Syntactic logical relations for System F with recursive types and call-by-name semantics

MSc Thesis *(Afstudeerscriptie)*

written by

Tex Aston Felix Schönlank
(born 1998-12-13 in Levallois-Perret, France)

under the supervision of **dr. Benno van den Berg** and **prof. dr. Herman Geuvers**, and submitted to the Examinations Board in partial fulfillment of the requirements for the degree of

MSc in Logic

at the *Universiteit van Amsterdam.*

Date of the public defense:	**Members of the Thesis Committee:**
August 27, 2020	dr. Ekaterina Shutova (Chair)
	dr. Benno van den Berg (Supervisor)
	prof. dr. Herman Geuvers (Daily supervisor)
	dr. Bahareh Afshari
	dr. Piet Rodenburg

Institute for Logic, Language and Computation

Abstract

We consider the second-order typed lambda calculus with full, or impredicative, polymorphism—i.e. System F—and contravariant iso-recursive types. (This language also appears under the names $\lambda\forall\mu$ and F^{μ}.) We equip it with call-by-name operational semantics.

Appel and McAllester devised a logical relations-based model for this language—minus the polymorphism—under call-by-value semantics.[4] Their goal was to obtain a relation that was sound w.r.t. contextual equivalence. The problem of type recursion was addressed by introducing a step index.

Ahmed expanded upon Appel-McAllester's work on these step-indexed logical relations by adding universally and existentially quantified types.[3, 16] They also adapted the model and proved its transitivity, soundness, and completeness.

We in turn adapt part of Ahmed's work to our language: we leave out the existential types and transpose the definitions and theorems from Ahmed's call-by-value to our call-by-name semantics. Transitivity of our semantic model remains an open problem and no attempt is made to obtain completeness. The key contribution of this thesis, then, is an analogue of Ahmed's soundness proof. We show that our semantic model for a call-by-name language is sound w.r.t. contextual equivalence. This yields the following proof technique in our language: in order to prove that two terms or programs e, e' are contextually equivalent, one need only show that they are related in the semantics.

The use of this proof technique is further discussed. Free theorems in the style of Wadler are investigated.[18] The obvious theorems turn out not to hold because of non-termination, a product of contravariant recursive types. We find that we can manually adjust the statements for non-termination such that they *do* hold, but find no general means of making these adjustments.

Acknowledgements

Mama, Papa, and Broer, thank you. (Since long before the Master of Logic, my mother and I joke that if I ever write acknowledgements for something important, she deserves to be mentioned right at the top. The moment has come and as it turns out, I am not joking.) Although to this day you don't really know what my thesis is about,[1] I could always count on your involvement. When I felt frustrated you supported me, when I needed help you advised me, when I felt MoL-unworthy you corrected me, when I was planning my further career you thought along with me, and when I wanted to share my love for a theorem you walked around the kitchen with me until you understood. My thanks go to you more than to anyone.

Maximilian, thank you for becoming my first friend in Amsterdam. I thought the best I would get out of Information theory would be a proof of Shannon's theorem, but it turned out to be you. After class—or the collaboration session or whatever—we walked to your home in Noord and its ever charming owner. Having had an unpleasantly turbulent first period, I smiled all the way home as I knew that Amsterdam had at least yielded a friend.

The intensity of the MoL made it a very welcome fact that I had just befriended the inventor of napping and doing nothing. I learned many things from you, those being only two of them. I will also fondly remember our hallway conversations that, through the magical words "let's go for a walk", were turned into midgnight discussions in the Outside on whether wheat sponges should doch or not be considered bread.

Jori, thank you for becoming my second friend in Amsterdam. We had spoken quite a few times—on the rare occasions that you appeared at the ILLC—but it was not until the fourth week of Set theory that we became friends. It was our shared dislike of cooperation that led us to cooperate. This resulted in a friendship based on: minimising cooperation on homework; clæssic memes; observations about Salsa Shop employees; the Kerguelen Islands; schemes, often revolving around migration and citizenship; mysteries involving my neighbours. I would have preferred a world with no pandemic, where we would have explored more of Amsterdam together. But whatever it takes, I know I can make it through.

Co-ouders and Ckasper, thank you for letting me go literally, and not letting me go at all figuratively. Whether or not to leave Leuven behind was my biggest life decision thus far, and you listened to all my doubts and worries, before and after I left. You are the most prominent reason that I feel at home in Leuven. I feared things would be unpleasantly different between us everytime I would see diens again, but it has become clear that even though we don't meet that often, what we have is probably for life.

Benno, thank you for being a good academic mentor. I don't know what my MoL career would have looked like without your help, but it almost certainly wouldn't be as good as it turned out now. You helped me pursue my interests by undertaking projects outside of the ILLC, something I would probably not have achieved without you. You thought along with my plans and introduced me to...

Herman, thank you for being a good thesis supervisor. You were not only approachable and a nice person, but also thorough and good at understanding and answering my more vague or hypothetical questions. You did a good job of introducing me to the field. Most importantly, you taught me why doors sometimes miss their top corner.

Tanja, I wish I could tell you what I thank you for, but you helped me with so many things I am afraid they don't form a set but a proper class. You helped me register at and apply for many different things and you have stamps and signed forms that make all the angry bureaucratic spirits go away. You also knew me way before I knew you, which is impressive because there are

[1] In fact, you don't even really know what my *master* is about. Then again, looking at the courses I took the past two years, I must admit that even *I* don't really know what my master is about.

≈ 90 of us and there's only one of you.

Then there are some people and circumstances that I am thankful to and for, respectively. I will not address those people (or those circumstances) in the second person, since they will almost certainly not read this.

I am thankful for finding the Any Blue choir. The degree to which I felt at home in Amsterdam increased dramatically after joining and I think the correlation was causal. I believe there is no purpose for me in life if I don't sing.

Mi estas dankema al Klára, pro ria Esperanta naskiĝtagfesto, kiu estis ege memorinda kaj sen kiu mi kredeble neniam lernus Esperanton.

I am thankful for Pati Pati's asking me back—twice—on their rehearsal weekends. It eased my departure from Leuven as they too make me feel at home there.

Finally, I am thankful to *de Heks*, *de Rokers*, *de Nachtwinkel*, *de Andere heks*, *de Zeeman*, *de Onzedelijken*, and *de Rijko's*. Their behaviour and sometimes their mere presence lit up my otherwise occasionally lonely Amsterdam life. A special word of thanks goes to *de AirBnB'ers* of course. Though I followed their story from the beginning, many questions remain unanswered. Perhaps I will never know where the five guitars went, what photos they were developing on the floor, whereto the carved wooden door leads, or if those things really were enormous waffle irons.

I conclude with an anecdote. In the second year of my bachelor of informatics in Leuven, when I did know I wanted to study something else somewhere else, but not what or where, my mother randomly came up to my room one day and said: "Hey, I found a master in Amsterdam that you might find interesting. It's called the *Master of Logic*." It took a year of hesitation, a few days of sitting in on Category theory, Set theory, and Modal logic lectures, and one encounter with a person with the Hilbert curve tattooed on their forearm for me to decide I wanted to do the MoL. It was a good decision.

Contents

Chapter 1

Introduction

1.1 Motivation and situation

The concept of *equivalence* of computer programs is an important one. For instance, in a complex program consisting of multiple separate modules, a suitable notion of program equivalence might allow the programmer to replace an existing module with a more efficient one, without breaking the intercompatibility. Another example might be a compiler, which translates the source into the target program. Knowledge of program equivalence can help reason about the compiler's behaviour and correctness.[3] The ultimate goal is then to have the source program be equivalent to the target, possibly via a chain of equivalences between intermediate representations.

There are multiple different notions of program equivalence.[10, p. 246] Some of them are based on the language's operational semantics, while others involve denotational semantics and mathematical apparatus like cartesian-closed categories and domain theory.[11, 1, 15, 13] We will concern ourselves with the operational notion of *contextual equivalence.* (The notation for the contextual equivalence of two terms e, e' is $e \approx^{ctx} e'$.) Loosely speaking, a *context* can be seen as a program with one "hole" or "slot" into which another program must be placed in order to obtain a finished program. Two programs A and B are then considered contextually equivalent iff inserting A into any suitable context results in a program with no observable differences from inserting B into the same context.[10, p. 249] Of course, the meaning of *suitable context* and *observable difference* are to be made precise.

Proving two terms' operational contextual equivalence directly is rather difficult, because of the universal quantification over all suitable contexts.[10] Even when the set of suitable contexts is defined inductively, the obvious attack route of proof by induction can turn out fruitless. The standard solution is to devise sound (and sometimes complete) proof techniques for contextual equivalence by showing that an easier-to-prove property is a sufficient condition for contextual equivalence. Some of these techniques and properties are syntactic in nature and some semantic. Among them are techniques involving bisimulations [17], domain theory [13], admissible relations, and biorthogonality or $\top\top$-closure.[12]

In this thesis, we apply one of these known techniques, namely *logical relations*, which are term relations defined by induction on the terms' types. We then prove that our logical relation is indeed sound w.r.t. contextual equivalence, i.e. relatedness of two terms under the logical relation implies their contextual equivalence. In this, we follow a line of work by, amongst others, Appel and McAllester and Ahmed on *syntactic* logical relations—syntactic as opposed to relations involving abstract semantics and, with it, significant mathematical structures.[4, 3, 7, 2] The relevant (programming) language is the second-order lambda calculus with full polymorphism and contravariant iso-recursive types. We will refer to this language as $\lambda\forall\mu$.

The central contribution of this thesis, then, is that the language for which we create the logical relation differs crucially from those of Appel-McAllester and Ahmed: our language has *call-by-name* (CBN) instead of *call-by-value* (CBV) operational semantics. Though this is not the only difference, it is the most crucial one. Informally, in CBV semantics, a function application is deferred until the argument is fully evaluated, i.e. evaluated to a *value*. Values form the well-defined subset of terms of which we declare they count as valid program results (such as booleans or integers, in some programming languages). In CBN semantics, a function application is reduced as soon as the left-hand side, i.e. the function, has taken the shape of a lambda abstraction. The occurrences of the lambda's variable are then all replaced by the unevaluated argument, as if they were not replaced by the intended argument (a value) but only by a token or *name* for it.

We replicate Ahmed's development of a semantics based on a sound syntactic logical relation for $\lambda\forall\mu$,[3] which we will call $\diamond$. Due to the differences between Ahmed's language and ours, we must redo Ahmed's work from the ground up, making several crucial changes along the way. The bulk of the work of this thesis was in getting the definition of this semantics right.

Recursive types preclude the well-foundedness of logical relations defined inductively on types. We avert this problem using the same technique as Appel-McAllester and Ahmed: we stratify the relation by parametrising the relation over a *step-index* and inducing on this index instead of on the terms' types. However, while Ahmed goes on to also prove the potentially desirable property of completeness w.r.t. contextual equivalence,[3] we content ourselves with mere soundness. In fact, the author believes completeness does not even hold for the logical relation defined in this thesis. Also, Appel and McAllester intended for their relation to be transitive but did not prove it. Part of Ahmed's contribution was to show the obvious proof attempt strands, and to "repair" the definition and prove the resulting transitivity. We make no attempt at transitivity. Our relation $\diamond$ suffers from roughly the same problem as Appel-McAllester's, namely that no proof of transitivity is known to the author (although no counterexample has been found either).

We then go on to prove soundness of $\diamond$ w.r.t. $\approx^{ctx}$. Finally, we put our relation to use and investigate by example which terms are related by $\diamond$ and which are not. We regard a free theorem in the style of Wadler [18]. We find that not even the simplest free theorem, stating that all terms of type $\forall\alpha.\alpha \to \alpha$ are essentially the identity function, holds. This is due to the possibility of non-termination introduced by recursive types. Such problems were predicted by Pierce.[10] We do, however, find an alternative translation of free theorems to our semantics that account for non-termination and thus *do* hold in our semantics. Unfortunately, these alternative translations require some manual work. We have not found a uniform formalisation deriving such alternative translations from Wadler-style free theorems.

1.2 Structure of this thesis

As mentioned, we closely follow the work of Ahmed.[3] In Chapter 2, we define the set of terms, its type system $\vdash$, and its operational semantics: call-by-name, explicitly typed lambda calculus with full, impredicative polymorphism and contravariant iso-recursive types, where folded-up types are considered values. This differs from Ahmed's language in the following ways. First, as mentioned, Ahmed's language uses call-by-value semantics. Second, our types are explicit or "Church-style". This contrasts with Ahmed's implicit or "Curry-style" types, where term-level (type-level) lambda abstractions mention no type (type variable) and type-level function application requires no type to which to instantiate. Third, Ahmed defines term *foldings* (a notion related to recursive types) to be evaluation contexts just like term *unfoldings*. This leads to Ahmed's language being non-deterministic. We make things slightly easier for ourselves by regarding folded-up terms as values, which yields determinism.

In Chapter 3, we set ourselves the goal of proving our language is *type-safe*. A term e is safe iff every evaluation path starting from e ends in a term that is either a value or itself reducible. (Informally, e "never gets stuck".) Type-safety then means that all typeable terms are safe. Note that this is not the same as *strong normalisation*, which says that every evaluation path starting at a typeable term is finite.

To prove type-safety, we define a unary logical relation $\vDash$ similar to Ahmed's, by a mutually recursive definition of sets parametrised over a step-index. It relates the necessary (type) environments Δ and Γ, a term e, and a type τ. We state the *fundamental property* of $\vDash$, namely that every well-typed term is in $\vDash$. In symbols: $\vdash \subseteq \vDash$ or $\Delta; \Gamma \vdash e : \tau \Rightarrow \Delta; \Gamma \vDash e : \tau$. We prove this by induction on the typing rules, with one *compatibility lemma* per rule. The second and last part of the proof is then to show that whenever a (closed) term e is in $\vDash$ with empty environments, e is safe. In symbols: $\emptyset; \emptyset \vDash e : \tau \Rightarrow \mathit{safe}\, e$. Combined with the first part, we then get that all typeable terms e are safe: $\emptyset; \emptyset \vdash e : \tau \Rightarrow \mathit{safe}\, e$. Note that this is the definition of type-safety.

In Chapter 4, we build up to and prove the main theorem of this thesis, using an argumentation structure similar to Ahmed's proof of soundness.[3] After some examples to build up intuition for the concept, we rigorously define the contextual equivalence relation $\approx^{ctx}$ for terms. This includes a formalisation of the notions of *suitable context* and *observable difference* mentioned earlier. We then define a second logical relation $\diamond$, similar to $\vDash$ from Chapter 3 but expecting two terms. The theorem we want to prove, then, is the soundness of the relation w.r.t. contextual equivalence.

In the first part of the proof, we show that $\diamond$ is reflexive on well-typed terms: $\vdash \subseteq \diamond$ or $\Delta; \Gamma \vdash e : \tau \Rightarrow \Delta; \Gamma \vDash e \diamond e : \tau$. We do this by induction on the rules of $\vdash$, in a way similar to our proof in Chapter 3.

In the second part, we deviate slightly from Ahmed's argumentation structure: we introduce the concept of *monotonicity* for contexts. We prove by induction that all suitable contexts are monotonic, reusing the compatibility lemmas from the first part of the proof. The theorem is then obtained by a direct proof of soundness, using the reflexivity of $\diamond$ and monotonicity of suitable contexts.

The final section, then, discusses some aspects of $\diamond$. Among other things, we revisit the example term pairs from the first section of this chapter and formally prove or refute their contextual equivalence. We also discuss two standard free theorems and the adjustments necessary for them to hold in our semantics.

1.3 Conventions and notation

The following is a list of some conventions and notations that *might* deviate from the standard notation to which the reader is used.

- In this thesis, we use "-" as a "hole" to define ad-hoc functions. For example, in Chapter 3, we will encounter a set denoted $\mathcal{V}^k_\Delta[\![\tau]\!]\delta$. The notation $\mathcal{V}^-_\Delta[\![\tau]\!]\delta$ is used to refer to the function that maps k onto $\mathcal{V}^k_\Delta[\![\tau]\!]\delta$. This practice requires some understanding of context, since strictly speaking, the domain and codomain of the function are not clear.

- Another note on functions: the application of functions to arguments is mostly written without parentheses around the arguments. The notation $f\ x$ is used for what the reader might be used to writing as $f(x)$. This is true not only for terms in the calculus, e.g. $(\lambda x : \tau.\, e)\ e'$, but also for functions in the mathematical realm. For example, in the thesis we will encounter mathematical operations that will be applied to terms. The subsequent application of such operations γ and δ (of which the reader need not yet know the meaning) to a term e is not written $\delta(\gamma(e))$ but $\delta\ (\gamma\ e)$ or even $(\delta \circ \gamma)\ e$. (Note that δ and γ are not part of the term string, they are mathematical operations that transform the term.)

- The notation $A \to B$ is used to denote the set of functions from A to B. Consequently, we speak of functions as being elements of such sets and write $f \in A \to B$ instead of $f : A \to B$. For sets of all partial functions we use $\rightharpoonup$, for finite partial functions $\rightharpoonup^{\mathrm{fin}}$.

  These arrow operators associate to the right. An example: $f \in A \rightharpoonup B \to C$ means that f is a partial function mapping some elements $a \in A$ to total functions from B to C. We also let function application associate to the left. This allows for easy expression of multivariable functions using “currying”: $f\ a\ b$ is the element of C to which b is mapped by $(f\ a) \in B$.

- Tuples are written with angle brackets: $\langle t_1, t_2, \ldots, t_n \rangle$.

- When X is a binary relation, X^* denotes the reflexive transitive closure of X. When X is any other set, X^* denotes the Kleene closure of X (the set of all finite words over X).

- Zero is a natural number.

- For brevity, we use logical symbols in the meta-language.

  Universal quantification in logical formulas and the meta-language is written $\forall x \in A, y_1, y_2 \in B, \ldots : \phi(x, y_1, y_2)$. When A, B are clear, we sometimes leave them out.

  Logical implication is written $\Rightarrow$ instead of $\to$, to avoid confusion with mathematical function sets (e.g. $\mathbb{N} \to \mathbb{N}$)) and the type constructor (e.g. $\tau_1 \to \tau_2$). Also, $\Rightarrow$ binds less strongly than $\wedge, \vee, \neg$, while the (meta-)quantifiers $\forall, \exists$ bind least strongly of all.

- The following are some association rules for type and term strings.

  In general, application of mathematical functions to components of term or type strings bind most strongly: $\delta\ \tau \to \sigma$ is the same as $(\delta\ \tau) \to \sigma$. The only exception is formed by single substitutions, e.g. $\gamma\ e[e'/x]$ is the same as $\gamma\ (e[e'/x])$.

  The scope of $\forall$ and μ extends as far to the right as possible, e.g. $\forall \alpha.\tau \to \mu\beta.\beta \to \alpha$ is the same as $\forall \alpha.(\tau_1 \to \mu\beta.(\beta \to \alpha))$.

  The same holds true for λ and Λ: $\lambda x\!:\!\tau.\ \mathtt{in}\ e\ \Lambda\alpha.e'\ e''$ is the same as $\lambda x\!:\!\tau.\ (\mathtt{in}\ e\ (\Lambda\alpha.(e'\ e'')))$.

Chapter 2

Language

In this chapter, we define the essential notions in order to be able to speak of a typed lambda calculus. We do not yet concern ourselves with the results that we will prove in Chapter 3 and Chapter 4. We start the next section by defining the language's syntactical elements: we specify what terms and types in our language are. Simple as this sounds, it will require a bit of attention to detail. In lambda calculus, there is a reduction rule (namely β-reduction) which, when not defined carefully, displays unintended behaviour. We will explain and make use of the standard solution to this problem, namely the Barendregt convention.[5]

The second section, then, continues the discussion of types and terms by discussing their relation. It defines how terms are assigned a type.

The subsequent section defines the operational semantics of the language, i.e. how lambda terms reduce to others. We prove an important property of the language that will be used in the rest of the thesis, namely that our operational semantics is deterministic.

The last section contains a discussion of some language properties that we can already observe without bringing in the step-indexed logical relations of the next chapters. We will discuss recursion, diverging terms, unicity of types, and the differences between our language and the one defined by Ahmed.[3]

2.1 Syntax: types and terms

We start by defining types and terms syntactically, as strings. We then introduce a few of the rules in the operational semantics and show that a naive following of these rules would lead to problems. An equivalence relation between type strings is established, as well as one between term strings. This allows us to *redefine* types and terms as equivalence classes of type strings and term strings, respectively. For the sake of simplicity and readability, we opt for carefully delineated abuse of notation. Despite types and terms being classes, we will talk about them as if they were strings. The *Barendregt convention*—Barendregt themself referred to it as the *variable convention*—then, is to refer to these strings by specific representatives. This will avoid the problems created by the naive rules. Readers familiar with this problem can skip Section 2.1.1, and those familiar with the Barendregt convention can skip Section 2.1.2 as well.

As mentioned in the introduction, the language we will be working with is the fully polymorphic lambda calculus with recursive types. This means that the set of types must be closed w.r.t. functions, universal quantification, and type folding. In contrast with Ahmed's language[3], ours uses explicit, Church-style typing, which means that term-level lambda abstractions mention the term's expected type, type-level abstractions mention the type variable, and type applications mention the type to which the term is applied.

Definition 1 (Types as strings). *We define the set Type of type strings inductively using the following grammar:*

$$\tau ::= \alpha \mid \tau \to \tau \mid \forall\alpha.\tau \mid \mu\alpha.\tau$$

where α ranges over a countably infinite set TVar of type-level variables. Throughout this document, unless noted otherwise, we let $\alpha, \beta \in \mathit{TVar}$ and we let $\tau, \sigma \in \mathit{Type}$.

Definition 2 (Terms as strings). *We define the set Term of term strings inductively using the following grammar:*

$$e ::= x \mid \lambda x\!:\!\tau.\, e \mid e\ e \mid \Lambda\alpha.e \mid e\ \tau \mid \texttt{in}\ e \mid \texttt{out}\ e$$

where x ranges over a countably infinite set Var of variables. Throughout this document, unless noted otherwise, we let $x, y \in \mathit{Var}$, $e \in \mathit{Term}$.

In order to define the naive rules of the operational semantics, we first need to define the notions of free (type) variables in type and term strings and their corresponding naive substitutions. Type variables occur in type and term strings, while term variables only occur in term strings. We also speak of type- and term-level variables' individual occurrences in type or term strings as being free or bound.

Definition 3 (Free type variables). *We define two functions with the same name,* ftv. *(Since their domains are disjoint, this should not lead to confusion.)*

The first maps a type string onto its set of free type variables: $\mathrm{ftv} \in \mathit{Type} \to \mathcal{P}\ \mathit{TVar}$. *It is defined by recursion on the type string, as displayed in Fig. 2.1.*

The second maps a term string onto its set of free type variables: $\mathrm{ftv} \in \mathit{Term} \to \mathcal{P}\ \mathit{TVar}$ *and is defined by recursion on the term string. It is displayed in Fig. 2.1.*

We also speak of type variables' separate occurrences as being free *or* bound. *An occurrence of α in a type or term string is called a* bound occurrence *iff it appears within the scope of an appropriate binder, otherwise it is a* free occurrence. *For type strings, the appropriate binders for α are $\forall\alpha$ and $\mu\alpha$. For term strings, it is only $\Lambda\alpha$.*

Definition 4 (Syntactic type substitution). *We define what it means to substitute a type string for a free type variable, both in type strings and in term strings.*

Given two type strings τ, τ' and a type variable α. Then the type string $\tau[\tau'/\alpha]$ is defined recursively w.r.t. τ as displayed in Fig. 2.2.

Given a term string e, a type string τ' and a type variable α. Then the term string $e[\tau'/\alpha]$ is defined recursively w.r.t. e as displayed in Fig. 2.2.

Definition 5 (Free variables). *We define a function* fv *which maps a term string onto its set of free variables:* $\mathrm{fv} \in \mathit{Term} \to \mathcal{P}\ \mathit{Var}$. *It is defined by recursion on the term string, as follows:*

$$\begin{aligned}
\mathrm{fv}\, x &:= \{x\},\\
\mathrm{fv}\, \lambda x\!:\!\tau.\, e &:= \mathrm{fv}\, e \setminus \{x\},\\
\mathrm{fv}(e_1\ e_2) &:= \mathrm{fv}\, e_1 \cup \mathrm{fv}\, e_2,\\
\mathrm{fv}\, \Lambda\alpha.e &:= \mathrm{fv}\, e,\\
\mathrm{fv}(e\ \tau) &:= \mathrm{fv}\, e,\\
\mathrm{fv}\, \texttt{in}\ e &:= \mathrm{fv}\, e,\\
\mathrm{fv}\, \texttt{out}\ e &:= \mathrm{fv}\, e.
\end{aligned}$$

$$
\begin{aligned}
\operatorname{ftv}\alpha &:= \{\alpha\},\\
\operatorname{ftv}(\tau_1 \to \tau_2) &:= \operatorname{ftv}\tau_1 \cup \operatorname{ftv}\tau_2,\\
\operatorname{ftv}(\forall\alpha.\tau) &:= \operatorname{ftv}\tau \setminus \{\alpha\},\\
\operatorname{ftv}(\mu\alpha.\tau) &:= \operatorname{ftv}\tau \setminus \{\alpha\}.
\end{aligned}
\qquad
\begin{aligned}
\operatorname{ftv} x &:= \emptyset,\\
\operatorname{ftv} \lambda x{:}\tau.\, e &:= \operatorname{ftv}\tau \cup \operatorname{ftv} e,\\
\operatorname{ftv}(e_1\ e_2) &:= \operatorname{ftv} e_1 \cup \operatorname{ftv} e_2,\\
\operatorname{ftv} \Lambda\alpha.e &:= \operatorname{ftv} e \setminus \{\alpha\},\\
\operatorname{ftv}(e\ \tau) &:= \operatorname{ftv} e \cup \operatorname{ftv}\tau,\\
\operatorname{ftv} \mathtt{in}\ e &:= \operatorname{ftv} e,\\
\operatorname{ftv} \mathtt{out}\ e &:= \operatorname{ftv} e.
\end{aligned}
$$

Figure 2.1: The recursive definitions of the free type variable functions, which both are called ftv. Note how the ftv function on terms makes use of the ftv functions on types, but not the other way around. Therefore, the definitions are wellfounded.

$$
\begin{aligned}
\alpha[\tau'/\alpha] &:= \tau',\\
\beta[\tau'/\alpha] &:= \beta \text{ if } \alpha \neq \beta,\\
(\tau_1 \to \tau_2)[\tau'/\alpha] &:= \tau_1[\tau'/\alpha] \to \tau_2[\tau'/\alpha],\\
(\forall\alpha.\tau)[\tau'/\alpha] &:= \forall\alpha.\tau,\\
(\forall\beta.\tau)[\tau'/\alpha] &:= \forall\beta.\tau[\tau'/\alpha] \text{ if } \alpha \neq \beta,\\
(\mu\alpha.\tau)[\tau'/\alpha] &:= \mu\alpha.\tau,\\
(\mu\beta.\tau)[\tau'/\alpha] &:= \mu\beta.\tau[\tau'/\alpha] \text{ if } \alpha \neq \beta,
\end{aligned}
\qquad
\begin{aligned}
x[\tau'/\alpha] &:= x,\\
(\lambda x{:}\tau.\, e)[\tau'/\alpha] &:= \lambda x{:}\tau[\tau'/\alpha].\, e[\tau'/\alpha],\\
(e_1\ e_2)[\tau'/\alpha] &:= e_1[\tau'/\alpha]\ e_2[\tau'/\alpha],\\
(\Lambda\alpha.e)[\tau'/\alpha] &:= \Lambda\alpha.e,\\
(\Lambda\beta.e)[\tau'/\alpha] &:= \Lambda\beta.e[\tau'/\alpha] \text{ if } \alpha \neq \beta,\\
(e\ \tau)[\tau'/\alpha] &:= e[\tau'/\alpha]\ \tau[\tau'/\alpha],\\
(\mathtt{in}\ e)[\tau'/\alpha] &:= \mathtt{in}\ e[\tau'/\alpha],\\
(\mathtt{out}\ e)[\tau'/\alpha] &:= \mathtt{out}\ e[\tau'/\alpha]
\end{aligned}
$$

Figure 2.2: The recursive definitions of syntactic type substitution on types (left) and terms (right). Note how the definition for terms makes use of the one for types, but not the other way around. Therefore, the definitions are wellfounded.

We also speak of term variables' separate occurrences as being free *or* bound. *An occurrence of x in a term string is called a* bound occurrence *iff it appears within the scope of an appropriate binder, λx, otherwise it is a* free occurrence.

Definition 6 (Syntactic term substitution)**.** *We define what it means to substitute a term string for a (free) variable in another term string. Given two term strings e, e' and a variable x. Then the type string $e[e'/x]$ is defined recursively w.r.t. e as follows:*

$$\begin{aligned}
x[e'/x] &:= e',\\
y[e'/x] &:= y \textit{ if } x \neq y,\\
(\lambda x\!:\!\tau.\, e)[e'/x] &:= \lambda x\!:\!\tau.\, e,\\
(\lambda y\!:\!\tau.\, e)[e'/x] &:= \lambda y\!:\!\tau.\, e[e'/x] \textit{ if } x \neq y,\\
(e_1\ e_2)[e'/x] &:= e_1[e'/x]\ e_2[e'/x],\\
(\Lambda\alpha.e)[e'/x] &:= \Lambda\alpha.e[e'/x],\\
(e\ \tau)[e'/x] &:= e[e'/x]\ \tau,\\
(\texttt{in}\ e)[e'/x] &:= \texttt{in}\ e[e'/x],\\
(\texttt{out}\ e)[e'/x] &:= \texttt{out}\ e[e'/x].
\end{aligned}$$

We give a few examples of these concepts.

Example 7 (Free and bound variables and substitutions)**.** *In the following examples, the bound variables are underlined once and the corresponding binder in whose scope they appear are underlined twice. The free variables are not underlined.*

$$\begin{array}{lll}
\lambda x\!:\!\forall\underline{\underline{\alpha}}.\underline{\alpha}.\, \lambda\underline{\underline{y}}\!:\!\forall\underline{\underline{\alpha}}.\underline{\alpha}.\, \underline{y} & (\Lambda\underline{\underline{\alpha}}.\lambda\underline{\underline{x}}\!:\!\underline{\alpha}.\, \lambda y\!:\!\underline{\alpha}.\, \underline{x})\ \beta\ y & \\
\lambda x\!:\!\forall\underline{\underline{\alpha}}.\underline{\alpha}.\, \lambda\underline{\underline{x}}\!:\!\forall\underline{\underline{\alpha}}.\underline{\alpha}.\, \underline{x}\ y & (\lambda\underline{\underline{x}}\!:\!\alpha.\, \lambda y\!:\!\alpha.\, \underline{x})[\beta/\alpha]\ y & = (\lambda\underline{\underline{x}}\!:\!\beta.\, \lambda y\!:\!\beta.\, \underline{x})\ y\\
\lambda x\!:\!\forall\underline{\underline{\alpha}}.\underline{\alpha}.\, \lambda\underline{\underline{x}}\!:\!\alpha.\, \underline{x}\ \underline{x} & (\lambda y\!:\!\beta.\, x)[y/x] & = (\lambda\underline{\underline{y}}\!:\!\beta.\, \underline{y})
\end{array}$$

2.1.1 The problem

Remark 8. *Readers already familiar with the problem of naive substitution can skip this section.*

Armed with these purposefully naive definitions, we will define the minimal number of rules of the operational semantics necessary to uncover the problem. Let us suppose we have (at least) the following two rules in our language:

$$\frac{}{(\lambda x\!:\!\tau.\, e)\ e' \mapsto e[e'/x]}\ \mapsto_{\mathbf{app}} \qquad \frac{}{(\Lambda\alpha.e)\ \tau \mapsto e[\tau/\alpha]}\ \mapsto_{\mathbf{tapp}},$$

where of course the rules should be regarded as schemas parametrised in x, τ, e, e', α. We will consider two out of many possible example strings that display unintended behaviour.

Example 9. *First define these two term strings:*

$$\begin{aligned}
t &:= \lambda x\!:\!\tau.\, \lambda y\!:\!\tau.\, x,\\
T &:= \Lambda\alpha.\Lambda\beta.\lambda x\!:\!\alpha.\, x,
\end{aligned}$$

where τ does not really matter that much and can be taken to be, e.g., $\forall\alpha'.\alpha'$. The term t is to be interpreted as the function taking two arguments of type τ and returning the first one, ignoring the second. In other words, $t\ e_1\ e_2$ will evaluate to e_1. The reader must understand that in order to achieve this meaning of t, we would have been equally happy writing it down as $\lambda y\!:\!\tau.\,\lambda x\!:\!\tau.\,y$ or $\lambda a : \tau.\,\lambda b : \tau.\,a$. It is not the names of the variables that matter, but how they relate to one another. T, then, takes two type variables and returns the identity function $\lambda x : \alpha.\,x$ for the first type it was given. Here also, the value returned by T does not depend on the second (type) argument.

Now for the problematic part, which is similar to Example 7. Suppose we want to apply t to y, which is a valid term and might have some meaning to us in context. We follow the rule $\mapsto_{\mathbf{app}}$ and see that $t\ y = (\lambda x\!:\!\tau.\,\lambda y\!:\!\tau.\,x)\ y \mapsto (\lambda y\!:\!\tau.\,x)[y/x] = \lambda y\!:\!\tau.\,y$. We thus see that $t\ y$ evaluates to the identity function on τ. Thus $t\ y\ e_2 \mapsto (\lambda y\!:\!\tau.\,y)\ e_2 \mapsto e_2$, which contradicts the intuitive meaning we gave to t, whenever $y \neq e_2$.

The problem with the t in Example 9 lies in the fact that y, which has a meaning outside of $\lambda y\!:\!\tau.\,x$, is substituted for x and therefore its meaning is overwritten by the binder λy. In other words, the free y got "captured" by λy.

A similar thing happens when we apply T to β and some other type τ'. The intended meaning of T is such that $T\ \beta\ \tau'$ evaluates to $\lambda x\!:\!\beta.\,x$, but it turns out that $T\ \beta\ \tau' \mapsto (\Lambda\beta.\lambda x : \alpha.\,x)[\beta/\alpha]\ \tau' = (\Lambda\beta.\lambda x : \beta.\,x)\ \tau' \mapsto (\lambda x : \beta.\,x)[\tau'/\beta] = \lambda x : \tau'.\,x$, i.e. it evaluates to the identity function on τ' instead of on β. Again, the problem is the first step: the β that is substituted for α is captured by the binding $\Lambda\beta$ in the scope of which it appears.

A final remark: we have seen a term-level variable being captured during a term substitution in a term and a type-level variable being captured during a type substitution in a term. The same problem can arise in a type-level variable being captured during a term substitution in a term. We now consider the fourth and last substitution problem: capturing of a type-level variable during a type substitutino in a type.

Consider the type

$$\tau := \forall\beta.\alpha \to \alpha,$$

intuitively meaning the type of functions taking a type and ignoring it, returning a function from α to α. Suppose we want to substitute β for α in τ. We *want* the result to be the type of functions still ignoring their given type variable, now returning a function from β to β. However, $\tau[\beta/\alpha] = \forall\beta.\beta \to \beta$ does not match this description: functions of this type will not ignore the type they are given, as the type of their returned value depends on it. Once again, this problem is caused by capturing of β by the binder $\forall\beta$.

2.1.2 The solution

Remark 10. *In this section, we introduce the Barendregt convention. We expand it to typed lambda calculus and adapt it to types mentioned within terms and to types themselves. The reader may want to skip this section if at least one of the following holds: 1) the fairly experienced reader may want to skip this section, or 2) the reader is content with the explanation "we rename bound (type) variables in types and terms such that we never get into trouble".*

We observed that the problem with substitutions as defined so far is that they let free variables be captured upon substitution. Simply disallowing substitutions whenever capturing would occur is not an option, since, like we saw, such situations can arise and are legitimate. Instead, we follow the solution presented by Barendregt.[5, Chapter 2] [8] Since Barendregt's solution was written with untyped lambda calculus in mind, it covers only terms. We supplement it with an analogue for types.

We first define a binary equivalence relation between terms—for historical reasons, Barendregt calls this *α-convertibility*—indicating that one can be changed into the other by renaming bindings of type and term variables. Then we redefine *terms* to be equivalence classes of term strings under this relation, reserving *term string* for the old notion of terms. We then do a similar thing for types. The problem is then solved by the Barendregt convention, which installs requirements on the representatives of types and terms that we will keep implicit throughout the rest of this thesis. This allows us to have the best of both worlds by discussing representatives—which is less cumbersome than considering their classes—without worrying about incorrectness.

Definition 11 (Term equality up to bound (type) variable names)**.** *We define a binary relation $\rightsquigarrow_{\text{bv}}$ on term strings. We say that $e_1 \rightsquigarrow_{\text{bv}} e_2$ iff exactly one of the following holds:*

1. *There is some subterm $\lambda x\!:\!\tau.\, e_1'$ of e_1 and a "fresh" variable y not appearing in e_1' (neither bound nor free), such that e_2 is equal to e_1 except that $\lambda x\!:\!\tau.\, e_1'$ is replaced by $\lambda y\!:\!\tau.\, e_1'[y/x]$.*
2. *There is some subterm $\Lambda\alpha.e_1'$ of e_1 and a "fresh" type variable β not appearing in e_1' (neither bound nor free), such that e_2 is equal to e_1 except that $\Lambda\alpha.e_1'$ is replaced by $\Lambda\beta.e_1'[\beta/\alpha]$.*

We then say that e_1 and e_2 are equal up to bound (type) variables *iff $e_1 \rightsquigarrow^*_{\text{bv}} e_2$.*

We see that, in general, $\rightsquigarrow_{\text{bv}}$ is not reflexive, transitive, or symmetric. However, $\rightsquigarrow^*_{\text{bv}}$ is not only reflexive and transitive—it is so by definition—but also symmetric, and therefore an equivalence relation. We do not formally prove this, but make it plausible through Example 13.

Example 12. *We provide a few examples of situations where term equality up to bound (type) variable names does or does not hold. Note that most of the terms used are nonsensical. (Whenever two different symbols are used for variables, we will assume they are indeed different variables.)*

We see that $\lambda x\!:\!\tau.\, x\; x \rightsquigarrow_{\text{bv}} \lambda y\!:\!\tau.\, y\; y$.

The following equality does not hold because two substitutions have been made at once: $\lambda x\!:\!\tau.\, \lambda y\!:\!\tau.\, y \not\rightsquigarrow_{\text{bv}} \lambda x'\!:\!\tau.\, \lambda y'\!:\!\tau.\, y'$.

This variant also does not hold: $\lambda x\!:\!\tau.\, \lambda y\!:\!\tau.\, y \not\rightsquigarrow_{\text{bv}} \lambda x'\!:\!\tau.\, \lambda y'\!:\!\tau.\, x'$. This time, the error is that the variable y in third position was replaced by x' instead of by y'.

Regarding renaming of types within terms, we see that $\Lambda\alpha.\lambda x\!:\!\alpha.\, x\; \alpha \not\rightsquigarrow_{\text{bv}} \Lambda\beta.\lambda x\!:\!\alpha.\, x\; \beta$. The reason is that we must rename all appearances of α: not only the one in the type application $x\ \alpha$, but also the one appearing in the type annotation $x : \alpha$.

