Subset Types and Partial Functions

Aaron Stump

Dept. of Computer Science and Engineering
Washington University in St. Louis
Web: http://www.cs.wustl.edu/~stump/

Abstract. A classical higher-order logic PF_{sub} of partial functions is defined. The logic extends a version of Farmer's logic PF by enriching the type system of the logic with subset types and dependent types. Validity in PF_{sub} is then reduced to validity in PF by a translation.

1 Introduction

Logics of partial functions are of practical interest for formal modelling and verification of hardware, software, and protocols. Such systems often use operations like division or selectors of inductive data types which are most naturally viewed as undefined on some inputs. Previous works studying such logics and their implementations include [16, 17, 10, 11, 4].

Subset types have been proposed for similar purposes. Intuitively, a subset type $A|P$ is formed from a type A and a predicate P on A; and something is in this subset type if it is in type A and satisfies predicate P. The widely used proof assistant PVS relies heavily on subset types for modelling and verification of systems [19, 23]. In PVS, higher-order functional terms are interpreted as total functions on their domains. An application of a function with domain $A|P$ to an argument a of type $A|Q$ where $Q(a)$ does not imply $P(a)$ is considered ill-typed. This quickly leads to undecidability of type-checking in PVS. Attempting to type check a goal formula leads to type-correctness conditions (TCCs) which if proved, establish that the formula is well-typed and hence possibly provable. Unfortunately, it can happen that the TCCs generated by PVS for a formula are unprovable, but the TCCs for what should intuitively be an equivalent formula are provable [23, 20]. The example given in Chapter 6 of [20] is

$$\textit{if } 1/i > 0 \textit{ then } i \neq 0 \textit{ else } \mathbf{F}.$$

The TCC generated for this formula by PVS is $i \neq 0$, which is not valid. But PVS generates the valid TCC $i \neq 0 \ \supset i \neq 0$ for the following formula, which we expect to be logically equivalent:

$$\textit{if } i \neq 0 \textit{ then } 1/i > 0 \textit{ else } \mathbf{F}.$$

Due to this difference, these two formulas are not provably equivalent in PVS.

This paper develops a unified approach to partial functions and subset types, which does not suffer from this anomalous behavior. We begin with a higher-order logic that allows functions to be undefined on some arguments. We extend this logic's type system to include subset types, but we retain decidability of type checking essentially by having the type system ignore subset types. So an application of a function expecting a non-zero number to zero will be well-typed. But the proof system for the logic will then state that that application is undefined. So the constraints determined by the subset types are enforced by the proof system, not the type system.

Section 2 formulates this system of partial functions and subset types, called $\mathrm{PF}_{\mathrm{sub}}$. Section 3 develops some basic proof theory of $\mathrm{PF}_{\mathrm{sub}}$. Section 4 defines a subtyping relation between types in terms of more primitive notions, and shows how standard subtyping rules, including contravariant subtyping of functional types, are derivable. Section 5 reduces validity in $\mathrm{PF}_{\mathrm{sub}}$ to validity in $\mathrm{PF}_{\mathrm{sub}}$ without subset types. This latter system we call PF, since it is essentially a version of Farmer's logic of partial functions PF [8, 9]. Section 6 summarizes the basic meta-theory of PF.

2 Definition of $\mathrm{PF}_{\mathrm{sub}}$

This section defines the logic $\mathrm{PF}_{\mathrm{sub}}$. The basic idea is to add subset types to Farmer's PF [9, 8]. PF is based on Andrews's system Q_0 [1], which is based in turn on Church's original higher-order logic [5]. While conceptually we begin with PF and add subset types, for clarity of presentation we develop $\mathrm{PF}_{\mathrm{sub}}$ first and then define PF as $\mathrm{PF}_{\mathrm{sub}}$ without subset types.

2.1 Type system

The type system of $\mathrm{PF}_{\mathrm{sub}}$ has basic types ι, for individuals, and o, for formulas. Subset types complicate matters, since type expressions can contain formulas as subexpressions. A further complication arises with typing lambda expressions like $\lambda\, x : \iota.\, \lambda\, y : (\iota|\lambda\, y : \iota.\, y \geq x).\, y - x$ The most natural type for this is $\Pi\, x : \iota.\, \Pi\, y : (\iota|\lambda\, y : \iota.\, y \geq x).\, \iota$ which uses dependent function types $\Pi\, x : A.\, B$ instead of simple function types $A \to B$. Hence, the type system of $\mathrm{PF}_{\mathrm{sub}}$ involves dependent types (see, e.g., [21]). We will rely on standard notations and definitions from typed lambda calculus.

Figure 1 defines the typing relation $\Vdash$ of $\mathrm{PF}_{\mathrm{sub}}$ inductively, using a standard style. For uniformity, the definition uses a single basic kind $*$ to classify types, which themselves classify terms. There are two different classifications which may be used: $X :: Y$ means that X is exactly described by type Y, while $X : Y$ means that X satisfies type Y, where Y is viewed as a specification. The former is the kind of type declaration we might like to have in typing contexts, while the latter is the kind we need for λ- or Π-bound variables to get contravariant subtyping of functional types. From a type declaration $f :: \iota|P \to \iota$, it will follow that $(f\ x)$ is undefined for all x such that $\neg(P\ x)$. On the other hand, it will

(ι-type)

$$\frac{}{\Vdash \iota : *}$$

(o-type)

$$\frac{}{\Vdash o : *}$$

(sym1)

$$\frac{\Sigma \Vdash A : *}{\Sigma,\ x : A \Vdash x : A}\ e(A) \equiv A$$

(weak1)

$$\frac{\Sigma \Vdash A : * \qquad \Sigma \Vdash M :_2 C}{\Sigma,\ x : A \Vdash M :_2 C}\ e(A) \equiv A$$

(sym2)

$$\frac{\Sigma \Vdash A : *}{\Sigma,\ x :: A \Vdash x :: A}\ e(A) \not\equiv A$$

(weak2)

$$\frac{\Sigma \Vdash A : * \qquad \Sigma \Vdash M :_2 C}{\Sigma,\ x :: A \Vdash M :_2 C}\ e(A) \not\equiv A$$

(Π)

$$\frac{\Sigma,\ y :_1 A \Vdash B : *}{\Sigma \Vdash \Pi\, x :_1 A.\, [x/y]B : *}$$

($|$)

$$\frac{\Sigma \Vdash A : * \qquad \Sigma \Vdash P :_1 A' \to o}{\Sigma \Vdash A|P : *}\ (1),\ (2)$$

($=$)

$$\frac{\Sigma \Vdash A : *}{\Sigma \Vdash =_A :: A \to A \to o}$$

(λ)

$$\frac{\Sigma,\ y :_1 A \Vdash M :_2 B \qquad \Sigma, y :_1 A \Vdash B : *}{\Sigma \Vdash \lambda\, x :_1 A.\, [x/y]M :_2 \Pi\, x :_1 A.\, [x/y]B}$$

(I)

$$\frac{\Sigma \Vdash \alpha : *}{\Sigma \Vdash I_\alpha :: (\alpha \to o) \to \alpha}\ e(\alpha) \equiv \alpha$$

(app)

$$\frac{\Sigma \Vdash M :_2 \Pi\, x :_1 A.\, B \qquad \Sigma \Vdash N :_3 A'}{\Sigma \Vdash M\ N :_2 [N/x]B}\ (2)$$

(strip)

$$\frac{\Sigma \Vdash M :_1 A|P}{\Sigma \Vdash M :_1 A}$$

Fig. 1. Type system of $\mathrm{PF}_{\mathrm{sub}}$

be consistent to have $f : \iota|P \to \iota$ and yet have $(f\ x)$ defined for x such that $\neg(P\ x)$ holds. In Figure 1 and subsequently, the metavariables $:_1$, $:_2$, etc. range over $\{:, ::\}$. The function $e()$ does the following two things: turn every subset type $A|P_1|\ldots|P_n$ into just A, where A is not a subset type; and change every $::$ into a $:$. When classifying an expression by $*$ or by A where $e(A) \equiv A$, we always use $:$.

As usual in type theory, we require that typing contexts Σ contain at most one typing declaration for a given symbol. The symbols x introduced by the rule (sym) are drawn from a countably infinite set *Symbols*. $[N/x]B$ denotes the result of safely substituting N for x in B, where safe substitution respects λ- and Π-bindings of variables in the usual way; bound variables may be renamed to avoid capture. We abbreviate $\Pi\, x : A.\, B$ by $A \to B$ if B does not contain x free, and also $\Pi\, x :: A.\, B$ by $A \Rightarrow B$. We say that x occurs free in typing context Σ iff there is a declaration $y : A$ in Σ such that x occurs free in A.

The rules ($=$) and (I) give types to an infinite family of logical symbols for equality and definite descriptions, respectively. We also have the following side conditions.

1. In the rule (|), we require $A \not\equiv o$. We gain little by allowing subsetting of type o, and it simplifies the presentation somewhat to disallow it.
2. Also, in the rules (|) and (app), we require $e(A) \equiv e(A')$. As remarked previously, type checking will become undecidable if the type system attempts to take all subset constraints into account. We avoid undecidability by having the type system ignore constraints imposed by subset types. The ignored constraints are taken into account in the deductive system for the logic (in Section 2.3).

The rule (strip) enables simpler statements of some axioms below. For simplicity, we consider only definite descriptions of things of type α, where α does not contain subset types. We could define a second family of equality symbols, each of type $A \Rightarrow A \Rightarrow o$, but again for simplicity we will not do so. Note that (weak1) is not a special case of (weak2), due to the different side conditions; and similarly for (sym1) and (sym2). Finally, a typing context Σ is *valid* iff $\Sigma \Vdash o : *$ is derivable.

