{X_p} \rangle) \to \tau_s^t]}\ s$$

Similarly, we get

$$\frac{\Omega^t \mid \Gamma^t \vdash x_{D,a} : \forall \overline{X_p}.\mathsf{fn}(S_D\ \langle \overline{X_p} \rangle) \to \tau_a^t = \Lambda \overline{X_p}.\mathsf{fn}\ (x_D : S_D\ \langle \overline{X_p} \rangle)\{x_D.x_a\}}{: \emptyset, [x_{D,a} : \forall \overline{X_p}.\mathsf{fn}(S_D\ \langle \overline{X_p} \rangle) \to \tau_a^t]}\ a$$

We then have $\overline{\Gamma_i^t} = \{\Gamma_1^t, \Gamma_2^t\}$ and $\overline{\Omega_i^t} = \{\Omega_1^t, \Omega_2^t\}$ as required.

- **Case** (impl):

  $item = \mathsf{impl}\ \langle \overline{X_k} \rangle\ D\ \langle \overline{u_i} \rangle\ \mathsf{for}\ u\ \mathsf{where}\ \overline{\pi_j}\ \{\, \mathsf{type}\ A = u_A; fun \,\}$

  $\Gamma' = \emptyset$

  $\Theta' = [\forall \overline{X_k}.\overline{\pi_j} \Rightarrow u : D\ \langle \overline{u_i}, A \sim u_A \rangle \rightsquigarrow \langle x_{u:D\ \langle \overline{u_i} \rangle}, c_{u:D\ \langle \overline{u_i} \rangle} \rangle]$

  $\overline{item_i^t} = \begin{array}{l} \mathsf{axiom}\ c_{u:D\ \langle \overline{u_i} \rangle} : \forall \overline{X_k}.A_D\ \langle \overline{u_p^t} \rangle \sim \forall \overline{X_k}.u_A^t \\ x_{u:D\ \langle \overline{u_i} \rangle} : \sigma_D^t = \Lambda \overline{X_k}.\Lambda \overline{c_j : \omega_j}.\mathsf{fn}\ (\overline{x_j : \tau_j^t})\,\{\, S_D\ \langle \overline{u_p^t} \rangle\,\{\, f : e_F^t, \overline{x_s : e_s^t}, \overline{x_a : e_a^t}\,\}\}; \end{array}$

  Side conditions:

  $(D\ \langle \overline{X_i} \rangle\ \mathsf{where}\ \overline{\pi_h}, Self : \overline{\beta_s \rightsquigarrow x_s}^{\,s}, A : \overline{\beta_a \rightsquigarrow x_a}^{\,a}, f, \_) \in \Gamma$

  $\overline{u_p} = u, \overline{u_i}$

  $\overline{X_p} = Self, \overline{X_i}$

  $\Theta^* = \Theta \setminus \{\overline{\forall \overline{X_p}.(Self : D\ \langle \overline{X_i} \rangle) \Rightarrow Self : \beta_s \rightsquigarrow \_}^{\,s}, \overline{\forall \overline{X_p}.(Self : D\ \langle \overline{X_i} \rangle) \Rightarrow A_D\ \langle \overline{X_p} \rangle : \beta_a \rightsquigarrow \_}^{\,a}\}$

  $$\frac{\frac{\Gamma, \overline{X_k} \mid \Theta^*, \overline{\pi_j \rightsquigarrow \langle x_j, c_j \rangle} \Vdash [\overline{u_p/X_p}](Self : \beta_s) \rightsquigarrow \langle e_s^t, \gamma_s \rangle}{\Gamma, \overline{X_k} \mid \Theta^*, \overline{\pi_j \rightsquigarrow \langle x_j, c_j \rangle} \Vdash [\overline{u_p/X_p}]\pi_h \rightsquigarrow \_}\ h \quad \Gamma, \overline{X_k} \mid \Theta^*, \overline{\pi_j \rightsquigarrow \langle x_j, c_j \rangle} \vdash_{\mathrm{WF}} u_A \rightsquigarrow u_A^t}{\Gamma, \overline{X_k} \mid \Theta^*, \overline{\pi_j \rightsquigarrow \langle x_j, c_j \rangle} \Vdash (A_D\ \langle \overline{u_p} \rangle : \overline{[u_p/X_p]\beta_a}) \rightsquigarrow \langle e_a^t, \gamma_a \rangle}\ a$$

  $(f : \forall \overline{X_p}.(Self : D\ \langle \overline{X_i} \rangle) \Rightarrow \sigma) \in \Gamma$

  $\Gamma, \overline{X_k} \mid \Theta^*, \overline{\pi_j \rightsquigarrow \langle x_j, c_j \rangle} \vdash fun : [f : \sigma'] \mid \emptyset \rightsquigarrow f : \sigma'^t = e_F^t$

  $\sigma' = [\overline{u_p/X_p}]\sigma$

  $\Gamma \mid \Theta \vdash_{\mathrm{WF}} \forall \overline{X_k}.\overline{\pi_j} \Rightarrow u : D\ \langle \overline{u_i} \rangle \rightsquigarrow \sigma_D^t$

  $\sigma_D^t = \forall \overline{X_k}.\forall \overline{\omega_j}.\mathsf{fn}(\overline{\tau_j^t}) \to S_D\ \langle \overline{u_p^t} \rangle$

  Using lemma B.3 we get

  $\overline{\Gamma, \overline{X_k} \mid \Theta, \overline{\pi_j \rightsquigarrow \_} \vdash_{\mathrm{WF}} \pi_j \rightsquigarrow \langle \tau_j^t, \omega_j \rangle}^{\,j}$,

  $\Gamma, \overline{X_k} \mid \Theta, \overline{\pi_j \rightsquigarrow \_} \vdash_{\mathrm{WF}} u : D\ \langle \overline{u_i} \rangle \rightsquigarrow S_D\ \langle \overline{u_p^t} \rangle$ and

  $\overline{\Gamma, \overline{X_k} \mid \Theta, \overline{\pi_j \rightsquigarrow \_} \vdash_{\mathrm{WF}} u_p \rightsquigarrow u_p^t}$.

Using lemma B.10 we get
$\Gamma, \overline{X_k} \mid \Theta, \overline{\pi_j \rightsquigarrow \_} \vdash_{\text{WF}} u_A \rightsquigarrow u_A^t$.

Using (wf-atype) we get
$\Gamma, \overline{X_k} \mid \Theta, \overline{\pi_j \rightsquigarrow \_} \vdash_{\text{WF}} A_D \langle \overline{u_p} \rangle \rightsquigarrow A_D \langle \overline{u_p^t} \rangle$.

Using (wf-eqcons), lemma B.10 and (wf-cocons) we get
$\Gamma, \overline{X_k} \mid \Theta, \overline{\pi_j \rightsquigarrow \_} \vdash_{\text{WF}} u : D \langle \overline{u_i}, A \sim u_A \rangle \rightsquigarrow \langle S_D \langle \overline{u_p^t} \rangle, A_D \langle \overline{u_p^t} \rangle \sim u_A^t \rangle$.

Then we can use (wf-cscheme) to get
$\Gamma \mid \Theta \vdash_{\text{WF}} \forall \overline{X_k}.\overline{\pi_j} \Rightarrow u : D \langle \overline{u_i}, A \sim u_A \rangle \rightsquigarrow \langle \sigma_D^t, \forall \overline{X_k}.A_D \langle \overline{u_p^t} \rangle \sim \forall \overline{X_k}.u_A^t \rangle$

Then using the environment translation rules we get
$\Omega_1^t = \emptyset$,
$\Gamma_1^t = \emptyset$,
$\Omega_2^t = [c_{u:D\langle \overline{u_i} \rangle} : \forall \overline{X_k}.A_D \langle \overline{u_p^t} \rangle \sim \forall \overline{X_k}.u_A^t]$,
$\Gamma_2^t = [x_{u:D\langle \overline{u_i} \rangle} : \sigma_D^t]$.

Using (t-axiom) we get
$\Omega^t \mid \Gamma^t \vdash \text{axiom } c_{u:D\langle \overline{u_i} \rangle} : \forall \overline{X_k}.A_D \langle \overline{u_p^t} \rangle \sim \forall \overline{X_k}.u_A^t : [c_{u:D\langle \overline{u_i} \rangle} : \forall \overline{X_k}.A_D \langle \overline{u_p^t} \rangle \sim \forall \overline{X_k}.u_A^t] \mid \emptyset$.

Using lemma B.20 we get $\langle \Gamma, \overline{X_k} \mid \Theta, \overline{\pi_j \rightsquigarrow \langle x_j, c_j \rangle} \rangle \rightsquigarrow \langle \Omega^t, \overline{c_j : \omega_j} \mid \Gamma^t, \overline{X_k}, \overline{x_j : \tau_j^t} \rangle$.