The last two examples show that the freshness condition is important. First, we see that $\Lambda\alpha.((\Lambda\beta.e)\; (e'\; \alpha)) \not\rightsquigarrow_{\text{bv}} \Lambda\beta.((\Lambda\beta.e)\; (e'\; \beta))$ since the choice of β as a new variable is not fresh. In this example, no real problem occurred, since the free variable α was replaced with β which happened still to be free. However, the following example shows how using a non-fresh variable can *in fact go wrong: $\Lambda\alpha.((\Lambda\beta.e\; \alpha)\; (e'\; \alpha)) \not\rightsquigarrow_{\text{bv}} \Lambda\beta.\Lambda\beta.((e\; \beta)\; (e'\; \beta))$. (In both examples, we can assume that e, e' contain no type variables for simplicity.)*

Example 13 ($\rightsquigarrow^*_{\text{bv}}$ is symmetric)**.** *Let us assume that $y \neq x$. We note that $\lambda x\!:\!\tau.\, \lambda x\!:\!\tau.\, x \rightsquigarrow_{\text{bv}} \lambda y : \tau.\, \lambda x : \tau.\, x$ but not the other way around. It would require that x appears neither free nor bound in $\lambda x\!:\!\tau.\, x$, which is not the case. However, we show that we are able to connect the two again in multiple steps, i.e. $\lambda y\!:\!\tau.\, \lambda x\!:\!\tau.\, x \rightsquigarrow^*_{\text{bv}} \lambda x\!:\!\tau.\, \lambda x\!:\!\tau.\, x$. We prove through a simple chain of $\rightsquigarrow_{\text{bv}}$ connections, where z is fresh variable equal neither to x nor to y:*

$$\begin{aligned}
&\lambda y\!:\!\tau.\, \lambda x\!:\!\tau.\, x\\
\rightsquigarrow_{\text{bv}}&\lambda y\!:\!\tau.\, \lambda z\!:\!\tau.\, z\\
\rightsquigarrow_{\text{bv}}&\lambda x\!:\!\tau.\, \lambda z\!:\!\tau.\, z\\
\rightsquigarrow_{\text{bv}}&\lambda x\!:\!\tau.\, \lambda x\!:\!\tau.\, x.
\end{aligned}$$

Definition 14 (Type equality up to bound type variable names)**.** *We define a binary relation* $\rightsquigarrow_{\mathrm{bvt}}$ *on type strings. We say that* $\tau_1 \rightsquigarrow_{\mathrm{bvt}} \tau_2$ *iff exactly one of the following holds:*

1. *There is some type string* $\forall\alpha.\tau_1'$ *occurring as a substring in* tau_1 *and a "fresh" type variable* β *not appearing in* τ_1' *(neither bound nor free), such that* τ_2 *is equal to* τ_1 *except that* $\forall\alpha.\tau_1'$ *is replaced by* $\forall\beta.\tau_1'[\beta/\alpha]$*.*

2. *The same holds, but with* $\forall$ *replaced by* μ*.*

We then say that τ_1 *and* τ_2 *are* equal up to bound type variables *iff* $\tau_1 \rightsquigarrow^*_{\mathrm{bvt}} \tau_2$.

Definition 15 (Types and terms as equivalence classes)**.** *We* redefine *Type to be the quotient of the set of type strings (i.e. the previous definition of Type) over the relation* $\rightsquigarrow^*_{\mathrm{bvt}}$ *of type equality up to bound type variable names.*

We similarly redefine Term to be the quotient of the set of term strings (i.e. the previous definition of Term) over the relation $\rightsquigarrow^*_{\mathrm{bv}}$ *of type equality up to bound (type) variable names.*

Convention 16 (Due to Barendregt)**.** *Although types are equivalence classes of type strings and similarly for terms, we will nonetheless refer to them using representatives. Within definitions, propositions, theorems, and similar contexts, all strings representative of types or terms mentioned in the context are implicitly assumed to be as follows. Every type variable* α *in a binder* $\forall\alpha$ *or* $\mu\alpha$ *in a type string, or in a binder* $\Lambda\alpha$ *in term string, is different from all type variables that occur free in any of the type or term strings mentioned in the context. Similarly, every term variable* x *in a binder* λx *in a term string is different from all term variables that occur free in any of the term strings mentioned in the context. Note that through a simple cardinality argument, such suitable representatives always exist as long as we have finitely many of them in any given context.*

Regarding the operations on types and terms that we have defined so far: those can be reinterpreted very well as operations on classes of strings instead of on strings, without reformulation. For example, it is clear that if $e \rightsquigarrow^*_{\mathrm{bv}} e'$, then $\mathrm{fv}\, e = \mathrm{fv}\, e'$ and similarly for ftv and for types. The problem that needed solving was that of variable capture. Indeed, when we look at, e.g., this excerpt from the definition of term substitutions:

$$(\lambda y\!:\!\tau.\, e)[e'/x] := \lambda y\!:\!\tau.\, e[e'/x] \text{ if } x \neq y$$

and think of Example 9 again, we see that the problem of capture arises only when $y \in \mathrm{fv}\, e'$. Fortunately, we now have the Barendregt convention that tells us that this cannot occur. In this definition of term substitutions, $\lambda y : \tau.\, e$ and e' are both to be seen as "term string mentioned in the context" of the definition. Therefore, the y in the binder λy must be different from term variables free in e', so $y \notin \mathrm{fv}\, e'$. Similar things hold for the other substitutions and their variable capture problems.

One can even go so far as to say that x is term string mentioned in the context, and therefore $y = x$ is impossible since $x \in \mathrm{fv}\, x$. This would imply that some cases in the substitution definitions can be omitted.

2.2 The type relation

In this section, we define which terms are to be considered of which type and under which circumstances. The fact that we are working in a polymorphic language requires us not only to use an environment to keep track of the types of term variables, but also a *type* environment for the type variables.

Definition 17 (Environment)**.** *We define Env to be the set of all finite partial functions from Var to Type:*

$$\mathit{Env} := \mathit{Var} \rightharpoonup^{\mathrm{fin}} \mathit{Type}.$$

Throughout this document, unless noted otherwise, we let $\Gamma \in \mathit{Env}$*. The empty environment, i.e. the* Γ *with* $\operatorname{dom}\Gamma = \emptyset$*, we denote* $\cdot$ *or* $\emptyset$*. When* $x \notin \operatorname{dom}\Gamma$*, we let* $\Gamma, (x : \tau)$ *denote the function that maps every* $x' \in \operatorname{dom}\Gamma$ *to* $\Gamma\ x'$ *and maps* x *to* τ*. Note that if* $x \in \operatorname{dom}\Gamma$*, then* $\Gamma, (x : \tau)$ *is not defined.*

Definition 18 (Type environment)**.** *We define TEnv to be the set of all finite subsets of TVar:*

$$\mathit{TEnv} := \mathcal{P}^{fin}\ \mathit{TVar}.$$

Throughout this document, unless noted otherwise, we let $\Delta \in \mathit{TEnv}$*. The empty type environment* $\Delta = \emptyset$ *is also denoted* $\cdot$*. When* $\alpha \notin \Delta$ *we let* Δ, α *denote the set* $\Delta \cup \{\alpha\}$*. Note that if* $\alpha \in \Delta$*, then* Δ, α *is not defined.*

We also introduce the notion that types and environments can be wellformed w.r.t. a type environment:

Definition 19 (Type wellformedness)**.** *We define the relation* $\Delta \vdash \tau$ *inductively:*

$$\frac{\Delta \ni \alpha}{\Delta \vdash \alpha} \qquad \frac{\Delta \vdash \tau_1 \qquad \Delta \vdash \tau_2}{\Delta \vdash \tau_1 \to \tau_2} \qquad \frac{\Delta, \alpha \vdash \tau}{\Delta \vdash \forall\alpha.\tau} \qquad \frac{\Delta, \alpha \vdash \tau}{\Delta \vdash \mu\alpha.\tau}.$$

We say that a type τ *is* closed *iff* $\cdot \vdash \tau$*. The set of all closed types is written CType.*

We then also say that

$$\Delta \vdash \Gamma :\Leftrightarrow\ \forall x \in \operatorname{dom}\Gamma : \Delta \vdash \Gamma\ x.$$

Remark 20 (Inductive definition with rules)**.** *The reader might not be used to inductive definitions using such rules. Informally, what is meant in Definition 19 is that* $\vdash$ *is the smallest relation over TEnv and Type satisfying, or* closed under*, the mentioned rules. More formally, it means that* $\vdash := \bigcap\{S \subseteq \mathit{Env} \times \mathit{Type} \mid \phi_1(S) \wedge \ldots \wedge \phi_4(S)\}$*, where* $\phi_i(S)$ *is a formula expressing that* S *is closed under the i-th rule. Such a formula is a rephrasing of the rule as an implication, where mentionings of* $\ldots \vdash \ldots$ *are replaced by* $\langle \ldots, \ldots \rangle \in S$ *and all variables are universally quantified. For example, the first three rules would become:*

$$\begin{aligned}
\phi_1(S) &:= \forall \Delta, \alpha : \Delta \ni \alpha \Rightarrow \langle \Delta, \alpha \rangle \in S,\\
\phi_2(S) &:= \forall \Delta, \tau_1, \tau_2 : \langle \Delta, \tau_1 \rangle \in S \wedge \langle \Delta, \tau_2 \rangle \in S \Rightarrow \langle \Delta, \tau_1 \to \tau_2 \rangle \in S,\\
\phi_3(S) &:= \forall \Delta, \alpha, \tau : \langle (\Delta, \alpha), \tau \rangle \in S \Rightarrow \langle \Delta, \forall\alpha.\tau \rangle \in S.
\end{aligned}$$

We also prove the following statement. Although it might seem quite obvious, we choose to prove it in full. It provides for a good introduction to the concept of proof by induction on rules *and* to applying the Barendregt convention. If the reader feels confident with both concepts, they can skip the proof. It provides no interesting further insight.

Lemma 21 (Characterisation of wellformedness of types)**.** $\Delta \vdash \tau \Leftrightarrow \operatorname{ftv}\tau \subseteq \Delta$.

Proof. We start with the proof that $\Delta \vdash \tau$ implies $\operatorname{ftv}\tau \subseteq \Delta$. We do so by induction on the wellformedness $\Delta \vdash \tau$ of τ. This means that we define a set $S := \{\langle \Delta, \tau \rangle \mid \operatorname{ftv}\tau \subseteq \Delta\}$ and prove that S is closed under all the rules by which the type wellformedness relation $\vdash$ is defined in Definition 19. The fact that $\vdash$ is defined inductively by them then means that, by definition, $\vdash \subseteq S$. This will result in the following—informally written—chain of reasoning: $(\Delta \vdash \tau) \Leftrightarrow (\langle \Delta, \tau \rangle \in \vdash) \Rightarrow (\langle \Delta, \tau \rangle \in S) \Leftrightarrow (\operatorname{ftv}\tau \subseteq \Delta)$.

- Suppose that $\alpha \in \Delta$. We must prove that $\mathrm{ftv}\,\alpha \subseteq \Delta$. This is clearly the case: $\mathrm{ftv}\,\alpha = \{\alpha\} \subseteq \Delta$ since $\alpha \in \Delta$.

- Suppose that $\langle \Delta, \tau_1 \rangle$ and $\langle \Delta, \tau_2 \rangle$ are elements of S. We must prove that $\langle \Delta, \tau_1 \to \tau_2 \rangle \in S$. Indeed, $\mathrm{ftv}(\tau_1 \to \tau_2) = \mathrm{ftv}\,\tau_1 \cup \mathrm{ftv}\,\tau_2 \subseteq \Delta$ by assumption.

- Suppose that $\langle (\Delta, \alpha), \tau_1 \rangle \in S$. (Note that the mentioning of (Δ, α) implicitly assumes that $\alpha \notin \Delta$. We must prove that $\langle \Delta, \forall \alpha.\tau_1 \rangle \in S$. We see that $\mathrm{ftv}\,\forall\alpha.\tau_1 = \mathrm{ftv}(\tau_1) \setminus \{\alpha\}$ and $\mathrm{ftv}(\tau_1) \subseteq (\Delta, \alpha)$, by assumption. Thus $\mathrm{ftv}(\tau_1) \setminus \{\alpha\} \subseteq \Delta$, which finishes the proof.

- The case for μ types is completely analogous to the previous case for $\forall$ types.

We now prove the other direction: $\mathrm{ftv}\,\tau \subseteq \Delta$ implies $\Delta \vdash \tau$, for all τ and Δ. This time, we cannot use a proof by induction on the type wellformedness relation, since it appears in the conclusion of the implication and not its premise. We also cannot induce on the statement $\mathrm{ftv}\,\tau \subseteq \Delta$. Therefore, we choose to induce on *the fact that τ is a type*. Remember that not only the type wellformedness definition is an inductive one, but the definition of *Type* is as well. More precisely, we perform an induction proof on $\tau \in \mathit{Type}$, within which we generalise over Δ. Formally, we again define a set $S := \{\tau \mid \forall \Delta : \mathrm{ftv}\,\tau \subseteq \Delta \Rightarrow \Delta \vdash \tau\}$, and prove that it is closed under the rules by which *Type* is defined.

- First, the type variable case. Let τ be any type variable α. We must prove that $\tau = \alpha \in S$. We therefore fix a Δ and suppose $\mathrm{ftv}\,\tau \subseteq \Delta$. We see that $\mathrm{ftv}\,\tau = \{\alpha\}$, and thus $\alpha \in \Delta$, which implies $\Delta \vdash \tau$ by definition.

- Now, we assume $\tau_1, \tau_2 \in S$ and $\mathrm{ftv}(\tau_1 \to \tau_2) \subseteq \Delta$. We must prove that $\Delta \vdash \tau_1 \to \tau_2$. We note that $\mathrm{ftv}\,\tau_1 \subseteq \Delta$ and $\mathrm{ftv}\,\tau_2 \subseteq \Delta$. By induction, then $\Delta \vdash \tau_1$ and $\Delta \vdash \tau_2$. This means, by definition, that $\Delta \vdash \tau_1 \to \tau_2$.

- (The following case is most interesting, because it involves the Barendregt convention.) Assume $\tau \in S$ and $\mathrm{ftv}(\forall\alpha.\tau) \subseteq \Delta$. We must prove that $\Delta \vdash \forall\alpha.\tau$. we note that $\mathrm{ftv}(\forall\alpha.\tau) = \mathrm{ftv}\,\tau \setminus \{\alpha\}$. Thus, $\mathrm{ftv}\,\tau \subseteq \Delta \cup \{\alpha\}$.

  For didactic purposes, we will temporarily forget the Barendregt convention. This misguided reasoning is written in *italics. Either $\alpha \in \Delta$ or $\alpha \notin \Delta$. In the first case, we apply the induction hypothesis $\tau \in S$ to* $\mathrm{ftv}\,\tau \subseteq \Delta \cup \{\alpha\} = \Delta$. *We get that $\Delta \vdash \tau$.* This is where we get stuck: we need $(\Delta, \alpha) \vdash \tau$ to arrive at the wanted conclusion $\Delta \vdash \forall\alpha.\tau$.

  However, the Barendregt convention tells us that, since $\forall\alpha.\tau$ was mentioned in the premise, α is a bound variable that must therefore be chosen different from all (free) variables in Δ. We conclude that $\alpha \notin \Delta$. The proof now follows smoothly: $\mathrm{ftv}\,\tau \subseteq \Delta \cup \{\alpha\} = (\Delta, \alpha)$ implies $(\Delta, \alpha) \vdash \tau$ and, finally, $\Delta \vdash \forall\alpha.\tau$.

- The case for μ types is completely analogous to the previous case for $\forall$ types.

□

We end this section with its major definition, the type relation.

Definition 22 (Type relation)**.** *We define a type relation $\vdash$.*[1] *A typing judgment takes the form $\Delta; \Gamma \vdash e : \tau$ and relates a type environment $\Delta \in \mathit{TEnv}$, an environment $\Gamma \in \mathit{Env}$, a term*

[1] Note that the symbol $\vdash$ is the same as the one used for type wellformedness. Since the relations have different arities, it should always be clear which one of the two relations is meant.

$e \in$ Term and a type $\tau \in$ Type. We define the relation inductively using the following rules:

$$\frac{\Delta \vdash \Gamma}{\Delta; \Gamma \vdash x : \Gamma\, x}\ \mathbf{:_{var}}$$

$$\frac{\Delta; \Gamma, (x : \tau_x) \vdash e : \tau_e}{\Delta; \Gamma \vdash \lambda x\!:\!\tau_x.\, e : \tau_x \to \tau_e}\ \mathbf{:_{abs}} \qquad \frac{\Delta; \Gamma \vdash e : \tau' \to \tau \qquad \Delta; \Gamma \vdash e' : \tau'}{\Delta; \Gamma \vdash e\ e' : \tau}\ \mathbf{:_{app}}$$

$$\frac{\Delta, \alpha; \Gamma \vdash e : \tau \qquad \Delta \vdash \Gamma}{\Delta; \Gamma \vdash \Lambda\alpha.e : \forall\alpha.\tau}\ \mathbf{:_{tabs}} \qquad \frac{\Delta; \Gamma \vdash e : \forall\alpha.\tau \qquad \Delta \vdash \tau'}{\Delta; \Gamma \vdash e\ \tau' : \tau[\tau'/\alpha]}\ \mathbf{:_{tapp}}$$

$$\frac{\Delta; \Gamma \vdash e : \tau[\mu\alpha.\tau/\alpha]}{\Delta; \Gamma \vdash \mathtt{in}\ e : \mu\alpha.\tau}\ \mathbf{:_{muin}} \qquad \frac{\Delta; \Gamma \vdash e : \mu\alpha.\tau}{\Delta; \Gamma \vdash \mathtt{out}\ e : \tau[\mu\alpha.\tau/\alpha]}\ \mathbf{:_{muout}}$$

Instead of $\cdot; \cdot \vdash e : \tau$ we occasionally write $\vdash e : \tau$ or even $e : \tau$, and say that e is well-typed *or* typeable*.*

We are now in the position to explain why we refer to our language as having *iso*-recursive types. The rules $:_{\mathbf{muin}}$ and $:_{\mathbf{muout}}$ state that e can be *folded* and *unfolded*, respectively, in order to fold and unfold its type. *iso*-recursion, then, is the term used for type systems like these, where this folding and unfolding must be done explicitly. Had the rules been such that e is of type $\tau[\mu\alpha.\tau/\alpha]$ iff it is of type $\mu\alpha.\tau$, without the need for `out` and `in`, we would have spoken of an *equi*-recursive type system.

We can also explain why we said our recursive types are *contravariant* in the introduction. Contravariant type recursion allows the relevant type variable to appear anywhere within the type. This contrasts with covariant type recursion, where the type variable can only appear *positively.* When viewing a type as a nested application of the type constructors $\to, \forall, \mu$ to type variables—or as a tree with type variables in the leaves and type constructors in the other nodes—then a variable occurs positively iff it appears on the left-hand side of an even number of $\to$ constructors. Otherwise, it occurs negatively. Looking at the type rules $:_{\mathbf{muin}}$ and $:_{\mathbf{muout}}$, we see that that we installed no positivity constraint. In the discussion in Section 2.4.2, the difference between covariant and contravariant type recursion will become more clear. For a thorough explanation involving type algebra, we refer to [6, Sec. 9C].

Finally, we clarify the last unexplained assertion made about the language's type system in the introduction. It was said that the language has "full, impredicative polymorphism". There are multiple type systems that are considered polymorphic, and *full*, *impredicative*, *strong*, or *System F-style* polymorphism is one of them.[8] It is contrasted with *weak* or *predicative* polymorphism, a system in which quantification over type variables can happen only at the "outside" of types. For example, $\forall\alpha.\alpha \to \alpha$ and $\forall\alpha.\forall\beta.\alpha \to \beta$ are considered types, but $\forall\alpha.\alpha \to \forall\beta.\beta$ is not, since the quantification $\forall\beta$ happens within a type constructor different from $\forall$. Our type system clearly does not have that restriction and is therefore considered impredicative.

Example 23. *We show an example derivation of the type of a term. We use the following abbreviations: $\tau_{\mathbb{B}} := \forall\alpha.\alpha \to \alpha \to \alpha$; $\Delta := (\beta)$; $\Gamma := (f : \beta, g : \beta)$.*

The term of which we will be deriving the type is $(\lambda x\!:\!\tau_{\mathbb{B}}.\, x\ \beta\ f\ g)\ (\Lambda\alpha.\lambda y\!:\!\alpha.\, \lambda z\!:\!\alpha.\, z)$. For *reasons of space, we first derive the type of the left-hand side:*

$$
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{\beta \vdash \Gamma, (x : \tau_{\mathbb{B}}))}{\beta; \Gamma, (x : \tau_{\mathbb{B}})) \vdash x : \tau_{\mathbb{B}}}\,\textbf{:var}
      \qquad \beta \vdash \beta
    }{\beta; \Gamma, (x : \tau_{\mathbb{B}})) \vdash x\ \beta : \beta \to \beta \to \beta}\,\textbf{:tapp}
    \qquad
    \dfrac{\beta \vdash \Gamma, (x : \tau_{\mathbb{B}}))}{\beta; \Gamma, (x : \tau_{\mathbb{B}})) \vdash f : \beta}\,\textbf{:var}
  }{\beta; \Gamma, (x : \tau_{\mathbb{B}})) \vdash x\ \beta\ f : \beta \to \beta}\,\textbf{:app}
  \qquad
  \dfrac{\beta \vdash \Gamma, (x : \tau_{\mathbb{B}}))}{\beta; \Gamma, (x : \tau_{\mathbb{B}})) \vdash g : \beta}\,\textbf{:var}
}{
  \dfrac{\beta; \Gamma, (x : \tau_{\mathbb{B}})) \vdash x\ \beta\ f\ g : \beta}{\beta; \Gamma \vdash \lambda x\!:\!\tau_{\mathbb{B}}.\, x\ \beta\ f\ g : \tau_{\mathbb{B}} \to \beta}\,\textbf{:abs}
}\,\textbf{:app}
$$

and then of the right-hand side:

$$
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{(\beta, \alpha) \vdash \Gamma, (y : \alpha, z : \alpha)}{(\beta, \alpha); \Gamma, (y : \alpha, z : \alpha) \vdash z : \alpha}\,\textbf{:var}
    }{(\beta, \alpha); \Gamma, (y : \alpha) \vdash \lambda z\!:\!\alpha.\, z : \alpha \to \alpha}\,\textbf{:abs}
  }{(\beta, \alpha); \Gamma \vdash \lambda y\!:\!\alpha.\, \lambda z\!:\!\alpha.\, z : \alpha \to \alpha \to \alpha}\,\textbf{:abs}
}{\beta; \Gamma \vdash \Lambda\alpha.\lambda y\!:\!\alpha.\, \lambda z\!:\!\alpha.\, z : \tau_{\mathbb{B}}}\,\textbf{:tabs}
$$

We combine the two derivation trees into one large one, which gives us the type of the whole term:

$$
\dfrac{
  \dfrac{\vdots}{\beta; \Gamma \vdash \lambda x\!:\!\tau_{\mathbb{B}}.\, x\ \beta\ f\ g : \tau_{\mathbb{B}} \to \beta}
  \qquad
  \dfrac{\vdots}{\beta; \Gamma \vdash \Lambda\alpha.\lambda y\!:\!\alpha.\, \lambda z\!:\!\alpha.\, z : \tau_{\mathbb{B}}}
}{\beta; \Gamma \vdash (\lambda x\!:\!\tau_{\mathbb{B}}.\, x\ \beta\ f\ g)\ (\Lambda\alpha.\lambda y\!:\!\alpha.\, \lambda z\!:\!\alpha.\, z) : \beta}\,\textbf{:app}
$$

2.3 Operational semantics

In this section, we describe the way in which lambda terms *reduce* or *evaluate* to other terms. We first introduce the concept of *values*. The values should be thought of as exactly the set of lambda terms of which we declare that we accept them as valid results of a program. The subsequent definition of *evaluation contexts*, then, allows for a clean and uniform definition of the actual operational semantics. We close the section with some extra notation that will come in handy and proofs that the operational semantics is deterministic and never increases a term's free (type) variables.

We start with the definition of a value.

Definition 24 (Value)**.** *We define the set Val of values in our language to be a subset of Term according to the following grammar:*

$$v ::= \lambda x\!:\!\tau.\, e \mid \Lambda\alpha.e \mid \texttt{in}\ e$$

Throughout this document, unless noted otherwise, we let $v \in Val$.

It might not be obvious to the reader why we choose to regard functions as values. After all, one could think of a function as an unfinished thing which will only return a value after sufficiently many arguments of the right type have been given to it. However, the reader will notice that we have no built-in atomic types representing concepts one would expect in a "real-world" programming language, like natural numbers or booleans. We therefore encode these concepts into polymorphic functions. For example, the two boolean values *true* and *false* can be encoded as polymorphic functions taking two arguments where, by convention, *true* always returns the first and *false* returns the second argument. Similarly, we can encode a natural number n as a polymorphic function that takes a function f and a suitable argument x and applies f to x exactly n times. This is known as the encoding into Church numerals.[5, 6]

$$\begin{aligned}
\overline{true} &:= \Lambda\alpha.\lambda x\!:\!\alpha.\, \lambda y\!:\!\alpha.\, x,\\
\overline{false} &:= \Lambda\alpha.\lambda x\!:\!\alpha.\, \lambda y\!:\!\alpha.\, y,\\
\overline{n} &:= \Lambda\alpha.\lambda f\!:\!\alpha \to \alpha.\, \lambda x\!:\!\alpha.\, \underbrace{f\ (\ldots\ (f}_{n}\ x)\ldots).
\end{aligned}$$

There are certain constructs in the language, among which the application of terms to terms or types, which can be seen as *evaluation contexts.* The purpose of this definition is mainly to make the definition of the operational semantics shorter and more uniform. The meaning of this becomes clear in the definition of the operational semantics.

Definition 25 (Evaluation context)**.** *We define the set ECtxt of evaluation contexts using the following grammar:*

$$E ::= (\text{-}) \mid E\ e \mid E\ \tau \mid \texttt{out}\ E$$

Throughout this document, unless noted otherwise, we let $E \in ECtxt$.

We also define a substitution for evaluation contexts. Substitution of a term e for the hole (-) *in evaluation context E is denoted $E[e]$ and is defined inductively in the natural way, clearly making $E[e]$ a term:*

$$\begin{aligned}
(\text{-})[e] &:= e\\
(E\ e')[e] &:= E[e]\ e'\\
(E\ \tau)[e] &:= E[e]\ \tau\\
(\texttt{out}\ E)[e] &:= \texttt{out}\ (E[e]).
\end{aligned}$$

Definition 26 (Operational semantics)**.** *We inductively define a binary relation $\mapsto$ on Term using the following rules:*

$$\frac{}{(\lambda x\!:\!\tau.\, e)\ e' \mapsto e[e'/x]}\ \mapsto_{\textbf{app}} \qquad \frac{}{(\Lambda\alpha.e)\ \tau \mapsto e[\tau/\alpha]}\ \mapsto_{\textbf{tapp}}$$

$$\frac{}{\texttt{out}\ (\texttt{in}\ e) \mapsto e}\ \mapsto_{\textbf{outin}} \qquad \frac{e \mapsto e'}{E[e] \mapsto E[e']}\ \mapsto_{\textbf{ectxt}}$$

If $e \mapsto^* e'$, *we say that* e reduces to e'.

Notation 27. *We say that a term e is* irreducible *iff it is not at the left-hand side of any tuple in* $\mapsto$*:*

$$irred\, e :\Leftrightarrow \nexists e' : e \mapsto e'.$$

We use the notation $e \Downarrow^k e'$ for the statement $e \mapsto^k e' \wedge irred\, e'$. We say that e evaluates to e' *in k steps.*

Similarly, we use $e \Downarrow e'$ for $e \mapsto^* e' \wedge irred\, e'$ and we say that e evaluates to e'.

We then also use and $e \Downarrow$ and $e \Downarrow^k$ for $\exists e' : e \Downarrow e'$ and $\exists e' : e \Downarrow^k e'$, respectively. Here, the terminology is that e terminates.

We can immediately make a few observations about these rules. First, we see that there are no rules governing how values are evaluated. Therefore, values are irreducible. This aligns with the intuitive meaning we gave to *Val* earlier. Values are not further reduced, precisely because we regard them as valid program results. Whether the converse, $\forall e : \mathit{irred}\, e \Rightarrow e \in \mathit{Val}$, holds, is the subject of Chapter 3.

Second, we see that all four rules dictate how to reduce some filled-in evaluation context. The rule $\mapsto_{\mathbf{ectxt}}$ shows how to reduce an evaluation context filled in with a term e that itself can be reduced, namely by reducing e first. The three other rules show how to reduce evaluation contexts filled in with specific sorts of values. We see that these rules remove at least one layer of structure in the process, e.g. in $\mapsto_{\mathbf{app}}$, the λx construct around e is reduced away. This gives a justification for the term *evaluation context*: if a term is an evaluation context filled in with e, then we first reduce e until it is fully *evaluated* before we let it interact with the *context* in which it is located.

Example 28. *As an example, we provide the derivation of a reduction path for a term. We will leave out the type annotations in lambda abstractions for legibility.*

We start with the term $(\lambda x.(\mathtt{out}(x\ \beta\ f))\ g)\ (\Lambda\alpha..\lambda y.\mathtt{in}\lambda z.z)$.

We apply the rule $\mapsto_{\mathbf{app}}$*:* $(\mathtt{out}((\Lambda\alpha..\lambda y.\mathtt{in}\lambda z.z)\ \beta\ f))\ g$*. From here, there is no rule that we can apply without making further assumptions. In fact, we will have to go several levels deep, as the following derivation tree shows:*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{(\Lambda\alpha.\lambda y.\mathtt{in}\lambda z.z)\ \beta \mapsto \lambda y.\mathtt{in}\lambda z.z}\ \mapsto_{\mathbf{tapp}}}{(\Lambda\alpha.\lambda y.\mathtt{in}\lambda z.z)\ \beta\ f \mapsto (\lambda y.\mathtt{in}\lambda z.z)\ f}\ \mapsto_{\mathbf{ectxt}}}{\mathtt{out}((\Lambda\alpha.\lambda y.\mathtt{in}\lambda z.z)\ \beta\ f) \mapsto \mathtt{out}((\lambda y.\mathtt{in}\lambda z.z)\ f)}\ \mapsto_{\mathbf{ectxt}}}{(\mathtt{out}((\Lambda\alpha.\lambda y.\mathtt{in}\lambda z.z)\ \beta\ f))\ g \mapsto (\mathtt{out}((\lambda y.\mathtt{in}\lambda z.z)\ f))\ g}\ \mapsto_{\mathbf{ectxt}}$$

The next step, starting from $(\mathtt{out}((\lambda y.\mathtt{in}\lambda z.z)\ f))\ g$*, requires a similar but slightly less deep tree:*

$$\dfrac{\dfrac{\dfrac{}{(\lambda y.\mathtt{in}\lambda z.z)\ f \mapsto \mathtt{in}\lambda z.z}\ \mapsto_{\mathbf{app}}}{\mathtt{out}((\lambda y.\mathtt{in}\lambda z.z)\ f) \mapsto \mathtt{out}(\mathtt{in}\lambda z.z)}\ \mapsto_{\mathbf{ectxt}}}{(\mathtt{out}((\lambda y.\mathtt{in}\lambda z.z)\ f))\ g \mapsto (\mathtt{out}(\mathtt{in}\lambda z.z))\ g}\ \mapsto_{\mathbf{ectxt}}$$

The final two steps can then be done without deep trees. The term $(\mathtt{out}(\mathtt{in}\lambda z.z))\ g$ *evaluates to* $(\lambda z.z)\ g$ *by application of* $\mapsto_{\mathbf{ectxt}}$*, since* $\mathtt{out}(\mathtt{in}\lambda z.z) \mapsto_{\mathbf{outin}} \lambda z.z$*. The final application, then, is* $(\lambda z.z)\ g \mapsto_{\mathbf{app}} g$*.*

A useful result that we can prove about the operational semantics is that it is deterministic. In other words, every term reduces to at most one new term. Thus, if we consider all possible reduction paths starting from a term, there is no branching.

Lemma 29 (Determinism)**.** *For all terms* $a, b, c \in \mathit{Term}$ *such that* $a \mapsto b \wedge a \mapsto c$*, we have that* $b = c$*. This means that* $a \mapsto^k b \wedge a \mapsto^k c$ *also implies* $b = c$*, by induction on* k*.*

Proof. By induction on the structure of a. First of all, the cases $a = x$, $a = \lambda x{:}\tau.\, e$, $a = \Lambda\alpha.e$, $a = \mathtt{in}\ e$ are trivial since they make a irreducible. The remaining cases are proved by generalising on b, c and realising that $a \mapsto b \wedge a \mapsto c$ can only be caused by one of two operational semantics rules, exactly one of which requires the use of the induction hypothesis. Looking at the reduction rules and considering the non-overlap between *Val* and the set of reducible terms, we can already see that there is no term two which two different rule schemas apply.

- $a = e_1\ e_2$. Then, $a \mapsto b$ and $a \mapsto c$ are both applications of $\mapsto_{\mathbf{app}}$ or both of $\mapsto_{\mathbf{ectxt}}$. In the first case, say $e_1 = \lambda x : \tau.\, e_1'$, which yields $b = c = e_1'[e_2/x]$. In the second case, there are e_b, e_c such that $e_1 \mapsto e_b$ and $a = e_1\ e_2 \mapsto e_b\ e_2 = b$, and similarly with $e_1 \mapsto e_c$ for c. We apply the induction hypothesis to e_1, e_b, e_c and find that $e_b = e_c$. Therefore, $b = e_b\ e_2 = e_c\ e_2 = c$.

- $a = e\ \tau$. Similar. Either the two reductions are both applications of $\mapsto_{\mathbf{tapp}}$ or both of $\mapsto_{\mathbf{ectxt}}$. In the first case, say $e = \Lambda\alpha.e'$, which yields $b = c = e'[\tau/\alpha]$. In the second case, there are e_b, e_c such that $e \mapsto e_b$ and $a = e\ \tau \mapsto e_b\ \tau = b$, and similarly with $e\ \tau \mapsto e_c\ \tau$ for c. We apply the induction hypothesis to e, e_b, e_c and find that $e_b = e_c$. Therefore, $b = e_b\ e_2 = e_c\ e_2 = c$.

- $a = \texttt{out}\ e$. Again, similar. Either the two reductions are both applications of $\mapsto_{\mathbf{outin}}$ or both of $\mapsto_{\mathbf{ectxt}}$. In the first case, say $e = \texttt{in}\ e'$, which yields $b = c = e'$. In the second case, there are e_b, e_c such that $e \mapsto e_b$ and $a = \texttt{out}\ e \mapsto \texttt{out}\ e_b = b$, and similarly $e \mapsto e_c$. We apply the induction hypothesis to e, e_b, e_c and find that $e_b = e_c$. Therefore, $b = \texttt{out}\ e_b = \texttt{out}\ e_c = c$.

□

The final result of this section discusses free (type) variables in terms. It turns out that during the reduction of a term, its set of free variables can only decrease. Neither do bound variables suddenly become free or do they appear out of nowhere.

Definition 30 (Closed terms and closed values)**.** *We define the set of* closed terms $\mathit{CTerm} := \{e \in \mathit{Term} \mid \mathrm{fv}\, e = \emptyset \wedge \mathrm{ftv}\, e = \emptyset\}$ *and the set of* closed values $\mathit{CVal} := \mathit{CTerm} \cap \mathit{Val}$.

Lemma 31 (Free (type) variables decrease with $\mapsto$)**.** *Given terms $a, b \in \mathit{Term}$ such that $a \mapsto b$. Then* $\mathrm{fv}\, a \supseteq \mathrm{fv}\, b$ *and* $\mathrm{ftv}\, a \supseteq \mathrm{ftv}\, b$. *By transitivity and reflexivity of set inclusion, this property thus also holds if we replace $\mapsto$ with $\mapsto^*$. We can also phrase this as:* fv *and* ftv *are monotonic functions from $\mathit{Term}, \mapsto^*$ to $\mathit{Var}, \supseteq$ and $\mathit{TVar}, \supseteq$, respectively.*

Proof. By induction on the $\mapsto$ relation, where the cases $a = x$, $a = \lambda x{:}\tau.\, e$, $a = \Lambda\alpha.e$, $a = \texttt{in}\ e$ are trivial since they make a irreducible. We prove the remaining cases separately.

- $a = e_1\ e_2$. Then, $\mathrm{fv}\, a = \mathrm{fv}\, e_1 \cup \mathrm{fv}\, e_2$ and $\mathrm{ftv}\, a = \mathrm{ftv}\, e_1 \cup \mathrm{ftv}\, e_2$.

  $a \mapsto b$ is an application of either $\mapsto_{\mathbf{app}}$ or $\mapsto_{\mathbf{ectxt}}$. In the first case, say $e_1 = \lambda x{:}\tau.\, e_1'$. Then $b = e_1'[e_2/x]$. We see that $\mathrm{fv}\, b \subseteq (\mathrm{fv}\, e_1' \setminus \{x\}) \cup \mathrm{fv}\, e_2 = \mathrm{fv}\, a$[2] and $\mathrm{ftv}\, b \subseteq \mathrm{ftv}\, e_1' \cup \mathrm{ftv}\, e_2 \subseteq \mathrm{ftv}\, a$.

  In the second case, say that $e_1 \mapsto e_1'$. Then $b = e_1'\ e_2$. Thus, $\mathrm{fv}\, b = \mathrm{fv}\, e_1' \cup \mathrm{fv}\, e_2 \subseteq \mathrm{fv}\, e_1 \cup \mathrm{fv}\, e_2 = \mathrm{fv}\, a$ and $\mathrm{ftv}\, b = \mathrm{ftv}\, e_1' \cup \mathrm{ftv}\, e_2 \subseteq \mathrm{ftv}\, e_1 \cup \mathrm{ftv}\, e_2 = \mathrm{ftv}\, a$, both by the induction hypothesis.

- $a = e\ \tau$. Then, $\mathrm{fv}\, a = \mathrm{fv}\, e$ and $\mathrm{ftv}\, a = \mathrm{ftv}\, e \cup \mathrm{ftv}\, \tau$.

  $a \mapsto b$ is an application of either $\mapsto_{\mathbf{tapp}}$ or $\mapsto_{\mathbf{ectxt}}$. In the first case, say $e = \Lambda\alpha.e'$. Then $b = e'[\tau/\alpha]$. We see that $\mathrm{fv}\, b = \mathrm{fv}\, e' = \mathrm{fv}\, e = \mathrm{fv}\, a$, while $\mathrm{ftv}\, b \subseteq (\mathrm{ftv}\, e' \setminus \{\alpha\}) \cup \mathrm{ftv}\, \tau = \mathrm{ftv}\, e \cup \mathrm{ftv}\, \tau = \mathrm{ftv}\, a$.[3]

  In the second case, say that $e \mapsto e'$. Then $b = e'\ \tau$. Thus, $\mathrm{fv}\, b = \mathrm{fv}\, e' \subseteq \mathrm{fv}\, e = \mathrm{fv}\, a$ and $\mathrm{ftv}\, b = \mathrm{ftv}\, e' \cup \mathrm{ftv}\, \tau \subseteq \mathrm{ftv}\, e \cup \mathrm{ftv}\, \tau = \mathrm{ftv}\, a$, both by the induction hypothesis.

[2] The subset assertion is not an equality if there is no x free in e_1', since then $b = e_1'[e_2/x] = e_1'$. Note also that $\mathrm{fv}\, e_1 = \mathrm{fv}\, e_1' \setminus \{x\}$.

[3] The subset assertion is not an equality if there is no α free in e', since then $b = e'[\tau/\alpha] = e'$. Note also that $\mathrm{ftv}\, e = \mathrm{ftv}\, e' \setminus \{x\}$.

- $a = \texttt{out}\ e$. Then, $\mathrm{fv}\, a = \mathrm{fv}\, e$ and $\mathrm{ftv}\, a = \mathrm{ftv}\, e$.

  $a \mapsto b$ is an application of either $\mapsto_{\mathbf{outin}}$ or $\mapsto_{\mathbf{ectxt}}$. Both cases are obvious and allow fv and ftv to move through the `out` and `in` constructs.

$\square$

Corollary 32. *Given terms $a, b \in Term$ such that $a \in CTerm$ and $a \mapsto^* b$. Then $b \in CTerm$.*

2.4 Discussion

In this section, we provide some discussion of the language we just defined. Most observations will not be directly useful in proving the results of this thesis. Those will be introduced and proved in the next two chapters. This section is meant for properties of the language that cannot naturally be discussed elsewhere, but are worth mentioning and exploring nonetheless. We will first provide the reader with a motivation for recursive *mu*-types (we assume the reader already understands the use of the functions and universally quantified types). Then, we will show that our language admits terms with infinite reduction paths and arbitrary recursion. In the third section, we show that in contrast with derivation of reduction paths in the operational semantics, our type relation is not deterministic, i.e. there are terms that receive multiple types. Finally, we take the time to compile a list of important differences between the call-by-value language as defined by Ahmed[3] and our call-by-name language.