2.2 Abbreviations

This section presents some abbreviations and syntactic conventions, mostly following [2, 8]. We write equalities $=_A\ M\ N$ as $M =_A N$. Abbreviations for logical connectives are given in Figure 2. Standard precedences and associativities are used. The unary postfix operators $\downarrow$ and $\uparrow$ are for definedness and undefinedness, respectively. Notice that subset types are erased in the definitions. This justifies omitting the subscripts on $\downarrow$ and $\uparrow$. These symbols will bind more tightly than the other logical connectives and the equality symbol. The abbreviation for $\propto$ is introduced to allow more concise formulations of some axioms in Section 2.3 below.

$$
\begin{array}{lllll}
\mathbf{T} & := (=_o\ =_{o \to o \to o}\ =_o) & \quad & X \neq_A Y & := \neg(X =_A Y) \\
\mathbf{F} & := (\lambda\, x : o.\, \mathbf{T}\ =_{o \to o}\ \lambda\, x : o.\, x) & & \phi \supset \psi & := (\phi\ =_o\ (\phi \wedge \psi)) \\
\phi \wedge \psi & := (\lambda\, C : o \to o \to o.\, (C\ \phi\ \psi)\ =_\sigma & & \forall\, x : A.\, \phi & := (\lambda\, x : A.\, \phi \simeq_\tau \lambda\, x : A.\, \mathbf{T}) \\
 & \qquad \lambda\, C : o \to o \to o.\, (C\ \mathbf{T}\ \mathbf{T})) & & \forall\, x :: A.\, \phi & := (\lambda\, x :: A.\, \phi \simeq_\tau \lambda\, x :: A.\, \mathbf{T}) \\
\neg\phi & := (\phi\ =_o\ \mathbf{F}) & & \exists\, x : A.\, \phi & := \neg(\forall\, x : A.\, \neg\phi) \\
\phi \vee \psi & := \neg(\neg\phi \wedge \neg\psi) & & \exists\, x :: A.\, \phi & := \neg(\forall\, x :: A.\, \neg\phi) \\
X \downarrow_A & := (\lambda\, x : e(A).\, \mathbf{T})\ X & & \propto^o M & := M = \mathbf{F} \\
X \uparrow_A & := \neg(X \downarrow_A) & & \propto^\alpha M & := M \uparrow_\alpha \quad \text{if } \alpha \not\equiv o \\
X \simeq_A Y & := (X \downarrow_A \vee\, Y \downarrow_A) \supset (X =_A Y) & & & \\
 & & & & \\
\multicolumn{2}{l}{\sigma \equiv (o \to o \to o) \to o} & & \multicolumn{2}{l}{\tau \equiv A \to A \to o}
\end{array}
$$

Fig. 2. Abbreviations for logical constants

Figure 3 defines two abbreviations $\lhd$ and $\unlhd$ which are crucial in what follows. They correspond to the classifications :: and :, respectively. Roughly, $t \unlhd A$ says that term t, if defined, can be used where an element of type A is required. The formula $t \lhd A$ makes the stronger statement that if t is defined, it is truly an element of A. The difference is the same as the difference between :: and :.

$$
\begin{array}{ll}
t \unlhd o & := \mathbf{T} \\
t \unlhd \iota & := \mathbf{T} \\
t \unlhd A|P & := (P\ t)\ \wedge\ t \unlhd A \\
t \unlhd \Pi\, x : A.\, B & := \\
\quad \forall\, x : e(A). x \unlhd A\ \wedge\ (t\ x)\!\downarrow\ \supset\ (t\ x) \unlhd B & \\
t \unlhd \Pi\, x :: A.\, B & := \\
\quad \forall\, x : e(A). x \lhd A\ \wedge\ (t\ x)\!\downarrow\ \supset\ (t\ x) \unlhd B &
\end{array}
\qquad
\begin{array}{ll}
t \lhd o & := \mathbf{T} \\
t \lhd \iota & := \mathbf{T} \\
t \lhd A|P & := (P\ t)\ \wedge\ t \lhd A \\
t \lhd \Pi\, x : A.\, B & := \\
\quad \forall\, x : e(A).\, (\neg(x \unlhd A)\ \supset\ \propto^{B}(t\ x))\ \wedge & \\
\qquad (x \unlhd A\ \wedge\ (t\ x)\!\downarrow\ \supset\ (t\ x) \lhd B) & \\
t \lhd \Pi\, x :: A.\, B & := \\
\quad \forall\, x : e(A).\, (\neg(x \lhd A)\ \supset\ \propto^{B}(t\ x))\ \wedge & \\
\qquad (x \lhd A\ \wedge\ (t\ x)\!\downarrow\ \supset\ (t\ x) \lhd B) &
\end{array}
$$

Fig. 3. Definition of abbreviations $\lhd$ and $\unlhd$

2.3 Valid formulas

Figure 4 gives an inductive definition of the set of logically valid sequents of $\mathrm{PF}_{\mathrm{sub}}$. These sequents are of the form $\Sigma\,; \Gamma \vdash \phi$. We elide the Σ from logical sequents in all the rules except (generalize), because it is always the same from premises to conclusion. Many of the rules are present or inspired by those in [9, 1] for PF and Q_0, but there are significant differences. For typographical reasons, the name of each rule and any side conditions of the rule are written above the rule.

Notation: The notation $\Sigma \ni x :_1 A$ means that Σ is a valid typing context containing the type declaration $x :_1 A$. Also, $\Sigma \ni x_1, \ldots, x_n$ means that for all $i \in \{1, \ldots, n\}$, $\Sigma \ni x_i :_1 A$ holds for some $:_1$ and A.

The side condition (**) on several of the rules is $x \in \mathit{Sym}$ and x is not declared in Σ. This keeps variables that were free in the premises from becoming inappropriately bound in the conclusions of those rules. The side condition (***) on rules (β-reduction) and (|-outer) is that $\lhd?$ is $\lhd$ if $:_1$ is ::, and $\unlhd$ if $:_1$ is :. These two places are where the connection between : and :: on the one hand and $\lhd$ and $\unlhd$ on the other is made.

The rules (replace), (generalize) and (weaken) are proper inference rules, in the sense that they have logical premises. All the other rules are logical axioms: they have typing sequents as premises, but not logical sequents. The rules (β-reduction-o), (β-reduction-$\wedge$), ($\mathbf{T}$), (weak equality-o) and (Leibniz) are technical, in the sense that they are used to derive more general rules which then entail them. The rule (Leibniz) is used in the proof of the Deduction Theorem in the same way as in [1]. It is easily derivable using the Deduction Theorem. The specialized β-reduction rules and the rule ($\mathbf{T}$) are used to derive modus ponens, which enables the more general (β-reduction) rule to be used. In [8], an axiom like the general (β-reduction) rule is used without the technical rules.

(assume)
$$\frac{\Sigma \Vdash \phi : o}{\phi \vdash \phi}$$

(weaken)
$$\frac{\Gamma \vdash \psi \qquad \Sigma \Vdash \phi : o}{\Gamma, \phi \vdash \psi}$$

(truth values): (**)
$$\frac{\Sigma \ni g : o \to o}{\vdash (g\ \mathbf{T}\ \wedge\ g\ \mathbf{F}) =_o (\forall x : o.\, g\ x)}$$

(sym convergence)
$$\frac{\Sigma \ni x :_1 A}{\vdash x\!\downarrow}$$

(o-convergence)
$$\frac{\Sigma \Vdash M\ N : o}{\vdash M\ N\!\downarrow}$$

(weak equality-o)
$$\frac{\Sigma \Vdash (M \simeq_o N) : o}{\vdash (M \simeq_o N) \simeq_o (M =_o N)}$$

(=-convergence)
$$\frac{\Sigma \Vdash A : *}{\vdash =_A \downarrow}$$

(I-convergence)
$$\frac{\Sigma \Vdash A : *}{\vdash I_A\!\downarrow}$$

(weak equality)
$$\frac{\Sigma \ni x, y \qquad \Sigma \Vdash (x \simeq_A y) : o}{\vdash (x \simeq_A y) \simeq_o (x =_A y)}$$

($\mathbf{T}$)
$$\frac{}{\vdash \mathbf{T}}$$

(λ-convergence)
$$\frac{\Sigma \Vdash \lambda x :_1 A.\, M :_1 \alpha}{\vdash \lambda x :_1 A.\, M\!\downarrow}$$

(generalize): $y \notin \mathrm{FV}(\Gamma)$
$$\frac{\Sigma, y :_1 A\,;\, \Gamma \vdash \phi}{\Sigma\,;\, \Gamma \vdash \forall x :_1 A.\, [x/y]\phi}$$

(extensionality): (**)
$$\frac{\Sigma \ni f, g \qquad \Sigma \Vdash (f =_{\Pi\, x :_1 C.\, D}\ g) : o}{\vdash (f =_{\Pi\, x :_1 C.\, D}\ g)\ =_o\ (\forall x :_1 C.\, (f\ x) \simeq_D (g\ x))}$$

(divergence)
$$\frac{\Sigma \Vdash (M\ N) :_1 \alpha}{\vdash (M\!\uparrow\ \vee\ N\!\uparrow)\ \supset\ \propto^{\alpha}(M\ N)}$$

(β-reduction): (***)
$$\frac{\Sigma \Vdash (\lambda x :_1 A.\, M)\ N :_2 \alpha}{\vdash N\!\downarrow\ \supset\ (N \lhd ?A)\ \supset\ ((\lambda x :_1 A.\, M)\ N\ \simeq_{e(\alpha)}\ [N/x]M)}$$

