Logical Methods in Computer Science
Volume 16, Issue 4, 2020, pp. 12:1–12:50
https://lmcs.episciences.org/

Submitted Mar. 28, 2018
Published Dec. 01, 2020

AN EXTENDED TYPE SYSTEM WITH LAMBDA-TYPED LAMBDA-EXPRESSIONS

MATTHIAS WEBER

Faculty of Computer Science, Technical University of Berlin
e-mail address: mattnweber@gmail.com

Abstract. We present the system d, an extended type system with lambda-typed lambda-expressions. It is related to type systems originating from the Automath project. d extends existing lambda-typed systems by an existential abstraction operator as well as propositional operators. β-reduction is extended to also normalize negated expressions using a subset of the laws of classical negation, hence d is normalizing both proofs and formulas which are handled uniformly as functional expressions. d is using a reflexive type axiom for a constant τ to which no function can be typed. Some properties are shown including confluence, subject reduction, uniqueness of types, strong normalization, and consistency. We illustrate how, when using d, due to its limited logical strength, additional axioms must be added both for negation and for the mathematical structures whose deductions are to be formalized.

1. Introduction and Overview

In this paper, we will present an extended type system with lambda-typed lambda-expressions. Since such systems have received little attention we begin with some general remarks to provide some context.

Most type systems contain subsystems that can be classified as instances of *pure type systems* (PTS) (*e.g.* [4]). As one of their properties, these systems use distinct operators to form dependent products and functional (i.e. λ)-abstractions. This reflects their underlying semantic distinction between the domains of functions and types. In contrast, in the semantics of lambda-typed systems there is a single domain of (partial) functions and typing is a binary relation between total functions. As a consequence a function participating in the type relation may play the role of an element or of a type. Moreover, since in general the domain and range of the type relation are not disjoint a function can (and will usually) have a double role both as element and as type. Therefore in lambda-typed calculi one has to separate three aspects of an operator: its functional interpretation, i.e. the equivalence class of entities it represents, its role as a type, i.e. the entities on the element side related to its entities in the type relation, and its role as an element, i.e. the entities on the type side related to its entities in the type relation. From this point of view, the semantic distinction between dependent products and λ-abstractions can be reduced to the distinction between

Key words and phrases: Typed lambda calculus, Higher-order logic, Strong normalization.

the type-role or the element-role of a single underlying function and therefore, when defining a calculus, a single operator is sufficient.

Type systems outside of PTS using a single operator for both dependent products and λ-abstractions (i.e. using λ-*structured types*) have been investigated in early type systems such as Λ [10, 25], as well as more recent approaches [15, 18]. In particular [18] introduces the single-binder based $\flat$-cube, a variant of the β-cube which does not keep uniqueness of types and studies λ-structured type variants of well-known systems within this framework.

Before we present the basic elements of `d`, we would like to point out its additional underlying semantic assumptions:

- The range of the type relation is a subset of its domain, or in other words, every type has a type.
- The type relation includes entities typing to themselves, which allows for generating type hierarchies of arbitrary size from a finite set of distinct base entities. We discuss below why in `d` this assumption does not lead to well-known paradoxes.
- There are no inequivalent types of an element, or in other words, the type relation is restricted to be a function. This means that `d` needs to satisfy the type uniqueness property (see Section 4.42). We will review this assumption in Section 5.4.

It is well known that types can be interpreted as propositions and elements as proofs [16]. In our semantic setting this analogy is valid and we will make use of it throughout the paper, for example when motivating and describing the roles of operators we will frequently use the viewpoint of propositions and proofs.

Note that in our setting the interest is in formalization of structured mathematical reasoning rather than in computational proof content.

Finally we would like to make a notational remark related to `d` but also to lambda-typed calculi in general. Since an operator in `d` can be interpreted both as element and as type (or proposition), there is, in our view, a notational and naming dilemma. For example, one and the same entity would be written appropriately as a typed lambda-abstraction $\lambda x\!:\!a.b$ in its role as an element and as a universal quantification $\forall x\!:\!a.b$ in its role as a proposition and neither notation would adequately cover both roles. Therefore, in this case we use a more neutral notation $[x : a]b$ called *universal abstraction*. As will be seen this notational issue also applies to other operators of `d`.

After these remarks we can now turn to the core of `d` which is the system λ^λ [14], a reconstruction of a variation [25] of Λ, modified with a reflexive type axiom with a constant τ, see Table 1 for its type rules. As usual, we use contexts $\Gamma = (x_1 : a_1, \dots, x_n : a_n)$ declaring

$$
\begin{array}{llll}
(\mathit{ax}) \ \dfrac{}{\vdash\ \tau : \tau} & & (\mathit{start}) \ \dfrac{\Gamma \vdash\ a : b}{\Gamma, x : a \vdash\ x : a} & \\[2ex]
(\mathit{weak}) \ \dfrac{\Gamma \vdash\ a : b \qquad \Gamma \vdash\ c : d}{\Gamma, x : c \vdash\ a : b} & & (\mathit{conv}) \ \dfrac{\Gamma \vdash\ a : b \qquad b =_\lambda c \qquad \Gamma \vdash\ c : d}{\Gamma \vdash\ a : c} & \\[2ex]
(\mathit{abs}_U) \ \dfrac{\Gamma, x : a \vdash\ b : c}{\Gamma \vdash\ [x : a]b : [x : a]c} & & (\mathit{appl}) \ \dfrac{\Gamma \vdash\ a : [x : b]c \qquad \Gamma \vdash\ d : b}{\Gamma \vdash\ (a\, d) : c[d/x]} &
\end{array}
$$

Table 1: The kernel of `d`: The system λ^λ [14] modified with $\tau : \tau$

types of distinct variables and a β-conversion induced congruence $=_\lambda$ on expressions. $\Gamma, x : a$

denotes the extension of Γ on the right by a binding $x : a$ where x is a variable not yet declared in Γ. Furthermore if $n = 0$ we omit writing contexts. We use the notation $a[b/x]$ to denote the substitution of all free occurrences of x in a by b.

Element-roles and type-roles can now be illustrated by two simple examples: Let $C = [x : \tau]\tau$ be the constant function delivering τ, for any given argument of type τ. In its role as a type C corresponds to an implication ("τ implies τ") and its proofs include the identity $I = [x : \tau]x$ over τ, i.e. $\vdash I : C$ using essentially the rules *start* and abs_U. In its role as an element C corresponds to a constant function and it will type to itself, i.e. $\vdash C : C$ using essentially the rules *ax* and abs_U. I, which as an element is obviously the identity function, in its role as a type corresponds to the proposition "everything of type τ is true" which should not have any elements (this is shown in Section 4.4).

More generally, in the rule *appl* the type-role of a universal abstraction ($[x : b]c$) can be intuitively understood as an infinite conjunction of instances ($c[b_1/x] \wedge c[b_2/x] \ldots$ where $b_i : b$ for all i). Individual instances ($c[b_i/x]$) can be projected by means of the application operator. Intuitively, a proof of a universal abstraction ($[x : a]c$) must provide an infinite list of instances ($b[a_1/x], b[a_2/x], \ldots$ where $b[a_i/x] : c[a_i/x]$ for all i). Consequently, in the rule abs_U a universal abstraction ($[x : a]b$ where $x : a \vdash b : c$) has the element-role of being a proof of another universal abstraction ($[x : a]c$).

As another example, the introduction and elimination rules for universal quantification can be derived almost trivially. Note that in this and the following examples we write $[a \Rightarrow b]$ to denote $[x : a]b$ if x does not occur free in b:

$$\begin{array}{rcrcl} P : [a \Rightarrow b] & \vdash & [x : [y : a](P\,y)]x & : & [[y : a](P\,y) \Rightarrow [y : a](P\,y)] \\ P : [a \Rightarrow b] & \vdash & [x : a][z : [y : a](P\,y)](z\,x) & : & [x : a][[y : a](P\,y) \Rightarrow (P\,x)] \end{array}$$

When substituting P by a constant function $[a \Rightarrow c]$ (where $\vdash c : b$) the types simplify to two variants of the modus ponens rule:

$$\begin{array}{rrcl} \vdash & [x : [a \Rightarrow c]]x & : & [[a \Rightarrow c] \Rightarrow [a \Rightarrow c]] \\ \vdash & [x : a][z : [a \Rightarrow c]](z\,x) & : & [a \Rightarrow [[a \Rightarrow c] \Rightarrow c]] \end{array}$$

In contrast to other type systems with λ-structured types, `d` is using a reflexive axiom (*ax*). This might seem very strange as the use of a reflexive axiom in combination with basic rules of PTS leads to paradoxes, *e.g.* [6, 17], see also [4, Section 5.5]. However, as will be seen, in our setting of λ-structured types, the axiom *ax* leads to a consistent system. This is actually not very surprising since λ^λ does not have an equivalent to the product rule used in PTS:

$$(\textit{product})\ \frac{\Gamma \vdash\ a : s_1 \quad \Gamma, x : a \vdash\ b : s_2}{\Gamma \vdash\ (\Pi x : a.b) : s_3} \quad \text{where } s_i \in S,\ S \text{ is a set of sorts}$$

Adding such a rule, appropriately adapted, to the kernel of `d` would violate uniqueness of types [18] and allow for reconstructing well-known paradoxes [32] (see also Section 5.4). Without such a rule, functions from τ such as I do not accept functional arguments such as C[1]. The lack of functions of type τ is the reason for achieving consistency when assuming $\tau : \tau$.

Unlike instances of PTS (*e.g.* [21]), systems with λ-structured types have never been extended by existential or classical propositional operators. While the kernel of `d` is sufficiently expressive to axiomatize basic mathematical structures (Sections 3.2, 3.3) the expressive

[1]In fact, $(I\ C)$ cannot be typed since $\vdash I : [x : \tau]\tau$, hence I is expecting arguments of type τ, but $\vdash C : C$ and $\tau \neq_\lambda C$

and structuring means of deductions can be enhanced by additional operators. We begin with an operator that effectively provides for a *deduction interface*, i.e. a mechanism to hide details of interdependent deductions. For this purpose, in analogy to a universal abstraction ($[x : a]b$), `d` introduces an *existential abstraction* ($[x!a]b$) (see Table 2 for its type rules). The

$$(def)\ \frac{\Gamma \vdash a : b \quad \Gamma \vdash c : d[a/x] \quad \Gamma, x : b \vdash d : e}{\Gamma \vdash [x \doteq a, c : d] : [x!b]d} \qquad (abs_E)\ \frac{\Gamma, x : a \vdash b : c}{\Gamma \vdash [x!a]b : [x : a]c}$$

$$(ch_I)\ \frac{\Gamma \vdash a : [x!b]c}{\Gamma \vdash a.1 : b} \qquad (ch_B)\ \frac{\Gamma \vdash a : [x!b]c}{\Gamma \vdash a.2 : c[a.1/x]}$$

Table 2: Type rules for existential abstractions

notation is intended to maximise coherence with universal abstraction. The type-role of an existential abstraction ($[x!b]d$) can be intuitively understood as an infinite disjunction ($d[b_1/x] \vee d[b_2/x] \dots$ where $b_i : b$ for all i). A proof of an existential abstraction ($[x!b]d$) must prove one of the instances (say $d[b_j/x]$), i.e. it must provide an instance of the quantification domain ($b_j : b$) and an element proving the instantiated formula ($c : d[b_j/x]$). This is formalized in rule *def* with a new operator called *protected definition* combining the two elements and a tag for the abstraction type ($[x \doteq a, c : d]$)[2]. This means that two deductions (a and c), where one (c) is using the other one (a) in its type, are simultaneously abstracted into an existential abstraction type[3]. The element-role of an existential abstraction ($[x!a]b$) is not the one of a logical operator but of an entity providing an infinite list of instances ($b[a_1/x]$, $b[a_2/x]$, ... where $a_i : a$ for all i). But, as we just discussed above with respect to typing of universal abstractions (abs_U), this is sufficient to type it to a universal abstraction (abs_E). Hence the element-roles of universal and existential abstraction are equivalent. Pragmatically the element-role of existential abstraction is less frequently used and of less importance. This is the reason why the notation for existential abstractions is more "type-oriented" than that of universal abstraction.

In analogy to the type-elimination of universal abstraction, *projections* ($a.1$ and $a.2$ [4]) are introduced as type-eliminators for existential abstraction with obvious equivalence laws.

$$[x \doteq a, b : c].1 =_\lambda a \qquad [x \doteq a, b : c].2 =_\lambda b$$

The type rules ch_I and ch_B for projections are similar to common rules for Σ types, *e.g.* [21].

As an example, the introduction and elimination rules for existential quantification can be derived as follows (with $\Gamma = (P, Q : [a \Rightarrow b])$):

$$\Gamma \vdash [x:a][z:(P\,x)][y \doteq x, z : (P\,y)] \;:\; [x : a][(P\,x) \Rightarrow [y!a](P\,y)]$$

$$\begin{aligned}\Gamma \vdash\ & [x:[y_1!a](Py_1)][z:[y_2:a][(Py_2) \Rightarrow (Qy_2)]] \\ & [y_3 \doteq x.1, ((z\,x.1)x.2) : (Qy_3)] \;:\; [[y_1!a](Py_1) \\ & \qquad \Rightarrow [[y_2:a][(Py_2) \Rightarrow (Qy_2)] \Rightarrow [y_3!a](Qy_3)]]\end{aligned}$$

A more detailed explanation of these deductions is given in Sections 2.2 and 2.3. Note how the second example illustrates in a nutshell the role of existential abstraction as a

[2]The type tag d in $[x \doteq a, c : d]$ is necessary to ensure uniqueness of types

[3]In Section 5.5, we discuss the restriction that x may not occur free in c and how it could be relaxed.

[4]A postfix notation has been chosen since it seems more intuitive for projection sequences.

deduction interface. It can be understood as the transformation of a deduction on the basis of a reference (x) to its interface ($[y_1!a](P y_1)$) followed by the creation of a new interface ($[y_3!a](Q y_3)$) hiding the transformation details (application of z to the extracted interdependent elements $x.1$ and $x.2$). More applications will be shown in Section 3, in particular in Sections 3.5 (Partial functions), 3.6 (Defining functions from deductions), and 3.8 (Proof structuring).

The type rule for existential abstractions (abs_E) has the consequence that existential abstractions can now be instantiated as functions in the elimination rule for universal abstraction, i.e. the rule *appl* can be instantiated as follows:

$$(appl)[[x!a_1]a_2/a] \quad \frac{\Gamma \vdash [x!a_1]a_2 : [x : b]c \quad \Gamma \vdash d : b}{\Gamma \vdash ([x!a_1]a_2\, d) : c[d/x]}$$

This motivates the extension of β-equality to existential abstractions, i.e. we have

$$([x : a]b\ c) \ =_\lambda \ b[c/x] \ =_\lambda \ ([x!a]b\ c)$$

Note that these properties merely state that both abstractions have equivalent functional behaviour when applied to arguments. However, they are not identical and this semantic distinction is represented by their role in the type relation. For example, note that from $x : [y!a]b$ and $z : a$ one cannot conclude $(x\, z) : ([y!a]b\, z)$. Note that this approach naturally precludes adding axioms of extensionality as they deny such distinctions.

Finally, one might wonder why not type an existential abstraction $[x!a]b$ to an existential abstraction $[x!a]c$ (assuming $x : a \vdash b : c$)? According to the intuitive understanding of the type role of an existential abstraction as an infinite disjunction this would be logically invalid and indeed it is quite easy to see such a rule would lead to inconsistency[5].

The remaining part of d consists of some propositional operators (see Table 3 for their type rules). d adds a *product* ($[a, b]$) as a binary variation of universal abstraction ($[x : a]b$),

$$(prd)\ \frac{\Gamma \vdash a : c \quad \Gamma \vdash b : d}{\Gamma \vdash [a, b] : [c, d]} \qquad (sum)\ \frac{\Gamma \vdash a : c \quad \Gamma \vdash b : d}{\Gamma \vdash [a + b] : [c, d]}$$

$$(pr_L)\ \frac{\Gamma \vdash a : [b, c]}{\Gamma \vdash a.1 : b} \qquad (pr_R)\ \frac{\Gamma \vdash a : [b, c]}{\Gamma \vdash a.2 : c}$$

$$(inj_L)\ \frac{\Gamma \vdash a : b \quad \Gamma \vdash c : d}{\Gamma \vdash [a, :c] : [b + c]} \qquad (inj_R)\ \frac{\Gamma \vdash a : b \quad \Gamma \vdash c : d}{\Gamma \vdash [:c, a] : [c + b]}$$

$$(case)\ \frac{\Gamma \vdash a : [x : c_1]d \quad \Gamma \vdash b : [y : c_2]d \quad \Gamma \vdash d : e}{\Gamma \vdash [a\,?\,b] : [z : [c_1 + c_2]]d}$$

$$(neg)\ \frac{\Gamma \vdash a : b}{\Gamma \vdash \neg a : b}$$

Table 3: Type rules for propositional operators

i.e. with an element-role as a binary pair and an type-role as a binary conjunction. Due to the same notation dilemma as for universal abstraction, i.e. neither the notation for pairs $\langle a, b\rangle$ nor for conjunctions $a \wedge b$ would be satisfactory, we use the neutral notation $[a, b]$. The

[5]First, with such a rule the type $P = [x : \tau][y!x]\tau$ would have itself as an element, i.e. $\vdash P : P$. However from a declaration of this type one could then extract a proof of $[z : \tau]z$ as follows $y : P \vdash [z : \tau](y\, z).1 : [z : \tau]z$.

type rules *prd*, pr_R, and pr_L for products directly encode the introduction and elimination rules for conjunctions. The equivalence laws for projection are extended in an obvious way.

$$[a,b].1 =_\lambda a \qquad [a,b].2 =_\lambda b$$

As a example we show that existential abstractions without dependencies (i.e. $[x!a]b$, where x does not occur free in b) are logically equivalent to products:

$$\begin{array}{rrcl} \vdash & [y:[x!a]b][y.1,y.2] & : & [[x!a]b \Rightarrow [a,b]] \\ \vdash & [y:[a,b]][x \doteq y.1, y.2:b] & : & [[a,b] \Rightarrow [x!a]b] \end{array}$$

Similarly d adds a sum ($[b_1 + b_2]$) as a binary variation of the existential abstraction ($[x!a]b$), i.e. with an element-role as a binary pair and an type-role as a binary disjunction. Consequently, there is a type sequence analogous to existential abstraction: Expressions ($a_1 : b_1$, $a_2 : b_2$) may be used within *injections* ($[a_1, : b_2]$, $[: b_1, a_2]$) (injections are the analogue to protected definitions), which type to a sum ($[b_1 + b_2]$) which types to a product. The bracket notation for products and abstractions is extended to injections and sums for notational coherence. Similarly to existential abstraction we use a more type-oriented notation for sums. The injection rules directly encode the introduction rules for disjunctions and rule *case* introduces a case distinction operator. Two equivalence laws describe its functional interpretation.

$$([a \mathbin{?} b]\,[c, : d]) =_\lambda (a\,c) \qquad ([a \mathbin{?} b]\,[: c, d]) =_\lambda (b\,d)$$

As an example, the introduction and elimination rules for disjunction can be derived as follows:

$$\begin{array}{rrcl} \vdash & [[x:a][x, : b], [x:a][: b, x]] & : & [[a \Rightarrow [a+b]], [a \Rightarrow [b+a]]] \\ \vdash & [x:[a+b]][y:[a \Rightarrow c]][z:[b \Rightarrow c]]([y \mathbin{?} z]\,x) & : & [[a+b] \Rightarrow [[a \Rightarrow c] \Rightarrow [[b \Rightarrow c] \Rightarrow c]]] \end{array}$$

In the second example, in analogy to existential abstractions, a reference (x) to an interface ($[a+b]$) is hiding the information on which particular proposition (a or b) is proven.
In analogy to the β-equality for existential abstraction, projection is extended to sums.

$$[a+b].1 =_\lambda a \qquad [a+b].2 =_\lambda a$$

Finally, to support common mathematical reasoning practices, d introduces a classical negation operator $\neg a$ which has a neutral type rule *neg* and which defines an equivalence class of propositions w.r.t. classical negation. The central logical properties are double negation and De Morgan's laws:

$$a =_\lambda \neg\neg a \qquad [a,b] =_\lambda \neg[\neg a + \neg b] \qquad [x:a]b =_\lambda \neg[x!a]\neg b$$

Furthermore, negation has no effect on operators to which no elimination operator can be applied when they are used as types

$$\neg\tau =_\lambda \tau \qquad \neg[x \doteq a, b:c] =_\lambda [x \doteq a, b:c]$$

$$\neg[a, : b] =_\lambda [a, : b] \qquad \neg[: a, b] =_\lambda [: a, b] \qquad \neg[a \mathbin{?} b] =_\lambda [a \mathbin{?} b]$$

The negation laws define many negated formulas as equivalent which helps to eliminate many routine applications of logical equivalences in deductions. For example, the following laws can be derived for arbitrary well-typed expressions a, b.

$$\begin{array}{rrcl}
\vdash & [x : a]x & : & [a \Rightarrow a] \ =_\lambda \ [\neg\neg a \Rightarrow a] \ =_\lambda \ [a \Rightarrow \neg\neg a] \\
\vdash & [x : \neg[a, b]]x & : & [\neg[a, b] \Rightarrow [\neg a + \neg b]] \ =_\lambda \ [[\neg a + \neg b] \Rightarrow \neg[a, b]] \\
\vdash & [x : \neg[x!a]b]x & : & [\neg[x!a]b \Rightarrow [x : a]\neg b] \ =_\lambda \ [[x : a]\neg b \Rightarrow \neg[x!a]b]
\end{array}$$

Truth and falsehood can now be defined as follows:

$$\mathbf{ff} := [x : \tau]x \qquad \mathbf{tt} := \neg\mathbf{ff}$$

Note that falsehood is just a new convenient notation for the type role of the identity ($\mathbf{ff} = I$). The expected properties follow almost directly:

$$\begin{array}{rrcl}
\vdash & [x : \tau][y : \mathbf{ff}](y\,x) & : & [x : \tau][\mathbf{ff} \Rightarrow x] \\
\vdash & [x \doteq \tau, \tau : \neg x] & : & \mathbf{tt}
\end{array}$$

In the proof of $\mathbf{tt}$ note that $\mathbf{tt} =_\lambda [x!\tau]\neg x$ and $\tau : \tau =_\lambda \neg\tau =_\lambda (\neg x)[\tau/x]$.

Equivalence rules for negation obviously do not yield all logical properties of negation, *e.g.* they are not sufficient to prove $[a + \neg a]$. Therefore one has to assume additional axioms. Similarly, due to the limited strength of d, additional axioms must also be added for the mathematical structures whose deductions are to be formalized (for both see Section 3).

As this completes the overview of d, one may ask for its general advantages w.r.t. PTS. While essentially equivalent expressive means could probably also be defined in a more classical semantic setting, the purely functional approach of d can be considered as conceptually more simple. Independently from the semantic setting the use of common logical quantifiers and propositional connectors including a classical negation with rich equivalence laws seems more suitable for modelling mathematical deductions than encoded operators or operators with constructive interpretation.

The remainder of this paper is structured as follows: A formal definition of d is presented in Section 2 and several application examples are shown in Section 3. Readers with less focus on the theoretical results can well read Section 3 before Section 2. The main part of the paper is Section 4 containing proofs of confluence, subject reduction, uniqueness of types, strong normalization, and consistency.

2. Definition

First we summarize the syntax (which has been motivated in Section 1) and define some basic notions such as free occurrences of variables. We then define the congruence relation $a =_\lambda b$ as transitive and symmetric closure of a reduction relation $a \rightarrow^* b$. Finally we define contexts Γ and the type relation $\Gamma \vdash a : b$.

2.1. Basic definitions.

Definition 2.1 (Expressions). Let $\mathcal{V} = \{x, y, z, \ldots\}$ be an infinite set of variables. The set of expressions $\mathcal{E}$ is generated by the following rules

$$\begin{array}{rcl}
\mathcal{E} & ::= & \tau \mid \mathcal{V} \\
& & \mid [\mathcal{V} : \mathcal{E}]\mathcal{E} \mid (\mathcal{E}\,\mathcal{E}) \\
& & \mid [\mathcal{V}!\mathcal{E}]\mathcal{E} \mid [\mathcal{V} \doteq \mathcal{E}, \mathcal{E} : \mathcal{E}] \mid \mathcal{E}.1 \mid \mathcal{E}.2 \\
& & \mid [\mathcal{E}, \mathcal{E}] \mid [\mathcal{E} + \mathcal{E}] \mid [\mathcal{E}, :\mathcal{E}] \mid [:\mathcal{E}, \mathcal{E}] \mid [\mathcal{E} \mathbin{?} \mathcal{E}] \mid \neg\mathcal{E}
\end{array}$$

Expressions will be denoted by $a, b, c, d, \ldots$,

- τ is the *primitive constant*, $x, y, z, \ldots \in \mathcal{V}$ are *variables*,
- $[x : a]b$ is a *universal abstraction*, $(a\,b)$ is an *application*,
- $[x!a]b$ is an *existential abstraction*, $[x \doteq a, b : c]$ is a *protected definition*, $a.1$ and $a.2$ are *left-* and *right-projection*,
- $[a, b]$ is a *product*, $[a + b]$ is a *sum*, $[a, :b]$ and $[:a, b]$ are *left-* and *right-injection*, $[a \mathbin{?} b]$ is a *case distinction*, and $\neg a$ is a *negation.*

For the sake of succinctness and homogenity in the following definitions we are using some additional notations for groups of operations:

$$\begin{array}{rll} [a \oplus b] & \text{stands for} & [a, b] \text{ or } [a + b] \\ \oplus(a_1, \ldots, a_n) & \text{stands for} & \begin{cases} a_1.1, a_1.2, \text{or } \neg a_1 & \text{if } n = 1 \\ (a_1\,a_2), [a_1 \oplus a_2], [a_1, :a_2], [:a_1, a_2] \text{ or } [a_1 \mathbin{?} a_2] & \text{if } n = 2 \end{cases} \\ \oplus_x(a_1, \ldots, a_n) & \text{stands for} & \begin{cases} [x : a_1]a_2 \text{ or } [x!a_1]a_2 & \text{if } n = 2 \\ [x \doteq a_1, a_2 : a_3] & \text{if } n = 3 \end{cases} \end{array}$$

If one of these notations is used more than once in an equation or inference rule it always denotes the same concrete operation.

Definition 2.2 (Free variables, substitution)**.** Variables occurring in an expression which do not occur in the range of a binding *occur free* in the expression. $FV(a)$ which denotes the set of *free variables* of an expression a is defined as follows:

$$\begin{array}{rcl} FV(\tau) & = & \{\} \\ FV(x) & = & \{x\} \\ FV(\oplus(a_1, \ldots, a_n)) & = & FV(a_1) \cup \ldots \cup FV(a_n) \\ FV(\oplus_x(a_1, \ldots, a_n) & = & FV(a_1) \cup \ldots \cup FV(a_{n-1}) \cup (FV(a_n) \backslash \{x\}) \end{array}$$

Note that in a protected definition $[x \doteq a_1, a_2 : a_3]$ the binding of x is for a_3 only. The *substitution* $a[b/x]$ of all free occurrences of variable x in expression a by expression b is defined as follows:

$$\begin{array}{rcl} \tau[b/x] & = & \tau \\ y[b/x] & = & \begin{cases} b & \text{if } x = y \\ y & \text{otherwise} \end{cases} \\ \oplus(a_1, \ldots, a_n)[b/x] & = & \oplus(a_1[b/x], \ldots, a_n[b/x]) \\ \oplus_y(a_1, \ldots, a_n)[b/x] & = & \begin{cases} \oplus_y(a_1[b/x], \ldots, a_{n-1}[b/x], a_n) \\ \quad \text{if } x = y \\ \oplus_y(a_1[b/x], \ldots, a_n[b/x]) \\ \quad \text{otherwise} \end{cases} \end{array}$$

A substitution $a[b/x]$ may lead to name clashes in case a variable y occurring free in the inserted expression b comes into the range of a binding of y in the original expression. These name clashes can be avoided by renaming of variables.