From lemma B.1 we get
$$\frac{}{\Gamma, \overline{X_p} \mid \Theta, \mathit{Self} : D \langle \overline{X_i} \rangle \rightsquigarrow \_ \vdash_{\text{WF}} \mathit{Self} : \beta_s \rightsquigarrow \langle \tau_s^t, \omega_s \rangle}\, s$$
$$\frac{\Gamma, \overline{X_p} \mid \Theta, \mathit{Self} : D \langle \overline{X_i} \rangle \rightsquigarrow \_ \vdash_{\text{WF}} \mathit{Self} : \beta_s \rightsquigarrow \langle \tau_s^t, \omega_s \rangle}{\Gamma, \overline{X_p} \mid \Theta, \mathit{Self} : D \langle \overline{X_i} \rangle \rightsquigarrow \_ \vdash_{\text{WF}} A_D \langle \overline{X_p} \rangle : \beta_a \rightsquigarrow \langle \tau_a^t, \omega_a \rangle}\, a,$$
$(f : \forall \overline{X_p}, \overline{X_n}.(\mathit{Self} : D \langle \overline{X_i} \rangle, \overline{\pi_l}) \Rightarrow \tau_F) \in \Gamma$,
$\Gamma, \overline{X_p} \mid \Theta, \mathit{Self} : D \langle \overline{X_i} \rangle \rightsquigarrow \_ \vdash_{\text{WF}} \forall \overline{X_n}.\overline{\pi_l} \Rightarrow \tau_F \rightsquigarrow \sigma^t$ and
$S_D \langle \overline{X_p} \rangle \{f : \sigma^t, \overline{x_s : \tau_s^t}, \overline{x_a : \tau_a^t}\} \in \Omega^t$
for some $\overline{\tau_s^t}, \overline{\omega_s}, \overline{\tau_a^t}, \overline{\omega_a}, \overline{X_n}, \overline{\pi_l}, \tau_F, \sigma^t$.

Then $\sigma = \forall \overline{X_n}.\overline{\pi_l} \Rightarrow \tau_F$ and $\sigma' = \forall \overline{X_n}.\overline{\pi_l'} \rightarrow \tau_F'$
where $\overline{\pi_l' = [\overline{u_p/X_p}]\pi_l}$ and $\tau_F' = [\overline{u_p/X_p}]\tau_F'^t$.

From lemma B.3 we get $\sigma^t = \forall \overline{X_n}.\forall \overline{\omega_l}.\mathsf{fn}(\overline{\tau_l^t}) \rightarrow \tau_F^t$ for some $\overline{\omega_l}, \overline{\tau_l^t}$ and τ_F^t.

(fun) is the only rule that could have been used as the last rule in the derivation
$\Gamma, \overline{X_k} \mid \Theta^*, \overline{\pi_j \rightsquigarrow \langle x_j, c_j \rangle} \vdash \mathit{fun} : [f : \sigma'] \mid \emptyset \rightsquigarrow f : \sigma'^t = e_F^t$.
Then from the rule's side conditions we get:
$\sigma' = \forall \overline{X_n}.\overline{\pi_l'} \Rightarrow \mathsf{fn}(\overline{\tau_m'}) \rightarrow \tau_R'$,
$\Gamma, \overline{X_k} \mid \Theta^*, \overline{\pi_j \rightsquigarrow \langle x_j, c_j \rangle} \vdash_{\text{WF}} \sigma' \rightsquigarrow \sigma'^t$,
$\sigma'^t = \forall \overline{X_n}.\forall \overline{\omega_l'}.\mathsf{fn}(\overline{\tau_l'^t}) \rightarrow \mathsf{fn}(\overline{\tau_m'^t}) \rightarrow \tau_R'^t$,
$\Gamma'' = \Gamma, \overline{X_k}, \overline{X_n}, \overline{x_m : \tau_m'}$,

$\Theta'' = \Theta^*, \overline{\pi_j \rightsquigarrow \langle x_j, c_j \rangle}, \overline{\pi_l \rightsquigarrow \langle x_l, c_l \rangle}$ and
$\Gamma'' \mid \Theta'' \vdash e : \tau_R \rightsquigarrow e^t$ and
$e^t_F = \Lambda \overline{X_n}.\Lambda \overline{c_l : \omega'_l}.\mathsf{fn}\ (\overline{x_l : \tau'^t_l})\{\ \mathsf{fn}\ (\overline{x_m : \tau'^t_m})\{\ e^t\ \}\}$.

Using lemma B.12 we get $\Gamma, \overline{X_k} \mid \Theta, \overline{\pi_j \rightsquigarrow \langle x_j, c_j \rangle} \vdash_{\mathrm{WF}} \sigma' \rightsquigarrow \sigma'^t$.

Let $\Theta''' = \Theta, \overline{\pi_j \rightsquigarrow \langle x_j, c_j \rangle}, \overline{\pi'_l \rightsquigarrow \langle x_l, c_l \rangle}$.

Using lemma B.13 we get $\Gamma'' \mid \Theta''' \vdash e : \tau'_R \rightsquigarrow e^t$.

Then by (fun) $\Gamma, \overline{X_k} \mid \Theta, \overline{\pi_j \rightsquigarrow \langle x_j, c_j \rangle} \vdash fun : [f : \sigma'] \mid \emptyset \rightsquigarrow f : \sigma'^t = e^t_F$.

Now we want to show that $\sigma'^t = [\overline{u^t_p / X_p}]\sigma^t$.

By lemma B.12 we get
$\Gamma, \overline{X_k}, \overline{X_p} \mid \Theta, \overline{\pi_j \rightsquigarrow \langle x_j, c_j \rangle}, Self : D\ \langle \overline{X_i} \rangle \rightsquigarrow \_ \vdash_{\mathrm{WF}} \sigma \rightsquigarrow \sigma^t$.

From lemma B.10 we get $\overline{\Gamma, \overline{X_k} \mid \Theta, \overline{\pi_j \rightsquigarrow \langle x_j, c_j \rangle}, u : D\ \langle \overline{u_i} \rangle \rightsquigarrow \_ \vdash_{\mathrm{WF}} u_p \rightsquigarrow u^t_p}$.

By lemma B.9 we get
$\Gamma, \overline{X_k} \mid \Theta, \overline{\pi_j \rightsquigarrow \langle x_j, c_j \rangle}, u : D\ \langle \overline{u_i} \rangle \rightsquigarrow \_ \vdash_{\mathrm{WF}} [\overline{u_p / X_p}]\sigma \rightsquigarrow [\overline{u_p / X_p}]\sigma^t$.

Since $(\forall \overline{X_k}.\overline{\pi_j} \Rightarrow u : D\ \langle \overline{u_i}, A \sim u_A \rangle \rightsquigarrow \langle x_{u:D\langle \overline{u_i} \rangle}, c_{u:D\langle \overline{u_i} \rangle} \rangle) \in \Theta$,
we can apply (c-ext) and (c-sep) to get
$\Gamma, \overline{X_k} \mid \Theta, \overline{\pi_j \rightsquigarrow \langle x_j, c_j \rangle} \Vdash u : D\ \langle \overline{u_i} \rangle \rightsquigarrow x_{u:D\langle \overline{u_i} \rangle}[\overline{X_k}][\![\overline{c_j}]\!](\overline{x_j})$

We can then apply lemma B.6 to get
$\Gamma \mid \overline{X_k} \mid \Theta, \overline{\pi_j \rightsquigarrow \langle x_j, c_j \rangle} \vdash_{\mathrm{WF}} [\overline{u_p / X_p}]\sigma \rightsquigarrow [\overline{u^t_p / X_p}]\sigma^t$ as required.

From lemmas B.3, B.10 and B.11 we get
$\overline{\Gamma'' \mid \Theta''' \vdash_{\mathrm{WF}} \pi'_l \rightsquigarrow \langle \tau'^t_l, \omega'_l \rangle}$,
$\overline{\Gamma'' \mid \Theta''' \vdash_{\mathrm{WF}} \tau'_m \rightsquigarrow \tau'^t_m}$ and
$\Gamma'' \mid \Theta''' \vdash_{\mathrm{WF}} \tau'_R \rightsquigarrow \tau'^t_R$.

By lemma B.20 we have $\langle \Gamma'', \Theta''' \rangle \rightsquigarrow \langle (\Omega^t, \overline{c_j : \omega_j}, \overline{c_l : \omega'_l}), (\Gamma^t, \overline{X_k}, \overline{x_j : \tau^t_j}, \overline{X_n}, \overline{x_l : \tau'^t_l}, \overline{x_m : \tau'^t_m}) \rangle$.

By theorem B.1, $\Omega^t, \overline{c_j : \omega_j}, \overline{c_l : \omega'_l} \mid \Gamma^t, \overline{X_k}, \overline{x_j : \tau'^t_j}, \overline{X_n}, \overline{x_l : \tau'^t_l}, \overline{x_m : \tau'^t_m} \mid \emptyset \vdash e^t : \tau'^t_R$.

Using (t-fabs) twice we get
$\Omega^t, \overline{c_j : \omega_j}, \overline{c_l : \omega'_l} \mid \Gamma^t, \overline{X_k}, \overline{x_j : \tau^t_j}, \overline{X_n} \mid \emptyset \vdash \mathsf{fn}\ (\overline{x_l : \tau'^t_l})\{\mathsf{fn}\ (\overline{x_m : \tau'^t_m})\{e^t\}\}$
$: \mathsf{fn}(\overline{\tau'^t_l}) \to \mathsf{fn}(\overline{\tau'^t_m}) \to \tau'^t_R$.