(|-outer): (***)
$$\frac{\Sigma \Vdash M :_1 A \qquad \Sigma \Vdash A : *}{\vdash M\!\downarrow\ \supset\ M \lhd ?A}$$

(Leibniz)
$$\frac{\Sigma \ni h, x, y \qquad \Sigma \Vdash x =_{A'} y : o \qquad \Sigma \Vdash (h\ x =_o h\ y) : o}{\vdash x =_{A'} y\ \supset\ (h\ x =_o h\ y)}$$

(replace): $A \equiv e(A)$ and (*)
$$\frac{\Gamma \vdash X \simeq_A Y \qquad \Gamma \vdash C}{\Gamma \vdash D}$$

(β-reduction-$\wedge$): $\tau \equiv o \to o \to o$
$$\frac{\Sigma \Vdash Q : o \qquad \Sigma \Vdash ((\lambda C : \tau.\, C\ P\ Q)\ \lambda u : o.\, \lambda v : o.v) : o}{\vdash (\lambda C : \tau.\, C\ P\ Q)\ \lambda u : o.\, \lambda v : o.v\ \simeq_o\ Q}$$

(β-reduction-o)
$$\frac{\Sigma \Vdash (\lambda x : o.\, M)\ N :_1 \alpha \qquad \Sigma \Vdash N : o}{\vdash (\lambda x : A.\, M)\ N\ \simeq_{e(\alpha)}\ [N/x]M}$$

(|-=)
$$\frac{\Sigma \Vdash (x =_{A|P} y) : o}{\vdash (x =_A y\ \wedge\ (P\ x)\ \wedge\ (P\ y))\ =_o\ (x =_{A|P} y)}$$

(dd-1): (**), $e(\alpha) \equiv \alpha$
$$\frac{\Sigma \Vdash (t =_\alpha t) : o}{\vdash t\!\downarrow\ \supset\ (I_\alpha(\lambda x : \alpha.\, x =_\alpha t) =_\alpha t)}$$

(dd-2): (**), $e(\alpha) \equiv \alpha$
$$\frac{\Sigma \Vdash (M =_{\alpha \to o} M) : o}{\vdash (\forall x : \alpha.\, M \neq_{\alpha \to o} (\lambda y : \alpha.\, y =_\alpha x))\ \supset\ \propto^{\alpha}(I_\alpha\ M)}$$

Fig. 4. Logical rules of $\mathrm{PF}_{\mathrm{sub}}$

This turns out to be too restrictive to allow many derivations to go through. [9] seeks to correct this error by using an inference rule of (β-reduction). But this then requires an additional case in the proof of the Deduction Theorem, which is omitted in [9]. The author has not been able to reconstruct this case. The technical axioms chosen here for $\mathrm{PF}_{\mathrm{sub}}$ are just strong enough to allow a derivation of modus ponens, but do not require an additional inference rule. The proof of the Deduction Theorem then proceeds much like in [1].

In the rule (replace), D is the result of replacing one occurrence of X by Y in a valid formula C when $X \simeq_A Y$ is valid. A side condition (*) is needed to deal with the case when $X \simeq_A Y$ contains free variables. Note that variables occurring free in a predicate P in a subset type $A|P$ are considered part of the free variables of that type. The occurrence of X that is replaced cannot be beneath a λ-binding of any symbol x which occurs free in $X \simeq_A Y$ and either Γ or the typing context Σ. Furthermore, suppose x is a symbol which occurs free in $X \simeq_A Y$ but not in Γ or Σ. Suppose further that $\Sigma \Vdash x :_1 A$ is derivable. Then it is required that if there are any λ-bindings of x above the occurrence of X to be replaced, then the nearest enclosing λ-binding of x above that occurrence must be of the form $\lambda\, x :_1 A.\, M'$ for some M'. Finally, if X is itself a symbol, then the occurrence which is replaced by Y is not allowed to be the binding occurrence of X in $\lambda\, X :_1 A.\, M$.

The (generalize) rule allows variables to be moved out of the typing context. Such a rule is not needed in [9, 1], since explicit typing contexts are not used, and a countable set of variables of every type is assumed. This approach cannot soundly be taken here, because if we have $\Sigma \ni x : A$ with $\Sigma \Vdash A : *$, then (sym convergence) and (|-outer) give us $\vdash x \lhd A$. If A is something like $(\iota|\lambda\, x : \iota.\, \mathbf{F})$, then the latter sequent is equivalent to $\vdash \mathbf{F}$. So if we always had $x : A$ available for all types A, our system would be inconsistent. By keeping track of variables in a typing context, we show below that we preserve consistency. For particular choices of Σ, like one containing $x : (\iota|\lambda\, x : \iota.\, \mathbf{F})$, it can still happen that $\Sigma;\ \vdash \mathbf{F}$ is derivable.

2.4 Examples

To create a theory of lists, we can declare function symbols *null*, *cons*, *car*, and *cdr*. It is convenient to declare that the domain type of *car* and *cdr* is

$$\iota|\lambda\, x : \iota.\ \mathit{cons?}\ x$$

where *cons?* abbreviates

$$\lambda\, x : \iota.\, \exists\, y : \iota.\, \exists\, z : \iota.\, x = \mathit{cons}\ y\ z$$

Using the results developed in the sequel, this definition and suitable other axioms about *null*, *cons*, *car*, and *cdr* lead to the validity of formulas like this

$$(\mathit{car}\ x = 3)\ \wedge\ (\mathit{cdr}\ x = \mathit{null})\ \supset\ (x = \mathit{cons}\ 3\ \mathit{null})$$

where x is of type ι. This is to be contrasted with constructive type theories like that of [15], where to type an application of *car*, that function would have to be applied to an inclusion $i(x)$, not just x. This inclusion will only be typable if *cons?* x is provable, but that is not the case here. Hence, formulas like the above would not even be typable in such type theories, let alone valid. At a high-level, this is because logical context plays no role in typing in systems like that of [15]. The propositional constants are viewed as any other symbols for purposes of typing. In $\mathrm{PF}_{\mathrm{sub}}$, whether or not subset constraints are satisfied in part of an expression is allowed to depend on the logical context determined by the rest of the expression.

Let us compare $\mathrm{PF}_{\mathrm{sub}}$ with PVS. In $\mathrm{PF}_{\mathrm{sub}}$, the formulas mentioned in the Introduction are provably equivalent, assuming suitable definitions and axioms. They are not provably equivalent in PVS. Furthermore, consider the following formula:

$$1/i > 0 \;\supset\; i \neq 0$$

Following the definitions in [20], the TCC for this formula is $i \neq 0$, which is not valid. Hence, this formula is not provable in PVS. It is easily provable in $\mathrm{PF}_{\mathrm{sub}}$, however, by the following argument. Let us assume $1/i > 0$. For this to be true, it must be the case that $i \neq 0$. This is because if $i = 0$, the term $1/i$ is undefined, and hence the assumption is false.

3 Basic proof theory of $\mathrm{PF}_{\mathrm{sub}}$

In this section we prove that modus ponens is derivable, and show a few other basic derived rules. Using modus ponens, we can derive the Tautology Theorem, which states that all propositional tautologies with the usual propositional connectives including $=_o$ are derivable. Using the Tautology Theorem, the Deduction Theorem can be derived, which states that $\Gamma, \phi \vdash \psi$ implies $\Gamma \vdash \phi \supset \psi$. The proofs of the Tautology Theorem and the Deduction Theorem may be found in the Appendix. In the following derivation of modus ponens, whenever the typing context is elided from a logical sequent, it is Σ; and whenever the logical context is elided, it is empty.

Derivation ($\simeq$-refl): Reflexivity of $\simeq$ is derived by using (β-reduction-o) twice to derive two copies of $(\lambda\, x : o.\, A)\ \mathbf{T}\ \simeq_{e(\alpha)}\ A$ where x is not free in A and $\Sigma \Vdash A : \alpha$. Then (replace) is used to replace the left hand side of one of the copies of the equation with A, to get $A \simeq_{e(\alpha)} A$. We can use (replace) because $e(o) \equiv o$. □

Derivation ($\simeq$-symm): Symmetry of $\simeq$ is derived by using (replace) with $\Gamma \vdash A \simeq_{e(\alpha)} B$ to replace the left occurrence of A in $A \simeq_{e(\alpha)} A$. □

Derivation ($=_o$-refl): Reflexivity of $=_o$ follows from ($\simeq$-refl) and (weak equality-o) using (replace). □

Derivation ($=_o$-replace): We can derive a version of (replace) that uses left premise $A =_o B$ instead of $A \simeq_o B$ by using (weak equality-o), ($\simeq$-symm), and (replace) to get $A \simeq_o B$, and then using (replace) to get the desired conclusion from the right premise. □

Derivation ($=_o$-symm): Symmetry of $=_o$ follows from ($=_o$-refl) and ($=_o$-replace). $\square$

Derivation ($\wedge$E-r): From $\Gamma \vdash p \ \wedge \ q$ we can derive $\Gamma \vdash q$. Recall that $p \ \wedge \ q$ abbreviates $\lambda C : o \to o \to o. C \ p \ q \ =_o \lambda C : o \to o \to o. C \ \mathbf{T} \ \mathbf{T}$. We use two instances of (β-reduction-$\wedge$) which apply the left and right sides of the above equation, respectively. Then we get $q =_o \mathbf{T}$ by using ($\simeq_o$-replace) twice with the results of the (β-reduction-$\wedge$)s on the expansion of $p \ \wedge \ q$. Now we use ($=_o$-symm) to get $\mathbf{T} =_o q$, and then ($=_o$-replace) with that equation and $\mathbf{T}$, which we have by axiom ($\mathbf{T}$). This gives us q, as we desired. $\square$

Derivation (modus ponens): Since $p \supset q$ is an abbreviation for $p = (p \wedge q)$, we first apply ($=_o$-replace) to the assumption p to get $p \wedge q$. Then we use ($\wedge$E-r). $\square$

We carry out a few other derivations to show how basic reasoning is done in $\mathrm{PF}_{\mathrm{sub}}$.