Definition 2.3 (α-conversion, Implicit renaming, name-independent representations)**.** The renaming relation on bound variables is usually called α-conversion and induced by the following axiom (using the notation $=_\alpha$).

$$\frac{y \notin FV(a_n)}{\oplus_x(a_1, \ldots, a_{n-1}, a_n) \;=_\alpha\; \oplus_y(a_1, \ldots, a_{n-1}, a_n[y/x])}$$

In order not to clutter the presentation, we will write variables as strings but always assume appropriate renaming of bound variables in order to avoid name clashes. This assumption is justified because one could also use a less-readable but name-independent presentation of expressions using *e.g.* de Bruijn indexes [8] which would avoid the necessity of renaming all together.

2.2. **Reduction and congruence.**

The functional interpretation is defined by the reduction of an expression into a more basic expression. We will later show that reduction, if it terminates, always leads to a unique result.

Definition 2.4 (Single-step reduction)**.** *Single-step reduction* $a \to b$ is the smallest relation satisfying the axioms and inference rules of Table 4. All reduction axioms have been

(β_1)	$([x : a]b\ c)$	$\to$	$b[c/x]$	(β_2)	$([x!a]b\ c)$	$\to$	$b[c/x]$
(β_3)	$([a\,?\,b]\,[c, : d])$	$\to$	$(a\,c)$	(β_4)	$([a\,?\,b]\,[: c, d])$	$\to$	$(b\,d)$
(π_1)	$[x \doteq a, b : c].1$	$\to$	a	(π_2)	$[x \doteq a, b : c].2$	$\to$	b
(π_3)	$[a, b].1$	$\to$	a	(π_4)	$[a, b].2$	$\to$	b
(π_5)	$[a + b].1$	$\to$	a	(π_6)	$[a + b].2$	$\to$	b
(ν_1)	$\neg\neg a$	$\to$	a				
(ν_2)	$\neg[a, b]$	$\to$	$[\neg a + \neg b]$	(ν_3)	$\neg[a + b]$	$\to$	$[\neg a, \neg b]$
(ν_4)	$\neg[x : a]b$	$\to$	$[x!a]\neg b$	(ν_5)	$\neg[x!a]b$	$\to$	$[x : a]\neg b$
(ν_6)	$\neg\tau$	$\to$	τ	(ν_7)	$\neg[x \doteq a, b : c]$	$\to$	$[x \doteq a, b : c]$
(ν_8)	$\neg[a, : b]$	$\to$	$[a, : b]$	(ν_9)	$\neg[: a, b]$	$\to$	$[: a, b]$
(ν_{10})	$\neg[a\,?\,b]$	$\to$	$[a\,?\,b]$				

$$(\oplus\overbrace{(\_, \ldots, \_)}^{n}{}_i) \qquad \frac{a_i \to b_i}{\oplus(a_1, \ldots, a_i, \ldots, a_n) \to \oplus(a_1, \ldots, b_i, \ldots, a_n)}$$

$$(\oplus_x\overbrace{(\_, \ldots, \_)}^{n}{}_i) \qquad \frac{a_i \to b_i}{\oplus_x(a_1, \ldots, a_i, \ldots, a_n) \to \oplus_x(a_1, \ldots, b_i, \ldots, a_n)}$$

Table 4: Axioms and rules for single-step reduction.

motivated and explained in the introduction in form of equivalence laws.

Definition 2.5 (Reduction)**.** *Reduction* $a \to^* b$ of an expression a to b is defined as the reflexive and transitive closure of single-step reduction $a \to b$. The notation $a \to^n b$ where $n \geq 0$ is used to indicate n consecutive single-step reductions. We use the notation $a \to^* b \to^* c \ldots$ to indicate reduction sequences. If two expressions a and b reduce to a common expression we write $a \nabla b$. To show arguments about equality in arguments about reduction we use the notation $a_1 = \cdots = a_n \;\; \to^* \;\; b_1 = \cdots = b_m \;\; \to^* \;\; c_1 = \cdots = c_k \cdots$.

This will also be used for sequences of n-step reductions $\rightarrow^n$ and accordingly for sequences containing both notations.

Definition 2.6 (Congruence)**.** *Congruence* of expressions, denoted by $a =_\lambda b$, is defined as the symmetric and transitive closure of reduction. The notations for reduction sequences are extended to contain congruences as well.

The following simple examples illustrate reduction and congruence:

$$\begin{array}{rcl}
(([y_2 : a][y : (P\,y_2)](Q\,y_2)\,x.1)\,x.2) & \rightarrow_{(\beta_1)} & ([y : (P\,x.1)](Q\,x.1)\,x.2) \\
& \rightarrow_{(\beta_1)} & (Q\,x.1) \\
[[x \doteq a, b : c], d].1.2 & \rightarrow_{(\pi_3)} & [x \doteq a, b : c].2 \\
& \rightarrow_{(\pi_2)} & b \\
[x : \neg[y : \tau]\tau]\neg[a, b] & \rightarrow_{(\nu_4)} & [x : [y!\tau]\neg\tau]\neg[a, b] \\
& \rightarrow_{(\nu_6)} & [x : [y!\tau]\tau]\neg[a, b] \\
& \rightarrow_{(\nu_2)} & [x : [y!\tau]\tau][\neg a + \neg b] \\
[x : \neg[a + b]][\neg a, \neg b] & =_\lambda & [x : [\neg a, \neg b]]\neg[a + b]
\end{array}$$

2.3. **Typing and Validity.**

Definition 2.7 (Context)**.** A *context*, denoted by Γ, is a finite sequence of declarations $(x_1 : a_1, \ldots, x_n : a_n)$, where x_i are variables with $x_i \neq x_j$ and a_i are expressions. The assumption about name-free representation of bound variables justifies the uniqueness assumption. The lookup of a variable in a context $\Gamma(x)$ is a partial function defined by $\Gamma(x_i) = a_i$. $dom(\Gamma) = \{x_1, \ldots, x_n\}$ and $ran(\Gamma) = \{a_1, \ldots, a_n\}$ denote the domain and range of a context Γ. $\Gamma, x : a$ denotes the extension of Γ on the right by a binding $x : a$ where x is a variable not yet declared in Γ. Γ_1, Γ_2 denotes the concatenation of two contexts declaring disjoint variables. The empty context is written as $()$. $[\Gamma]a = [x_1 : a_1] \ldots [x_1 : a_n]a$ denotes the abstraction of a context over an expression.

Definition 2.8 (Typing)**.** Typing $\Gamma \vdash a : b$ of a to b, the *type* of a, under a context Γ is the smallest ternary relation on contexts and two expressions satisfying the inference rules of Table 5 all of which have been motivated and explained in Section 1. We often use the notation $\Gamma \vdash a_1 = \cdots = a_n \ : \ b_1 = \cdots = b_m$ to indicate arguments about equality of expressions in proofs about the type relation. Sometimes we also mix the use of $=_\lambda$ and $=$ in this notation.

As a simple example consider the introduction and elimination laws for existential quantification (see Section 1). The type determination of the first law can be seen as follows: With $\Gamma_1 = (P : [y : a]b, x : a, z : (P\,x)))$ where $y \notin FV(b)$ and since $(P\,y)[x/y] = P(\,x)$ by the rules *start*, *weak*, and *appl* we obtain

$$\Gamma_1 \vdash\ x : a \qquad \Gamma_1 \vdash\ z : (P\,y)[x/y] \qquad \Gamma_1 \vdash\ (P\,y) : b$$

which due to the rule *def* implies that

$$\Gamma_1 \quad \vdash \quad [y \doteq x, z : (P\,y)] : [y!a](P\,y)$$

$$(ax)\ \vdash \tau : \tau \qquad (start)\ \frac{\Gamma\vdash a : b}{\Gamma, x : a \vdash x : a}$$

$$(weak)\ \frac{\Gamma\vdash a : b \quad \Gamma\vdash c : d}{\Gamma, x : c\vdash a : b} \qquad (conv)\ \frac{\Gamma\vdash a : b \quad b =_\lambda c \quad \Gamma\vdash c : d}{\Gamma\vdash a : c}$$

$$(abs_U)\ \frac{\Gamma, x : a\vdash b : c}{\Gamma\vdash [x : a]b : [x : a]c} \qquad (abs_E)\ \frac{\Gamma, x : a\vdash b : c}{\Gamma\vdash [x!a]b : [x : a]c}$$

$$(appl)\ \frac{\Gamma\vdash a : [x : c]d \quad \Gamma\vdash b : c}{\Gamma\vdash (a\, b) : d[b/x]}$$

$$(def)\ \frac{\Gamma\vdash a : b \quad \Gamma\vdash c : d[a/x] \quad \Gamma, x : b\vdash d : e}{\Gamma\vdash [x \doteq a, c : d] : [x!b]d}$$

$$(ch_I)\ \frac{\Gamma\vdash a : [x!b]c}{\Gamma\vdash a.1 : b} \qquad (ch_B)\ \frac{\Gamma\vdash a : [x!b]c}{\Gamma\vdash a.2 : c[a.1/x]}$$

$$(prd)\ \frac{\Gamma\vdash a : c \quad \Gamma\vdash b : d}{\Gamma\vdash [a, b] : [c, d]} \qquad (sum)\ \frac{\Gamma\vdash a : c \quad \Gamma\vdash b : d}{\Gamma\vdash [a + b] : [c, d]}$$

$$(pr_L)\ \frac{\Gamma\vdash a : [b, c]}{\Gamma\vdash a.1 : b} \qquad (pr_R)\ \frac{\Gamma\vdash a : [b, c]}{\Gamma\vdash a.2 : c}$$

$$(inj_L)\ \frac{\Gamma\vdash a : b \quad \Gamma\vdash c : d}{\Gamma\vdash [a, : c] : [b + c]} \qquad (inj_R)\ \frac{\Gamma\vdash a : b \quad \Gamma\vdash c : d}{\Gamma\vdash [: c, a] : [c + b]}$$

$$(case)\ \frac{\Gamma\vdash a : [x : c_1]d \quad \Gamma\vdash b : [y : c_2]d \quad \Gamma\vdash d : e}{\Gamma\vdash [a\,?\,b] : [z : [c_1 + c_2]]d}$$

$$(neg)\ \frac{\Gamma\vdash a : b}{\Gamma\vdash \neg a : b}$$

Table 5: Axiom and rules for typing.

The type relation in Section 1 follows from rule abs_U. The type determination of the second law can be seen as follows, let

$$\Gamma_2 \quad = \quad (P : [y : a]b, Q : [y : a]b, x : [y_1!a](P\, y_1), z : [y_2 : a][y : (P\, y_2)](Q\, y_2))$$

By rules *start*, *weak*, and *appl* we obtain:

$$\Gamma_2 \vdash\ (z\, x.1) : [u : (P\, x.1)](Q\, x.1)$$

From this, by the same rules we obtain:

$$\Gamma_2 \vdash\ ((z\, x.1)\, x.2) : (Q\, x.1) = (Q\, y_3)[x.1/y_3]$$

Similarly to the first law, by rule *def* this implies

$$\Gamma_2 \quad \vdash \quad [y_3 \doteq x.1, ((z\, x.1)\, x.2) : (Q\, y_3)] \quad : \quad [y_3!a](Q\, y_3)$$

Again, the remainder follows from rule abs_U.

Definition 2.9 (Validity)**.** Validity $\Gamma\vdash a$ of an expression a under a context Γ is defined as the existence of a type:

$$\Gamma \vdash\ a \qquad \text{if and only if there is an expression } b \text{ such that } \ \Gamma\vdash a : b$$

Similarly to typing we use the notation $\Gamma_1 = \cdots = \Gamma_n \vdash a_1 = \cdots = a_n$ to indicate arguments about equality of contexts and expressions in proofs about validity. Similarly for congruence $=_\lambda$ or combinations of both. We also use the notation $\Gamma \vdash a_1, \ldots, a_n$ as an abbreviation for $\Gamma \vdash a_1$, …, $\Gamma \vdash a_n$. As for typing, we also omit writing the empty context.

3. Examples

The purpose of this section is to illustrate the basic style of axiomatizing theories and describing deductions when using `d`. Note that the example are presented with fully explicit expressions of `d`, i.e. we do not omit any subexpressions which could be inferred from other parts using pattern matching or proof tactics and we do not use a module concept for theories. Such features should of course be part of any practically useful approach for formal deductions on the basis of `d`.

Remark 3.1 (Notational conventions)**.** For convenience, in the examples we write $[x_1 : a_1] \ldots [x_n : a_n]a$ as $[x_1 : a_1; \ldots ; x_n : a_n]a$ and $[x_1 : a] \ldots [x_n : a]b$ as $[x_1, \ldots, x_n : a]b$ and similar for existential abstractions. We also use combinations of these abbreviations such that *e.g.* $[x : a; y_1, y_2!b]c$ is shorthand for $[x : a][y_1!b][y_2!b]c$. We write $[a \Rightarrow b]$ to denote $[x : a]b$ if $x \notin FV(b)$. Similarly, we write $[a_1 \Rightarrow \ldots [a_n \Rightarrow a] \ldots]$ as $[a_1; \ldots ; a_n \Rightarrow a]$. We write nested applications $(\ldots((a\,a_1)\,a_2) \ldots\, a_n)$ as $(a\,a_1 \ldots a_n)$. We also write $\Gamma \vdash a : b$ as $a : b$ if Γ is empty or has been made clear from the context. When writing $a : b$ (*name*) we introduce *name* as abbreviation for a of type b.

3.1. Additional axioms for negation.

In Section 1 we explained the necessity of additional negation axioms. Note that, without additional axioms, even constructive laws of negation are missing, as for example, the contrapositive laws is not valid. In principle many different axioms schemes are possible. For example, one could be inspired by the rules for negation in sequent calculus ($\underline{A}$ stands for a sequence of formulas A_1, …, A_n):

$$\frac{\underline{C} \vdash A, \underline{B}}{\underline{C}, \neg A \vdash \underline{B}}\ (L\neg) \qquad \frac{\underline{C}, A \vdash \underline{B}}{\underline{C} \vdash \neg A, \underline{B}}\ (R\neg)$$

Theses rule together with the negations axioms inspire the following axiom schemes indexed over expressions a and b:

$$\neg^+_{a,b} : [[a + b] \Rightarrow [\neg a \Rightarrow b]] \qquad \neg^-_{a,b} : [[a \Rightarrow b] \Rightarrow [\neg a + b]] \qquad (*)$$

where $\neg^+_{a,b}, \neg^-_{a,b} \in \mathcal{V}$ are from an infinite set of variables I_{Ax}. Formally, typing an expression c to an expression d under the above axiom schemes could be defined so as to require that $FV(d) = \emptyset$ and that there is a context Γ consisting of (a finite sequence of) declarations of variables from I_{Ax} such that $\Gamma \vdash c : d$.

Assuming these axiom schemes we can now show the law of the excluded middle:

$$(\neg^-_{\neg a, \neg a}\ [x : \neg a]x) \quad : \quad [a + \neg a]$$

Note that $\neg^-_{a,b}$ and $\neg^+_{\neg a,b}$ state that universal abstractions without dependencies (i.e. $[a \Rightarrow b]$) are logically equivalent to the classical definition of implications:

$$\neg^-_{a,b} : [[a \Rightarrow b] \Rightarrow [\neg a + b]] \qquad \neg^+_{\neg a,b} : [[\neg a + b] \Rightarrow [a \Rightarrow b]]$$

On the basis of this equivalence it is easy to show the remaining laws of negation.

3.2. **Equality.** Basic axioms about an equality congruence relation on expressions of equal type can be formalized as context *Equality*:

$$\begin{array}{rcl}
\mathit{Equality} := (\ (.) =_{(.)} (.) & : & [S : \tau][S; S \Rightarrow \tau] \\
E_1 & : & [S : \tau; x : S] x =_S x \\
E_2 & : & [S : \tau; x, y : S][x =_S y \ \Rightarrow\ y =_S x] \\
E_3 & : & [S : \tau; x, y, z : S][x =_S y; y =_S z \ \Rightarrow\ x =_S z] \\
E_4 & : & [S_1, S_2 : \tau; x, y : S_1; F : [S_1 \Rightarrow S_2]][x =_{S_1} y \ \Rightarrow\ (F\,x) =_{S_2} (F\,y)])
\end{array}$$

Here S is an variable used to abstract over the type of the expressions to be equal. Note that, for better readability, we use the infix notation $x =_S y$, introduced by a declaration $(.) =_{(.)} (.) : [S : \tau][S; S \Rightarrow S]$. Equivalently we could have written $((.) =_{(.)} (.)\, S\, x\, y)$.

3.3. **Natural Numbers.** Assuming the context *Equality*, well-known axioms about naturals numbers including an induction principle can be formalized as context *Naturals*:

$$\begin{array}{rrcl}
\mathit{Naturals} := & (\quad N & : & \tau \\
& 0 & : & N \\
& s & : & [N \Rightarrow N] \\
& (.)+(.), (.)*(.) & : & [N; N \Rightarrow N] \\
& S_1 & : & [n : N] \neg((s\,n) =_N 0) \\
& S_2 & : & [n, m : N][(s\,n) =_N (s\,m) \ \Rightarrow\ n =_N m] \\
& A_1 & : & [n : N] 0 + n =_N n \\
& A_2 & : & [n, m : N](s\,n) + m =_N (s\,(n + m)) \\
& M_1 & : & [n : N] 0 * n =_N 0 \\
& M_2 & : & [n, m : N](s\,n) * m =_N m + (n * m) \\
& \mathit{ind} & : & [P : [N \Rightarrow \tau]][(P\,0); [n : N][(P\,n) \Rightarrow (P\,(s\,n))] \Rightarrow [n : N](P\,n)] \quad)
\end{array}$$

Note that, for better readability, we use infix notations $n + m$ and $n * m$ abbreviating the expressions $(+\, n\, m)$ and $(*\, n\, m)$. Note that the induction principle ranges over propositions of type τ only and is thus weaker than the common one (see also Section 3.4).

As an example, a simple property can be (tediously) deduced under the context $(\mathit{Equality}, \mathit{Naturals}, n : N)$ where $1 := (s\,0)$:

$$\begin{array}{l}
(E_3\ N\ (1{+}n)\ (s\,(0 + n))\ (s\,n) \\
\quad (A_2\,0\,n)\ (E_4\ N\ N\ (0{+}n)\ n\ [k : N](s\,k)\ (A_1\,n)))\ :\ 1 + n =_N (s\,n)
\end{array}$$

We give two examples of predicates defined on natural numbers as follows (where $2 := (s\,1)$):

$$\begin{array}{rcl}
(.) \geq (.) & := & [n, m : N; k!N] n =_N m + k \\
\mathit{even} & := & [n : N; m!N] n = 2 * m
\end{array}$$

The property $[n : N][(\mathit{even}\,n) \Rightarrow (\mathit{even}\,(2 + n))]$ about even numbers can be deduced on the basis of the following typing

$$n{:}N, x{:}(\mathit{even}\,n) \vdash\ [m \doteq 1 + x.1, (\mathit{law}\,n\,x.1\,x.2) : 2 + n =_N 2 * m]\ :\ (\mathit{even}\,(2 + n))$$

Here *law* is an abbreviation for a proof of the following property

$$\mathit{law}\ \ :\ \ [n, m : N][n =_N 2{*}m \Rightarrow\ 2{+}n =_N 2{*}(1{+}m)]$$

A definition of *law* can be derived from the axioms in a style similar (and even more tedious) to the above deduction. This deduction of the property about even numbers is correct since

$$\begin{array}{rcl}(\mathit{law}\, n\, x.1\, x.2) & : & 2 + n =_N 2 * (1 + x.1) \\ & =_\lambda & (2 + n =_N 2 * m)[1{+}x.1/m]\end{array}$$

3.4. **Casting types to the primitive constant.** In many cases, the constraint of universally quantified variables to be of primitive type, i.e. $x : \tau$, can be relaxed to arbitrary well typed expressions $a : b$ using an operation to cast arbitrary types to τ. For this reason we introduce an axiom scheme for a τ-casting function $()_a$[6] and we also assume the axioms schemes $()^+_a$ and $()^-_a$ essentially stating equivalence between casted and uncasted types:

$$\begin{array}{rcl}()_a & : & [a \Rightarrow \tau] \\ ()^+_a & : & [x : a][x \Rightarrow (()_a\, x)] \\ ()^-_a & : & [x : a][(()_a\, x) \Rightarrow x]\end{array}$$

As an example, we can generalize the property of $\mathbf{ff}$ in Section 1 to arbitrary well-typed a:

$$x : a \vdash [y : \mathbf{ff}]((()^-_a\; x\; (y\; (()_a\, x))) : [\mathbf{ff} \Rightarrow x]$$

As an another example, assuming one already has axiomatized the more general theory of integers, *e.g.* with type *Int*, a constant 0_I, and the relation $\geq$, one could instantiate the declaration of N of the context *Naturals* using casting as follows:

$$(N \;:=\; (()_{[\mathit{Int} \Rightarrow \tau]}\, [n!\mathit{Int}](n \geq 0_I)),\;\; 0 \;:= \ldots)$$

3.5. **Formalizing partial functions.** In the previous section we have used the application operator in `d` to model the application of total functions. As an example of a partial function consider the predecessor function on natural numbers. To introduce this function in `d`, several approaches come to mind:

- The predecessor function can be axiomatized as a total function over the type N.

$$\begin{array}{rcl}p & : & [N \Rightarrow N] \\ P & : & [n : N](p\,(s\, n)) =_N n\end{array}$$

  The (potential) problem of this approach is the interpretation of $(p\, 0)$ which may lead to unintuitive or harmful consequences. Furthermore, if additional axioms are to be avoided, the declaration of p must eventually by instantiated by some (total) function which defines a value for 0.
- The predecessor function can be defined by separately formalizing the condition.

$$\begin{array}{rcl}\mathit{nonZero} & := & [i : N; j!N]i =_N (s\, j) \\ p & := & [n : N; q : (\mathit{nonZero}\; n)]q.1\end{array}$$

  While mathematically clean, this definition requires to always provide an additional argument instantiating q when using the predecessor function.

[6]For a discussion of axioms schemes see also Section 3.1.

- As a variant of the previous approach, the additional argument can be hidden into an adapted type of the predecessor function.

$$\begin{array}{rcl} N^{>0} & := & [i, j!N] i =_N (s\, j) \\ p & := & [n : N^{>0}] n.2.1 \end{array}$$

  While mathematically clean, this approach requires a more complex handling of the argument $n : N^{>0}$ when using the predecessor function. For example, in $N^{>0}$ one can define the number $\mathbf{1}$ and apply the predecessor as follows:

$$\mathbf{1} \;\; := \;\; [i \doteq (s\,0), [j \doteq 0, (E_1\, N\, (s\,0)) : (s\,0) =_N (s\,j)] : [j!N] i =_N (s\,j)] : N^{>0}$$

As a consequence $p(\mathbf{1}) =_\lambda \mathbf{1}.2.1 =_\lambda 0$. Which of these (or other) approaches is best to use seems to depend on the organization and the goals of the formalization at hand.

3.6. **Defining functions from deductions.** Note that while the predecessor function can be directly defined, more complex functions and their (algorithmic) properties can be derived from the proofs of properties. As a sketch of an example consider the following well-known property

$$GCD \;\; := \;\; [x, y : N; k!N](gcd\ k\ x\ y)$$

where $(gcd\ k\ x\ y)$ denotes the property that k is the greatest common divisor of x and y. Given a (not necessarily constructive) deduction P_{GCD} of type GCD, one can then define the greatest common divisor $x \downarrow y$ and define deductions d_1 and d_2 proving well-known algorithmic properties.

$$\begin{array}{rcl} (.) \downarrow (.) & := & [x, y : N](P_{GCD}\ x\ y).1 \\ d_1 & : & [x, y : N](x + y) \downarrow x =_N y \downarrow x \\ d_2 & : & [x, y : N] x \downarrow (x + y) =_N x \downarrow y \end{array}$$

3.7. **Sets.** When formalizing mathematical deductions, besides natural numbers and equality one needs formal systems for many more basic structures of mathematics. For example, sets can be axiomatized by the following context using a formalized set comprehension principle.

$$\begin{array}{rrcl} Sets := & (\quad \mathbb{P} & : & [\tau \Rightarrow \tau], \\ & (.) \in_{(.)} (.) & : & [S : \tau][S; (\mathbb{P}\, S) \Rightarrow \tau], \\ & \{(.)\}_{(.)} & : & [S : \tau][[S \Rightarrow \tau] \Rightarrow (\mathbb{P}\, S)], \\ & I & : & [S : \tau; x : S; P : [S \Rightarrow \tau]][(P\, x) \Rightarrow x \in_S \{[y : S](P\, y)\}_S], \\ & O & : & [S : \tau; x : S; P : [S \Rightarrow \tau]][x \in_S \{[y : S](P\, y)\}_S \Rightarrow (P\, x)] \quad) \end{array}$$

Note that, for better readability, we use the notation $x \in_S y$ and $\{P\}_S$ abbreviating the expressions $((.) \in_{(.)} (.)\ S\, x\, y)$ and $(\{(.)\}_{(.)}\ S\, P)$. One can now define sets and set operators using set comprehension. Note the use of the τ-casting function to ensure the set-defining properties are of type τ.

$$\begin{array}{rcl}
\emptyset & := & [S:\tau]\{[x:S](()_{[\tau\Rightarrow\tau]}\,\mathbf{ff})\}_S \quad : \quad [S:\tau](\mathbb{P}\,S) \\
(.)\cup_{(.)}(.) & := & [S:\tau, A, B:(\mathbb{P}\,S)]\{[x:S](()_{[\tau,\tau]}\,[x\in_S A + x\in_S B])\}_S \\
 & & : \quad [S:\tau][(\mathbb{P}\,S);(\mathbb{P}\,S)\Rightarrow(\mathbb{P}\,S)] \\
\mathit{Even} & := & \{[x:N](()_{[N\Rightarrow\tau]}\,(\mathit{even}\ x))\}_N \quad : \quad (\mathbb{P}\,N)
\end{array}$$

Properties of individual elements can be deduced on the basis of the axiom O, for example we can extract the property $P := [n!N](x =_N 2 * n)$ of a member x of Even using the cast-removal axiom as follows (where $\mathit{Even}_P := [x:N](()_{[N\Rightarrow\tau]}\,(\mathit{even}\ x))$):

$$x:N, \mathit{asm}: x\in_N \mathit{Even} \ \vdash\ (()^{-}_{[N\Rightarrow\tau]}\,P\,(O\ N\ x\ \mathit{Even}_P\ \mathit{asm}))\ :\ P$$

Note that in this formalization of sets the axiom of choice can be immediately derived as follows:

$$\begin{array}{rcl}
\Gamma & = & (X, I:\tau, A:[I\Rightarrow(\mathbb{P}\,X)], u:[x:I;y!X]y\in_X (A\,x)) \\
\Gamma & \vdash & [F \doteq [i:I](u\,i).1\,, [i:I](u\,i).2 : ([i:I](F\,i)\in_X (A\,i))] \\
 & : & [F![I\Rightarrow X]; i:I](F\,i)\in_X (A\,i)
\end{array}$$

Note also that an alternative definition of naturals from integers can be given with sets as follows:

$$(N\ :=\ (()_{(\mathbb{P}\,\mathit{Int})}\,\{[n:\mathit{Int}](()_\tau\,(n\geq 0_I))\}_{\mathit{Int}}),\ \ 0\ :=\ \ldots)$$

3.8. **Proof structuring.** To illustrate some proof structuring issues, we formalize the property of being a group as follows (writing $[a_1, a_2, \ldots]$ for $[a_1, [a_2, \ldots]]$):

$$\begin{array}{rll}
\mathit{Group} := [S:\tau;(.)*(.)![S;S\Rightarrow S];e!S] & [& [x,y,z:S](x*y)*z =_S x*(y*z) \\
 & , & [x:S]e*x =_S x \\
 & , & [x:S;x'!S]x'*x =_S e \quad]
\end{array}$$

As an example, assume *Integers* as the context *Naturals* (see Section 3.3) extended with a subtraction operator $a-b$ with corresponding axioms. First, we can show that $+$ and 0 form a group: Obviously, one can construct a deduction *ded* with

$$\mathit{Equality}, \mathit{Integers} \vdash\ \mathit{ded} : P_g$$

where P_g describes the group laws:

$$\begin{array}{rll}
P_g := & [& [x,y,z:N]\,(x+y)+z =_N x+(y+z) \\
 & , & [x:N]\,0+x =_N x \\
 & , & [x:N, x'!N]\,x'+x =_N 0 \quad]
\end{array}$$

ded can be turned into a proof of $(\mathit{Group}\ N)$ as follows:

$$\begin{array}{rcl}
\mathit{isGroup} & := & [*\doteq+, [e\doteq 0, \mathit{ded}: P_g[e/0]] : P_g[e/0][*/+]] \\
\mathit{Equality}, \mathit{Integers} & \vdash & \mathit{isGroup} : (\mathit{Group}\ N)
\end{array}$$

It is well-known that the left-neutral element is also right-neutral, this means, when assuming g to be a group over S there is a proof p such that

$$\mathit{Equality}, S : \tau, g : (\mathit{Group}\ S) \vdash\ p : [x : S](g.1\ x\ g.2.1) =_S x$$

Here $g.1$ is the function of g and $g.2.1$ is the neutral element of g. Note that the use of existential declarations is supporting the proof structuring as it hides $*$ and e in the assumptions inside the $g : (\mathit{Group}\, S)$ assumption. On the other hand, one has to explicitly access the operators using projections.

p can be abstracted to an inference $p' := [S : \tau; g : (\mathit{Group}\, S)]p$ about a derived property of groups as follows:

$$\mathit{Equality} \quad \vdash \quad p' : [S : \tau; g : (\mathit{Group}\ S)][x : S]\ (g.1\ x\ g.2.1) =_S x$$

Hence we can instantiate p' to obtain the right-neutrality property of 0 for integers.

$$\mathit{Equality}, \mathit{Integers} \vdash\ (p'\ N\ \mathit{isGroup}) : [x : N]\, x + 0 =_N x$$

4. Properties

In this Section we show confluence of reduction, several properties of typing, including uniqueness of types, and strong normalisation of valid, i.e. typable, expressions. On the basis of these properties we show consistency of `d` in the sense that no expression is typing to $[x : \tau]x$ under the empty context. We show the main proof ideas and indicate important steps in detail, significantly more detail can be found in [32].

