Acta Informatica Manuscript-Nr.
(will be inserted by hand later)

A typed logic of partial functions reconstructed classically

C.B. Jones[1] and C.A. Middelburg[2,3]

[1] Department of Computer Science, University of Manchester, Manchester M13 9PL, England
[2] Department of Computer Science, PTT Research, P.O. Box 421, 2260 AK Leidschendam, The Netherlands
[3] Department of Philosophy, Utrecht University, P.O. Box 80.126, 3508 TC Utrecht, The Netherlands

Received: April 2, 1993 / Accepted: October 4, 1993

Abstract. This paper gives a comprehensive description of a typed version of the logic known as LPF. This logic is basic to formal specification and verified design in the software development method VDM. If appropriately extended to deal with recursively defined functions, the data types used in VDM, etc., it gives the VDM notation and its associated rules of reasoning. The paper provides an overview of the needed extensions and examines some of them in detail. It is shown how this non-classical logic – and the extensions – can be reconstructed classically by embeddings into classical infinitary logic.

1. Introduction

Functions specified in – for example – the VDM notation are in general partial. Thus

$$\begin{array}{l} \mathit{diff} \ : \ \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \\ \mathit{diff}(i,j) \triangleq \text{if } i = j \text{ then } 0 \text{ else } \mathit{diff}(i, j + 1) + 1 \end{array}$$

is a recursive function which computes the difference between two integers *providing* its first argument is greater than or equal to the second. Partial functions can give rise to non-denoting terms in formulae (i.e. terms that do not refer to objects of the intended type) – they are loosely referred to as undefined terms. There are problems when reasoning about partial functions in classical first-order logic. Consider what might appear to be a reasonable formalization of the property above:

$$\forall i, j : \mathbb{Z} \cdot i \geq j \ \Rightarrow \ \mathit{diff}(i, j) = i - j \ .$$

The truth of this plausible formula depends on implications such as

$$1 \geq 2 \ \Rightarrow \ \mathit{diff}(1, 2) = 1 - 2$$

in which $\mathit{diff}(1, 2)$ does not denote an integer. If the equality (=) is strict (which is the case with normal computational – or weak – equality) the right-hand side of this implication does not denote a truth value. (In fact, the *diff* example is purposely chosen because there is not a convenient subtype to use for the domain over which its application

is defined.) There are several ways of handling the difficulty with such a formula. One possibility is to read logical connectives like implication as though they were defined by conditional expressions which are non-strict in their second argument. Unfortunately, with this viewpoint, one loses intuitive properties such as commutativity for disjunctions and conjunctions; it also fails to help with examples such as

$$\forall i, j : \mathbb{Z} \cdot \mathit{diff}(i, j) = i - j \vee \mathit{diff}(j, i) = j - i \ .$$

A range of approaches to this problem are reviewed in [CJ91] and [MR91]. The former presents arguments for the logic which is used with VDM (see [Jon90]). This logic is known as the 'Logic of Partial Functions' (LPF) and uses non-classical meanings for the logical connectives and quantifiers. Atomic formulae that contain non-denoting terms may be logically neither-true-nor-false and the logical connectives and quantifiers are extended to cope with operands that are neither-true-nor-false; the only apparent disadvantage is that one has to give up the 'law of the excluded middle'. Yet, the classical truth-conditions and falsehood-conditions for logical connectives and quantifiers are retained: LPF provides extensions to the connectives and quantifiers in which the formula concerned is classified as neither-true-nor-false exactly when it cannot be classified as true or false by these conditions. An untyped version of LPF is presented in [BCJ84] and elaborated in [Che86].

Another approach to the difficulty discussed above stays within the world of classical two-valued logics by viewing atomic formulae that contain non-denoting terms as logically false. In this way, the 'law of the excluded middle' does not have to be abandoned. When a formula cannot be classified as true, it is inexorably classified as false; no further distinction is made. This approach is attributed to Scott [Sco67] and has been followed in, for example, MPL_ω [KR89].