Derivation ($\forall$E): This rule of universal instantiation is also derivable,

$$\frac{\Sigma \Vdash M :_1 A' \quad \Gamma \vdash M \downarrow \quad \Gamma \vdash M \lhd ? A \quad \Gamma \vdash \forall \, x :_1 A. \, \phi}{\Gamma \vdash [M/x]\phi}$$

where $e(A') \equiv e(A)$ and $\lhd ?$ is $\unlhd$ if $:_1$ is $:$ and $\lhd$ otherwise. We use (replace) with the fourth premise on a suitably weakened instance of ($=_o$-refl) for $(\lambda \, x :_1 A. \, \phi) \ M$, followed by two uses of (β-reduction) and (replace). The (β-reduction)s are enabled by (modus ponens) and several premises. This gives $\Gamma \vdash [M/x]\phi =_o \mathbf{T}$. We use ($\mathbf{T}$), ($=_o$-symm), and ($=_o$-replace) to get the desired conclusion. $\square$

Derivation (subst-keep): Now that (modus ponens) is available, we can derive the following substitution rule. Suppose $e(\alpha) \equiv \alpha$, and $x :_1 \gamma$ is the last declaration in Σ, but x is not free in Γ or α. Then

$$\frac{\Gamma \vdash C \lhd ? \gamma \quad \Gamma \vdash C \downarrow \quad \Sigma \, ; \Gamma \vdash A \simeq_\alpha B}{\Gamma \vdash [C/x]A \simeq_\alpha [C/x]B}$$

where $\lhd ?$ is $\unlhd$ if $:_1$ is $:$ and $\lhd$ otherwise. The (replace) rule is used to replace the second occurrence of A by B in the following instance of ($\simeq$-refl), which has been suitably weakened with Γ: $\Gamma \vdash (\lambda \, x :_1 \gamma. \, A) \ C \ \simeq_\alpha \ (\lambda \, x :_1 \gamma. \, A) \ C$. Then it uses equations obtained with (modus ponens) and two of the premises on (β-reduction), and then (replace). This derives $\Gamma \vdash [C/x]A \ \simeq_\alpha \ [C/x]B$. $\square$

Derivation (subst): From all the premises of (subst-keep) together with the additional premise that C does not contain the replaced variable x free, we can derive the same conclusion as (subst-keep), but without retaining the declaration of x in the typing context. We just use (generalize) to get $\Gamma \vdash \forall \, x :_1 \gamma. \, [C/x]A \simeq_\alpha [C/x]B$, and then we use ($\forall$E) to instantiate that quantified formula with C. Since the quantified formula contains no free occurrences of x, its instantiation is just $[C/x]A \simeq_\alpha [C/x]B$. But the variable x has been removed from the typing context. $\square$

4 Subtyping in $\mathrm{PF_{sub}}$

In this section, a subtyping relation $<:$ between types is defined, and standard subtyping rules are shown to be derivable. For types A and B with $e(A) \equiv e(B)$, define $A <: B$ to be an abbreviation for

$$\forall\, x : e(A).\, x \trianglelefteq A \;\supset\; x \trianglelefteq B$$

For types A and B with $e(A) \not\equiv e(B)$, we just define $A <: B$ to be $\mathbf{F}$. This makes $e(A) \equiv e(B)$ a necessary condition for $A <: B$, which is reasonable in our system.

Figure 5 gives subtyping rules in terms of $<:$. Theorem 1 states that these are derivable in $\mathrm{PF_{sub}}$. The rules generalize standard ones like those in Section 10.2 of [18] and, for dependent function types, [3], by giving subtyping rules for subset types under logical hypotheses. The formal semantics of PVS presented in [20] does not define a subtyping relation between types. A further difference is that in PVS, subtyping is not contravariant in the domain of functional types, whereas the proof below of derivability of rule ($<:$-Π) shows that subtyping is contravariant in $\mathrm{PF_{sub}}$. We first make this observation:

Lemma 1 *The rule ($<:$-$|$) is not invertible: the conclusion can hold where the premises fail to hold.*

Proof. A counter-example to invertibility is given by the following two types, for predicate symbols P and Q: $\iota|P|Q$ and $\iota|Q|P$. Clearly, $\iota|P \;<:\; \iota|Q$ is not valid. A canonical form for type expressions could probably defined in such a way that ($<:$-$|$) is invertible, but this is not necessary. The above definition of $<:$ makes these two types subtypes of each other without putting them in a canonical form. □

$$(<:\text{-refl})\ \frac{}{\Gamma \vdash A <: A}$$

$$(<:\text{-trans})\ \frac{\Gamma \vdash A <: B \qquad \Gamma \vdash B <: C}{\Gamma \vdash A <: C}$$

$$(<:\text{-}\Pi)\ \frac{\Sigma\,;\,\Gamma \vdash A' <: A \qquad \Sigma, y : A\,;\,\Gamma \vdash [y/x]B <: [y/x]B'}{\Sigma\,;\,\Gamma \vdash \Pi\, x : A.\, B <: \Pi\, x : A'.\, B'}$$

$$(<:\text{-}|)\ \frac{\Gamma \vdash A <: A' \qquad \Gamma \vdash \forall\, x : A.\, (P\ x) \;\supset\; (P'\ x)}{\Sigma\,;\,\Gamma \vdash A|P <: A'|P'}$$

$$(\text{subsume})\ \frac{\Gamma \vdash M\!\downarrow \qquad \Gamma \vdash M \trianglelefteq A \qquad \Gamma \vdash A <: A'}{\Gamma \vdash M \trianglelefteq A'}$$

Fig. 5. Subtyping rules derivable in $\mathrm{PF_{sub}}$

Theorem 1 (derivability of subtyping rules) *The subtyping rules in Figure 5 are derivable in* $\mathrm{PF}_{\mathrm{sub}}$.

Proof. Omitting some details, we prove derivability of ($<:$-Π). Expanding the definition of $<:$ and using the Deduction Theorem, it suffices to prove $(f\ x) \trianglelefteq B'$ using the following assumptions (both typing assumptions and logical ones), in addition to the premises of the rule:

1. $f : \Pi\, x : e(A).\, e(B(x))$
2. $\forall\, x : e(A).\, x \trianglelefteq A \ \wedge\ (f\ x)\!\downarrow\ \supset\ (f\ x) \trianglelefteq B$
3. $x : e(A')$
4. $x \trianglelefteq A'$
5. $f\ x\!\downarrow$

From the left premise of ($<:$-Π), and (4) we get $x \trianglelefteq A$. We use this together with (5) to conclude $(f\ x) \trianglelefteq B$ from (2). Using (generalize), we conclude $\forall\, y : A.\, [y/x]B <: [y/x]B'$ from the right premise of ($<:$-Π). We instantiate this using $x \trianglelefteq A$ to get $B <: B'$. From this and the previously deduced fact $(f\ x) \trianglelefteq B$, we conclude $(f\ x) \trianglelefteq B'$, which we desired to prove.

The derivations of the other rules of Figure 5 are easier. The need for the left premise in (subsume) is demonstrated by the following example. Suppose $M : \iota$ and $M\!\uparrow$ holds. Clearly $M \trianglelefteq \iota$, since this just abbreviates $\mathbf{T}$. It is also easily seen that $\iota <: (\iota | \lambda\, x : \iota.\, \mathbf{T})$ holds. But $M \trianglelefteq (\iota | \lambda\, x : \iota.\, \mathbf{T})$ is equivalent to $M\!\downarrow$. Hence, as long as $\mathrm{PF}_{\mathrm{sub}}$ is consistent, the latter cannot be derived without contradicting the assumption $M\!\uparrow$. So derivability of (subsume) requires $M\!\downarrow$. □

Note that a rule similar to ($<:$-Π) for subtyping Π-abstractions of the form $\Pi\, x :: A.\, B$ is not derivable. Formally, a modified version of the proof above for ($<:$-Π) gets stuck where we would try to conclude $x \triangleleft A$ from $x \trianglelefteq A$, which is not in general valid. Intuitively, $\Pi\, x :: A.\, B$ is the type for functions which take in an x of exactly type A and return a B. Contravariance should not hold for such types.

5 Reduction to PF

In this section, we show that a sequent S is derivable in $\mathrm{PF}_{\mathrm{sub}}$ iff $t(S)$ is derivable in PF, where $t()$ is a translation eliminating subset types.

5.1 Translation from $\mathrm{PF}_{\mathrm{sub}}$ to PF

The definition of a translation $t_\Sigma()$, which forms the basis for our translation of sequents, is given in Figure 6. This function $t_\Sigma()$ is used to get rid of subset types. We omit the subscript Σ when it is the empty context. For simplicity, we assume that bound variables have been named in such a way that the context Σ that is built up contains at most one declaration for any symbol x. This can readily be done. Recall that we require $A \equiv e(A)$ to type I_A, so we do not need to translate definite description operators to anything other than themselves.

$$
\begin{array}{lll}
t_\Sigma(c) & := c, \text{ if } c \in \Sigma \\
t_\Sigma(=_A) & := \lambda\, x : e(A).\, \lambda\, y : e(A). \\
& \quad x \lhd A \ \wedge \ y \lhd A \ \wedge \ x =_{e(A)} y \\
t_\Sigma(I_A) & := I_A \\
t_\Sigma(M\ N) & := (t_\Sigma(M)\ t_\Sigma(N))
\end{array}
$$

$$
\frac{\Sigma \Vdash \lambda\, x : A.\, M : \Pi\, x : A.\, B}{t_\Sigma(\lambda\, x :_1 A.\, M) := \lambda\, x : e(A).\, (u_{e(B)}\ t_\Sigma(x \lhd ? A)\ t_{\Sigma+}(M))}
$$

where

$\Sigma+ \ \equiv \ \Sigma, x :_1 A$

$\lhd?$ is $\unlhd$ if $:_1$ is $:$ and $\lhd$ otherwise

$(u_B\ \phi\ t) \ := \ (I_B\ (\lambda\, y : B.\, \phi \ \wedge \ y = t))$.