Then, using (t-cabs) and (t-tabs) we get
$\Omega^t, \overline{c_j : \omega_j} \mid \Gamma^t, \overline{X_k}, \overline{x_j : \tau^t_j} \mid \emptyset \vdash e^t_F : \sigma'^t$.

We already know $\overline{\Gamma, \overline{X_p} \mid \Theta, Self : D\ \langle \overline{X_i} \rangle \rightsquigarrow \_ \vdash_{\mathrm{WF}} Self : \beta_s \rightsquigarrow \langle \tau^t_s, \omega_s \rangle}^{s}$.

By lemmas B.11, B.8 and B.5 we get
$\overline{\Gamma, \overline{X_k} \mid \Theta, \overline{\pi_j \rightsquigarrow \_} \vdash_{\mathrm{WF}} [\overline{u_p / X_p}](Self : \beta_s) \rightsquigarrow \langle [\overline{u^t_p / X_p}]\tau^t_s, [\overline{u^t_p / X_p}]\omega_s \rangle}^{s}$.

Following the same process we get

$$\frac{}{\Gamma, \overline{X_k} \mid \Theta, \overline{\pi_j \rightsquigarrow \_} \vdash_{\text{WF}} [\overline{u_p/X_p}](A_D \langle \overline{u_p} \rangle : \beta_a) \rightsquigarrow \langle \overline{[u_p^t/X_p]}\tau_a^t, \overline{[u_p^t/X_p]}\omega_a \rangle}\, a \ .$$

Then using lemma B.18 we get

$$\frac{}{\Omega^t, \overline{c_j : \omega_j} \mid \Gamma^t, \overline{X_k}, \overline{x_j : \tau_j^t} \mid \emptyset \vdash e^t : \overline{[u_p^t/X_p]}\tau_s^t}\, s \text{ and}$$

$$\frac{}{\Omega^t, \overline{c_j : \omega_j} \mid \Gamma^t, \overline{X_k}, \overline{x_j : \tau_j^t} \mid \emptyset \vdash e^t : \overline{[u_p^t/X_p]}\tau_a^t}\, a$$

We can then get the following derivation:

$$\frac{\dfrac{\dfrac{\Omega^t, \overline{c_j : \omega_j} \mid \Gamma^t, \overline{X_k}, \overline{x_j : \tau_j^t} \mid \emptyset \vdash S_D \langle \overline{u_p^t} \rangle \{ f : e_F^t, \overline{x_s : e_s^t}, \overline{x_a : e_a^t} \} : S_D \langle \overline{u_p^t} \rangle}{\Omega^t, \overline{c_j : \omega_j} \mid \Gamma^t, \overline{X_k} \mid \emptyset \vdash \text{fn } (\overline{x_j : \tau_j^t}) \{ S_D \langle \overline{u_p^t} \rangle \{ f : e_F^t, \overline{x_s : e_s^t}, \overline{x_a : e_a^t} \} \} : \text{fn}(\overline{\tau_j^t}) \to S_D \langle \overline{u_p^t} \rangle}}{\Omega^t \mid \Gamma^t \mid \emptyset \vdash \Lambda \overline{X_k}.\Lambda \overline{c_j : \omega_j}.\text{fn } (\overline{x_j : \tau_j^t}) \{ S_D \langle \overline{u_p^t} \rangle \{ f : e_F^t, \overline{x_s : e_s^t}, \overline{x_a : e_a^t} \} \} : \sigma_D^t}$$

Using (t-fun) we get
$\Omega^t \mid \Gamma^t \vdash x_{u:D\langle \overline{u_i} \rangle} : \sigma_D^t = \Lambda \overline{X_k}.\Lambda \overline{c_j : \omega_j}.\text{fn } (\overline{x_j : \tau_j^t}) \{ S_D \langle \overline{u_p^t} \rangle \{ f : e_F^t, \overline{x_s : e_s^t}, \overline{x_a : e_a^t} \} \}$
$: [x_{u:D\langle \overline{u_i} \rangle} : \sigma_D^t] \mid \emptyset.$

We then have
$\overline{\Omega_i^t} = [c_{u:D\langle \overline{u_i} \rangle} : \forall \overline{X_k}.A_D \langle \overline{u_p^t} \rangle \sim \forall \overline{X_k}.u_A^t]$ and
$\overline{\Gamma_i^t} = [x_{u:D\langle \overline{u_i} \rangle} : \sigma_D^t]$
as required.

- **Case** (obj-trt): Note that the outputs of (trait) are a subset of the outputs of (obj-trt). As such, their proof is the same as in the proof for (trait). We only show the proof for the additional outputs.

$$\Theta' = \{ \ldots, \overline{\frac{}{\forall \overline{X_i}, \overline{X_d}.\overline{\pi_{dm}}^m \Rightarrow \pi_d \rightsquigarrow \langle x_d, c_d \rangle}\, d} \}$$

$$\overline{item_i^t} = \overline{\begin{array}{l} x_d : \forall \overline{X_i}, \overline{X_d}.\forall \overline{\omega_{dm}}.\text{fn}(\overline{\tau_{dm}^t}) \to \tau_d^t = \Lambda \overline{X_i}, \overline{X_d}.\Lambda \overline{c_{dm} : \omega_{dm}}.\text{fn } (\overline{x_{dm} : \tau_{dm}^t}) \{ e_d^t \}; \quad d \\ \text{axiom } c_d : \forall \overline{X_i}, \overline{X_d}.u_d^t \mapsto X_d; \quad d \end{array}}$$

Side conditions:
$\Gamma, \overline{X_p} \vdash D \langle \overline{X_i} \rangle \Uparrow \{ \overline{D_d \langle \overline{u_{dn}}^n, A_d \mapsto \_ \rangle}^d \}$
$\overline{A_d :: X_d}$
$u_{\text{obj}} = \exists T : D \langle \overline{X_i}, \overline{A_{Dd} \langle T, [T/\text{Self}_{\mathbf{U}}]\overline{u_{dn}}^n \rangle \sim X_d}^d \rangle$
$\dfrac{}{\pi_d = u_{\text{obj}} : D_d \langle \overline{[u_{\text{obj}}/\text{Self}_{\mathbf{U}}]u_{dn}}^n, A_d \mapsto X_d \rangle}\, d$
$\dfrac{}{\Gamma, \overline{X_i}, \overline{X_d} \mid \Theta \mid \overline{\forall \overline{X_i}, \overline{X_d}.\pi_d \rightsquigarrow \langle x_d, c_d \rangle} \vdash_{\text{os}} \pi_d \rightsquigarrow \langle e_d^t, \overline{\pi_{dm} \rightsquigarrow x_{dm}} \rangle}\, d$

$$\frac{}{\Gamma \mid \Theta \vdash_{\text{WF}} \forall \overline{X_i}, \overline{X_d}.\overline{\pi_{dm}}^m \Rightarrow \pi_d \rightsquigarrow \left\langle \forall \overline{X_i}, \overline{X_d}.\forall \overline{\omega_{dm}}.\text{fn}(\overline{\tau_{dm}^t}) \to \tau_d^t,\ \forall \overline{X_i}.\overline{X_d}.u_d^t \sim \forall \overline{X_i}.\overline{X_d}.X_d \right\rangle}\, d$$

Using lemma B.21 and lemma B.8 we get

$$\frac{}{\Gamma, \overline{X_i}, \overline{X_d}, T \mid \Theta, T : D \langle \overline{X_i} \rangle \vdash_{\text{WF}} T : D_d \langle \overline{[T/\text{Self}_{\mathbf{U}}]u_{ns}}, A_d \mapsto \star \rangle}\, d \ .$$

We can then use (wf-desc) to show $\Gamma, \overline{X_i}, \overline{X_d} \mid \Theta \vdash_{\text{WF}} u_{\text{obj}}$.

Using lemma B.21 again we get $\overline{\Gamma, \overline{X_i}, \overline{X_d} \mid \Theta, u_{\text{obj}} : D\ \langle \overline{X_i} \rangle \vdash_{\text{WF}} \pi_d}$.

Let $\overline{\theta_d = \forall \overline{X_i}.\overline{X_d}.\pi_d}$.

From the definition of the relation $\Uparrow$ we know that $D\ \langle \overline{X_i}, A \mapsto \star \rangle \in \overline{\pi_d}$,
so $\Gamma, \overline{X_d}, \overline{X_i} \mid \Theta, \overline{\theta_d} \vdash_{\text{WF}} u_{\text{obj}} : D\ \langle \overline{X_i} \rangle$.

Then, we can use lemma B.5 to prove $\overline{\Gamma, \overline{X_i}, \overline{X_d} \mid \Theta, \overline{\theta_d} \vdash_{\text{WF}} \pi_d}$.