2.4.1 Type-level recursion

At this point, the reader might wonder what the actual use of μ and type recursion is. In short, μ should be thought of as the *least fixpoint operator on the type level.* In this section, we will illustrate this with an example. In order not to spend too much time on details, we will approach this subject in a not-so-rigorous manner.

A way of looking at the set $\mathbb{N}$ of natural numbers is that it is the smallest set (up to isomorphism as sets, i.e. bijection) such that expanding it with one new element does not change it (up to isomorphism). We will not go into the mathematics of this.

Claim 33 (Characterisation of $\mathbb{N}$ up to isomorphism)**.** *$\mathbb{N}$ is the smallest set X such that $X + 1 \cong X$, with $+$ denoting disjoint union and 1 any singleton set.*

We will translate the characterisation of $\mathbb{N}$ into our lambda calculus. Then, we will look for a type that *loosely* satisfies this lambda calculus analogue of the characterisation and thus can fulfill the role of "natural number type". We will see that μ-recursive types are the way to achieve this.

We now find analogues in our language for both disjoint unions and singletons, starting with the latter. The first term that comes to the reader's mind when looking at the type $\tau_{id} := \forall\alpha.\alpha \to \alpha$ is probably the polymorphic identity function, $id := \Lambda\alpha.\lambda x{:}\alpha.\, x$. Indeed, it is clearly of this type: $\cdot;\cdot \vdash id : \tau_{id}$. As we will see in Chapter 4 id is in some sense the *only* term of type τ_{id}. We therefore choose to regard τ_{id} as the "singleton type" and we will sometimes write it as 1:

$$1 := \tau_{id} = \forall\alpha.\alpha \to \alpha.$$

The second concept to translate into our language is that of disjoint unions. Given two types τ, σ, we want to construct a new type that seems to "contain" all terms of type τ as well as those of type σ. We also want a term to be contained twice if it happens to be of both types. We draw inspiration from the standard set-theoretic definition of $A+B$, which uses a labeling "trick": $A+B := (\{0\}\times A)\cup(\{1\}\times B)$. We can therefore use the following definition of disjunction[8] and its accompanying constructors:

$$\begin{aligned}\tau+\sigma &:= \forall\alpha.(\tau\to\alpha)\to(\sigma\to\alpha)\to\alpha,\\ inl(\tau,\sigma) &:= \lambda x\!:\!\tau.\,\Lambda\alpha.\lambda f\!:\!\tau\to\alpha.\,\lambda g\!:\!\sigma\to\alpha.\,f\ x,\\ inr(\tau,\sigma) &:= \lambda x\!:\!\tau.\,\Lambda\alpha.\lambda f\!:\!\tau\to\alpha.\,\lambda g\!:\!\sigma\to\alpha.\,g\ x,\end{aligned}$$

where by the Barendregt convention, $\alpha\notin \mathrm{fv}\,\tau, \mathrm{fv}\,\sigma$. Here, *inl* and *inr* provide the same labeling we had in disjoint unions of sets. Indeed, we can see that if e is of type τ, then $inl(\tau,\sigma)\ e$ is of type $\tau+\sigma$, and similarly for σ and $inr(\tau,\sigma)$. Thus, if e is both of type τ and σ, it is still contained twice: $inl(\tau,\sigma)\ e$ and $inr(\tau,\sigma)\ e$ are meaningfully different.

Now that we have translated all necessary concepts, we can formulate the analogue of the characterisation of $\mathbb{N}$ completely within the terminology of our lambda calculus:

We are looking for a smallest type τ^* such that $\tau^*\cong\tau^*+1$.

It turns out that $\tau := \mu\alpha.\alpha+1$ comes close to satisfying this isomorphism requirement. Given an argument t of type τ. We can *unfold* t to `out` t of type $(\mu\alpha.\alpha+1)+1$. This last type is equal to $\tau+1$, so we are done. The other way around, given an argument t' of type $\tau+1=(\mu\alpha.\alpha+1)+1=(\alpha+1)[\mu\alpha.\alpha+1/\alpha]$, we can *fold* it into `in` t' of type $\mu\alpha.\alpha+1=\tau$.

Note that several imperfections remain. First, we have not addressed minimality of τ. In fact, we have not even truly made it precise. Second, the two-way operation we described is not exactly an isomorphism. In one direction, we find that although `in out` t and t seem to be very similar in some sense, they are not equal. In the other direction we see that `out in` t' is also different from t, although we do have `out in` $t\mapsto t$. As this section is merely meant to give the reader an idea of the use of μ-recursive types, we do not intend to fix these problems.

Besides defining the natural numbers, we could also add other concepts to the language through the use of recursive types. For example, consider the fact that for any set L, its Kleene closure $L^*=L^0+L^1+L^2+\ldots$, representing lists over L, has the property that $L^*\cong 1+(L\times L^*)$, where $L^0=1$. Then defining something analogous to cartesian products but for types could start a search for a type τ such that $\tau\cong 1+(\tau_L\times\tau)$, for any given type τ_L. This search would then lead us to $\tau:=\mu\alpha.1+(\tau_L\times\alpha)$.

2.4.2 Diverging terms and term-level recursion

In this section, we turn to some interesting terms that can be constructed in our language $\lambda\forall\mu$. We will see that we can create terms whose reduction never finishes, which we will call *diverging terms*. For this, we will get inspiration from the untyped lambda calculus. We will also find there is a term in our language that can perform arbitrary recursion, again taken from the untyped lambda calculus. We close the section by making it plausible that the first two things were in fact special cases of a stronger statement: the entirety of the untyped lambda calculus can be embedded into $\lambda\forall\mu$. It would lead us to far to prove this, though.

Divergence Let us start with diverging terms. In the untyped lambda calculus, one has the term $\Omega := \omega\ \omega$, where $\omega := \lambda.\,x\ x$. Its only reduction path is $\Omega\mapsto\Omega\mapsto\ldots$.[5] We wish to translate this term to our language in such a way that the translation is well-typed. This poses the immediate problem that we need to provide types for the lambda abstractions. However, the following loose argumentation will get us there.

We start with the idea of $\omega := \lambda x : \tau_1 .\, x\ x$ and $\Omega := \omega\ \omega$. However, this does not work. If $\cdot ; \cdot \vdash \omega : \tau_1 \to \tau_2$, for Ω to be well-typed we need ω to accept type $\tau_1 \to \tau_2$ and thus $\tau_1 = \tau_1 \to \tau_2$. This is impossible with our finite type strings. Fortunately, we can "solve this type equation" using type recursion: we make up our mind and define $\omega := \lambda x : \mu\alpha.\alpha \to \tau_2 .\, (\texttt{out}\ x)\ x$. We see that this leads to $\cdot ; \cdot \vdash \omega : (\mu\alpha.\alpha \to \tau_2) \to \tau_2$. This still leaves us with an untypeable $\Omega = \omega\ \omega$, since the left-hand ω expects the right hand ω to be of type $\mu\alpha.\alpha \to \tau_2$ instead of $\mu\alpha.(\mu\alpha.\alpha \to \tau_2) \to \tau_2$. These last two types are very similar however, and we find that one alteration of Ω does the trick: $\Omega := \omega\ (\texttt{in}\ \omega)$. It is an easy exercise to see that $\cdot ; \cdot \vdash \Omega : \tau_2$. We conclude that we can define such a term for any type $\tau \in \mathit{CType}$:[4]

$$\Omega_\tau := \omega\ (\texttt{in}\ \omega) = (\lambda x{:}\mu\alpha.\alpha \to \tau .\, (\texttt{out}\ x)\ x)\ (\texttt{in}\ \lambda x{:}\mu\alpha.\alpha \to \tau .\, (\texttt{out}\ x)\ x).$$

Now that we successfully translated the untyped lambda calculus term Ω into $\lambda\forall\mu$, we must still prove that it diverges. This turns out to be rather straightforward (we leave out the subscripts for readability):

Definition 34 (Basic divergent term)**.** *We say that Ω_τ is the* basic divergent term *of type τ:*

$$\begin{aligned}\Omega\ &= \omega\ (\texttt{in}\ \omega) = (\lambda x{:}\mu\alpha.\alpha \to \tau .\, (\texttt{out}\ x)\ x)\ (\texttt{in}\ \lambda x{:}\mu\alpha.\alpha \to \tau .\, (\texttt{out}\ x)\ x)\\ &\mapsto ((\texttt{out}\ x)\ x)[\texttt{in}\ \omega/x]\\ &= (\texttt{out}\ (\texttt{in}\ \omega))\ (\texttt{in}\ \omega)\\ &\mapsto \omega\ (\texttt{in}\ \omega)\\ &= \Omega.\end{aligned}$$

Since $\Omega \notin \mathit{Val}$ and $\mapsto$ is deterministic, this means that $\Omega \not\Downarrow$.

These discoveries combine into the following proposition:

Proposition 35 (Every type has a diverging term)**.** *Given a type τ such that $\Delta \vdash \tau$. Then there exists a term Ω_τ such that $\Delta ; \cdot \vdash \Omega_\tau : \tau$ and $\Omega_\tau \not\Downarrow$. We will often refer to it as Ω instead of Ω_τ.*

Proof. We provide the term

$$\Omega_\tau := \omega\ (\texttt{in}\ \omega) = (\lambda x{:}\mu\alpha.\alpha \to \tau .\, (\texttt{out}\ x)\ x)\ (\texttt{in}\ \lambda x{:}\mu\alpha.\alpha \to \tau .\, (\texttt{out}\ x)\ x).$$

The proof of divergence has been done. We now turn to the type derivation. First, we provide a derivation tree for ω. We use Γ as an abbreviation for $(x : \mu\alpha.\alpha \to \tau)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Delta \vdash \mu\alpha.\alpha \to \tau}{\Delta;\Gamma \vdash x : \mu\alpha.\alpha \to \tau}\,\textbf{:var}}{\Delta;\Gamma \vdash \texttt{out}\ x : (\mu\alpha.\alpha \to \tau) \to \tau}\,\textbf{:muout} \qquad \dfrac{\Delta \vdash \mu\alpha.\alpha \to \tau}{\Delta;\Gamma \vdash x : \mu\alpha.\alpha \to \tau}\,\textbf{:var}}{\Delta;\Gamma \vdash (\texttt{out}\ x)\ x : \tau}\,\textbf{:app}}{\Delta;\cdot \vdash \lambda x{:}\mu\alpha.\alpha \to \tau .\, (\texttt{out}\ x)\ x : (\mu\alpha.\alpha \to \tau) \to \tau}\,\textbf{:abs}$$

From here, deriving the full derivation tree for the term Ω is not difficult:

$$\dfrac{\dfrac{\vdots}{\Delta;\cdot \vdash \omega : (\mu\alpha.\alpha \to \tau) \to \tau} \qquad \dfrac{\dfrac{\vdots}{\Delta;\cdot \vdash \omega : (\mu\alpha.\alpha \to \tau) \to \tau}}{\Delta;\cdot \vdash \texttt{in}\ \omega : \mu\alpha.\alpha \to \tau}\,\textbf{:muin}}{\Delta;\cdot \vdash \Omega : \tau}\,\textbf{:app}$$

□

[4] We can make this slightly more general and say that $\Delta \vdash \tau$ implies that $\Delta;\cdot \vdash \Omega_\tau : \tau$.

Note that the term would not have been typeable if our type recursion had been covariant. This makes it such that, even though α appears negatively in the type $\mu\alpha.\alpha \to \tau$ of x, the unfolding $\texttt{out}\ x$ is still welltyped. Since Ω does not terminate, Mendler's strong normalisability theorem states that it would not be welltyped in a system with only covariant recursive types.[6, Sec. 9C]

Term-level recursion Not only can we recreate the basic divergent term Ω from the untyped lambda calculus, we can also recreate the fixpoint combinator Y. For a more detailed explanation of Y, why it is called a fixpoint combinator, and what it has to do with recursion, we refer to [5]. The process of translation from untyped to typed lambda calculus is similar to what we did for Ω. In the untyped lambda calculus, $Y := \lambda f.\,(\lambda x.\,f\ (x\ x))\ (\lambda x.\,f\ (x\ x))$. We again find that for every type τ, there is a translation to $\lambda\forall\mu$:

$$Y := \lambda f{:}\tau \to \tau.\,(\lambda x{:}\mu\alpha.\alpha \to \tau.\,f\ ((\texttt{out}\ x)\ x))\ (\texttt{in}\ \lambda x{:}\mu\alpha.\alpha \to \tau.\,f\ ((\texttt{out}\ x)\ x)).$$

In slight contrast with the fixpoint combinator in the untyped lambda calculus, this one does not have the property that $Y\ F \mapsto^* F\ (Y\ F)$ for any term F. It does come very close, though:

$$\begin{aligned} Y\ F &\mapsto ((\lambda x{:}\mu\alpha.\alpha \to \tau.\,f\ ((\texttt{out}\ x)\ x))\ (\texttt{in}\ \lambda x{:}\mu\alpha.\alpha \to \tau.\,f\ ((\texttt{out}\ x)\ x)))[F/f] \\ &= (\lambda x{:}\mu\alpha.\alpha \to \tau.\,F\ ((\texttt{out}\ x)\ x))\ (\texttt{in}\ \lambda x{:}\mu\alpha.\alpha \to \tau.\,F\ ((\texttt{out}\ x)\ x)) \\ &\mapsto F\ ((\texttt{out in}\ \lambda x{:}\mu\alpha.\alpha \to \tau.\,F\ ((\texttt{out}\ x)\ x))\ (\texttt{in}\ \lambda x{:}\mu\alpha.\alpha \to \tau.\,F\ ((\texttt{out}\ x)\ x))). \end{aligned}$$

If we abbreviate $T := \lambda x{:}\mu\alpha.\alpha \to \tau.\,F\ ((\texttt{out}\ x)\ x)$, then we found that

$$Y\ F \mapsto T\ (\texttt{in}\ T) \mapsto F\ ((\texttt{out in}\ T)\ (\texttt{in}\ T)).$$

We see that this is not exactly the same as $F\ (Y\ F)$ or even $F\ (T\ (\texttt{in}\ T))$, although we do see that upon evaluation the `out`-`in` pair gets cancelled out.

In general We conjecture that we can translate any term from the untyped lambda calculus into $\lambda\forall\mu$. We define the type $L := \mu\alpha.\alpha \to \alpha$ such that $(\alpha \to \alpha)[L/\alpha] = L \to L$. The translation of an untyped term e is written $\overline{e}$ and is defined recursively as follows:

$$\begin{aligned} \overline{x} &:= x, \\ \overline{\lambda x.\,e} &:= \texttt{in}\ \lambda x{:}L.\,\overline{e}, \\ \overline{e_1\ e_2} &:= (\texttt{out}\ e_1)\ e_2. \end{aligned}$$

We conjecture the following about the typing of the terms' translations.

Conjecture 36. *Given a term e in the untyped lambda calculus. Say that the free variables of e are $x_1, \ldots, x_n$. Then $\cdot; (x_1 : L), \ldots, (x_n : L) \vdash \overline{e} : L$.*

We formalise no conjecture on the relation between a term's semantics and its tranlation's semantics, as this would lead us too far.

2.4.3 Type uniqueness

We find that the typing relation is non-functional. In other words, $\lambda\forall\mu$ allows for terms to have multiple types. As this is worthy of mentioning but not extremely insightful, we pay minor attention to it.

The crux lies in $:_{\mathbf{muin}}$. Though we will not reproduce it here, we found that an induction proof of type uniqueness fails only at this rule. The reason is the following. In general, if $\tau_1[\mu\alpha.\tau_1/\alpha] = \tau_2[\mu\alpha.\tau_2/\alpha]$, it is not the case that $\mu\alpha.\tau_1 = \mu\alpha.\tau_2$. Take for example the pair of types $\tau_1 := \alpha$ and $\tau_2 := \mu\alpha.\alpha$. (Note that the latter, being an equivalence class of type strings, is equal to $\mu\beta.\beta$ and contains no free α.) Then $\tau_1[\mu\alpha.\tau_1/\alpha] = \mu\alpha.\alpha$ is equal to $\tau_2[\mu\alpha.\tau_2/\alpha] = \tau_2 = \mu\alpha.\alpha$. Nevertheless, $\mu\alpha.\tau_1 = \mu\alpha.\alpha$ is different from $\mu\alpha.\tau_2 = \mu\alpha.\mu\alpha.\alpha$.

We have discovered that the most obvious direction of attack to prove type uniqueness does not go through in this particular case. But a stronger statement holds: type uniqueness is simply not true, i.e. we can find a term that is assigned multiple different types by our type relation $\vdash$. We will make use of our knowledge of the "problematic" typing rule $:_{\mathbf{muin}}$.

Proposition 37 (Counterexample against type uniqueness)**.** *There are types τ, σ and a term e such that $\tau \neq \sigma$, while both $\cdot;\cdot \vdash e : \tau$ and $\cdot;\cdot \vdash e : \sigma$.*

Proof. We consider the term $\Omega_{\mu\alpha.\alpha}$ as defined in Prop. 35, but refer to it as Ω for brevity. Since $\cdot \vdash \mu\alpha.\alpha$, we have $\cdot;\cdot \vdash \Omega : \mu\alpha.\alpha$. As discussed before, we then know by $:_{\mathbf{muin}}$ that both $\cdot;\cdot \vdash \texttt{in}\,\Omega : \mu\alpha.\alpha$ and $\cdot;\cdot \vdash \texttt{in}\,\Omega : \mu\alpha.\mu\alpha.\alpha$. ◻

2.4.4 Differences with Ahmed's language

In this section, we reiterate over some differences between the language used by Ahmed[3] and our language $\lambda\forall\mu$. The first, perhaps most noticeable difference is that our language has no built-in existential types. As it turns out, it is possible to define some form of existential types using universal types, which we do have, but nonetheless they do not occur as part of the language definition.[8]

CBV vs. CBN We have stated before that our language is CBN (call-by-name) while Ahmed's is CBV (call-by-value). To understand the meaning of this, we must compare our operational semantics to that of Ahmed. We see that Ahmed's definitions have two important differences. First, $v\ E$ is considered an evaluation context, which means that when $e \mapsto e'$, then $(\lambda x\!:\!\tau.\, e'')\ e$ reduces to $(\lambda x : \tau.\, e'')\ e'$. Second, this reducing of the argument only stops when it is fully evaluated, to a value v. Then the reduction $(\lambda x\!:\!\tau.\, e'')\ v \mapsto e''[v/x]$ is performed. This constitutes a great difference with our language, where the argument e' in $e\ e'$ is substituted in the function as soon as e is a lambda abstraction. (When that is not the case, the two languages agree: always reduce the left-hand side of an application first.)

Does this make any practical difference, though? The answer is yes. Let us return to the fixpoint combinator Y defined in Section 2.4.2. (Let us also reuse the abbrevation T.) Recall how in our language, $Y\ F$ does not necessarily reduce to $F\ (T\ (\texttt{in}\,T))$. Under CBV semantics, though, this *does* always happen, as long as F is a lambda abstraction:

$$\begin{aligned}
Y\ F\ &\mapsto ((\lambda x\!:\!\mu\alpha.\alpha \to \tau.\, f\ ((\texttt{out}\ x)\ x))\ (\texttt{in}\ \lambda x\!:\!\mu\alpha.\alpha \to \tau.\, f\ ((\texttt{out}\ x)\ x)))[F/f] \\
&\mapsto ((\lambda x\!:\!\mu\alpha.\alpha \to \tau.\, F\ ((\texttt{out}\ x)\ x))\ (\texttt{in}\ \lambda x\!:\!\mu\alpha.\alpha \to \tau.\, F\ ((\texttt{out}\ x)\ x))) \\
&= T\ (\texttt{in}\,T) \\
&\mapsto F\ ((\texttt{out in}\ \lambda x\!:\!\mu\alpha.\alpha \to \tau.\, F\ ((\texttt{out}\ x)\ x))\ (\texttt{in}\ \lambda x\!:\!\mu\alpha.\alpha \to \tau.\, F\ ((\texttt{out}\ x)\ x))) \\
&= F\ ((\texttt{out in}\,T)\ (\texttt{in}\,T)) \\
&\mapsto F\ (T\ (\texttt{in}\,T)).
\end{aligned}$$

Implicit vs. explicit typing Another difference is that we use explicit typing while Ahmed uses implicit typing. In other words, Ahmed writes down term-level lambda abstractions without mentioning the argument's expected type, type-level abstractions without mentioning the bound type variable, and term applications to a type without mentioning which type. We list some examples and provide information on the types they are assigned under Ahmed's type system:

1. $\lambda x.\, x$ can have type $\tau \to \tau$, for any type τ. Note that this is the case even though there is no type-level lambda abstraction. Consequently, the type is not $\forall\alpha.(\alpha \to \alpha)$, which is different from $\tau \to \tau$ for every τ.

2. $\Lambda.\, \lambda x.\, x$ can have type $\forall\alpha.\alpha \to \alpha$, but also $\forall\alpha.\tau \to \tau$ for any type τ. In our language, the difference would be clear because of the type annotation in the λx binder.

3. $\Lambda.\, \Lambda.\, \lambda x.\, \lambda y.\, \lambda z.\, z$ can have many types. One class of them is $\forall\alpha.\forall\beta.\alpha \to \beta \to \tau \to \tau$ and the other is $\forall\alpha.\forall\beta.\beta \to \alpha \to \tau \to \tau$. Again, it is also possible that τ is α or β.

All these type ambiguities are due to the fact that the type- and term-level abstraction do not mention their type variables and types, respectively.

A final example: $(\Lambda.\, \lambda x.\, x)\ []$, where $[]$ is Ahmed's notation for application of a term to a type, can have absolutely any type, precisely because $[]$ does not specify which type to apply the term to.

Non-determinism vs. determinism The last difference we will discuss is that Ahmed's operational semantics is non-deterministic. This follows from the fact that they consider not only `out` E but also `in` E an evaluation context. (Note that Ahmed uses the names `unfold` and `fold` for `out` and `in`, respectively.) Consequently, they also only consider `in` e a value if $e \in \mathit{Val}$, while we always consider `in` e a value. What happens with these semantics, then, is that `out in` $e \mapsto$ `out in` e' if $e \mapsto e'$. At the same time, though, `out in` $e \mapsto e$ because of the evaluation rule $\mapsto_{\texttt{outin}}$. Though this constitutes a proof of non-determinism of Ahmed's language, we conjecture that the non-determinism is "contained". Indeed, the circumstances under which we have detected the phenomenon to occur are such that the divergence immediately collapses again:

$$\texttt{out in}\ e \mapsto \texttt{out in}\ e' \mapsto e',$$

$$\texttt{out in}\ e \mapsto e \mapsto e'.$$

We therefore conjecture that some confluence property holds.

Conjecture 38. *Under Ahmed's operational semantics, $e \mapsto^* e_1'$ and $e \mapsto^* e_2'$ together imply that $e_1' \mapsto^* e''$ and $e_2' \mapsto^* e''$ for some e''. This would mean that $\Downarrow$* is *deterministic under Ahmed's operational semantics.*

Chapter 3

Unary relation and type-safety

In Chapter 2, we saw that $e \in \mathit{Val} \Rightarrow \mathit{irred}\, e$ follows from the definition of the operational semantics. In this chapter, we will find that in general, the converse does not hold. The first section will start with an investigation through some examples. From these examples, we will distill a hypothesis stating that an extra requirement, namely the well-typedness of e, suffices to let the converse go through. We will find that a direct proof of the hypothesis by induction does not work.

The second section will state a language property stronger than the earlier hypothesis, called *type-safety*.

In the third section, we turn to the method of step-indexed logical relations as presented by Appel-McAllester and by Ahmed.[4, 3] We will use this technique to prove type-safety. This section will contain the definition of the step-indexed logical relation, which we will name $\vDash$. (Since our relation is unary, another name would be *predicate*, but we stick with the standard terminology.)

In the fourth section, we will then start the type-safety proof with the *fundamental property* of $\vDash$, namely that all well-typed terms e are members of $\vDash$. This proof will be by induction on the type relation and will be similar in structure to that of Ahmed. Nevertheless, the lemmas making up the induction steps will be crucially different, taking in account all the differences between the languages. The very end of the section will then continue the proof. From the fundamental property, we will derive that type-safety does indeed hold.

3.1 Forming a hypothesis

The definition of Val and of the operational semantics in Chapter 2 showed us that $e \in \mathit{Val}$ implies $\mathit{irred}\, e$. We briefly posed the question whether the converse holds. A very simple counterexample shows that it does not: the variable x is an irreducible term but is not a value. We wish to gain more insight into the properties of counterexamples. Looking at the relevant definitions, we find a term e to be a counterexample—i.e. to be such that $\mathit{irred}\, e \wedge e \notin \mathit{Val}$—iff one of the following conditions holds:

- e is some variable, say $e = x$.
- e is an application of a term to a term, say $e = e'\ e''$, such that $\mathit{irred}\, e'$ and e' is not a term-level lambda abstraction.

- e is an application of a term to a type, say $e = e'\ \tau$, such that $\mathit{irred}\, e'$ and e' is not a type-level lambda abstraction.
- e is an unfolding, say $e = \texttt{out}\ e'$, such that $\mathit{irred}\, e'$ and e' is not some folding ($\texttt{in}\ \ldots$).

These conditions lead us to the idea that $\mathit{irred}\, e \wedge e \notin \mathit{Val}$ might correlate with well-typedness of e. The first condition, $e = x$, implies that there is no τ such that $\cdot;\cdot \vdash e : \tau$. The other conditions do not *prove* the ill-typedness of e, but do seem to suggest it. For example, how can we construct a well-typed term $e = e'\ e''$ when e' must be irreducible but cannot be exactly the sort of value that we would expect on the left-hand side of an application, namely a term-level lambda abstraction?

We come to the following conjecture:

Conjecture 39. *$\mathit{irred}\, e \wedge e \notin \mathit{Val}$ implies that e is ill-typed, i.e. there is no τ such that $\cdot;\cdot \vdash e : \tau$.*

Failed proof attempt. We prove the contrapositive of the conjecture by induction. Define $S := \{\langle \Delta, \Gamma, e, \tau\rangle \mid \neg\, \mathit{irred}\, e \vee e \in \mathit{Val}\}$. If we prove that S is closed under the rules by which $\vdash$ is defined, then we know that $\Delta; \Gamma \vdash e : \tau$ implies $\neg\, \mathit{irred}\, e \vee e \in \mathit{Val}$, which proves the conjecture a fortiori. However, this fails at the first rule, $:_{\mathbf{var}}$. Suppose that $\Delta \vdash \Gamma$ and x is a variable in the domain of Γ. Then we must prove that $\langle \Delta, \Gamma, x, \Gamma\ x\rangle \in S$. This is not the case, since $\mathit{irred}\, x \wedge x \notin \mathit{Val}$. $\square$

In this failed proof attempt, we tried to prove something stronger than what we actually needed. The Δ, Γ, τ were ignored in the definition of S, which led to a proof obligation for terms with free variables, something the type relation $\vdash$ was built for but S was not. In the remaining sections of this chapter, we will try another proof by induction, where S is replaced by a more carefully constructed relation.

We end this section by answering the question whether the well-typedness property interacts with $\mathit{irred}\, e$ and $e \in \mathit{Val}$ in other ways than we have conjectured so far. We look at a brute-force overview of all eight possible combinations of the three properties in Tab. 3.1. The observation $e \in \mathit{Val} \Rightarrow \mathit{irred}\, e$ from Chapter 2 rules out all terms e satisfying combination 5 or 6. The conjecture is equivalent to the negation of combination 4. We now ask ourselves which of combinations 1, 2, 3, 7, 8 are possible, i.e. have terms satisfying them. It turns out that all of them are possible. We define $\mathit{id} := \Lambda\alpha.\lambda x : \alpha.\, x$ and see that $\cdot;\cdot \vdash \mathit{id} : \forall\alpha.\alpha \to \alpha$. The following terms then constitute examples of all five remaining combinations:

1	$\mathit{id}\ \beta$
2	$\mathit{id}\ (\forall\beta.\beta)$
3	$\texttt{out}\ x$
7	$\texttt{in}\ x$
8	id

Thus, in order to fully understand the relationship between the three properties, the only thing that remains to be done is proving (or disproving) the conjecture, stating the impossibility of combination 4.

3.2 Setting the precise goal

In this section and the next, we try to prove the conjecture from the previous section:

Conjecture 39. *$\mathit{irred}\, e \wedge e \notin \mathit{Val}$ implies that e is ill-typed, i.e. there is no τ such that $\cdot;\cdot \vdash e : \tau$.*

#	V	$\not\mapsto$	$\vdash$
1	·	·	·
2	·	·	✓
3	·	✓	·
4	·	✓	✓
5	✓	·	·
6	✓	·	✓
7	✓	✓	·
8	✓	✓	✓

Table 3.1: The eight possible combinations of the three properties. V stands for $e \in \mathit{Val}$. $\not\mapsto$ stands for $\mathit{irred}\, e$. $\vdash$ stands for $\exists \tau : \cdot; \cdot \vdash e : \tau$. We use a checkmark ✓ for satisfied properties and a dot · for unsatisfied properties. The numbers are there for ease of referring to individual rows from within the text.

We first realise that this is propositionally equivalent to the statement $(\cdot; \cdot \vdash e : \tau) \wedge (\mathit{irred}\, e) \Rightarrow (e \in \mathit{Val})$. In fact, we will prove a stronger property of the language, *type-safety*:[1]

Definition 40 (Safety of a term)**.** *We say that* $\mathit{safe}\, e$ *holds iff* $\forall e' : e \Downarrow e' \Rightarrow e' \in \mathit{Val}$.

Definition 41 (Type-safety)**.** *Our language is type-safe iff every well-typed term is safe, i.e.*

$$\forall e \in \mathit{Term}, \forall \tau \in \mathit{Type} : \cdot; \cdot \vdash e : \tau \Rightarrow \mathit{safe}\, e.$$

Indeed, it is clear that if our language is type-safe, then our conjecture holds.

The structure of these sections will be as follows. First, we will define a four-place relation $\vDash$, relating Δ, Γ, e, τ. This will be a step-indexed logical relation in the style of Ahmed.[3] This is a slightly complex matter and we will follow Ahmed's construction of the relation and its constituting concepts rather closely. However, we will make important changes to accommodate for the differences between the languages described in Section 2.4.4.

Then, we will perform the first part of the proof of type-safety. We prove $\vdash \subseteq \vDash$, i.e. of the fact that $\Delta; \Gamma \vdash e : \tau \Rightarrow \Delta; \Gamma \vDash e : \tau$, by induction on the type rules. This is know as the *fundamental property* of $\vDash$. All induction steps will be proved as separate lemmas, which we will call *compatibility lemmas*, and most of them will require some auxiliary lemmas.

The second part of the type-safety proof is to show that $\cdot; \cdot \vDash e : \tau$ implies that $\mathit{safe}\, e$. This concludes the proof, as together with the first part, it makes that $\cdot; \cdot \vdash e : \tau \Rightarrow \mathit{safe}\, e$, which is type-safety.

We now get to work and define the necessary concepts for $\vDash$.

3.3 Definitions

In the naive proof by induction in Section 3.1, we found that e having free variables prevented the proof of the conjecture from going through. We now define a unary step-indexed logical relation in the style of Ahmed that takes care of this using substitutions.[3] Free (type) variables in the term will be closed off.

In order to define these substitutions, we first need to define *semantic type relations*:

[1] Actually, when one proves that $(\cdot; \cdot \vdash e : \tau) \wedge (e \mapsto^* e')$ implies $\cdot; \cdot \vdash e' : \tau$, then type-safety turns out to be equivalent to our earlier notion.

Definition 42 (Semantic type relation)**.** *We define Rel(-) to be the following function taking a type τ such that $\tau \in CType$ with codomain $\mathcal{P}(\mathbb{N} \to \mathcal{P}(CVal))$:*

$$Rel(\tau) := \{\chi^{\text{-}} \in (\mathbb{N} \to \mathcal{P}(CVal)) \mid \forall k \in \mathbb{N} : \forall v \in \chi^k : (\cdot; \cdot \vdash v : \tau) \wedge (\forall j < k : v \in \chi^j)\}.$$

In words, we say that $Rel(\tau)$ for a closed type τ contains exactly those $\chi^{\text{-}}$ which have only values of type τ and which are downward-closed w.r.t. the index. (We will not always write the superscript hole in $\chi^{\text{-}}$.)

We also say that $Rel := \bigcup_{\tau \in Type} Rel(\tau)$.

Notation 43 (Index filtering)**.** *Suppose X is a function and Y is a set such that $X \in (\mathbb{N} \to \mathcal{P}(Y))$, and $n \in \mathbb{N}$. Then X^n is notation for the application of X to n, i.e. $X\ n$. Consequently, $X^n \subseteq Y$.*

A semantic type relation $\chi^{\text{-}} \in Rel(\tau)$ is to be viewed as a set of well-typed values, parametrised in an index k, such that the set can only decrease with increasing index. Some examples of semantic type relations are: the one that is empty for every index, $\chi^k = \emptyset$; one that is otherwise constant for every index, $\chi^k = \{v_1, \ldots, v_n\}$; one that starts of with infinitely many different values $v_1, v_2, \ldots$ of type τ for $k = 0$ and loses one value at every next index, $\chi^k = \{v_{k+1}, v_{k+2}, \ldots\}$.

The substitutions we mentioned earlier come in two kinds, one for type variables and one for variables, i.e. term-level variables. The former, we can already define. It will be used to replace type variables with closed types and an accompanying semantic interpretation.

Definition 44 (Type relation substitution)**.** *Given a type environment $\Delta \in TEnv$. We define the set $\mathcal{D}[\![\Delta]\!]$ of type relation substitutions to be the set of $\delta \in (\Delta \to (Type \times Rel))$ for which $\delta\ \alpha = \langle \tau, \chi \rangle$ implies $\tau \in CType$ and $\chi \in Rel(\tau)$.*

When $\Delta = \cdot$ then $\delta \in \mathcal{D}[\![\Delta]\!]$ is the empty function, which we will write $\cdot$ or $\emptyset$.

We let $\delta^{\text{syn}}\ \alpha$ and $\delta^{\text{sem}}\ \alpha$ denote the left- and right-hand side component of $\delta\ \alpha$, respectively.

We define $\delta\ \tau$ to be the simultaneous syntactic type substitution of $\delta^{\text{syn}}\ \alpha$ for every type variable α in Δ that occurs freely in τ.

Regarding terms, we let $\delta\ e$ denote the result of simultaneous substitution of $\delta\ \alpha$ for every type variable α occurring freely in e.

When $\alpha \notin \Delta$, we define $\delta[\alpha \mapsto \langle \tau, \chi \rangle]$ to be a type relation substitution in $\mathcal{D}[\![\Delta, \alpha]\!]$ that maps all $\alpha' \in \Delta$ to $\delta\ \alpha'$ and α to $\langle \tau, \chi \rangle$. If $\alpha \in \Delta$, then $\delta[\alpha \mapsto \ldots]$ is undefined.

Note that there is a difference between our definition of $\mathcal{D}[\![\Delta]\!]$ and the definition given by Ahmed in the unary model of their version of $\lambda\forall\mu$.[3, Appendix C] In Ahmed, $\delta \in \mathcal{D}[\![\Delta]\!]$ only returns a semantic type relation, while under our definition, it returns a tuple of both a semantic type relation χ and a syntactic type τ, which constrained by $\chi \in Rel(\tau)$. The exact reason for this will become clear in a moment.

Lemma 45. *Given the following: Δ; $\alpha \notin \Delta$; $\tau' \in Type$; $\chi \in Rel(\tau)$; δ such that $\delta' \in \mathcal{D}[\![\Delta, \alpha]\!]$, where $\delta' := \delta[\alpha \mapsto \langle \tau', \chi \rangle]$ (and thus $\delta \in \mathcal{D}[\![\Delta]\!]$).*

Then $(\delta\ \tau)[\tau'/\alpha] = \delta'\ \tau$ for all types $\tau \in Type$ and $(\delta\ e)[\tau'/\alpha] = (\delta'\ e)$ for all terms $e \in Term$.

Proof. Almost immediate. The key insight is that δ^{syn} only outputs closed types τ such that $\tau \in CType$, i.e. $\text{ftv}\,\tau = \emptyset$. Thus it does not matter whether all substitutions of (closed) types for type variables happen at the same time, as is the case in both equations' right-hand side, or that the substitution of τ' for α happens after the others. Formal proofs would be done by induction on (the shape of) $\tau \in Type$ and by induction on (the shape of) $e \in Term$. □

Lemma 46. *Given the following: Δ; α, τ, τ' such that $\Delta, \alpha \vdash \tau$ and $\Delta \vdash \tau'$; $\delta \in \mathcal{D}[\![\Delta]\!]$; $\chi \in Rel(\delta\ \tau')$.*

Let $\delta' := \delta[\alpha \mapsto \langle \delta\ \tau', \chi \rangle]$. Then $\delta'\ \tau = \delta\ (\tau[\tau'/\alpha])$.

Proof. By induction on τ.

- $\tau = \alpha$. Then $\delta' \ \tau = \delta' \ \alpha = \delta \ \tau'$, while also $\delta \ (\tau[\tau'/\alpha]) = \delta \ \tau'$.
- $\tau = \beta \neq \alpha$ with $\beta \in \Delta$. Then $\delta' \ \tau = \delta' \ \beta = \delta \ \beta$, while also $\delta \ (\tau[\tau'/\alpha]) = \delta \ \beta$.
- $\tau = \tau_1 \to \tau_2$. Follows easily by induction: the application of δ' and δ and the substitution move through the structure of $\to$.
- $\tau = \forall\beta.\sigma$. We see that $\delta' \ \tau = \forall\beta.\delta' \ \sigma$, while $\delta \ (\tau[\tau'/\alpha]) = \forall\beta.\delta \ (\sigma[\tau'/\alpha])$. (We use the Barendregt convention to prevent collision between β and other type variables.) By induction, those are equal.
- $\tau = \mu\beta.\sigma$. This case is completely analogous to the previous one, with $\forall$ replaced by μ.

□

Before we can define the other kind of substitution, we must first introduce the second key component of our step-indexed relation. We define two kinds of *step-indexed, mutually recursive term sets*, one for values and one for all terms. Both kinds are sets of terms (well-typed values in case of the former kind), parametrised in a step-index k. The basic intuition behind them is that $e \in \mathcal{E}^k_\Delta[\![\tau]\!]\delta$ means that e will "display behaviour as if it were of type τ" for at least k steps, however it is evaluated. Δ, δ are used for correct interpretation of types in connection with polymorphism. For a more detailed intuitive explanation, we refer to [3, 16].

Definition 47 (Step-indexed, mutually recursive term sets)**.** *We define two functions, $\mathcal{V}$ and $\mathcal{E}$, at once using mutual recursion.*

We let DTD denote the set of tuples $\langle\Delta,\tau,\delta\rangle$ such that $\Delta \vdash \tau$ and $\delta \in \mathcal{D}[\![\Delta]\!]$. Both $\mathcal{V}$ and $\mathcal{E}$ take a natural number $k \in \mathbb{N}$ and a tuple $\langle\Delta,\tau,\delta\rangle \in DTD$. The notation of $\mathcal{V}$ applied to its arguments is $\mathcal{V}^k_\Delta[\![\tau]\!]\delta$ and similarly for $\mathcal{E}$. The codomains are such that $\mathcal{V}^k_\Delta[\![\tau]\!]\delta \subseteq CVal$ and $\mathcal{E}^k_\Delta[\![\tau]\!]\delta \subseteq CTerm$.