The approach followed in LPF can be explained at the same time as showing the thrust of the description of LPF set out below. Consider the formula

$$\mathit{diff}(1, 2) = 1 \vee \neg\, (\mathit{diff}(1, 2) = 1) \ .$$

This is *not* a tautology in LPF. It can be translated into classical logic as follows:

$$\mathit{diff}(1, 2) \not= \uparrow \wedge\ 1 \not= \uparrow \wedge\ \mathit{diff}(1, 2) = 1 \vee\ \mathit{diff}(1, 2) \not= \uparrow \wedge\ 1 \not= \uparrow \wedge\ \mathit{diff}(1, 2) \not= 1 \ ,$$

where $\uparrow$ is a constant corresponding to undefined and $=$ is classical equality which yields true when its operands are the same – even if undefined – and false otherwise. Essentially, the equality used ($=$) is being made to absorb the undefinedness.

Since it has been described elsewhere, the case for LPF is not addressed further here: the purpose of this paper is to give a firm foundation to a typed version of LPF. One method employed is that indicated above: all formulae are mapped into classical logic. The version of LPF treated in this paper is used as the basis of formal specification and verified design in the software development method VDM. In order to be usable in software development, it has to be extended to deal with the base types and type formers used in VDM, subtypes via type invariants, recursively defined types and functions, etc. This gives essentially the VDM notation (VDM-SL) and its associated rules of reasoning.

In addition to the usual non-logical – model-theoretic – justification of the inference rules of LPF, a logical justification is given in this paper by means of an embedding into classical logic. This shows how this non-classical logic can be reconstructed classically. Classical logic is used meta-logically here: it provides a classical explanation of LPF

which is illuminating for those people who use this logic but have a stronger intuition about classical logic.

Following the presentation of LPF, the above-mentioned extensions are described. The rules given for reasoning about (some of) the base types and type formers, subtypes and recursively defined types as well as the rules given for reasoning about recursively defined functions are justified by means of an embedding of the extended LPF into classical infinitary logic [Kei71]. Classical logic with countably infinite conjunctions and disjunctions (L_ω) is used here to deal with recursion in type and function definitions. It would have been possible to use classical finitary logic extended with a minimal fixpoint operator but this alternative was rejected because it is further from being our *ultima ratio*.

The extended LPF provides essentially the VDM notation and its associated rules of reasoning. Like other specification languages, the VDM notation is meant to permit formulating claims concerning specifications for software systems – such as VDM proof obligations – in a mathematically precise way and constructing formal proofs to justify these claims. These central issues are shared with logic, but they are focused on software systems instead of abstract structures. Because these issues have been extensively studied in logic, an embedding into (classical) logic appears to be very useful. Besides, it makes formal justification of proof rules possible. A similar embedding of VVSL (which is a variant of the VDM notation) into MPL_ω – a weak extension of L_ω – can be found in [Mid93].

2. A basic logic of partial functions

A language of LPF is constructed with type symbols, function symbols and predicate symbols that belong to a certain set which is called a signature. For a given signature, say Σ, the language concerned is called the language of LPF over signature Σ or the language of LPF(Σ). The corresponding proof system and interpretation are analogously called the proof system of LPF(Σ) and the interpretation of LPF(Σ), respectively.

In this section LPF is described precisely. First, the assumptions which are made about type, function and predicate symbols are given and the notion of signature is introduced. Thereafter, the language, proof system and interpretation of LPF are defined.

2.1. Signatures for LPF

We assume a set *TYPE* of *type symbols*, a set *FUNC* of *function symbols*, and a set *PRED* of *predicate symbols*. Every $f \in$ *FUNC* and every $P \in$ *PRED* has an *arity* n ($n \geq 0$). To denote this arity, we use the notation *arity*(f) and *arity*(P). Function symbols of arity 0 are called *constant symbols*. There is a special predicate symbol = of arity 2, called *weak equality*.

A *signature* Σ is a finite subset of *TYPE* $\cup$ *FUNC* $\cup$ *PRED*. We write T(Σ) for $\Sigma \cap$ *TYPE*, F(Σ) for $\Sigma \cap$ *FUNC*, P(Σ) for $\Sigma \cap$ *PRED*. *SIG* denotes the set of all signatures for LPF.

We also assume a set *VAR* of *variable symbols*. Furthermore, it is assumed that *TYPE*, *FUNC*, *PRED*, *VAR* and $\{=\}$ are mutually disjoint sets. We write $\mathcal{V}_{\mathrm{LPF}}$ for *TYPE* $\cup$ *FUNC* $\cup$ *PRED* $\cup$ *VAR*. We use the notation $w \equiv w'$ ($w, w' \in \mathcal{V}_{\mathrm{LPF}}$) to indicate that w and w' are identical symbols.