From lemma B.3, if $\overline{\Gamma \mid \Theta \vdash_{\text{WF}} \forall \overline{X_i}, \overline{X_d}.\overline{\pi_{dm}}^{m} \Rightarrow \pi_d \rightsquigarrow \langle \_, \vartheta_d \rangle}^{d}$ and
$\overline{\Gamma \mid \Theta \vdash_{\text{WF}} \theta_d \rightsquigarrow \langle \_, \vartheta'_d \rangle}^{d}$ then $\overline{\vartheta_d = \vartheta'_d}$.

Then we can use lemma B.22 to get $\overline{\overline{\Gamma, \overline{X_i}, \overline{X_d} \mid \Theta \vdash_{\text{WF}} \pi_{dm}}}$ and
$\Omega^t, \overline{c_d : \vartheta_d} \mid \Gamma^t, \overline{x_{dm} : \tau^t_{dm}} \mid \emptyset \vdash_T \tau^t_d$.

Since $\Theta' \subseteq \Theta$, we conclude that $\overline{c_d : \vartheta_d} \in \Omega^t$.

Then we can use rules (t-fabs), (t-cabs), (t-capp) and (t-decl) to prove that $\overline{x_d}$ are well-typed. Similarly we can use (t-axiom) to prove that axioms $\overline{c_d}$ are well-typed.

□

Theorem B.3 (Translation of programs preserves well-typedness)**.** *If* $\vdash_P pgm \rightsquigarrow pgm^t : \Gamma \mid \Theta$ *and then there are some* Ω^t, Γ^t *such that* $\langle \Gamma, \Theta \rangle \rightsquigarrow \langle \Omega^t, \Gamma^t \rangle$ *and* $\vdash_T pgm^t : \Omega^t \mid \Gamma^t$.

Proof. By inversion of the well-typed programs judgment we get if $\vdash_P \overline{item_i} : \Gamma \mid \Theta \rightsquigarrow pgm^t$
then $\overline{\Gamma \mid \Theta \vdash item_i : \Gamma_i \mid \Theta_i \rightsquigarrow pgm^t_i}$, and $\Gamma = \overline{\Gamma_i}$, and $\Theta = \Theta_i$ and $pgm^t = \overline{pgm^t_i}$.
Using lemma B.25 we get some $\overline{\Omega^t_{1i}, \Omega^t_{2i}, \Gamma^t_{1i}, \Gamma^t_{2i}}$ such that
$\overline{\Gamma \mid \Theta \vdash \Gamma_i \rightsquigarrow \langle \Omega^t_{1i}, \Gamma^t_{1i} \rangle}$ and $\overline{\Gamma \mid \Theta \vdash \Theta_i \rightsquigarrow \langle \Omega^t_{2i}, \Gamma^t_{2i} \rangle}$.
Let $\Gamma^t = \overline{\Gamma^t_{1i}, \Gamma^t_{2i}}$ and $\Omega^t = \overline{\Omega^t_{1i}, \Omega^t_{2i}}$.
By lemma B.20 we get $\langle \Gamma, \Theta \rangle \rightsquigarrow \langle \Omega^t, \Gamma^t \rangle$.
Then using theorem B we get $\overline{\Gamma \mid \Theta \vdash item_i : \Omega^t_{1i}, \Omega^t_{2i} \mid \Gamma^t_{1i}, \Gamma^t_{2i}}^{i}$.
We can then use (t-pgm) to show $\vdash_T \overline{item^t_i} : \Omega^t \mid \Gamma^t$. □

Appendix C

Trait coherence proofs

Definition C.1. *Define δ_Γ as follows:*

$$\begin{aligned}
\delta_\Gamma(()) &= () \\
\delta_\Gamma(T) &= T \\
\delta_\Gamma(fn(\overline{\tau_i}) \to \tau) &= fn(\overline{\delta_\Gamma(\tau_i)}) \to \delta_\Gamma(\tau) \\
\delta_\Gamma(\&\tau) &= \&\delta_\Gamma(\tau) \\
\delta_\Gamma(S\,\langle\overline{\tau_i}\rangle) &= \begin{cases} S\,\langle\overline{\delta_\Gamma(\tau_i)}\rangle & \text{if } S\,\langle\overline{T_i}\rangle\{\ldots\} \in \Gamma \\ T_\delta & \text{otherwise} \end{cases} \\
\delta_\Gamma(\tau : D\,\langle\overline{\tau_i}\rangle) &= \delta_\Gamma(\tau) : D\,\langle\overline{\delta_\Gamma(\tau_i)}\rangle
\end{aligned}$$

Lemma C.1 (Well-formedness inversion lemma)**.**

1. *If $\Gamma \vdash_{\text{WF}} T$ then $T \in \Gamma$.*
2. *If $\Gamma \vdash_{\text{WF}} fn(\overline{\tau_i}) \to \tau$ then $\overline{\Gamma \vdash_{\text{WF}} \tau_i}$ and $\Gamma \vdash_{\text{WF}} \tau$.*
3. *If $\Gamma \vdash_{\text{WF}} \&\tau$ then $\Gamma \vdash_{\text{WF}} \tau$.*
4. *If $\Gamma \vdash_{\text{WF}} S\,\langle\overline{\tau_i}\rangle$ then $S\,\langle\overline{T_i}\rangle\{\ldots\} \in \Gamma$ for some $\overline{T_i}$ and $\overline{\Gamma \vdash_{\text{WF}} \tau_i}$.*
5. *If $\Gamma \vdash_{\text{WF}} \forall\overline{T_i}.\overline{\pi_j} \Rightarrow \pi$ then $\overline{\Gamma, \overline{T_i} \vdash_{\text{WF}} \pi_j}$ and $\Gamma, \overline{T_i} \vdash_{\text{WF}} \pi$.*
6. *If $\Gamma \vdash_{\text{WF}} \tau : D\,\langle\overline{\tau_i}\rangle$. then $\Gamma \vdash_{\text{WF}} D\,\langle\overline{T_i}\rangle$ for some T_i and $\Gamma \vdash_{\text{WF}} \tau$ and $\overline{\Gamma \vdash_{\text{WF}} \tau_i}$.*

Proof. Straightforward induction on derivations in the NanoRust well-formedness judgments in figure 3.2. □

Lemma C.2. *If $\Pi \vdash_{\text{OR}} K \triangleright \tau : D\,\langle\overline{\tau_n}\rangle$ then either* **ltenv** *$(\Pi, K) \vdash D$ or there is some $\tau' \in \{\tau, \overline{\tau_n}\}$ such that $\Pi \vdash_{\text{LT}} K \triangleright \tau'$ and $FV(\overline{\tau_i})$ where $\overline{\tau_i}$ are all types that precede τ' in the sequence $[\tau, \overline{\tau_n}]$.*

Proof. Straightforward from the orphan relation. □

Lemma C.3. *If* $\Gamma, \overline{T} \vdash_{\text{WF}} \pi$ *then* $\delta_\Gamma(\pi) = \pi$.

Proof. Suppose $\pi = \tau : D\,\langle\overline{\tau_i}\rangle$.
By lemma C.1, $\Gamma, \overline{T} \vdash_{\text{WF}} \tau$ and $\overline{\Gamma, \overline{T} \vdash_{\text{WF}} \tau_i}$.
We prodeed by induction on τ to show that if $\Gamma, \overline{T} \vdash_{\text{WF}} \tau$ then $\delta_\Gamma(\tau) = \tau$.
There is only one interesting case: (wf-struct) where $\tau = S\,\langle\overline{\tau_j}\rangle$ and , $S\,\langle\overline{T_j}\rangle\,\{\ldots\} \in \Gamma$.
Then $\delta_\Gamma(S\,\langle\overline{T_j}\rangle) = S\,\langle\overline{T_j}\rangle$.
The same reasoning applies for for $\overline{\Gamma, \overline{T} \vdash_{\text{WF}} \tau_i}$.
Then, using the definition of δ_Γ we get if $\delta_\Gamma(\pi) = \pi$. □

Lemma C.4. *If* $\Pi \vdash_{\text{LT}} K \rhd \tau$ *then* $\tau = \overline{\&} S\,\langle\overline{\tau_i}\rangle$ *where* $S\,\langle\overline{T_i}\rangle\,\{\ldots\} \in \mathbf{ltenv}\,(\Pi, K)$ *for some* $S, \overline{\tau_i}, \overline{T_i}$.

Proof. Straightforward from the definition of the local type relation. □

Lemma C.5. *Suppose no overlapping identifiers in environments in* Π.
If $\Pi \vdash_{\text{LT}} K \rhd \tau$ *and* $\Pi \vdash K' \not\sqsubseteq K$ *then* $\mathbf{tenv}\,(\Pi, K') \not\vdash_{\text{WF}} \tau$.

Proof. Using lemma C.4 we get $\tau = \overline{\&} S\,\langle\overline{\tau_i}\rangle$ where $S\,\langle\overline{T_i}\rangle\,\{\ldots\} \in \mathbf{ltenv}\,(\Pi, K)$ for some $S, \overline{\tau_i}, \overline{T_i}$.
Since $\Pi \vdash K' \not\sqsubseteq K$ and because of the no-overlap assumption, we can conclude that there are no $\overline{T_i}$ such that $S\,\langle\overline{T_i}\rangle\,\{\ldots\} \in \mathbf{tenv}\,((\Pi, K'))$, which based on lemma C.1 lets us conclude that $\mathbf{tenv}\,(\Pi, K') \not\vdash_{\text{WF}} \tau$.