We define the functions by recursion on the index k.

$$\begin{aligned}
\mathcal{V}^k_\Delta[\![\alpha]\!]\delta &:= (\delta^{\text{sem}}\ \alpha)^k, && (3.1)\\
\mathcal{V}^k_\Delta[\![\tau_1 \to \tau_2]\!]\delta &:= T(\delta\ \tau_1 \to \delta\ \tau_2) \cap \{(\lambda x{:}\delta\ \tau_1.\, e) \in CVal \mid \forall j < k : \forall e' \in \mathcal{E}^j_\Delta[\![\tau_1]\!]\delta : && (3.2)\\
&\qquad e[e'/x] \in \mathcal{E}^j_\Delta[\![\tau_2]\!]\delta\},\\
\mathcal{V}^k_\Delta[\![\forall\alpha.\tau]\!]\delta &:= T(\forall\alpha.\delta\ \tau) \cap \{(\Lambda\alpha.e) \in CVal \mid \forall j < k : \forall\tau' \in CType : \forall\chi \in Rel(\tau') :\\
&\qquad e[\tau'/\alpha] \in \mathcal{E}^j_{\Delta,\alpha}[\![\tau]\!]\delta[\alpha \mapsto \langle\tau',\chi\rangle]\},\\
\mathcal{V}^k_\Delta[\![\mu\alpha.\tau]\!]\delta &:= T(\mu\alpha.\delta\ \tau) \cap \{(\texttt{in}\ e) \in CVal \mid \forall j < k : e \in \mathcal{E}^j_\Delta[\![\tau[\mu\alpha.\tau/\alpha]]\!]\delta\},\\
\mathcal{E}^k_\Delta[\![\tau]\!]\delta &:= \{e \in CTerm \mid \forall j \leq k : \forall e' : e \Downarrow^j e' \Rightarrow e' \in \mathcal{V}^{k-j}_\Delta[\![\tau]\!]\delta\},
\end{aligned}$$

where $T(\tau) := \{e \in Term \mid \cdot;\cdot \vdash e : \tau\}$. To understand Eq. (3.1) we remember that $\delta^{\text{sem}}\ \alpha \in Rel(\delta\ \alpha) \subseteq (\mathbb{N} \to \mathcal{P}(CVal))$. The superscript k thus represents application to k, and $(\delta^{\text{sem}}\ \alpha)^k \subseteq CVal$.

Notation 48. *Given $\Delta \in TEnv, \tau \in Type, \delta \in \mathcal{D}[\![\Delta]\!]$ such that $\Delta \vdash \tau$, i.e. $\langle\Delta,\tau,\delta\rangle \in DTD$. We write $\mathcal{V}^{\cdot}_\Delta[\![\tau]\!]\delta$ to denote the function that maps a natural number k onto $\mathcal{V}^k_\Delta[\![\tau]\!]\delta$, thus $\mathcal{V}^{\cdot}_\Delta[\![\tau]\!]\delta \in (\mathbb{N} \to \mathcal{P}(CVal))$.*

Remark 49. *Now we are in the position to clarify the statement about step indexing made in the introduction. We claimed that it was necessary to perform step indexing in order to be able to properly define the relation. We can now indeed see from the μ case in the definition of $\mathcal{V}$, that if we did not have step indexing but had defined $\mathcal{V}$ and $\mathcal{E}$ by recursion on the type τ, we would have ended up with non-welldefined functions.*

Instead, we based our definitions around a step index, on which recursion is performed. Note that we are using mutual recursion in the definition of $\mathcal{V}$ and $\mathcal{E}$ and that it is not immediately clear that such a definition is valid. Therefore, we prove in Appendix (A) that the mutually recursive term sets $\mathcal{V}$ and $\mathcal{E}$ defined in Definition 47 are well-defined.

The first difference we note between our definition of these sets and the one by Ahmed is that in the case of $\tau_1 \to \tau_2$, $\mathcal{V}$ requires that e' be elements of $\mathcal{E}$ instead of $\mathcal{V}$. This is crucial given the fact that we are using call-by-name semantics instead of call-by-value. Some of the proofs in Section 3.4 would fail to go through had we not made this change.

Another difference is that the recursive definition of $\mathcal{V}$ in the case of $\mu\alpha.\tau$ uses syntactic rather than semantic substitution. In the later lemmas used to prove the fundamental property in Section 3.4, it turns out not to make a large difference, though.

Finally, the difference between the definitions of type relation substitutions in $\mathcal{D}[\![\Delta]\!]$ which we noted earlier also becomes clear now. A relevant difference between our language and Ahmed's that surfaces here is that of explicit vs. implicit typing. As was discussed in Section 2.4.4, Ahmed's type-level abstractions mention no type variable. In Ahmed's definition of $\mathcal{V}^k_\Delta[\![\forall\alpha.\tau]\!]\delta$, then, we see that $\Lambda.\, e$ is dismantled into e, which requires no further substitution of syntactic types. This differs from our definition, where $\Lambda\alpha.e$ is dismantled into $e[\tau/\alpha]$, and thus we must dispose of some syntactic type τ. Our type relation substitution δ keeps such syntactic types τ associated with semantic type relations χ.

We are now ready to define the second kind of substitution, the one for variables. After that and some extra notation, everything is in place to define the step-indexed relation $\vDash$.

Definition 50 (Semantic term substitutions)**.** *We define the function $\mathcal{G}$, which takes a natural number k, a type environment Δ, an environment Γ such that $\Delta \vdash \Gamma$, and a type relation substitution $\delta \in \mathcal{D}[\![\Delta]\!]$. The notation is $\mathcal{G}^k_\Delta[\![\Gamma]\!]\delta$ and the codomain is such that $\mathcal{G}^k_\Delta[\![\Gamma]\!]\delta \in \mathcal{P}(\operatorname{dom}\Gamma \to \mathit{CTerm})$.*

$$\mathcal{G}^k_\Delta[\![\Gamma]\!]\delta := \{\gamma \in (\operatorname{dom}\Gamma \to \mathit{CTerm}) \mid \forall x \in \operatorname{dom}\Gamma : \gamma\ x \in \mathcal{E}^k_\Delta[\![\Gamma\ x]\!]\delta\}.$$

When $\Gamma = \cdot$ then $\gamma \in \mathcal{D}[\![\Gamma]\!]$ is the empty function, which we will write $\cdot$ or $\emptyset$.

A comment similar to what we said about the $\tau_1 \to \tau_2$ case in the definition of $\mathcal{V}$ can be made here. Our definition of $\mathcal{G}$ differs in that $\gamma\ x$ is required to be an element of $\mathcal{E}$ instead of $\mathcal{V}$. This again is necessary in order to let the proofs go through, because of call-by-name semantics.

Notation 51 (Substititution shorthand)**.** *We will write $\delta\gamma$ to denote the composition of δ with γ, i.e. $\delta \circ \gamma$.*

Notation 52 (Simultaneous substitution)**.** *When $\gamma \in (\mathit{Var} \rightharpoonup \mathit{Term})$, we define $\gamma\ e$ to be the simultaneous syntactic substitution of $\gamma\ x$ for every variable $x \in \operatorname{dom}\gamma$ that occurs freely in e.*

Notation 53 (Substitution expansion)**.** *When $\gamma \in (\mathit{Var} \rightharpoonup \mathit{Term})$, $x \notin \operatorname{dom}\gamma$, and $e \in \mathit{Term}$, we let $\gamma[x \mapsto e]$ denote the function that maps any $x' \in \operatorname{dom}\gamma$ onto $\gamma\ x'$ and x onto e. If $x \in \operatorname{dom}\gamma$, then $\gamma[x \mapsto e]$ is undefined.*

Definition 54 (Unary relation)**.** *We define a (unary) relation* $\vDash$ *on terms. More specifically, it relates a type environment* Δ*, an environment* Γ*, a term* e*, and a type* τ*. The notation for* $\langle \Delta, \Gamma, e, \tau \rangle \in \vDash$ *is* $\Delta; \Gamma \vDash e : \tau$*.*

$$\Delta; \Gamma \vDash e : \tau :\Leftrightarrow \Delta \vdash \Gamma \wedge \forall k \geq 0 : \forall \delta \in \mathcal{D}[\![\Delta]\!] : \forall \gamma \in \mathcal{G}^k_\Delta[\![\Gamma]\!]\delta : \delta\gamma\ e \in \mathcal{E}^k_\Delta[\![\tau]\!]\delta.$$

3.4 Proof of the fundamental property

In this section, we prove the fundamental property $\vdash \subseteq \vDash$. We do this by induction on the type relation $\vdash$ and every induction step gets its own subsection. Most of these induction steps or *compatibility lemmas* use extra auxiliary lemmas. Those auxiliary lemmas unique to a compatibility lemma are grouped with it in its subsection. The other ones are presented here.

In general, the structure is quite similar to that of Ahmed.[3] However, once again, there are important differences. In particular, the auxiliary lemmas take into account the differences discussed in Section 3.3. For instance, since our definition of $\mathcal{V}$ in the μ case involves syntactic rather than semantic substitution, the proof of the compatibility lemma for the type rule $:_{\mathbf{muin}}$ is greatly simplified. Other adjustments had to be made in order to accommodate for our CBN semantics and explicit types.

Lemma 55 (There is always a type relation substitution and semantic term substitution)**.** *Suppose* $\Delta; \Gamma \vDash e : \tau$ *and* $k \geq 0$*. Then there exist* $\delta \in \mathcal{D}[\![\Delta]\!]$ *and* $\gamma \in \mathcal{G}^k_\Delta[\![\Gamma]\!]\delta$*.*

Proof. For every $\alpha \in \Delta$, let $\delta^{\mathrm{syn}}\ \alpha$ be $\forall \beta. \beta \to \beta$ and let $\delta^{\mathrm{sem}}\ \alpha$ be the function that maps every $k \geq 0$ onto $\{\Lambda\beta.\lambda x\!:\!\beta.\, x\}$. Clearly, $\delta \in \mathcal{D}[\![\Delta]\!]$.

We define the behaviour of γ on a variable x by parts: if $\Gamma\ x$ is a type variable, say $\Gamma\ x = \alpha \in \Delta$, then we define $\gamma\ x := \Lambda\beta.\lambda x\!:\!\beta.\, x$, which clearly yields $\gamma\ x \in \mathcal{V}^k_\Delta[\![\Gamma\ x]\!]\delta \subseteq \mathcal{E}^k_\Delta[\![\Gamma\ x]\!]\delta$. In all other cases, we define $\gamma\ x :=$ (`out in out in` ... `out in` e), where the nesting of `out-in` pairs goes exactly $k+1$ deep and e is any closed term. This implies that $\gamma\ x$ never evaluates to an irreducible term in $j \leq k$ steps, vacuously yielding $\gamma\ x \in \mathcal{E}^k_\Delta[\![\tau]\!]\delta$. This holds even though $\gamma\ x$ is not necessarily of type τ, or any type for that matter. Generalisation on x implies $\gamma \in \mathcal{G}^k_\Delta[\![\Gamma]\!]\delta$, which finishes the proof. □

Lemma 56 (Free (type) variables in the unary relation)**.** *Suppose that* $\Delta; \Gamma \vDash e : \tau$*. Then* $\mathrm{fv}\, e \subseteq \mathrm{dom}\, \Gamma$ *and* $\mathrm{ftv}\, e \subseteq \Delta$*.*

Proof. By contradiction. Fix an arbitrary $k \geq 0$ and let $\delta \in \mathcal{D}[\![\Delta]\!], \gamma \in \mathcal{G}^k_\Delta[\![\Gamma]\!]\delta$—by Lemma 55 we know these always exist. The assumption then tells us that $\delta\gamma\ e \in \mathcal{E}^k_\Delta[\![\tau]\!]\delta$.

$x \in \mathrm{fv}\, e \setminus \mathrm{dom}\, \Gamma$ would imply that $x \in \mathrm{fv}(\gamma\ e)$ and thus $x \in \mathrm{fv}(\delta\gamma\ e)$, contradicting the statement $\delta\gamma\ e \in \mathcal{E}^k_\Delta[\![\tau]\!]\delta \subseteq \mathit{CTerm}$ derived from instantiation of the assumption.

Similarly, $\alpha \in \mathrm{ftv}\, e \setminus \Delta$ would imply that $\alpha \in \mathrm{ftv}(\delta\gamma\ e)$, contradicting the same statement. □

Lemma 57 (Downward closedness)**.** $\mathcal{V}, \mathcal{E}, \mathcal{G}$ *are* downward closed*. This has the following meaning.*

- *If* $v \in \mathcal{V}^k_\Delta[\![\tau]\!]\delta$ *and* $j < k$*, then* $v \in \mathcal{V}^j_\Delta[\![\tau]\!]\delta$*.*
- *If* $e \in \mathcal{E}^k_\Delta[\![\tau]\!]\delta$ *and* $j < k$*, then* $e \in \mathcal{E}^j_\Delta[\![\tau]\!]\delta$*.*
- *If* $\gamma \in \mathcal{G}^k_\Delta[\![\Gamma]\!]\delta$ *and* $j < k$*, then* $\gamma \in \mathcal{G}^j_\Delta[\![\Gamma]\!]\delta$*.*

Proof. Below, we will prove downward closedness of $\mathcal{V}$ by induction on $\Delta \vdash \tau$.

Downward closedness of $\mathcal{E}$ can be derived from that of $\mathcal{V}$. Fix $e \in \mathcal{E}^k_\Delta[\![\tau]\!]\delta$ and $j < k$. To prove $e \in \mathcal{E}^j_\Delta[\![\tau]\!]\delta$, fix $i \leq j$ and e' such that $e \Downarrow^i e'$. Since $i \leq j$, also $i < k$ so $e' \in \mathcal{V}^{k-i}_\Delta[\![\tau]\!]\delta$. Downward closedness of $\mathcal{V}$ then allows us to finish the proof by concluding $e' \in \mathcal{V}^{j-i}_\Delta[\![\tau]\!]\delta$, since $j - i < k - i$.

Downward closedness of $\mathcal{G}$ is easily derived pointwise from that of $\mathcal{E}$.

Now for the proof of the downward closedness of $\mathcal{V}$. We prove this by induction on $\Delta \vdash \tau$. The case where τ is a type variable is dealt with easily, since $\mathcal{V}^{\text{-}}_\Delta[\![\alpha]\!]\delta$ is constant and thus independent of the index. The other three cases—namely where τ is an arrow type, a universal type quantification, and a recursive type—are similar: we can simply makes use of the transitivity of the strict order $<$ on the naturals and the fact that $v \in \mathcal{V}^k_\Delta[\![\tau]\!]\delta$. ☐

Lemma 58 (Interaction between $\mathcal{E}$ and $\mapsto$)**.** *Suppose $e_1 \in \mathcal{E}^k_\Delta[\![\tau]\!]\delta$ and $e_1 \mapsto^j e_2$ with $j < k$. Then $e_2 \in \mathcal{E}^{k-j}_\Delta[\![\tau]\!]\delta$.*

Proof. Fix a number $i \leq k - j$ and a term e_3 such that $e_2 \Downarrow^i e_3$. We must prove that $e_3 \in \mathcal{V}^{k-j-i}_\Delta[\![\tau]\!]\delta$, which follows immediately from $e_1 \mapsto^{j+i} e_3$ and the assumption on e_1. The proof is finished by the fact that $e_2 \in \mathit{CTerm}$, which follows from $e_1 \mapsto^j e_2$ and Lemma 31. ☐

3.4.1 Variable

Lemma 59 (Compatibility lemma)**.** *Suppose $\Delta \vdash \Gamma$ and $x \in \operatorname{dom}\Gamma$. Then $\Delta; \Gamma \vDash x : \Gamma\ x$.*

Proof. Suppose $\Delta \vdash \Gamma$. We must prove $\Delta; \Gamma \vDash x : \Gamma\ x$. Fix $k \geq 0$, $\delta \in \mathcal{D}[\![\Delta]\!]$, $\gamma \in \mathcal{G}^k_\Delta[\![\Gamma]\!]\delta$. What is left to prove is $\delta\ (\gamma\ x) \in \mathcal{E}^k_\Delta[\![\Gamma\ x]\!]\delta$. We note that $\operatorname{ftv}(\gamma\ x) = \emptyset$ and thus $\delta\ (\gamma\ x) = \gamma\ x$. The definition of γ then gives us that $\gamma\ x \in \mathcal{E}^k_\Delta[\![\Gamma\ x]\!]\delta$. ☐

3.4.2 Term application

Lemma 60. *Given the following: $k \in \mathbb{N}$; Δ; $\delta \in \mathcal{D}[\![\Delta]\!]$; e_1, e_2, e_3 such that $\mathit{irred}\, e_3$ and $e_1\ e_2 \Downarrow^k e_3$; τ_2, τ_3 such that $e_1 \in \mathcal{E}^k_\Delta[\![\tau_2 \to \tau_3]\!]\delta$. Then there exists a natural number $j < k$ and a term b such that $e_1 \mapsto^j \lambda x\!:\!\delta\ \tau_2.\, b$.*

Proof. We prove by induction on k.

First, the base case $k = 0$. Suppose $\Delta, \delta, e_1, e_2, e_3, \tau_2, \tau_3$ are as described. Since $k = 0$ and $e_1\ e_2 \mapsto^k e_3$, we get that $e_1\ e_2 = e_3$, which is irreducible per assumption. Inspection of the rules for the operational semantics shows us that $\mathit{irred}(e_1\ e_2)$ implies that $\mathit{irred}\, e_1$ and that e_1 is not a lambda abstraction. By the assumption on e_1 and by its irreducibility, $e_1 \in \mathcal{V}^0_\Delta[\![\tau_2 \to \tau_3]\!]\delta$. Therefore e_1 *is* in fact a lambda abstraction, which contradicts our earlier finding. Consequently, the base case is vacuously true.

Now, suppose the property holds for k. Fix $\Delta, \delta, e_1, e_2, e_3, \tau_2, \tau_3$ such that all the prerequisites mentioned in the lemma are fulfilled (though for $k + 1$ instead of k). Our proof goal is

$$\exists j < k + 1, b \in \mathit{Term} : e_1 \mapsto^j \lambda x\!:\!\delta\ \tau_2.\, b. \tag{3.3}$$

We split the assumed evaluation $e_1\ e_2 \mapsto^{k+1} e_3$ into two parts, say

$$e_1\ e_2 \mapsto^1 e_4 \mapsto^k e_3,$$

and perform exhaustive case distinction on the evaluation rule used for $e_1\ e_2 \mapsto^1 e_4$:

- It is an application of $\mapsto_{\mathbf{app}}$. This means that e_1 is a lambda abstraction, say $\lambda x : \tau'.\, b'$. What remains to prove is that e_1 specifies the desired input type, i.e. $\tau' = \delta\ \tau_2$. By assumption on e_1 and the fact that *irred* e_1, we see that $e_1 \in \mathcal{V}_{\Delta}^{k+1}[\![\tau_2 \to \tau_3]\!]\delta$, which indeed proves that $\tau' = \delta\ \tau_2$.

- It is an application of $\mapsto_{\mathbf{ectxt}}$. This means that there is some e_5 such that $e_1 \mapsto^1 e_5$ and that $e_4 = e_5\ e_2$. To recapitulate:

$$e_1\ e_2 \mapsto^1 e_5\ e_2 = e_4 \mapsto^k e_3.$$

  We want to apply the induction hypothesis on e_5 and e_2. For this, we need that $e_5 \in \mathcal{E}_{\Delta}^{k}[\![\tau_2 \to \tau_3]\!]\delta$. This follows from Lemma 58 and the fact that $e_1 \in \mathcal{E}_{\Delta}^{k+1}[\![\tau_2 \to \tau_3]\!]\delta$ and $e_1 \mapsto^1 e_5$.

  We can now apply the induction hypothesis to $k, \Delta, \Gamma, e_5, e_2, e_3, \tau_2, \tau_3$. Therefore there exist $i < k$ and b such that $e_5 \mapsto^i \lambda x{:}\delta\ \tau_2.\, b$. Seeing as $i+1 < k+1$ and $e_1 \mapsto^1 e_5 \mapsto^i \lambda x{:}\delta\ \tau_2.\, b$, this proves Eq. (3.3).

□

Lemma 61 (Compatibility lemma)**.** *Suppose* $\Delta;\Gamma \vDash e_1 : \tau_2 \to \tau_3$ *and* $\Delta;\Gamma \vDash e_2 : \tau_2$. *Then* $\Delta;\Gamma \vDash e_1\ e_2 : \tau_3$.

Proof. Suppose $\Delta;\Gamma \vDash e_1 : \tau_2 \to \tau_3$ and $\Delta;\Gamma \vDash e_2 : \tau_2$. We must prove $\Delta;\Gamma \vDash e_1\ e_2 : \tau_3$. Fix $k \geq 0$, $\delta \in \mathcal{D}[\![\Delta]\!]$, $\gamma \in \mathcal{G}_{\Delta}^{k}[\![\Gamma]\!]\delta$. What is left to prove is $\delta\gamma\ (e_1\ e_2) \in \mathcal{E}_{\Delta}^{k}[\![\tau_3]\!]\delta$. Note that $\delta\gamma\ (e_1\ e_2) = (\delta\gamma\ e_1)\ (\delta\gamma\ e_2)$. Fix a term e_3 and a number of steps $j \leq k$ such that $(\delta\gamma\ e_1)\ (\delta\gamma\ e_2) \Downarrow^j e_3$. We need to prove that $e_3 \in \mathcal{V}_{\Delta}^{k-j}[\![\tau_3]\!]\delta$. Using the induction hypothesis, we instantiate Lemma 60 with $j, \Delta, \delta, (\delta\gamma\ e_1), (\delta\gamma\ e_2), e_3, \tau_2, \tau_3$, which results in there being a term e_1' and a number of steps $i < j$ such that $\delta\gamma\ e_1 \mapsto^i \lambda x{:}\delta\ \tau_2.\, e_1'$. In total, we get

$$(\underbrace{(\delta\gamma\ e_1)}_{(1)}\ (\delta\gamma\ e_2)) \mapsto^i \underbrace{((\lambda x{:}\delta\ \tau_2.\, e_1')}_{(2)}\ \underbrace{(\delta\gamma\ e_2))}_{(3)} \mapsto^1 \underbrace{e_1'[\delta\gamma\ e_2/x]}_{(4)} \mapsto^{j-i-1} \underbrace{e_3}_{(5)},$$

where $\underbrace{a}_{(b)}$ represents a claim about a made in item (b) of the following list.

(1) $(\delta\gamma\ e_1) \in \mathcal{E}_{\Delta}^{k}[\![\tau_2 \to \tau_3]\!]\delta$. By assumption on e_1 and $\delta\gamma$.

(2) $(\lambda x{:}\delta\ \tau_2.\, e_1') \in \mathcal{V}_{\Delta}^{k-i}[\![\tau_2 \to \tau_3]\!]\delta$. By definition of $\mathcal{E}$ and by (1).

(3) $(\delta\gamma\ e_2) \in \mathcal{E}_{\Delta}^{k-i-1}[\![\tau_2]\!]\delta$. By assumption on e_2 and $\delta\gamma$. We could have chosen any index, but we choose the greatest one that is still strictly smaller than $k-i$, so that the next step can go through.

(4) $e_1'[\delta\gamma\ e_2/x] \in \mathcal{E}_{\Delta}^{k-i-1}[\![\tau_3]\!]\delta$. By the application of (2) to (3).

(5) $e_3 \in \mathcal{V}_{\Delta}^{k-j}[\![\tau_3]\!]\delta$. By (4) and the fact that $k-i-1-(j-i-1) = k-j$.

□

3.4.3 Term abstraction

Lemma 62. *Suppose* $\gamma \in (\mathit{Var} \rightharpoonup \mathit{Term})$ *with* $x \notin \operatorname{dom}\gamma$*. Suppose* $\forall x' \in \operatorname{dom}\gamma : x \notin \operatorname{fv}(\gamma\ x')$*. Then* $(\gamma\ e)[e'/x] = \gamma[x \mapsto e']\ e$*.*

Proof. We prove this by induction on the shape of e. The cases $e = e_1\ e_2, e = e_1\ \tau, e = \Lambda\alpha.e_1, e = \texttt{in}\ e_1, e = \texttt{out}\ e_1$ are analogous to each other. The simultaneous substitution γ and the single substitution $[e'/x]$ from the left-hand side of the equation move through the relevant level of structure, as does the $\gamma[x \mapsto e']$ from the right-hand side, so that the induction hypothesis can be easily applied. We now deal with the remaining two cases, which we split up further:

- $e = x$. Then $(\gamma\ e)[e'/x] = (\gamma\ x)[e'/x] = x[e'/x] = e' = \gamma[x \mapsto e']\ x = \gamma[x \mapsto e']\ e$, where the second equality holds since $x \notin \operatorname{dom}\gamma$.
- $e = x'$, with $x' \neq x$. Then $(\gamma\ e)[e'/x] = (\gamma\ x')[e'/x] = \gamma\ x' = \gamma[x \mapsto e']\ x' = \gamma[x \mapsto e']\ e$, where the second equality holds since $x \notin \operatorname{fv}(\gamma\ x')$.
- $e = \lambda x' \!:\! \tau.\, e''$. Note that the Barendregt convention has us choose x' such that $x' \neq x$ and $x' \notin \operatorname{dom}\gamma$. Then $(\gamma\ e)[e'/x] = (\lambda x' \!:\! \tau.\, \gamma\ e'')[e'/x] = \lambda x' \!:\! \tau.\, (\gamma\ e'')[e'/x] = \lambda x' \!:\! \tau.\, (\gamma[x \mapsto e']\ e'') = \gamma[x \mapsto e']\ (\lambda x' \!:\! \tau.\, e'') = \gamma[x \mapsto e']\ e$, where the third equation is an application of the induction hypothesis. (Remember that we can do this because we are performing induction on e, not on γ.)

□

Lemma 63 (Compatibility lemma)**.** *Suppose* $\Delta; \Gamma, (x : \tau_x) \vDash e : \tau_e$*. Then* $\Delta; \Gamma \vDash \lambda x \!:\! \tau_x.\, e : \tau_x \to \tau_e$*.*

Proof. Suppose $\Delta; \Gamma, (x : \tau_x) \vDash e : \tau_e$. We must prove that $\Delta; \Gamma \vDash \lambda x : \tau_x.\, e : \tau_x \to \tau_e$. Fix $k \geq 0, \delta \in \mathcal{D}[\![\Delta]\!], \gamma \in \mathcal{G}^k_\Delta[\![\Gamma]\!]\delta$. Since $x \notin \operatorname{dom}\Gamma$,[2] we see that $\delta\gamma\ \lambda x : \tau_x.\, e = \lambda x : \delta\ \tau_x.\, \delta\gamma\ e$, which is already irreducible. Thus it suffices to prove that $\lambda x : \delta\ \tau_x.\, \delta\gamma\ e \in \mathcal{V}^k_\Delta[\![\tau_x \to \tau_e]\!]\delta$. Fix $j < k$ and $e_x \in \mathcal{E}^j_\Delta[\![\tau_x]\!]\delta$. We must prove that $(\delta\ (\gamma\ e))[e_x/x] \in \mathcal{E}^j_\Delta[\![\tau_e]\!]\delta$. First, we note that $(\delta\ (\gamma\ e))[e_x/x] = \delta\ ((\gamma\ e)[e_x/x])$, since $\operatorname{ftv} e_x = \emptyset$. Lemma 62 tells us we can content ourselves with proving $\delta\ (\gamma[x \mapsto e_x]\ e) \in \mathcal{E}^j_\Delta[\![\tau_e]\!]\delta$. For brevity, we will say $\Gamma' := \Gamma, (x : \tau_x)$ and $\gamma' := \gamma[x \mapsto e_x]$.

We will now show that $\gamma' \in \mathcal{G}^j_\Delta[\![\Gamma']\!]\delta$. Looking at the definition of $\mathcal{G}$, we see that we must show that $\forall x' \in \operatorname{dom}\Gamma' : \delta\ (\gamma'\ x') \in \mathcal{E}^j_\Delta[\![\Gamma'\ x']\!]\delta$. By definition of γ, this is true for all $x' \in \operatorname{dom}\Gamma$. The only other case, where $x' = x$, is satisfied by the fact that $\delta\ (\gamma'\ x) = \delta\ e_x = e_x \in \mathcal{E}^j_\Delta[\![\tau_x]\!]\delta$, by definition of γ' and because $\operatorname{ftv} e_x = \emptyset$.

We can now invoke the induction hypothesis—i.e. instantiate $\Delta; \Gamma, (x : \tau_x) \vDash e : \tau_e$—with j, δ, γ' and conclude that indeed $\delta\ (\gamma'\ e)$, which is equal to $(\delta\ (\gamma\ e))[e_x/x]$, is a member of $\mathcal{E}^j_\Delta[\![\tau_e]\!]\delta$. □

3.4.4 Type application

Lemma 64 (Values in the recursive value set are well-typed)**.** *Given* $k \geq 0$ *and* $\langle\Delta, \tau, \delta\rangle \in \mathit{DTD}$*. Then for all* $v \in \mathcal{V}^k_\Delta[\![\tau]\!]\delta$*, we have* $\cdot; \cdot \vdash v : \delta\ \tau$*.*

Proof. We prove by exhaustive case distinction on τ.

The first case is $\tau = \alpha \in \Delta$. We know that $\mathcal{V}^k_\Delta[\![\alpha]\!]\delta = (\delta^{\text{sem}}\ \alpha)^k$, where $\delta^{\text{sem}}\ \alpha \in \mathit{Rel}(\delta\ \tau)$. This implies that $\cdot; \cdot \vdash v : \delta\ \tau$ for all $v \in (\delta^{\text{sem}}\ \alpha)^k$.

[2] This follows implicitly from our mentioning of $\Gamma, (x : \ldots)$, which is only defined if $x \notin \operatorname{dom}\Gamma$.

The other cases are proved by looking at the definition of $\mathcal{V}$ and realising that $\delta\ \tau_1 \to \delta\ \tau_2 = \delta\ (\tau_1 \to \tau_2)$, that $\forall\alpha.\delta\ \tau' = \delta\ (\forall\alpha.\tau')$, and similarly for μ. The $\forall$ case follows from the assumption that $\Delta \vdash \forall\alpha.\tau'$, which implies $\Delta, \alpha \vdash \tau'$. This in turn implies $\alpha \notin \Delta$ (by Definition 18) and therefore $\alpha \notin \operatorname{dom}\delta$. Similar reasoning applies to μ. $\square$

Lemma 57 and Lemma 64 together give us the following property of $\mathcal{V}$.

Corollary 65 (Recursive value sets are semantic type relations)**.** *Given* $\langle\Delta, \tau, \delta\rangle \in DTD$*. Then* $\mathcal{V}^{-}_{\Delta}[\![\tau]\!]\delta \in Rel(\delta\ \tau)$*.*

Lemma 66 (Type substitution in term sets)**.** *Given te following:* $k \geq 0$*;* Δ*;* $\alpha \notin \Delta$*;* $\delta \in \mathcal{D}[\![\Delta]\!]$*;* τ, τ' *such that* $\Delta, \alpha \vdash \tau$ *and* $\Delta \vdash \tau'$*. Let* $\Delta' := (\Delta, \alpha)$ *and* $\delta' := \delta[\alpha \mapsto \langle\delta\ \tau', \mathcal{V}^{-}_{\Delta}[\![\tau']\!]\delta\rangle]$*. Then* $\mathcal{V}^{k}_{\Delta'}[\![\tau]\!]\delta' = \mathcal{V}^{k}_{\Delta}[\![\tau[\tau'/\alpha]]\!]\delta$*. The same equality then clearly also holds when we replace* $\mathcal{V}$ *with* $\mathcal{E}$*.*

Proof. We prove by induction on k, within which we generalise on $\Delta, \alpha, \delta, \tau, \tau'$ and perform case distinction on τ.

Suppose $k = 0$. Fix $\Delta, \alpha, \delta, \tau, \tau'$ as mentioned in the lemma. We will use the abbreviations LHS := $\mathcal{V}^{k}_{\Delta'}[\![\tau]\!]\delta'$ and RHS := $\mathcal{V}^{k}_{\Delta}[\![\tau[\tau'/\alpha]]\!]\delta$. We recall that $\Delta, \alpha \vdash \tau$ and distinguish the following cases.

Suppose $\tau = \alpha$. Then LHS $= \mathcal{V}^{k}_{\Delta'}[\![\alpha]\!]\delta' = (\delta'^{\text{sem}}\ \alpha)^k = \mathcal{V}^{k}_{\Delta}[\![\tau']\!]\delta$. Since $\tau = \alpha$, $\tau[\tau'/\alpha] = \tau'$ and thus LHS = RHS.

Suppose $\tau = \beta$, with $\Delta \ni \beta \neq \alpha$. Then LHS $= \mathcal{V}^{k}_{\Delta'}[\![\beta]\!]\delta' = (\delta'^{\text{sem}}\ \beta)^k = (\delta^{\text{sem}}\ \beta)^k$, which is equal to RHS since $\tau[\tau'/\alpha] = \beta$.

There are three more cases, namely $\tau = \tau_1 \to \tau_2$, $\tau = \forall\beta.\sigma$, and $\tau = \mu\beta.\sigma$. In all of these cases, we realise that the right-hand set in intersection mentioned in the definition of $\mathcal{V}$ is the same in LHS and RHS. (It is “vacuously equal” since $\nexists j < k = 0$.) It thus suffices to prove that in all three cases, $\delta'\ \tau = \tau[\delta\ \tau'/\alpha]$, which is true by Lemma 46.

Now for the induction step. We assume that the property holds for k and prove it holds for $k+1$. We will use the abbreviations LHS := $\mathcal{V}^{k+1}_{\Delta'}[\![\tau]\!]\delta'$ and RHS := $\mathcal{V}^{k+1}_{\Delta}[\![\tau[\tau'/\alpha]]\!]\delta$. Again, we recall $\Delta, \alpha \vdash \tau$ and distinguish five cases. The first two cases—namely $\tau = \alpha$ and $\tau = \beta \neq \alpha$—can be proved similarly to how they were for $k = 0$. The other three cases go as follows:

- $\tau = \tau_1 \to \tau_2$. Looking at the definition of $\mathcal{V}$, we see that

$$\begin{aligned}\text{LHS} &= T(\delta'\ (\tau_1 \to \tau_2)) \cap \{\lambda x\!:\!\delta'\ \tau_1.\, e \mid \forall j < k+1, e' \in \mathcal{E}^{j}_{\Delta'}[\![\tau_1]\!]\delta' : e[e'/x] \in \mathcal{E}^{j}_{\Delta'}[\![\tau_2]\!]\delta'\}\\ \text{RHS} &= T(\delta\ ((\tau_1 \to \tau_2)[\tau'/\alpha])) \cap \{\lambda x\!:\!\delta\ (\tau_1[\tau'/\alpha]).\, e \mid \forall j < k+1, e' \in \mathcal{E}^{j}_{\Delta}[\![\tau_1[\tau'/\alpha]]\!]\delta :\\ &\qquad e[e'/x] \in \mathcal{E}^{j}_{\Delta}[\![\tau_2[\tau'/\alpha]]\!]\delta\}.\end{aligned}$$

  We see that $\mathcal{E}^{j}_{\Delta'}[\![\tau_1]\!]\delta' = \mathcal{E}^{j}_{\Delta}[\![\tau_1[\tau'/\alpha]]\!]\delta$ and $\mathcal{E}^{j}_{\Delta'}[\![\tau_2]\!]\delta' = \mathcal{E}^{j}_{\Delta}[\![\tau_2[\tau'/\alpha]]\!]\delta$ for all $j < k+1$ by induction. All other apparent differences between LHS and RHS are resolved by Lemma 46.

- $\tau = \forall\beta.\sigma$. Then Barendregt has us assume that $\beta \notin (\Delta, \alpha)$. We see that

$$\begin{aligned}\text{LHS} &= T(\forall\beta.\delta'\ \sigma) \cap \{(\Lambda\beta.e) \in CVal \mid \forall j < k : \forall\rho \in CType : \forall\chi \in Rel(\rho) :\\ &\qquad e[\rho/\beta] \in \mathcal{E}^{j}_{\Delta',\beta}[\![\sigma]\!]\delta'[\beta \mapsto \langle\rho, \chi\rangle]\}\\ \text{RHS} &= T(\forall\beta.\delta\ (\sigma[\tau'/\alpha])) \cap \{(\Lambda\beta.e) \in CVal \mid \forall j < k : \forall\rho \in CType : \forall\chi \in Rel(\rho) :\\ &\qquad e[\rho/\beta] \in \mathcal{E}^{j}_{\Delta,\beta}[\![\sigma[\tau'/\alpha]]\!]\delta[\beta \mapsto \langle\rho, \chi\rangle]\}.\end{aligned}$$

  Again, the typing issues are resolved by Lemma 46: $\forall\beta.(\delta'\ \sigma) = \forall\beta.\delta\ (\sigma[\tau'/\alpha])$.

Also, we apply the induction hypothesis to show that

$$\mathcal{E}^{j}_{\Delta',\beta}[\![\sigma]\!]\delta'[\beta \mapsto \langle\rho,\chi\rangle] = \mathcal{E}^{j}_{\Delta,\beta}[\![\sigma[\tau'/\alpha]]\!]\delta[\beta \mapsto \langle\rho,\chi\rangle].$$

For this to go through, it is essential that we realise that the generalisation over type environments and type relation substitutions happens within the induction. The reader must not make the mistake to think that the role of the type variable mentioned in the lemma is fulfilled by β. The notation is misleading: to reflect that the relevant type variable is still α, perhaps writing Δ', β as $(\Delta, \beta), \alpha$ would have been better, were it not so lengthy. A similar thing holds for $\delta'[\beta \mapsto \langle\rho,\chi\rangle]$, which, we must not forget, is equal to $\delta[\beta \mapsto \langle\rho,\chi\rangle][\alpha \mapsto \langle\delta\ \tau', \mathcal{V}^{-}_{\Delta}[\![\tau']\!]\delta\rangle] \in \mathcal{D}[\![(\Delta,\beta),\alpha]\!]$.

- $\tau = \mu\beta.\sigma$. Again, $\beta \notin (\Delta,\alpha)$ by the Barendregt convention. Now,

  $$\text{LHS} = T(\mu\beta.\delta'\ \sigma) \cap \{(\texttt{in}\ e) \in \mathit{CVal} \mid \forall j < k : e \in \mathcal{E}^{j}_{\Delta'}[\![\sigma[\mu\beta.\sigma/\beta]]\!]\delta'\}$$
  $$\text{RHS} = T(\mu\beta.\delta\ (\sigma[\tau'/\alpha])) \cap \{(\texttt{in}\ e) \in \mathit{CVal} \mid \forall j < k : e \in \mathcal{E}^{j}_{\Delta}[\![\sigma[\tau'/\alpha][\mu\beta.\sigma[\tau'/\alpha]/\beta]]\!]\delta\}.$$

  Once again, the typing difference is resolved by Lemma 46: $\mu\beta.\delta'\ \sigma = \mu\beta.\delta\ (\sigma[\tau'/\alpha])$.

  To prove $\mathcal{E}^{j}_{\Delta'}[\![\sigma[\mu\beta.\sigma/\beta]]\!]\delta'$ is equal to $\mathcal{E}^{j}_{\Delta}[\![\sigma[\tau'/\alpha][\mu\beta.\sigma[\tau'/\alpha]/\beta]]\!]\delta$, we first realise that $\sigma[\tau'/\alpha][\mu\beta.\sigma[\tau'/\alpha]/\beta] = \sigma[\mu\beta.\sigma/\beta][\tau'/\alpha]$, and then apply the induction hypothesis. This is the first time we apply the induction hypothesis to another type than the given one, i.e. in this case, not to σ. It does not matter, however, since we perform induction not on the type but on the index.

$\square$

Lemma 67. *Given the following: $k \in \mathbb{N}$; Δ; $\delta \in \mathcal{D}[\![\Delta]\!]$; $e_1, e_2, alpha, \tau, \tau'$ such that $e_1 \in \mathcal{E}^{k}_{\Delta}[\![\forall\alpha.\tau]\!]\delta$ and $e_1\ \tau' \Downarrow^{k} e_2$. Then there exists a natural number $j < k$ and a term b such that $e_1 \mapsto^{j} \Lambda\alpha.b$.*

Proof. Similar to that of Lemma 60. We prove by induction on k.