Remark 4.1 (Inductive principles)**.** Besides structural induction on the definition of $\mathcal{E}$, we frequently show properties about reduction relations by induction on the definition of single-step reduction. Similarly for properties about typing.

Remark 4.2 (Renaming of variables)**.** Typically when we prove a property using some axiom, inference rule, or derived property, we just mention the identifier or this axiom, rule, or property and then use it with an instantiation *renaming its variables so as to avoid name clashes with the proposition to be shown.* In order not to clutter the presentation, these renamings are usually not explicitly indicated.

Remark 4.3 (Introduction of auxiliary identifiers)**.** We usually introduce explicitly all auxiliary identifiers appearing in deduction steps. However, there are two important exceptions.

- When using structural induction, if we consider a specific operator and decompose an expression a *e.g.* by $a = \oplus(a_1, , \dots, a_n)$ we usually introduce implicitly the new auxiliary identifiers a_i.
- When using induction on the definition of reduction, if we consider a specific axiom or structural rule which requires a syntactic pattern we usually introduce implicitly the new auxiliary identifiers necessary for this pattern.

We begin with some basic properties of substitution and its relation to reduction.

Lemma 4.4 (Basic properties of substitution)**.** *For all a, b, c and x, y:*

(1) *If $x \notin FV(a)$ then $a[b/x] = a$.*
(2) *If $x \neq y$ and $x \notin FV(c)$ then $a[b/x][c/y] = a[c/y][b[c/y]/x]$.*
(3) *If $x \neq y$, $x \notin FV(c), y \notin FV(b)$ then $a[b/x][c/y] = a[c/y][b/x]$.*

Proof. Proof is straightforward by structural induction on a. □

Lemma 4.5 (Substitution and reduction)**.** *For all a, b, c and x: $a \to^* b$ implies $a[c/x] \to^* b[c/x]$, $c[a/x] \to^* c[b/x]$, and $FV(b) \subseteq FV(a)$.*

Proof. Proof is straightforward by induction on the definition of single-step reduction. □

Next we turn to basic decomposition properties of reduction.

Lemma 4.6 (Reduction decomposition)**.** *For all $a_1, \ldots, a_n, b, b_1, \ldots, b_n$ and x:*

(1) *$\oplus(a_1, a_2) \to^* b$, where $\oplus(a_1, a_2) \neq (a_1\, a_2)$ implies $b = \oplus(b_1, b_2)$ where $a_1 \to^* b_1$ and $a_2 \to^* b_2$.*
(2) *$\oplus_x(a_1, \ldots, a_n) \to^* b$ implies $b = \oplus_x(b_1, \ldots, b_n)$, $a_i \to^* b_i$, for $1 \leq i \leq n$.*
(3) *$a.i \to^* \oplus_x(b_1, \ldots, b_n)$, $i = 1, 2$, implies $a \to^* [c_1 \oplus c_2]$ or $a \to^* [y \doteq c_1, c_2 : c_3]$, for some c_1, c_2, c_3 with $c_i \to^* \oplus_x(b_1, \ldots, b_n)$.*
(4) *$a.i \to^* \oplus(a_1, a_2)$, $i = 1, 2$, where $\oplus(a_1, a_2)$ is a sum, product or injection implies $a \to^* [c_1 \oplus' c_2]$ or $a \to^* [y \doteq c_1, c_2 : c_3]$, for some c_1, c_2, c_3 with $c_i \to^* \oplus(a_1, a_2)$.*
(5) *$a_1(a_2) \to^* \oplus_x(b_1, \ldots, b_n)$ implies, for some c_1, c_2, c_3, c_4, one of the following cases*
   (a) *$a_1 \to^* \oplus'_y(c_1, c_2)$, $a_2 \to^* c_3$ and $c_2[c_3/y] \to^* \oplus_x(b_1, \ldots, b_n)$, or*
   (b) *$a_1 \to^* [c_1 \,?\, c_2]$ and either $a_2 \to^* [c_3, : c_4]$ and $(c_1\, c_3) \to^* \oplus_x(b_1, \ldots, b_n)$ or $a_2 \to^* [: c_3, c_4]$ and $(c_2\, c_4) \to^* \oplus_x(b_1, \ldots, b_n)$.*
(6) *$a_1(a_2) \to^* \oplus(b_1, \ldots, b_n)$ where $\oplus(b_1, \ldots, b_n) \neq (b_1\, b_2)$ implies, for some c_1, c_2, c_3, and c_4, one of the following cases*
   (a) *$a_1 \to^* \oplus'_y(c_1, c_2)$, $a_2 \to^* c_3$, and $c_2[c_3/x] \to^* \oplus(b_1, \ldots, b_n)$, or*
   (b) *$a_1 \to^* [c_1 \,?\, c_2]$ and either $a_2 \to^* [c_3, : c_4]$ and $(c_1\, c_3) \to^* \oplus(b_1, \ldots, b_n)$ or $a_2 \to^* [: c_4, c_3]$ and $(c_2\, c_3) \to^* \oplus(b_1, \ldots, b_n)$.*

Proof. Straightforward inductions, as these properties are very close to the definition of the reduction relation. □

Lemma 4.7 (Reduction decomposition for negation)**.** *For all $a_1, \ldots, a_n, b, b_1, \ldots, b_n$ and x:*

(1) *$\neg a \to^* [x : b_1]b_2$ implies $a \to^* [x!c_1]c_2$ for some c_1, c_2 where $c_1 \to^* b_1$ and $\neg c_2 \to^* b_2$.*
(2) *$\neg a \to^* [b_1, b_2]$ implies $a \to^* [c_1 + c_2]$ for some c_1, c_2 where $\neg c_1 \to^* b_1$ and $\neg c_2 \to^* b_2$.*
(3) *$\neg a \to^* [x!b_1]b_2$ implies $a \to^* [x : c_1]c_2$ for some c_1, c_2 where $c_1 \to^* b_1$ and $\neg c_2 \to^* b_2$.*
(4) *$\neg a \to^* [b_1 + b_2]$ implies $a \to^* [c_1, c_2]$ for some c_1, c_2 where $\neg c_1 \to^* b_1$ and $\neg c_2 \to^* b_2$.*
(5) *$\neg a \to^* [x \doteq b_1, b_2 : b_3]$ implies $a \to^* [x \doteq b_1, b_2 : b_3]$.*
(6) *$\neg a \to^* [b_1 \,?\, b_2]$ implies $a \to^* [b_1 \,?\, b_2]$.*
(7) *$\neg a \to^* [b_1, : b_2]$ implies $a \to^* [b_1, : b_2]$.*
(8) *$\neg a \to^* [: b_1, b_2]$ implies $a \to^* [: b_1, b_2]$.*

Proof. Straightforward inductions, as these properties are very close to the definition of the reduction relation. □

4.1. Confluence of $\to$.

Classical confluence proofs for untyped λ-calculus could be used, *e.g.* [4] or [33] (using parallel reduction) could be adapted to include the operators of d. Due to the significant number of reduction axioms of d, we use an alternative approach using explicit substitutions and an auxiliary relation of *reduction with explicit substitution* which has detailed substitution steps on the basis of a definitional environment (this approach was basically already adopted in the Automath project [9]) and which comprises sequences of negation-related reduction-steps into single steps.

The underlying idea is that reduction with explicit substitution can be shown to be directly confluent which implies its confluence. We then show that this implies confluence of $\to$. There are several approaches to reduction with explicit substitutions, *e.g.* [1, 2]. Furthermore, there is a significant amount of more recent work in this context, however, as explicit substitution is not the main focus of this article we do not give an overview here. However, we should note that, in general, confluence of calculi with explicit substitutions is proved by using confluence of the underlying calculus without explicit substitutions. Here it is the other way around: confluence of reduction with explicit substitution is used to show confluence of $\to$.

The approach introduced below introduces a definitional environment as part of the reduction relation to explicitly unfold single substitution instances and then discard substitution expressions when all instances are unfolded. This approach, as far as basic lambda calculus operators are concerned, is essentially equivalent to the system Λ_{sub} which has been defined using substitution [23] or placeholders [20] to indicate particular occurrences to be substituted. Both approaches slightly differ from ours as they duplicate the substituted expression on the right-hand side of the β-rule thus violating direct confluence.

We begin by defining expressions with substitution.

Definition 4.8 (Expressions with substitution)**.** The set $\dot{\mathcal{E}}$ of *expressions with substitution* is an extension of the set $\mathcal{E}$ of expressions adding a substitution operator.

$$\dot{\mathcal{E}} \ ::= \ \underbrace{\{\tau\} \mid \ldots \mid \neg\dot{\mathcal{E}}}_{\text{(see Definition 2.1)}} \mid [\mathcal{V}{:=}\dot{\mathcal{E}}]\dot{\mathcal{E}}$$

Expressions with substitution will be denoted by $\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \ldots$. $[x{:=}\mathbf{a}]\mathbf{b}$ is an *internalized substitution.* As indicated by its name, the purpose of $[x := \mathbf{a}]\mathbf{b}$ is to internalize the substitution function.

The function computing free variables (Definition 2.2) is extended so as to treat internalized substitutions identical to abstractions.

$$FV([x{:=}\mathbf{a}]\mathbf{b}) \ = \ FV(\mathbf{a}) \cup (FV(\mathbf{b})\backslash\{x\})$$

The substitution function (Definition 2.2) is extended analogously.

$$([z{:=}\mathbf{a}]\mathbf{b})[y/x] \ = \ \begin{cases} [z{:=}\mathbf{a}[y/x]]\mathbf{b} & \text{if } z = x, \\ [z{:=}\mathbf{a}[y/x]](\mathbf{b}[y/x]) & \text{otherwise} \end{cases}$$

Similar for α-conversion (Definition 2.3).

$$\frac{y \notin FV(\mathbf{b})}{[x{:=}\mathbf{a}]\mathbf{b} \ =_\alpha \ [y{:=}\mathbf{a}]\mathbf{b}[y/x]}$$

As for expressions we will write variables as strings but always assume appropriate renaming of bound variables in order to avoid name clashes. In order to define reduction with explicit substitution as a directly confluent relation, we first define the auxiliary notion of *negation-reduction* $\mathbf{a} \to_\neg^* \mathbf{b}$ which comprises application sequences of axioms $\nu_1, \ldots, \nu_5$ in a restricted context.

Definition 4.9 (Negation-reduction)**.** Single-step *negation-reduction* $\mathbf{a} \to_\neg \mathbf{b}$ is the smallest relation on expressions with explicit substitution satisfying the axiom and the inference rules of Table 6. n-step negation-reduction $\mathbf{a} \to_\neg^n \mathbf{b}$ $(n \geq 0)$ and negation-reduction $\mathbf{a} \to_\neg^* \mathbf{b}$ are

$$
\begin{array}{llll}
(\nu_1) & \neg\neg\mathbf{a} \to_\neg \mathbf{a} & & \\
(\nu_2) & \neg[\mathbf{a},\mathbf{b}] \to_\neg [\neg\mathbf{a}+\neg\mathbf{b}] & (\nu_3) & \neg[\mathbf{a}+\mathbf{b}] \to_\neg [\neg\mathbf{a},\neg\mathbf{b}] \\
(\nu_4) & \neg[x:\mathbf{a}]\mathbf{b} \to_\neg [x!\mathbf{a}]\neg\mathbf{b} & (\nu_5) & \neg[x!\mathbf{a}]\mathbf{b} \to_\neg [x:\mathbf{a}]\neg\mathbf{b} \\
([\_\oplus\_]_1) & \dfrac{\mathbf{a}_1 \to_\neg \mathbf{a}_2}{[\mathbf{a}_1\oplus\mathbf{b}] \to_\neg [\mathbf{a}_2\oplus\mathbf{b}]} & ([\_\oplus\_]_2) & \dfrac{\mathbf{b}_1 \to_\neg \mathbf{b}_2}{[\mathbf{a}\oplus\mathbf{b}_1] \to_\neg [\mathbf{a}\oplus\mathbf{b}_2]} \\
(\oplus_\_(\_,\_)_2) & \dfrac{\mathbf{b}_1 \to_\neg \mathbf{b}_2}{\oplus_x(\mathbf{a},\mathbf{b}_1) \to_\neg \oplus_x(\mathbf{a},\mathbf{b}_2)} & (\neg\_{}_1) & \dfrac{\mathbf{a}_1 \to_\neg \mathbf{a}_2}{\neg\mathbf{a}_1 \to_\neg \neg\mathbf{a}_2}
\end{array}
$$

Table 6: Axioms and rules for negation-reduction.

defined as follows:

$$
\begin{aligned}
\mathbf{a} \to_\neg^n \mathbf{b} &:= \exists \mathbf{b}_1,\dots \mathbf{b}_{n-1} : \mathbf{a} \to_\neg \mathbf{b}_1, \dots, \mathbf{b}_{n-1} \to_\neg \mathbf{b}. \\
\mathbf{a} \to_\neg^* \mathbf{b} &:= \exists k \geq 0 : \mathbf{a} \to_\neg^k \mathbf{b}
\end{aligned}
$$

Not surprisingly, $\to_\neg$ is confluent.

Lemma 4.10 (Confluence of $\to_\neg$)**.** *$\to_\neg$ is confluent, i.e. For all $\mathbf{a},\mathbf{b},\mathbf{c}$: $\mathbf{a} \to_\neg^* \mathbf{b}$ and $\mathbf{a} \to_\neg^* \mathbf{c}$ imply $\mathbf{b} \to_\neg^* \mathbf{d}$, and $\mathbf{c} \to_\neg^* \mathbf{d}$ for some $\mathbf{d}$.*

Proof. The proof of confluence of $\to_\neg$ is straightforward by induction on the size $S(\mathbf{a})$ which counts all nodes and leaves of $\mathbf{a}$'s expression tree [7]. For the inductive base, the case $S(\mathbf{a}) = 1$ is obviously true. Consider an expression a with $S(\mathbf{a}) = n > 0$ and assume confluence for all expressions $\mathbf{b}$ with $S(\mathbf{b}) < S(\mathbf{a})$. By systematic investigation of critical pairs one can show local confluence of $\to_\neg$ from $\mathbf{a}$. Furthermore, $\to_\neg^*$ is terminating from $\mathbf{a}$ which can be seen by defining a weight function that squares the weight (incremented by one) in case of negations. Hence one can apply the diamond lemma [26] to obtain confluence of $\to_\neg$ for $\mathbf{a}$, which completes the inductive step. ☐

In order to define reduction with explicit substitution, we need to introduce the notion of environments, which are used to record the definitions which are currently available for substitution.

Definition 4.11 (Environment)**.** *Environments*, denoted by E, E_1, E_2, etc. are finite sequences of definitions $(x_1 := \mathbf{a}_1, \dots, x_n := \mathbf{a}_n)$, where x_i are variables and $x_i \neq x_j$. $\{x_1, \dots, x_n\}$ is called the *domain* of E. The lookup of an variable x in the domain of an environment E is defined by $E(x) = \mathbf{a}_i$. $E, x{:=}\mathbf{a}$ denotes the extension of E on the right by a definition $x{:=}\mathbf{a}$. E_1, E_2 denotes the concatenation of two environments. The empty environment is written as $()$.

Definition 4.12 (Single-step reduction with explicit substitution)**.** *Single-step reduction reduction with explicit substitution* $E \vdash \mathbf{a} \to_{:=} \mathbf{b}$ is the smallest relation on expressions with explicit substitution satisfying the axiom and the inference rules of Table 7. Compared to (general) reduction $\to$, the axiom β_1 and β_2 have been decomposed into several axioms:

- β_1^μ and β_2^μ are reformulation of β_1 and β_2 using internalized substitution
- *use* is unfolding single usages of definitions
- *rem* is removing a definition without usage

[7]This ensures we can apply the inductive hypothesis *e.g.* to $\neg\mathbf{a}_1$ when considering an $\mathbf{a} = [\mathbf{a}_1, \mathbf{a}_2]$, since $S(\neg\mathbf{a}_1) = S(\mathbf{a}_1) + 1 < S(\mathbf{a}_1) + S(\mathbf{a}_2) + 1 = S(\mathbf{a})$.

$$
\begin{array}{llll}
(\beta_1^\mu) & E\vdash([x:\mathbf{a}]\mathbf{b}\,\mathbf{c}) \rightarrow_{:=} [x:=\mathbf{c}]\mathbf{b} & (\beta_2^\mu) & E\vdash([x!\mathbf{a}]\mathbf{b}\,\mathbf{c}) \rightarrow_{:=} [x:=\mathbf{c}]\mathbf{b} \\
(\beta_3) & E\vdash([\mathbf{a}\,?\,\mathbf{b}]\,[\mathbf{c},:\mathbf{d}]) \rightarrow_{:=} (\mathbf{a}\,\mathbf{c}) & (\beta_4) & E\vdash([\mathbf{a}\,?\,\mathbf{b}]\,[:\mathbf{c},\mathbf{d}]) \rightarrow_{:=} (\mathbf{b}\,\mathbf{d}) \\
(use) & E\vdash x \rightarrow_{:=} \mathbf{a} & & \text{if } E(x)=\mathbf{a} \\
(rem) & E\vdash [x:=\mathbf{a}]\mathbf{b} \rightarrow_{:=} \mathbf{b} & & \text{if } x\notin FV(\mathbf{b}) \\
(\pi_1) & E\vdash [x\dot{=}\mathbf{a},\mathbf{b}:\mathbf{c}].1 \rightarrow_{:=} \mathbf{a} & (\pi_2) & E\vdash [x\dot{=}\mathbf{a},\mathbf{b}:\mathbf{c}].2 \rightarrow_{:=} \mathbf{b} \\
(\pi_3) & E\vdash [\mathbf{a},\mathbf{b}].1 \rightarrow_{:=} \mathbf{a} & (\pi_4) & E\vdash [\mathbf{a},\mathbf{b}].2 \rightarrow_{:=} \mathbf{b} \\
(\pi_5) & E\vdash [\mathbf{a}+\mathbf{b}].1 \rightarrow_{:=} \mathbf{a} & (\pi_6) & E\vdash [\mathbf{a}+\mathbf{b}].2 \rightarrow_{:=} \mathbf{b} \\
(\nu_6) & E\vdash \neg\tau \rightarrow_{:=} \tau & (\nu_7) & E\vdash \neg[x\dot{=}\mathbf{a},\mathbf{b}:\mathbf{c}] \rightarrow_{:=} [x\dot{=}\mathbf{a},\mathbf{b}:\mathbf{c}] \\
(\nu_8) & E\vdash \neg[\mathbf{a},:\mathbf{b}] \rightarrow_{:=} [\mathbf{a},:\mathbf{b}] & (\nu_9) & E\vdash \neg[:\mathbf{a},\mathbf{b}] \rightarrow_{:=} [:\mathbf{a},\mathbf{b}] \\
(\nu_{10}) & E\vdash \neg[\mathbf{a}\,?\,\mathbf{b}] \rightarrow_{:=} [\mathbf{a}\,?\,\mathbf{b}] & &
\end{array}
$$

$$(\nu)\quad \frac{\exists k>0:\mathbf{a}\rightarrow_{\neg}^{k}\mathbf{b}}{E\vdash\ \mathbf{a}\rightarrow_{:=}\mathbf{b}}$$

$$(\oplus\overbrace{(\_,\ldots,\_)}^{n}{}_i)\quad \frac{E\vdash\ \mathbf{a}_i\rightarrow_{:=}\mathbf{b}_i}{E\vdash\ \oplus(\mathbf{a}_1,\ldots,\mathbf{a}_i,\ldots,\mathbf{a}_n)\rightarrow_{:=}\oplus(\mathbf{a}_1,\ldots,\mathbf{b}_i,\ldots,\mathbf{a}_n)}$$

$$(\oplus_x\overbrace{(\_,\ldots,\_)}^{n}{}_i)\quad \frac{E\vdash\ \mathbf{a}_i\rightarrow_{:=}\mathbf{b}_i}{E\vdash\ \oplus_x(\mathbf{a}_1,\ldots,\mathbf{a}_i,\ldots,\mathbf{a}_n)\rightarrow_{:=}\oplus_x(\mathbf{a}_1,\ldots,\mathbf{b}_i,\ldots,\mathbf{a}_n)}$$

$$(L_{:=})\quad \frac{E\vdash\ \mathbf{a}\rightarrow_{:=}\mathbf{b}}{E\vdash\ [x:=\mathbf{a}]\mathbf{c}\rightarrow_{:=}[x:=\mathbf{b}]\mathbf{c}} \qquad (R_{:=})\quad \frac{E,x:=\mathbf{a}\vdash\ \mathbf{b}\rightarrow_{:=}\mathbf{c}}{E\vdash\ [x:=\mathbf{a}]\mathbf{b}\rightarrow_{:=}[x:=\mathbf{a}]\mathbf{c}}$$

Table 7: Axioms and rules for single-step reduction with explicit substitution.

The axioms $\nu_1,\ldots,\nu_5$, which are not directly confluent *e.g.* for $\neg\neg[a,b]$, have been removed and replaced by the rule ν. Furthermore there are two more structural rules $(L_{:=})$ and $(R_{:=})$ related to substitution. Note that the rule $(R_{:=})$ is pushing a definition onto the environment E when evaluating the body of a definition. Furthermore renaming may be necessary before using this rule to ensure that $E,x:=\mathbf{a}$ is well defined.

Definition 4.13 (Reduction with explicit substitution)**.** Reduction with explicit substitution $E\vdash\mathbf{a}\rightarrow_{:=}^{*}\mathbf{b}$ of $\mathbf{a}$ to $\mathbf{b}$ is defined as the reflexive and transitive closure of $E\vdash\mathbf{a}\rightarrow_{:=}\mathbf{b}$. If two expressions $\mathbf{a}$ and $\mathbf{b}$ $\rightarrow_{:=}^{*}$-reduce to a common expression we write $E\vdash\mathbf{a}\nabla_{:=}\mathbf{b}$. Zero-or-one-step reduction with explicit substitution and n-step reduction with explicit substitution are defined as follows

$$
\begin{array}{lcl}
E\vdash\mathbf{a}\rightarrow_{:=}^{01}\mathbf{b} & := & E\vdash\mathbf{a}\rightarrow_{:=}\mathbf{b}\ \vee\ \mathbf{a}=\mathbf{b} \\
E\vdash\mathbf{a}\rightarrow_{:=}^{n}\mathbf{b} & := & \exists\mathbf{b}_1,\ldots\mathbf{b}_{n-1}:E\vdash\mathbf{a}\rightarrow_{:=}\mathbf{b}_1,\ldots,E\vdash\mathbf{b}_{n-1}\rightarrow_{:=}\mathbf{b}.
\end{array}
$$

Remark 4.14 (Avoidance of name clashes through appropriate renaming)**.** Note that renaming is necessary to prepare use of the axiom *use*: For example when $\rightarrow_{:=}^{*}$-reducing $y:=x\vdash[x:\tau][y,x]$, the expression $[x:\tau][y,x]$ needs to be renamed to *e.g.* $[z:\tau][y,z]$ before substituting y by x.

Reduction $(\rightarrow^{*})$ can obviously be embedded into $\rightarrow_{:=}^{*}$.

Lemma 4.15 (Reduction implies reduction with explicit substitution)**.** *For all* a,b*:* $a\rightarrow^{*}b$ *implies* $()\vdash a\rightarrow_{:=}^{*}b$.

Proof. Proof is by induction on the definition of $a \rightarrow^* b$. □

Remark 4.16 (Sketch of the confluence proof of $\rightarrow_{:=}$)**.** First we show that $\rightarrow_{:=}$ commutes with $\rightarrow_{\neg}^{*}$. On the basis of these results, by induction on expressions with substitution one can establish direct confluence of $\rightarrow_{:=}$, i.e. $E \vdash \mathbf{a} \rightarrow_{:=} \mathbf{b}$ and $E \vdash \mathbf{a} \rightarrow_{:=} \mathbf{c}$ imply $E \vdash \mathbf{b} \rightarrow_{:=}^{01} \mathbf{d}$ and $E \vdash \mathbf{c} \rightarrow_{:=}^{01} \mathbf{d}$ for some $\mathbf{d}$. Confluence follows by two subsequent inductions.

Lemma 4.17 (Commutation of single-step reduction with explicit substitution and negation-reduction)**.** *For all* $\mathbf{a}$, $\mathbf{b}$ *and* $\mathbf{c}$*:* $E \vdash \mathbf{a} \rightarrow_{:=} \mathbf{b}$ *and* $\mathbf{a} \rightarrow_{\neg}^{*} \mathbf{c}$ *imply* $\mathbf{b} \rightarrow_{\neg}^{*} \mathbf{d}$ *and* $E \vdash \mathbf{c} \rightarrow_{:=}^{01} \mathbf{d}$ *for some* $\mathbf{d}$*. This can be graphically displayed as follows (leaving out the environment* E*):*

$$\begin{tikzcd}
\mathbf{a} \arrow[r, "\overset{*}{\neg}"] \arrow[d, ":="] & \mathbf{c} \arrow[d, "\overset{01}{:=}"] \\
\mathbf{b} \arrow[r, "\overset{*}{\neg}"] & \mathbf{d}
\end{tikzcd}$$

Proof. First we prove by induction on $\mathbf{a}$ that for all $\mathbf{b}$ and $\mathbf{c}$, $E \vdash \mathbf{a} \rightarrow_{:=} \mathbf{b}$ and $\mathbf{a} \rightarrow_{\neg} \mathbf{c}$ imply $\mathbf{b} \rightarrow_{\neg}^{*} \mathbf{d}$ and $E \vdash \mathbf{c} \rightarrow_{:=}^{01} \mathbf{d}$ for some $\mathbf{d}$. Obviously we only have to consider those cases in which there exists a $\mathbf{c}$ such that $\mathbf{a} \rightarrow_{\neg} \mathbf{c}$ (see Table 6). In case of the axioms ν_1 to ν_5 the proposition follows directly. In case of the rules for $\mathbf{a} = [\mathbf{a}_1 \oplus \mathbf{a}_2]$ or $\mathbf{a} = \oplus_x(\mathbf{a}_1, \mathbf{a}_2)$ the proposition follows either immediately or from the inductive hypothesis. In case of the rule for $\mathbf{a} = \neg\mathbf{a}_1$ the proposition follows from confluence of $\rightarrow_{\neg}$ (Lemma 4.10).