□

Lemma C.6. *Suppose no overlapping identifiers in environments in* Π.
If $\mathbf{ltenv}\,(\Pi, K) \vdash D$ *and* $\Pi \vdash K' \not\sqsubseteq K$ *then* $\mathbf{tenv}\,(\Pi, K') \not\vdash D$.

Proof. Analogous to the proof of lemma C.5. □

Lemma C.7.

1. *If* $\theta \in \mathbf{cenv}\,(\Pi, K)$ *then there is some crate* K' *such that* $\Pi \vdash K \sqsubseteq K'$ *and* $\theta \in \mathbf{lcenv}\,(\Pi, K')$.
2. *If* $\mathbf{tenv}\,(\Pi, K) \vdash D$ *then there is some crate* K' *such that* $\Pi \vdash K \sqsubseteq K'$ *and* $\mathbf{ltenv}\,(\Pi, K') \vdash D$.
3. *If* $\mathbf{tenv}\,(\Pi, K) \vdash S$ *then there is some crate* K' *such that* $\Pi \vdash K \sqsubseteq K'$ *and* $\mathbf{ltenv}\,(\Pi, K') \vdash S$.

Proof.

1. By induction on the relation $\sqsubseteq$, assuming that the lemma holds for $\theta \in \mathbf{cenv}\,(\Pi, K')$ all K' such that $\Pi \vdash K \sqsubset K'$. We can make this inductive argument because of the restriction on Π, which prevents circular crate dependencies, which in turn makes the relation well-founded.

   Let $\Pi(K) = \langle\{\overline{K_i}\}, \Gamma, \Theta\rangle$. Then $\mathbf{cenv}\,(\Pi, K) = \overline{\mathbf{cenv}\,(\Pi, K_i)}^{\,i}, \Theta$.
   Either $\theta \in \mathbf{cenv}\,(\Pi, K'')$ for some $K'' \in \overline{K_i}$ or $\theta \in \Theta$.

If the $\theta \in \Theta$ then $\theta \in \mathbf{lcenv}(\Pi, K)$ by def. of **lcenv**.

If $\theta \in \mathbf{cenv}(\Pi, K'')$, then $\Pi \vdash K \sqsubset K''$.
By IH, there is some K' such that $\Pi \vdash K'' \sqsubseteq K'$ and $\theta \in \mathbf{lcenv}(\Pi, K')$.
Then, by def. of the relation $\sqsubseteq$, $\Pi \vdash K \sqsubseteq K'$ as required.

2. Analogous to the proof of 1.

3. Analogous to the proof of 1.

□

Lemma C.8. *If* $\Pi \vdash_{\text{LT}} K \rhd \tau$ *and there are no overlapping identifiers between* Γ *and* $\mathbf{ltenv}(\Pi, K)$*, then* $\delta_\Gamma(\tau) \in \text{BLANKETTYPE}$.

Proof. By induction on derivations $\Pi \vdash_{\text{LT}} K \rhd \tau$.

- **Case** (l-ref): $\tau = \&\tau'$ and $\Pi \vdash_{\text{LT}} K \rhd \tau'$.
  Then by IH $\delta_\Gamma(\tau') \in \text{BLANKETTYPE}$. Then $\delta_\Gamma(\&\tau') = \&\delta_\Gamma(\tau') \in \text{BLANKETTYPE}$.

- **Case** (l-struct): $\tau = S\,\langle\overline{\tau_i}\rangle$ and $S\,\langle\overline{T_i}\rangle\{\ldots\} \in \mathbf{ltenv}(\Pi, K)$.
  There are no T_i' such that $S\,\langle\overline{T_i'}\rangle\{\ldots\} \in \Gamma$ because of the no-overlap assumption.
  So $\delta_\Gamma(S\,\langle\overline{\tau_i}\rangle) = T_\delta \in \text{BLANKETTYPE}$.

□

Lemma C.9. *If* $\Pi \vdash_{\text{LT}} K \rhd \tau$ *then* $\Pi \vdash_{\text{LT}} K \rhd [\overline{\tau_i/T_i}]\tau$ *for any types* $\overline{\tau_i}$ *and type variables* $\overline{T_i}$.

Proof. By induction on derivations $\Pi \vdash_{\text{LT}} K \rhd \tau$.

- **Case** (l-ref): $\tau = \&\tau'$ and $\Pi \vdash_{\text{LT}} K \rhd \tau'$.
  Then by IH $\Pi \vdash_{\text{LT}} K \rhd [\overline{\tau_i/T_i}]\tau'$ and by (l-ref) and def. of substitution, $\Pi \vdash_{\text{LT}} K \rhd [\overline{\tau_i/T_i}](\&\tau')$.

- **Case** (l-struct): $\tau = S\,\langle\overline{\tau_j}\rangle$ and $S\,\langle\overline{T_j}\rangle\{\ldots\} \in \mathbf{ltenv}(\Pi, K)$.
  $[\overline{\tau_i/T_i}]S\,\langle\overline{\tau_j}\rangle = S\,\langle\overline{[\overline{\tau_i/T_i}]\tau_j}\rangle$ and so by (l-struct) $\Pi \vdash_{\text{LT}} K \rhd S\,\langle\overline{[\overline{\tau_i/T_i}]\tau_j}\rangle$.

□

Lemma C.10. *If* $\Pi \vdash_{\text{LT}} K \rhd \delta_\Gamma(\tau)$ *then* $\Pi \vdash_{\text{LT}} K \rhd \tau$.

Proof. By induction on derivations $\Pi \vdash_{\text{LT}} K \rhd \tau$.

- **Case** (l-ref): If $\delta_\Gamma(\tau) = \&\tau'$ then $\tau' = \delta_\Gamma(\tau'')$ and $\tau = \&\tau''$ for some τ'' by definition of δ_Γ.
  Then $\Pi \vdash_{\text{LT}} K \rhd \delta_\Gamma(\tau'')$ by the rule's premise and, by IH, $\Pi \vdash_{\text{LT}} K \rhd \tau''$.
  Then by (l-ref) $\Pi \vdash_{\text{LT}} K \rhd \&\tau''$ as required.

- **Case** (l-struct): If $\delta_\Gamma(\tau) = S\,\langle\overline{\tau_i}\rangle$ then $\overline{\tau_i = \delta_\Gamma(\tau'_i)}$ and $\tau = S\,\langle\overline{\tau'_i}\rangle$ with $S\,\langle\overline{T_i}\rangle\{\ldots\} \in \Gamma$ for some $\overline{\tau'_i}$ and $\overline{T_i}$.
  From the rule's premise $S\,\langle\overline{T_i}\rangle \in \mathbf{ltenv}\,(\Pi, K)$.
  Then by (l-struct), $\Pi \vdash_{\mathrm{LT}} K \rhd S\,\langle\overline{\tau'_i}\rangle$ as required.

□

Lemma C.11. *If $FV(\delta_\Gamma(\tau)) = \emptyset$ then $FV(\tau) = \emptyset$.*

Proof. By structural induction on τ

- **Case** (): Trivial.
- **Case** $\mathsf{fn}(\overline{\tau_i}) \to \tau$:
  $\delta_\Gamma(\mathsf{fn}(\overline{\tau_i}) \to \tau) = \mathsf{fn}(\overline{\delta_\Gamma(\tau_i)}) \to \delta_\Gamma(\tau)$.
  By IH, $FV(\overline{\tau_i}, \tau) = \emptyset$ and so $FV(\mathsf{fn}(\overline{\tau_i}) \to \tau)$ as required.
- **Case** $\&\tau$: $\delta_\Gamma(\&\tau) = \&\delta_\Gamma(\tau)$.
  By IH, $FV(\tau) = \emptyset$ so $FV(\&\tau) = \emptyset$.
- **Case** T: $\delta_\Gamma(T) = T$ and $FV(T) = T$ so this case is vacuous.
- **Case** $S\,\langle\overline{\tau_i}\rangle$: If $\delta_\Gamma(S\,\langle\overline{\tau_i}\rangle) = T_\delta$ then this case is vacuous as $FV(T_\delta) = T_\delta$.
  Otherwise, $\delta_\Gamma(S\,\langle\overline{\tau_i}\rangle) = S\,\langle\overline{\delta_\Gamma(\tau_i)}\rangle$. By IH, $FV(\overline{\tau_i}) = \emptyset$ so $FV(S\,\langle\overline{\tau_i}\rangle) = \emptyset$.

□

Lemma C.12. *If $\Pi \vdash_{\mathrm{OR}} K \rhd \delta_\Gamma(\pi)$ then $\Pi \vdash_{\mathrm{OR}} K \rhd \pi$.*

Proof. By induction on derivations $\Pi \vdash_{\mathrm{OR}} K \rhd \delta_\Gamma(\pi)$.