Suppose $k = 0$. Then $e_1\ \tau'$ is irreducible, and therefore *irred* e_1 with e_1 *not* being a type abstraction. At the same time, the assumption on e_1 and $e_1 \Downarrow^{0} e_1$ gives us $e_1 \in \mathcal{V}^{0}_{\Delta}[\![\forall\alpha.\tau]\!]\delta$, meaning that e_1 *is* a type abstraction. Contradiction.

Now suppose the property holds for k. Fix $\Delta, \delta, \tau, \tau', \alpha, e_1, e_2$ such that all the prerequisites mentioned in the lemma are fulfilled (though for $k+1$ instead of k). Our proof goal is

$$\exists j < k+1, b \in \mathit{Term} : e_1 \mapsto^{j} \Lambda\alpha.b. \tag{3.4}$$

We split up the assumed evaluation $e_1\ \tau' \mapsto^{k+1} e_2$ into two parts, say

$$e_1\ \tau' \mapsto^{1} e_3 \mapsto^{k} e_2,$$

and perform exhaustive case distinction on the evaluation rule used for $e_1\ \tau' \mapsto^{1} e_3$.

- It is an application of $\mapsto_{\mathbf{tapp}}$. This means that e_1 is already a type abstraction. Therefore *irred* e_1 and thus $e_1 \in \mathcal{V}^{k+1}_{\Delta}[\![\forall\alpha.\tau]\!]\delta$. This proves Eq. (3.4).

- It is an application of $\mapsto_{\mathbf{ectxt}}$. This means that there is some e_4 such that $e_1 \mapsto^{1} e_4$ and $e_3 = e_4\ \tau'$. To recapitulate:

  $$e_1\ \tau' \mapsto^{1} e_4\ \tau' = e_3 \mapsto^{k} e_2.$$

  We apply the induction hypothesis on e_4. Note that the requirement, $e_4 \in \mathcal{E}^{k}_{\Delta}[\![\forall\alpha.\tau]\!]\delta$, follows from Lemma 58 and the fact that $e_1 \in \mathcal{E}^{k+1}_{\Delta}[\![\forall\alpha.\tau]\!]\delta$. Therefore there exist $i < k$ and b such that $e_1 \mapsto^{1} e_4 \mapsto^{i} \Lambda\alpha.b$. Seeing as $i + 1 < k + 1$, this proves Eq. (3.4).

$\square$

Lemma 68 (Compatibility lemma)**.** *Suppose* $\Delta; \Gamma \vDash e : \forall\alpha.\tau$ *and* $\Delta \vdash \tau'$*. Then* $\Delta; \Gamma \vDash e\ \tau' : \tau[\tau'/\alpha]$.

Proof. Suppose $\Delta; \Gamma \vDash e : \forall\alpha.\tau$ and $\Delta \vdash \tau'$. We must prove $\Delta; \Gamma \vDash e\ \tau' : \tau[\tau'/\alpha]$. Fix $k \geq 0, \delta \in \mathcal{D}[\![\Delta]\!], \gamma \in \mathcal{G}^k_\Delta[\![\Gamma]\!]\delta$. Fix $j \leq k, e' \in \mathit{Term}$ such that $\delta\gamma\ (e\ \tau') \Downarrow^j e'$. (Note that $\delta\gamma\ (e\ \tau') = (\delta\gamma\ e)\ (\delta\ \tau')$.) We must prove that $e' \in \mathcal{V}^{k-j}_\Delta[\![\tau[\tau'/\alpha]]\!]\delta$. We instantiate Lemma 67 with $j, \Delta, \delta, \tau, \alpha, (\delta\gamma\ e), e', \delta\ \tau'$. (Note that we apply the induction hypothesis to get that $\delta\gamma\ e \in \mathcal{E}^k_\Delta[\![\forall\alpha.\tau]\!]\delta$.) This gives us the existence of a number $i < j$ and a term b such that $\delta\gamma\ e \mapsto^i \Lambda\alpha.b$. In total, we get

$$\underbrace{(\delta\gamma\ e)}_{(1)}\ \delta\ \tau' \mapsto^i \underbrace{(\Lambda\alpha.b)}_{(2)}\ \delta\ \tau' \mapsto^1 \underbrace{b[\delta\ \tau'/\alpha]}_{(3)} \mapsto^{j-i-1} \underbrace{e'}_{(4)},$$

where $\underbrace{a}_{(b)}$ represents a claim about a made in item (b) of the following list.

(1) $\delta\gamma\ e \in \mathcal{E}^k_\Delta[\![\forall\alpha.\tau]\!]\delta$. By the assumption on e and γ.

(2) $\Lambda\alpha.b \in \mathcal{V}^{k-i}_\Delta[\![\forall\alpha.\tau]\!]\delta$. By (1).

(3) $b[\delta\ \tau'/\alpha] \in \mathcal{E}^{k-i-1}_{\Delta,\alpha}[\![\tau]\!]\delta[\alpha \mapsto \langle\delta\ \tau', \chi\rangle]$, where $\chi := \mathcal{V}^{\text{-}}_\Delta[\![\tau']\!]\delta$. We know by (2) and the definition of $\mathcal{V}$ that the statement holds for all $\chi \in \mathit{Rel}(\delta\ \tau')$ and all indices strictly smaller than $k-i$. That our choice of χ is element of $\mathit{Rel}(\delta\ \tau')$, is a direct consequence of Corollary (65).

(4) $e' \in \mathcal{V}^{k-j}_{\Delta,\alpha}[\![\tau]\!]\delta[\alpha \mapsto \langle\delta\ \tau', \mathcal{V}^{\text{-}}_\Delta[\![\tau']\!]\delta\rangle]$. This follows from (3), the fact that $\mathit{irred}\, e'$, and the fact that $k-i-1-(j-i-1) = k-j$.

We then finish the proof by realising that $\mathcal{V}^{k-j}_{\Delta,\alpha}[\![\tau]\!]\delta[\alpha \mapsto \langle\delta\ \tau', \mathcal{V}^{\text{-}}_\Delta[\![\tau']\!]\delta\rangle] = \mathcal{V}^{k-j}_\Delta[\![\tau[\tau'/\alpha]]\!]\delta$, because of Lemma 66. $\square$

3.4.5 Type abstraction

Lemma 69 (Redundant type variables make no difference)**.** *Given the following:* $k \geq 0$*;* $\langle\Delta, \tau, \delta\rangle \in \mathit{DTD}$*;* $\alpha \notin \Delta$*. Let* $\Delta' := (\Delta, \alpha)$ *and* $\delta' := \delta[\alpha \mapsto \langle\sigma, \chi\rangle]$ *for some* $\sigma \in \mathit{Type}, \chi \in \mathit{Rel}(\sigma)$*. (Clearly,* $\delta' \in \mathcal{D}[\![\Delta']\!]$*.) Then* $\mathcal{V}^k_\Delta[\![\tau]\!]\delta = \mathcal{V}^k_{\Delta'}[\![\tau]\!]\delta'$.

Under the same conditions, $\mathcal{E}^k_\Delta[\![\tau]\!]\delta = \mathcal{E}^k_{\Delta'}[\![\tau]\!]\delta'$ *holds. If, for any given* Γ*, we replace the condition* $\langle\Delta, \tau, \delta\rangle \in \mathit{DTD}$ *with* $(\delta \in \Delta \wedge \Delta \vdash \Gamma)$*, then* $\mathcal{G}^k_\Delta[\![\Gamma]\!]\delta = \mathcal{G}^k_{\Delta'}[\![\Gamma]\!]\delta'$ *holds.*

Proof. We first realise, by expanding the definition of $\mathcal{E}$, that if the lemma holds for $\mathcal{V}$, it holds for $\mathcal{E}$. We also see that the lemma for $\mathcal{G}$ holds as long as it holds for $\mathcal{E}$: we can apply the equality for $\mathcal{E}$ pointwise to all $x \in \operatorname{dom}\Gamma$. Therefore, only the lemma for $\mathcal{V}$ remains to prove. We do this by induction on k, within which we perform exhaustive case distinction on $\Delta \vdash \tau$.

The base case $k = 0$ is easy. If $\tau = \beta$, then $\beta \neq \alpha$ and thus $\delta\ \beta = \delta'\ \beta$ proves the equality. In all other cases, looking at the definition of $\mathcal{V}$ as the intersection of two sets, we see that it suffices to prove two equalities.

First, the left-hand side sets $T(\delta\ \tau)$ and $T(\delta\ \tau')$ must be equal. This is clearly the case since $\delta' = \delta[\alpha \mapsto \ldots]$, while $\alpha \notin \operatorname{ftv}\tau$. (The latter holds by Lemma 21, because $\Delta \vdash \tau$ with $\alpha \notin \Delta$.)

Second, the right-hand side sets governing "term shape" must be equal. They are, since in all cases the restrictions mentioned in the set builder notation are vacuously satisfied. (Remember that $k = 0$.)

The induction step is proved in much the same way. The case of $\tau = \beta \neq \alpha$ is identical. In all other cases, the typing sets $T(\delta\ \tau)$ and $T(\delta'\ \tau)$ are again equal. The term shape sets are also equal, though this time not by vacuous satisfaction but by induction on the index. $\square$

Lemma 70 (Compatibility lemma)**.** *Suppose* $\Delta, \alpha; \Gamma \vDash e : \tau$ *and* $\Delta \vdash \Gamma$*. Then* $\Delta; \Gamma \vDash \Lambda\alpha.e : \forall\alpha.\tau$*.*

Proof. Suppose $\Delta, \alpha; \Gamma \vDash e : \tau$ and $\Delta \vdash \Gamma$. We must prove that $\Delta; \Gamma \vDash \Lambda\alpha.e : \forall\alpha.\tau$. Fix $k \geq 0, \delta \in \mathcal{D}[\![\Delta]\!], \gamma \in \mathcal{G}^k_\Delta[\![\Gamma]\!]\delta$. Since $\delta\gamma\ \Lambda\alpha.e = \Lambda\alpha.\delta\gamma\ e$ is already irreducible,[3] we must prove that $\Lambda\alpha.\delta\gamma\ e \in \mathcal{V}^k_\Delta[\![\forall\alpha.\tau]\!]\delta$. Fix a number $j < k$, a type $\tau' \in \mathit{CType}$, and a semantic relation $\chi \in \mathit{Rel}(\tau')$. If we let $\Delta' := (\Delta, \alpha)$ and $\delta' := \delta[\alpha \mapsto \langle\tau', \chi\rangle]$, what remains to be proved can be written as $(\delta\gamma\ e)[\tau'/\alpha] \in \mathcal{E}^j_{\Delta'}[\![\tau]\!]\delta'$.

Note that $\delta' \in \mathcal{D}[\![\Delta']\!]$ and $\gamma \in \mathcal{G}^k_{\Delta'}[\![\Gamma]\!]\delta'$, because of Lemma 69. Therefore, by instantiation of the induction hypothesis with δ' and γ, $\delta'\ (\gamma\ e) \in \mathcal{E}^j_{\Delta'}[\![\tau]\!]\delta'$. The equality $\delta'\ (\gamma\ e) = (\delta\gamma\ e)[\tau'/\alpha]$, derived from Lemma 45, finishes the proof. $\square$

3.4.6 Type unfolding

Lemma 71. *Given the following:* $k \in \mathbb{N}$*;* Δ*;* $\delta \in \mathcal{D}[\![\Delta]\!]$*;* e_1, e_2, α, τ *such that* $e_1 \in \mathcal{E}^k_\Delta[\![\mu\alpha.\tau]\!]\delta$ *and* $(\texttt{out}\ e_1) \Downarrow^k e_2$*. Then there exists a natural number* $j < k$ *and a term* b *such that* $e_1 \mapsto^j \texttt{in}\ b$*.*

Proof. Similar to Lemma 60 and Lemma 67. We prove by induction on k.

Suppose $k = 0$. Then $\mathit{irred}(\texttt{out}\ e_1)$ and thus also $\mathit{irred}\, e_1$ (by contraposition of $\mapsto_{\textbf{ectxt}}$). By assumption, then, $e_1 \in \mathcal{V}^{0-0}_\Delta[\![\mu\alpha.\tau]\!]\delta$. This means e_1 is equal to $\texttt{in}\ e$ for some term e. This contradicts the earlier finding that $\texttt{out}\ e_1 = \texttt{out in}\ e$ is irreducible (by $\mapsto_{\textbf{outin}}$).

Suppose the property holds for k. Fix $\Delta, \delta, \tau, \alpha, e_1, e_2$ such that all the prerequisites mentioned in the lemma are fulfilled (though for $k+1$ instead of k). Our proof goal is

$$\exists j < k, b \in \mathit{Term} : e_1 \mapsto^j \texttt{in}\ b.$$

We split up the assumed evaluation $\texttt{out}\ e_1 \mapsto^{k+1} e_2$ into two parts, say

$$\texttt{out}\ e_1 \mapsto^1 e_3 \mapsto^k e_2,$$

and perform exhaustive case distinction on the evaluation rule used for $\texttt{out}\ e_1 \mapsto^1 e_3$.

- It is an application of $\mapsto_{\textbf{outin}}$. Then $e_1 = \texttt{in}\ e_3$. Thus, $e_1 \mapsto^0 \texttt{in}\ e_3$, with $0 < k+1$, which ends the proof.
- It is an application of $\mapsto_{\textbf{ectxt}}$. Then there is some e_4 such that $e_1 \mapsto^1 e_4$ and $e_3 = \texttt{out}\ e_4$. When we then apply the induction hypothesis to $\texttt{out}\ e_4 \mapsto^k e_2$—note that $e_4 \in \mathcal{E}^k_\Delta[\![\mu\alpha.\tau]\!]\delta$ by Lemma 58—we get the existence of $j < k$ and b such that $e_4 \mapsto^j \texttt{in}\ b$. We then see that $e_1 \mapsto^{j+1} \texttt{in}\ b$.

$\square$

Lemma 72 (Compatibility lemma)**.** *Suppose* $\Delta; \Gamma \vDash e : \mu\alpha.\tau$*. Then* $\Delta; \Gamma \vDash \texttt{out}\ e : \tau[\mu\alpha.\tau/\alpha]$*.*

Proof. Suppose $\Delta; \Gamma \vDash e : \mu\alpha.\tau$. We must prove that $\Delta; \Gamma \vDash \texttt{out}\ e : \tau[\mu\alpha.\tau/\alpha]$. Fix $k \geq 0, \delta \in \mathcal{D}[\![\Delta]\!], \gamma \in \mathcal{G}^k_\Delta[\![\Gamma]\!]\delta$. Then fix $j \leq k, e' \in \mathit{Term}$ such that $\delta\gamma\ (\texttt{out}\ e) \Downarrow^j e'$. We must prove that $e' \in \mathcal{V}^{k-j}_\Delta[\![\tau[\mu\alpha.\tau/\alpha]]\!]\delta$.

[3] Note that the δ moves into the Λ construction since $\alpha \notin \operatorname{dom}\delta$.

Note that $\delta\gamma$ $(\texttt{out}\ e) = \texttt{out}\ (\delta\gamma\ e)$. We apply the induction hypothesis and get that $\delta\gamma\ e \in \mathcal{E}^k_\Delta[\![\mu\alpha.\tau]\!]\delta$. We can then instantiate Lemma 71 with j, Δ, δ, τ, α, $(\delta\gamma\ e)$, e'. This gives us the existence of a number $i < j$ and a term e'' such that $\delta\gamma\ e \mapsto^i \texttt{in}\ e''$. In total, we get

$$\texttt{out}\ \underbrace{(\delta\gamma\ e)}_{(1)} \mapsto^i \texttt{out}\ \underbrace{(\texttt{in}\ e'')}_{(2)} \mapsto^1 \underbrace{e''}_{(3)} \mapsto^{j-i-1} \underbrace{e'}_{(4)},$$

where $\underbrace{a}_{(b)}$ represents a claim about a made in item (b) of the following list.

(1) $\delta\gamma\ e \in \mathcal{E}^k_\Delta[\![\mu\alpha.\tau]\!]\delta$. By the induction hypothesis.

(2) $\texttt{in}\ e'' \in \mathcal{V}^{k-i}_\Delta[\![\mu\alpha.\tau]\!]\delta$. By (1) and the fact that $\delta\gamma\ e \mapsto^i \texttt{in}\ e''$.

(3) $e'' \in \mathcal{E}^{k-i-1}_\Delta[\![\tau[\mu\alpha.\tau/\alpha]]\!]\delta$. By (2). This follows from the definition of $\mathcal{V}$ for μ-types. It holds for all indices strictly smaller than $k - i$, including $k - i - 1$.

(4) $e' \in \mathcal{V}^{k-j}_\Delta[\![\tau[\mu\alpha.\tau/\alpha]]\!]\delta$. This follows directly from (3) and the fact that e' is assumed to be irreducible. (Note that $k - j = (k - i - 1) - (j - i - 1)$.)

□

3.4.7 Type folding

Lemma 73 (Compatibility lemma)**.** *Suppose* $\Delta;\Gamma \vDash e : \tau[\mu\alpha.\tau/\alpha]$. *Then* $\Delta;\Gamma \vDash \texttt{in}\ e : \mu\alpha.\tau$.

Proof. Suppose $\Delta;\Gamma \vDash e : \tau[\mu\alpha.\tau/\alpha]$. We must prove that $\Delta;\Gamma \vDash \texttt{in}\ e : \mu\alpha.\tau$. Fix $k \geq 0, \delta \in \mathcal{D}[\![\Delta]\!], \gamma \in \mathcal{G}^k_\Delta[\![\Gamma]\!]\delta$. Since $\delta\gamma\ (\texttt{in}\ e) = \texttt{in}\ \delta\gamma\ e$ is already irreducible, we must prove that $(\texttt{in}\ \delta\gamma\ e) \in \mathcal{V}^k_\Delta[\![\mu\alpha.\tau]\!]\delta$. In order to do that, we must prove that $(\delta\gamma\ e) \in \mathcal{E}^j_\Delta[\![\tau[\mu\alpha.\ \tau/\alpha]]\!]\delta$ for all $j < k$. We get this by instantiation of the assumption with δ, γ, together with Lemma 57. □

3.5 Conclusion

From all the compatibility lemmas in the previous sections, that function as induction steps, we conclude that indeed the induction goes through. This constitutes the first part of the proof that our language is type-safe.

Proposition 74 (Syntactic typing implies semantic typing)**.** *Given* Δ, Γ, e, τ. *Suppose that* $\Delta;\Gamma \vdash e : \tau$. *Then* $\Delta;\Gamma \vDash e : \tau$.

Proof. We prove this by induction on $\Delta;\Gamma \vdash e : \tau$. In other words, we prove that $\vDash$ is closed under the type rules on which $\vdash$ is defined inductively. All necessary induction steps have been proved. See Lemmas 59, 61, 63, 68, 70, 72, 73. □

We now prove the second, shorter part of the type-safety proof:

Proposition 75 (Semantic typability implies safety)**.** *Given* e, τ. *Suppose* $\cdot;\cdot \vDash e : \tau$. *Then* *safe e.*

Proof. This follows from unwinding and applying the definitions. Fix a term e' and a number of steps $k \geq 0$ such that $e \Downarrow^k e'$. We must prove that $e' \in \mathit{Val}$. We instantiate the assumption with k, $\emptyset \in \mathcal{D}[\![\cdot]\!]$, and $\emptyset \in \mathcal{G}^k_\Delta[\![\cdot]\!]\emptyset$. We get that $\emptyset\ (\emptyset\ e) = e \in \mathcal{E}^k_\Delta[\![\tau]\!]\emptyset$. This we instantiate with $e \Downarrow^k e'$. We get $e' \in \mathcal{V}^0_\Delta[\![\tau]\!]\emptyset$. By definition of (the codomain of) $\mathcal{V}$, this means $e' \in \mathit{Val}$. □

We conclude that the language is type-safe:

Theorem 76 (Type-safety)**.** *The language $\lambda\forall\mu$ is type-safe.*

Proof. Suppose that $\cdot;\cdot \vdash e : \tau$. Then, by Prop. 74, we get $\cdot;\cdot \vDash e : \tau$. Prop. 75 then tells us that *safe* e, which finishes the proof. $\square$

Chapter 4

Binary relation and contextual equivalence

In this chapter, we build up to and achieve the main result of this thesis. We start by making precise the *contextual equivalence* relation between terms. Loosely speaking, two terms e, e' are *contextually equivalent* (notation: $e \approx^{ctx} e'$) iff they "behave the same way in all situations". The concept of "situation" will be captured by *contexts*, which are essentially terms with exactly one hole (like evaluation contexts). We then define a binary step-indexed logical relation $\diamond$ (notation: $e \diamond e'$) based on Ahmed's relation $\approx$.[3] This step will be very similar to the definition of $\vDash$ in Chapter 3.

Once all these definitions are in place, we start proving the intended theorem: $\diamond \subseteq \approx^{ctx}$, or $e \diamond e' \Rightarrow e \approx^{ctx} e'$. We will refer to this property as the *soundness* of $\diamond$, and to $\diamond$ as a *sound proof technique* for contextual equivalence. In order to prove that two terms are contextually equivalent, we then need only prove that they are related by $\diamond$. We will see in Section 4.5 that at least in some cases, the latter is indeed easier than the former. This makes proving $\approx^{ctx}$ via $\diamond$ a useful proof technique. Note that it is not our goal to achieve *completeness*, i.e. the converse $\approx^{ctx} \subseteq \diamond$.

Section three performs the first part of the proof by showing that a fundamental property $\vdash \subseteq \diamond$ holds for $\diamond$: all well-typed terms are related to themselves by $\diamond$. We will prove this by induction on $\vdash$, using one separate compatibility lemma per induction step. Though we will not use this result in the proof of the main theorem, we *will* reuse its compatibility lemmas in the second step.

This second step is displayed in section four. There, we prove a property called *context monotonicity*, which says that filling in two terms related by $\diamond$ into the same context results in two terms that are again related by $\diamond$. The final step is then to combine this into a direct proof that terms related by $\diamond$ are contextually equivalent.

Before we do this all, even before we properly define contextual equivalence, we take the time to look at some examples. This will give us a feeling for which terms are contextually equivalent and which are not. Note that we will not prove any contextual equivalences in the example section. For one, the loose definitions used will not allow for rigorous proof. For another, if it were so easy, we might not be so interested in finding other sound proof techniques.

4.1 Examples

Suppose, in this section only, that we add an extra type $\mathbb{B}$ to the language, namely that of *booleans*. The intended meaning of $\mathbb{B}$ is that we can express the concepts *true* and *false* within the language.

We add three disjuncts to the grammar for *Term*:

$$e ::= \dots \mid \texttt{if}\, e\, \texttt{then}\, e\, \texttt{else}\, e \mid \texttt{true} \mid \texttt{false}$$

the latter two of which are also added to the grammar for *Val*. Concerning the operational semantics, we add $\texttt{if}\, E\, \texttt{then}\, e\, \texttt{else}\, e$ to the grammar of evaluation contexts and add the following two rules to $\mapsto$:

$$\frac{}{\texttt{if}\,\texttt{true}\,\texttt{then}\, e_2\, \texttt{else}\, e_3 \mapsto e_2} \qquad \frac{}{\texttt{if}\,\texttt{false}\,\texttt{then}\, e_2\, \texttt{else}\, e_3 \mapsto e_3}.$$

This makes the `ifthenelse` construct strict in the first argument and lazy in the other two: the first must be fully evaluated before the construct is eliminated, while the latter are kept untouched. We also add the typing rule

$$\frac{\Delta;\Gamma \vdash e_1 : \mathbb{B} \qquad \Delta;\Gamma \vdash e_2 : \tau \qquad \Delta;\Gamma \vdash e_3 : \tau}{\Delta;\Gamma \vdash \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3 : \tau}.$$

We now define *contexts* similarly to evaluation contexts. They should be thought of as terms with exactly one hole to be filled in with a term. More formally, we define the set *Ctx* to be generated by the same grammar as *Term*, except that the non-terminal is named C instead of e, an extra disjunct (-) is added, and in every disjunct exactly one of the occurring non-terminals e is replaced by C:

$$\begin{aligned} C ::= {} & (\text{-}) \mid C\ e \mid e\ C \mid \lambda x{:}\tau.\, C \mid C\ \tau \mid \Lambda\alpha.C \mid \texttt{in}\ C \mid \texttt{out}\ C \\ & \mid \texttt{if}\, C\, \texttt{then}\, e\, \texttt{else}\, e \mid \texttt{if}\, e\, \texttt{then}\, C\, \texttt{else}\, e \mid \texttt{if}\, e\, \texttt{then}\, e\, \texttt{else}\, C \end{aligned}$$

The notion of *filling in* a context is defined similarly to evaluation contexts.

We are now ready to say what it means for terms to be contextually equivalent.

Definition 77 (In this section only)**.** *Two terms e, e' are* contextually equivalent *iff for every "suitable" context C and every value v of type $\mathbb{B}$, we have that $C[e] \Downarrow v$ iff $C[e'] \Downarrow v$.*

Defining "suitable" would lead us too far right now, but a suitable C can be considered one such that $C[e]$ and $C[e']$ are of type $\mathbb{B}$. This notion of contextual equivalence attempts to capture that whatever situation e and e' are used in, the result will be the same and an observer will see no difference.

Why the booleans At this point, the reader might wonder why we introduced the booleans and what is so special about them that only they seem to matter in the definition of contextual equivalence. In this paragraph, we will show with imprecise arguments what would happen if we defined contextual equivalence such that the mentioned value v can be of any, universally quantified-over type τ.

Definition 78 (Alternative definition for this paragraph only)**.** *Two terms e, e' are* contextually equivalent *iff for every type τ, every "suitable" context C, and every value v of type τ, we have that $C[e] \Downarrow v$ iff $C[e'] \Downarrow v$. (This time "suitable" means, approximately, that $C[e], C[e']$ are of type τ.)*

Consider the terms

$$\begin{aligned} l &:= id = \Lambda\alpha.\lambda x\!:\!\alpha.\, x, \\ l' &:= \Lambda\alpha.\lambda x\!:\!\alpha.\, \texttt{out in}\ x. \end{aligned}$$

The reader should be convinced that we want these terms to be considered contextually equivalent. Though they might not be syntactically equal, we see that $l\ \tau\ e$ and $l'\ \tau\ e$ reduce to the same term e.

We now show through a counterexample, though, that under the new, hypothetical definition of this paragraph, with a universally quantified-over type τ instead of just booleans, l, l' are not contextually equivalent. We propose the type $\forall\alpha.(\alpha \to \alpha)$ and the trivial context $C :=$ (-). Note that C is "suitable" since $C[l], C[l']$ are of type $\forall\alpha.(\alpha \to \alpha)$. Now the value to which $C[l] = l$ evaluates is simply l itself and, similarly, $C[l'] = l'$ evaluates to l'. However, these values l, l' are not the same, as the definition of contextual equivalence demands. Therefore, l, l' are not contextually equivalent under this hypothetical definition.

This might lead one to change the hypothetical definition once again, namely such that the values to which $C[l], C[l']$ evaluate need not be equal but merely equivalent in some sense. However, this would be circular, as equivalence is exactly the property we are trying to capture. The problem in the counterexample was that it took our human interpretation to see that l and l' should be equivalent because they reduced to the same thing after applying them to a type τ and a term e. Other pairs of terms we consider equivalent might need different numbers of arguments before their reduction paths coincide.

This is the reason we introduced $\mathbb{B}$. It acts as a "ground type", in the sense that we need no extra arguments to determine if two values of type $\mathbb{B}$ behave the same; we can distinguish between different values of type $\mathbb{B}$ syntactically. Of course, beside the intended `true` and `false`, other terms like $\Omega_{\mathbb{B}}$ are of type $\mathbb{B}$. However, note that those are not values.

Examples Rejecting the alternative Definition 78 in favor of Definition 77, we discuss a few examples. First of all, we make it clear that for two terms to be of the same type is not sufficient to be contextually equivalent. Consider, for example:

$$\begin{aligned} first &:= \Lambda\alpha.\lambda x\!:\!\alpha.\, \lambda y\!:\!\alpha.\, x, \\ second &:= \Lambda\alpha.\lambda x\!:\!\alpha.\, \lambda y\!:\!\alpha.\, y. \end{aligned}$$

They are both of type $\forall\alpha.\alpha \to \alpha \to \alpha$, but clearly have different behaviour. We can show this by providing a counterexample against the property of contextual equivalence as defined in Definition 77. Let C be (-) $\mathbb{B}$ `true false`. Then $C[first] = first\ \mathbb{B}\ \texttt{true false} \Downarrow \texttt{true}$ and $C[second] = second\ \mathbb{B}\ \texttt{true false} \Downarrow \texttt{false}$. We see that $\texttt{true} \neq \texttt{false}$.

Second, we recall the example pair l, l' used to expose the problems with Definition 78:

$$\begin{aligned} l &:= id = \Lambda\alpha.\lambda x\!:\!\alpha.\, x, \\ l' &:= \Lambda\alpha.\lambda x\!:\!\alpha.\, \texttt{out in}\ x. \end{aligned}$$

This time, since we are using Definition 77, the context $C :=$ (-) is not considered suitable, since filling in l or l' does not result in a term of type $\mathbb{B}$. In fact, we will give a loose argument

$$\begin{tikzcd}
(\lambda x\!:\!\tau_{id}.\, (x\ \mathbb{B})\ (x\ \mathbb{B}\ true))\ l \arrow[r, no head] & (\lambda x\!:\!\tau_{id}.\, (x\ \mathbb{B})\ (x\ \mathbb{B}\ true))\ l' \\
(l\ \mathbb{B})\ (l\ \mathbb{B}\ true) \arrow[r, no head] & (l'\ \mathbb{B})\ (l'\ \mathbb{B}\ true) \\
(\lambda x\!:\!\mathbb{B}.\, x)\ (l\ \mathbb{B}\ true) \arrow[r, no head] & (\lambda x\!:\!\mathbb{B}.\, \texttt{out in}\ x)\ (l'\ \mathbb{B}\ true) \\
l\ \mathbb{B}\ true \arrow[r, no head] \arrow[dr, no head] & \texttt{out in}\ (l'\ \mathbb{B}\ true) \\
 & l'\ \mathbb{B}\ true \\
(\lambda x\!:\!\mathbb{B}.\, x)\ true \arrow[r, no head] & (\lambda x\!:\!\mathbb{B}.\, \texttt{out in}\ x)\ true \\
true \arrow[r, no head] \arrow[dr, no head] & \texttt{out in}\ true \\
 & true
\end{tikzcd}$$

Figure 4.1: A comparison of the reduction paths of $C[l]$ and $C[l']$, for the example context $C := (\lambda x : \tau_{id}.\, (x\ \mathbb{B})\ (x\ \mathbb{B}\ true))$ (-). Note how C is "suitable", since inserting a term e of type τ_{id} into C will result in a term $C[e]$ of type $\mathbb{B}$. In the diagram, the reduction path of $C[l]$ is displayed on the left-hand side, opposite that of $C[l']$. Every displayed term reduces—in one step—to the term immediately below it. The lateral connections represent that the relevant terms are related by the relation R defined in the text.

why l and l' *are* contextually equivalent. (If the reader is already convinced, they can skip this argument.)

We consider any suitable C such that $C[l'] \Downarrow v$ for some $v \in \{\texttt{true}, \texttt{false}\}$. We recognise that though C has only one hole (-) into which l' was filled, l' might get duplicated during reduction. For example, if C contains $((\lambda x : \forall\alpha.\alpha \to \alpha.\, x\ (\forall\alpha.\alpha \to \alpha)\ x)$ (-)) then $C[l']$ might reduce to a term containing $(l'\ (\forall\alpha.\alpha \to \alpha)\ l')$. If we define an ad-hoc relation R such that terms e, e' are R-related iff e' is equal to e except that some terms might be surrounded by an `out-in` pair, then clearly $R(C[l], C[l'])$. (See Fig. 4.1 for an illustration using an example context.) Looking at the reduction paths of $C[l]$ and $C[l']$, we see that R resembles a simulation. If two points e, e' in the reduction paths are R-related and $e \mapsto e_1$, then e' reduces to something that is R-related to e_1 in a number of steps that depends on how many superfluous `out-in` pairs have to be removed from e'. (In Fig. 4.1, this number of `out-in` pairs to be removed is always 0 or 1.) Through this reasoning, we expect the end v of the reduction path of $C[l']$ to be R-related to the end of the reduction path of $C[l]$. By definition of R, this means that $C[l] \Downarrow v$, since v cannot contain `out-in` pairs. A more rigorous proof of $l \approx^{ctx} l'$ will be given near the end of the chapter, using $\diamond$.

Finally, we see that it matters whether terms terminate or not. For example, the basic free theorem[1] saying that all terms of type $\tau_{id} := \forall\alpha.\alpha \to \alpha$ are equivalent to the identity function *id*, does not hold in our language. Take, for example, the basic divergent term Ω of type τ_{id}. We know it to be of the correct type and also that its reduction does not terminate. Then, the context $C :=$ (-) $\mathbb{B}$ `true` exposes the difference between Ω and *id*. $C[id] \Downarrow$ `true`, while $C[\Omega] = \Omega\ \mathbb{B}\ \texttt{true} \mapsto \Omega\ \mathbb{B}\ \texttt{true}$ gets stuck in an endless loop.

4.2 Definitions

In this section, we make precise the notions of *context* and *contextual equivalence*, which we introduced in Section 4.1. Technically, contextual equivalence $\approx^{ctx}$ will just be a symmetric

[1]Note that this theorem does not actually appear in Wadler's paper on free theorems.[18]

variant of *contextual approximation* $\leq^{ctx}$, the notion that we will work with most. We will also give a more precise definition of which contexts are *suitable* in which situations. We will do so through an inductively defined relation $\vdash^c$ for contexts similar to the type relation for terms.

After we have done that, we will define a binary step-indexed logical relation in a way very similar to how we defined $\diamond$ in Chapter 3. Again, $\diamond$ will simply be a symmetric variant of $\triangleright$, the actual logical relation we will define. The next sections will put these definitions to use.

4.2.1 Contexts and contextual equivalence

We define contexts, their wellformedness relation, and contextual equivalence in the style of Ahmed.[3]

Definition 79 (Context)**.** *The set Ctx of contexts is defined inductively using the following grammar:*

$$C ::= (\text{-}) \mid C\ e \mid e\ C \mid \lambda x\!:\!\tau.\, C \mid C\ \tau \mid \Lambda\alpha.C \mid \texttt{in}\ C \mid \texttt{out}\ C$$

The substitution $C[e]$ of a term e for the hole (-) *in a context C is similar to the analogue for evaluation contexts (Definition 25).*

Definition 80 (Context wellformedness relation)**.** *We define a new relation $\vdash^c$, resembling the typing relation on terms, but for contexts. The rules used for the inductive definition are displayed in Fig. 4.2.*

Intuitively, the meaning of a wellformedness statement $\Delta^c; \Gamma^c \vdash^c C : (\Delta; \Gamma \triangleright \tau) \rightsquigarrow \tau^c$ is the following: Given a term e that can be typed τ under environment Γ and type environment Δ. If we insert e into C, then the filled in context $C[e]$ can be typed τ^c under the environment Γ^c and type environment Δ^c. The attentive reader will note that it is not clear that the wellformedness relation $\vdash^c$ as presented in Definition 80 actually satisfies this property. The connection between the wellformedness $\vdash^c$ of C on the one hand, and the welltypedness $\vdash$ of the terms e and $C[e]$ on the other hand, is not immediate. We note, though, that we never claim, prove, or in fact need this property. Thinking of the relation this way, however, might help the reader to make sense of an otherwise rather abstract relation definition.

Definition 81 (Contextual approximation and equivalence)**.** *We define a binary term relation $\leq^{ctx}$),* contextual approximation. *More specifically, it relates a type environment Δ, an environment Γ, two terms e^l, e^r, and a type τ. The notation for $\langle \Delta, \Gamma, e^l, e^r, \tau \rangle \in \leq^{ctx}$ is $\Delta; \Gamma \vDash e^l \leq^{ctx} e^r : \tau$ and we say it holds iff the termination of e^l filled into any wellformed context implies the termination of e^r filled into the same context. Formally:*

$$\begin{aligned}(\Delta; \Gamma \vDash e^l \leq^{ctx} e^r : \tau) :\Leftrightarrow\ & \forall C \in Ctx, \tau^c \in Type : \\ & (\cdot; \cdot \vdash^c C : (\Delta; \Gamma \triangleright \tau) \rightsquigarrow \tau^c) \Rightarrow (C[e^l] \Downarrow \Rightarrow C[e^r] \Downarrow).\end{aligned}$$

We then define a symmetrisation $\approx^{ctx}$ over the same set as mutual contextual approximation. Formally:

$$(\Delta; \Gamma \vDash e^l \approx^{ctx} e^r : \tau) :\Leftrightarrow (\Delta; \Gamma \vDash e^l \leq^{ctx} e^r : \tau) \wedge (\Delta; \Gamma \vDash e^r \leq^{ctx} e^l : \tau).$$

When we say e^l, e^r are contextually equivalent or $e^l \approx^{ctx} e^r$ without further specification, we often mean $\cdot; \cdot \vDash e^l \approx^{ctx} e^r : \tau$ where e^l, e^r are of type τ.

The reader might wonder why we choose precisely this definition of contextual approximation and equivalence. It differs from the loose Definition 77 we gave in the examples in Section 4.1 in

$$\frac{\Delta \subseteq \Delta^c \quad \Gamma \subseteq \Gamma^c \quad \Delta^c \vdash \Gamma^c}{\Delta^c; \Gamma^c \vdash^c (\text{-}) : (\Delta; \Gamma \rhd \tau) \rightsquigarrow \tau} :^c_{\textbf{id}}$$

$$\frac{\Delta^c; \Gamma^c, (x : \tau_x) \vdash^c C : (\Delta; \Gamma, (x : \tau_x) \rhd \tau) \rightsquigarrow \tau^c}{\Delta^c; \Gamma^c \vdash^c \lambda x{:}\tau_x.\, C : (\Delta; \Gamma, (x : \tau_x) \rhd \tau) \rightsquigarrow \tau_x \to \tau^c} :^c_{\textbf{abs}}$$

$$\frac{\Delta^c; \Gamma^c \vdash^c C : (\Delta; \Gamma \rhd \tau) \rightsquigarrow \tau_2 \to \tau_3 \quad \Delta^c; \Gamma^c \vdash e_2 : \tau_2}{\Delta^c; \Gamma^c \vdash^c C\ e_2 : (\Delta; \Gamma \rhd \tau) \rightsquigarrow \tau_3} :^c_{\textbf{appright}}$$

$$\frac{\Delta^c; \Gamma^c \vdash^c C : (\Delta; \Gamma \rhd \tau) \rightsquigarrow \tau_2 \quad \Delta^c; \Gamma^c \vdash e_1 : \tau_2 \to \tau_3}{\Delta^c; \Gamma^c \vdash^c e_1\ C : (\Delta; \Gamma \rhd \tau) \rightsquigarrow \tau_3} :^c_{\textbf{appleft}}$$

$$\frac{\Delta^c; \Gamma^c \vdash^c C : (\Delta; \Gamma \rhd \tau) \rightsquigarrow \forall \alpha.\tau^c \quad \Delta^c \vdash \sigma}{\Delta^c; \Gamma^c \vdash^c C\ \sigma : (\Delta; \Gamma \rhd \tau) \rightsquigarrow \tau^c[\sigma/\alpha]} :^c_{\textbf{tapp}}$$

$$\frac{\Delta^c, \alpha; \Gamma^c \vdash^c C : (\Delta, \alpha; \Gamma \rhd \tau) \rightsquigarrow \tau^c}{\Delta^c; \Gamma^c \vdash^c \Lambda\alpha.C : (\Delta, \alpha; \Gamma \rhd \tau) \rightsquigarrow \forall \alpha.\tau^c} :^c_{\textbf{tabs}}$$

$$\frac{\Delta^c; \Gamma^c \vdash^c C : (\Delta; \Gamma \rhd \tau) \rightsquigarrow \tau^c[\mu\alpha.\tau^c/\alpha]}{\Delta^c; \Gamma^c \vdash^c \texttt{in}\ C : (\Delta; \Gamma \rhd \tau) \rightsquigarrow \mu\alpha.\tau^c} :^c_{\textbf{muin}}$$

$$\frac{\Delta^c; \Gamma^c \vdash^c C : (\Delta; \Gamma \rhd \tau) \rightsquigarrow \mu\alpha.\tau^c}{\Delta^c; \Gamma^c \vdash^c \texttt{out}\ C : (\Delta; \Gamma \rhd \tau) \rightsquigarrow \tau^c[\mu\alpha.\tau^c/\alpha]} :^c_{\textbf{muout}}$$

Figure 4.2: The context wellformedness relation $\vdash^c$, as defined by Ahmed. Note that three relations—four for pedants—with similar or just plain equal symbols are used. First, of course, the context wellformedness relation $\vdash^c$. Second, the type relation $\vdash$ on terms, in $:^c_{\textbf{appright}}$ and $:^c_{\textbf{appleft}}$. Third, the type wellformedness relation $\vdash$ on types, in $:^c_{\textbf{id}}$ and $:^c_{\textbf{tapp}}$. (Fourth, in $:^c_{\textbf{id}}$, actually the type wellformedness relation *lifted over environments* is used instead of the relation itself.)

two important ways. First, quantification over all types is performed, while we argued against it in the examples section. Second, the criterium by which filled-in contexts are judged is not to which value they evaluate, but if they terminate at all.