We can now prove the main property by induction on the length n of $\mathbf{a} \rightarrow_{\neg}^{n} \mathbf{c}$: In case of $n = 0$ the property is trivial. Otherwise, let $E \vdash \mathbf{a} \rightarrow_{:=} \mathbf{b}$ and $\mathbf{a} \rightarrow_{\neg}^{n} \mathbf{c}' \rightarrow_{\neg} \mathbf{c}$ for some $\mathbf{c}'$. By inductive hypothesis we know there is an expression $\mathbf{d}'$ such that $E \vdash \mathbf{c}' \rightarrow_{:=}^{01} \mathbf{d}'$ and $\mathbf{b} \rightarrow_{\neg}^{*} \mathbf{d}'$. This situation can be graphically summarized as follows (leaving out the environment E):

$$\begin{tikzcd}
\mathbf{a} \arrow[r, "\overset{n}{\neg}"] \arrow[d, ":="] & \mathbf{c}' \arrow[r, "\neg"] \arrow[d, "\overset{01}{:=}"] & \mathbf{c} \\
\mathbf{b} \arrow[r, "\overset{*}{\neg}"] & \mathbf{d}' &
\end{tikzcd}$$

Be definition of $E \vdash \mathbf{c}' \rightarrow_{:=}^{01} \mathbf{d}'$ there are two cases:

(1) $\mathbf{c}' = \mathbf{d}'$: Then we know that $\mathbf{b} \rightarrow_{\neg}^{*} \mathbf{c}'$ and hence $\mathbf{b} \rightarrow_{\neg}^{*} \mathbf{c}$. Hence $\mathbf{d} = \mathbf{c}$ where $\mathbf{b} \rightarrow_{\neg}^{*} \mathbf{d}$ and $E \vdash \mathbf{c} \rightarrow_{:=}^{01} \mathbf{d}$. This situation can be graphically summarized as follows (leaving out the environment E):

$$\begin{tikzcd}
\mathbf{a} \arrow[r, "\overset{n}{\neg}"] \arrow[d, ":="] & \mathbf{d}' = \mathbf{c}' \arrow[r, "\neg"] & \mathbf{d} = \mathbf{c} \\
\mathbf{b} \arrow[ur, "\overset{*}{\neg}"] & &
\end{tikzcd}$$

(2) $E \vdash \mathbf{c}' \rightarrow_{:=} \mathbf{d}'$. By the argument at the beginning of this proof we know there is an expression $\mathbf{d}$ such that $\mathbf{d}' \rightarrow_{\neg}^{*} \mathbf{d}$ and $E \vdash \mathbf{c} \rightarrow_{:=}^{01} \mathbf{d}$. Hence $\mathbf{b} \rightarrow_{\neg}^{*} \mathbf{d}' \rightarrow_{\neg}^{*} \mathbf{d}$ and $E \vdash \mathbf{c} \rightarrow_{:=}^{01} \mathbf{d}$ which completes the proof. This situation can be graphically summarized as follows (leaving out the environment E):

$$\begin{tikzcd}
\mathbf{a} \arrow[r, "\overset{n}{\neg}"] \arrow[d, ":="] & \mathbf{c}' \arrow[r, "\neg"] \arrow[d, "\overset{01}{:=}"] & \mathbf{c} \arrow[d, "\overset{01}{:=}"] \\
\mathbf{b} \arrow[r, "\overset{*}{\neg}"] & \mathbf{d}' \arrow[r, "\overset{*}{\neg}"] & \mathbf{d}
\end{tikzcd}$$
□

Lemma 4.18 (Direct confluence of $\rightarrow_{:=}$)**.** *For all* $E, \mathbf{a}, \mathbf{b}, \mathbf{c}$*:* $E \vdash \mathbf{a} \rightarrow_{:=} \mathbf{b}$ *and* $E \vdash \mathbf{a} \rightarrow_{:=} \mathbf{c}$ *imply* $E \vdash \mathbf{b} \rightarrow_{:=}^{01} \mathbf{d}$*, and* $E \vdash \mathbf{c} \rightarrow_{:=}^{01} \mathbf{d}$ *for some* $\mathbf{d}$*.*

Proof. Proof is by induction on $\mathbf{a}$ where in each inductive step we investigate critical pairs.

Due to the definition of $\rightarrow_{:=}$, critical pairs of $E \vdash \mathbf{a} \rightarrow_{:=} \mathbf{b}$ and $E \vdash \mathbf{a} \rightarrow_{:=} \mathbf{c}$ in which at least one of the steps is using axiom ν on top-level, i.e. where $\exists k > 0 : \mathbf{a} \rightarrow_{\neg}^{k} \mathbf{b}$ or $\exists k > 0 : \mathbf{a} \rightarrow_{\neg}^{k} \mathbf{c}$, can be resolved thanks to Lemmas 4.10 and 4.17. Note also that the reduction axioms and rules of $\rightarrow_{:=}^{*}$ do not duplicate on their right hand side any element appearing on the left hand side, hence the axiom *use* may never violate direct confluence in a critical pair.

Next we show the interesting cases of explicit definitions and applications, the other cases are straightforward.

- $\mathbf{a} = [x := \mathbf{a}_1]\mathbf{a}_2$: The matching axiom and rules are *rem*, $L_{:=}$, and $R_{:=}$. The use of $L_{:=}$ versus $R_{:=}$ follow directly from the inductive hypothesis. The interesting cases are the use of *rem*, versus $L_{:=}$ or $R_{:=}$: Hence we may assume that $x \notin FV(\mathbf{a}_2)$ and need to consider the following cases:
  (1) $\mathbf{b} = \mathbf{a}_2$ and $\mathbf{c} = [x := \mathbf{b}_1]\mathbf{a}_2$ where $E \vdash \mathbf{a}_1 \rightarrow_{:=} \mathbf{b}_1$. We have $\mathbf{d} = \mathbf{a}_2$ since $x \notin FV(\mathbf{a}_2)$ and $\mathbf{c}$ reduces in one-step to $\mathbf{a}_2$. This situation can be graphically summarized as follows (leaving out the environment E):

$$\begin{tikzcd}
\mathbf{a} = [x := \mathbf{a}_1]\mathbf{a}_2 \arrow[r, ":="] \arrow[d, ":="] & \mathbf{d} = \mathbf{b} = \mathbf{a}_2 \\
\mathbf{c} = [x := \mathbf{b}_1]\mathbf{a}_2 \arrow[ur, ":="] &
\end{tikzcd}$$

  (2) $\mathbf{b} = \mathbf{a}_2$ and $\mathbf{c} = [x := \mathbf{a}_1]\mathbf{c}_2$ where $E, x := \mathbf{a}_1 \vdash \mathbf{a}_2 \rightarrow_{:=} \mathbf{c}_2$: Obviously $x \notin FV(\mathbf{c}_2)$ and therefore $E \vdash \mathbf{a}_2 \rightarrow_{:=} \mathbf{c}_2$. Therefore $\mathbf{d} = \mathbf{c}_2$ where $E \vdash \mathbf{c} \rightarrow_{:=} \mathbf{c}_2$ and $E \vdash \mathbf{b} \rightarrow_{:=} \mathbf{c}_2$. This situation can be graphically summarized as follows (leaving out the environment E):

$$\begin{tikzcd}
\mathbf{a} = [x := \mathbf{a}_1]\mathbf{a}_2 \arrow[r, ":="] \arrow[d, ":="] & \mathbf{b} = \mathbf{a}_2 \arrow[d, ":="] \\
\mathbf{c} = [x := \mathbf{a}_1]\mathbf{c}_2 \arrow[r, ":="] & \mathbf{d} = \mathbf{c}_2
\end{tikzcd}$$

  (3) The other two cases are symmetric (exchange of $\mathbf{b}$ and $\mathbf{c}$).
- $\mathbf{a} = (\mathbf{a}_1\,\mathbf{a}_2)$: The four matching axiom and two matching rules are β_1^{μ}, β_2^{μ}, β_3, β_4, $(\_\,\_)_1$, and $(\_\,\_)_2$. Several cases have to be considered: The use of $(\_\,\_)_1$ versus $(\_\,\_)_2$ can be argued in a straightforward way using the inductive hypothesis. The simultaneous application of two different axioms on top-level is obviously not possible. The interesting remaining cases are the usage of one of the four axioms versus one of the structural rules.
  (1) The first case is the use of β_1^{μ}, i.e. $\mathbf{a}_1 = [x : \mathbf{a}_3]\mathbf{a}_4$ and $\mathbf{b} = [x := \mathbf{a}_2]\mathbf{a}_4$, versus one of the rules $(\_\,\_)_1$, and $(\_\,\_)_2$. Two subcases need to be considered:
    (a) Use of rule $(\_\,\_)_1$, i.e. $\mathbf{c} = (\mathbf{c}_1\,\mathbf{a}_2)$ where $E \vdash \mathbf{a}_1 = [x : \mathbf{a}_3]\mathbf{a}_4 \rightarrow_{:=} \mathbf{c}_1$: By definition of $\rightarrow_{:=}$, there are two subcases:
      (i) $\mathbf{c}_1 = [x : \mathbf{c}_3]\mathbf{a}_4$ where $E \vdash \mathbf{a}_3 \rightarrow_{:=} \mathbf{c}_3$: This means that $E \vdash \mathbf{c} \rightarrow_{:=} [x := \mathbf{a}_2]\mathbf{a}_4 = \mathbf{b}$, i.e. $\mathbf{d} = \mathbf{b}$ is a single-step reduct of $\mathbf{c}$. This situation can be graphically summarized as follows (leaving out the environment E):

$$\begin{tikzcd}
\mathbf{a} = (\mathbf{a}_1\,\mathbf{a}_2) = ([x : \mathbf{a}_3]\mathbf{a}_4\,\mathbf{a}_2) \arrow[r, ":="] \arrow[d, ":="] & \mathbf{d} = \mathbf{b} = [x := \mathbf{a}_2]\mathbf{a}_4 \\
\mathbf{c} = (\mathbf{c}_1\,\mathbf{a}_2) = ([x : \mathbf{c}_3]\mathbf{a}_4\,\mathbf{a}_2) \arrow[ur, ":="] &
\end{tikzcd}$$

(ii) $\mathbf{c}_1 = [x : \mathbf{a}_3]\mathbf{c}_4$ where $E \vdash \mathbf{a}_4 \rightarrow_{:=} \mathbf{c}_4$: By definition of $\rightarrow_{:=}$ we know that also $E \vdash [x := \mathbf{a}_2]\mathbf{a}_4 \rightarrow_{:=} [x := \mathbf{a}_2]\mathbf{c}_4$. Hence $\mathbf{d} = [x := \mathbf{a}_2]\mathbf{c}_4$ with $E \vdash \mathbf{b} = [x := \mathbf{a}_2]\mathbf{a}_4 \rightarrow_{:=} [x := \mathbf{a}_2]\mathbf{c}_4 = \mathbf{d}$ and $E \vdash \mathbf{c} = (\mathbf{c}_1\,\mathbf{a}_2) = ([x : \mathbf{a}_3]\mathbf{c}_4\,\mathbf{a}_2) \rightarrow_{:=} [x{:=}\mathbf{a}_2]\mathbf{c}_4 = \mathbf{d}$. This situation can be graphically summarized as follows (leaving out the environment E):

$$\begin{tikzcd}
\mathbf{a} = (\mathbf{a}_1\,\mathbf{a}_2) = ([x : \mathbf{a}_3]\mathbf{a}_4\,\mathbf{a}_2) \arrow[r, ":="] \arrow[d, ":="] & \mathbf{b} = [x{:=}\mathbf{a}_2]\mathbf{a}_4 \arrow[d, ":="] \\
\mathbf{c} = (\mathbf{c}_1\,\mathbf{a}_2) = ([x : \mathbf{a}_3]\mathbf{c}_4\,\mathbf{a}_2) \arrow[r, ":="] & \mathbf{d} = [x{:=}\mathbf{a}_2]\mathbf{c}_4
\end{tikzcd}$$

(b) The use of rule $(\_\,\_)_2$ can be argued in a similar style.

(2) The second case is the use of β_2^μ and can be argued in a similar style.

(3) The third case is the use of β_3, i.e. $\mathbf{a}_1 = [\mathbf{a}_3\,?\,\mathbf{a}_4]$, $\mathbf{a}_2 = [\mathbf{a}_5, : \mathbf{a}_6]$ and $\mathbf{b} = \mathbf{a}_3(\mathbf{a}_5)$, versus one of the rules $(\_\,\_)_1$, and $(\_\,\_)_2$. The property follows by an obvious case distinction on whether $E \vdash [\mathbf{a}_3\,?\,\mathbf{a}_4] \rightarrow_{:=} \mathbf{c}$ is reducing in $\mathbf{a}_3$ or $\mathbf{a}_4$, and similarly for $E \vdash [\mathbf{a}_5, : \mathbf{a}_6] \rightarrow_{:=} \mathbf{c}$.

(4) The fourth case is symmetric to the third one. □

Lemma 4.19 (Confluence of $\rightarrow_{:=}$)**.** *For all $E, \mathbf{a}, \mathbf{b}, \mathbf{c}$: $E \vdash \mathbf{a} \rightarrow^*_{:=} \mathbf{b}$ and $E \vdash \mathbf{a} \rightarrow^*_{:=} \mathbf{c}$ implies $E \vdash \mathbf{b} \nabla_{:=} \mathbf{c}$.*

Proof. Using Lemma 4.18, by structural induction on the number n of transition steps one can show that $E \vdash \mathbf{a} \rightarrow^n_{:=} \mathbf{b}$ and $E \vdash \mathbf{a} \rightarrow_{:=} \mathbf{c}$ implies $E \vdash \mathbf{b} \rightarrow^{01}_{:=} \mathbf{d}$, and $E \vdash \mathbf{c} \rightarrow^*_{:=} \mathbf{d}$ where for some $\mathbf{d}$. Using this intermediate result, by structural induction on the number of transition steps n one can show for any m that $E \vdash \mathbf{a} \rightarrow^n_{:=} \mathbf{b}$ and $E \vdash \mathbf{a} \rightarrow^*_{:=} \mathbf{c}$ implies $E \vdash \mathbf{b} \rightarrow^*_{:=} \mathbf{d}$ and $E \vdash \mathbf{c} \rightarrow^*_{:=} \mathbf{d}$ for some $\mathbf{d}$. This proves confluence of $\rightarrow_{:=}$. □

By Lemma 4.15 we know that reduction implies reduction with explicit substitution. The reverse direction is obviously not true. However we can show that $E \vdash \mathbf{a} \rightarrow^*_{:=} \mathbf{b}$ implies $\mathbf{a}' \rightarrow^* \mathbf{b}'$ where $\mathbf{a}'$ and $\mathbf{b}'$ result from $\mathbf{a}$ and $\mathbf{b}$ by maximal evaluation of definitions, i.e. by maximal application of the axioms *use* and *rem*. Therefore we introduce the relation of *definition evaluation*

Definition 4.20 (Single-step definition-evaluation)**.** *Single-step definition-evaluation* $E \vdash \mathbf{a} \rightarrow_{=} \mathbf{b}$ is defined just like single-step reduction with explicit substitution $E \vdash \mathbf{a} \rightarrow_{:=} \mathbf{b}$ (see Table 7) but without the rule ν and without any axioms except *rem* and *use.*

Lemma 4.21 (Confluence of $\rightarrow_{=}$)**.** *$\rightarrow_{=}$ is confluent.*

Proof. By removing all the axiom cases except *rem* and *use* and the rule ν in the proof of Lemma 4.18 (where these axioms only interacted with the structural rules for internalized substitution) it can be turned into a proof of direct confluence of $\rightarrow_{=}$. The confluence of $\rightarrow_{=}$ then follows as in the proof of Lemma 4.19. □

Lemma 4.22 (Strong normalization of definition-evaluation)**.** *There are no infinite chains $E \vdash \mathbf{a}_1 \rightarrow_{=} \mathbf{a}_2 \rightarrow_{=} \mathbf{a}_3 \rightarrow_{=} \ldots$*

Proof. Since all definitions are non-recursive the property is straightforward. One way to see this is to define a weight $W(E, \mathbf{a})$ such that

$$E \vdash \mathbf{a} \rightarrow_{=} \mathbf{b} \quad \text{implies} \quad W(E, \mathbf{b}) < W(E, \mathbf{a})$$

This can be achieved by defining $W(E, x) = W(E, E(x)) + 1$ if x is in the domain of E otherwise $W(E, x) = 1$, $W(E, [x{:=}\mathbf{a}]\mathbf{b}) = W(E, \mathbf{a}) + W((E, x := \mathbf{a}), \mathbf{b}) + 1$, and so on. □

Definition 4.23 (Definitional normal form)**.** For any $\mathbf{a}$, let $\mathcal{N}^{=}_{E}(\mathbf{a})$ denote the *definitional normal form*, i.e. the expression resulting from $\mathbf{a}$ by maximal application of definition evaluation steps under environment E. This definition is sound since due to Lemma 4.22 the evaluation of definitions always terminates and by Lemma 4.21 the maximal evaluation of definitions delivers a unique result.

Lemma 4.24 (Basic properties of $\mathcal{N}^{=}_{E}(\mathbf{a})$)**.** *For all* $E, x, \mathbf{a}_1, \dots \mathbf{a}_n$*:*

(1) $\mathcal{N}^{=}_{E}(\oplus(\mathbf{a}_1, \dots, \mathbf{a}_n)) = \oplus(\mathcal{N}^{=}_{E}(\mathbf{a}_1), \dots, \mathcal{N}^{=}_{E}(\mathbf{a}_n))$.
(2) $\mathcal{N}^{=}_{E}(\oplus_x(\mathbf{a}_1, \dots, \mathbf{a}_n)) = \oplus_x(\mathcal{N}^{=}_{E}(\mathbf{a}_1), \dots, \mathcal{N}^{=}_{E}(\mathbf{a}_n))$.
(3) $\mathcal{N}^{=}_{E}([x:=\mathbf{a}_1]\mathbf{a}_2) = \mathcal{N}^{=}_{E}(\mathbf{a}_2[\mathbf{a}_1/x]) = \mathcal{N}^{=}_{E}(\mathbf{a}_2)[\mathcal{N}^{=}_{E}(\mathbf{a}_1)/x] = \mathcal{N}^{=}_{E,x:=\mathbf{a}_1}(\mathbf{a}_2)$.

Proof.

(1) The proof is by induction on the length of an arbitrary definition evaluation $E \vdash \oplus(\mathbf{a}_1, \dots, \mathbf{a}_n) \rightarrow_{=} \mathcal{N}^{=}_{E}(\oplus(\mathbf{a}_1, \dots, \mathbf{a}_n))$. It is obvious since the outer operation is never affected by definition evaluation.
(2) Similarly to (1)
(3) Using (1), (2), and the Lemmas 4.4, 4.15, and 4.21, these equalities can be shown either directly or by straightforward inductive arguments. □

Lemma 4.25 (Embedding property of reduction with explicit substitution)**.** *For all* $E, \mathbf{a}, \mathbf{b}$*:* $E \vdash \mathbf{a} \rightarrow^{*}_{:=} \mathbf{b}$ *implies* $\mathcal{N}^{=}_{E}(\mathbf{a}) \rightarrow^{*} \mathcal{N}^{=}_{E}(\mathbf{b})$*. As a consequence, for all* a *and* b*,* $\vdash a \nabla_{:=} b$ *implies* $a \nabla b$*.*

Proof. First we show by induction on single-step reduction with explicit substitution that $E \vdash \mathbf{a} \rightarrow_{:=} \mathbf{b}$ implies $\mathcal{N}^{=}_{E}(\mathbf{a}) \rightarrow^{*} \mathcal{N}^{=}_{E}(\mathbf{b})$. The proof is straightforward using Lemmas 4.5 and 4.24, and the fact that, for any E, $\mathbf{a} \rightarrow^{n}_{\neg} \mathbf{b}$ implies $\mathcal{N}^{=}_{E}(\mathbf{a}) \rightarrow^{n} \mathcal{N}^{=}_{E}(\mathbf{b})$ which follows by a straightforward induction on the definition of $\mathbf{a} \rightarrow^{n}_{\neg} \mathbf{b}$.

Now we turn to the main proposition. Obviously, we have $E \vdash \mathbf{a} \rightarrow^{n}_{:=} \mathbf{b}$ for some n. The proof is by induction on n. For $n = 0$ the property is trivial. For $n > 0$ assume $E \vdash \mathbf{a} \rightarrow_{:=} \mathbf{c} \rightarrow^{n-1}_{:=} \mathbf{b}$. By inductive hypothesis $\mathcal{N}^{=}_{E}(\mathbf{c}) \rightarrow^{*} \mathcal{N}^{=}_{E}(\mathbf{b})$. By the above argument we know that $\mathcal{N}^{=}_{E}(\mathbf{a}) \rightarrow^{*} \mathcal{N}^{=}_{E}(\mathbf{c})$ which implies the proposition.

For the immediate consequence assume that there is an expression $\mathbf{c}$ such that $\vdash a \rightarrow^{*}_{:=} \mathbf{c}$ and $\vdash b \rightarrow^{*}_{:=} \mathbf{c}$. We have just shown that also $\mathcal{N}^{=}_{()}(a) \rightarrow^{*} \mathcal{N}^{=}_{()}(\mathbf{c})$ and $\mathcal{N}^{=}_{()}(b) \rightarrow^{*} \mathcal{N}^{=}_{()}(\mathbf{c})$. The property follows since obviously $a = \mathcal{N}^{=}_{()}(a)$ and $b = \mathcal{N}^{=}_{()}(b)$. Hence $a \nabla b$. □

Theorem 4.26 (Confluence of $\rightarrow$)**.** *For all* a, b, c*:* $a \rightarrow^{*} b$ *and* $a \rightarrow^{*} c$ *implies* $b \nabla c$*. As a consequence* $a =_{\lambda} b$ *implies* $a \nabla b$*.*

Proof. By Lemma 4.15 we know that $\vdash a \rightarrow^{*}_{:=} b$ and $\vdash a \rightarrow^{*}_{:=} c$. Due to confluence of $\rightarrow_{:=}$ (Lemma 4.19) we know that $\vdash b \nabla_{:=} c$. By Lemma 4.25 we obtain $b \nabla c$. The consequence follows by induction on the definition of $a =_{\lambda} b$. □

As an immediate consequence of confluence we note some basic properties of congruence.

Corollary 4.27 (Basic properties of congruence)**.** *For all* x, y, a, b, c, d*:*

(1) $a =_{\lambda} b$ *implies* $a[c/x] =_{\lambda} b[c/x]$ *and* $c[a/x] =_{\lambda} c[b/x]$*.*
(2) $\oplus_x(a_1, \dots, a_n) =_{\lambda} \oplus_y(b_1, \dots, b_n)$ *iff* $x = y$ *and* $a_i =_{\lambda} b_i$ *for* $i = 1, \dots, n$*.*
(3) $[a \oplus b] =_{\lambda} [c \oplus d]$ *iff* $a =_{\lambda} c$ *and* $b =_{\lambda} d$*.*

Proof. Follow from Lemmas 4.5 and 4.6, and Theorem 4.26. □

4.2. **Properties of typing.** Based on the confluence result we can now show the main properties of typing, in particular subject reduction and uniqueness of types. We frequently show properties of typing by induction on the definition of typing where the inductive base corresponds to the type axiom and the inductive step corresponds to one of the type inference rules. We begin by two properties (4.28,4.29) related to the rule *weak*, a property (4.30) related to the rule *start*, and a property (4.31) related to the rule *conv*.

Lemma 4.28 (Context weakening). *For all* $\Gamma_1, \Gamma_2, x, a, b, c$*:* $(\Gamma_1, \Gamma_2) \vdash a : b$ *and* $\Gamma_1 \vdash c$ *imply* $(\Gamma_1, x : c, \Gamma_2) \vdash a : b$.

Proof. Proof is by induction on the definition of $(\Gamma_1, \Gamma_2) \vdash a : b$. For all the different type rules the property follows directly from the definition of typing and the inductive hypothesis. □

Lemma 4.29 (Context extraction). *For all* Γ_1, Γ_2, a, b*:* $(\Gamma_1, x : a, \Gamma_2) \vdash b$ *implies* $\Gamma_1 \vdash a$

Proof. $(\Gamma_1, x : a, \Gamma_2) \vdash b$ means that there is an expression c where $(\Gamma_1, x : a, \Gamma_2) \vdash b : c$. The property that $(\Gamma_1, x : a, \Gamma_2) \vdash b : c$ implies $\Gamma_1 \vdash a$ is shown by induction on the definition of $(\Gamma_1, x : a, \Gamma_2) \vdash b : c$. For all the different type rules the property follows directly from the definition of typing and the inductive hypothesis. □

Lemma 4.30 (Start property). *For all* Γ, x, a, b*:* $(\Gamma, x : a) \vdash x : b$ *implies* $a =_\lambda b$.

Proof. The proof is by induction on $\Gamma, x : a \vdash x : b$. Obviously the only relevant rules are *start* and *conv*. In case of *start*, we directly obtain $a = b$. In case of *conv*, we know that $\Gamma, x : a \vdash x : c$ for some c where $c =_\lambda b$. By inductive hypothesis it follows that $a =_\lambda c$ and hence obviously $a =_\lambda b$. □

Lemma 4.31 (Context equivalence and typing). *Let* $\Gamma_a = (\Gamma_1, x : a, \Gamma_2)$ *and* $\Gamma_b = (\Gamma_1, x : b, \Gamma_2)$ *for some* $\Gamma_1, \Gamma_2, x, a, b$ *where* $a =_\lambda b$ *and* $\Gamma_1 \vdash b$*: For all* c, d*: If* $\Gamma_a \vdash c : d$ *then* $\Gamma_b \vdash c : d$.

Proof. Let Γ_a and Γ_b where $a =_\lambda b$ and $\Gamma_1 \vdash b$ as defined above: The property is shown by induction on the definition of $(\Gamma_1, x : a, \Gamma_2) \vdash c : d$. In all cases, the property follows directly from the definition of typing and the inductive hypothesis.