- **Case** (orph1): Trivial.
- **Case** (orph2): $\pi = \tau : D\,\langle\overline{\tau_i}\rangle$ and $\Pi \vdash_{\mathrm{LT}} K \rhd \delta_\Gamma(\tau)$. Using lemma C.10 we get $\Pi \vdash_{\mathrm{LT}} K \rhd \tau$ so $\Pi \vdash_{\mathrm{OR}} K \rhd \pi$ using (orph2).
- **Case** (orph2): $\pi = \tau_0 : D\,\langle\overline{\tau_i}, \tau, \overline{\tau_j}\rangle$ and $\Pi \vdash_{\mathrm{LT}} K \rhd \delta_\Gamma(\tau)$ and $FV(\delta_\Gamma(\tau_0), \delta_\Gamma(\tau_i)) = \emptyset$.
  Then by lemma C.10 $\Pi \vdash_{\mathrm{LT}} K \rhd \tau$ and by lemma C.11 $FV(\tau_0, \overline{\tau_i}) = \emptyset$
  so by (orph2), $\Pi \vdash_{\mathrm{OR}} K \rhd \pi$.

□

Lemma C.13. *If $\tau \notin$ BLANKETTYPE and $\mathbf{tenv}\,(\Pi, K_1) \vdash_{\mathrm{WF}} \tau$ and $\Pi \vdash K_2 \sqsubset K_1$ then for any type substitution ϕ, $\Pi \not\vdash_{\mathrm{LT}} K_2 \rhd \phi(\tau)$.*

Proof. If $\Pi \vdash K_2 \sqsubset K_1$ then $\mathbf{ltenv}(\Pi, K_2)$ and $\mathbf{tenv}(\Pi, K_1)$ have no overlapping identifiers.
Suppose for contradiction that $\Pi \vdash_{\text{LT}} K_2 \rhd \phi(\tau)$.
Then, by lemma C.4, $\phi(\tau) = \overline{\&} S\, \langle \overline{\tau_i} \rangle$ for some S and $\overline{\tau_i}$ and $\mathbf{ltenv}(\Pi, K) \vdash S$.
Since $\Pi \vdash K_1 \not\sqsupseteq K_2$, lemma C.5 gives us $\mathbf{tenv}(\Pi, K_1) \nvdash_{\text{WF}} \phi(\tau)$.
Since τ is not a blanket type it can only have form $\overline{\&} S\, \langle \overline{\tau_i'} \rangle$. But then, since $\mathbf{tenv}(\Pi, K_1) \nvdash S$, this is impossible. □

Lemma C.14. *Let K be a consistent crate under Π where $\Pi = \Pi'' \cup \{K : \langle \{\overline{K_a}\}, \Gamma_K, \Theta_K \rangle\}$ and let $\Pi' = [K'/K]\Pi'' \cup \{K' : \langle \{\overline{K_a}\}, \Gamma_K, \Theta_K \cup \{\theta_1\} \rangle\}$ where K' is consistent in Π'.*
Then, for any crate K_1, if $\Pi \vdash_{\text{OR}} K_1 \rhd \pi$ then $\Pi' \vdash_{\text{OR}} [K'/K]K_1 \rhd \pi$.
Moreover, if $\Pi \vdash_{\text{LT}} K_1 \rhd \tau$ then $\Pi' \vdash_{\text{LT}} [K'/K]K_1 \rhd \tau$.

Proof. Straightforward from the definition of the orphan rule and local type relations as the relations don't use constraint environments. □

Theorem 8.2 (Trait coherence)**.** *Suppose:*

- $(K : \langle \{\overline{K_r}\}, \Gamma_K, \Theta_K \rangle) \in \Pi$,
- *each crate $K' \in \overline{K_r}$ is consistent in Π,*
- $\langle \mathbf{tenv}(\Pi, K), \mathbf{cenv}(\Pi, K) \rangle$ *well-formed,*
- *every impl in Θ_K obeys the orphan rule w.r.t. crate K, and*
- *for every pair of impl constraints schemes $\theta_1 \in \Theta_K$ and $\theta_2 \in \mathbf{cenv}(\Pi, K)$, θ_1 and θ_2 don't overlap under Π and $\mathbf{cenv}(\Pi, K)$.*

Then K is consistent in Π.

Proof. Let $\Gamma = \mathbf{tenv}(\Pi, K)$ and $\Theta = \mathbf{cenv}(\Pi, K)$.
We must show that for every pair of distinct impl constraint schemes θ_1 and $\theta_2 \in \Theta$ there is no overlap.
We know that $\Theta = \mathbf{cenv}(\Pi, K) = \bigcup \{\overline{\mathbf{cenv}(\Pi, K_r)}^{r}, \Theta_K\}$ by definition.
From the theorem's assumptions,
we know that if $\theta_1 \in \Theta_K$ and $\theta_2 \in \Theta$ then θ_1, θ_2 don't overlap in Π, Θ.
We must show the same for each $\theta_1 \in \Theta'$ where $\Theta' = \bigcup \{\overline{\mathbf{cenv}(\Pi, K_r)}^{r}\}$.
Let $K_1 \in \overline{K_r}$ and let $\Gamma_1 = \mathbf{tenv}(\Pi, K_1)$ and $\Theta_1 = \mathbf{cenv}(\Pi, K_1)$.
Then let $\Gamma_2 = \Gamma \setminus \Gamma_1$ and $\Theta_2 = \Theta \setminus \Theta_1$.
Let $\theta_1 \in \mathbf{lcenv}(\Pi, K_3)$ and $\theta_2 \in \mathbf{lcenv}(\Pi, K_4)$ for some crates K_3, K_4.
We consider two cases:

- **Case** 1: $\Pi \vdash K_3 \not\sqsubseteq K_4$ and $\Pi \vdash K_4 \not\sqsubseteq K_3$.
  Suppose that there exists a substitution ϕ such that $\phi(\tau_1 : D\ \langle\overline{\tau_{1i}}\rangle) = \phi(\tau_2 : D\ \langle\overline{\tau_{2i}}\rangle)$.
  By lemma C.1, $\mathbf{tenv}\,(\Pi, K_3) \vdash D$ and $\mathbf{tenv}\,(\Pi, K_4) \vdash D$.
  Since we are assuming non-overlapping trait identifiers in Π, by lemma C.7 there is some crate K_5 such that $\mathbf{ltenv}\,(\Gamma, K_5) \vdash D$ where $\Pi \vdash K_3 \sqsubseteq K_5$ and $\Pi \vdash K_4 \sqsubseteq K_5$.
  Since $\Pi \vdash K_3 \not\sqsubseteq K_4$ and $\Pi \vdash K_4 \not\sqsubseteq K_3$, we can conclude that $K_5 \neq K_3$ and $K_5 \neq K_4$.
  From the definition of consistent crates,
  $\Pi \vdash_{\text{OR}} K_3 \rhd \tau_1 : D\ \langle\overline{\tau_{1i}}\rangle$ and $\Pi \vdash_{\text{OR}} K_4 \rhd \tau_2 : D\ \langle\overline{\tau_{2i}}\rangle$.
  Let $\tau_1, \overline{\tau_{1i}} = [\tau_{11}, \ldots, \tau_{1n}]$ and $\tau_2, \overline{\tau_{2i}} = [\tau_{21}, \ldots, \tau_{2n}]$ where n is the length of the sequence.
  Then from the orphan rules we know that there are some τ_{1p} and τ_{2q}
  such that $\Pi \vdash_{\text{LT}} K_3 \rhd \tau_{1p}$ and $\Pi \vdash_{\text{LT}} K_4 \rhd \tau_{2q}$
  with $FV(\tau_{11}, \ldots, \tau_{1r}, \tau_{21}, \ldots, \tau_{2s}) = \emptyset$
  where $r = p - 1$ and $s = q - 1$.
  Then one of the following is true:

  - **Case** $p = q$: From lemma C.4, $\tau_{1p} = \overline{\&} S_1\ \langle\overline{\tau}\rangle$ where $S_1\ \langle\overline{T}\rangle\ \{\ldots\} \in \mathbf{ltenv}\,(\Pi, K_3)$.
    Similarly, $\tau_{2p} = \overline{\&} S_2\ \langle\overline{\tau}\rangle$ where $S_2\ \langle\overline{T}\rangle\ \{\ldots\} \in \mathbf{ltenv}\,(\Pi, K_4)$.
    Since the local environments of K_3 and K_4 are not allowed to overlap, $S_1 \neq S_2$ and so there is no substitution ϕ such that $\phi(\tau_{1p}) = \phi(\tau_{2p})$.
  - **Case** $p < q$: From lemma C.4, $\tau_{1p} = \overline{\&} S_1\ \langle\overline{\tau}\rangle$ where $S_1\ \langle\overline{T}\rangle\ \{\ldots\} \in \mathbf{ltenv}\,(\Pi, K_3)$
    and from lemma C.2, $FV(\tau_{2p}) = \emptyset$ since $p < q$.
    So, since $\phi(\tau_{1p}) = \phi(\tau_{2p})$, then $\tau_{2p} = \overline{\&} S_1\ \langle\overline{\tau'}\rangle$.
    Using lemma C.1, we get $S_1\ \langle\overline{T}\rangle\ \{\ldots\} \in \mathbf{tenv}\,(\Pi, K_4)$ for some $\overline{T}$.
    However, due to non-overlapping identifiers in Π
    and the fact that $\Pi \vdash K_4 \not\sqsubseteq K_3$, this is impossible by lemma C.5.
  - **Case** $p > q$: Analogous to the previous case.