Fig. 6. Definition of the translation $t()$

Sequents S of $\mathrm{PF_{sub}}$ are translated into sequents $t(S)$ of PF as follows. A logical sequent $\Sigma\,;\,\Gamma \vdash \phi$ is translated to

$$e(\Sigma)\,;\, t(m(\Sigma)), t_\Sigma(\Gamma) \vdash t_\Sigma(\phi).$$

where $t_\Sigma()$ is extended homorphically to logical contexts; the function $e()$ is extended in the natural way to erase all subset types from typing contexts and turn $::$ into $:$ everywhere; and $m()$ is defined to map typing contexts to logical contexts like this:

Definition 5.1

$$
\begin{aligned}
m(\Delta, x : A) &:= m(\Delta), (x \unlhd A) \\
m(\Delta, x :: A) &:= m(\Delta), (x \lhd A)
\end{aligned}
$$

5.2 Elimination theorem

The elimination theorem is then as follows, where we write $S \in X$ to indicate that sequent S is derivable in system X.

Theorem 2 (Elimination) *Let S be $\Gamma\,;\,\Sigma \vdash \phi$. Then $S \in \mathrm{PF_{sub}}$ iff $t(S) \in \mathrm{PF}$.*

The outline for the proof is the following. The first step is to show that $\Sigma\,;\,\Gamma \vdash \phi$ is derivable in $\mathrm{PF_{sub}}$ iff the following also is:

$$e(\Sigma)\,;\,\Gamma, m(\Sigma) \vdash \phi$$

This shows how to eliminate subsets from the typing context Σ. Second, we prove that $\Sigma \Vdash M : A$ implies $\vdash M \simeq_{e(A)} t_\Sigma(M)$. If the translation of a sequent is derivable in PF, it certainly is derivable in $\mathrm{PF_{sub}}$. So we can simply use the fact that terms equal their translations to replace translations with terms. This shows that a sequent with no subset types in its typing context is derivable in $\mathrm{PF_{sub}}$ if its translation is derivable in PF. The converse of this is proved by induction on the structure of $\mathrm{PF_{sub}}$ derivations.

A crucial lemma used in the first step is the following.

Lemma 2 (Weakening at a type declaration) *If $\Sigma_1, x :_1 e(A), \Sigma_2 ; \Gamma \vdash \phi$ is derivable in* $\mathrm{PF}_{\mathrm{sub}}$*, then so is $\Sigma_1, x :_1 A, \Sigma_2 ; \Gamma \vdash \phi$, as long as $\Sigma_1, x :_1 A, \Sigma_2$ is a valid typing context.*

For this lemma to go through smoothly, the rules of Figure 4 have been carefully designed so that except for (|-outer), types A in the premises of the rule which could be changed by changing the type of a symbol in the typing context do not appear in the conclusion unless in $e(A)$. For example, in (β-reduction), two types in the premises appear in the conclusion, namely the type A of the λ-bound variable, and the type α of the β-redex. But in the conclusion α appears just in an expression $e(\alpha)$, and the type of a λ-bound variable cannot change by strengthening a type declaration in the typing context.

The only exception to this property is in the rule (|-outer). In that case, suppose $M :_2 B$ is derivable in the context with $x :_1 e(A)$, and $M :_2 B'$ is derivable in the context with $x :_1 A$. We argue by induction on the structure of types that $M \lhd ? B'$ implies $M \lhd ? B$, where $\lhd ?$ is the appropriate one of $\{\unlhd, \lhd\}$. This shows how to derive the conclusion of (|-outer) when a type declaration has been strengthened (thus weakening the whole sequent).

6 Meta-theory of PF

We take PF to be $\mathrm{PF}_{\mathrm{sub}}$ without the typing rule (|), without the logical rules (|-outer) and (|-=), and with the conclusion of (β-reduction) not requiring $N \lhd A$ or $N \unlhd A$. Also, we can drop uses of $e()$, since no types contain subset constraints. Similarly, we need only the single type classifier :. We can drop ::. The logical rule (|-outer) is clearly redundant in the absence of (|), since it is easily shown that if $A \equiv e(A)$, $M \unlhd A$ and $M \lhd A$ are both equivalent to $\mathbf{T}$. The basic proof theory of $\mathrm{PF}_{\mathrm{sub}}$ as developed in Section 3 continues to hold.

The model theory of PF can then be developed following [8], where it is carried through in detail. One minor difference is that PF as presented in [8] requires the interpretations of all predicates to be defined. Here, we have simplified matters somewhat by insisting just that the interpretations of formulas are defined. The interpretation of a predicate may be undefined. For this reason, we have taken the rule (o convergence) in Figure 4 instead of a rule stating that all predicates are defined.

7 Conclusion

A uniform system $\mathrm{PF}_{\mathrm{sub}}$ of subset types and partial functions has been studied. This system retains decidability of type-checking by ignoring subset constraints, which are then handled in the logical part of the system. The approach makes certain formulas valid which intuitively seem desirable to have valid, but which are not even typable in systems like that of [15] or have unprovable TCCs in PVS.

Future work could examine further extensions of the type system, for example to include predicative polymorphism. This would eliminate the need for special typing rules for $=_A$ and I_A, since a polymorphic type could be assigned to a single equality symbol and a single definite description operator. One must, of course, be careful to avoid paradoxes like those discussed in [13, 6, 7]. Another interesting extension would be to try to add type constructors. This would be sufficient to yield a type theory similar to LF [14]. In addition to the constraints arising from subset types, one would also get equality constraints when checking whether a type like *array* 6 is equivalent to a type like *array* $(3+3)$. To retain decidability of type checking, these constraints will also need to be handled in the logical part of the system.

It would also be useful to try to get an elimination theorem like the one in Chapter 4 of [24], which reduces validity of ϕ in a first-order logic of partial functions to validity of a translation of ϕ in classical first-order logic. The translation conjoins definedness conditions for the terms of ϕ with the atomic formulas containing those terms. This assumes that function symbols have an associated predicate saying exactly when they are defined and when they are undefined. This result immediately applies to the first-order fragment of PF, and should extend to larger fragments, if not the whole system. A further question is whether or not one could translate from $\mathrm{PF}_{\mathrm{sub}}$ to PF in a reasonable way without using definite description operators, which are quite essential to the translation given here. Another refinement would be to try to optimize the logical rules to remove some of the technical ones like (β-reduction-$\wedge$).

One could consider logics where functions need not be strict. For example, it would be nice to define a non-strict if-then-else operator. In a certain sense, however, PF can already support non-strict operators. While we will not consistently be able to have if-then-else (at type ι, say) as a function symbol and have it be non-strict, we can define ($\mathit{if}\ \phi\ \mathit{then}\ M\ \mathit{else}\ N$) as an abbreviation for $(\mathit{ite}\ \phi\ (\lambda\, x : o.\, M)\ (\lambda\, x : o.\, N))$, where *ite* is a function symbol with axioms

$$\begin{array}{lcl} \phi & \supset & (\mathit{ite}\ \phi\ F\ G) \ \simeq\ (F\ \mathbf{T}) \\ \neg\phi & \supset & (\mathit{ite}\ \phi\ F\ G) \ \simeq\ (G\ \mathbf{T}) \end{array}$$

Since λ-abstractions are always defined in PF, these definitions will allow an if-then-else expression to be defined even in cases where the then- or else-parts are undefined.

The author thanks Nikolaj Björner for valuable critcism and encouragement about this work, Bill Farmer for numerous profitable exchanges about PF, and John Mitchell for encouragement about studying type theory with subset types.

References

1. P. Andrews. *An Introduction to Mathematical Logic and Type Theory: to Truth through Proof.* Academic Press, 1965.
2. P. Andrews. *A Transfinite Type Theory with Type Variables.* North-Holland, 1965.
3. D. Aspinall and A. Compagnoni. Subtyping dependent types. *Theoretical Computer Science*, 266(1-2), 2001.