As an example we show the case of rule *start*: We have $\Gamma_a = (\Gamma_1, x : a, \Gamma_2) = (\Gamma_3, y : d)$ and $c = y$ for some y and Γ_3 where $\Gamma_3 \vdash d$. There are two cases:

- If $x = y$ then $\Gamma_a \vdash x : a$, i.e. $d = a$, $\Gamma_2 = ()$, $\Gamma_1 = \Gamma_3$, $\Gamma_a = (\Gamma_1, x : a)$, $\Gamma_b = (\Gamma_1, x : b)$, and $\Gamma_1 \vdash a$. Since $\Gamma_1 \vdash b$, by rule *weak* it follows that $\Gamma_b \vdash a$. By rule *start* we obtain $\Gamma_b \vdash x : b$. Hence by rule *conv* it follows that $\Gamma_b \vdash x : a$. This can be graphically illustrated as follows:

$$\textit{conv}\ \dfrac{\textit{start}\ \dfrac{\Gamma_1 \vdash b}{\Gamma_1, x : b \vdash\ x : b} \qquad b =_\lambda a \qquad \textit{weak}\ \dfrac{\Gamma_1 \vdash a \quad \Gamma_1 \vdash b}{\Gamma_1, x : b \vdash a}}{\Gamma_b = (\Gamma_1, x : b) \vdash\ x : a}$$

  Note that the proof of this case one does not need the assumption $\Gamma_a = (\Gamma_1, x : a) \vdash x : a$.
- If $x \neq y$ then we have $\Gamma_2 = (\Gamma_4, y : d)$ for some Γ_4 where $\Gamma_3 = (\Gamma_1, x : a, \Gamma_4)$. Since $\Gamma_1, x : a, \Gamma_4 \vdash d$, by inductive hypothesis we know that also $\Gamma_1, x : b, \Gamma_4 \vdash d$ and therefore, by rule *start*, we obtain $\Gamma_b = (\Gamma_1, x : b, \Gamma_2) \vdash y : d$. □

Next we list several decomposition properties of typing and of validity.

Lemma 4.32 (Typing decomposition). *For all* $\Gamma, x, a_1, a_2, a_3, b$*:*

(1) $\Gamma \vdash \oplus_x(a_1, a_2) : b$ *implies* $b =_\lambda [x : a_1]c$ *for some* c *where* $(\Gamma, x : a_1) \vdash a_2 : c$.

(2) $\Gamma \vdash [a_1, : a_2] : b$ *implies* $b =_\lambda [c + a_2]$ *for some* c *where* $\Gamma \vdash a_1 : c$ *and* $\Gamma \vdash a_2$.
$\Gamma \vdash [: a_1, a_2] : b$ *implies* $b =_\lambda [a_1 + c]$ *for some* c *where* $\Gamma \vdash a_2 : c$ *and* $\Gamma \vdash a_1$.
(3) $\Gamma \vdash [a_1 \oplus a_2] : b$ *implies* $b =_\lambda [c_1, c_2]$ *for some* c_1, c_2 *where* $\Gamma \vdash a_1 : c_1$ *and* $\Gamma \vdash a_2 : c_2$.
(4) $\Gamma \vdash \neg a_1 : b$ *implies* $b =_\lambda c$ *for some* c *where* $\Gamma \vdash a_1 : c$.
(5) $\vdash \tau : b$ *implies* $b =_\lambda \tau$.
(6) $\Gamma \vdash (a_1\, a_2) : b$ *implies* $b =_\lambda c_2[a_2/x]$ *where* $\Gamma \vdash a_1 : [x : c_1]c_2$ *and* $\Gamma \vdash a_2 : c_1$ *for some* c_1,c_2.
(7) $\Gamma \vdash a_1.1 : b$ *implies* $b =_\lambda c_1$ *where* $\Gamma \vdash a_1 : [x!c_1]c_2$ *or* $\Gamma \vdash a_1 : [c_1, c_2]$ *for some* c_1, c_2.
(8) $\Gamma \vdash a_1.2 : b$ *implies, for some* c_1, c_2, *that either* $b =_\lambda c_2$ *where* $\Gamma \vdash a_1 : [c_1, c_2]$ *or* $b =_\lambda c_2[a_1.1/x]$ *where* $\Gamma \vdash a_1 : [x!c_1]c_2$.
(9) $\Gamma \vdash [x \doteq a_1, a_2 : a_3] : b$ *implies* $b =_\lambda [x!c]a_3$ *for some* c *where* $\Gamma \vdash a_1 : c$, $\Gamma \vdash a_2 : a_3[a_1/x]$, *and* $(\Gamma, x : c) \vdash a_3$.
(10) $\Gamma \vdash [a_1\, ?\, a_2] : b$ *implies* $b =_\lambda [x : [c_1 + c_2]]c$ *for some* c_1, c_2, *and* c *where* $\Gamma \vdash a_1 : [x : c_1]c$, $\Gamma \vdash a_2 : [x : c_2]c$ *and* $\Gamma \vdash c$.

Proof. In all cases the proof is by induction on the definition of typing or by a straightforward application of previous properties. In particular, 4.6, 4.26, and 4.28 are needed. □

Lemma 4.33 (Validity decomposition)**.** *For all* $\Gamma, x, a_1, \ldots, a_n$*:*
(1) $\Gamma \vdash \oplus(a_1, , \ldots a_n)$ *implies* $\Gamma \vdash a_1$, …, $\Gamma \vdash a_n$.
(2) $\Gamma \vdash \oplus_x(a_1, a_2)$ *implies* $\Gamma \vdash a_1$ *and* $(\Gamma, x : a_1) \vdash a_2$
(3) $\Gamma \vdash [x \doteq a_1, a_2 : a_3]$ *implies* $\Gamma \vdash a_2$ *and* $\Gamma \vdash a_1 : b$ *for some* b *where* $(\Gamma, x : b) \vdash a_3$

Proof. In all cases the proof is by induction on the definition of typing or by a straightforward application of previous properties. In particular, Lemma 4.29 is needed. □

A central prerequisite to the proof of closure of reduction and typing against validity is the following substitution property of typing. In order to state the property, we need an auxiliary definition.

Definition 4.34 (Context substitution)**.** The substitution function (Definition 2.2) is extended to contexts $\Gamma[a/x]$, where a is an expression, as follows:

$$\begin{array}{rcl} ()[a/x] & = & () \\ (y : b, \Gamma)[a/x] & = & \begin{cases} (y : b[a/x], \Gamma) & \text{if } x = y \\ (y : b[a/x], \Gamma[a/x]) & \text{otherwise} \end{cases} \end{array}$$

Lemma 4.35 (Substitution and typing)**.** *Assume that* $\Gamma_a = (\Gamma_1, x : a, \Gamma_2)$ *and* $\Gamma_b = (\Gamma_1, \Gamma_2[b/x])$ *for some* $\Gamma_1, \Gamma_2, x, a, b$ *where* $\Gamma_1 \vdash b : a$. *For all* c, d*: If* $\Gamma_a \vdash c : d$ *then* $\Gamma_b \vdash c[b/x] : d[b/x]$.

Proof. Let Γ_a and Γ_b as defined above and assume $\Gamma_1 \vdash b : a$. The proof that $\Gamma_a \vdash c : d$ implies $\Gamma_b \vdash c[b/x] : d[b/x]$ is by induction on the definition of $\Gamma_a \vdash c : d$. We show two interesting cases.

- Rule *conv*: We have $\Gamma_a \vdash c : d'$ for some d' where $d' =_\lambda d$ and $\Gamma_a \vdash d' : e$ for some e. By inductive hypothesis $\Gamma_b \vdash c[b/x] : d'[b/x]$ and $\Gamma_b \vdash d'[b/x] : e[b/x]$. By Lemma 4.27(1) we can infer that $d'[b/x] =_\lambda d[b/x]$. Therefore we can apply the rule *conv* to obtain $\Gamma_b \vdash c[b/x] : d[b/x]$. This can be graphically illustrated as follows (IH denotes the use of an inductive hypothesis):

$$conv\ \frac{4.27(1)\ \dfrac{d' =_\lambda d}{d'[b/x] =_\lambda d[b/x]} \quad \mathrm{IH}\ \dfrac{\Gamma_a \vdash c : d'}{\Gamma_b \vdash c[b/x] : d'[b/x]} \quad \mathrm{IH}\ \dfrac{\Gamma_a \vdash d' : e}{\Gamma_b \vdash d'[b/x] : e[b/x]}}{\Gamma_b \vdash\ c[b/x] : d[b/x]}$$

- Rule *appl*: We have $c = (c_1\, c_2)$, $d = d_2[c_2/y]$ for some y, c_1, c_2, and d_2 where $\Gamma_a \vdash (c_1\, c_2) : d$, $\Gamma_a \vdash c_1 : [y : d_1]d_2$, and $\Gamma_a \vdash c_2 : d_1$. Obviously, we can assume that $x \neq y$. We need to show that $\Gamma_b \vdash (c_1\, c_2)[b/x] : d_2[c_2/y][b/x]$. We have:

$$\begin{array}{rl} \Gamma_b \vdash & c_1[b/x] \\ : & \quad \text{(inductive hypothesis on } \Gamma_a \vdash c_1 : [y : d_1]d_2) \\ & ([y : d_1]d_2)[b/x] \\ = & \quad \text{(definition of substitution)} \\ & [y : d_1[b/x]](d_2[b/x]) \end{array}$$

Furthermore, since $\Gamma_a \vdash c_2 : d_1$, by inductive hypothesis we know that $\Gamma_b \vdash c_2[b/x] : d_1[b/x]$. Hence by type rule *appl* we obtain

$$\Gamma_b \vdash (c_1[b/x]\, c_2[b/x]) : d_2[b/x][c_2[b/x]/y] \qquad (*)$$

We can now argue as follows:

$$\begin{array}{rl} \Gamma_b \vdash & (c_1\, c_2)[b/x] \\ = & \quad \text{(definition of substitution)} \\ & (c_1[b/x]\, c_2[b/x]) \\ : & \quad (\text{see } *) \\ & d_2[b/x][c_2[b/x]/y] \\ = & \quad \text{(Lemma 4.4(2))} \\ & d_2[c_2/y][b/x] \end{array}$$

The other cases can be shown in a similar style. □

We begin the closure properties with the relatively straightforward property that validity is closed against typing.

Lemma 4.36 (Valid expressions have valid types)**.** *For all* Γ, a, b*: If* $\Gamma \vdash a : b$ *then* $\Gamma \vdash b$*.*

Proof. By induction on the definition of $\Gamma \vdash a : b$ we show that $\Gamma \vdash a : b$ implies $\Gamma \vdash b$. As an example, we illustrate the rule for applications

- Rule *appl*: We have $a = (a_1\, a_2)$ and $b = b_2[a_2/x]$ for some x, a_1, a_2, and b_2 where $\Gamma \vdash a_1 : [x : b_1]b_2$ for some b_1 and $\Gamma \vdash a_2 : b_1$. From the inductive hypothesis we know that $\Gamma \vdash [x : b_1]b_2$. By Lemma 4.33(2) this implies $\Gamma, x : b_1 \vdash b_2$. From $\Gamma, x : b_1 \vdash b_2$ and from substitution (Lemma 4.35, note that $\Gamma \vdash a_2 : b_1$) we can infer that $\Gamma \vdash b_2[a_2/x] = b$.

The other cases can be shown in a similar style. □

We are now ready to show the preservation of types under a reduction step.

Lemma 4.37 (Preservation of types under reduction steps)**.** *For all* Γ, a, b, c*:* $\Gamma \vdash a : c$ *and* $a \to b$ *imply* $\Gamma \vdash b : c$*.*

Proof. Proof by induction on the definition of $a \to b$. Note that by Lemma 4.36 we have $\Gamma \vdash c$. This means that in order to show $\Gamma \vdash b : c$, it is sufficient to show $\Gamma \vdash b : b_1$ for some b_1 where $b_1 =_\lambda c$. $\Gamma \vdash b : c$ then follows by applying rule *conv*. We show the cases of axiom β_1 and of structural rule $\oplus_x(\_,\_)_1$ in detail:

- Axiom β_1: We have $a = ([x : a_1]a_2\ a_3)$ and $b = a_2[a_3/x]$ and $\Gamma \vdash ([x : a_1]a_2\ a_3) : c$ where $\Gamma \vdash a_3 : a_1$. The following type relations can be derived:

$$\begin{array}{ll} & \Gamma \vdash ([x : a_1]a_2\ a_3) : c \\ \Rightarrow & \quad \text{(Lemma 4.32(6))} \\ & \Gamma \vdash [x : a_1]a_2 : [x : d]e \quad \text{where } c =_\lambda e[a_3/x]\ (*) \text{ and } \Gamma \vdash a_3 : d \\ \Rightarrow & \quad \text{(Lemma 4.32(1))} \\ & (\Gamma, x : a_1) \vdash a_2 : b_2 \quad \text{where } [x : d]e =_\lambda [x : a_1]b_2 \\ \Rightarrow & \quad \text{(Lemma 4.35, since } \Gamma \vdash a_3 : a_1) \\ & \Gamma \vdash a_2[a_3/x] : b_2[a_3/x] \end{array}$$

  From $[x : d]e =_\lambda [x : a_1]b_2$, by Lemma 4.27(2) it follows that $e =_\lambda b_2$. We can therefore argue as follows:

$$\begin{array}{ll} & b_2[a_3/x] \\ =_\lambda & \quad \text{(by Lemma 4.27(1) since } e =_\lambda b_2) \\ & e[a_3/x] \\ =_\lambda & \quad \text{(see } * \text{ in the first argument chain)} \\ & c \end{array}$$

  Therefore we can apply rule *conv* to derive $\Gamma \vdash b = a_2[a_3/x] : c$. This can be graphically illustrated as follows:

$$conv\ \frac{\Gamma \vdash a_2[a_3/x] : b_2[a_3/x] \qquad b_2[a_3/x] =_\lambda c \qquad \Gamma \vdash c}{\Gamma \vdash\ a_2[a_3/x] : c}$$

- Rule $\oplus_x(\_,\_)_1$: We have $a = \oplus_x(a_1, a_3)$, $b = \oplus_x(a_2, a_3)$ where $a_1 \to a_2$, and $\Gamma \vdash \oplus_x(a_1, a_3) : c$. By Lemma 4.32(1), $c =_\lambda [x : a_1]c_3$ for some c_3 where $(\Gamma, x : a_1) \vdash a_3 : c_3$. By Lemma 4.29 this implies that $\Gamma \vdash a_1$ and therefore by inductive hypothesis we know that $\Gamma \vdash a_2$. Therefore we can apply Lemma 4.31 which implies that $(\Gamma, x : a_2) \vdash a_3 : c_3$. By definition of typing $\Gamma \vdash \oplus_x(a_2, a_3) : [x : a_2]c_3$. Since $[x : a_2]c_3 =_\lambda [x : a_1]c_3 =_\lambda c$, we can apply type rule *conv* to derive $\Gamma \vdash b = \oplus_x(a_2, a_3) : c$.

The other cases can be shown in a similar style using also Lemma 4.5. □

A simple inductive argument extends Lemma 4.37 to general reduction. This property is often referred to as *subject reduction.*

Theorem 4.38 (Subject reduction: Types are preserved under reduction)**.** *For all* Γ, a, b, c*:* $a \to^* b$ *and* $\Gamma \vdash a : c$ *imply that* $\Gamma \vdash b : c$*.*

Theorem 4.38 can be reformulated using the validity notation.

Corollary 4.39 (Valid expressions are closed against reduction)**.** *For all* Γ, a, b*:* $\Gamma \vdash a$ *and* $a \to^* b$ *implies* $\Gamma \vdash b$

Subject reduction can be extended analogously to types.

Lemma 4.40 (Elements are preserved under type reduction)**.** *For all* Γ, a, b, c*:* $\Gamma \vdash a : b$ *and* $b \rightarrow^* c$ *implies* $\Gamma \vdash a : c$*.*

Proof. By Lemma 4.36 $\Gamma \vdash b$. By Corollary 4.39 $\Gamma \vdash c$. The proposition then follows from rule *conv.* □

Before we show uniqueness of types, we need a Lemma about the effect of removing unnecessary type declarations from a context.

Lemma 4.41 (Context contraction)**.** *For all* $\Gamma_1, \Gamma_2, x, a, b, c$*:* $(\Gamma_1, x : c, \Gamma_2) \vdash a : b$ *and* $x \notin FV([\Gamma_2]a)$ *implies* $(\Gamma_1, \Gamma_2) \vdash a : b'$ *and* $b \rightarrow^* b'$ *for some* b'*.*

Note that in this lemma $x \notin FV([\Gamma_2]a)$ does not necessarily imply $x \notin FV([\Gamma_2]b)$, for example take $x : \tau \vdash \tau : ([y : \tau]\tau\, x)$, and hence the reduction of the type to some expression where x does not occur free, in the example $([y : \tau]\tau\, x) \rightarrow^* \tau$, is necessary.

Proof. The proof is by induction on a. In the base cases of $a = \tau$ and $a = y$ context contraction can be show by a simple induction on the definition of typing using Theorem 4.26 and Lemma 4.38. We show two other interesting cases.

- $a = \oplus_x(a_1, a_2)$: By 4.32(1) $(\Gamma_1, x : c, \Gamma_2) \vdash \oplus_y(a_1, a_2) : b$ implies $b =_\lambda [y : a_1]d$ for some d where $(\Gamma_1, x : c, \Gamma_2, y : a_1) \vdash a_2 : d$. Since $x \notin FV([\Gamma_2] \oplus_x (a_1, a_2)) = FV([\Gamma_2, y : a_1]a_2)$, by inductive hypothesis $(\Gamma_1, \Gamma_2, y : a_1) \vdash a_2 : d'$ where $d \rightarrow^* d'$ for some d'. By definition of typing this implies $(\Gamma_1, \Gamma_2) \vdash \oplus_y(a_1, a_2) : [y : a_1]d'$. Since $b =_\lambda [y : a_1]d'$ by 4.26 and 4.40 we know that $(\Gamma_1, \Gamma_2) \vdash \oplus_y(a_1, a_2) : b'$ where b' is a common reduct of b and $[y : a_1]d'$.
- $a = (a_1\, a_2)$: By 4.32(6) we have $b =_\lambda c_2[a_2/y]$ where $y \neq x$, $(\Gamma_1, x : c, \Gamma_2) \vdash a_1 : [y : c_1]c_2$ and $(\Gamma_1, x : c, \Gamma_2) \vdash a_2 : c_1$ for some c_1, c_2. Since $x \notin FV([\Gamma_2](a_1\, a_2))$ we know that $x \notin FV([\Gamma_2]a_1) \cup FV([\Gamma_2]a_2)$. By inductive hypothesis $(\Gamma_1, \Gamma_2) \vdash a_1 : d$ where $[y : c_1]c_2 \rightarrow^* d$, for some d, and $(\Gamma_1, \Gamma_2) \vdash a_2 : e$ where $c_1 \rightarrow^* e$, for some e.

  By 4.6(1) we know that $d = [y : d_1]d_2$ where $c_1 \rightarrow^* d_1$ and $c_2 \rightarrow^* d_2$, for some d_1 and d_2. Since $e =_\lambda d_1$, by 4.26 and 4.40 we know that $(\Gamma_1, \Gamma_2) \vdash a_2 : e'$ where e' is a common reduct of e and d_1. Since $[y : d_1]d_2 \rightarrow^* [y : e']d_2$, by 4.40 we know that $(\Gamma_1, \Gamma_2) \vdash a_1 : [y : e']d_2$.

  Hence by definition of typing $(\Gamma_1, \Gamma_2) \vdash (a_1\, a_2) : d_2[a_2/y]$. Since $b =_\lambda c_2[a_2/y] =_\lambda d_2[a_2/y]$, by 4.26 and 4.40 we know that $(\Gamma_1, \Gamma_2) \vdash a : b'$ where b' is a common reduct of b and $d_2[a_2/y]$.
- $a = [y \dot= a_1, a_2 : a_3]$: By 4.32(9) we have $b =_\lambda [y!d]a_3$ for some d where $(\Gamma_1, x : c, \Gamma_2) \vdash a_1 : d$, $(\Gamma_1, x : c, \Gamma_2) \vdash a_2 : a_3[a_1/y]$, and $(\Gamma_1, x : c, \Gamma_2, y : d) \vdash a_3$. Since $x \notin FV([\Gamma_2][y \dot= a_1, a_2 : a_3])$ we know that $x \notin FV([\Gamma_2]a_1) \cup FV([\Gamma_2]a_2) \cup FV([\Gamma_2]a_3)$. By inductive hypothesis $(\Gamma_1, \Gamma_2) \vdash a_1 : d'$ where $d \rightarrow^* d'$ for some d' and $(\Gamma_1, \Gamma_2) \vdash a_2 : e$ where $a_3[a_1/y] \rightarrow^* e$ for some e.

  By 4.36 we know that $(\Gamma_1, \Gamma_2) \vdash d'$. By 4.29 we have $\Gamma_1 \vdash c$ and therefore by 4.28 we know that $(\Gamma_1, x : c, \Gamma_2) \vdash d'$. Hence from $(\Gamma_1, x : c, \Gamma_2, y : d) \vdash a_3$ by 4.31 we can conclude that $(\Gamma_1, x : c, \Gamma_2, y : d') \vdash a_3$. By inductive hypothesis, since $x \notin FV([\Gamma_2, y : d']a_3)$, we obtain $(\Gamma_1, \Gamma_2, y : d') \vdash a_3$. Since $(\Gamma_1, \Gamma_2) \vdash a_1 : d'$ we can apply 4.35 to obtain $(\Gamma_1, \Gamma_2) \vdash a_3[a_1/y]$. Since $(\Gamma_1, \Gamma_2) \vdash a_2 : e$ and $e =_\lambda a_3[a_1/y]$ by rule *conv* we obtain $(\Gamma_1, \Gamma_2) \vdash a_2 : a_3[a_1/y]$.

  Hence by definition of typing $\Gamma_1, \Gamma_2 \vdash a : [y!d']a_3$ which implies $\Gamma_1, \Gamma_2 \vdash a : b'$ where $b \rightarrow^* b' = [y!d']a_3$.

The other cases can be shown in a similar style. □

We can now show uniqueness of types. Most cases are straightforward, except for the weakening rule where we need the context contraction result above.

Theorem 4.42 (Uniqueness of types)**.** *For all* Γ, a, b, c*:* $\Gamma \vdash a : b$ *and* $\Gamma \vdash a : c$ *implies* $b =_\lambda c$*.*

Proof. Proof by induction on the definition of $\Gamma \vdash a : b$. We show three interesting cases: In each case, if we look at a type rule $\Gamma \vdash a : b$ we have to show that if also $\Gamma \vdash a : c$ then $b =_\lambda c$.

- Rule *start*: We have $a = x$, $\Gamma = (\Gamma', x : b)$ for some Γ' and x where $(\Gamma', x : b) \vdash x : b$ and $\Gamma' \vdash b : d$ for some d: Let $(\Gamma', x : b) \vdash x : c$. By Lemma 4.30 we know that $b =_\lambda c$.
- Rule *weak*: We have $\Gamma = (\Gamma', x : d)$ for some Γ' and x where $(\Gamma', x : d) \vdash a : b$, $\Gamma' \vdash a : b$, and $\Gamma' \vdash d : e$ for some e.

  Let $(\Gamma', x : d) \vdash a : c$. Since $x \notin FV(a)$, by Lemma 4.41 we know that $\Gamma' \vdash a : c'$ for some c' with $c \to^* c'$. By inductive hypothesis $b =_\lambda c'$ which implies $b =_\lambda c$.
- Rule *appl*: We have $a = (a_1\, a_2)$ and $b = b_2[a_2/x]$ for some x, a_1, a_2, and b_2 where $\Gamma \vdash (a_1\, a_2) : b_2[a_2/x]$, $\Gamma \vdash a_1 : [x : b_1]b_2$, and $\Gamma \vdash a_2 : b_1$.

  Let $\Gamma \vdash (a_1\, a_2) : c$. By Lemma 4.32(6) we know that $c =_\lambda c_2[a_2/y]$ for some c_1, c_2 where $\Gamma \vdash a_1 : [y : c_1]c_2$ and $\Gamma \vdash a_2 : c_1$. By inductive hypothesis applied to $\Gamma \vdash a_1 : [x : b_1]b_2$ it follows that $[x : b_1]b_2 =_\lambda [y : c_1]c_2$. Hence obviously $x = y$ and by basic properties of congruence (Lemma 4.27(2)) it follows that $b_2 =_\lambda c_2$. Using Lemma 4.27(1) we can argue $b =_\lambda b_2[a_2/x] =_\lambda c_2[a_2/x] =_\lambda c$.

The other cases can be shown in a similar style. □

4.3. **Strong normalization.** The idea for the proof of strong normalization of valid expressions in d is to classify expressions according to structural properties, in order to make an inductive argument work. For this purpose we define structural skeletons called *norms* as the subset of expressions built from the primitive τ and products only and we define a partial function assigning norms to expressions. Norms are a reconstruction of a concept of *simple types*, consisting of atomic types and product types, within d[8]. Analogously to simple types, norms allow for classifying valid expressions by different degrees of structural complexity. The idea of the strong normalisation argument is first to prove that all valid expressions are *normable*, i.e they are in the domain of the norming function, and then to prove that all normable expressions are *strongly normalizable*, i.e they are not the origin of an infinite reduction sequence.

The good news is that in d we are not dealing with unconstrained parametric types as for example in System F (see *e.g.* [12]), and therefore we will be able to use more elementary methods to show strong normalisation as used for the simply typed lambda calculus. A common such method is to define a notion of *reducible* expressions satisfying certain *reducibility conditions* suitable for inductive arguments both on type structure and on reduction length, and then to prove that all reducible expressions satisfy certain *reducibility properties* including strong normalisation, and to prove that that all typable expressions are reducible ([29],[11]). We will basically adopt this idea, but the bad news is that common definitions of reducibility (*e.g.* [12]) apparently cannot be adapted to include the reduction of negation. A more suitable basis for our purposes was found to be the notion of *computable expressions* as defined in language theoretical studies of Automath [31], see also [30]. This approach is basically extended here to cover additional operators, including negation.

We motivate the basic idea of the proof (precise definitions can be found below): Consider the following condition necessary to establish strong normalization for an application $(a\, b)$

[8]The construction of norms inside d is due to convenient reuse of existing structures and definitions, norms could also be introduced as a separate mathematical structure

in the context of an inductive proof (where $\mathcal{S}$ denotes the set of strongly normalizable expressions):

- If $a \to^* [x : c_1]c_2 \in \mathcal{S}$ and $b \in \mathcal{S}$ then $c_2[b/x] \in \mathcal{S}$.

Similarly, the following condition is necessary to establish strong normalization for a negation $\neg a$ in the context of an inductive proof.

- If $a \to^* [x : c_1]c_2 \in \mathcal{S}$ then $\neg c_2 \in \mathcal{S}$.

These and other properties inspire the definition of the set of *computable* expressions C_Γ which are normable, strongly normalizable, and satisfy the property that $a, b \in C_\Gamma$ implies $\neg a, (a\,b) \in C_\Gamma$. Unfortunately, the closure properties of computable expressions do not include existential and universal abstraction. Instead, we need to prove the stronger property that normable expressions are computable under norm-preserving substitutions of their free variables to computable expressions.

This property implies that all normable expressions are computable and therefore that normability and computability are equivalent notions. The logical relations between the various notions can be summarized as follows ($\Gamma \models a$ denotes normability of a under Γ):

$$\Gamma \vdash a \quad \Rightarrow \quad \Gamma \models a \quad \Leftrightarrow \quad a \in C_\Gamma \quad \Rightarrow \quad a \in \mathcal{S}$$

We begin with some basic definitions and properties related to strong normalization.

Definition 4.43 (Strongly normalizable expressions)**.** The set of *strongly normalizable expressions* is denoted by $\mathcal{S}$. An expression a is in $\mathcal{S}$ iff there is no infinite sequence of one-step reductions $a \to a_1 \to a_2 \to a_3 \ldots$

Lemma 4.44 (Basic properties of strongly normalizable expressions)**.** *For all a, b, c, x:*

(1) *$a[b/x] \in \mathcal{S}$ and $b \to^* c$ imply $a[c/x] \in \mathcal{S}$*

(2) *$a, b, c \in \mathcal{S}$ implies $\oplus_x(a, b) \in \mathcal{S}$, $a.1$, $a.2 \in \mathcal{S}$, $[x \doteq a, b : c] \in \mathcal{S}$, $[a, b]$, $[a + b]$, $[a, : b]$, $[: a, b]$, and $[a \mathbin{?} b] \in \mathcal{S}$.*

Proof. These properties can be shown through a proof by contradiction using elementary properties of reduction (Lemmas 4.5 and 4.6). □

Next we show conditions under which applications and negations are strongly normalizing.