  Since all above cases are impossible, we have derived a contradiction: no such ϕ as defined above can exist.

- **Case** 2: $\Pi \vdash K_3 \sqsubseteq K_4$.
  Let $\Theta_1 = \mathbf{cenv}\,(\Pi, K_1)$ and $\Gamma_1 = \mathbf{cenv}\,(\Pi, K_1)$. Note that $\Theta_1 \subseteq \Theta$ and $\Gamma_1 \subseteq \Gamma$. We know that θ_1 and θ_2 don't overlap under environments Π, Γ_1 and Θ_1 from consistency of K_1.

  To derive a contradiction, we want to show that if θ_1 and θ_2 overlap under Π, Γ, Θ then they also overlap under Π, Γ_1, Θ_1.
  Specifically we'll show that if there is a substitution ϕ such that
  $\phi(\tau_1 : D\ \langle\overline{\tau_{1i}}\rangle) = \phi(\tau_2 : D\ \langle\overline{\tau_{2i}}\rangle)$ and
  $\Pi \mid \Theta \Vdash K_4 \rhd \phi(\pi)$ for each $\pi \in \{\overline{\pi_k}, \overline{\pi_m}\}$

then there is a substitution ϕ' such that $\phi'(\tau_1 : D\,\langle\overline{\tau_{1i}}\rangle) = \phi'(\tau_2 : D\,\langle\overline{\tau_{2i}}\rangle)$ and
$\Pi \mid \Theta_1 \Vdash K_4 \rhd \phi'(\pi)$ for each $\pi \in \{\overline{\pi_k}, \overline{\pi_m}\}$.

Wlog, suppose $\phi = [\overline{\tau_n/T_n}]$. Then let $\phi' = [\overline{\delta_{\Gamma 1}(\tau_n)/T_n}]$.
From consistency of K_1 it follows that $\Gamma_1 \vdash_{\text{WF}} \theta_1$ and $\Gamma_2 \vdash_{\text{WF}} \theta_2$.
From lemma C.1 we get $\Gamma_1, \overline{T_j} \vdash_{\text{WF}} \tau_1 : D\,\langle\overline{\tau_{1i}}\rangle$ and $\Gamma_1, \overline{T_l} \vdash_{\text{WF}} \tau_2 : D\,\langle\overline{\tau_{2i}}\rangle$
with $\Gamma_1 \vdash D\,\langle\overline{T_i}\rangle$.
Then, we can use lemma C.3 to get $\delta_{\Gamma 1}(\tau_1 : D\,\langle\overline{\tau_{1i}}\rangle) = \tau_1 : D\,\langle\overline{\tau_{1i}}\rangle$
and $\delta_{\Gamma 1}(\tau_2 : D\,\langle\overline{\tau_{2i}}\rangle) = \tau_2 : D\,\langle\overline{\tau_{2i}}\rangle$.
It then follows that $\phi'(\tau_1 : D\,\langle\overline{\tau_{1i}}\rangle) = \delta_{\Gamma 1}(\phi(\tau_1 : D\,\langle\overline{\tau_{1i}}\rangle))$
and $\phi'(\tau_2 : D\,\langle\overline{\tau_{2i}}\rangle) = \delta_{\Gamma 1}(\phi(\tau_2 : D\,\langle\overline{\tau_{2i}}\rangle))$.
As $\delta_{\Gamma 1}$ is a deterministic function, we can conclude that $\phi'(\tau_1 : D\,\langle\overline{\tau_{1i}}\rangle) = \phi'(\tau_2 : D\,\langle\overline{\tau_{2i}}\rangle)$.

Using the well-formedness inversion lemma we also get
$\Gamma_1, \overline{T_j}, \overline{T_l} \vdash_{\text{WF}} \pi$ for all $\pi \in \{\overline{\pi_k}, \overline{\pi_m}\}$.
It then follows that $\phi'(\pi) = \delta_{\Gamma 1}(\phi(\pi))$ for each such π.
Let $\pi = \tau_3 : D_3\,\langle\overline{\tau_m}\rangle$.
Then from the well-formedness inversion lemma we get $\Gamma_1 \vdash D_3\,\langle\overline{T_m}\rangle$.
We now use induction to show that if $\pi = \tau : D\,\langle\overline{\tau_i}\rangle$, and $\Gamma_1 \vdash D\,\langle\overline{T_i}\rangle$, and $\Pi \mid \Theta \Vdash K_4 \rhd \pi$
then $\Pi \mid \Theta \Vdash K_4 \rhd \delta_{\Gamma 1}(\pi)$.
We proceed by rule induction on $\Pi \mid \Theta \Vdash K_4 \rhd \pi$:

- **Case** (cons): Then $\pi = [\overline{\tau_i/T_i}]\pi'$ and $\forall\overline{T_i}.\overline{\pi_j} \Rightarrow \pi' \in \Theta$ and $\overline{\Pi \mid \Theta \Vdash K_4 \rhd [\overline{\tau_i/T_i}]\pi_j}$.
  Let $\Theta_2 = \Theta \setminus \Theta_1$ and $\Gamma_2 = \Gamma \setminus \Gamma_1$. Then $\forall\overline{T_i}.\overline{\pi_j} \Rightarrow \pi'$ is either in Θ_1 or Θ_2. We consider both cases separately:
  - **Case** $\forall\overline{T_i}.\overline{\pi_j} \Rightarrow \pi' \in \Theta_1$:
    Then from consistency of K_1 we know that Θ_1 is well-formed,
    and so $\overline{\Gamma_1, \overline{T_i} \vdash_{\text{WF}} \pi_j}$ and $\overline{\Gamma_1, \overline{T_i} \vdash_{\text{WF}} \pi'}$.
    Moreover, suppose wlog that $\overline{\pi_j = \tau_j : D_j\,\langle\overline{\tau}\rangle}$. Then $\overline{\Gamma_1 \vdash D_j\,\langle\overline{\tau}\rangle}$.
    By IH, we get $\overline{\Pi \mid \Theta_1 \Vdash K_4 \rhd \delta_{\Gamma 1}([\overline{\tau_i/T_i}]\pi_j)}$.
    Using lemma C.3 we get $\delta_{\Gamma 1}([\overline{\tau_i/T_i}]\pi') = [\overline{\delta_{\Gamma 1}(\tau_i)/T_i}]\pi'$ and
    $\overline{\delta_{\Gamma 1}([\overline{\tau_i/T_i}]\pi_j) = [\overline{\delta_{\Gamma 1}(\tau_i)/T_i}]\pi_j}$.
    Then using (cons) we have $\Pi \mid \Theta_1 \Vdash K_4 \rhd \delta_{\Gamma 1}(\pi)$ as required.
  - **Case** $\forall\overline{T_i}.\overline{\pi_j} \Rightarrow \pi' \in \Theta_2$: Wlog, let $\pi' = \tau : D\,\langle\overline{\tau_n}\rangle$.
    From assumptions we have $\Gamma_1 \vdash D\,\langle\overline{T_i}\rangle$.
    There is some crate K such that
    $\Pi \vdash K_2 \sqsubseteq K$ and $\Pi \nvdash K_1 \sqsubseteq K$ and $\forall\overline{T_i}.\overline{\pi_j} \Rightarrow \pi' \in \mathbf{lcenv}\,(\Pi, K)$.
    From consistency of K_2, we know π' satisfies orphan rules in K_2.
    Specifically, $\Pi \vdash_{\text{OR}} K \rhd \pi'$.

From lemma C.2, we get either:
ltenv $(\Pi, K) \vdash D$ or $\Pi \vdash_{\mathrm{LT}} K \rhd \tau'$ for some $\tau' \in \{\tau, \overline{\tau_n}\}$.
ltenv $(\Pi, K) \vdash D$ is not possible because **ltenv** $(\Pi, K) \subseteq \Gamma_2$ and $\Gamma_1 \vdash D$ and we assume there are no overlapping identifiers between Γ_1 and Γ_2.
If $\Pi \vdash_{\mathrm{LT}} K \rhd \tau'$ for some $\tau' \in \{\tau, \overline{\tau_n}\}$ then by lemma C.9, $\Pi \vdash_{\mathrm{LT}} K \rhd [\overline{\tau_i/T_i}]\tau'$ and by lemma C.8, $\delta_{\Gamma 1}([\overline{\tau_i/T_i}]\tau') \in$ BLANKETTYPE.
Then we can use either rule (blanket1) or (blanket2), depending on the position of τ' in π, to prove $\Pi \mid \Theta_1 \Vdash K_4 \rhd \delta_{\Gamma 1}(\pi)$ as required.