4. M. Beeson. *Foundations of Constructive Mathematics: Metamathematical Studies.* Springer, 1985.
5. A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
6. T. Coquand. An analysis of Girard's paradox. In *1st Symposium on Logic in Computer Science*, pages 227–236, 1986.
7. T. Coquand. A new paradox in type theory. In *9th International Conference on Logic, Methodology, and Philosophy of Science*, pages 555–570, 1994.
8. W. Farmer. A partial functions version of Church's simple theory of types. *The Journal of Symbolic Logic*, 55(3):1269–91, 1990.
9. W. Farmer. A simple type theory with partial functions and subtypes. *Annals of Pure and Applied Logic*, 64(3), 1993.
10. W. Farmer and J. Guttman. A Set Theory with Support for Partial Functions. *Logica Studia*, 66(1):59–78, 2000.
11. S. Feferman. Definedness. *Erkenntnis*, 43:295–320, 1995.
12. H. Geuvers. *Logics and Type systems.* PhD thesis, University of Nijmegen, September 1993.
13. J.Y. Girard. Une extension de l'interpretation de Gödel a l'analyse, et son application a l'elimination des coupures dans l'analyse et la theorie des types. In *2nd Scandinavian Logic Symposium*, pages 63–92, 1971.
14. R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.
15. B. Jacobs. *Categorical Logic and Type Theory.* Elsevier, 1999.
16. M. Kerber and M. Kohlhase. A Mechanization of Strong Kleene Logic for Partial Functions. In A. Bundy, editor, *12th International Conference on Automated Deduction*, volume 814 of *LNAI*, pages 371–385. Springer Verlag, 1994.
17. M. Kerber and M. Kohlhase. Mechanising Partiality without Re-Implementation. In *21st Annual German Conference on Artificial Intelligence*, volume 1303 of *LNAI*, pages 123–134. Springer Verlag, 1997.
18. J. Mitchell. *Foundations for Programming Languages.* The MIT Press, 1996.
19. S. Owre, J. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *11th International Conference on Automated Deduction*, volume 607 of *LNAI*, pages 748–752. Springer-Verlag, 1992.
20. Sam Owre and Natarajan Shankar. The formal semantics of PVS. http://www.csl.sri.com/papers/csl-97-2/, March 1999.
21. F. Pfenning. *Logical Frameworks*, chapter XXI. Volume 2 of Robinson and Voronkov [22], 2001.
22. A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning.* Elsevier and MIT Press, 2001.
23. J. Rushby, S. Owre, and N. Shankar. Subtypes for Specifications: Predicate Subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, September 1998.
24. A. Stump. *Checking Validities and Proofs with CVC and flea.* PhD thesis, Stanford University, 2002. available from http://www.cs.wustl.edu/˜ stump/.
25. A. Stump. Subset types and partial functions. *draft paper available from author's homepage*, 2003.

A Proofs of the Tautology and Deduction Theorems

The proofs of the Tautology Theorem and the Deduction Theorem follow [1] closely.

A.1 Tautology Theorem

Derivation (=-refl): If $\alpha \equiv e(\alpha)$, we can derive $\Gamma \vdash A =_\alpha A$ from $\Gamma \vdash A \trianglelefteq \alpha$ and $\Gamma \vdash A\downarrow$ as follows. We get $(A \simeq_\alpha A) \;\simeq_o\; (A =_\alpha A)$ from (weak equality) using (subst). Then we use ($\simeq$-refl) and (replace). □

Derivation ($\simeq$-to-=): Suppose we have the following:

- $e(\alpha) \equiv \alpha$
- $\Gamma \vdash A_i \trianglelefteq \alpha$, for $1 \leq i \leq 2$
- $\Gamma \vdash A_i\downarrow$, for $1 \leq i \leq 2$

Then from $\Gamma \vdash A_1 \simeq_\alpha A_2$, we can derive $\Gamma \vdash A_1 =_\alpha A_2$ by using (subst) twice to get $\Gamma \vdash (A_1 \simeq_\alpha A_2) \simeq_o (A_1 =_\alpha A_2)$ from (weak equality), suitably weakened; and then using (replace). Similar reasoning allows us to conclude $\Gamma \vdash A_1 \simeq_\alpha A_2$ from $\Gamma \vdash A_1 =_\alpha A_2$. □

Derivation (=-replace): We can now derive a version of (replace) that uses left premise $A_1 =_\alpha A_2$ instead of the premise with $\simeq_\alpha$ by using ($\simeq$-to-=) to get $A_1 \simeq_\alpha A_2$, and then using (replace) to get the desired conclusion from the right premise. This requires $\Gamma \vdash A_i \trianglelefteq \alpha$ and $\Gamma \vdash A_i\downarrow$, for $1 \leq i \leq 2$, as well as $e(\alpha) \equiv \alpha$. □

Derivation ($\boldsymbol{T}$=-refl): We derive $\Gamma \vdash \mathbf{T} =_o (A =_\alpha A)$ from $\Gamma \vdash A \trianglelefteq \alpha$ and $\Gamma \vdash A\downarrow$, where $e(\alpha) \equiv \alpha$, as follows. Let Id temporarily abbreviate $\lambda\, y : \alpha .\, y$. By (extensionality) and (subst), we have $\Gamma \vdash (Id =_{\alpha \to \alpha} Id) =_o \forall\, x : \alpha .\, Id\, x \simeq_\alpha Id\, x$. For this inference to go through, we need a proof of $Id\downarrow$, which we get by (λ-convergence); and a proof of $Id \trianglelefteq \alpha \to \alpha$, which we get by (|-outer) and (modus ponens). Then using ($=_o$-replace) on the equation just derived and an instance of (=-refl), we get $\Gamma \vdash \forall\, x : \alpha .\, Id\, x \simeq_\alpha Id\, x$. We next get an instance of (=-refl) enabled by

$$((\lambda\, x : \alpha .\, Id\, x \simeq_\alpha Id\, x)\ A)\downarrow,$$

which we have by (o-convergence); and then use (replace) on that instance and the equation derived just before, making use of the definition of $\forall$, to get

$$(\lambda\, x : \alpha .\, \mathbf{T})\ A \;=_o\; (\lambda\, x : \alpha .\, Id\, x \simeq_\alpha Id\, x)\ A.$$

Then using (β-reduction) and (modus ponens) twice with the hypotheses about A, we get $\mathbf{T} =_o (Id\ A \simeq_\alpha Id\ A)$. Another ($\beta$-reduction) and two more uses of (replace), followed by ($\simeq$-to-=) and (replace), give the desired conclusion. □

Derivation ($\wedge I$-$=_o$): We can derive $\Gamma \vdash (A =_o B) \;\wedge\; (C =_o D)$ from $\Gamma \vdash A =_o B$ and $\Gamma \vdash C =_o D$ as follows. We use ($=_o$-replace) on the first hypothesis and $\mathbf{T} =_o (A =_o A)$, which we have by ($\mathbf{T}$ =-refl); and similarly for $\mathbf{T} =_o (C =_o C)$ with the second hypothesis. We then use two instances of ($=_o$-replace) with the results and $\mathbf{T} \wedge \mathbf{T}$, which we have by (=-refl) and the definition of $\wedge$. □

Derivation ($\boldsymbol{T} \wedge \boldsymbol{T}$): We derive $(\mathbf{T} \wedge \mathbf{T}) =_o \mathbf{T}$ as follows. Let Z abbreviate $\lambda\, y : o .\, \mathbf{T}$. By (subst) on (truth values), we get $((Z\ \mathbf{T}) \;\wedge\; (Z\ \mathbf{F})) =_o \forall\, x : o .\, Z\ x$. Using three ($\beta$-reduction-o)s and three (replace)s, we get $(\mathbf{T} \wedge \mathbf{T}) =_o \forall\, x : o .\, \mathbf{T}$. To conclude, we use ($=_o$-replace) on this formula with $(\forall\, x : o .\, \mathbf{T}) =_o \mathbf{T}$, which we get as follows. We instantiate ($\mathbf{T}$ =-refl) using $\lambda\, x : o.\, \mathbf{T}$ and then use ($=_o$-symm). This gives us $(\lambda\, x : o.\, \mathbf{T} =_{o\to o} \lambda\, x : o.\, \mathbf{T}) =_o \mathbf{T}$. We can use (subst) on (weak equality) and then (replace) to change the $=_{o\to o}$ on the left hand side of this equation into $\simeq_{o\to o}$. This gives us $(\forall\, x : o.\, \mathbf{T}) =_o \mathbf{T}$, by the definition of $\forall$. To use (subst), we need $\lambda\, x : o.\, \mathbf{T}\!\downarrow$, which we have by ($\lambda$-convergence), and $\lambda\, x : o.\, \mathbf{T} \;\trianglelefteq\; o \to o$, which we have by ($|$-outer) and (modus ponens). $\square$

*Derivation (**F**2):* We derive $\mathbf{F} =_o \forall\, x : o.\, x$ from an instance of ($=_o$-refl) for $\forall\, x : o.\, x$ using (replace) with the equation

$$\forall\, x : o.\, x \simeq_o F$$

which, by the definitions of $\mathbf{F}$ and $\forall$, is derived using (subst) on (weak equality). This use of (subst) requires $\lambda\, x : o.\, \mathbf{T}\!\downarrow$ and $\lambda\, x : o.\, x\!\downarrow$, which we have by ($\lambda$-convergence); and also $\lambda\, x : o.\, \mathbf{T} \trianglelefteq o \to o$ and $\lambda\, x : o.\, x \trianglelefteq o \to o$. The latter two formulas are obtained using ($|$-outer) and (modus ponens). $\square$

*Derivation (**T** ∧ **F**):* We derive $(\mathbf{T} \wedge \mathbf{F}) =_o \mathbf{F}$ as follows. Let Id abbreviate $\lambda\, y : o.\, y$. By (subst) on (truth values), we get $((Id\, \mathbf{T}) \;\wedge\; (Id\, \mathbf{F})) =_o \forall\, x : o.\, Id\, x$. Using three ($\beta$-reduction-o)s and three (replace)s, we get $(\mathbf{T}\wedge\mathbf{F}) =_o \forall\, x : o.\, x$. We next use ($=_o$-symm) on ($\mathbf{F}$2) and then ($=_o$-replace) to get the desired formula. $\square$

Derivation (cases): The following is derivable:

$$\frac{\Sigma \Vdash x : o \qquad \Gamma \vdash [\mathbf{T}/x]A \;\wedge\; [\mathbf{F}/x]A}{\Gamma \vdash A}$$