Lemma 4.45 (Strong normalization conditions)**.**

(1) *For all $a, b \in \mathcal{S}$: $(a\,b) \in \mathcal{S}$ if for any c_1, c_2, d_1, d_2, and x, the following conditions are satisfied:*

(C_1) *$a \to^* \oplus_x(c_1, c_2)$ implies that $c_2[b/x] \in \mathcal{S}$*

(C_2) *$a \to^* [c_1 \mathbin{?} c_2]$ and $b \to^* [d_1, : d_2]$ implies $(c_1\,d_1) \in \mathcal{S}$*

(C_3) *$a \to^* [c_1 \mathbin{?} c_2]$ and $b \to^* [: d_1, d_2]$ implies $(c_2\,d_2) \in \mathcal{S}$*

(2) *For all $a \in \mathcal{S}$: $\neg a \in \mathcal{S}$ if for any b and c, the following conditions are satisfied:*

(C_1) *$a \to^* [b \oplus c]$ implies $\neg b$, $\neg c \in \mathcal{S}$*

(C_2) *$a \to^* \oplus_x(b, c)$ implies $\neg c \in \mathcal{S}$*

Proof. These properties can be shown through a proof by contradiction using elementary properties of reduction (Lemmas 4.5, 4.6, and 4.7) and of strong normalisation (Lemma 4.44). □

Norms are a subset of expressions representing structural skeletons of expressions. Norms play an important role to classify expressions in the course of the proof of strong normalization.

Definition 4.46 (Norm, norming, normable expressions)**.** The set of norms $\bar{\mathcal{E}}$ is generated by the following rules

$$\bar{\mathcal{E}} \ ::= \ \tau \mid [\bar{\mathcal{E}}, \bar{\mathcal{E}}]$$

Obviously $\bar{\mathcal{E}} \subset \mathcal{E}$. We will use the notation $\bar{a}$, $\bar{b}$, $\bar{c}$, ... to denote norms. The partial *norming* function $\|a\|_\Gamma$ defines for some expression a the norm of a under a context Γ. It is defined by the equations in Table 8. The partial norming function is well-defined, in the sense that

$$\begin{array}{rcl}
\|\tau\|_\Gamma & = & \tau \\
\|x\|_\Gamma & = & \|\Gamma(x)\|_\Gamma \quad \text{if } \Gamma(x) \text{ is defined} \\
\|\oplus_x(a,b)\|_\Gamma & = & [\, \|a\|_\Gamma,\ \|b\|_{\Gamma,x:a}] \\
\|(a\,b)\|_\Gamma & = & \bar{c} \quad \text{if} \quad \|a\|_\Gamma = [\, \|b\|_\Gamma, \bar{c}] \\
\|[x \dot{=} a, b : c]\|_\Gamma & = & [\, \|a\|_\Gamma,\ \|b\|_\Gamma] \quad \text{if} \quad \|b\|_\Gamma = \|c\|_{\Gamma,x:a} \\
\|[a \oplus b]\|_\Gamma & = & [\, \|a\|_\Gamma,\ \|b\|_\Gamma] \\
\|a.1\|_\Gamma & = & \bar{a} \quad \text{if} \quad \|a\|_\Gamma = [\bar{a}, \bar{b}] \\
\|a.2\|_\Gamma & = & \bar{b} \quad \text{if} \quad \|a\|_\Gamma = [\bar{a}, \bar{b}] \\
\|[a, : b]\|_\Gamma & = & [\, \|a\|_\Gamma,\ \|b\|_\Gamma] \\
\|[: a, b]\|_\Gamma & = & [\, \|a\|_\Gamma,\ \|b\|_\Gamma] \\
\|[a \,?\, b]\|_\Gamma & = & [[\bar{a}, \bar{b}], \bar{c}] \quad \text{if} \quad \|a\|_\Gamma = [\bar{a}, \bar{c}], \ \ \|b\|_\Gamma = [\bar{b}, \bar{c}] \\
\|\neg a\|_\Gamma & = & \|a\|_\Gamma
\end{array}$$

Table 8: Norming

one can show by structural induction on a that, if defined, $\|a\|_\Gamma$ is unique. An expression a is normable relative under context Γ iff $\|a\|_\Gamma$ is defined. This is written as $\Gamma \models a$. Similarly to typing we use the notation $\Gamma \models a_1, \ldots, a_n$ as an abbreviation for $\Gamma \models a_1$, ..., $\Gamma \models a_n$.

Remark 4.47 (Examples)**.** There are valid and invalid normable expressions and there are strongly normalisable expression which are neither valid nor normable. We present some examples. We will show later that all valid expressions are normable (4.52) and that all normable expressions are strongly normalizable (4.65 and 4.66). Let $\Gamma = (p, q : \tau,\ z : [x : p][y : q]\tau,\ w : [x : \tau]x)$. Consider the expression $[x : p](z\,x)$:

- We have $\Gamma \vdash [x : p](z\,x)$.
- We have $\Gamma \models [x : p](z\,x)$ since $\|\,[x : p](z\,x)\,\|_\Gamma = [\tau,\ \|\,(z\,x)\,\|_{\Gamma,x:p}] = [\tau, [\tau, \tau]]$. The latter equality is true since $\|z\|_{\Gamma,x:p} = \|[x : p][y : q]\tau\|_{\Gamma,x:p} = [\tau, [\tau, \tau]]$ and $\|x\|_{\Gamma,x:p} = \|p\|_{\Gamma,x:p} = \tau$.

Consider the expression $[x : p](z\,p)$:

- We do not have $\Gamma \vdash [x : p](z\,p)$ since we do not have $\Gamma \vdash p : p$.
- We have $\Gamma \models [x : p](z\,p)$ since $\|[x : p](z\,p)\|_\Gamma = [\tau, [\tau, \tau]]$ and $\|p\|_{\Gamma,x:p} = \tau$

As a third example consider the expression $[x : [y : \tau]y](x\,x)$:

- Obviously $[x : [y : \tau]y](x\,x) \in \mathcal{S}$.
- We do not have $\vdash [x : [y : \tau]y](x\,x)$ since the application $(x\,x)$ cannot be typed.
- We do not have $\models [x : [y : \tau]y](x\,x)$: The definition of norming leads us to the expression $\|(x\,x)\|_{x:[y:\tau]y}$ which is not defined since this would require that $\|x\|_{x:[y:\tau]y} = [\, \|x\|_{x:[y:\tau]y}, \bar{a}]$ for some $\bar{a}$. Hence the norming condition for application is violated.

We show several basic properties of normable expressions culminating in the property that all valid expressions are normable. Some of these properties and proofs are structurally similar to the corresponding ones for valid expressions. However, due to the simplicity of norms, the proofs are much shorter.

Lemma 4.48 (Norm equality in context)**.** *Let* $\Gamma_a = (\Gamma_1, x : a, \Gamma_2)$ *and* $\Gamma_b = (\Gamma_1, x : b, \Gamma_2)$ *for some* $\Gamma_1, \Gamma_2, x, a, b$. *For all* c*: If* $\Gamma_1 \models a, b$, $\|a\|_{\Gamma_1} = \|b\|_{\Gamma_1}$, *and* $\Gamma_a \models c$ *then* $\Gamma_b \models c$ *and* $\|c\|_{\Gamma_a} = \|c\|_{\Gamma_b}$.

Proof. The straightforward proof is by structural induction on c. □

Lemma 4.49 (Substitution and norming)**.** *Let* $\Gamma_a = (\Gamma_1, x : a, \Gamma_2)$ *and* $\Gamma_b = (\Gamma_1, \Gamma_2[b/x])$ *for some* $\Gamma_1, \Gamma_2, x, a, b$. *For all* c*: If* $\Gamma_1 \models a, b$, $\|a\|_{\Gamma_1} = \|b\|_{\Gamma_1}$, *and* $\Gamma_a \models c$ *then* $\Gamma_b \models c[b/x]$ *and* $\|c\|_{\Gamma_a} = \|c[b/x]\|_{\Gamma_b}$.

Proof. The proof is by structural induction on c and follows from the definition of norming and of substitution. □

Lemma 4.50 (Reduction preserves norms)**.** *For all* Γ, a, b*:* $\Gamma \models a$ *and* $a \to^* b$ *implies* $\|a\|_\Gamma = \|b\|_\Gamma$

Proof. It is obviously sufficient to show the property for single-step reduction which we do here by induction on the definition of single-step reduction. The proof is straightforward, *e.g.* in case of axiom β_1, we have $a = ([x : a_1]a_2\, a_3)$, $b = a_2[a_3/x]$, and $\|([x : a_1]a_2\, a_3)\|_\Gamma = \|a_2\|_{\Gamma,x:a_1}$ where $\|a_1\|_\Gamma = \|a_3\|_\Gamma$. Therefore by Lemma 4.49 we know that $\|a_2\|_{\Gamma,x:a_1} = \|a_2[a_3/x]\|_\Gamma$ which implies the proposition.

Similarly, *e.g.* in case of structural rule $\oplus_x(\_,\_)_1$ we have $a = \oplus_x(a_1, a_3)$, $b = \oplus_x(a_2, a_3)$, and $a_1 \to a_2$. By inductive hypothesis $\|a_1\|_\Gamma = \|a_2\|_\Gamma$. By Lemma 4.48 we know that $\|a_3\|_{\Gamma,x:a_1} = \|a_3\|_{\Gamma,x:a_2}$. Therefore $\|a\|_\Gamma = [\, \|a_1\|_\Gamma,\ \|a_3\|_{\Gamma,x:a_1}] = [\, \|a_2\|_\Gamma,\ \|a_3\|_{\Gamma,x:a_2}] = \|b\|_\Gamma$. □

Lemma 4.51 (Context extension)**.** *For all* $\Gamma_1, \Gamma_2, x, a, b$ *where* $x \notin dom(\Gamma_1, \Gamma_2)$*:* $\Gamma_1, \Gamma_2 \models a$ *implies* $(\Gamma_1, x : b, \Gamma_2) \models a$ *and* $\|a\|_{\Gamma_1,\Gamma_2} = \|a\|_{\Gamma_1,x:b,\Gamma_2}$.

Proof. Obviously $\Gamma_1, \Gamma_2 \models a$ implies that $FV([\Gamma_1, \Gamma_2]a) = \emptyset$. Therefore the additional declaration $x : b$ will never be used when evaluating $\|a\|_{\Gamma_1,x:b,\Gamma_2}$. Hence the successful evaluation of $\|a\|_{\Gamma_1,\Gamma_2}$ can be easily transformed into an evaluation of $\|a\|_{\Gamma_1,x:b,\Gamma_2}$ with identical result. □

Lemma 4.52 (Typing implies normability and preserves norm)**.** *For all* Γ, a, b*: If* $\Gamma \vdash a : b$ *then* $\models [\Gamma]a, [\Gamma]b$ *and* $\|a\|_\Gamma = \|b\|_\Gamma$. *As a consequence* $\Gamma \vdash a$ *implies* $\Gamma \models a$.

Proof. Proof by induction on the definition of $\Gamma \vdash a : b$. Note that $\models [\Gamma]a$ implies $\Gamma \models a$ but not vice-versa. The stronger conclusion $\models [\Gamma]a$ is needed *e.g.* in the rule abs_U. We consider some interesting cases in detail:

- Rule *ax*: Obvious, since $a = b = \tau$ and $\Gamma = ()$.
- Rule *start*: We have $a = x$, $\Gamma = (\Gamma', x : b)$ for some Γ' and x, and $\Gamma' \vdash b : c$ for some c. By inductive hypothesis $\models [\Gamma']b$ and hence obviously $\Gamma' \models b$. Since $x \notin FV(b)$, by Law 4.51 we know that $\Gamma \models b$ and $\|b\|_\Gamma = \|b\|_{\Gamma'}$. Hence, by definition of norming $\models [\Gamma]x$, where $\|x\|_\Gamma = \|b\|_\Gamma$.
- Rule *weak*: We have $\Gamma = (\Gamma', x : c)$ for some Γ', x, and c where $\Gamma' \vdash c$ and $\Gamma' \vdash a : b$. By inductive hypothesis $\models [\Gamma']a, [\Gamma']b, [\Gamma']c$ (obviously this implies $\Gamma' \models a, b, c$) and $\|a\|_{\Gamma'} = \|b\|_{\Gamma'}$. Since $x \notin FV(a) \cup FV(b)$, by Law 4.51 we know that $\Gamma \models a$ and $\|a\|_\Gamma = \|a\|_{\Gamma'}$ as well as $\Gamma \models b$ and $\|b\|_\Gamma = \|b\|_{\Gamma'}$. Hence by definition of norming $\models [\Gamma]a, [\Gamma]b$ and $\|a\|_\Gamma = \|b\|_\Gamma$.

- Rule *conv*: We have $\Gamma \vdash a : b$ where $\Gamma \vdash a : c$, $c =_\lambda b$, and $\Gamma \vdash b : d$ for some c and d (note that we are here using the rule *conv* with b and c exchanged). By inductive hypothesis $\models [\Gamma]a, [\Gamma]b, [\Gamma]c$ (obviously this implies $\Gamma \models a, b, c$) and $\|a\|_\Gamma = \|c\|_\Gamma$. Hence by Laws 4.26 and 4.50 we know that $\|c\|_\Gamma = \|b\|_\Gamma$ which implies $\|a\|_\Gamma = \|b\|_\Gamma$.
- Rule abs_U: We have $a = [x : c]a_1$ and $b = [x : c]a_2$, for some c, a_1, and a_2 where $(\Gamma, x : c) \vdash a_1 : a_2$. By inductive hypothesis we know that $\models [\Gamma, x : c]a_1, [\Gamma, x : c]a_2$ (which can also be written as $\models [\Gamma][x : c]a_1, [\Gamma][x : c]a_2$) and $\|a_1\|_{\Gamma,x:c} = \|a_2\|_{\Gamma,x:c}$. Furthermore, *e.g.* $\models [\Gamma][x : c]a_1$ implies $\Gamma \models c$ and therefore by definition of norming $\|a\|_\Gamma = [\,\|c\|_\Gamma,\ \|a_1\|_{\Gamma,x:c}] = [\,\|c\|_\Gamma,\ \|a_2\|_{\Gamma,x:c}] = \|b\|_\Gamma$.
- Rule *appl*: We have $a = (a_1\, a_2)$ and $b = c_2[a_2/x]$, for some a_1, a_2, x, and c_2 where $\Gamma \vdash a_1 : [x : c_1]c_2$ and $\Gamma \vdash a_2 : c_1$ for some c_1. By inductive hypothesis we know that $\models [\Gamma]a_1, [\Gamma]a_2, [\Gamma][x : c_1]c_2, [\Gamma]c_1$ (obviously this implies $\Gamma \models a_1, a_2, [x : c_1]c_2, c_1$) and $\|\,[x : c_1]c_2\,\|_\Gamma = \|\,a_1\,\|_\Gamma$ as well as $\|\,c_1\,\|_\Gamma = \|\,a_2\,\|_\Gamma$. Hence $\|\,a_1\,\|_\Gamma = [\,\|\,a_2\,\|_\Gamma,\ \|\,c_2\,\|_{\Gamma,x:c_1}]$ and therefore by definition of norming $\|(a_1\, a_2)\|_\Gamma = \|c_2\|_{\Gamma,x:c_1}$ which implies $\Gamma \models a$ and (obviously) $\models [\Gamma]a$. Since $\|\,a_2\,\|_\Gamma = \|\,c_1\,\|_\Gamma$ we can apply Law 4.49 to obtain $\|\,c_2\,\|_{\Gamma,x:c_1} = \|\,c_2[a_2/x]\,\|_\Gamma$. Hence $\Gamma \models b$ and (obviously) $\models [\Gamma]b$ and $\|a\|_\Gamma = \|(a_1\, a_2)\|_\Gamma = \|c_2[a_2/x]\|_\Gamma = \|b\|_\Gamma$.

For the consequence, $\Gamma \vdash a$ means that $\Gamma \vdash a : b$, for some b. By the property just shown this implies $\Gamma \models a$. □

We introduce a norm-based induction principle that we will use several times in the following proofs.

Definition 4.53 (Induction on the size of norms)**.** The *size* of a norm $\bar{a}$ is defined as the number of primitive constants τ it contains. A property $P(\Gamma, \bar{a})$ is shown by *norm-induction* iff for all $\bar{b}$ we know that: If $P(\Gamma, \bar{c})$ for all $\bar{c}$ of size strictly smaller that $\bar{b}$ then $P(\Gamma, \bar{b})$. This can be reformulated into a more convenient form for its use in proofs.

- **Inductive base:** $P(\Gamma, \tau)$.
- **Inductive step:** For all $\bar{b}, \bar{c}$: If $P(\Gamma, \bar{a})$ for all $\bar{a}$ of size strictly smaller than the size of $[\bar{b}, \bar{c}]$ then $P(\Gamma, [\bar{b}, \bar{c}])$.

Computable expressions are organized according to norm structure and satisfy a number of additional conditions necessary for an inductive proof of strong normalization.

Definition 4.54 (Computable expressions)**.** The set of *computable expressions* of norm $\bar{a}$ under context Γ is denoted by $C_\Gamma(\bar{a})$. $a \in C_\Gamma(\bar{a})$ iff $a \in \mathcal{S}$, $\Gamma \models a$ where $\|a\|_\Gamma = \bar{a}$, and if $\bar{a} = [\bar{b}, \bar{c}]$ for some $\bar{b}$ and $\bar{c}$ then the following *computability conditions* are satisfied:

(α) For all x, b, c: If $a \rightarrow^* \oplus_x(b, c)$ or $\neg a \rightarrow^* \oplus_x(b, c)$ then $c \in C_{\Gamma,x:b}(\bar{c})$ and $c[d/x] \in C_\Gamma(\bar{c})$ for any $d \in C_\Gamma(\bar{b})$.
(β) For all b, c: If $a \rightarrow^* [b \oplus c]$ or $\neg a \rightarrow^* [b \oplus c]$ then $b \in C_\Gamma(\bar{b})$ and $c \in C_\Gamma(\bar{c})$.
(γ) For all x, b, c, d: $a \rightarrow^* [x \doteq b, c : d]$ implies both $b \in C_\Gamma(\bar{b})$ and $c \in C_\Gamma(\bar{c})$, $a \rightarrow^* [b, : d]$ implies $b \in C_\Gamma(\bar{b})$, and $a \rightarrow^* [: d, c]$ implies $c \in C_\Gamma(\bar{c})$.
(δ) If $\bar{b} = [\bar{b}_1, \bar{b}_2]$, for some $\bar{b}_1$ and $\bar{b}_2$, then, for all b_1, b_2: $a \rightarrow^* [b_1 \,?\, b_2]$ implies both $b_1 \in C_\Gamma([\bar{b}_1, \bar{c}])$ and $b_2 \in C_\Gamma([\bar{b}_2, \bar{c}])$.

Remark 4.55 (Motivation for the computability conditions)**.** The four conditions are motivated by the strong normalization condition for applications (Lemma 4.45(1)) and negations (Lemma 4.45(2)).

Lemma 4.56 (Computable expressions are well-defined for all norms)**.** *For all* Γ *and* $\bar{a}$*, the set* $C_\Gamma(\bar{a})$ *exists and is well-defined.*

Proof. Proof by norm-induction on $\bar{a}$ follows directly from the definition of computable expressions. □

We begin with some basic properties of computable expressions.

Lemma 4.57 (Basic properties of computable expressions)**.** *For all* $\Gamma, \Gamma_1, \Gamma_2, a, a_1, a_2, b, x$*:*
(1) $a \in C_\Gamma(\|a\|_\Gamma)$ *and* $a \to^* b$ *imply* $b \in C_\Gamma(\|a\|_\Gamma)$*.*
(2) $\Gamma_1 \models a_1$*,* $a \in C_{\Gamma_1, x:a_1, \Gamma_2}(\bar{a})$*, and* $a_1 \to^* a_2$ *imply* $a \in C_{\Gamma_1, x:a_2, \Gamma_2}(\bar{a})$*.*

Proof. For (1), we assume that $a \in C_\Gamma(\bar{a})$ where $\|a\|_\Gamma = \bar{a}$. Obviously $a \in \mathcal{S}$ and therefore also $b \in \mathcal{S}$ and by Lemma 4.50 we obtain $\|b\|_\Gamma = \|a\|_\Gamma$. We have to show that $b \in C_\Gamma(\bar{a})$: It is easy to prove the computability conditions, since from $b \to^* c$ we can always infer $a \to^* c$ and hence use the corresponding condition from the assumption $a \in C_\Gamma(\bar{a})$.

For (2) the proof is by norm-induction on $\bar{a}$ and only requires properties 4.48 and 4.50 of norming. □

The closure of computable expressions against negation is shown first due to its necessity in other monotonicity arguments.

Lemma 4.58 (Computable expressions are closed against negation)**.** *For all* Γ, a*:* $a \in C_\Gamma(\|a\|_\Gamma)$ *implies* $\neg a \in C_\Gamma(\|a\|_\Gamma)$*.*

Proof. Assume that $a \in C_\Gamma(\|a\|_\Gamma)$. Let $\bar{a} = \|a\|_\Gamma$. By definition of norming obviously $\|\neg a\|_\Gamma = \bar{a}$. We show that $a \in C_\Gamma(\bar{a})$ implies $\neg a \in C_\Gamma(\bar{a})$ by norm-induction on $\bar{a}$.

Inductive base: We have $\bar{a} = \tau$, therefore the computability conditions become trivial and it remains to show that $\neg a \in \mathcal{S}$. Since $a \in \mathcal{S}$, we can apply Lemma 4.45(2) whose conditions (C_1) and (C_2) become trivial since by definition of norming and Lemma 4.50 they imply that $\bar{a} \neq \tau$.

Inductive step: Let $\bar{a} = [\bar{b}, \bar{c}]$ for some $\bar{b}$ and $\bar{c}$. First, we show that $\neg a \in \mathcal{S}$. Since $a \in \mathcal{S}$, according to Lemma 4.45(2), for any x, b, c, we need to show conditions (C_1) and (C_2):

- (C_1): Let $a \to^* [b \oplus c]$. Since $a \in C_\Gamma(\bar{a})$, by computability condition β we know that $b \in C_\Gamma(\bar{b})$ and $c \in C_\Gamma(\bar{c})$. By inductive hypothesis $\neg b \in C_\Gamma(\bar{b})$ and $\neg c \in C_\Gamma(\bar{c})$ and therefore obviously $\neg b, \neg c \in \mathcal{S}$.
- (C_2): Let $a \to^* \oplus_x(b, c)$. Since $a \in C_\Gamma(\bar{a})$, by computability condition α we know that $c \in C_{\Gamma, x:b}(\bar{c})$. By inductive hypothesis $\neg c \in C_{\Gamma, x:b}(\bar{c})$ and therefore obviously $\neg c \in \mathcal{S}$.

Therefore we know that $\neg a \in \mathcal{S}$. It remains to show the computability conditions for $\neg a$:

(α) We have to consider two cases:

(1) $\neg a \to^* [x : a_1]a_2$ or $\neg a \to^* [x!a_1]a_2$ for some x, a_1, and a_2. By Lemma 4.7(1,3) we know that $a \to^* \oplus_x(a_1', a_2')$ for some a_1' and a_2' where $a_1' \to^* a_1$ and $\neg a_2' \to^* a_2$. The first part of α can be argued as follows:

$$\begin{array}{ll} & a \in C_\Gamma([\bar{b}, \bar{c}]) \\ \Rightarrow & \quad \text{(by } \alpha\text{, since } a \to^* \oplus_x(a_1', a_2')) \\ & a_2' \in C_{\Gamma, x:a_1'}(\bar{c}) \\ \Rightarrow & \quad \text{(inductive hypothesis)} \\ & \neg a_2' \in C_{\Gamma, x:a_1'}(\bar{c}) \\ \Rightarrow & \quad \text{(by Lemma 4.57(1,2), since } a_1' \to^* a_1 \text{ and } \neg a_2' \to^* a_2) \\ & a_2 \in C_{\Gamma, x:a_1}(\bar{c}) \end{array}$$

For the second clause, for any $d \in C_\Gamma(\bar{b})$, we can argue as follows:

$$\begin{array}{ll} & a \in C_\Gamma([\bar{b},\bar{c}]) \\ \Rightarrow & \quad \text{(by } \alpha\text{, since } a \to^* \oplus_x(a_1', a_2')) \\ & a_2'[d/x] \in C_\Gamma(\bar{c}) \\ \Rightarrow & \quad \text{(inductive hypothesis)} \\ & \neg(a_2'[d/x]) \in C_\Gamma(\bar{c}) \\ \Rightarrow & \quad \text{(definition of substitution)} \\ & (\neg a_2')[d/x] \in C_\Gamma(\bar{c}) \\ \Rightarrow & \quad \text{(by Lemmas 4.57(1) and 4.5, since } \neg a_2' \to^* a_2) \\ & a_2[d/x] \in C_\Gamma(\bar{c}) \end{array}$$

(2) $\neg\neg a \to^* [x : a_1]a_2$ or $\neg\neg a \to^* [x!a_1]a_2$ for some x, a_1, and a_2. By Lemma 4.7(1,3) we know that $\neg a \to^* \oplus_x(a_1', a_2')$ for some a_1' and a_2' where $a_1' \to^* a_1$ and $\neg a_2' \to^* a_2$. Applying Lemma 4.7(1,3) again we know that $a \to^* \oplus_x'(a_1'', a_2'')$ for some a_1'' and a_2'' where $a_1'' \to^* a_1'$ and $\neg a_2'' \to^* a_2'$. This means that $a_1'' \to^* a_1$ and $\neg\neg a_2'' \to^* a_2$. This case can then be argued similarly to the first case just applying the inductive hypothesis twice.

(β) Similarly to α, we have to consider two cases:

(1) $\neg a \to^* [a_1, a_2]$ or $\neg a \to^* [a_1 + a_2]$ for some a_1 and a_2. By Lemma 4.7(2,4) we know that $a \to^* [a_1' \oplus a_2']$ for some a_1' and a_2' where $\neg a_1' \to^* a_1$ and $\neg a_2' \to^* a_2$.
Since $a \in C_\Gamma(\bar{a})$ and $a \to^* [a_1' \oplus a_2']$ by computability condition β we obtain $a_1' \in C_\Gamma(\bar{b})$ and $a_2' \in C_\Gamma(\bar{c})$. By inductive hypotheses we know that also $\neg a_1' \in C_\Gamma(\bar{b})$ and $\neg a_2' \in C_\Gamma(\bar{c})$. By Lemma 4.57(1) we get $a_1 \in C_\Gamma(\bar{b})$ and $a_2 \in C_\Gamma(\bar{c})$.

(2) $\neg\neg a \to^* [a_1, a_2]$ or $\neg\neg a \to^* [a_1 + a_2]$ for some a_1 and a_2. By Lemma 4.7(2,4) we know that $\neg a \to^* [a_1' \oplus a_2']$ for some a_1' and a_2' where $\neg a_1' \to^* a_1$ and $\neg a_2' \to^* a_2$. Applying Lemma 4.7(2,4) again we know that $a \to^* [a_1'' \oplus' a_2'']$ for some a_1'' and a_2'' where $\neg a_1'' \to^* a_1'$ and $\neg a_2'' \to^* a_2'$. This means that $\neg\neg a_1'' \to^* a_1$ and $\neg\neg a_2'' \to^* a_2$.
Since $a \in C_\Gamma(\bar{a})$ and $a \to^* [a_1'' \oplus' a_2'']$ by computability condition β we obtain $a_1'' \in C_\Gamma(\bar{b})$ anf $a_2'' \in C_\Gamma(\bar{c})$. By inductive hypotheses (applied twice) we know that also $\neg\neg a_1' \in C_\Gamma(\bar{b})$ and $\neg\neg a_2' \in C_\Gamma(\bar{c})$. By Lemma 4.57(1) we get $a_1 \in C_\Gamma(\bar{b})$ and $a_2 \in C_\Gamma(\bar{c})$.