2.2. Language of LPF(Σ)

Terms and formulae

The language of LPF(Σ) contains terms and formulae. They are constructed according to the formation rules given below.

The logical connectives and quantifiers of classical logic have counterparts in LPF. In addition, LPF has the logical connectives $*$ and $\triangle$. These additional connectives are not needed for specifying software systems but they make LPF an expressively complete three-valued logic (i.e., any function on the three-valued domain of truth values can be defined by a formula). The proof rules for the connectives $*$ and $\triangle$ are seldom needed for reasoning about specifications; indeed, this is precisely one of the advantages claimed for LPF. The reader is referred to [CJ91] for further discussion. False (false), definedness ($\downarrow$) and strong equality ($==$), which are defined below by means of $*$ and $\triangle$, are also seldom employed in proofs using LPF; of course, they play a larger role in the current paper which concerns the foundations of the whole of LPF.

The terms of LPF(Σ) are inductively defined by the following formation rules:

1. variable symbols are terms;
2. if $f \in$ F(Σ), ***arity***(f) = n and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.

The formulae of LPF(Σ) are inductively defined by the following formation rules:

1. $*$ is a formula;
2. if $P \in$ P(Σ), ***arity***(P) = n and $t_1, \ldots, t_n$ are terms, then $P(t_1, \ldots, t_n)$ is a formula;
3. if t_1 and t_2 are terms, then $t_1 = t_2$ is a formula;
4. if t is a term and $T \in$ T(Σ), then $t : T$ is a formula;
5. if A is a formula, then $\triangle A$ and $\neg A$ are formulae;
6. if A_1 and A_2 are formulae, then $A_1 \wedge A_2$ is a formula;
7. if A is a formula, x is a variable symbol and $T \in$ T(Σ), then $\forall x : T \cdot A$ is a formula.

The string representation of formulae suggested by these formation rules can lead to syntactic ambiguities: parentheses are used to avoid such ambiguities.

$\mathcal{T}_{\mathrm{LPF}}(\Sigma)$ and $\mathcal{L}_{\mathrm{LPF}}(\Sigma)$ denote the set of all terms of LPF(Σ) and the set of all formulae of LPF(Σ), respectively.

We henceforth use (with or without subscripts):

T and T' to stand for arbitrary type symbols in T(Σ),
c to stand for an arbitrary constant symbol in F(Σ),
f and g to stand for arbitrary function symbols in F(Σ),
P and Q to stand for arbitrary predicate symbols in P(Σ),
x, y and z to stand for arbitrary variable symbols in ***VAR***,
t and t' to stand for arbitrary terms in $\mathcal{T}_{\mathrm{LPF}}(\Sigma)$,
A, A' and A'' to stand for arbitrary formulae in $\mathcal{L}_{\mathrm{LPF}}(\Sigma)$.

The formula $*$ is neither-true-nor-false. $\triangle A$ is true if A is either true or false and $\triangle A$ is false otherwise. So $\triangle *$ is false. For the connectives $\neg$ and $\wedge$ as well as the quantifier $\forall$, the classical truth-conditions and falsehood-conditions are retained. A formula is classified as neither-true-nor-false exactly when it cannot be classified as true or false by these conditions. Equality is treated in the same way: $t_1 = t_2$ is neither-true-nor-false if and only if t_1 or t_2 is non-denoting.

The formula $t : T$ is a typing assertion. If $t : T$ is true then t must be denoting, which means that $t = t$ is true as well. If t is non-denoting, then $t : T$ is neither-true-nor-false.