From our logical premise we get $\Gamma \vdash ((\lambda\, x : o.\, A)\; \mathbf{T}) \;\wedge\; ((\lambda\, x : o.\, A)\; \mathbf{F})$ by first using (replace) with $\Gamma \vdash [\mathbf{T}/x]A \simeq_o (\lambda\, x : o.\, A)\; \mathbf{T}$, which we get by (β-reduction-o) and ($\simeq$-symm); and then using (replace) with a similar equation that has $\mathbf{F}$ instead of $\mathbf{T}$. Then we use (subst) to obtain a suitable instance of the axiom (truth values), which we use with (replace) and the conjunction just derived to obtain $\Gamma \vdash \forall\, x : o.\, (\lambda\, x : o.\, A)\; x$. Note that to use (subst), we need a proof of $\lambda\, x : o.\, A\!\downarrow$, which we get by ($\lambda$-convergence); and a proof of $\lambda\, x : o.\, A \trianglelefteq o \to o$, which we get using ($|$-outer). We use ($\forall$E) to get $(\lambda\, x : o.\, A)\; x$, and then ($\beta$-reduction-o) to get A. $\square$

*Derivation (**T**∧):* If $\Sigma \Vdash A : o$, we can derive $(\mathbf{T} \wedge A) =_o A$ as follows. We conjoin the formulas of ($\mathbf{T} \wedge \mathbf{T}$) and ($\mathbf{T} \wedge \mathbf{F}$) using ($\wedge$I-$=_o$), and then use (cases) followed by (subst). $\square$

*Derivation ((**T** = **F**) = **F**):* We derive $(\mathbf{T} =_o \mathbf{F}) =_o \mathbf{F}$ as follows. Let Z temporarily abbreviate $\lambda\, x : o.\, \mathbf{T} =_o x$. By (subst) on (truth values), we get $((Z\;\; \mathbf{T}) \;\wedge\; Z\;\; \mathbf{F}) =_o \forall\, x : o.\, Z\;\, x$. From this we get $((\mathbf{T} =_o \mathbf{T}) \;\wedge\; (\mathbf{T} =_o \mathbf{F})) \;=_o\; \forall\, x : o.\, \mathbf{T} =_o x$ by three uses of (β-reduction-o) and (replace). By (subst) on ($\mathbf{T}$ =-refl) followed by ($=_o$-symm), we get $(\mathbf{T} =_o \mathbf{T}) =_o \mathbf{T}$, which we use with ($=_o$-replace) on the last derived equation to get $(\mathbf{T} \;\wedge\; (\mathbf{T} =_o \mathbf{F})) \;=_o\; \forall\, x : o.\, \mathbf{T} =_o x$. We can simplify this to $(\mathbf{T} =_o \mathbf{F}) =_o \forall\, x : o.\, \mathbf{T} =_o x$ using ($=_o$-replace) and ($\mathbf{T}\wedge$). We then conclude by using ($=_o$-replace) on this last equation and $(\forall\, x : o.\, \mathbf{T} =_o x) =_o \mathbf{F}$, which we derive as follows. We can use (subst) on (extensionality), followed by two (β-reduction-o)s and two (replace)s to get $((\lambda x : o.\,\mathbf{T}) =_{o\to o} (\lambda x : o.\,x)) =_o \forall x : o.\,(\mathbf{T} =_o x)$. The left hand side of this equation is the definition of $\mathbf{F}$, so we just need ($=_o$-symm) to get $\forall x : o.\,(\mathbf{T} =_o x) =_o \mathbf{F}$. $\square$

Derivation ($\boldsymbol{T}$ ==): We derive $(\mathbf{T} =_o A) =_o A$ by first conjoining $(\mathbf{T} =_o \mathbf{T}) =_o \mathbf{T}$ and $(\mathbf{T} =_o \mathbf{F}) =_o \mathbf{F}$ using ($\wedge$I-$=_o$). The first conjunct we get by (subst) on ($\mathbf{T}$ =-refl) followed by ($=_o$-symm). The second is from $((\mathbf{T} = \mathbf{F}) = \mathbf{F})$. Then we use (cases) on the conjunction, followed by (subst). $\square$

Derivation ($\boldsymbol{F}\wedge$): We prove $(\mathbf{F} \wedge A) =_o \mathbf{F}$ as follows. First, note that by the definition of $\supset$ and ($=_o$-symm), it suffices to prove $\mathbf{F} \supset A$. Recall that $\mathbf{F}$ abbreviates $(\lambda x : o.\,\mathbf{T} =_{o\to o} \lambda x : o.\,x)$. We can use (subst) on (Leibniz) with these two functions to get (writing ζ for $o \to o$)

$$\mathbf{F} \;\supset\; ((\lambda f : \zeta.\, f\ A)\ \lambda x : o.\,\mathbf{T} =_o (\lambda f : \zeta.\, f\ A)\ \lambda x : o.\,x).$$

where f does not occur in A. By (β-reduction-o) and (replace) four times, we get $\mathbf{F} \supset (\mathbf{T} =_o A)$. Then by ($\mathbf{T}$ ==) and ($=_o$-replace), we get the desired conclusion. $\square$

Derivation (Tautology Theorem): Suppose A is a propositional formula; i.e., one built just from propositional variables, $\mathbf{T}$, $\mathbf{F}$, and propositional connectives $=_o$, $\neg$, $\wedge$, $\vee$, and $\supset$. If A is a propositional tautology in the usual sense, then $\vdash A$ is derivable. This is proved by induction on the number of (propositional) variables in A. If there are $n{+}1$ such including x, then $[\mathbf{T}/x]A \wedge [\mathbf{F}/x]A$ is clearly also a tautology, and the induction hypothesis applies, so $\vdash [\mathbf{T}/x]A \wedge [\mathbf{F}/x]A$. Then (cases) can be used to conclude A. If there are no variables in A, then we prove by induction on the structure of A that $\vdash A =_o \mathbf{T}$ holds if A evaluates to $\mathbf{T}$, and $\vdash A =_o \mathbf{F}$ holds if A evaluates to $\mathbf{F}$. The result will then follow by ($\mathbf{T}$), ($=_o$-symm), and ($=_o$-replace).

If A is $\mathbf{T}$, then we get $A =_o \mathbf{T}$ by ($=_o$-refl); and similarly for $\mathbf{F}$. Suppose A is $\neg A'$ for some A'. If A' evaluates to $\mathbf{F}$, then by the induction hypothesis, $\vdash A' =_o \mathbf{F}$. We then use ($=_o$-replace) on this equation and $(A' =_o \mathbf{F}) =_o (\mathbf{T} =_o (A' =_o \mathbf{F}))$, which we get by (subst) from ($\mathbf{T}$ ==), followed by a ($=_o$-symm). We use ($=_o$-symm) to get $(A' =_o \mathbf{F}) =_o \mathbf{T}$, which is what we desired to prove. On the other hand, if A' evaluates to $\mathbf{T}$, we get $\vdash A' =_o \mathbf{T}$ by the induction hypothesis, and then $\vdash \mathbf{T} =_o A'$ by ($=_o$-symm). We then obtain $\vdash \neg A' = \mathbf{F}$ by ($=_o$-replace) with $((\mathbf{T} = \mathbf{F}) = \mathbf{F})$.

Now suppose A is of the form $A_1 \wedge A_2$. Using the induction hypothesis and then ($=_o$-replace) twice, it suffices to prove the four instances of $(P \wedge Q) =_o R$ for $P, Q \in \{\mathbf{T}, \mathbf{F}\}$ and suitable R. We have two instances by ($\mathbf{T}\wedge\mathbf{T}$) and ($\mathbf{T}\wedge\mathbf{F}$). We get the other two instances by (subst) from ($\mathbf{F}\wedge$).

If A is of the form $A_1 \supset A_2$, similar reasoning to the case for $A \equiv A_1 \wedge A_2$ is used. $(\mathbf{F} \supset P) =_o \mathbf{T}$ is obtained from ($\mathbf{F}\wedge$), the definition of $\supset$, and then ($=_o$-replace) with an instance of ($\mathbf{T}$ ==), reversed using ($=_o$-symm). The cases $(\mathbf{T} \supset \mathbf{T}) =_o \mathbf{T}$ and $(\mathbf{T} \supset \mathbf{F}) =_o \mathbf{F}$ follow easily from instances of $\mathbf{T}\wedge$ and similar use of ($\mathbf{T}$ ==).

If A is of the form $A_1 \vee A_2$, we use similar reasoning as in the previous two cases. We use the definition of $\vee$ and then the definition of $\neg$ with either $(\mathbf{T} =_o$

$\mathbf{F}) =_o \mathbf{F}$ or $(\mathbf{F} =_o \mathbf{F}) =_o \mathbf{T}$, which are easily obtained using $((\mathbf{T} = \mathbf{F}) = \mathbf{F})$ and ($\mathbf{T}$ =-refl), respectively. This allows us to reduce the cases for $\vee$ to the cases for $\wedge$.

If A is of the form $A_1 =_o A_2$, similar reasoning using $((\mathbf{T} = \mathbf{F}) = \mathbf{F})$, or ($\mathbf{T}$ ==) and ($\mathbf{T}$ =-refl) works for all cases but $(\mathbf{F} =_o \mathbf{T}) =_o \mathbf{F}$, which is derived as follows. Temporarily let H abbreviate $\lambda\, x : o. \neg x$. By (subst) on (Leibniz), we obtain, $(\mathbf{F} =_o \mathbf{T}) \supset (H\ \mathbf{F} =_o H\ \mathbf{T})$. By two ($\beta$-reduction-o)s and two (replace)s, we then get $(\mathbf{F} =_o \mathbf{T}) \supset ((\mathbf{F} =_o \mathbf{F}) =_o (\mathbf{T} =_o \mathbf{F}))$. The consequent of this implication is easily replaced by just $\mathbf{F}$, using the reasoning for the other subcases of this derivation when A is an equation. By the definition of $\supset$, we then have $(\mathbf{F} =_o \mathbf{T}) =_o ((\mathbf{F} =_o \mathbf{T}) \wedge \mathbf{F})$. We can replace the right hand side of this equation with $\mathbf{F}$, using ($=_o$-replace) with a formula obtained by using (subst) on $(x \wedge \mathbf{F}) =_o \mathbf{F}$. This latter formula is readily proved using (cases), making use of ($\wedge$I-$=_o$) and the reasoning from the part of this derivation for when A is a conjunction. □

A.2 Deduction theorem

In this section we prove the Deduction Theorem, which states that if $\Gamma, \phi \vdash \psi$ is derivable, then so is $\Gamma \vdash \phi \supset \psi$.