So far, in defining contextual equality, we have explored two dimensions along which to design our definition, with two possibilities each. The first dimension is that of universal type quantification (used by Definition 78 and the actual definition of $\approx^{ctx}$, Definition 81) versus using a ground type (used by Definition 77). Along the second dimension we have requiring the equality of the values to which $C[e^l], C[e^r]$ evaluate on the one hand (used by both definitions from the examples section), and requiring that their mere terminations are equivalent on the other hand: $C[e^l] \Downarrow \Leftrightarrow C[e^r] \Downarrow$ (used by the actual definition of $\approx^{ctx}$).

Recall our attempt Definition 78 at a definition of contextual equivalence quantifying universally over all types and not just $\mathbb{B}$. Requiring that the values to which $C[e^l]$ and $C[e^r]$ evaluate be *equal*, then, turned out to be too strict, i.e. it left out term pairs e^l, e^r which we *did* want to consider contextually equal. Thus, the combination of universal type quantification and value equality yields a wrong notion of contextual equality.

Since we chose to keep our language $\lambda\forall\mu$ lean, ground types like $\mathbb{B}$ were left out. Along the first dimension, then, defining contextual equivalence using a ground type is not an option and we must use universal type quantification. We also know from Definition 78 that combining this with the requirement of value equality does not work. In Appendix (B) we show that the only remaining combination, universal type quantification and mere termination equivalence, can be believed to yield a sufficient notion of contextual equivalence. This is exactly the combination used in the actual definition of $\approx^{ctx}$, Definition 81.

4.2.2 Binary step-indexed logical relation

We define a binary step-indexed logical relation $\rhd$. The definition will be similar to that of $\vDash$, see Definition 54. All concepts, like semantic type relations (Definition 42) and the mutually recursive term sets (Definition 47), will have to be adapted to our binary setting. We reuse the relevant symbols, like $\delta, \mathcal{V}, \mathcal{E}$, in the new definitions and expect the reader to understand that in this chapter, they refer to the binary versions of the concepts. A similar thing goes for lemmas and propositions, although some rephrasings are so minor that we leave them to the reader. We have attempted a reasonable tradeoff here, referring to the statement from the unary context if its translation is obvious, and properly proving it otherwise.

Definition 82 (Semantic type relation)**.** *We define $Rel(\text{-},\text{-})$ to be the following function taking two types τ^l, τ^r such that $\tau^l, tau^r \in CType$, with codomain $\mathcal{P}(\mathbb{N} \to \mathcal{P}(CVal^2))$.*

$$Rel(\tau^l, \tau^r) := \{\chi^{\text{-}} \in (\mathbb{N} \to \mathcal{P}(CVal^2)) \mid \forall k \in \mathbb{N} : \forall \langle v^l, v^r \rangle \in \chi^k : \\ (\cdot;\cdot \vdash v^l : \tau^l) \wedge (\cdot;\cdot \vdash v^r : \tau^r) \wedge (\forall j < k : \langle v^l, v^r \rangle \in \chi^j)\}.$$

We also say that $Rel := \bigcup_{\tau^l, \tau^r \in Type} Rel(\tau^l, \tau^r)$.

As Ahmed hinted at, the fact that we pose typing requirements for both types in the tuples has led us to define a notion of semantic type relations paramerised in not one but two types.[3]

Definition 83 (Type relation substitution)**.** *Given a type environment $\Delta \in TEnv$. We define the set $\mathcal{D}[\![\Delta]\!]$ of type relation substitutions to be the set of $\delta \in (\Delta \to (Type \times Type \times Rel))$ for which $\delta\ \alpha = \langle \tau^l, \tau^r, \chi \rangle$ implies $\tau^l, \tau^r \in CType$, and $\chi \in Rel(\tau^l, \tau^r)$.*

When $\Delta = \cdot$ then $\delta \in \mathcal{D}[\![\Delta]\!]$ is the empty function, which we will write $\cdot$ or $\emptyset$.

If $\delta\ \alpha = \langle \tau^l, \tau^r, \chi \rangle$, we define $\delta^l\ \alpha := \tau^l$, $\delta^r\ \alpha := \tau^r$, and $\delta^{\text{sem}}\ \alpha := \chi$.

We define $\delta^l\ \tau^l$ to be the simultaneous syntactic type substitution of $\delta^l\ \alpha$ for every type variable α in Δ that occurs freely in τ^l, and similarly if we replace l with r.

When $\alpha \notin \Delta$, we define $\delta[\alpha \mapsto \langle \tau^l, \tau^r, \chi\rangle]$ to be a type relation substitution in $\mathcal{D}[\![\Delta, \alpha]\!]$ that maps all $\alpha' \in \Delta$ to $\delta\ \alpha'$ and α to $\langle \tau^l, \tau^r, \chi\rangle$.

Regarding terms, we let $\delta^l\ e$ denote the result of simultaneous substitution of $\delta^l\ \alpha$ for every type variable α occurring freely in e, and similarly if we replace l with r.

Analogues for Lemma 45 and Lemma 46 hold, where in both lemmas, the equality assertion for δ is replaced by two assertions for δ^l and δ^r separately.

Definition 84 (Step-indexed, mutually recursive term sets)**.** *We define two functions, $\mathcal{V}$ and $\mathcal{E}$, at once using mutual recursion.*

We let DTD denote the set of tuples $\langle \Delta, \tau, \delta\rangle$ such that $\Delta \vdash \tau$ and $\delta \in \mathcal{D}[\![\Delta]\!]$. Both $\mathcal{V}$ and $\mathcal{E}$ take a natural number $k \in \mathbb{N}$ and a tuple $\langle \Delta, \tau, \delta\rangle \in DTD$. The notation of $\mathcal{V}$ applied to its arguments is $\mathcal{V}^k_\Delta[\![\tau]\!]\delta$ and similarly for $\mathcal{E}$. The codomains are such that $\mathcal{V}^k_\Delta[\![\tau]\!]\delta \subseteq CVal^2$ and $\mathcal{E}^k_\Delta[\![\tau]\!]\delta \subseteq CTerm^2$.

We define the functions by recursion on the index k.

$$\mathcal{V}^k_\Delta[\![\alpha]\!]\delta := (\delta^{\text{sem}}\ \alpha)^k, \tag{4.1}$$

$$\begin{aligned}\mathcal{V}^k_\Delta[\![\tau_1 \to \tau_2]\!]\delta := {} & T^\delta(\tau_1 \to \tau_2) \cap \{\langle \lambda x^l : \delta^l\ \tau_1.\, e^l, \lambda x^r : \delta^r\ \tau_1.\, e^r\rangle \in CVal^2 \mid \\ & \forall j < k : \forall \langle e'^l, e'^r\rangle \in \mathcal{E}^j_\Delta[\![\tau_1]\!]\delta : \langle e^l[e'^l/x^l], e^r[e'^r/x^r]\rangle \in \mathcal{E}^j_\Delta[\![\tau_2]\!]\delta\},\end{aligned} \tag{4.2}$$

$$\begin{aligned}\mathcal{V}^k_\Delta[\![\forall\alpha.\tau]\!]\delta := {} & T^\delta(\forall\alpha.\tau) \cap \{\langle \Lambda\alpha.e^l, \Lambda\alpha.e^r\rangle \in CVal^2 \mid \\ & \forall j < k : \forall \tau'^l, \tau'^r \in CType : \forall \chi \in Rel(\tau'^l, \tau'^r) : \\ & \langle e^l[\tau'^l/\alpha], e^r[\tau'^r/\alpha]\rangle \in \mathcal{E}^j_{\Delta,\alpha}[\![\tau]\!]\delta[\alpha \mapsto \langle \tau'^l, \tau'^r, \chi\rangle]\},\end{aligned} \tag{4.3}$$

$$\mathcal{V}^k_\Delta[\![\mu\alpha.\tau]\!]\delta := T^\delta(\mu\alpha.\tau) \cap \{\langle \texttt{in}\ e^l, \texttt{in}\ e^r\rangle \in CVal^2 \mid \forall j < k : \langle e^l, e^r\rangle \in \mathcal{E}^j_\Delta[\![\tau[\mu\alpha.\tau/\alpha]]\!]\delta\},$$

$$\begin{aligned}\mathcal{E}^k_\Delta[\![\tau]\!]\delta := {} & \{\langle e^l, e^r\rangle \in CTerm^2 \mid \forall j \leq k : \forall e'^l : e^l \Downarrow^j e'^l \Rightarrow \\ & \exists e'^r : e^r \Downarrow e'^r \wedge \langle e'^l, e'^r\rangle \in \mathcal{V}^{k-j}_\Delta[\![\tau]\!]\delta\},\end{aligned} \tag{4.4}$$

where $T^\delta(\tau) := \{\langle e^l, e^r\rangle \in Term^2 \mid (\cdot; \cdot \vdash e^l : \delta^l\ \tau) \wedge (\cdot; \cdot \vdash e^r : \delta^r\ \tau)\}$. To understand Eq. (4.1) we remember that $\delta^{\text{sem}}\ \alpha \in Rel((\delta^l\ \alpha), (\delta^r\ \alpha)) \subseteq (\mathbb{N} \to \mathcal{P}(CVal^2))$. By Notation 43, the superscript k thus represents application to k, and $(\delta^{\text{sem}}\ \alpha)^k \subseteq CVal^2$.

This concept generalised fairly obviously from its unary variant Definition 47. Note that many comments we gave on that definition, regarding the adaptations of Ahmed's definitions to our language, hold true here as well.

One non-obvious generalisation from unary to binary is the asymmetrical definition of $\mathcal{E}$. We might be tempted to require that $e^r \Downarrow^j e'^r$ instead of $e^l \Downarrow e'^r$, or to find other ways of keeping the definition more symmetrical. However, this would lead to too strong a constraint on terms, since then terms should terminate in an equal number of steps in order to be related. For example, the terms l, l' from Section 4.1 would not be related under such a definition. (Consider, e.g. $\langle l\ \tau\ v, l'\ \tau\ v\rangle$, which would not be an element of $\mathcal{E}^k_\cdot[\![\tau]\!]\cdot$ for sufficiently high index k.) As Ahmed notes, many "nice", symmetrical-seeming definitions lead to such strict constraints.[3]

Well-definedness of $\mathcal{V}, \mathcal{E}$ can be proved similarly to how we did in Chapter 3 for their unary counterparts. We will not repeat the proof, neither here nor in the appendix.

Definition 85 (Semantic term substitutions)**.** *We define the function $\mathcal{G}$, which takes a natural number k, a type environment Δ, an environment Γ such that $\Delta \vdash \Gamma$, and a type relation* *substitution $\delta \in \mathcal{D}[\![\Delta]\!]$. The notation is $\mathcal{G}^k_\Delta[\![\Gamma]\!]\delta$ and the codomain is such that $\mathcal{G}^k_\Delta[\![\Gamma]\!]\delta \in \mathcal{P}(\operatorname{dom}\Gamma \to \mathit{CTerm})$.*

$$\mathcal{G}^k_\Delta[\![\Gamma]\!]\delta := \{\gamma \in (\operatorname{dom}\Gamma \to \mathit{CTerm}^2) \mid \forall x \in \operatorname{dom}\Gamma : \gamma\ x \in \mathcal{E}^k_\Delta[\![\Gamma\ x]\!]\delta\}.$$

When $\Gamma = \cdot$ then $\gamma \in \mathcal{D}[\![\Gamma]\!]$ is the empty function, which we will write $\cdot$ or $\emptyset$.

Similarly to how we defined δ^l, we say that γ^l, γ^r are the two projections of γ: if $\gamma\ x = \langle e^l, e^r\rangle$, then $\gamma^l\ x = e^l$ and $\gamma^r\ x = e^r$. Also, $\gamma\ e$ is the simultaneous substitution of $\gamma\ x$ for every free x in e.

Notation 86 (Substitution distribution over tuples)**.** *We write $\delta\gamma$ for $\delta \circ \gamma$. We also write $\delta\gamma^l$ for $\delta^l \circ \gamma^l$, and similarly for $\delta\gamma^r$.*

Whenever a type relation substitution δ or a semantic term substitution γ is applied to a 2-tuple $\langle t^l, t^r\rangle$ of elements to which it could be applied (e.g. terms or types), the application is distributed over the tuple, i.e. $\delta\ \langle t^l, t^r\rangle := \langle \delta^l\ t^l, \delta^r\ t^r\rangle$ and similarly for γ.

Notation 51 through Notation 53 are to be translated to binary versions as well. This means that $\gamma\ \langle e^l, e^r\rangle := \langle \gamma^l\ e^l, \gamma^r\ e^r\rangle$ and that $\gamma[x \mapsto \langle e^l, e^r\rangle]$ is defined as $\langle \gamma^l[x \mapsto e^l], \gamma^r[x \mapsto e^r]\rangle$. Note how the latter is written as a tuple of functions but should be interpreted as one function returning tuples. It is precisely this kind of pedantry we will not concern ourselves with too much.

Definition 87 (Binary relation)**.** *We define a (binary) relation $\triangleright$ on terms. More specifically, it relates a type environment Δ, an environment Γ, two terms e^l, e^r, and a type τ. The notation for $\langle \Delta, \Gamma, e^l, e^r, \tau\rangle \in \triangleright$ is $\Delta; \Gamma \vDash e^l \triangleright e^r : \tau$.*

$$\begin{aligned}\Delta; \Gamma \vDash e^l \triangleright e^r : \tau :\Leftrightarrow\ & \Delta \vdash \Gamma \wedge \\ & \forall k \geq 0 : \forall \delta \in \mathcal{D}[\![\Delta]\!] : \forall \gamma \in \mathcal{G}^k_\Delta[\![\Gamma]\!]\delta : \delta\gamma\ \langle e^l, e^r\rangle \in \mathcal{E}^k_\Delta[\![\tau]\!]\delta.\end{aligned}$$

Remember that in our notation, $\delta\gamma\ \langle e^l, e^r\rangle = \langle \delta^l\ (\gamma^l\ e), \delta^r\ (\gamma^r\ e)\rangle$.

As we mentioned before, from the binary logical relation we derive another relation. This is merely the symmetrisation of the logical relation.

Definition 88 (Equivalence relation)**.** *We define a (binary) relation $\diamond$ on terms. More specifically, it relates a type environment Δ, an environment Γ, two term e^l, e^r, and a type τ. The notation for $\langle \Delta, \Gamma, e^l, e^r, \tau\rangle \in \diamond$ is $\Delta; \Gamma \vDash e^l \diamond e^r : \tau$, and it holds iff*

$$(\Delta; \Gamma \vDash e^l \triangleright e^r : \tau) \wedge (\Delta; \Gamma \vDash e^r \triangleright e^l : \tau).$$

We find that translations of the three lemmas at the end of Section 3.3 hold as well.

Lemma 89 (There is always a type relation substitution and semantic term substitution)**.** *Suppose $\Delta; \Gamma \vDash e^l \triangleright e^r : \tau$ and $k \geq 0$. Then there exist $\delta \in \mathcal{D}[\![\Delta]\!]$ and $\gamma \in \mathcal{G}^k_\Delta[\![\Gamma]\!]\delta$.*

Proof. Analogous to Lemma 55. □

Lemma 90 (Free (type) variables in the binary relation)**.** *Suppose that $\Delta; \Gamma \vDash e^l \triangleright e^r : \tau$. Then* $\operatorname{fv} e \subseteq \operatorname{dom}\Gamma$ *and* $\operatorname{ftv} e \subseteq \Delta$.

Proof. Analogous to Lemma 56. □

Lemma 91 (Downward closedness)**.** $\mathcal{V}, \mathcal{E}, \mathcal{G}$ *are* downward closed. *This has the following meaning.*

- *If $\langle v^l, v^r \rangle \in \mathcal{V}^k_\Delta[\![\tau]\!]\delta$ and $j < k$, then $\langle v^l, v^r \rangle \in \mathcal{V}^j_\Delta[\![\tau]\!]\delta$.*
- *If $\langle e^l, e^r \rangle \in \mathcal{E}^k_\Delta[\![\tau]\!]\delta$ and $j < k$, then $\langle e^l, e^r \rangle \in \mathcal{E}^j_\Delta[\![\tau]\!]\delta$.*
- *If $\gamma \in \mathcal{G}^k_\Delta[\![\Gamma]\!]\delta$ and $j < k$, then $\gamma \in \mathcal{G}^j_\Delta[\![\Gamma]\!]\delta$.*

Proof. The proof for $\mathcal{V}$ is similar to Lemma 57. The proof for $\mathcal{G}$ can again be done pointwise, based on the downward closedness of $\mathcal{E}$.

We now show how downward closedness of $\mathcal{E}$ can be derived from that of $\mathcal{V}$. Fix $\langle e^l, e^r \rangle \in \mathcal{E}^k_\Delta[\![\tau]\!]\delta$ and $j < k$. To prove $\langle e^l, e^r \rangle \in \mathcal{E}^j_\Delta[\![\tau]\!]\delta$, fix $i \leq j$ and e'^l such that

$$e^l \Downarrow^i e'^l. \tag{4.5}$$

We must prove the existence of some e'^r with $e^r \Downarrow e'^r$ such that $\langle e'^l, e'^r \rangle \in \mathcal{V}^{j-i}_\Delta[\![\tau]\!]\delta$. By the assumption $j < k$, we have $i \leq k$. Combined with Eq. (4.5), we get the existence of some e'^r with $e' \Downarrow e'^r$, such that $\langle e'^l, e'^r \rangle \in \mathcal{V}^{k-i}_\Delta[\![\tau]\!]\delta$. We finish the proof by invoking downward closedness of $\mathcal{V}$ to obtain $\langle e'^l, e'^r \rangle \in \mathcal{V}^{j-i}_\Delta[\![\tau]\!]\delta$. $\square$

We take the time to briefly discuss a few relational properties of the binary relation $\triangleright$ we just defined, namely reflexivity, anti-symmetry, and transitivity. (When regarding $\triangleright$ and $\diamond$ as truly binary relations, we keep Δ, Γ, τ fixed.) The first property, reflexivity, will turn out to hold for our relation. This is the subject of Section 4.3. Anti-symmetry will turn out not to hold, as will become clear in the discussion in Section 4.5.

The property of transitivity is a bit more tricky. Note that one of the improvements of Ahmed's binary relation—on which ours is based—relative to Appel-McAllester's PER model, was that its transitivity could be proved.[3, 4] Our relations $\triangleright$ and thus also $\diamond$ suffer from a problem similar to that of Appel-McAllester's model: though we have not found a counterexample against their transitivity, the obvious proof does not go through. We do not show this as the problem is very similar to that of Appel-McAllester. The problem is explained in detail in [3].

Thus, where Ahmed obtained a preorder $\leq$ along with its derived equivalence relation $\approx := \leq \cap \geq$, our corresponding $\triangleright$ and $\diamond$ cannot be proved to be a preorder and an equivalence relation, respectively, because of their lack of provable transitivity. Do note, however, that we neither claim nor need transitivity in order to reach the goal of this thesis. The fact that contextual equivalence is clearly transitive makes transitivity a necessary condition for *completeness.* However, we only concern ourselves with *soundness* of $\diamond$ w.r.t. $\approx^{ctx}$. Even if there is a triple e_1, e_2, e_3 such that $e_1 \diamond e_2 \diamond e_3$ but not $e_1 \diamond e_3$, then still this would imply $e_1 \approx^{ctx} e_2 \approx^{ctx} e_3$ and thus $e_1 \approx^{ctx} e_3$.

4.2.3 Statement of the theorem

Now that we have all definitions in place, we are ready to state—but not yet prove—the central theorem of this thesis:

Theorem 92 (Terms related in binary relation are contextually equivalent)**.** *Suppose $\Delta; \Gamma \models e^l \diamond e^r : \tau$. Then $\Delta; \Gamma \models e^l \approx^{ctx} e^r : \tau$.*

4.3 Fundamental property of the logical relation

In this section, we prove that if a term is well-typed, it is related to itself by $\triangleright$. From the definition, it is immediately clear this also implies it is related to itself by $\diamond$. Formally, we prove the following proposition:

Proposition 93 (Reflexive arrows for well-typed terms)**.** *Suppose that* $\Delta; \Gamma \vdash e : \tau$. *Then* $\Delta; \Gamma \vDash e \rhd e : \tau$.

We prove this by induction on $\Delta; \Gamma \vdash e : \tau$. In other words, we prove that $\rhd$ is closed under the type rules on which $\vdash$ is defined inductively.[2] The following subsections all contain a compatibility lemma that allows us to prove exactly one of the induction steps. We follow the structure of Section 3.4 closely and make similar necessary adaptations to Ahmed's work as before. However, we will not actually use the result Prop. 93 proved in this section. The only use of the compatibility lemmas is to be reused as parts of induction steps in the proof of context monotonicity in the next section. That proof requires the lemmas to be formulated slightly broader than what is necessary for Prop. 93. In order to preserve space and avoid repetition, this section features the "enhanced" compatibility lemmas straight away.

4.3.1 Variable

Lemma 94 (Compatibility lemma)**.** *Suppose* $\Delta \vdash \Gamma$ *and* $x \in \operatorname{dom} \Gamma$. *Then* $\Delta; \Gamma \vDash x \rhd x : \Gamma\, x$

Proof. Fix $k \geq 0$, $\delta \in \mathcal{D}[\![\Delta]\!]$, $\gamma \in \mathcal{G}^k_\Delta[\![\Gamma]\!]\delta$. We must prove that $\delta\ (\gamma\ x)$, which is equal to $\langle \delta^l\ (\gamma^l\ x), \delta^r\ (\gamma^r\ x)\rangle$, is an element of $\mathcal{E}^k_\Delta[\![\Gamma\ x]\!]\delta$. It is clear from the definition of γ that $\gamma\ x \in \mathit{CTerm}^2$ and therefore $\delta\ (\gamma\ x) = \gamma\ x \in \mathcal{E}^k_\Delta[\![\Gamma\ x]\!]\delta$. □

4.3.2 Term application

Lemma 95 (Interaction between $\mathcal{E}$ and $\mapsto$)**.** *Suppose* $\langle e^l, e^r\rangle \in \mathcal{E}^k_\Delta[\![\tau]\!]\delta$ *and* $e^l \mapsto^j e'^l$ *with* $j \leq k$. *Then* $\langle e'^l, e^r\rangle \in \mathcal{E}^{k-j}_\Delta[\![\tau]\!]\delta$.

Proof. Suppose e'^l evaluates to some e''^l with $\mathit{irred}\, e''^l$ in $i \leq k - j$ steps. Then $e^l \mapsto^{j+i} e''^l$, thus by assumption there is a term e'^r with $\mathit{irred}\, e'^r$ to which e^r evaluates, with $\langle e''^l, e'^r\rangle \in \mathcal{E}^{k-(j+i)}_\Delta[\![\tau]\!]\delta$. We realise that $k - (j + i) = k - j - i$. □

Lemma 96. *Suppose* $\langle e^l_1, e^r_1\rangle \in \mathcal{E}^k_\Delta[\![\tau_2 \to \tau_3]\!]\delta$ *and* $e^l_1\ e^l_2 \Downarrow^k e^l_3$. *Then there exist terms* b^l, b^r *and a natural number* $j < k$ *such that* $e^l_1 \mapsto^j \lambda x : \delta\ \tau_2.\, b^l$ *and* $e^r_1 \mapsto^* \lambda x' : \delta\ \tau_2.\, b^r$, *with* $\langle \lambda x{:}\delta\ \tau_2.\, b^l, \lambda x'{:}\delta\ \tau_2.\, b^r\rangle \in \mathcal{V}^{k-j}_\Delta[\![\tau_2 \to \tau_3]\!]\delta$.

Proof. The proof follows the same pattern as Lemma 60: we perform induction on k. In the base case $\mathit{irred}(e^l_1\ e^l_2)$ implies $\mathit{irred}\, e^l_1$ with e^l_1 not being a term abstraction. Together with the assumption this means $\langle e^l_1, \ldots\rangle \in \mathcal{V}^0_\Delta[\![\tau_2 \to \tau_3]\!]\delta$, which contradicts e^l_1 not being a term abstraction.

In the induction step, we assume the statement holds for k. We also assume $\langle e^l_1, e^r_1\rangle \in \mathcal{E}^{k+1}_\Delta[\![\tau_2 \to \tau_3]\!]\delta$ and $e^l_1\ e^l_2 \Downarrow^{k+1} e^l_3$. If e^l_1 is not already a term abstraction—which would end the proof—we can split this evaluation into, say

$$e^l_1\ e^l_2 \mapsto^1 e'^l_1\ e^l_2 \mapsto^k e^l_3.$$

Lemma 95 gives us that $\langle e'^l_1, e^r_1\rangle \in \mathcal{E}^k_\Delta[\![\tau_2 \to \tau_3]\!]\delta$. We instantiate the induction hypothesis with this and realise that e^l_1 evaluates in $j + 1 < k + 1$ steps to the term abstraction to which e'^l_1 evaluates in $j < k$ steps. The proof goal $e^r_1 \mapsto^* \lambda x' : \delta\ \tau_2.\, b^r$ is also met. The final proof goal, $\langle \lambda x{:}\delta\ \tau_2.\, b^l, \lambda x'{:}\delta\ \tau_2.\, b^r\rangle \in \mathcal{V}^{k-j}_\Delta[\![\tau_2 \to \tau_3]\!]\delta$, is met by induction and the fact that $k+1-(j+1) = k - j$. □

[2] A small remark here is that $\vdash$ and $\rhd$ have different arities and are thus defined over different sets. When we say that $\rhd$ is closed under the rules of $\vdash$, we actually mean that the reflexive part $\{\langle \Delta, \Gamma, e, e, \tau\rangle \mid \Delta; \Gamma \vDash e \rhd e : \tau\}$ of $\rhd$ is closed under these rules.

Lemma 97 (Compatibility lemma)**.** *Suppose* $\Delta; \Gamma \vDash e_1^l \triangleright e_1^r : \tau_2 \to \tau_3$ *and* $\Delta; \Gamma \vDash e_2^l \triangleright e_2^r : \tau_2$*. Then* $\Delta; \Gamma \vDash e_1^l\ e_2^l \triangleright e_1^r\ e_2^r : \tau_3$*.*

Proof. The proof follows the pattern of Prop. 74. Fix $k \geq 0$, $\delta \in \mathcal{D}[\![\Delta]\!]$, $\gamma \in \mathcal{G}_\Delta^k[\![\Gamma]\!]\delta$. We must prove that $\delta\gamma\ \langle (e_1^l\ e_2^l), (e_1^r\ e_2^r)\rangle \in \mathcal{E}_\Delta^k[\![\tau_3]\!]\delta$. Fix a term e_3^l and a number of steps $j \leq k$ such that $(\delta\gamma^l\ e_1^l)\ (\delta\gamma^l\ e_2^l) \Downarrow^j e_3^l$. We instantiate Lemma 96 with this, and find that $\delta\gamma^l\ e_1^l$ evaluates to $\lambda x : \delta^l\ \tau_2 .\, b^l$ in some $i < j$ steps, $\delta\gamma^r\ e_1^r$ to $\lambda x' : \delta^r\ \tau_2 .\, b^r$ in an unspecified number of steps, and $\langle \delta\gamma^l\ e_1^l, \delta\gamma^r\ e_1^r\rangle \in \mathcal{V}_\Delta^{k-i}[\![\tau_2 \to \tau_3]\!]\delta$. We must prove the existence of some e_3^r with $(\delta\gamma\ e_1^r)\ (\delta\gamma\ e_2^r) \Downarrow e_3^r$ such that $\langle e_3^l, e_3^r\rangle \in \mathcal{V}_\Delta^{k-j}[\![\tau_3]\!]\delta$.

We visualise the evaluation as follows:

$$\begin{array}{cccccccccc}
(\overbrace{(\delta\gamma^l\ e_1^l)} & (\delta\gamma^l\ e_2^l)) & \mapsto^i & (\overbrace{(\lambda x : \delta\ \tau_2 .\, b^l)} & \overbrace{(\delta\gamma^l\ e_2^l)}) & \mapsto^1 & \overbrace{b^l[\delta\gamma^l\ e_2^l/x]} & \mapsto^{j-i-1} & \overbrace{e_3^l} \\
(\underbrace{(\delta\gamma^r\ e_1^r)}_{(1)} & (\delta\gamma^r\ e_2^r)) & \mapsto^* & (\underbrace{(\lambda x' : \delta\ \tau_2 .\, b^r)}_{(2)} & \underbrace{(\delta\gamma^r\ e_2^r)}_{(3)}) & \mapsto^1 & \underbrace{b^r[\delta\gamma^r\ e_2^r/x]}_{(4)} & \mapsto^* & \underbrace{e_3^r}_{(5)}\ ,
\end{array}$$

where $\begin{array}{c}\overbrace{a}\\ \underbrace{b}_{(c)}\end{array}$ signifies a claim about $\langle a, b\rangle$ made in item (c) of the following list, and every claim builds upon the previous ones.

(1) This tuple is element of $\mathcal{E}_\Delta^k[\![\tau_2 \to \tau_3]\!]\delta$, by instantiation of the first assumption.

(2) This tuple is element of $\mathcal{V}_\Delta^{k-i}[\![\tau_2 \to \tau_3]\!]\delta$, the result of Lemma 96.

(3) This tuple is element of $\mathcal{E}_\Delta^k[\![\tau_2]\!]\delta$, by instantiation of the second assumption. We make use of downward closedness (Lemma 91) to replace k by $k - i - 1$, in order for the next step to go through.

(4) This tuple is element of $\mathcal{E}_\Delta^{k-i-1}[\![\tau_3]\!]\delta$, by instantiation of (2) with (3).

(5) This tuple is element of $\mathcal{E}_\Delta^{k-j}[\![\tau_3]\!]\delta$. We assumed that $(\delta\gamma^l\ e_1^l)\ (\delta\gamma^l\ e_2^l) \Downarrow^j e_3^l$, which, given our previous information and determinism, means that $b^l[\delta\gamma^l\ e_2^l/x] \mapsto^{j-i-1} e_3^l$. Therefore, the existence of e_3^r with $b^r[\delta\gamma^r\ e_2^r/x] \Downarrow e_3^r$ and $\langle e_3^l, e_3^r\rangle \in \mathcal{V}_\Delta^{k-i-1-(j-i-1)}[\![\tau_3]\!]\delta$ is guaranteed, by (4). We finish the proof by realising that $k - i - 1 - (j - i - 1) = k - j$.

□

4.3.3 Term abstraction

Lemma 98 (Compatibility lemma)**.** *Suppose* $\Delta; \Gamma, (x : \tau_x) \vDash e^l \triangleright e^r : \tau$*. We must prove that* $\Delta; \Gamma \vDash \lambda x {:} \tau_x .\, e^l \triangleright \lambda x {:} \tau_x .\, e^r : \tau_x \to \tau$*.*

Proof. Fix $k \geq 0$, $\delta \in \mathcal{D}[\![\Delta]\!]$, $\gamma \in \mathcal{G}_\Delta^k[\![\Gamma]\!]\delta$. Since both terms are already irreducible, We must prove that $\delta\gamma\ \langle \lambda x {:} \tau_x .\, e^l, \lambda x {:} \tau_x .\, e^r\rangle \in \mathcal{V}_\Delta^k[\![\tau_x \to \tau]\!]\delta$. Fix $j < k$ and $\langle e'^l, e'^r\rangle \in \mathcal{E}_\Delta^j[\![\tau_x]\!]\delta$. Now we must prove that $\langle (\delta\gamma^l\ e^l)[e'^l/x], (\delta\gamma^r\ e^r)[e'^r/x]\rangle \in \mathcal{E}_\Delta^j[\![\tau]\!]\delta$.

We realise that $\delta^l\ e'^l = e'^l$ and instantiate Lemma 62 for δ^l, γ^l, e^l, and find that $(\delta\gamma^l\ e^l)[e'^l/x] = \delta^l\ (\gamma'^l\ e^l)$, where we define $\gamma'^l := \gamma^l[x \mapsto e'^l]$. We do a similar thing for r, resulting in $\gamma' := \gamma[x \mapsto \langle e'^l, e'^r\rangle]$. We thus need only prove that $\langle \delta^l\ (\gamma'^l\ e^l), \delta^r\ (\gamma'^r\ e^r)\rangle \in \mathcal{E}_\Delta^j[\![\tau]\!]\delta$. This proof goal can be met by instantiating the assumption with j, δ, and $\gamma' \in \mathcal{G}_\Delta^k[\![\Gamma]\!]\delta \subseteq \mathcal{G}_\Delta^j[\![\Gamma]\!]\delta$. □

4.3.4 Type application

We expand upon Lemma 66:

Lemma 99 (Type substitution in term sets)**.** *Given te following: $k \geq 0$; Δ; $\alpha \notin \Delta$; $\delta \in \mathcal{D}[\![\Delta]\!]$; τ, τ' such that $\Delta, \alpha \vdash \tau$ and $\Delta \vdash \tau'$. Let $\Delta' := (\Delta, \alpha)$ and $\delta' := \delta[\alpha \mapsto \langle \delta^l\ \tau', \delta^r\ \tau', \mathcal{V}^{-}_{\Delta}[\![\tau']\!]\delta\rangle]$. Then $\mathcal{V}^k_{\Delta'}[\![\tau]\!]\delta' = \mathcal{V}^k_{\Delta}[\![\tau[\tau'/\alpha]]\!]\delta$. The same equality then clearly also holds when we replace $\mathcal{V}$ with $\mathcal{E}$.*

Proof. Analogous to Lemma 66. □

Lemma 100 (Values in the recursive value set are well-typed)**.** *Given $k \geq 0$ and $\langle \Delta, \tau, \delta\rangle \in DTD$. Then for all $\langle v^l, v^r\rangle \in \mathcal{V}^k_{\Delta}[\![\tau]\!]\delta$, we have $\cdot; \cdot \vdash v^l : \delta^l\ \tau$ and $\cdot; \cdot \vdash v^r : \delta^r\ \tau$.*

Proof. Just like Lemma 64, we can prove be exhaustive case distinction, where the proof goal follows immediately from the relevant definition of $\mathcal{V}$. □

Lemma 91 and Lemma 100 together give us the following property of $\mathcal{V}$.

Corollary 101 (Recursive value sets are semantic type relations)**.** *Given $\langle \Delta, \tau, \delta\rangle \in DTD$. Then $\mathcal{V}^{-}_{\Delta}[\![\tau]\!]\delta \in Rel(\delta^l\ \tau, \delta^r\ \tau)$.*

Lemma 102. *Suppose $\langle e^l, e^r\rangle \in \mathcal{E}^k_{\Delta}[\![\forall\alpha.\tau]\!]\delta$ and $e^l\ \tau' \Downarrow^k e'^l$. Then there exist terms b^l, b^r and a natural number $j < k$ such that $e^l \mapsto^j \Lambda\alpha.b^l$ and $e^r \mapsto^* \Lambda\alpha.b^r$, with $\langle \Lambda\alpha.b^l, \Lambda\alpha.b^r\rangle \in \mathcal{V}^{k-j}_{\Delta}[\![\forall\alpha.\tau]\!]\delta$.*

Proof. Analogous to Lemma 67, adapting the proof to the binary relation the way we changed Lemma 60 to Lemma 96. □

Lemma 103 (Compatibility lemma)**.** *Suppose $\Delta; \Gamma \vDash e^l \triangleright e^r : \forall\alpha.\tau$ and $\Delta \vdash \tau'$. Then $\Delta; \Gamma \vDash e^l\ \tau' \triangleright e^r\ \tau' : \tau[\tau'/\alpha]$.*

Proof. Fix $k \geq 0$, $\delta \in \mathcal{D}[\![\Delta]\!]$, $\gamma \in \mathcal{G}^k_{\Delta}[\![\Gamma]\!]\delta$. Also fix $j \leq k$ and e'^l such that $\delta\gamma^l\ (e^l\ \tau') \Downarrow^j e'^l$. (Note that $\delta\gamma^l\ (e^l\ \tau') = (\delta\gamma^l\ e^l)\ (\delta^l\ \tau')$.) We instantiate Lemma 102 with this and find that $\delta\gamma^l\ e^l$ evaluates to $\Lambda\alpha.b^l$ in some $i < j$ steps, $\delta\gamma^r\ e^r$ evaluates to $\Lambda\alpha.b^r$ in an unspecified number of steps, and $\langle \delta\gamma^l\ e^l, \delta\gamma^r\ e^r\rangle \in \mathcal{V}^{k-i}_{\Delta}[\![\forall\alpha.\tau]\!]\delta$. We must prove the existence of some e'^r with $(\delta\gamma^r\ e^r)\ (\delta^r\ \tau') \Downarrow e'^r$ such that $\langle e'^l, e'^r\rangle \in \mathcal{V}^{k-j}_{\Delta}[\![\tau[\tau'/\alpha]]\!]\delta$. We visualise the evaluation as follows:

$$\begin{array}{lclclcl}
((\overbrace{\delta\gamma^l\ e^l})\ (\delta^l\ \tau')) & \mapsto^i & ((\overbrace{\Lambda\alpha.b^l})\ \overbrace{(\delta^l\ \tau')}) & \mapsto^1 & \overbrace{b^l[\delta^l\ \tau'/\alpha]} & \mapsto^{j-i-1} & \overbrace{e'^l} \\
((\underbrace{\delta\gamma^r\ e^r}_{(1)})\ (\delta^r\ \tau')) & \mapsto^* & ((\underbrace{\Lambda\alpha.b^r}_{(2)})\ \underbrace{(\delta^r\ \tau')}_{(3)}) & \mapsto^1 & \underbrace{b^r[\delta^r\ \tau'/\alpha]}_{(4)} & \mapsto^* & \underbrace{e'^r}_{(5)},
\end{array}$$

where $\underbrace{\overbrace{a}^{}\atop b}_{(c)}$ signifies a claim about $\langle a, b\rangle$ made in item (c) of the following list, and every claim builds upon the previous ones.

(1) This tuple is element of $\mathcal{E}^k_{\Delta}[\![\forall\alpha.\tau]\!]\delta$, by instantiation of the assumption.

(2) This tuple is element of $\mathcal{V}^{k-i}_{\Delta}[\![\forall\alpha.\tau]\!]\delta$, by the result of Lemma 102.

(3) Both types are closed, i.e. $\delta^l\ \tau', \delta^r\ \tau' \in CType$.

(4) This tuple is element of $\mathcal{E}^{k-i-1}_{\Delta}[\![\tau]\!]\delta[\alpha \mapsto \langle \delta^l\ \tau', \delta^r\ \tau', \chi\rangle]$, where $\chi := \mathcal{V}^{-}_{\Delta}[\![\tau']\!]\delta$. This holds for every index strictly smaller than $k - i$, including $k - i - 1$, and all $\chi \in \mathit{Rel}(\delta^l\ \tau', \delta^r\ \tau')$. That our choice of χ is an element of $\mathit{Rel}(\delta^l\ \tau', \delta^r\ \tau')$, is a direct consequence of Corollary (101).