Abbreviations and notational conventions

Additional connectives and quantifiers are defined as abbreviations:

$$\begin{array}{rcl} \mathsf{false} & := & \triangle * \ , \\ \delta\, A & := & A \vee \neg\ A \ , \\ A_1 \vee A_2 & := & \neg\,(\neg\ A_1 \wedge \neg\ A_2) \ , \\ A_1 \ \Rightarrow\ A_2 & := & \neg\ A_1 \vee A_2 \ , \\ A_1 \ \Leftrightarrow\ A_2 & := & (A_1 \ \Rightarrow\ A_2) \wedge (A_2 \ \Rightarrow\ A_1) \ , \\ \exists x : T \cdot A & := & \neg\ \forall x : T \cdot \neg\ A \ . \end{array}$$

Definedness ($\downarrow$) and strong equality (==) are used in Sect. 3. They are defined by the following abbreviations:

$$\begin{array}{rcl} t\downarrow & := & \triangle(t = t) \ , \\ t_1 == t_2 & := & (t_1\downarrow \vee\ t_2\downarrow) \ \Rightarrow\ (t_1 = t_2 \wedge \triangle(t_1 = t_1 \wedge t_2 = t_2)) \ . \end{array}$$

So $t\downarrow$ is true if t is denoting and $t\downarrow$ is false otherwise. Strong equality is very much like equality in classical logic: $t_1 == t_2$ is true if t_1 and t_2 denote the same object or both are non-denoting and $t_1 == t_2$ is false otherwise.

For convenience, non-equality is also defined as abbreviation:

$$t_1 \neq t_2 \quad := \quad \neg\,(t_1 = t_2) \ .$$

The need to use parentheses in the string representation of formulae is reduced by ranking the precedence of the logical connectives $\triangle$, δ, $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$. The enumeration presents this order from the highest precedence to the lowest precedence. Furthermore the scope of the quantifiers extends as far as possible to the right and $\forall x_1 : T_1 \cdot \ldots \forall x_n : T_n \cdot A$ is usually written as $\forall x_1 : T_1, \ldots, x_n : T_n \cdot A$. Parentheses are usually omitted in terms of the form $f(t_1, \ldots, t_n)$ whenever *arity*(f) = 0: constant symbols are used as terms.

Free variables and substitution

For a term or formula e of LPF(Σ), *free*(e) denotes the set of *free variables* of e, which is defined in the usual way. A variable symbol x is called *free in* e if $x \in$ *free*(e). We write *free*(Γ), where Γ is a set of formulae, for $\bigcup\{\mathit{free}(A) \mid A \in \Gamma\}$.

Substitution for variables is also defined in the usual way. Let x be a variable symbol, t be a term and e be a term or formula. Then $[x := t]e$ is the result of replacing the term t for the free occurrences of the variable symbol x in e, avoiding – by means of renaming of bound variables – free variables becoming bound in t.

2.3. Proof system of LPF(Σ)

Sequents

The proof system of LPF(Σ) is formulated as a sequent calculus for proofs in natural deduction style.[1] The inference rules have formulae and sequents amongst their hypotheses (called ordinary hypotheses and sequent hypotheses, respectively).

[1] For a comparison of this and other proof styles as well as other kinds of proof systems, see e.g., [Sun83].

A *sequent* is an expression of the form $\Gamma \vdash A$, where Γ and A are a finite set of formulae and a formula, respectively, of LPF(Σ). Instead of $\{\,\} \vdash A$ we write $\vdash A$. Furthermore, we write Γ, Γ' for $\Gamma \cup \Gamma'$ and A for $\{A\}$.

The intended meaning of the sequent $\Gamma \vdash A$ is that the formula A is a consequence of the formulae Γ. There are several sensible notions of consequence for three-valued logics; that underlying LPF is precisely defined in Sect. 2.4. It corresponds to the intuitive idea that one can draw conclusions that are true from premises that are true (called strong conclusions and strong premises, respectively, in [KTB88]). Formulae and sequents are proved by (natural deduction) proofs obtained by using the rules of inference given below.

Rules of inference

The essential point about LPF is that the law of the excluded middle ($A \vee \neg A$) does not hold (A might be neither-true-nor-false). Since this is implied by

$$\boxed{\neg\text{-I}}\ \frac{A_1 \vdash A_2 \quad A_1 \vdash \neg A_2}{\neg A_1}$$

and the rule ($\wedge$-E) given below (which can be used in LPF as well as in classical logic), it follows that the rule ($\neg$-I) – or any other rule corresponding to the principle of proof by contradiction – cannot be used. In consequence, rules concerning the negation of negations, conjunctions and universal quantifications are needed in the proof system of LPF. (Other distinguishing points are discussed after the rules.)