Lemma 3 ((sym-rename)) *Suppose $\Sigma \Vdash M : A$ and x' is a symbol not declared in Σ. Let $x :_1 \alpha$ be a declaration in Σ. Then $\Sigma' \Vdash M' : A'$, where Σ' contains $x' :_1 \alpha$ instead of $x :_1 \alpha$, and $y :_2 [x'/x]\gamma$ instead of $y :_2 \gamma$ for all other declarations $y :_2 \gamma$. Similarly, $M' \equiv [x'/x]M$ and $A' \equiv [x'/x]A$.*

Proof. Easy induction on the structure of typing derivations.

Derivation (trivial-$\trianglelefteq$): Suppose $\Sigma \Vdash e(A) : *$ and $\Sigma \Vdash M : A$. Then $\Gamma \vdash (M \trianglelefteq e(A)) =_o \mathbf{T}$ is derived as follows by induction on the structure of A. The cases for $A \equiv \iota$ and $A \equiv o$ follow using ($=_o$-refl) and the definition of $\trianglelefteq$. If A is of the form $A'|P$, we use (strip) and then the induction hypothesis. Suppose A is of the form $\Pi\, x :_1 C.\, D$. For some z not declared in Σ, we can use (weak1) to get $\Sigma, z : e(C) \Vdash M z : [z/x]D$. By the induction hypothesis, we can now get $((M\ z) \trianglelefteq [z/x]D) =_o \mathbf{T}$. By tautological reasoning (i.e., a combination of (modus ponens), (subst), and the Tautology Theorem), followed by (generalize) and a little more tautological reasoning, we then get our desired sequent, where $\triangleleft?$ is the appropriate on of $\{\trianglelefteq, \triangleleft\}$:

$$\Sigma;\ \vdash (\forall\, x : C.\, ((x \triangleleft ? C)\ \wedge\ (M\ x)\downarrow)\ \supset\ (M\ x) \trianglelefteq D) =_o \mathbf{T}$$

□

Lemma 4 ((extended weaken)) *If $\Sigma_1, \Sigma_2\, ;\, \Gamma \vdash \phi$ is derivable and $\Sigma_1, x : A, \Sigma_2$ is a valid typing context, then $\Sigma_1, x : A, \Sigma_2\, ;\, \Gamma \vdash \phi$ is derivable.*

Proof. Easy induction on the structure of derivations. Note that (generalize) allows symbols to be renamed when they are added to the typing context.

Derivation (Deduction Theorem): We proceed by induction on the derivation d of sequent $\Gamma, \phi \vdash \psi$. The derivation cannot be by any logical axiom except (assume), since all logical axioms except (assume) require the logical context to be empty. Suppose d ends in (assume). By the Tautology Theorem we get $\vdash P \supset P$, which we instantiate with (subst) to get $\vdash \psi \supset \psi$. Suppose d ends in (weaken). By the Tautology Theorem we have $P \supset (Q \supset P)$, which we can instantiate with (subst) to get $\psi \supset (\phi \supset \psi)$. By (modus ponens) with this formula and the left premise of the instance of (weaken), we get $\phi \supset \psi$.

Suppose d ends in (generalize), with ϕ of the form $\forall\, x :_1 A.\, \psi'$. By the induction hypothesis, we can derive $\Sigma, x :_1 A\,;\, \Gamma \vdash \phi \supset \psi'$. By the Tautology Theorem we have $(P \supset Q) \supset (\neg P \vee Q)$, which we can instantiate with (subst) and then apply using (modus ponens) to get $\Sigma, x :_1 A\,;\, \Gamma \vdash \neg\phi \vee \psi'$. Then we use (generalize) to get $\Sigma\,;\, \Gamma \vdash \forall\, x :_1 A.\, \neg\phi \vee \psi'$. From this we derive $\Sigma\,;\, \Gamma \vdash \neg\phi \vee \forall\, x :_1 A.\, \psi'$ as follows; note that from this sequent we can get the desired one using the Tautology Theorem and (modus ponens) again. We get $(\forall\, x :_1 A.\, \mathbf{T} \vee \psi') \supset (\mathbf{T} \vee \forall\, x :_1 A.\, \psi')$ by using (subst) on the tautology $P \supset (\mathbf{T} \vee Q)$, proved by the Tautology Theorem. We next get $\forall\, x :_1 A.\, \psi' \supset \forall\, x :_1 A.\, \psi'$ by (subst) on a tautology. With this formula, we use the fact, whose proof is easily obtained from the case for $\vee$ in the proof of the Tautology Theorem, that $(\mathbf{F} \vee A) =_o A$, to get $(\forall\, x :_1 A.\, \mathbf{F} \vee \psi') \supset (\mathbf{F} \vee \forall\, x :_1 A.\, \psi')$ using ($=_o$-replace). We then conjoin this formula with the similar one derived just previously using ($\wedge$I-$=_o$). We then get the desired formula using (cases) and (subst).

Suppose d ends in (replace), with left premise $\Gamma, \phi \vdash X \simeq_A Y$ and right premise $\Gamma, \phi \vdash C$, where $A \equiv e(A)$. By the induction hypothesis, we can get sequents $\Gamma \vdash \phi \supset (X \simeq_A Y)$ and $\Gamma \vdash \phi \supset C$. Let $\{x_1, \ldots, x_n\}$ be the set of variables occurring free in $X \simeq_A Y$ and bound above the occurrence of X replaced in C to get D. If there are no such variables, the reasoning is simpler than what follows, so suppose $n > 0$. The side condition on (replace) ensures that these variables are not free in Γ,ϕ, or the typing context. By induction on n, we can derive

$$\Sigma\,;\, \Gamma \vdash \phi \supset \forall\, x_1 :_1 \alpha_1.\, \ldots \forall\, x_n :_n \alpha_n.\, X \simeq_A Y$$

for suitable types $\alpha_1, \ldots, \alpha_n$, and suitable $:_1, \ldots :_n$; this is done using a similar argument as in the case for (generalize) just above. One minor difference is that we keep Σ the same. This can be done by using (extended weaken). We then use tautological reasoning with this sequent, $\Gamma \vdash \phi \supset C$, and

$$\vdash (\forall\, x_1 :_1 \alpha_1.\, \ldots \forall\, x_n :_n \alpha_n.\, X \simeq_A Y) \ \supset\ (C =_o D)$$

to get $\Gamma \vdash \phi \supset D$. (Recall that $=_o$ in $C =_o D$ is considered a propositional operator.) This last displayed sequent is derived as follows.

Let π be the position of the occurrence of A in C that gets replaced to get D. Let G be C except with $z\ x_1\ \ldots\ x_n$ instead of A at position π. For $Q \in \{X, Y\}$, let F_Q be $\lambda\, x_1 :_1 \alpha_1.\, \ldots \lambda\, x_n :_n \alpha_n.\, Q$. We can use (subst) on (Leibniz) to get

$$(F_X \ =_\zeta\ F_Y) \ \supset\ ((\lambda\, z : \zeta.\, G)\ F_X) =_o ((\lambda\, z : \zeta.\, G)\ F_Y)$$

where ζ abbreviates $\Pi\, x_1 :_1 \alpha_1. \ldots \Pi\, x_n :_n \alpha_n. A$. We use ($\lambda$-convergence) and (trivial-$\trianglelefteq$) to establish the conditions needed to use (subst). In a similar way, we get the formulas needed to use (modus ponens) on (β-reduction) with the left and right hand sides of the equation in the consequent of the displayed implication. Using (replace), we can finally get

$$(F_X \ =_\zeta \ F_Y) \supset (C =_o D)$$

We just need the following and we are done:

$$(F_X \ =_\zeta \ F_Y) =_o \forall\, x_1 :_1 \alpha_1. \ldots \forall\, x_n :_n \alpha_n. X \simeq_A Y$$

We prove this by induction on n as follows. If $n = 1$, we first get

$$(F_X =_\zeta F_Y) =_o \forall\, x_1 : \alpha_1. (F_X \ x_1) \simeq_A (F_Y \ x_1)$$

by (subst) on (extensionality). We easily reduce $(F_X \ x_1)$ to X using (β-reduction), and similarly for $(F_Y \ x_1)$; recall that the variable x_1 was retained in the typing context.

If $n = n' + 1$, we get

$$(F'_X \ =_\zeta \ F'_Y) =_o \forall\, x_1 :_1 \alpha_1. \ldots \forall\, x_{n-1} :_{n-1} \alpha_{n-1}. (X' \simeq_{\Pi\, x_n :_n \alpha_n. A} Y')$$

by the induction hypothesis, where X' is $\lambda\, x_n :_n \alpha_n. X$ and similarly for Y'. Then using ($=_o$-replace) with a similar equation to the one we derived for the base case of the proof, we can replace $\lambda\, x_n :_n \alpha_n. X \simeq_{\Pi\, x_n :_n \alpha_n. A} \lambda\, x_n :_n \alpha_n. Y$ in the displayed formula with $\forall\, x_n :_n \alpha_n. X \simeq_A Y$ to get the desired formula. $\square$