(5) This tuple is an element of $\mathcal{V}^{k-j}_{\Delta}[\![\tau]\!]\delta[\alpha \mapsto \langle \delta^l\ \tau', \delta^r\ \tau', \mathcal{V}^{-}_{\Delta}[\![\tau']\!]\delta\rangle]$. We made the assumption that $(\delta\gamma^l\ e^l)\ (\delta^l\ \tau') \Downarrow^j e'^l$, which, given our previous reasoning and determinism of the operational semantics, means that $b^l[\delta^l\ \tau'/\alpha] \mapsto^{j-i-1} e'^l$. Therefore, the existence of a term e'^r with $b^r[\delta^r\ \tau'/\alpha] \Downarrow e'^r$ and $\langle e'^l, e'^r\rangle \in \mathcal{V}^{k-j}_{\Delta}[\![\tau]\!]\delta[\alpha \mapsto \langle \delta^l\ \tau', \delta^r\ \tau', \chi\rangle]$ is guaranteed, by (4).

We finish the proof by using Lemma 99. ☐

4.3.5 Type abstraction

Lemma 104 (Redundant type variables make no difference)**.** *Given the following: $k \geq 0$; $\langle \Delta, \tau, \delta\rangle \in DTD$; $\alpha \notin \Delta$. Let $\Delta' := (\Delta, \alpha)$ and $\delta' := \delta[\alpha \mapsto \langle \sigma^l, \sigma^r, \chi\rangle]$ for some $\sigma^l, \sigma^r \in CType, \chi \in Rel(\sigma^l, \sigma^r)$. (Clearly, $\delta' \in \mathcal{D}[\![\Delta']\!]$.) Then $\mathcal{V}^k_{\Delta}[\![\tau]\!]\delta = \mathcal{V}^k_{\Delta'}[\![\tau]\!]\delta'$.*

Under the same conditions, $\mathcal{E}^k_{\Delta}[\![\tau]\!]\delta = \mathcal{E}^k_{\Delta'}[\![\tau]\!]\delta'$ holds. If, for any given Γ, we replace the condition $\langle \Delta, \tau, \delta\rangle \in DTD$ with $(\delta \in \Delta \wedge \Delta \vdash \Gamma)$, then $\mathcal{G}^k_{\Delta}[\![\Gamma]\!]\delta = \mathcal{G}^k_{\Delta'}[\![\Gamma]\!]\delta'$ holds.

Proof. We first realise, by expanding the definition of $\mathcal{E}$, that if the lemma holds for $\mathcal{V}$, it holds for $\mathcal{E}$. We also see that the lemma for $\mathcal{G}$ holds as long as it holds for $\mathcal{E}$: we can apply the equality for $\mathcal{E}$ pointwise to all $x \in \operatorname{dom}\Gamma$. The lemma for $\mathcal{V}$ is proved analogously to Lemma 69. ☐

Lemma 105 (Compatibility lemma)**.** *Suppose $(\Delta, \alpha); \Gamma \vDash e^l \triangleright e^r : \tau$. Then $\Delta; \Gamma \vDash \Lambda\alpha.e^l \triangleright \Lambda\alpha.e^r : \forall\alpha.\tau$.*

Proof. Fix $k \geq 0$, $\delta \in \mathcal{D}[\![\Delta]\!]$, $\gamma \in \mathcal{G}^k_{\Delta}[\![\Gamma]\!]\delta$. Since both terms are already irreducible, we must prove that $\delta\gamma\ \langle \Lambda\alpha.e^l, \Lambda\alpha.e^r\rangle \in \mathcal{V}^k_{\Delta}[\![\forall\alpha.\tau]\!]\delta$. Fix $j < k$, two closed types $\tau^l, \tau^r \in CType$, and a $\chi \in Rel(\tau^l, \tau^r)$. We write $\Delta' := (\Delta, \alpha)$ and $\delta' := \delta[\alpha \mapsto \langle \tau^l, \tau^r, \chi\rangle]$. We see that $(\delta\gamma^l\ e^l)[\tau^l/\alpha] = \delta'^l\ (\gamma^l\ e^l)$ and similarly for R, by Lemma 45.

It now suffices to prove that $\langle \delta'^l\ (\gamma^l\ e^l), \delta'^r\ (\gamma^r\ e^r)\rangle \in \mathcal{E}^j_{\Delta'}[\![\tau]\!]\delta'$. Since $\delta' \in \mathcal{D}[\![\Delta']\!]$ and $\gamma \in \mathcal{G}^k_{\Delta'}[\![\Gamma]\!]\delta'$ by Lemma 104, we can instantiate the induction hypothesis to achieve this proof goal. ☐

4.3.6 Type unfolding

Lemma 106. *Suppose $\langle e^l, e^r\rangle \in \mathcal{E}^k_{\Delta}[\![\mu\alpha.\tau]\!]\delta$ and $\texttt{out}\ e^l \Downarrow^k e'^l$. Then there exist terms b^l, b^r and a natural number $j < k$ such that $e^l \mapsto^j \texttt{in}\ b^l$ and $e^r \mapsto^* \texttt{in}\ b^r$, with $\langle \texttt{in}\ b^l, \texttt{in}\ b^r\rangle \in \mathcal{V}^{k-j}_{\Delta}[\![\mu\alpha.\tau]\!]\delta$.*

Proof. Analogous to Lemma 71, adapting the proof to the binary relation the way we changed Lemma 60 to Lemma 96. ☐

Lemma 107 (Compatibility lemma)**.** *Suppose $\Delta; \Gamma \vDash e^l \triangleright e^r : \mu\alpha.\tau$. Then $\Delta; \Gamma \vDash \texttt{out}\ e^l \triangleright \texttt{out}\ e^r : \tau[\mu\alpha.\tau/\alpha]$.*

Proof. Fix $k \geq 0$, $\delta \in \mathcal{D}[\![\Delta]\!]$, $\gamma \in \mathcal{G}^k_{\Delta}[\![\Gamma]\!]\delta$. Also fix $j \leq k$ and e'^l with $\texttt{out}\ e^l \Downarrow^j e'^l$. We instantiate Lemma 106 with this and find that $\delta\gamma^l\ e^l$ evaluates to some $\texttt{in}\ b^l$ in some $i < j$ steps, $\delta\gamma^r\ e^r$ evaluates to some $\texttt{in}\ b^r$ in some unspecified number of steps, and $\langle \delta\gamma^l\ e^l, \delta\gamma^r\ e^r\rangle \in$

$\mathcal{V}^{k-i}_{\Delta}[\![\tau[\mu\alpha.\tau/\alpha]]\!]\delta$. We must prove the existence of some e'^r with $\texttt{out}\ e^r \Downarrow^j e'^r$ such that $\langle e'^l, e'^r\rangle \in \mathcal{V}^{k-j}_{\Delta}[\![\tau[\mu\alpha.\tau/\alpha]]\!]\delta$. We visualise the evaluation as follows:

$$\begin{array}{lllllll}
\texttt{out}\ \overbrace{\delta\gamma^l\ e^l} & \mapsto^i & \texttt{out}\ \overbrace{(\texttt{in}\ \delta\gamma^l\ e^l)} & \mapsto^1 & \overbrace{\delta\gamma^l\ e^l} & \mapsto^{j-i-1} & \overbrace{e'^l} \\
\texttt{out}\ \underbrace{\delta\gamma^r\ e^r}_{(1)} & \mapsto^* & \texttt{out}\ \underbrace{(\texttt{in}\ \delta\gamma^r\ e^r)}_{(2)} & \mapsto^1 & \underbrace{\delta\gamma^r\ e^r}_{(3)} & \mapsto^* & \underbrace{e'^r}_{(4)},
\end{array}$$

where $\overbrace{\underbrace{\begin{matrix}a\\ b\end{matrix}}_{(c)}}$ signifies a claim about $\langle a, b\rangle$ made in item (c) of the following list, and every claim builds upon the previous ones.

(1) This tuple is element of $\mathcal{E}^{k}_{\Delta}[\![\mu\alpha.\tau]\!]\delta$, by instantiation of the assumption.

(2) This tuple is element of $\mathcal{V}^{k-i}_{\Delta}[\![\mu\alpha.\tau]\!]\delta$, by the result of Lemma 106.

(3) This tuple is element of $\mathcal{E}^{k-i-1}_{\Delta}[\![\tau[\mu\alpha.\tau/\alpha]]\!]\delta$, by definition of $\mathcal{V}$. Note that it holds for all indices strictly smaller than $k-i$, including $k-i-1$.

(4) This tuple is element of $\mathcal{E}^{k-j}_{\Delta}[\![\tau[\mu\alpha.\tau/\alpha]]\!]\delta$. We assumed that $\texttt{out}\ \delta\gamma^l\ e^l \Downarrow^j e'^l$, which, given our previous reasoning and determinism of the operational semantics, means that $\delta\gamma^l\ e^l \mapsto^{j-i-1} e'^l$. Therefore, the existence of a term e'^r with $\delta\gamma^r\ e^r \Downarrow e'^r$ and $\langle e'^l, e'^r\rangle \in \mathcal{V}^{k-j}_{\Delta}[\![\tau[\mu\alpha.\tau/\alpha]]\!]\delta$ is guaranteed, by (4).

□

4.3.7 Type folding

Lemma 108 (Compatibility lemma). *Suppose* $\Delta;\Gamma \vDash e^l \rhd e^r : \tau[\mu\alpha.\tau/\alpha]$. *Then* $\Delta;\Gamma \vDash \texttt{in}\ e^l \rhd \texttt{in}\ e^r : \mu\alpha.\tau$.

Proof. Fix $k \geq 0$, $\delta \in \mathcal{D}[\![\Delta]\!]$, $\gamma \in \mathcal{G}^k_{\Delta}[\![\Gamma]\!]\delta$. Since both terms are already irreducible, we must prove that $\delta\gamma\ \langle \texttt{in}\ e^l, \texttt{in}\ e^r\rangle \in \mathcal{V}^k_{\Delta}[\![\mu\alpha.\tau]\!]\delta$. Therefore, we fix $j < k$. The proof goal is now reduced to $\delta\gamma\ \langle e^l, e^r\rangle \in \mathcal{E}^j_{\Delta}[\![\tau[\mu\alpha.\tau/\alpha]]\!]\delta$, which is true by instantiation of the assumption with δ, γ, and by downward closedness of $\mathcal{E}$ (Lemma 91). □

4.3.8 Conclusion

We can now prove the proposition that was presented at the beginning of this section.

Proposition 93 (Reflexive arrows for well-typed terms). *Suppose that* $\Delta;\Gamma \vdash e : \tau$. *Then* $\Delta;\Gamma \vDash e \rhd e : \tau$.

Proof. We prove by induction on $\Delta;\Gamma \vdash e : \tau$. More formally, we prove that $\{\langle \Delta, \Gamma, e, \tau\rangle \mid \Delta;\Gamma \vDash e \rhd e : \tau\}$ is closed under the same rules that were used to inductively define $\vdash$, i.e. $\{\langle \Delta, \Gamma, e, \tau\rangle \mid \Delta;\Gamma \vdash e : \tau\}$. Lemmas 94, 97, 98, 103, 105, 107, 108 all prove one of the induction steps. With the exception of Lemma 94, they all prove a more general statement than we need right now. To achieve what we want, we can instantiate those statements with two equal terms e instead of a separate e^l, e^r. □

4.4 Context monotonicity and contextual equivalence

In this section, we perform the second step of the proof. We define what it means for a context to be *monotonic*. We say that C is monotonic iff for every two terms $e^l \triangleright e^r$, we have that $C[e^l] \triangleright C[e^r]$. First we will prove that all wellformed contexts are monotonic. Then, we will use this fact to give a direct proof that $e^l \triangleright e^r$ implies $e^l \preceq^{ctx} e^r$. The main theorem then follows through double application of that implication, once for $e^l \triangleright e^r$ and once for $e^r \triangleright e^l$.

This constitutes a deviation from Ahmed's argumentation structure. They do not have a concept of context monotonicity and instead find another way to lift their binary relation from the term to the context level. The ends of the argumentations are similar again, though. Ahmed's use of their binary relation for contexts is used in a way similar to how we use context monotonicity in the concluding proof.

Definition 109 (Context monotonicity)**.** *We also introduce a relation $\nearrow$ capturing the notion of* context monotonicity*. The notation is $\Delta^c; \Gamma^c \vDash C \nearrow : (\Delta; \Gamma \triangleright \tau) \rightsquigarrow \tau^c$, which holds iff "filling in larger terms into C leads to larger terms". Formally,*

$$\begin{aligned}&\Delta^c; \Gamma^c \vDash C \nearrow : (\Delta; \Gamma \triangleright \tau) \rightsquigarrow \tau^c :\Leftrightarrow \\ &\quad \forall e^l, e^r : (\Delta; \Gamma \vDash e^l \triangleright e^r : \tau) \Rightarrow (\Delta^c; \Gamma^c \vDash C[e^l] \triangleright C[e^r] : \tau^c).\end{aligned}$$

Proposition 110 (Wellformed contexts are monotonic)**.** *Suppose $\Delta^c; \Gamma^c \vdash^c C : (\Delta; \Gamma \triangleright \tau) \rightsquigarrow \tau^c$. Then $\Delta^c; \Gamma^c \vDash C \nearrow : (\Delta; \Gamma \triangleright \tau) \rightsquigarrow \tau^c$.*

Proof. By induction. More specifically, we prove that $\nearrow$ is closed under the rules by which $\vdash^c$ is inductively defined.

- $:^c_{\mathbf{id}}$. Suppose $\Delta \subseteq \Delta^c$, $\Gamma \subseteq \Gamma^c$, and $\Delta^c \vdash \Gamma^c$. We must prove $\Delta^c; \Gamma^c \vDash (\text{-}) \nearrow : (\Delta; \Gamma \triangleright \tau) \rightsquigarrow \tau$. Fix e^l, e^r and suppose that $\Delta; \Gamma \vDash e^l \triangleright e^r : \tau$. We must prove that $\Delta^c; \Gamma^c \vDash e^l \triangleright e^r : \tau$. We therefore fix $k \geq 0, \delta^c \in \mathcal{D}[\![\Delta^c]\!], \gamma^c \in \mathcal{G}^k_{\Delta^c}[\![\Gamma^c]\!]\delta^c$ and write $\delta\gamma^c$ for $\delta^c \circ \gamma^c$.

  Now we make use of the set inclusion assumptions: we define δ to be the restriction of δ^c to Δ and γ to be the restriction of γ^c to $\operatorname{dom}\Gamma$. We immediately see that $\delta \in \mathcal{D}[\![\Delta]\!]$.

  From the definitions of γ^c and γ we can clearly see that $\gamma \in \mathcal{G}^k_{\Delta^c}[\![\Gamma]\!]\delta^c$. We now use Lemma 104 "in reverse" to prove that $\gamma \in \mathcal{G}^k_{\Delta}[\![\Gamma]\!]\delta$. Our assumption $\Delta; \Gamma \vDash e^l \triangleright e^r : \tau$ tells us that $\Delta \vdash \Gamma$. By repeated application of Lemma 104—once for every variable $\alpha \in \Delta^c \setminus \Delta$—we conclude that $\mathcal{G}^k_{\Delta^c}[\![\Gamma]\!]\delta^c = \mathcal{G}^k_{\Delta}[\![\Gamma]\!]\delta$.

  The assumption $\Delta; \Gamma \vDash e^l \triangleright e^r : \tau$ then tells us that $\delta\gamma\ \langle e^l, e^r \rangle \in \mathcal{E}^k_{\Delta}[\![\tau]\!]\delta$. We show this last statement is equivalent to our proof goal—$\delta\gamma^c\ \langle e^l, e^r \rangle \in \mathcal{E}^k_{\Delta^c}[\![\tau]\!]\delta^c$—by showing that the statements' left-hand sides are equal as well as their right-hand sides. First, $\delta\gamma\ \langle e^l, e^r \rangle = \delta\gamma^c\ \langle e^l, e^r \rangle$ because neither of the terms contain free (type) variables outside of Δ or $\operatorname{dom}\Gamma$ (Lemma 90). Second, $\mathcal{E}^k_{\Delta}[\![\tau]\!]\delta = \mathcal{E}^k_{\Delta^c}[\![\tau]\!]\delta^c$ by another chain of invocations of Lemma 104.

The proofs for the other rules differ from the previous one but are very similar to each other. Roughly, we get to assume that C is monotonic and need to prove some C' is monotonic as well. We assume $e^l \triangleright e^r$ and instantiate the assumption with it to get $C[e^l] \triangleright C[e^r]$. We then apply one of the lemmas used to prove reflexivity of $\triangleright$, i.e. Lemma 97 through Lemma 107. (Remember than we purposefully defined these lemmas too broadly back then so that we could recycle them here.) We respectively show one case that follows this pattern perfectly and one which deviates from it slightly:

- $:^{c}_{\mathbf{appright}}$. Suppose $\Delta^c;\Gamma^c \vDash C \nearrow : (\Delta;\Gamma \triangleright \tau) \leadsto \tau_2 \to \tau_3$ and $\Delta^c;\Gamma^c \vdash e_2 : \tau_2$ We must prove $\Delta^c;\Gamma^c \vDash C\ e_2 \nearrow : (\Delta;\Gamma \triangleright \tau) \leadsto \tau_3$.

  Assume $\Delta;\Gamma \vDash e^l \triangleright e^r : \tau$. We apply the first premise to this and find $\Delta^c;\Gamma^c \vDash C[e^l] \triangleright C[e^r] : \tau_2 \to \tau_3$. We invoke Prop. 93 on the second premise: $\Delta^c;\Gamma^c \vDash e_2 \triangleright e_2 : \tau_2$. We can now apply Lemma 97 and arrive at $\Delta^c;\Gamma^c \vDash C[e^l]\ e_2 \triangleright C[e^r]\ e_2 : \tau_3$.

- $:^{c}_{\mathbf{tabs}}$. Suppose $\Delta^c,\alpha;\Gamma^c \vDash C \nearrow : (\Delta^c,\alpha;\Gamma \triangleright \tau) \leadsto \tau^c$. We must prove $\Delta^c;\Gamma^c \vDash \Lambda\alpha.C \nearrow : (\Delta,\alpha;\Gamma \triangleright \tau) \leadsto \forall\alpha.\tau^c$.

  Assume $\Delta,\alpha;\Gamma \vDash e^l \triangleright e^r : \tau$. We apply the first premise to this and find $\Delta^c,\alpha;\Gamma^c \vDash C[e^l] \triangleright C[e^r] : \tau^c$. We can now apply Lemma 105 and arrive at $\Delta^c;\Gamma^c \vDash \Lambda\alpha.C[e^l] \triangleright \Lambda\alpha.C[e^r] : \forall\alpha.\tau^c$.

□

Theorem 111 (Small terms are contextual approximation of large terms)**.** *Suppose* $\Delta;\Gamma \vDash e^l \triangleright e^r : \tau$*. Then* $\Delta;\Gamma \vDash e^l \leq^{ctx} e^r : \tau$*. Another phrasing is* $\triangleright \subseteq \leq^{ctx}$*.*

Proof. Suppose that $\cdot;\cdot \vdash^c C : (\Delta;\Gamma \triangleright \tau) \leadsto \tau^c$ and that $C[e^l] \Downarrow$, say $C[e^l] \Downarrow^k e'^l$. We must prove that $C[e^r] \Downarrow$.

We apply Prop. 110 to the assumption on C and find $\cdot;\cdot \vDash C \nearrow : (\Delta;\Gamma \triangleright \tau) \leadsto \tau^c$. We instantiate this with the premise in the lemma and get $\cdot;\cdot \vDash C[e^l] \triangleright C[e^r] : \tau^c$. We realise that $\delta := \emptyset \in \mathcal{D}[\![\cdot]\!]$ and $\gamma := \emptyset \in \mathcal{G}^k_{\cdot}[\![\cdot]\!]\delta$. This means that $\delta\gamma\ \langle C[e^l], C[e^r]\rangle = \langle C[e^l], C[e^r]\rangle \in \mathcal{E}^k_{\cdot}[\![\tau^c]\!]\delta$. We can instantiate this with the statement $C[e^l] \Downarrow^k e'^l$. Therefore, there exists some value e'^r with $C[e^r] \Downarrow e'^r$, which finishes the proof. □

Looking at the definitions of $\diamond$ and $\approx^{ctx}$, this clearly implies that the main theorem holds:

Theorem 92 (Terms related in binary relation are contextually equivalent)**.** *Suppose* $\Delta;\Gamma \vDash e^l \diamond e^r : \tau$*. Then* $\Delta;\Gamma \vDash e^l \approx^{ctx} e^r : \tau$*.*

4.5 Discussion

4.5.1 Examples revisited

In this section, we revisit the introductory examples from Section 4.1. Unlike when we first introduced them, this time we will be able to formally discuss whether the given pairs are contextually equivalent. The first and third example pairs will turn out not to satisfy contextual equivalence.[3] We will prove this by providing a type and a wellformed context, in such a way that whether or not the context terminates differs depending on which of the two terms is filled in. This constitutes a counterexample. The terms l, l' from the second pair *are* contextually equivalent. We will prove this not through a direct proof via the definition of course, but by proving $l \diamond l'$. This is an example use of Theorem 92.

First example: type equality is not enough

Recall the example:

$$\begin{aligned} \mathit{first} &:= \Lambda\alpha.\lambda x\!:\!\alpha.\,\lambda y\!:\!\alpha.\,x, \\ \mathit{second} &:= \Lambda\alpha.\lambda x\!:\!\alpha.\,\lambda y\!:\!\alpha.\,y, \end{aligned}$$

[3] Recall that when left unspecified, the (type) environments are assumed to be empty and the type is assumed to be the that of the terms. In symbols, $e^l \approx^{ctx} e^r$ is shorthand for $\cdot;\cdot \vDash e^l \approx^{ctx} e^r : \tau$ where e^l, e^r are of type τ.

$$\dfrac{\dfrac{\dfrac{\dfrac{\cdot \subseteq \cdot \qquad \cdot \subseteq \cdot \qquad \cdot \vdash \cdot}{\cdot;\cdot \vdash^c (\text{-}) : (\cdot;\cdot \triangleright \tau) \rightsquigarrow \tau} \qquad \cdot \vdash \tau_{id}}{\cdot;\cdot \vdash^c (\text{-})\ \tau_{id} : (\cdot;\cdot \triangleright \tau) \rightsquigarrow \tau_{id} \to \tau_{id} \to \tau_{id}} \qquad \cdot;\cdot \vdash id : \tau_{id}}{\cdot;\cdot \vdash^c (\text{-})\ \tau_{id}\ id : (\cdot;\cdot \triangleright \tau) \rightsquigarrow \tau_{id} \to \tau_{id}} \qquad \cdot;\cdot \vdash \Omega : \tau_{id}}{\cdot;\cdot \vdash^c (\text{-})\ \tau_{id}\ id\ \Omega : (\cdot;\cdot \triangleright \tau) \rightsquigarrow \tau_{id}}$$

Figure 4.3: Derivation of the fact that the context $C = (\text{-})\ \tau_{id}\ \Omega_{\tau_{id}}$ is wellformed. Informally, the meaning is that when we insert a term e that can be typed τ with an empty environment and type environment, then the whole term $e\ \tau_{id}\ \Omega_{\tau_{id}}$ can be typed τ_{id}, also with an empty environment and type environment.

and note that both terms are of type $\tau := \forall\alpha.\alpha \to \alpha \to \alpha$. As we have said before, it is immediately intuitively clear that these terms are not contextually equal. $first$ reduces to the first (term) argument given to it, and $second$ to the second. It suffices to construct a context such that (-) is applied to two arguments, only one of which terminates:

$$\begin{aligned} id &:= \Lambda\alpha.\lambda x\!:\!\alpha.\, x, \\ \tau_{id} &:= \forall\alpha.\alpha \to \alpha, \\ C &:= (\text{-})\ \tau_{id}\ id\ \Omega, \end{aligned}$$

where Ω represents the basic divergent term of type τ_{id}. We see that C is wellformed: $\cdot;\cdot \vdash^c C : (\cdot;\cdot \triangleright \tau) \rightsquigarrow \tau_{id}$. (The derivation is shown in Fig. 4.3.) We also see that $C[first] \Downarrow id$ and $C[second] \mapsto^* \Omega \not\Downarrow$. Therefore, by definition, $\cdot;\cdot \vDash first \approx^{ctx} second : \tau_{id}$ does not hold.

Second example

We recall the example:

$$\begin{aligned} l &:= id = \Lambda\alpha.\lambda x\!:\!\alpha.\, x, \\ l' &:= \Lambda\alpha.\lambda x\!:\!\alpha.\, \texttt{out in}\ x, \end{aligned}$$

and note that both terms are of type $\tau_{id} := \forall\alpha.\alpha \to \alpha$. In the Section 4.1, we argued that these terms are contextually equivalent. We now prove that that is indeed the case, through use of our binary relation $\diamond$. We prove both directions separately.

Note how we are about to prove that $l \triangleright l'$ and $l' \triangleright l$, with $l \neq l'$. This will be a counterexample against the anti-symmetry of $\triangleright$.

Claim 112. *$\cdot;\cdot \vDash l \triangleright l' : \tau_{id}$ holds.*

Proof. Fix $k \geq 0, \delta \in \mathcal{D}[\![\cdot]\!], \gamma \in \mathcal{G}^k_{\cdot}[\![\cdot]\!]\delta$. We see that $\delta = \cdot$ and $\gamma = \cdot$. We must thus prove that $\langle l, l' \rangle \in \mathcal{E}^k_{\cdot}[\![\tau_{id}]\!]\cdot$, which, since both terms are values, amounts to proving $\langle l, l' \rangle \in \mathcal{V}^k_{\cdot}[\![\tau_{id}]\!]\cdot$.

Fix an index $k_1 < k$, types $\tau, \tau' \in CType$, and a semantic type relation $\chi \in Rel(\tau, \tau')$. We must prove that $\langle \lambda x\!:\!\tau.\, x, \lambda x\!:\!\tau'.\, \texttt{out in}\ x \rangle \in \mathcal{E}^{k_1}_{\alpha}[\![\alpha \to \alpha]\!][\alpha \mapsto \langle \tau, \tau', \chi \rangle]$. Again, both terms are values so this amounts to proving that the tuple is in $\mathcal{V}^{k_1}_{\alpha}[\![\alpha \to \alpha]\!][\alpha \mapsto \langle \tau, \tau', \chi \rangle]$.

Fix an index $k_2 < k_1$ and terms e, e' such that

$$\langle e, e' \rangle \in \mathcal{E}^{k_2}_{\alpha}[\![\alpha]\!][\alpha \mapsto \langle \tau, \tau', \chi \rangle]. \tag{4.6}$$

We must prove that $\langle x[e/x], (\texttt{out in } x)[e'/x]\rangle = \langle e, \texttt{out in } e'\rangle \in \mathcal{E}_\alpha^{k_2}[\![\alpha]\!][\alpha \mapsto \langle \tau, \tau', \chi\rangle]$.

Fix an index $k_3 \leq k_2$ and a term e_f such that $e \Downarrow^{k_3} e_f$. We must prove that there exists a term e'_f such that $\texttt{out in } e' \Downarrow e'_f$ and $\langle e_f, e'_f\rangle \in \mathcal{V}_\alpha^{k_2-k_3}[\![\alpha]\!][\alpha \mapsto \langle \tau, \tau', \chi\rangle]$.

From Eq. (4.6) combined with $e \Downarrow^{k_3} e_f$ and $k_3 \leq k_2$, we get that there exists a term e'_f such that $e' \Downarrow e'_f$ and $\langle e_f, e'_f\rangle \in \mathcal{V}_\alpha^{k_2-k_3}[\![\alpha]\!][\alpha \mapsto \langle \tau, \tau', \chi\rangle]$. We see that $\texttt{out in } e' \mapsto e' \Downarrow e'_f$, which finishes the proof. □

Claim 113. *$\cdot;\cdot \vDash l' \triangleright l : \tau_{id}$ holds.*

Proof. The proof is identical up to the switching of sides of the arguments, until the point where the proof obligation is $\langle \texttt{out in } e', e\rangle \in \mathcal{E}_\alpha^{k_2}[\![\alpha]\!][\alpha \mapsto \langle \tau', \tau, \chi\rangle]$. From there, the proof diverges because of the asymmetry in the definition of $\mathcal{E}$.

Fix an index $k_3 \leq k_2$ and a term e'_f such that $\texttt{out in } e' \Downarrow^{k_3} e'_f$. We must prove that there exists a term e_f such that $e \Downarrow e_f$ and $\langle e'_f, e_f\rangle \in \mathcal{V}_\alpha^{k_2-k_3}[\![\alpha]\!][\alpha \mapsto \langle \tau, \tau', \chi\rangle]$.

We note that $\texttt{out in } e' \mapsto e'$ and $\texttt{out in } e' \Downarrow^{k_3} e'_f$ together imply $e' \Downarrow^{k_3-1} e'_f$, by Lemma 29. We combine this with the symmetric analogue of Eq. (4.6) to get the existence of a term e_f such that $e \Downarrow e_f$ and $\langle e'_f, e_f\rangle \in \mathcal{V}_\alpha^{k_2-k_3+1}[\![\alpha]\!][\alpha \mapsto \langle \tau', \tau, \chi\rangle]$. The downward closedness of $\mathcal{V}$ (Lemma 91) then finishes the proof. □

Third example: termination matters

In the third example, we gave a minimalist demonstration of the fact that our definition of contextual equivalence depends on whether the given terms themselves terminate or not. The counterexample we gave involved booleans. As discussed in Section 4.2.1, we use a different definition of contextual equivalence (Definition 81) than we did in Section 4.1, from where this example originates. However, a necessary condition for this to be a didactically good example is that it remains valid under our new, final definition.

We therefore seek a new context C and type τ^c such that $\cdot;\cdot \vdash^c C : (\cdot;\cdot \triangleright \tau_{id}) \rightsquigarrow \tau^c$. We then refute $C[id] \Downarrow \Leftrightarrow C[\Omega] \Downarrow$ by showing that $C[id] \Downarrow$ and $C[\Omega] \not\Downarrow$. We realise that we need not construct our context such that its "outer type" τ^c is some specific one (like $\mathbb{B}$). Therefore, the most trivial context $C :=$ (-)—note that$\cdot;\cdot \vdash^c C : (\cdot;\cdot \triangleright \tau_{id}) \rightsquigarrow \tau_{id}$—will do. The termination equivalenc is then clearly refuted: $C[id] = id$ is irreducible and $C[\Omega] = \Omega$ does not terminate.

We observe that this "proof" generalises: for any two terms e^l, e^r where one terminates and the other does not, a counterexample against $e^l \approx^{ctx} e^r$ can be made by choosing $C :=$ (-).

4.5.2 Free theorems

We investigate the relation between $\lambda\forall\mu$ and *free theorems*. By this, we mean the kind of theorem as discussed by Wadler.[18] Generally, a free theorem is a (non-trivial) statement that can be derived from a type, such that all terms in the language of that type satisfy the theorem. For the purposes of this thesis, free theorems will thus be claims about the behaviour of terms and their relatedness under $\diamond$. In this section, we take a close look at two such free theorems to see if they hold in our language. We do note that these theorems, arguably the simplest possible ones, do not appear in Wadler's publication. They are, however, discussed in e.g. [12, 16].

Identity type We state the following free theorem (which, as we will see, does not hold in our language), where τ_{id} is the *identity type* $\forall\alpha.(\alpha \to \alpha)$ and id is the term $\Lambda\alpha.\lambda x\!:\!\alpha.\, x$ of type τ_{id}:

Claim 114 (False free theorem)**.** *Suppose e is of type τ_{id}. Then $\cdot;\cdot \vDash e \diamond id : \tau_{id}$.*

Informally, the statement claims that all terms of the identity type are "almost the same as" the identity function. As was shown in Section 4.1, this claim clearly does not hold:

Counterexample. As a counterexample, we let e be the basic divergent term Ω of type τ_{id}.

First, we recall the refutation of contextual equivalence that was used in Section 4.1, and transform it into a counterexample that can be used under our current definition of $\approx^{ctx}$. If we define a context $C := (\text{-})$, then $C[\Omega] \not\Downarrow$ while $C[id] \Downarrow$. Therefore, $\Omega \not\approx^{ctx} id$, and by contraposition of Theorem 92, $\neg(\Omega \diamond id)$.

An alternative approach would be to derive a contradiction, using the definition of $\diamond$, from the assumpition $\Omega \diamond id$. By definition, then, $\langle id, \Omega\rangle \in \mathcal{E}^k_{\cdot}[\![\tau_{id}]\!]\cdot$ for every $k \geq 0$. However, $id \Downarrow^0 id$ then implies that Ω terminates in some finite number of steps as well. We know this is not the case. $\square$

We see that the possibility of non-termination provides for counterexamples against claims of relatedness under $\diamond$, both via $\approx^{ctx}$ and directly. We wonder, though, if we can formulate a similar free theorem on the identity type without using $\diamond$, such that it *does* hold in our language. We try again:

Claim 115 (Free theorem)**.** *Suppose that e is of type τ_{id}. Then for all closed types τ and terms t, t' of type τ such that $t \Downarrow t'$, we have that if $e\ \tau\ t \Downarrow$, then $e\ \tau\ t \Downarrow t'$.*

We see that we have now built in an assumption of termination into our claim, while staying as true as possible to the intuitive meaning "all terms of the identity type are almost the same as the identity function". Note that assuming the termination of e or $e\ \tau$ would not have been enough, since the counterexamples $e := \Lambda\alpha.\Omega_{\alpha\to\alpha}$ and $e := \Lambda\alpha.\lambda x : \alpha.\,\Omega_\alpha$, respectively, would still prevent $e\ \tau\ t$ from terminating, let alone evaluating to t'. Realising that termination might pop up at almost any point, we thus specifically assume the termination of the entire term with all its arguments: $e\ \tau\ t \Downarrow$.

It turns out that we can indeed prove this version of the free theorem. We start from the fact that e is $\diamond$-related to itself. Most of the proof consists of unwinding the definitions from Section 4.2.2. As we will see towards the end, the interesting part lies in the choice of one semantic type relation.

Proof of the free theorem. We know from Prop. 93 that $\langle e, e\rangle \in \mathcal{E}^k_{\cdot}[\![\forall\alpha.(\alpha \to \alpha)]\!]\cdot$, where we take k such that $e\ \tau\ t \Downarrow^k$. From this termination assumption, we get that $e \Downarrow^{k_1}$ evaluates to some $\Lambda\alpha.e_1$ in some $k_1 < k$ steps. Thus, $\langle \Lambda\alpha.e_1, \Lambda\alpha.e_1\rangle \in \mathcal{V}^{k-k_1}_{\cdot}[\![\tau_{id}]\!]\cdot$.

If we now fix any $\chi \in Rel(\tau, \tau)$ and say that $\delta := [\alpha \mapsto \langle \tau, \tau, \chi\rangle]$, then $\langle e_1[\tau/\alpha], e_1[\tau/\alpha]\rangle \in \mathcal{E}^{k-k_1-1}_{\alpha}[\![\alpha \to \alpha]\!]\delta$. (According to the definition of $\mathcal{V}$, any index strictly smaller than $k - k_1$ would have worked.) So far, we know the following about the reduction path of $e\ \tau\ t$:

$$e\ \tau\ t \mapsto^{k_1} (\Lambda\alpha.e_1)\ \tau\ t \mapsto^1 e_1[\tau/\alpha]\ t \Downarrow^{k-k_1-1}.$$

Invoking Lemma 96 with this and $\langle e_1[\tau/\alpha], e_1[\tau/\alpha]\rangle \in \mathcal{E}^{k-k_1-1}_{\alpha}[\![\alpha \to \alpha]\!]\delta$, we get that $e_1[\tau/\alpha]$ evaluates to some $\lambda x : \tau.\, e_2$ in some $k_2 < k - k_1 - 1$ steps, with $\langle \lambda x : \tau.\, e_2, \lambda x : \tau.\, e_2\rangle \in \mathcal{V}^{k-k_1-1-k_2}_{\alpha}[\![\alpha]\!]\delta$. Our most up to date knowledge of the reduction path is now:

$$e\ \tau\ t \mapsto^{k_1} (\Lambda\alpha.e_1)\ \tau\ t \mapsto^1 e_1[\tau/\alpha]\ t \mapsto^{k_2} (\lambda x{:}\tau.\, e_2)\ t \mapsto^1 e_2[t/x] \Downarrow^{k-k_1-k_2-2}.$$

At this point, there is one obvious step we might want to perform: to apply the fact that $\langle t, t\rangle \in \mathcal{E}^{k-k_1-k_2-2}_{\alpha}[\![\alpha]\!]\delta$, in order to obtain $\langle e_2[t/x], e_2[t/x]\rangle \in \mathcal{E}^{k-k_1-k_2-2}_{\alpha}[\![\alpha]\!]\delta$. However, neither the truth of the former nor the usefulness of the latter is immediately clear. Let us compute more meaningful characterisations of the two statements. First, $\langle t, t\rangle \in \mathcal{E}_\alpha^{k-k_1-k_2-2}[\![\alpha]\!]\delta$ is equivalent to

$$(\exists j \leq k - k_1 - k_2 - 2 : t \Downarrow^j) \Rightarrow \langle t', t'\rangle \in \mathcal{V}_\alpha^{k-k_1-k_2-2}[\![\alpha]\!]\delta = \chi^{k-k_1-k_2-2}. \tag{4.7}$$

Regarding the second statement, we already know that $e_2[t/x]$ evaluates in $k - k_1 - k_2 - 2$ steps, say $e_2[t/x] \Downarrow^{k-k_1-k_2-2} e'$. Thus $\langle e_2[t/x], e_2[t/x]\rangle \in \mathcal{E}_\alpha^{k-k_1-k_2-2}[\![\alpha]\!]\delta$ is simply equivalent to

$$\langle e', e'\rangle \in \chi^{k-k_1-k_2-2}. \tag{4.8}$$

We see that our proof goal $e\ \tau\ t \Downarrow t'$ can now be stated as $e' = t'$. In order to achieve it, it thus suffices to instantiate χ such that Eq. (4.7) holds and that Eq. (4.8) implies $e' = t'$. This is where the true creativity lies in $\diamond$-based proofs of free theorems. We must choose χ large enough such that Eq. (4.7) holds, but not too large. It must still be so small that membership of χ "contains enough information" to imply Eq. (4.8). We note that this balancing act resembles the search for a good hypothesis in induction proofs.

We propose to choose χ such that for all $n \geq 0$, $\chi^n = \{\langle t', t'\rangle\}$. Regarding Eq. (4.7), it does not matter that we do not know whether the antecedent holds, since we constructed χ such that the conclusion does. We also see that Eq. (4.8) holds.

We conclude that $e' = t'$ and therefore $e\ \tau\ t \Downarrow t'$. $\square$

We introduce a new relation on terms in order to reformulate this free theorem in a more elegant way.

Definition 116 (Coterminality)**.** *We say that terms e, e' are coterminal, notation $(e, e') \Downarrow$, iff the following holds: if both e and e' terminate, then they evaluate to the same term. In symbols: $(e, e') \Downarrow :\Leftrightarrow (\, (e \Downarrow) \land (e' \Downarrow) \Rightarrow \exists e'' : e \Downarrow e'' \land e' \Downarrow e'')$.*

We can now rephrase our proved claim as a proposition, using the notion of coterminality. Note that it is indeed equivalent, by the fact that $(id\ \tau\ t, t) \Downarrow$ and some propositionally valid equivalences. Note also how similar it has become to Claim (114). We quantify over some additional arguments in order to prevent non-termination, thus replacing e by $e\ \tau\ t$ and id by $id\ \tau\ t$, and replace the claim of $\diamond$-relatedness by coterminality:

Proposition 117 (Free theorem about identity type)**.** *Suppose that e is of type τ_{id}, that τ is a closed type, and that t is a term of type τ. Then $(e\ \tau\ t, id\ \tau\ t) \Downarrow$.*

Absurd type In the previous paragraph, we investigated the truth of a free theorem about the identity type τ_{id}. Now, we will do the same for a free theorem about the absurd type $\tau_\perp := \forall\alpha.\alpha$.