- **Case** (blanket1):
  Simple induction on the structure of B shows that $\delta_{\Gamma 1}(B) \in$ BLANKETTYPE for any B.
- **Case** (blanket2):
  Same argument as for (blanket1).
- **Case** (orphan):
  We want to show that if $\Pi \vdash_{\mathrm{OR}} K_4 \vdash \delta_{\Gamma 1}(\pi)$ then $\Pi \vdash_{\mathrm{OR}} K_4 \vdash \pi$, which holds by lemma C.12, so if $\Pi \nvdash_{\mathrm{OR}} K_4 \vdash \pi$ then $\Pi \nvdash_{\mathrm{OR}} K_4 \vdash \delta_{\Gamma 1}(\pi)$.

- **Case** 3: $\Pi \vdash K_3 \sqsubseteq K_4$. Analogous to case 2.

□

Theorem 8.3 (Crate extensibility)**.** *Let K be a consistent crate under Π where $\Pi = \Pi'' \cup \{K : \langle \{\overline{K_a}\}, \Gamma_K, \Theta_K \rangle\}$. Then let $\Pi' = [K'/K]\Pi'' \cup \{K' : \langle \{\overline{K_a}\}, \Gamma_K, \Theta_K \cup \{\theta_1\} \rangle\}$ where K' is consistent in Π' and $\theta_1 = \forall \overline{T_k}.\overline{\pi_j} \Rightarrow \tau_1 : D\ \langle \overline{\tau_{1n}} \rangle$ with $\tau \notin$ BLANKETTYPE for each $\tau \in \{\tau_1, \overline{\tau_{1n}}\}$. Then, for each crate K_1 such that $\Pi' \vdash K_1 \sqsubseteq K'$, if K_1 is consistent under Π, then it is also consistent under Π'.*

Proof. We prove this by induction on the crate dependency relation, assuming in the induction hypothesis that K_2 is consistent for all K_2 such that $\Pi' \vdash K_2 \sqsubset K_1$ (i.e., $\Pi \vdash K_2 \sqsubseteq K_1$ and $K_1 \neq K_2$). The only interesting thing we have to check for is overlap of constraint schemes in **cenv** (Π', K_1). We split the proof into two cases:

- **Case** 1: We want to show that θ_1 does not overlap with any other $\theta_2 \in$ **lcenv** (Π', K_1). The property is already guaranteed if $\Pi' \vdash K' \sqsubseteq K_1$ from the consistency of K' under Π'. We then consider the case where $\Pi' \nvdash K' \sqsubseteq K_1$.

  Let $\theta_2 = \forall \overline{T_l}.\overline{\pi_m} \Rightarrow \tau_2 : D\ \langle \overline{\tau_{2n}} \rangle$ and
  let ϕ be a type substitution such that $\phi(\tau_1 : D\ \langle \overline{\tau_{1n}} \rangle) = \phi(\tau_2 : D\ \langle \overline{\tau_{2n}} \rangle)$.
  From consistency of K_1 under Π, $\tau_2 : D\ \langle \overline{\tau_{2n}} \rangle$ satisfies the orphan rule under Π.
  By lemma C.14, the constraint also satisfies the orphan rule under Π'.

So either $\mathbf{ltenv}(\Pi', K_1) \vdash D$ or there is some type $\tau \in \{\tau_2, \overline{\tau_{2n}}\}$ s.t. $\Pi' \vdash_{\text{LT}} K_1 \rhd \tau$.
From lemma C.1 and consistency of K' we know that
$\mathbf{tenv}(\Pi', K') \vdash D$ and $\mathbf{tenv}(\Pi', K') \vdash_{\text{WF}} \tau'$ for each $\tau' \in \{\tau_2, \overline{\tau_{2n}}\}$.
However, from lemma C.6, $\mathbf{tenv}(\Pi', K') \vdash D$ is not possible.
Now suppose wlog that $\Pi' \vdash_{\text{LT}} K_1 \rhd \tau_2$.
Then $\tau_2 = \overline{\&} S\,\langle\overline{\tau}\rangle$ where $S\,\langle\overline{T}\rangle\{\ldots\} \in \mathbf{ltenv}(\Pi', K_2)$. Since we assumed that $\phi(\tau_1) = \phi(\tau_2)$, either $\tau_1 = \overline{\&} S\,\langle\overline{\tau'}\rangle$ for some $\overline{\tau'}$
or $\tau_1 = \overline{\&} T$ for some type variable T.
Since $\Pi' \vdash_{\text{LT}} K_1 \rhd \overline{\&} S\,\langle\overline{\tau'}\rangle$, we can use lemma C.5 to show that $\mathbf{tenv}(\Pi', K') \not\vdash_{\text{WF}} \overline{\&} S\,\langle\overline{\tau'}\rangle$.
On the other hand τ_1 cannot be equal to $\overline{\&} T$ as $\overline{\&} T \in$ BLANKETTYPE and that contradicts our initial assumption.
The same argument holds for any $\tau' \in \overline{\tau_{2n}}$.
Therefore, there is no possible substitution ϕ such that $\phi(\tau_1 : D\,\langle\overline{\tau_{1n}}\rangle) = \phi(\tau_2 : D\,\langle\overline{\tau_{2n}}\rangle)$ and so, θ_1 and θ_2 don't overlap.

- **Case** 2: We want to show that any $\theta_2, \theta_3 \in \mathbf{cenv}(\Pi, K_1)$ such that $\theta_2 \neq \theta_1$ and $\theta_3 \neq \theta_1$ don't overlap under Π' and K_1.
  Suppose some K_2, K_3 such that $\theta_2 \in \mathbf{lcenv}(\Pi', K_2)$ and $\theta_2 \in \mathbf{lcenv}(\Pi', K_3)$.
  Using theorem 8.2 we can conclude that if $K_2 \neq K_1$ and $K_3 \neq K_1$ then θ_2 and θ_3 don't overlap.
  We must then consider the case where $\theta_2 \in \mathbf{lcenv}(\Pi', K_1)$ and $\theta_3 \in \mathbf{lcenv}(\Pi', K_3)$ for some $K_3 \neq K_1$ (the proof for the opposite case is the same).
  Let $\theta_2 = \forall\overline{T_l}.\overline{\pi_m} \Rightarrow \tau_2 : D'\,\langle\overline{\tau_{2p}}\rangle$ and $\theta_3 = \forall\overline{T_q}.\overline{\pi_r} \Rightarrow \tau_3 : D'\,\langle\overline{\tau_{3p}}\rangle$.
  Suppose there is a type substitution ϕ such that $\phi(\tau_2 : D'\,\langle\overline{\tau_{2p}}\rangle) = \phi(\tau_3 : D'\,\langle\overline{\tau_{3p}}\rangle)$.
  Let $\Theta' = \mathbf{lcenv}(\Pi', K_1) = \Theta \cup \{\theta_1\}$.
  We want to show that for every $\pi \in \{\overline{\pi_m}, \overline{\pi_r}\}$, if $\Pi' \mid \Theta' \Vdash K_1 \rhd \phi(\pi)$
  then $\Pi \mid \Theta \Vdash K_1 \rhd \phi(\pi)$ from which we can derive a contradiction that no such ϕ exists and that θ_2 and θ_3 can't overlap in Π', Θ'.
  We proceed by induction on derivations $\Pi' \mid \Theta' \Vdash K_1 \rhd \pi'$ to show that $\Pi \mid \Theta \Vdash K_1 \rhd \pi'$.
  The only interesting case is (cons) where $\pi' = \overline{[\tau_i/T_i]}\pi$
  and $\forall\overline{T_i}.\overline{\pi_j} \Rightarrow \pi \in \Theta'$ and $\overline{\Pi' \mid \Theta' \Vdash K_1 \rhd [\overline{\tau_i/T_i}]\pi_j}$.
  If $\forall\overline{T_i}.\overline{\pi_j} \Rightarrow \pi \in \Theta$ then we can use IH to conclude
  $\overline{\Pi \mid \Theta \Vdash K_1 \rhd [\overline{\tau_i/T_i}]\pi_j}$ and $\Pi \mid \Theta \Vdash K_1 \rhd \pi'$.
  On the other hand, consider $\forall\overline{T_i}.\overline{\pi_j} \Rightarrow \pi = \theta_1$. Then $\pi = \tau_1 : D\,\langle\overline{\tau_{1n}}\rangle$.
  We will now show that $\Pi \not\vdash_{\text{OR}} K_1 \rhd \pi$.
  For contradiction suppose that $\Pi \vdash_{\text{OR}} K_1 \rhd \pi$.
  Then, by lemma C.2, either $\mathbf{ltenv}(\Pi, K_1) \vdash D$

or there is some $\tau \in \{\tau_1, \overline{\tau_{1n}}\}$ such that $\Pi \vdash_{\mathrm{LT}} K_1 \rhd \tau'$.
Since we assume that $\Pi \vdash K' \sqsubset K_1$, using lemma C.6 we get $\mathbf{ltenv}\,(\Pi, K_1) \nvdash D$.
Also, by lemma C.13 there is no $\tau \in \{\tau_1, \overline{\tau_{1n}}\}$ such that $\Pi \vdash_{\mathrm{LT}} K_1 \rhd \tau'$.
Then we can use (orphan) to prove $\Pi \mid \Theta \Vdash K_1 \rhd \pi$ as required.

□