(γ) We have three cases: First, if $\neg a \to^* [x \doteq a_1, a_2 : a_3]$ for some x, a_1, a_2, a_3, then by Lemma 4.7(5) we know that $a \to^* [x \doteq a_1, a_2 : a_3]$. Since $a \in C_\Gamma([\bar{b},\bar{c}])$, by computability condition γ, we know that $a_1 \in C_\Gamma(\bar{b})$, $a_2 \in C_\Gamma(\bar{c})$. Second, if $\neg a \to^* [a_1, : a_2]$for some a_1 and a_2, then by Lemma 4.7(7) we know that $a \to^* [a_1, : a_2]$. Since $a \in C_\Gamma([\bar{b},\bar{c}])$, by computability condition γ, we know that $a_1 \in C_\Gamma(\bar{b})$. The third case $\neg a \to^* [: a_1, a_2]$ is shown in a similar way.

(δ) Let $\bar{b} = [\bar{b}_1, \bar{b}_2]$ for some $\bar{b}_1$ and $\bar{b}_2$. Let $\neg a \to^* [a_1 \,?\, a_2]$ for some a_1 and a_2. By Lemma 4.7(6) we know that $a \to^* [a_1 \,?\, a_2]$. By computability condition δ we know that $a_1 \in C_\Gamma([\bar{b}_1, \bar{c}])$ and $a_2 \in C_\Gamma([\bar{b}_2, \bar{c}])$. □

Closure of computability against application has a proof that is not very difficult but somewhat lengthy due to a repetition of similar arguments for the different computability conditions.

Lemma 4.59 (Closure of computable expressions against application)**.** *For all* Γ*,* a*,* b*:* $\Gamma \models (a\,b)$*,* $a \in C_\Gamma(\,\|a\|_\Gamma)$*, and* $b \in C_\Gamma(\,\|b\|_\Gamma)$ *implies* $\|a\|_\Gamma = [\,\|b\|_\Gamma, \bar{c}]$ *and* $(a\,b) \in C_\Gamma(\bar{c})$ *for some* $\bar{c}$*.*

Proof. Assume that $\Gamma \models (a\,b)$, $a \in C_\Gamma(\,\|a\|_\Gamma)$, and $b \in C_\Gamma(\,\|b\|_\Gamma)$. Let $\bar{a} = \|a\|_\Gamma$ and $\bar{b} = \|b\|_\Gamma$. $\Gamma \models (a\,b)$ implies that $\bar{a} = [\bar{b}, \bar{c}]$ for some $\bar{c}$. By norm-induction on $\bar{a}$ we will show that $a \in C_\Gamma(\bar{a})$ and $b \in C_\Gamma(\bar{b})$ implies $(a\,b) \in C_\Gamma(\bar{c})$.

The inductive base is trivial since $\bar{a} \neq \tau$. For the inductive step we first need to show that $(a\,b) \in \mathcal{S}$. By Lemma 4.45(1) we have to show conditions (C_1), (C_2), and (C_3). For any x, b_1, c_1, c_2, d_1, d_2:

- (C_1): Let $a \to^* \oplus_x(b_1, c_1)$. Since $a \in C_\Gamma([\bar{b}, \bar{c}])$, by computability condition α, for any $d \in C_\Gamma(\bar{b})$ we know that $c_1[d/x] \in C_\Gamma(\bar{c})$. Hence also $c_1[b/x] \in C_\Gamma(\bar{c})$ and therefore obviously $c_1[b/x] \in \mathcal{S}$ hence condition (C_1) is satisfied.
- (C_2): Let $a \to^* [c_1 \,?\, c_2]$ and $b \to^* [d_1, : d_2]$. From $b \to^* [d_1, : d_2]$, by Lemma 4.50 and by definition of norming we know that $\bar{b} = [\bar{d}_1, \bar{d}_2]$, for some $\bar{d}_1$, $\bar{d}_2$, and hence $\bar{a} = [[\bar{d}_1, \bar{d}_2], \bar{c}]$. Since $b \in C_\Gamma([\bar{d}_1, \bar{d}_2])$ by computability condition γ we know that $d_1 \in C_\Gamma(\bar{d}_1)$. Since $a \in C_\Gamma([[\bar{d}_1, \bar{d}_2], \bar{c}])$ by computability condition δ we know that $c_1 \in C_\Gamma([\bar{d}_1, \bar{c}])$, $c_2 \in C_\Gamma([\bar{d}_2, \bar{c}])$.

  By inductive hypothesis (the size of $[\bar{d}_1, \bar{c}]$ is strictly smaller than that of $\bar{a}$), we know that $(c_1\,d_1) \in C_\Gamma(\bar{c})$. By definition of computability therefore $(c_1\,d_1) \in \mathcal{S}$.
- (C_3): Proof is similar to (C_2).

Therefore by Lemma 4.45(1) we know that $(a\,b) \in \mathcal{S}$ It remains to show the computability conditions for $(a\,b)$. Let $\bar{c} = [\bar{d}, \bar{e}]$ for some $\bar{d}$ and $\bar{e}$:

(α) If $(a\,b) \to^* \oplus_x(a_1, a_2)$ or $\neg(a\,b) \to^* \oplus_x(a_1, a_2)$, for some a_1 and a_2 then, since $\Gamma \models (a\,b)$, by Lemma 4.50 we have

$$\|\oplus_x(a_1, a_2)\|_\Gamma = [\,\|a_1\|_\Gamma,\ \|a_2\|_{\Gamma, x:a_1}] = [\bar{d}, \bar{e}]$$

Therefore $\|a_1\|_\Gamma = \bar{d}$ and $\|a_2\|_{\Gamma,x:a_1} = \bar{e}$. We have to show that $a_2 \in C_{\Gamma,x:a_1}(\bar{e})$ and that $a_2[d/x] \in C_\Gamma(\bar{e})$ for any $d \in C_\Gamma(\bar{d})$. We have to distinguish two cases:

(1) If $(a\,b) \to^* \oplus_x(a_1, a_2)$ then by Lemma 4.6(5) we know that there are two subcases

  (a) $a \to^* [y : a_3]a_4$, $b \to^* b'$, and $a_4[b'/y] \to^* \oplus_x(a_1, a_2)$ for some y, a_3, a_4, and b'. We can argue as follows:

$$\begin{aligned} & a \in C_\Gamma([\bar{b}, \bar{c}]) \\ \Rightarrow\ & (\text{by } \alpha\text{, since by Lemma 4.57(1) we know that } b' \in C_\Gamma(\bar{b})) \\ & a_4[b'/x] \in C_\Gamma(\bar{c}) \\ \Rightarrow\ & (\text{by Lemma 4.57(1), since } a_4[b'/y] \to^* \oplus_x(a_1, a_2)) \\ & \oplus_x(a_1, a_2) \in C_\Gamma(\bar{c}) \end{aligned}$$

  (b) $a \to^* [c_1 \,?\, c_2]$ and either $b \to^* [b_1, : b_2]$ and $(c_1\,b_1) \to^* \oplus_x(a_1, a_2)$ or $b \to^* [: b_1, b_2]$ and $(c_2\,b_2) \to^* \oplus_x(a_1, a_2)$ for some c_1, c_2, b_1, and b_2. This means that $\bar{b} = [\bar{b}_1, \bar{b}_2]$ for some $\bar{b}_1$ and $\bar{b}_2$ where $\|c_1\|_\Gamma = [\bar{b}_1, \bar{c}]$ and $\|c_2\|_\Gamma = [\bar{b}_2, \bar{c}]$. Since $\Gamma \models (a\,b)$ we also know that $\|b_1\|_\Gamma = \bar{b}_1$ and $\|b_2\|_\Gamma = \bar{b}_2$. We will show the first case $b \to^* [b_1, : b_2]$, the proof of the second case is similar.

  From $b \to^* [b_1, : b_2]$, by computability condition γ we know that $b_1 \in C_\Gamma(\bar{b}_1)$. From $a \to^* [c_1 \,?\, c_2]$, by computability condition δ we know that $c_1 \in C_\Gamma([\bar{b}_1, \bar{c}])$ and $c_2 \in C_\Gamma([\bar{b}_2, \bar{c}])$. Since the size of $[\bar{b}_1, \bar{c}]$ is strictly smaller than the size of $\bar{a}$

we can apply the inductive hypothesis to obtain $(c_1\, b_1) \in C_\Gamma(\bar{c})$. Hence, since $(c_1\, b_1) \to^* \oplus_x(a_1, a_2)$, by Lemma 4.57(1) we have $\oplus_x(a_1, a_2) \in C_\Gamma(\bar{c})$.

Therefore in both cases we have shown $\oplus_x(a_1, a_2) \in C_\Gamma(\bar{c})$. From computability condition α we obtain $a_2 \in C_{\Gamma,x:a_1}(\bar{e})$ and $a_2[d/x] \in C_\Gamma(\bar{e})$.

(2) If $\neg(a\, b) \to^* \oplus_x(a_1, a_2)$, then we show the case $\oplus_x(a_1, a_2) = [x : a_1]a_2$ (the case $\oplus_x(a_1, a_2) = [x!a_1]a_2$ runs analogously): By Lemma 4.7(1) we know that $(a\, b) \to^* [x!a'_1]a'_2$ for some a'_1 and a'_2 where $a'_1 \to^* a_1$ and $\neg a'_2 \to^* a_2$. By reasoning similarly to the first case we can show that $[x!a'_1]a'_2 \in C_\Gamma(\bar{c})$.

The first part of α can be seen as follows: Since $[x!a'_1]a'_2 \in C_\Gamma(\bar{c})$ we know that $a'_2 \in C_{\Gamma,x:a_1}(\bar{e})$. By Lemma 4.58, this implies $\neg a'_2 \in C_{\Gamma,x:a_1}(\bar{e})$. Therefore by Lemma 4.57(1), since we have $\neg a'_2 \to^* a_2$, we know that $a_2 \in C_{\Gamma,x:a_1}(\bar{e})$.

Let $d \in C_\Gamma(\bar{d})$. We can show the second part of α as follows:

$$\begin{array}{ll} & [x!a'_1]a'_2 \in C_\Gamma(\bar{c}) \\ \Rightarrow & (\text{by } \alpha, \text{ since } d \in C_\Gamma(\bar{d})) \\ & a'_2[d/x] \in C_\Gamma(\bar{e}) \\ \Rightarrow & (\text{by Lemma 4.58}) \\ & \neg(a'_2[d/x]) \in C_{\Gamma,x:a_1}(\bar{e}) \\ \Rightarrow & (\text{definition of substitution}) \\ & (\neg a'_2)[d/x] \in C_{\Gamma,x:a_1}(\bar{e}) \\ \Rightarrow & (\text{by Lemma 4.57(1) and 4.5, since } \neg a'_2 \to^* a_2) \\ & a_2[d/x] \in C_{\Gamma,x:a_1}(\bar{e}) \end{array}$$

(β) We have to distinguish two cases:

(1) If $(a\, b) \to^* [a_1 \oplus a_2]$ for some a_1 and a_2, then, since $\Gamma \models (a\, b)$, by Lemma 4.50 we have

$$\|(a\, b)\|_\Gamma = [\, \|a_1\|_\Gamma,\ \|a_2\|_\Gamma] = [\bar{d}, \bar{e}]$$

Similarly to the corresponding case for condition α we can show that $[a_1 \oplus a_2] \in C_\Gamma(\bar{c})$. By computability condition β we obtain $a_1 \in C_\Gamma(\bar{d})$ and $a_2 \in C_\Gamma(\bar{e})$.

(2) If $\neg(a\, b) \to^* [a_1 \oplus a_2]$ for some a_1 and a_2, then, since obviously $\Gamma \models \neg(a\ b)$, by Lemma 4.50 we have

$$\|\neg(a\, b)\|_\Gamma = [\, \|a_1\|_\Gamma,\ \|a_2\|_\Gamma] = [\bar{d}, \bar{e}]$$

We show the case $[a_1 \oplus a_2] = [a_1, a_2]$ (the case $[a_1 \oplus a_2] = [a_1 + a_2]$ runs analogously): By Lemma 4.7(2) we know that $(a\, b) \to^* [a'_1 + a'_2]$ where $\neg a'_1 \to^* a_1$ and $\neg a'_2 \to^* a_2$ for some a'_1 and a'_2. Therefore by elementary properties of reduction (Lemma 4.6(6)) there are two cases

(a) $a \to^* \oplus_x(a_3, a_4)$, $b \to^* b'$, and $\neg a_4[b'/x] \to^* [a'_1 + a'_2]$ for some x, a_3, a_4, and b'. By the same argument as in the corresponding case for condition α we can show that $[a'_1 + a'_2] \in C_\Gamma(\bar{c})$.

(b) $a \to^* [c_1 \mathbin{?} c_2]$ and either $b \to^* [b_1, : b_2]$ and $(c_1\, b_1) \to^* [a'_1 + a'_2]$ or $b \to^* [: b_1, b_2]$ and $(c_2\, b_2) \to^* [a'_1 + a'_2]$ for some c_1, c_2, b_1, and b_2. By the same argument as in the corresponding case for condition α we can show that $[a'_1 + a'_2] \in C_\Gamma(\bar{c})$.

Hence in all cases we have $[a'_1 + a'_2] \in C_\Gamma(\bar{c})$. By Lemma 4.58 we have $\neg[a'_1 + a'_2] \in C_\Gamma(\bar{c})$ and hence by Lemma 4.57(1) we have $[\neg a'_1, \neg a'_2] \in C_\Gamma(\bar{c})$ and therefore by definition of computable expressions (condition β) we obtain $\neg a'_1 \in C_\Gamma(\bar{d})$ and $\neg a'_2 \in C_\Gamma(\bar{e})$. Since $\neg a'_1 \to^* a_1$ and $\neg a'_2 \to^* a_2$ by Lemma 4.57(1) we have $a_1 \in C_\Gamma(\bar{d})$ and $a_2 \in C_\Gamma(\bar{e})$.

(γ) We have to distinguish three cases:

(1) If $(a\,b) \to^* [x \doteq a_1, a_2 : c]$ for some a_1, a_2, and c then, since $\Gamma \models (a\,b)$, by Lemma 4.50 we have

$$\|(a\,b)\|_\Gamma = [\,\|a_1\|_\Gamma,\ \|a_2\|_\Gamma] = [\bar{d}, \bar{e}]$$

Similarly to the corresponding case for condition α we can shown that $[x \doteq a_1, a_2 : c] \in C_\Gamma(\bar{c})$. By computability condition γ we know that $a_1 \in C_\Gamma(\bar{d})$ and $a_2 \in C_\Gamma(\bar{e})$.

(2) If $(a\,b) \to^* [a_1, : c]$ for some a_1 and c, then, since $\Gamma \models (a\,b)$, by Lemma 4.50 we have

$$\|(a\,b)\|_\Gamma = [\,\|a_1\|_\Gamma,\ \|c\|_\Gamma] = [\bar{d}, \bar{e}]$$

Similarly to the corresponding case for condition α we can shown that $[a_1, : c] \in C_\Gamma(\bar{c})$. By computability condition γ we obtain $a_1 \in C_\Gamma(\bar{d})$.

(3) $(a\,b) \to^* [: c, a_2]$ for some a_2 and c: This case is similar to the previous one.

(δ) Let $\bar{d} = [\bar{d_1}, \bar{d_2}]$ for some $\bar{d_1}$ and $\bar{d_2}$. If $(a\,b) \to^* [a_1 \,?\, a_2]$ for some a_1 and a_2 then, since $\Gamma \models (a\,b)$, by Lemma 4.50 we have

$$\|[a_1 \,?\, a_2]\|_\Gamma = [\bar{d}, \bar{e}]$$

where $\|\, a_1 \,\|_\Gamma = [\bar{d_1}, \bar{e}]$ and $\|\, a_2 \,\|_\Gamma = [\bar{d_2}, \bar{e}]$. Similarly to the corresponding case for condition α we can shown that $[a_1 \,?\, a_2] \in C_\Gamma(\bar{c})$. By computability condition δ we obtain $a_1 \in C_\Gamma([\bar{d_1}, \bar{e}])$ and $a_2 \in C_\Gamma([\bar{d_2}, \bar{e}])$. □

We now show the remaining closure properties of computable expressions.

Lemma 4.60 (Closure properties of computable expressions)**.** *For all* $\Gamma, x, a, b, c, \bar{a}, \bar{b}, \bar{c}$*:*

(1) $a \in C_\Gamma(\bar{a})$*,* $b \in C_\Gamma(\bar{b})$*, and* $c \in C_{\Gamma,x:a}(\bar{b})$ *implies* $[a \oplus b], [a, : b], [: a, b] \in C_\Gamma([\bar{a}, \bar{b}])$*, and* $[x \doteq a, b : c] \in C_\Gamma([\bar{a}, \bar{b}])$*.*

(2) $a \in C_\Gamma([\bar{b}, \bar{c}])$ *implies* $a.1 \in C_\Gamma(\bar{b})$ *and* $a.2 \in C_\Gamma(\bar{c})$*.*

(3) $a \in C_\Gamma([\bar{a}, \bar{c}])$ *and* $b \in C_\Gamma([\bar{b}, \bar{c}])$ *implies* $[a \,?\, b] \in C_\Gamma([[\bar{a}, \bar{b}], \bar{c}])$*.*

Proof. For all Γ, x, a, b, c, $\bar{a}$, $\bar{b}$, $\bar{c}$:

(1) Let $a \in C_\Gamma(\bar{a})$, $b \in C_\Gamma(\bar{b})$, and $c \in C_{\Gamma,x:a}(\bar{b})$. Hence $\bar{a} = \|a\|_\Gamma$ and $\bar{b} = \|b\|_\Gamma$.

First we show that $[a, b], [a+b], [a, : b], [: a, b] \in C_\Gamma([\bar{a}, \bar{b}])$. Obviously $\Gamma \models [a, b], [a+b], [a, : b], [: a, b]$ and $[a, b], [a+b], [a, : b], [: a, b] \in \mathcal{S}$. Using Lemmas 4.6, 4.7, 4.57 and 4.58, it is straightforward to show the computability conditions for these operators.

Next we turn to the case of a protected definition: From the assumptions we know that $\|\,[x \doteq a, b : c]\,\|_\Gamma = [\bar{a}, \bar{b}]$ where $\|c\|_{\Gamma,x:a} = \bar{b}$. Assume $a \in C_\Gamma(\bar{a})$, $b \in C_\Gamma(\bar{b})$, and $c \in C_{\Gamma,x:a}(\bar{b})$. We have to show $[x \doteq a, b : c] \in C_\Gamma([\bar{a}, \bar{b}])$.

Since by definition of computable expressions $a, b, c \in \mathcal{S}$, by Lemma 4.44(2) we have $[x \doteq a, b : c] \in \mathcal{S}$. It remains to show the computability conditions: By Lemmas 4.6(2) and 4.7(5), $[x \doteq a, b : c] \to^* d$ and $\neg[x \doteq a, b : c] \to^* d$, for some d, each imply $d = [x \doteq a', b' : c']$ for some a', b', and c'. Therefore, the computability conditions α, β, and δ are trivially satisfied. The condition γ is trivially satisfied except for the case $[x \doteq a, b : c] \to^* [x \doteq a', b' : c']$ for some a', b', and c'. By Lemma 4.6(2) we know that $a \to^* a'$, $b \to^* b'$, and $c \to^* c'$. By Lemma 4.57(1) we know that $a' \in C_\Gamma(\bar{a})$ and $b' \in C_\Gamma(\bar{b})$.

(2) Let $a \in C_\Gamma([\bar{b}, \bar{c}])$: From the assumptions we know that that $\|a.1\|_\Gamma = \bar{b}$ where $\|a\|_\Gamma = [\bar{b}, \bar{c}]$. Assume $a \in C_\Gamma([\bar{b}, \bar{c}])$. We have to show $a.1 \in C_\Gamma(\bar{b})$. Since $a \in \mathcal{S}$ by Lemma 4.44(2) we know that $a.1 \in \mathcal{S}$. The computability conditions for $a.1$ can be shown in a similar type of argument as for the case of application (Lemma 4.59). The case $a.2 \in C_\Gamma(\bar{c})$ can be shown in a similar style.

(3) Assume that $a \in C_\Gamma([\bar{a},\bar{c}])$ and $b \in C_\Gamma([\bar{b},\bar{c}])$. We will show that $[a\,?\,b] \in C_\Gamma([[\bar{a},\bar{b}],\bar{c}])$. Obviously $\Gamma \models [a\,?\,b]$ and $[a\,?\,b] \in \mathcal{S}$. Since case distinction or negated case distinctions always reduce to case distinctions, the conditions α, β, and γ are trivially satisfied. It remains to show the condition δ: Assume that $[a\,?\,b] \to^* [a_1\,?\,a_2]$. By Lemma 4.6(1) we know that $a \to^* a_1$ and $a \to^* a_2$. The required conclusions $a_1 \in C_\Gamma([\bar{a},\bar{c}])$ and $a_2 \in C_\Gamma([\bar{b},\bar{c}])$ follow by Lemma 4.57(1). □

Note that abstraction is missing from the properties of Lemma 4.60 since from $a \in C_{\Gamma,x:b}(\bar{a})$ it is not clear how to conclude that $a[c/x] \in C_\Gamma$ for any $c \in C_\Gamma(\,\|b\|_\Gamma)$. This inspires the following Lemma.

Lemma 4.61 (Abstraction closure for computable expressions)**.** *For all Γ, x, a, and b where $\Gamma \models \oplus_x(a,b)$: If $a \in C_\Gamma(\,\|a\|_\Gamma)$, $b \in C_{\Gamma,x:a}(\,\|b\|_{\Gamma,x:a})$, and for all c with $c \in C_\Gamma(\,\|a\|_\Gamma)$ we have $b[c/x] \in C_\Gamma(\,\|\,b\,\|_{\Gamma,x:a})$ (this last assumption about substitution is crucial) then $\oplus_x(a,b) \in C_\Gamma(\,\|\oplus_x(a,b)\|_\Gamma)$.*

Proof. Let $\bar{a} = \|a\|_\Gamma$ and $\bar{b} = \|b\|_{\Gamma,x:a}$. From $a \in C_\Gamma(\bar{a})$ and $b \in C_{\Gamma,x:a}(\bar{b})$ we obviously obtain $a, b \in \mathcal{S}$. By Lemma 4.44(2) this implies $[x : a]b \in \mathcal{S}$.

To show that $\oplus_x(a,b) \in C_\Gamma([\bar{a},\bar{b}])$, it remains to show the computability conditions for $\oplus_x(a,b)$: The computability conditions β, γ, and δ are trivially satisfied since (negated) abstractions reduce to abstractions only. It remains to show the condition α: By Lemma 4.6(2), only the cases $[x : a]b \to^* [x : c]d$, $[x!a]b \to^* [x!c]d$, $\neg[x : a]b \to^* [x!c]d$, and $\neg[x!a]b \to^* [x : c]d$ for some c and d are possible. By Lemmas 4.6(2) and 4.7(1,3) we know that $a \to^* c$ and either $b \to^* d$ or $\neg b \to^* d$. We have to show the following properties:

- We have to show $d \in C_{\Gamma,x:c}(\bar{b})$: From the assumption $b \in C_{\Gamma,x:a}(\bar{b})$ either by directly using Lemma 4.57(1) or by first applying Lemma 4.58 we obtain $d \in C_{\Gamma,x:a}(\bar{b})$. Since $a \to^* c$, by Lemma 4.57(2) we obtain that $d \in C_{\Gamma,x:c}(\bar{b})$.
- Let $e \in C_\Gamma(\bar{a})$. We have to show that $d[e/x] \in C_\Gamma(\bar{b})$ which follows from the assumption about substitution (by instantiating c to e). □

To prove computability of all normable expressions, due to Lemma 4.61, we need to prove the stronger property that normable expressions are computable under any substitution of their free variables to computable expressions. First we need to extend the notion of substitution.

Definition 4.62 (Extended substitution)**.** The substitution operation $a[b/x]$ to replace free occurrences of x in a by b can be extended as follows: Given sequences of pairwise disjoint variables $X = (x_1, \ldots, x_n)$ and (arbitrary) expressions $B = (b_1, \ldots, b_n)$ where $n \geq 0$, a *substitution function* $\sigma_{X,B}$ is defined on expressions and contexts as follows:

$$\sigma_{X,B}(a) = a[b_1/x_1]\ldots[b_n/x_n]$$

$$\sigma_{X,B}(()) = ()$$

$$\sigma_{X,B}(x : a, \Gamma) = \begin{cases} (x : \sigma_{X,B}(a), \sigma_{X,B}(\Gamma)) & \text{if } x \neq x_i \\ \sigma_{X,B}(\Gamma) & \text{otherwise} \end{cases}$$

If $x \neq x_i$ we write $\sigma_{X,B}[b/x]$ for $\sigma_{(x_1,\ldots,x_n,x),(b_1,\ldots,b_n,b)}$.

Definition 4.63 (Norm-matching substitution)**.** A substitution $\sigma_{X,B}$ where $X = (x_1, \ldots, x_n)$ and $B = (b_1, \ldots, b_n)$ is called *norm matching w.r.t.* Γ iff $\Gamma = (\Gamma_0, x_1 : a_1, \Gamma_1 \ldots x_n : a_n, \Gamma_n)$, for some Γ_0 and a_i, Γ_i where $1 \leq i \leq n$ and furthermore for all these i we have, with σ abbreviating $\sigma_{X,B}$:

$$\sigma(\Gamma) \models \sigma(a_i), \quad \sigma(\Gamma) \models \sigma(b_i) \quad \text{and} \quad \|\sigma(a_i)\|_{\sigma(\Gamma)} = \|\sigma(b_i)\|_{\sigma(\Gamma)}$$

Norm-matching substitutions indeed preserve norms:

Lemma 4.64 (Norm preservation of norm-matching substitutions)**.** *Let $\sigma_{X,B}$ be norm-matching w.r.t. Γ, then for all Γ and a we have $\|\sigma_{X,B}(a)\|_{\sigma_{X,B}(\Gamma)} = \|a\|_\Gamma$.*

Proof. Proof is by induction on the number k of variables in X and follows directly from the definition of norm-matching substitution. □

As a consequence of Lemma 4.64, for any norm-matching substitution $\sigma_{X,B}$ we have $\Gamma \models a$ if and only if $\sigma_{X,B}(\Gamma) \models \sigma_{X,B}$. We now come to the last piece missing to show strong normalisation.