In a language where all terms terminate, we might expect the following to hold:

Claim 118 (False free theorem)**.** $\neg\exists e : (\cdot; \cdot \vdash e : \tau_\perp)$,

i.e. there are no terms of the absurd type. In our language, this of course does not hold: by Prop. 35, we have $\cdot; \cdot \vdash \Omega_{\tau_\perp} : \tau_\perp$. Still, so far, all terms of type $\tau_\perp$ seem to be non-terminating in some sense. First, there is $\Omega_{\tau_\perp}$, which does not terminate. Second, there is $\Lambda\alpha.\Omega_\alpha$, which, admittedly, does terminate, but becomes non-terminating the moment it is applied to a type.

We will draw inspiration from the previous paragraph. There, we started out with a seemingly true free theorem, "all e of type τ_{id} have $e \diamond id$", which turned out to be false. We discovered that an altered version of the theorem *was* true, where application to sufficiently many universally quantified arguments—in that case τ and t—made non-termination a non-issue. We ended up with "all e, τ, t subject to some constraints have $(e\ \tau\ t, id\ \tau\ t) \Downarrow$".

We wish to do something similar for this free theorem. We know of no general technique for this, but we do note that in the previous paragraph, we applied $e : \forall\alpha.(\alpha \to \alpha)$ as much as possible, namely to a type τ and a term t. It seems reasonable to perform maximal application here as well. Since $\tau_\bot = \forall \alpha . \alpha$ we apply e to one argument, a closed type τ:

Proposition 119 (Free theorem about absurd type)**.** *Suppose that e is of type $\tau_\bot$ and τ is a closed type. Then $e\ \tau \not\Downarrow$.*

Proof. Suppose that $e\ \tau \Downarrow e'$ for some e'. Say that this evaluation takes k steps. We will work towards a contradiction.

We start by applying Prop. 93 to obtain $\cdot;\cdot \vDash e \diamond e : \tau_\bot$. Thus, $\langle e, e\rangle \in \mathcal{E}^k_\cdot[\![\tau_\bot]\!]\cdot$. Lemma 96, applied to $e\ \tau \Downarrow^k$, then tells us that e evaluates to some $\Lambda\alpha.e_1$ in some $k_1 < k$ steps, with $\langle \Lambda\alpha.e_1, \Lambda\alpha.e_1\rangle \in \mathcal{V}^{k-k_1}_\cdot[\![\tau_\bot]\!]\cdot$.

Fix arbitrary $\chi \in \mathit{Rel}(\tau,\tau)$. Then, by definition of $\mathcal{V}$, $\langle e_1[\tau/\alpha], e_1[\tau/\alpha]\rangle \in \mathcal{E}^{k-k_1-1}_\alpha[\![\alpha]\!]\delta$, where $\delta = [\alpha \mapsto \langle \tau, \tau, \chi\rangle]$.

The termination assumption $e\ \tau \Downarrow^k e'$, combined with $e \mapsto^{k_1} \Lambda\alpha.e_1$, yields that $e\ \tau \mapsto^{k_1} (\Lambda\alpha.e_1)\ \tau \mapsto^1 e_1[\tau/\alpha] \Downarrow^{k-k_1-1} e'$. Therefore, by definition of $\mathcal{E}$, $\langle e', e'\rangle \in \mathcal{V}^0_\alpha[\![\alpha]\!]\delta = \chi^0$.

Note that our reasoning so far is true for any $\chi \in \mathit{Rel}(\tau,\tau)$. We therefore instantiate with the valid choice of the constantly empty $\chi^n := \emptyset$. This leads to the contradiction $\langle e', e'\rangle \in \chi^0 = \emptyset$. $\square$

Further generalisation So far, we have seen two free theorems. In both cases, the first, natural-seeming proposed claim quickly turned out to be incorrect. We had to make adjustments in order to rule out problems caused by non-termination. However, we did this in an ad-hoc manner. Ideally, we would like a uniform way of adjusting such natural free theorems, of which we conjecture they hold in terminating languages, to versions that hold in our language. We now present one such strategy for obtaining uniform adjustments and briefly discuss why it fails.

We consider imposing the technique used in Prop. 117 on Prop. 119. We rephrase the latter as a statement about $\diamond$. Then, though, we replace every statement $e \diamond e'$ by the coterminality statement $(e, e') \Downarrow$, just like we did in Prop. 117. The process of quantification over the necessary arguments remains the same.

We begin by rephrasing the (false) free theorem $\neg\exists e : (\cdot;\cdot \vdash e : \tau_\bot)$ to $\forall e : (\cdot;\cdot \vdash e : \tau_\bot) \Rightarrow \neg(\cdot;\cdot \vDash e \diamond e : \tau_\bot)$, which is equivalent by Prop. 93. We then transform it into $\forall e : (\cdot;\cdot \vdash e : \tau_\bot) \Rightarrow \neg(e, e) \Downarrow$. The final step is to quantify over the additional argument for reasons of non-termination: $\forall e, \tau : (\cdot;\cdot \vdash e : \tau_\bot) \wedge (\tau \in \mathit{CType}) \Rightarrow \neg(e\ \tau, e\ \tau) \Downarrow$.

However, we see that this adjusted free theorem still does not hold. By definition of coterminality, we are not asserting that $e\ \tau$ terminates, but rather that if it does, then $e\ \tau$ evaluates to the same term as $e\ \tau$ does. The conclusion of the implication is therefore trivially false and thus the universal quantification states that there are no terms of type $\tau_\bot$. We have thus produced an incorrect theorem, since $\Omega_{\tau_\bot}$ is of type $\tau_\bot$.

Chapter 5

Related work

The main contribution of this thesis was, essentially, to redo some of the work done by Ahmed[3] on call-by-value System F with contravariant iso-recursive types in a slightly different language. The most important difference was that we used call-by-name semantics instead of call-by-value. As we have seen, the soundness theorem, $\diamond \subseteq \approx^{ctx}$, holds.

However, while Ahmed went on also to prove completeness $\approx^{ctx} \subseteq \diamond$, we did not. A first area of exploration from here might be to follow the path of Ahmed further and see if completeness also holds in our semantics. We observe a slight indication that it does not hold, or that it would at least be difficult to prove. In Appel-McAllester, the obvious proof of transitivity does not go through.[4] Ahmed solves this by adding asymmetrical type requirements.[3] In this thesis, we did not copy this asymmetrical definition. Also, when trying to prove transitivity, our work displays a problem similar to that of Appel-McAllester. This leads us to believe that transitivity of $\diamond$ does not hold, although a counterexample is not known. Contextual equivalence certainly *does* have transitivity and therefore it is a necessary condition for completeness of $\diamond$ w.r.t. $\approx^{ctx}$. It might be worthwhile to try to change the definitions to resemble the asymmetrical ones of Ahmed.

Second, we consider the free theorems. In the discussion in Section 4.5, we concluded that our semantics does not satisfy some basic free theorems. Both the statement that the identity function is the only one of its type $\forall\alpha.\alpha \to \alpha$ and the statement that there are no functions of the absurd type $\forall\alpha.\alpha$—both to be considered up to contextual equivalence—turn out not to be true. In both cases, all problems seem to stem from the possibility of non-termination. We managed to translate both free theorems to a version that is true in our semantics, but have found no generally applicable way of doing so. The main problem was that we were required to add assumptions of termination of multiple terms. The choice of terms did not clearly correlate with the type on which the free theorem was based.

It might be worth researching in what way Wadler-style free theorems—or perhaps even Wadler's theorems themselves—are to be altered in order to be satisfied by our relation semantics.[18] Ahmed and Skorstengaard do not perform a structural attempt of proving (altered versions of) free theorems either.[3, 16] They do manage to achieve results by Sumii and Pierce, which might be interesting to look at in our call-by-name language as well.[17]

An alternative route we might explore in free theorems is to integrate our results with the work of Pitts.[12] In our language, the existence of the fixpoint operator and divergent terms is mostly to be seen as a consequence of contravariant type recursion. Pitts, however, defines a language without type recursion but with a fixpoint operator built in. Their theory, involving biorthogonality, is purposefully built to take non-termination into account. The same two free theorems we examined are considered by Pitts as well. However, the adaptations to non-termination are part of the theory, and not the product of manual alterations as is the case in this thesis.

Finally, we consider one more tangent to our work. Plotkin and Abadi developed a logic to reason about System F, which is essentially our language $\lambda\forall\mu$ without the recursive types.[14] Dreyer et al., then, combined this with Appel-McAllester's line of work on step-indexed logical relations.[7] They created a logic with which one can reason about the operational aspects of the language. A key difference with the step-indexed logical relations, then, is that abstraction is made of the actual step indices and their sometimes obstructive arithmetic. While Plotkin and Abadi treated System F, this logic covers recursive types as well. It might be interesting to adapt this logic from the call-by-value semantics it was built for, to call-by-name. Then, we could investigate whether, e.g., the soundness and completeness properties of Dreyer et al.'s logic for call-by-value semantics transfer to the call-by-name semantics of the language $\lambda\forall\mu$ from this thesis.

Appendices

Appendix A

Recursive term sets are well-defined

Definition 47 makes use of mutual recursion to define $\mathcal{V}, \mathcal{E}$. Therefore, the existence and uniqueness of functions $\mathcal{V}, \mathcal{E}$ satisfying the mentioned equations is not trivial. In this appendix, we prove that, assuming the regular axioms of set theory, there is a unique pair V, E of sets (in this case functions) that satisfies the equations and required function signatures for $\mathcal{V}, \mathcal{E}$ in Definition 47. From a mathematically rigorous standpoint, the "definition" of $\mathcal{V}, \mathcal{E}$ in Definition 47 should be seen as a characterisation of the V, E pair defined in this appendix.

A.1 Existence

Our proof of existince of V, E is a formalisation of the following, informal argument. Looking at the definition of $\mathcal{V}$, we see that $\mathcal{V}$ applied to k, i.e. $\mathcal{V}^k_-[\![-]\!]_-$, depends on $\mathcal{E}^j_-[\![-]\!]_-$, but only for $j < k$. Meanwhile, $\mathcal{E}^j_-[\![-]\!]_-$ depends on $\mathcal{V}^i_-[\![-]\!]_-$, but only for $i \leq j$. Hence, we can "inline $\mathcal{E}$ in $\mathcal{V}$", i.e. every time $\mathcal{E}$ is mentioned in the definition of $\mathcal{V}$, we can replace it with the definition of $\mathcal{E}$. This shows us that $\mathcal{V}^k_-[\![-]\!]_-$ can be expressed in terms of $\mathcal{V}^i_-[\![-]\!]_-$, where $i < k$. This means V can be defined simply through a recursion theorem based on strong induction. We can then define E in terms of the newly defined V.

To make the above more rigorous, we make use of a classic theorem in set theory. Goldrei calls it the *recursion principle for ordinals.*[9, p. 222] We will use a less general instantiation:

Theorem 120 (Recursion principle for ordinals)**.** *Suppose we have a formula $\phi(k, y, z)$ such that for all ordinals k and all sets y, there is exactly one set z such that $\phi(k, y, z)$. Then there is exactly one function f from $\mathbb{N}$ such that for all $k \in \mathbb{N}$, $\phi(k, (f \restriction k), (f\ k))$.*

Here, we make use of the fact that the naturals form an ordinal, where $j < k \Leftrightarrow j \in k$. Thus, $f \restriction k$ is the restriction of f to naturals strictly smaller than k.

The outline of our existence proof will be: define a ϕ; prove it satisfies the requirements; let V be the function resulting from the recursion theorem; define a function E in terms of V; prove that V, E satisfy the equations and function signatures for $\mathcal{V}, \mathcal{E}$ in Definition 47.

In order to define ϕ, we first need to define the following helper function:

Definition 121. *H is a function that takes a function $X \in (\mathbb{N} \rightharpoonup DTD \to \mathcal{P}\ CVal)$, a natural number k such that X is defined on all $j < k$, and a tuple in DTD. Its output is as follows:*

$$H\ X\ k\ \langle \Delta, \tau, \delta \rangle := \{e \in CTerm \mid \forall j \leq k : \forall e' : e \Downarrow^j e' \Rightarrow e' \in X\ (k-j)\ \langle \Delta, \tau, \delta \rangle\}.$$

Definition 122 (ϕ)**.** *We let $\phi(k, y, z)$ be the following sentence:*

$$\begin{aligned}
\phi(k,y,z) := \quad & (k \notin \mathbb{N} \vee y \notin (k \to \mathit{DTD} \to \mathcal{P}\ \mathit{CVal}) \Rightarrow z = 0) \\
& \wedge (k \in \mathbb{N} \wedge y \in (k \to \mathit{DTD} \to \mathcal{P}\ \mathit{CVal}) \Rightarrow z \in (\mathit{DTD} \to \mathcal{P}\ \mathit{CVal}) \\
& \quad \wedge (\forall \langle \Delta, \tau, \delta \rangle \in \mathit{DTD} : \\
& \quad\quad \wedge (\forall \alpha \in \mathit{TVar} : (\tau = \alpha) \Rightarrow \psi_{\mathrm{var}}(z)) \\
& \quad\quad \wedge (\forall \tau_1, \tau_2 \in \mathit{Type} : (\tau = \tau_1 \to \tau_2) \Rightarrow \psi_{\to}(z)) \\
& \quad\quad \wedge (\forall \tau' \in \mathit{Type}, \alpha \in \mathit{TVar} : (\tau = \forall \alpha . \tau') \Rightarrow (\psi_{\forall})) \\
& \quad\quad \wedge (\forall \tau' \in \mathit{Type}, \alpha \in \mathit{TVar} : (\tau = \mu \alpha . \tau') \Rightarrow (\psi_{\mu})))),
\end{aligned}$$

where we have separate abbreviations $\psi_{\mathrm{var}}, \psi_{\to}, \psi_{\forall}, \psi_{\mu}$ for the cases of τ:

$$\begin{aligned}
\psi_{\mathrm{var}}(z) :=& (z\ \langle \Delta, \alpha, \delta \rangle = (\delta^{\mathrm{sem}}\ \alpha)^k), \\
\psi_{\to}(z) :=& (z\ \langle \Delta, \tau_1 \to \tau_2, \delta \rangle = T(\delta\ \tau_1 \to \delta\ \tau_2) \cap \{\lambda x{:}\delta\ \tau_1.\, e \in \mathit{CVal} \mid \\
& \forall j < k : \forall e' \in H\ y\ j\ \langle \Delta, \tau_1, \delta \rangle : e[e'/x] \in H\ y\ j\ \langle \Delta, \tau_2, \delta \rangle \}), \\
\psi_{\forall}(z) :=& (z\ \langle \Delta, \forall \alpha . \tau', \delta \rangle = T(\forall \alpha . \delta\ \tau') \cap \{\Lambda \alpha . e \in \mathit{CVal} \mid \\
& \forall j < k : \forall \tau'' \in \mathit{Type} : \forall \chi \in \mathit{Rel}(\tau'') : e[\tau''/\alpha] \in H\ y\ j\ \langle \Delta, \tau', \delta[\alpha \mapsto \langle \tau'', \chi \rangle] \rangle \}), \\
\psi_{\mu}(z) :=& (z\ \langle \Delta, \mu \alpha . \tau', \delta \rangle = T(\mu \alpha . \delta\ \tau') \cap \{\texttt{in}\ e \in \mathit{CVal} \mid \\
& \forall j < k : e \in H\ y\ j\ \langle \Delta, \tau'[\mu \alpha . \tau'/\alpha], \delta \rangle \}),
\end{aligned}$$

Now to prove that ϕ satisfies the requirements.

Lemma 123. *For every ordinal k and every set y, there is exactly one set z such that $\phi(k, y, z)$.*

Proof. Fix an arbitrary ordinal k and set y. We must prove there is exactly one set z such that $\phi(k, y, z)$. The definition of ϕ clearly lets us distinguish between two cases: out of $k \notin \mathbb{N}$ and $y \notin (k \to \mathit{DTD} \to \mathcal{P}\ \mathit{CVal})$, either at least one of them is not true, or they are both true. The first case clearly leaves 0 as the only possible value for z. We turn to the more interesting case where $k \in \mathbb{N} \wedge y \in (k \to \mathit{DTD} \to \mathcal{P}\ \mathit{CVal})$. The set of possible values for z is restricted to $\mathit{DTD} \to \mathcal{P}\ \mathit{CVal}$. For every input value $\langle \Delta, \tau, \delta \rangle \in \mathit{DTD}$ we know that $\Delta \vdash \tau$, and therefore exactly one of the four cases described in ϕ applies. All of them uniquely specify the output of z. We conclude that the unique suitable value of z is the function in $\mathit{DTD} \to \mathcal{P}\ \mathit{CVal}$ that maps all $\langle \Delta, \tau, \delta \rangle$ in DTD onto the unique values described by $\psi_{\mathrm{var}}, \psi_{\to}, \psi_{\forall}, \psi_{\mu}$. ☐

This now allows us to apply the recursion theorem and obtain a function. This function will be our V and we use it to define E.

Definition 124 (V)**.** *We apply the recursion principle for ordinals (Theorem 120) to ϕ and obtain the existence and uniqueness of a function, which we call V, such that $\phi(k, (V \restriction k), (V\ k))$ for all $k \in \mathbb{N}$.*

Definition 125 (E)**.** *We define the function E to be $H\ V$.*

We now show that if we substitute V, E for $\mathcal{V}, \mathcal{E}$ in Definition 47, the equations are satisfied. This finishes the existence proof, since it shows that functions as "defined" in Definition 47 exist. We will need the following two lemmas.

Lemma 126. *For all $k \in \mathbb{N}$, we have that $(V \restriction k) \in (k \to \mathit{DTD} \to \mathcal{P}\ \mathit{CVal})$. Since* $\operatorname{dom} V = \mathbb{N}$, *we see that $V \in (\mathbb{N} \to \mathit{DTD} \to \mathcal{P}\ \mathit{CVal})$.*

Proof. By induction on k with a vacuously true base case. Supposing the lemma holds for k, we see that $V \restriction (k+1)$ maps every $j < k$ to something in $\mathit{DTD} \to \mathcal{P}\ \mathit{CVal}$. We also know that $\phi(k, (V \restriction k), (V\ k))$. Looking at the definition of ϕ and invoking the induction hypothesis, we see that $V\ k \in \mathit{DTD} \to \mathcal{P}\ \mathit{CVal}$. This finishes the proof, since we now know every $j < k+1$ gets mapped to something in $\mathit{DTD} \to \mathcal{P}\ \mathit{CVal}$ by V and thus also by $V \restriction (k+1)$. $\square$

Lemma 127. *For all $k \in \mathbb{N}$ and $j < k$, we have that $E\ j = H\ (V \restriction k)\ j$.*

Proof. Fix $k \in \mathbb{N}$ and $j < k$. Recall that $E\ j = H\ V\ j$. Looking at the definition of H, we see that the only apparent difference between $H\ V\ j$ and $H\ (V \restriction k)\ j$ is in their use of $V\ (j-i)$ and $(V \restriction k)\ (j-i)$, respectively, with $i \le j$. Since these $j - i$ are all values strictly smaller than k, there really is no difference. $\square$

We are now ready to show that V, E indeed satisfy the equations for $\mathcal{V}, \mathcal{E}$ in Definition 47.

Proposition 128 (Existence of solutions to $\mathcal{V}, \mathcal{E}$ equations)**.** *When one substitutes V for $\mathcal{V}$ and E for $\mathcal{E}$ in the five equations of Definition 47, the equations are satisfied.*

Proof. We prove the first, second and fifth equations; the other two are similar. The first equation requires for every $k \in \mathbb{N}$ and $\langle \Delta, \alpha, \delta \rangle \in \mathit{DTD}$ that $V\ k\ \langle \Delta, \alpha, \delta \rangle := (\delta^{\text{sem}}\ \alpha)^k$. Because of Lemma 126, $\phi(k, (V \restriction k), (V\ k))$ tells us that $\psi_{\text{var}}(V\ k)$ holds, i.e. $V\ k\ \langle \Delta, \alpha, \delta \rangle = (\delta^{\text{sem}}\ \alpha)^k$ for all $k \in \mathbb{N}$ and $\langle \Delta, \alpha, \delta \rangle \in \mathit{DTD}$. This proves the first equation.

The second equation requires for all k and $\langle \Delta, \tau_1 \to \tau_2, \delta \rangle$ that

$$\begin{aligned} V\ k\ \langle \Delta, \tau_1 \to \tau_2, \delta \rangle = T(\delta\ \tau_1 \to \delta\ \tau_2) \cap \{ (\lambda x{:}\delta\ \tau_1.\, e) \in \mathit{CVal} \mid \\ \forall j < k : \forall e' \in E\ j\ \langle \Delta, \tau_1, \delta \rangle : e[e'/x] \in E\ j\ \langle \Delta, \tau_2, \delta \rangle \}. \end{aligned} \tag{A.1}$$

Because of Lemma 126, $\phi(k, (V \restriction k), (V\ k))$ tells us that $\psi_{\to}(V\ k)$ holds for all k and $\langle \Delta, \tau_1 \to \tau_2, \delta \rangle$, i.e.

$$\begin{aligned} V\ k\ \langle \Delta, \tau_1 \to \tau_2, \delta \rangle = T(\delta\ \tau_1 \to \delta\ \tau_2) \cap \{ (\lambda x{:}\delta\ \tau_1.\, e) \in \mathit{CVal} \mid \\ \forall j < k : \forall e' \in H\ (V \restriction k)\ j\ \langle \Delta, \tau_1, \delta \rangle : e[e'/x] \in H\ (V \restriction k)\ j\ \langle \Delta, \tau_2, \delta \rangle \}. \end{aligned}$$

The apparent difference between $E\ j$ and $H\ (V \restriction k)\ j$ is shown by Lemma 127 to be non-existent. This proves the second equation.

The fifth equation requires

$$E\ k\ \langle \Delta, \tau, \delta \rangle = \{ e \in \mathit{CTerm} \mid \forall j \le k : \forall e' : e \Downarrow^j e' \Rightarrow e' \in V\ (k-j)\ \langle \Delta, \tau, \delta \rangle \} \tag{A.2}$$

for all k and $\langle \Delta, \tau, \delta \rangle$. We see that Eq. (A.2) follows from $E = H\ V$ and the definition of H. $\square$

A.2 Uniqueness

We now prove that V, E are in fact the only functions that satisfy the five equations of Definition 47.

Proposition 129 (Uniqueness of solution to $\mathcal{V}, \mathcal{E}$ equations)**.** *Suppose that V', E' are functions of the signatures required for $\mathcal{V}, \mathcal{E}$ in Definition 47. Also suppose that V', E' are such that if one substitutes them for $\mathcal{V}, \mathcal{E}$ in the equations of Definition 47, the equations are satisfied. Then $V' = V$ and $E' = E$.*

Proof. Fix such V', E'. Since $V' \in (\mathbb{N} \to DTD \to \mathcal{P}\ CVal)$ and $E' \in (\mathbb{N} \to DTD \to \mathcal{P}\ CTerm)$, it suffices to prove that $V'\ k\ \langle\Delta, \tau, \delta\rangle = V\ k\ \langle\Delta, \tau, \delta\rangle$ and $E'\ k\ \langle\Delta, \tau, \delta\rangle = E\ k\ \langle\Delta, \tau, \delta\rangle$ for all $k \in \mathbb{N}$ and $\langle\Delta, \tau, \delta\rangle \in DTD$. We will prove this by strong induction on k, within which we perform exhaustive case distincition on $\Delta \vdash \tau$ in much the same way as we proved Lemma 69.

For the strong induction, suppose that $k \geq 0$ and suppose that $V'\ j = V\ j$ and $E'\ j = E\ j$ for all $j < k$. We first prove $V'\ k = V\ k$, using extensionality, by generalisation on $\langle\Delta, \tau, \delta\rangle \in DTD$ and by exhaustive case distinction on $\Delta \vdash \tau$. The case where τ is some type variable α does not even need induction: the first equation tells us that both $V'\ k$ and $V\ k$ map $\langle\Delta, \tau, \delta\rangle$ onto $(\delta^{\text{sem}}\ \alpha)^k$.

In all other cases, we see that the only apparent differences between the images of $\langle\Delta, \tau, \delta\rangle$ under $V'\ k$ and $V\ k$ is in their mentioning of E' and E, respectively. However, these are only applied to natural numbers strictly below k, thus by the induction hypothesis, those applications are equal and therefore so are $V'\ k$ and $V\ k$.

In this final step we prove, again by extensionality, that $E'\ k = E\ k$. Looking at the fifth equation, we see that the only apparent difference between the images of $\langle\Delta, \tau, \delta\rangle$ under $E'\ k$ and $E\ k$ is in their mentioning of V' and V, respectively. However, since these are only applied to natural numbers $j \leq k$, the above proof of $V'\ k = V\ k$, combined with the induction hypothesis, tells us that $E'\ k\ \langle\Delta, \tau, \delta\rangle = E\ k\ \langle\Delta, \tau, \delta\rangle$.

□

Appendix B

Contextual equivalence

In this appendix, we argue why Definition 81 is a good definition of contextual equivalence. First, we must realise that in this thesis, we are only interested in obtaining a *sound* proof technique w.r.t. contextual equivalence, not a complete one. We proved that $e^l \diamond e^r$ implies $e^l \approx^{ctx} e^r$ (Theorem 92). Since we are currently questioning the use of $\approx^{ctx}$ as we defined it in Definition 81, we will propose a new, perhaps more acceptable definition $\approx^{ctx}_{\mathbb{B}v}$, and then prove that $e^l \approx^{ctx} e^r$ implies $e^l \approx^{ctx}_{\mathbb{B}v} e^r$. (We will refer to $\approx^{ctx}_{\mathbb{B}v}$ as the "trusted definition".) That way, $\diamond$ will still be a sound proof technique for contextual equivalence understood as $\approx^{ctx}_{\mathbb{B}v}$.

Looking back at Definition 77 of contextual equivalence in Section 4.1 using a ground type $\mathbb{B}$, and at the discussion in Section 4.2.1, we conclude that the best definition of contextual equivalence is in fact Definition 77, since it allows for a syntactic distinction between terms. However, as we have pointed out before, we do not actually dispose of the necessary ground type $\mathbb{B}$ in $\lambda\forall\mu$.

We work around this problem in the following way. We single out $\tau_{\mathbb{B}} := \forall\alpha.\alpha \to \alpha \to \alpha$ as the type that will represent booleans. The intended meaning is that terms of type $\tau_{\mathbb{B}}$ either always reduce to their first or always reduce to their second argument and can have no mixed or other behaviour. We can then say, per convention, that the former kind of term is *true-like* and the latter is *false-like*. The advantage of this is that the syntactic term distinction we had in $\mathbb{B}$ in Section 4.1 would be regained. Determining whether e of type $\tau_{\mathbb{B}}$ is true-like or false-like is then decidable, since $e\ \tau_{id}\ id\ \Omega_{\tau_{id}}\ \tau\ x$ evaluates to x and $e\ \tau_{id}\ \Omega_{\tau_{id}}\ id\ \tau\ x$ does not evaluate if e is true-like, and it is precisely the other way around for false-like terms.

Of course there is one problem. The basic divergent term Ω of type $\tau_{\mathbb{B}}$ never terminates and thus is neither true-like nor false-like. We therefore must divide the terms of type $\tau_{\mathbb{B}}$ into not two but three equivalence classes: true-like, false-like, and *omega-like*. We now formally define the three classes:

Definition 130 (Equivalence classes of $\tau_{\mathbb{B}}$)**.** *We define the following three classes of terms e of type $\tau_{\mathbb{B}}$:*

e is true-like *iff it "behaves the same as its first argument". In symbols:*

$$\forall \tau \in \mathit{CType}, e_1, e_2 \in \mathit{Term} : (\cdot;\cdot \vdash e_1 : \tau_{\mathbb{B}}) \wedge (\cdot;\cdot \vdash e_2 : \tau_{\mathbb{B}}) \Rightarrow$$
$$(e_1 \Downarrow e_1' \Rightarrow e\ \tau\ e_1\ e_2 \Downarrow e_1') \wedge (e_1 \not\Downarrow \Rightarrow e\ \tau\ e_1\ e_2 \not\Downarrow).$$

e is false-like *iff it "behaves the same as its second argument". In symbols:*

$$\forall \tau \in \mathit{CType}, e_1, e_2 \in \mathit{Term} : (\cdot;\cdot \vdash e_1 : \tau_{\mathbb{B}}) \wedge (\cdot;\cdot \vdash e_2 : \tau_{\mathbb{B}}) \Rightarrow$$
$$(e_2 \Downarrow e_2' \Rightarrow e\ \tau\ e_1\ e_2 \Downarrow e_2') \wedge (e_2 \not\Downarrow \Rightarrow e\ \tau\ e_1\ e_2 \not\Downarrow).$$

e is omega-like *iff it "behaves the same as Ω". In symbols:*

$$\forall \tau \in \mathit{CType}, e_1, e_2 \in \mathit{Term} : (\cdot; \cdot \vdash e_1 : \tau_{\mathbb{B}}) \wedge (\cdot; \cdot \vdash e_2 : \tau_{\mathbb{B}}) \Rightarrow e\ \tau\ e_1\ e_2 \not\Downarrow .$$

We say terms are $\tau_{\mathbb{B}}$-equivalent (notation: $e \sim_{\tau_{\mathbb{B}}} e'$) iff they are of the same class.

We conjecture the following trichotomy:

Conjecture 131 (Trichotomy of $\tau_{\mathbb{B}}$)**.** *Every term e of type $\tau_{\mathbb{B}}$ belongs to exactly one of the following classes: true-like, false-like, omega-like.*

Looking at the examples $e\ \tau\ id\ \Omega$ and $e\ \tau\ \Omega\ id$ from earlier, we see that the classes are indeed mutually exclusive. We do not attempt to prove that e is in *at least* one of the classes.

We can now define this new, "trusted" notion $\approx^{ctx}_{\mathbb{B}v}$ of contextual equivalence, in the spirit of Definition 77. Contexts are wellformed w.r.t. the type $\tau_{\mathbb{B}}$, which is our closest approximation of $\mathbb{B}$. To account for the impossibility of syntactic distinction between values of type $\tau_{\mathbb{B}}$, we use the division into the three equivalence classes:

Definition 132 (Trusted version of contextual equivalence)**.** *We define a binary term relation $\approx^{ctx}_{\mathbb{B}v}$. More specifically, it relates a type environment Δ, an environment Γ, two terms e^l, e^r, and a type τ. The notation for $\langle \Delta, \Gamma, e^l, e^r, \tau \rangle \in \approx^{ctx}_{\mathbb{B}v}$ is $\Delta; \Gamma \vDash e^l \approx^{ctx}_{\mathbb{B}v} e^r : \tau$ and we say it holds iff e^l and e^r filled into the same wellformed $\tau_{\mathbb{B}}$-context evaluate to $\tau_{\mathbb{B}}$-equivalent values. Formally:*

$$(\Delta; \Gamma \vDash e^l \approx^{ctx}_{\mathbb{B}v} e^r : \tau) :\Leftrightarrow \forall C \in \mathit{Ctx}, v^l, v^r \in \mathit{Val} :$$
$$(\cdot; \cdot \vdash^c C : (\Delta; \Gamma \rhd \tau) \rightsquigarrow \tau_{\mathbb{B}}) \wedge (C[e^l] \Downarrow v^l) \wedge (C[e^r] \Downarrow v^r) \Rightarrow (v^l \sim_{\tau_{\mathbb{B}}} v^r).$$

The central point of this appendix, then, is to show that whenever $e \approx^{ctx} e'$ in the sense of Definition 81—the one used in the thesis—then also $e \approx^{ctx}_{\mathbb{B}v} e'$. Thus, the sound proof technique using $\diamond$ remains sound w.r.t. this new, trusted notion of contextual equivalence: $\diamond \subseteq \approx^{ctx} \subseteq \approx^{ctx}_{\mathbb{B}v}$.

Proposition 133 (Contextual equivalence from the thesis implies the new notion)**.** $\approx^{ctx} \subseteq \approx^{ctx}_{\mathbb{B}v}$. *In other words, $\Delta; \Gamma \vDash e^l \approx^{ctx} e^r : \tau$ implies $\Delta; \Gamma \vDash e^l \approx^{ctx}_{\mathbb{B}v} e^r : \tau$.*

Proof. Suppose that $\Delta; \Gamma \vDash e^l \approx^{ctx} e^r : \tau$. Also fix an arbitrary context C such that $\cdot; \cdot \vdash^c C : (\Delta; \Gamma \rhd \tau) \rightsquigarrow \tau_{\mathbb{B}}$ and two values v^l, v^r such that $C[e^l] \Downarrow v^l$ and $C[e^r] \Downarrow v^r$. It thus remains to prove that $v^l \sim_{\tau_{\mathbb{B}}} v^r$.

- In the first case, suppose that v^l is omega-like. We prove that v^r is omega-like as well by transforming C such that we can apply the assumption $e^l \approx^{ctx} e^r$ to it.

  Define $C' := C\ \tau_{id}\ id\ id$. Clearly, $\cdot; \cdot \vdash^c C' : (\Delta; \Gamma \rhd \tau) \rightsquigarrow \tau_{id}$. We can thus apply $\Delta; \Gamma \vDash e^l \approx^{ctx} e^r : \tau$ to C' and τ_{id}. It tells us that $C'[e^l] \Downarrow \Leftrightarrow C'[e^r] \Downarrow$.

  Now $C'[e^l] = C[e^l]\ \tau_{id}\ id\ id$ reduces to $v^l\ \tau_{id}\ id\ id$, which does not terminate since v^l is omega-like. Therefore, $C'[e^r]$, which reduces to $v^r\ \tau_{id}\ id\ id$, does not terminate either. This eliminates the possibility that v^r is true-like or false-like. By the trichotomy (Prop. 131), then, v^r is omega-like.

- In the second case, we suppose that v^l is true-like. This time we construct two separate contexts C' and C''. Using C', we will eliminate the possibility that v^r is omega-like, and C'' will help us eliminate the possibility that v^r is false-like.

  We define $C' := C\ \tau_{id}\ id\ \Omega$, with Ω of type τ_{id}. We apply the assumption to C' and τ_{id} to get $C'[e^l] \Downarrow \Leftrightarrow C'[e^r] \Downarrow$. Now $C'[e^l]$ reduces to $v^l\ \tau_{id}\ id\ \Omega$, which evaluates to id since

v^l is true-like. Therefore, $C'[e^r]$, which reduces to $v^r\ \tau_{id}\ id\ \Omega$, terminates as well. This eliminates the possibility that v^r is omega-like.

We move to the second context. It is defined $C'' := C\ \tau_{id}\ \Omega\ id$. We again apply the assumption and get $C''[e^l] \Downarrow \Leftrightarrow C''[e^r] \Downarrow$. Now $C''[e^l]$ reduces to $v^l\ \tau_{id}\ \Omega\ id$, which does not terminate since v^l is true-like. Therefore, $C''[e^r]$, which reduces to $v^r\ \tau_{id}\ \Omega\ id$, does not terminate either. This eliminates the possibility that v^r is false-like.

By the trichotomy (Prop. 131), then, v^r is true-like.

The case where v^l is false-like is analogous to the second case. $\square$

Bibliography

[1] Samson Abramsky and Achim Jung. “Domain Theory”. In: *Handbook of Logic in Computer Science*. Ed. by S. Abramsky, Dov M. Gabbay and T. S. E. Maibaum. Vol. 3. Clarendon Press, 1994. Chap. 1, pp. 1–168.

[2] Amal Ahmed. “Semantics of Types for Mutable State”. PhD thesis. Princeton University, 2004. URL: `http://www.ccs.neu.edu/home/amal/ahmedsthesis.pdf` (visited on 2020-05-26).

[3] Amal Ahmed. “Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types”. In: *Programming Languages and Systems. 15th European Symposium on Programming, ESOP 2006*. Ed. by Peter Sestoft. Note that the url leads to the technical report. Berlin, Heidelberg: Springer-Verlag Berlin Heidelberg, 2006, pp. 69–83. ISBN: 978-3-540-33096-7. DOI: `10.1007/11693024`. URL: `http://www.ccs.neu.edu/home/amal/papers/lr-recquant-techrpt.pdf` (visited on 2020-05-26).

[4] Andrew W. Appel and David McAllester. “An Indexed Model of Recursive Types for Foundational Proof-Carrying Code”. In: *ACM Trans. Program. Lang. Syst.* 23.5 (Sept. 2001), pp. 657–683. ISSN: 0164-0925. DOI: `10.1145/504709.504712`.

[5] H. P. Barendregt. *The Lambda Calculus. Its Syntax and Semantics*. Elsevier, 1984. ISBN: 0-444-87508-5.

[6] Henk Barendregt, Wil Dekkers and Richard Statman. *Lambda Calculus with Types*. Cambridge University Press, 2013. ISBN: 9780521766142.

[7] Derek Dreyer, Amal Ahmed and Lars Birkedal. “Logical Step-Indexed Logical Relations”. In: *Logical Methods in Computer Science* 7.2 (June 2011). DOI: `10.2168/LMCS-7(2:16)2011`. URL: `https://arxiv.org/abs/1103.0510v2`.

[8] Herman Geuvers. “Introduction to Type Theory”. 2008. URL: `http://www.cs.ru.nl/~herman/onderwijs/provingwithCA/paper-lncs.pdf` (visited on 2020-07-30).

[9] Derek Goldrei. *Classic Set Theory. For Guided Independent Study*. New York : Chapman & Hall, 1996. ISBN: 978-0-412-60610-6.

[10] Benjamin C. Pierce, ed. *Advanced Topics in Types and Programming Languages*. MIT Press, 2014. ISBN: 9780262162289.

[11] Andrew M. Pitts. “Operationally-Based Theories of Program Equivalence”. In: *Semantics and Logics of Computation*. Ed. by Andrew M. Pitts and Peter Dybjer. Cambridge University Press, 1997. Chap. 6, pp. 241–298. ISBN: 978-0521580571. URL: `https://www.cl.cam.ac.uk/~amp12/papers/opebtp/opebtp.pdf`.

[12] Andrew M. Pitts. “Parametric Polymorphism and Operational Equivalence”. In: *Mathematical Structures in Computer Science* 10.3 (2000), pp. 321–359. DOI: `10.1017/S0960129500003066`. URL: `https://www.cl.cam.ac.uk/~amp12/papers/parpoe/parpoe.pdf`.

[13] Andrew M. Pitts, Glynn Winskel and Marcelo Fiore. “Lecture Notes on Denotational Semantics”. 2018. URL: `https://www.cl.cam.ac.uk/teaching/1819/DenotSem/DenotSem2018.pdf` (visited on 2020-03-30).

[14] Gordon Plotkin and Martín Abadi. “A logic for parametric polymorphism”. In: *Typed Lambda Calculi and Applications*. Ed. by Marc Bezem and Jan Friso Groote. Springer Berlin Heidelberg, 1993, pp. 361–375. ISBN: 978-3-540-47586-6. DOI: `10.1007/BFb0037118`.

[15] Dana S. Scott. “Domains for denotational semantics”. In: *Automata, Languages and Programming*. Ed. by Mogens Nielsen and Erik Meineche Schmidt. Springer Berlin Heidelberg, 1982, pp. 577–610. ISBN: 978-3-540-39308-5. DOI: `10.1007/BFb0012801`.

[16] Lau Skorstengaard. “An Introduction to Logical Relations: Proving Program Properties Using Logical Relations”. 2017. URL: `https://cs.au.dk/~lask/main.pdf` (visited on 2020-05-26).

[17] Eijiro Sumii and Benjamin C. Pierce. “A Bisimulation for Type Abstraction and Recursion”. In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’05. Long Beach, California, USA: Association for Computing Machinery, 2005, pp. 63–74. ISBN: 158113830X. DOI: `10.1145/1040305.1040311`.

[18] Philip Wadler. “Theorems for Free!” In: *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*. FPCA ’89. Imperial College, London, United Kingdom: Association for Computing Machinery, 1989, pp. 347–359. ISBN: 0897913280. DOI: `10.1145/99370.99404`.