The proof system of LPF is defined by the following rules of inference:

$$\boxed{\neg\text{-E}}\ \frac{A_1 \quad \neg A_1}{A_2}$$

$$\boxed{\neg\neg\text{-I}}\ \frac{A}{\neg\neg A} \qquad \boxed{\neg\neg\text{-E}}\ \frac{\neg\neg A}{A}$$

$$\boxed{\wedge\text{-I}}\ \frac{A_1 \quad A_2}{A_1 \wedge A_2} \qquad \boxed{\wedge\text{-E}}\ \frac{A_1 \wedge A_2}{A_i}\ \text{for } i = 1,2$$

$$\boxed{\neg\wedge\text{-I}}\ \frac{\neg A_i}{\neg(A_1 \wedge A_2)}\ \text{for } i = 1,2 \qquad \boxed{\neg\wedge\text{-E}}\ \frac{\neg(A_1 \wedge A_2) \quad \neg A_1 \vdash A_3 \quad \neg A_2 \vdash A_3}{A_3}$$

$$\boxed{\forall\text{-I}}\ \frac{x : T \vdash A}{\forall x : T \cdot A} \qquad \boxed{\forall\text{-E}}\ \frac{t : T \quad \forall x : T \cdot A}{[x := t]A}$$

$$\boxed{\neg\forall\text{-I}}\ \frac{t : T \quad \neg [x := t]A}{\neg \forall x : T \cdot A} \qquad \boxed{\neg\forall\text{-E}}\ \frac{\neg \forall x : T \cdot A_1 \quad x : T, \neg A_1 \vdash A_2}{A_2}\ \ddagger$$

$$\boxed{=\text{-refl}}\ \frac{t : T}{t = t} \qquad \boxed{=\text{-sub}}\ \frac{t_1 = t_2 \quad [x := t_1]A}{[x := t_2]A}$$

$$\boxed{\delta=\text{-I}}\ \frac{t_1 = t_1 \quad t_2 = t_2}{\delta(t_1 = t_2)} \qquad \boxed{\delta=\text{-E}}\ \frac{\delta(t_1 = t_2)}{t_1 = t_1 \wedge t_2 = t_2}$$

$$\boxed{\delta:\text{-I}}\ \frac{t = t}{\delta(t : T)} \qquad \boxed{\delta:\text{-E}}\ \frac{\delta(t : T)}{t = t}$$

$$\frac{}{\delta(x:T)}\ x\text{-den} \qquad \frac{*}{A}\ *\text{-E} \qquad \frac{\neg\ *}{A}\ \neg *\text{-E}$$

$$\frac{A}{\triangle A}\ \triangle\text{-I-1} \qquad \frac{\neg\ A}{\triangle A}\ \triangle\text{-I-2} \qquad \frac{\triangle A_1 \quad A_1 \vdash A_2 \quad \neg\ A_1 \vdash A_2}{A_2}\ \triangle\text{-E}$$

$$\frac{\triangle A_1 \vdash A_2 \quad \triangle A_1 \vdash \neg\ A_2}{\neg\ \triangle A_1}\ \neg\triangle\text{-I} \qquad \frac{\neg\ \triangle A_1 \vdash A_2 \quad \neg\ \triangle A_1 \vdash \neg\ A_2}{\triangle A_1}\ \neg\triangle\text{-E}$$

‡ Restriction on the rule ($\neg\ \forall$-E): x not free in A_2.

The rule of reflexivity for equality is slightly adapted from the classical case because it does not satisfy the usual law in case of non-denoting terms. The additional rules for equality are also needed because of the extension to the three-valued case – $t_1 = t_2$ is true or false exactly when t_1 and t_2 are denoting.

Similar rules are needed for typing assertions – $t : T$ is true or false exactly when t is denoting.[2] The other rule concerning typing is needed because variables are always denoting in LPF.

The rules for $*$ and $\triangle$ are seldom used in practice. However, exactly these rules are used to justify the derived rules of inference (false-E) and ($\neg$ false-I) given below. Further we have the following law of the excluded fourth in LPF: $A \vee \neg\ A \vee \neg\ \triangle A$.

Proofs

A natural deduction *proof* consists of:

1. a finite set of formulae, called the *hypotheses* of the proof;
2. a non-empty finite sequence of formulae and proofs, called the *steps* of the proof, the last of which must be a formula which is called the *conclusion* of the proof.