Lemma 4.65 (Normability implies computability)**.** *For all Γ and a: If $\Gamma \models a$ then for any norm-matching substitution $\sigma_{X,B}$ w.r.t. Γ with $\sigma_{X,B}(x_i) \in C_{\sigma_{X,B}(\Gamma)}(\|b_i\|_\Gamma)$, for $1 \leq i \leq n$, we have $\sigma_{X,B}(a) \in C_{\sigma_{X,B}(\Gamma)}(\|a\|_\Gamma)$. As a consequence normability implies computability.*

Proof. Let $\bar{a} = \| a \|_\Gamma$ and let $\sigma = \sigma_{X,B}$ be a norm matching substitution w.r.t. Γ with $\sigma(x_i) \in C_{\sigma(\Gamma)}(\bar{b}_i)$ where $\bar{b}_i = \|b_i\|_\Gamma$, for $1 \leq i \leq n$. The proof is by induction on a:

- $a = \tau$: Obviously $\sigma(\tau) = \tau$ and $\tau \in C_{\sigma_{X,B}(\Gamma)}(\tau)$.
- $a = x$: We have $\Gamma \models x$ and $\bar{a} = \|x\|_\Gamma$. There are two cases:
  - $x = x_i$, for some i. Obviously $\sigma(x) = \sigma(b_i)$. We know that $\sigma(b_i) \in C_{\sigma(\Gamma)}(\bar{b}_i)$.

$$\begin{array}{rcl} \bar{a} & = & \|x\|_\Gamma \\ & = & \quad \text{(Lemma 4.64)} \\ & & \|\sigma(x)\|_{\sigma(\Gamma)} \\ & = & \quad \text{(property of substitution)} \\ & & \|\sigma(b_i)\|_{\sigma(\Gamma)} \\ & = & \quad \text{(Lemma 4.64)} \\ & & \|b_i\|_\Gamma \\ & = & \quad \text{(definition of } \bar{b}_i) \\ & & \bar{b}_i \end{array}$$

    Hence $\sigma(x) = \sigma(b_i) \in C_{\sigma(\Gamma)}(\bar{a})$.
  - If $x \neq x_i$ then $\sigma(x) = x$. Obviously $x \in \mathcal{S}$. From $\Gamma \vdash x$, by Lemma 4.52 we obtain $\Gamma \models x$. The computability conditions are trivially satisfied therefore we have $\sigma(x) \in C_{\sigma(\Gamma)}(\bar{a})$.
- $a = \oplus_x(b, c)$: From $\Gamma \models \oplus_x(b, c)$ by definition of norming we obtain $\Gamma \models b$ and $(\Gamma, x : b) \models c$. Let $\bar{b} = \| b \|_\Gamma$ and $\bar{c} = \| c \|_{\Gamma, x:b}$. Applying the inductive hypothesis with the empty substitution $\sigma_{(),()}$, which is trivially norm-matching w.r.t. any context, we obtain $b \in C_\Gamma(\bar{b})$ and $c \in C_{\Gamma,x:b}(\bar{c})$. By Lemma 4.64 we know that $\sigma(\|b\|_{\sigma(\Gamma)}) = \|b\|_\Gamma = \bar{b}$. By inductive hypothesis $\sigma(b) \in C_{\sigma(\Gamma)}(\bar{b})$.

  Consider an expression d where $d \in C_\Gamma(\bar{b})$. In order to apply Lemma 4.61 we have to show that $\sigma(c)[d/x] \in C_{\sigma(\Gamma)}(\bar{c})$. If we define $X' = (x_1, \ldots, x_n, x)$, $B' = (b_1, \ldots, b_n, d)$, and $\sigma' = \sigma[d/x]$ then obviously σ' is norm-matching w.r.t $(\Gamma, x : a)$ and substitutes to computable expressions. Therefore by inductive hypothesis for c we know that $\sigma'(c) = \sigma(c)[d/x] \in$

$C_{\sigma(\Gamma)}(\bar{c})$ (obviously $\sigma'(\Gamma) = \sigma(\Gamma)$). Hence by Lemma 4.61 it follows that $\oplus_x(\sigma(b), \sigma(c)) \in C_{\sigma(\Gamma)}([\bar{b}, \bar{c}])$ which by definition of substitution is equivalent to $\sigma(\oplus_x(b, c)) \in C_{\sigma(\Gamma)}([\bar{b}, \bar{c}])$.

- $a = [x \dot{=} b, c : d]$: From $\Gamma \models [x \dot{=} a_1, a_2 : a_3]$ by definition of norming we obtain $\Gamma \models a_1, a_2$, $(\Gamma, x : a_1) \models a_3$, and $\|a_2\|_\Gamma = \|a_3\|_{\Gamma, x:a_1}$. Let $\bar{b} = \|b\|_\Gamma$, $\bar{c} = \|c\|_\Gamma$. By inductive hypothesis with σ we know that $\sigma_{X,B}(b) \in C_{\sigma(\Gamma)}(\bar{b})$, $\sigma(c) \in C_{\sigma(\Gamma)}(\bar{c})$, and $\sigma_{X,B}(d) \in C_{\sigma(\Gamma, x:b)}(\bar{c}) = C_{(\sigma(\Gamma), x:\sigma(b))}(\bar{c})$. By Lemma 4.60(1) we then obtain $[x \dot{=} \sigma(b), \sigma(c) : \sigma(d)] \in C_{\sigma(\Gamma)}([\bar{b}, \bar{c}])$. By definition of substitution this is equivalent to $\sigma([x \dot{=} b, c : d]) \in C_{\sigma(\Gamma)}([\bar{b}, \bar{c}])$.
- $a = \oplus(a_1, \ldots, a_n)$: We have $\Gamma \models a$ and $\Gamma \models a_i$. Let $\bar{a} = \|a\|_\Gamma$, $\bar{a}_i = \|a_i\|_\Gamma$. By inductive hypothesis $\sigma(a_i) \in C_{\sigma(\Gamma)}(\bar{a}_i)$. By Lemmas 4.58, 4.59, and the various cases of Lemma 4.60 we obtain $\oplus(\sigma(a_1), \ldots, \sigma(a_n)) \in C_{\sigma(\Gamma)}(\bar{a})$. By definition of substitution this is equivalent to $\sigma(\oplus(a_1, \ldots, a_n)) \in C_{\sigma(\Gamma)}(\bar{a})$.

For the consequence, we just take the empty substitution $\sigma_{(),()}$ which is trivially norm-matching. □

Hence we know that normability implies computability. Lemmas 4.52 and 4.65 together yield strong normalisation.

Theorem 4.66 (Strong normalization of valid expressions)**.** *For all* Γ *and expressions* a*:* $\Gamma \vdash a$ *implies* $a \in \mathcal{S}$*.*

Proof. Assume $\Gamma \vdash a$. By Lemma 4.52 this implies $\Gamma \models a$. By Lemma 4.65 this implies $a \in C_\Gamma(\|a\|_\Gamma)$. Obviously this implies $a \in \mathcal{S}$. □

Definition 4.67 (Normal form)**.** If $a \in \mathcal{S}$ then $\mathcal{N}(a)$ denotes the unique expression to which a is maximally reducible. Note that this definition is well-founded due to confluence of $\rightarrow$ (Theorem 4.26). Furthermore, due to strong normalisation (Theorem 4.66), $\Gamma \vdash a$ implies that $\mathcal{N}(a)$ exists.

We note an easy consequence of confluence and strong normalization.

Corollary 4.68 (Decidability of the type relation)**.** *For any expression* a *and context* Γ *there is a terminating algorithm such that* $\Gamma \vdash a$ *iff the algorithm is not failing but computing an expression* b *with* $\Gamma \vdash a : b$*.*

Proof. The algorithm to attempt to compute a type b of a is recursive on a under the context Γ. It basically works by checking the type conditions on unique normal forms. □

4.4. Consistency.

Due to confluence and the strong normalization result for valid expressions it is often sufficient to consider the normal form $\mathcal{N}(a)$ instead of the expression a itself when proving properties about expressions of `d`. In this section, we study this more rigorously and use a characterization of valid normal forms to show consistency of `d`.

Definition 4.69 (Valid normal forms)**.** The set of *valid normal forms* is a subset of $\mathcal{E}$ and denoted by $\mathcal{N}$. The recursive characterization of valid normal forms in Table 9 also uses the auxiliary set of *dead ends* denoted by $\mathcal{D}$.

$\mathcal{N}$ indeed characterizes the normal forms of valid expressions.

Lemma 4.70 (Normal forms of valid expressions)**.** *For all* a *where* $\vdash a$ *we have that* $a \in \mathcal{N}$ *iff* $\mathcal{N}(a) = a$

$$
\begin{array}{rcl}
\mathcal{N} & = & \{\tau\} \,\cup\, \{[x:a]b, [x!a]b, [x \doteq a, b : c] \mid a,b,c \in \mathcal{N}\} \\
 & & \cup\, \{[a,b], [a+b], [a,:b], [:a,b], [a\,?\,b] \mid a,b \in \mathcal{N}\} \,\cup\, \mathcal{D} \\
\mathcal{D} & = & \{x \mid x \in \mathcal{V}\} \,\cup\, \{(a\,b), a.1, a.2, ([b\,?\,c]\,a) \mid a \in \mathcal{D}, b,c \in \mathcal{N}\} \\
 & & \cup\, \{\neg a \mid a \in \mathcal{D}, a \text{ is not a negation}\}
\end{array}
$$

Table 9: Valid normal forms

Proof. Obviously, by construction, all elements of $\mathcal{N}$ are irreducible. For the reverse, we prove the more general property that for all a and Γ with $\Gamma \vdash a$ we have that $a \in \mathcal{N}$ implies $\mathcal{N}(a) = a$. The proof is by induction on a. ☐

We need a couple of easy lemmas for the consistency proof.

Lemma 4.71 (Valid normal forms of universal abstraction type)**.** *For all x, a, b, and c: If $a \in \mathcal{N}$ and $\vdash a : [x:b]c$ then there is some $d \in \mathcal{N}$ such that $a = \oplus_x(b,d)$ and $x : b \vdash d : c$.*

Proof. Since $FV(a) = \emptyset$, by definition of $\mathcal{D}$ we know that $a \notin \mathcal{D}$. We need to check the following remaining cases

(1) a is τ, a protected definition, an injection, a sum, or a product, and $\vdash a : [x:b]c$: By definition of typing and properties of reduction this cannot be the case.
(2) $a = \oplus_x(a_1,a_2)$. From $\vdash \oplus_x(a_1,a_2) : [x:b]c$, by Lemma 4.32(1) we have $x : a_1 \vdash a_2 : d'$ for some d' where $[x:a_1]d' =_\lambda [x:b]c$. Hence, by Lemma 4.27(2) we know that $a_1 =_\lambda b$ and $d' =_\lambda c$. From $\vdash a : [x:b]c$, by Lemmas 4.36 and 4.33(2) we know that $\vdash b$ and $x : b \vdash c$. Therefore, since $x : a_1 \vdash a_2 : d'$ and $a_1 =_\lambda b$, by Lemma 4.31 we know that $x : b \vdash a_2 : d'$. Similarly since $d' =_\lambda c$, by type rule *conv* we know that $x : b \vdash a_2 : c$. Hence we have shown the property with $d = a_2$. ☐

We also need the following obvious variation of Lemma 4.32(1).

Lemma 4.72 (Abstraction property)**.** *For all Γ, x, a, b, and c: If $\Gamma \vdash \oplus_x(a,b) : [x:a]c$ then $\Gamma, x : a \vdash b : c$.*

Proof. By Lemma 4.32(1) we know that $\Gamma, x : a \vdash b : d$ for some d with $[x:a]d =_\lambda [x:a]c$. By Lemma 4.27(2) we know that $d =_\lambda c$. By Lemma 4.36 we know that $\vdash [x:a]c$. By Lemma 4.33(2) we know that $x : a \vdash c$. Hence by rule *conv* we can infer that $\Gamma, x : a \vdash b : c$. ☐

Finally, we need a lemma for the case of a declaration $x : \tau$.

Lemma 4.73 (τ-Declaration property)**.** *For all a and b: $x : \tau \vdash a : b$ implies that $b \neq_\lambda x$.*

Proof. Due to Lemma 4.38 (subject reduction) we may assume that $a \in \mathcal{N}$. The various cases of Lemma 4.32 imply that $b \neq_\lambda x$ in case a is an abstraction, a sum, a product, a protected definition, a case distinction, or an injection. Hence by definition of $\mathcal{N}$, it remains to look at the case $a \in \mathcal{D}$. By definition of $\mathcal{D}$, since $x : \tau \vdash a$, a can only be a variable x or a negated variable $\neg x$ and therefore obviously $b =_\lambda \tau$. The property follows since $\tau \neq_\lambda x$. ☐

Theorem 4.74 (Consistency of d)**.** *There is no expression a such that $\vdash a : [x:\tau]x$.*

Proof. Assume that there is an expression a with $\vdash a : [x:\tau]x$. By Theorem 4.66 (strong normalization) we know that there is a normal form $a' := \mathcal{N}(a)$ with $a \rightarrow^* a'$. By Lemma 4.39 we know that $\vdash a'$. By Theorem 4.38 (subject reduction) we know that $\vdash a' : [x:\tau]x$.

By Lemma 4.70 we know that $a' \in \mathcal{N}$, hence, by Lemma 4.71 there is an expression c where $\vdash \oplus_x(\tau, c) : [x : \tau]x$. By Lemma 4.72 this implies $x : \tau \vdash c : x$. By Lemma 4.73 this implies that $x \neq_\lambda x$. Thus we have inferred a contradiction and therefore the proposition is true. □

Remark 4.75 (Limitations of the consistency result)**.** Lemma 4.74 shows that there is no inherent flaw in the type mechanism of `d` by which one could prove anything from nothing. Note that the consistency result is limited to empty contexts, hence it does not cover the use of negation or casting axioms (see Section 2.3). Furthermore, it remains an open issue if we can generalize the empty type $[x : \tau]x$ to $[x : a]x$ with $\vdash a$.

5. Comparison to other systems and possible variations and extensions

Due to the use of λ-structured types, `d` falls outside the scope of PTS (see *e.g.* [4]). In Section 1 we have indicated the differences between the core of `d` and PTS. Due to its origins from λ^λ and its use of a reflexive type axiom `d` does not use the concept of dependent product and it does not use a type relation that can intuitively be interpreted as set membership. Instead `d` introduces a number of operators which can be functionally interpreted by untyped λ-expressions, *e.g.* by stripping of the type tags and negations, by interpreting sums, products, and protected definitions as binary pairs and both universal and existential abstraction as λ-abstraction. Of course other interpretations are possible which preserve more semantic detail. In any case, the type rules of `d` would then induce a relation between untyped λ-expressions. Furthermore, `d` has computationally-irrelevant proofs, i.e. it is not possible to extract for example primitive recursive functions from valid expressions. In this section we discuss the use of `d` as a logic and then discuss several extensions of `d`.

5.1. Encoding of logical operators.

In λ^λ, common encodings of logical operators can be used (see Table 10, where $[a \Rightarrow b]$ abbreviates $[x : a]b$ if x does not occur free in b). Hence

false	:=	$[x : \tau]x$
true	:=	$[x : \tau][y : x]y$
implies	:=	$[x : \tau][y : \tau][x \Rightarrow y]$
not	:=	$[x : \tau][x \Rightarrow \text{false}]$
and	:=	$[x : \tau][y : \tau][z : \tau][[x \Rightarrow [y \Rightarrow z]] \Rightarrow z]$
or	:=	$[x : \tau][y : \tau][z : \tau][[x \Rightarrow z] \Rightarrow [[y \Rightarrow z] \Rightarrow z]]$
forall	:=	$[x : \tau][y : [x \Rightarrow \tau]][z : x](y\, z)$
exists	:=	$[x : \tau][y : [x \Rightarrow \tau]][[z : x][(y\, z) \Rightarrow x] \Rightarrow x]$

Table 10: Encoding of logical operators

one could argue that no further logical operators (apart from the law of the excluded middle) seem necessary. We do not follow this argument in `d` because of properties such as

$$\frac{c : a \quad d : b}{[z : \tau][x : [a \Rightarrow [b \Rightarrow z]]]((x\, c)\, d) : \text{and}(a, b)}$$

which we consider less intuitive for deductions involving conjunction as compared to the approach in `d`:

$$\frac{c : a \quad d : b}{[c, d] : [a, b]}$$

5.2. **Logical interpretation.** `d` is treating proofs and formulas uniformly as typed λ-expressions, and allows each of its operators to be used on both sides of the type relation. A subset of the operators of `d`, if used as types, can be associated with common logical predicates and connectors:

$$\begin{array}{rcl} \tau,\ (x\, a_1 \dots a_n) & \simeq & \text{atomic formulas} \\ [x : a]b & \simeq & \text{universal quantification} \\ [x!a]b & \simeq & \text{existential quantification} \\ [a, b] & \simeq & \text{conjunction} \\ [a + b] & \simeq & \text{disjunction} \\ \neg a & \simeq & \text{negation} \end{array}$$

In Sections 1 and 3.1 we have shown that on the basis of the type system of `d` many logical properties of these connectors can be derived without further assumptions. Furthermore, on the basis of a strong normalization result (4.74) we have shown that `d` is consistent in the sense that the type $[x : \tau]x$ is empty in `d` under the empty context. In this sense, `d` can be seen as a logic where typing can be interpreted as the relation between a deduction and the proposition it has shown [16].

However, in order to have the complete properties of negation, additional axioms have to be assumed and we did not show consistency of the type system under these axioms by means of strong normalization. Similarly, formalization of mathematical structures in `d` must be done axiomatically. In this sense the expressive power of `d` is limited and each axiomatization has to be checked for consistency. Furthermore, there are two important pragmatic issues which differ from common approaches:

- First, inference systems for higher-order logic on the basis of typed-λ-calculus such as [5, 28] typically make a distinction between the type of propositions and one or more types of *individuals*. In `d`, one the one hand there is no such distinction, all such types must either be τ itself or declared using τ. On the other hand, due to the restricted formation rules which serve to ensure consistency as well as uniqueness of types, in `d`, τ does not allow to quantify over all propositions of `d` and additional axioms schemes must be used when reasoning with formulas of complex structure.
- Second, `d` has several operators which are not common logical connectors: protected definitions $[x \doteq a, b : c]$, projections $a.1$, $a.2$, case distinction $[a \,?\, b]$, as well as left and right injection $[a, :b]$, and $[:a, b]$. However, these operators have meaningful type-roles for defining functions over propositions.

Note also that there is a strong relation between left projection $a.1$ on a deduction a and Hilbert's ϵ-operator $\epsilon x.P$ on a formula P as sometimes used in higher-order logic [5, 28] with a law like:

$$\forall x.(P \Rightarrow P[\epsilon x.P/x])$$

In a classical logical setting this is obviously implied by

$$(\exists x.P) \Rightarrow P[\epsilon x.P/x]$$

The latter property can be approximated in `d` by

$$[y : [x!\tau](P\,x)](P\,y.1)$$

and actually is a law since

$$[y : [x!\tau](P\,x)]y.2 \;:\; [y : [x!\tau](P\,x)](P\,y.1)$$

This illustrates again how existential abstraction and the projection operators together embody a strong axiom of choice.

Finally there is no equality operator in `d`, a notion of equality is defined indirectly only through congruence of expressions.

5.3. **Negation.** In `d` rather than defining negation by implication to falsehood, negation is incorporated by defining a subset of the equivalence laws of negation as equalities ($=_\lambda$). The purpose is to have unique formal forms with respect to negation in order to simplify deductions. A direct isomorphism between $\neg\neg a$ and a has been advocated in [24]. De Morgan laws for propositional operators have been used to define an involutive negation in a type language [3].

As shown in Section 3.1, additional axioms schemes must be assumed to have the full set of properties of logical negation. While this is adequate for deductive reasoning where proofs are not computationally relevant, it leaves open the issue of consistency of the axiomatic extensions, i.e. the question if typing with additional negation axioms is consistent.

Several approaches have been proposed to internalize classical reasoning into λ-calculus. $\lambda\mu$-calculus [27, 7] is an extension of λ-calculus formalizing inference in classical natural deduction by additional operators that give explicit control over the context in which specified subexpressions are evaluated. Classical natural deduction has apparently not yet been studied in the context of lambda-typed systems. In order to prove consistency for `d` with negation axioms it would seem an interesting step to try to extend its reduction through appropriate control operators.

5.4. **Uniqueness of types.** Note that uniqueness of types (4.42) was not needed in the strong normalisation proof but only the weaker property 4.52.

Protected definitions $[x \doteq a, c : d]$ carry a type tag d allowed to use x in order to ensure uniqueness of types. Law 4.52 would be retained if we remove the type tag and the binding of x from protected definitions:

$$\frac{\Gamma \vdash a : b \quad \Gamma \vdash c : d[a/x] \quad \Gamma, x : b \vdash d : e}{\Gamma \vdash [\_ \doteq a, c] : [x!b]d}$$

However, this may lead to a significant number of type variants of a protected definition as in the following example where the four possible types are separated by comma:

$$\frac{\Gamma \vdash a : b \quad \Gamma \vdash c : [d \Rightarrow d] \quad \Gamma, x : b \vdash d : e}{\Gamma \vdash [\_ \doteq a, c] : [x!b][d \Rightarrow d],\ [x!b][x \Rightarrow d],\ [x!b][d \Rightarrow x],\ [x!b][x \Rightarrow x]}$$

In case of universal abstractions the situation is fundamentally different: Adding the following type rule for universal abstractions

$$\frac{\Gamma, x : a \vdash b : \tau}{\Gamma \vdash [x : a]b : \tau}$$

would violate both 4.42 and 4.52 and together with $\vdash \tau : \tau$ result in a paradoxical system [32].

5.5. **Abbreviation systems.** Complex systems of abbreviations spanning over several abstraction levels play a major conceptual role in mathematical work. A multitude of proposals for incorporating definitions into typed λ-calculi have been made, *e.g.* [10] [15] [19]. Support for definitional extensions for systems closely related to Automath's Λ have been investigated in [13]. Note that definitional extensions would allow for a relaxation of a premise of the type rule for existential abstractions. Recall the introduction rule for existential abstractions (*def*) which does not allow free occurrences of x in c.

$$(def) \qquad \frac{\Gamma \vdash a : b \quad \Gamma \vdash c : d[a/x] \quad \Gamma, x : b \vdash d : e}{\Gamma \vdash [x \doteq a, c : d] : [x!b]d}$$

Since c may use a in its type, it could also use an abbreviation x of a. Hence it seems plausible that also c itself may use an abbreviation x of a. In a calculus that includes definitional extensions this dependency could be modelled. In `d` this is not possible and maximally unfolded expressions must be used.

While support for definitional extensions is undoubtedly important, in our setting, they have not been necessary to formulate `d`. Note that in other settings this might be different and abbreviation systems become indispensable, *e.g.* [22].

6. Acknowledgements

My thanks go to the anonymous reviewers of this paper for their detailed and constructive comments.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1:375–416, 10 1991. URL: `http://journals.cambridge.org/article_S0956796800000186`, `doi:10.1017/S0956796800000186`.

[2] Beniamino Accattoli and Delia Kesner. The Structural λ-calculus. *Proceedings of the 24th International Conference/19th Annual Conference on Computer Science Logic*, LNCS 6247, pages 381-395, Brno, Czech Republic, August 2010.

[3] F. Barbanera and S. Berardi. A symmetric lambda calculus for classical program extraction. *Information and Computation*, 125(2):103–117, 03 1996.

[4] H. P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*, volume 2. Oxford University Press, 1991.

[5] Alonzo Church. A formulation of the simple theory of types. *Jurnal of Symbolic Logic*, 5(2):56–68, June 1940.

[6] Thierry Coquand. An analysis of girard's paradox. In *LICS*, pages 227–236. IEEE Computer Society, 1986. URL: `http://dblp.uni-trier.de/db/conf/lics/lics86.html#Coquand86`.

[7] Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In Martin Odersky and Philip Wadler, editors, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000.*, pages 233–243. ACM, 2000. URL: http://doi.acm.org/10.1145/351240.351262, doi:10.1145/351240.351262.

[8] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *INDAG. MATH*, 34:381–392, 1972.

[9] N.G. de Bruijn. A survey of the project automath. In Jonathan P. Seldin and J. Roger Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press, 1980.

[10] N.G. de Bruijn. Generalizing automath by means of a lambda-typed lambda calculus. In J.H. Geuvers R.P. Nederpelt and R.C. de Vrijer, editors, *Selected Papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics*, pages 313 – 337. Elsevier, 1994. URL: http://www.sciencedirect.com/science/article/pii/S0049237X08702131.

[11] J.Y. Girard. *Interprétation fonctionnelle et élimination des coupures dans l`arithmetique d`ordre supérieur*. Université Paris VII, 1972. Ph.D. Thesis.

[12] J.Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989. URL: https://books.google.de/books?id=6JOEQgAACAAJ.

[13] Philippe de Groote. Nederpelt's calculus extended with a notion of context as a logical framework. In Gerard Huet and G. Plotkin, editors, *Logical Frameworks*, pages 69–86. Cambridge University Press, 1991. Cambridge Books Online. URL: http://dx.doi.org/10.1017/CBO9780511569807.005.

[14] Philippe de Groote. Defining lambda-typed lambda-calculi by axiomatizing the typing relation. In *Proceedings of the 10th Annual Symposium on Theoretical Aspects of Computer Science*, STACS '93, pages 712–723, London, UK, UK, 1993. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=646509.694657.

[15] Ferruccio Guidi. The formal system lambda-delta. *ACM Trans. Comput. Logic*, 11(1):5:1–5:37, November 2009. URL: http://doi.acm.org/10.1145/1614431.1614436, doi:10.1145/1614431.1614436.

[16] William A. Howard. The formulae-as-types notion of construction. In Jonathan P. Seldin and J. Roger Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980. Original paper manuscript from 1969.

[17] Antonius J. C. Hurkens. *A simplification of Girard's paradox*, pages 266–278. Springer Berlin Heidelberg, Berlin, Heidelberg, 1995. URL: http://dx.doi.org/10.1007/BFb0014058, doi:10.1007/BFb0014058.

[18] Fairouz Kamareddine. Typed λ-calculi with one binder. *Journal of Functional Programming*, 15:771–796, 9 2005. URL: http://journals.cambridge.org/article_S095679680500554X, doi:10.1017/S095679680500554X.

[19] Fairouz Kamareddine, Roel Bloo, and Rob Nederpelt. On Π-conversion in the λ-cube and the combination with abbreviations. *Annals of Pure and Applied Logic*, 97(1):27 – 45, 1999. URL: http://www.sciencedirect.com/science/article/pii/S0168007298000190.

[20] D. Kesner and S. Ó Conchúir. *Milner`s Lambda Calculus with Partial Substitutions*. Technical Report, Université Paris, URL: http://www.pps.univ-paris-diderot.fr/~kesner/papers/shortpartial.pdf, 2008.

[21] Zhaohui Luo. Ecc, an extended calculus of constructions. In *Logic in Computer Science, 1989. LICS '89, Proceedings., Fourth Annual Symposium on*, pages 386–395, June 1989. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=39193&tag=1, doi:10.1109/LICS.1989.39193.

[22] Zhaohui Luo. Pal+: a lambda-free logical framework. *J. Funct. Program.*, 13:317–338, 2003.

[23] Robin Milner. Local Bigraphs and Confluence: Two Conjectures In *EXPRESS 2006*, volume 175 of *Electronic Notes in Theoretical Computer Science*, Elsevier, 2007.

[24] Guillaume Munch-Maccagnoni. Formulae-as-types for an involutive negation. In Thomas A. Henzinger and Dale Miller, editors, *Joint Meeting of the 23rd EACSL Annual Conference on Computer Science Logic(CSL) and the 29th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS'14*, pages 70:1–70:10, Vienna, Austria, July 2014. ACM Press.

[25] R.P. Nederpelt. *Strong normalisation in a typed lambda calculus with lambda structured types*. Eindhoven University of Technology, 1973. Ph.D. Thesis.

[26] M. H. A. Newman. On theories with a combinatorial definition of equivalence. *Annals of Math.*, 43(2):223 –243, 1994.

[27] Michel Parigot. *On the Computational Interpretation of Negation*. In P.G.Clote and H. Schwichtenberg, editors, *Computer Science Logic*, volume 1862 of *Lecture Notes in Computer Science*, Springer Verlag.

[28] L. C. Paulson. The foundation of a generic theorem prover. *J. Autom. Reason.*, 5(3):363–397, September 1989. URL: `http://dx.doi.org/10.1007/BF00248324`, `doi:10.1007/BF00248324`.

[29] William W. Tait. Intensional interpretations of functionals of finite type i. *The journal of symbolic logic*, 32(2):198–212, 1967.

[30] L.S. van Benthem Jutting. The language theory of lambda infinity, a typed lambda-calculus where terms are types. *Studies in Logic and the Found. of Math.*, 133:655 – 683, 1994.

[31] D.T. van Daalen. *The language theory of automath.* Technische Hogeschool Eindhoven, 1977.

[32] Matthias Weber. An extended type system with lambda-typed lambda-expressions (extended version). Technical report, arXiv identifier 1803.06488, 2020.

[33] Masako Takahashi Parallel Reductions in λ-Calculus *Journal of Symbolic Computation*, 118(4), pp.113-123, 1995.