Each step that is a formula must be a hypothesis of the proof or the conclusion of an instance of an inference rule. In the latter case, each of the ordinary hypotheses of the rule instance concerned must be a hypothesis or preceding step of the proof (or of an enclosing proof) and each of the sequent hypotheses of the rule instance concerned must be established by a preceding step of the proof (or of an enclosing proof). A sequent $\Gamma \vdash A$ is established by a step iff the step is a (sub-)proof, every hypothesis of the proof is in Γ and the conclusion of the proof is A.

A sequent $\Gamma \vdash A$ is *provable* if there exists a proof with Γ as hypotheses and A as conclusion. A formula A is provable if the sequent $\vdash A$ is provable. To indicate this, we write LPF(Σ) : $\Gamma \vdash A$ and LPF(Σ) : A, respectively.

Derived rules

The following are some *derived* rules, i.e. for each instance of these rules, if the hypotheses are provable then so is the conclusion:

$$\frac{}{\neg\ \text{false}}\ \neg\ \text{false-I} \qquad \frac{\text{false}}{A}\ \text{false-E} \qquad \frac{t \neq t}{A}\ \neq\text{-E}$$

[2] These rules make the rule of reflexivity for equality superfluous.

$$\boxed{=\text{-sym}}\ \frac{t_1 = t_2}{t_2 = t_1} \qquad \boxed{\neq\text{-sym}}\ \frac{t_1 \neq t_2}{t_2 \neq t_1}$$

$$\boxed{\delta\text{=-I}'}\ \frac{t_1 : T_1 \quad t_2 : T_2}{\delta(t_1 = t_2)} \qquad \boxed{\delta\,\forall\text{-I}}\ \frac{x : T \vdash \delta\, A}{\delta(\forall x : T \cdot A)}$$

The following derived rules show how weak equality and strong equality are related:

$$\boxed{=\!\!\rightarrow\!\!==}\ \frac{t_1 = t_2}{t_1 == t_2} \qquad \boxed{==\!\!\rightarrow\!\!=}\ \frac{t_1 == t_2 \quad t_1 = t_1}{t_1 = t_2}$$

The formulae and sequents of LPF are translated to formulae and sequents of classical infinitary logic (L_ω) in Sect. 2.5. The translation concerned has the property that what can be proved in LPF remains the same after translation. This implies that the inference rules of LPF become derived rules of L_ω after translation. The translation provides one justification for the inference rules of LPF; another justification is afforded by the interpretation given below.

2.4. Interpretation of LPF(Σ)

The proof system of LPF is based on the interpretation of terms and formulae presented below: the rules of inference preserve validity under this interpretation.

Structures

Terms and formulae of LPF(Σ) are interpreted in structures which consist of a universal domain of values and an interpretation of every symbol in the signature Σ as well as the equality symbol. The universal domain of values must be a set containing a special element $\bot$. When a term is non-denoting, $\bot$ is used as its interpretation. Analogously, when a formula is neither *true* (T) nor *false* (F), N is used as its interpretation.

A structure **A**, with signature Σ, consists of:

1. a set $\mathcal{U}^{\mathbf{A}}$, the *domain* of **A**, such that $\bot \in \mathcal{U}^{\mathbf{A}}$ and $\mathcal{U}^{\mathbf{A}} - \{\bot\} \neq \{\,\}$;
2. for every $T \in \mathrm{T}(\Sigma)$,
   a set $T^{\mathbf{A}}$ such that $T^{\mathbf{A}} \subseteq \mathcal{U}^{\mathbf{A}} - \{\bot\}$;
3. for every $f \in \mathrm{F}(\Sigma)$, *arity*(f) = n,
   a total map $f^{\mathbf{A}} : \underbrace{\mathcal{U}^{\mathbf{A}} \times \cdots \times \mathcal{U}^{\mathbf{A}}}_{n \text{ times}} \to \mathcal{U}^{\mathbf{A}}$;
4. for every $P \in \mathrm{P}(\Sigma)$, *arity*(P) = n,
   a total map $P^{\mathbf{A}} : \underbrace{\mathcal{U}^{\mathbf{A}} \times \cdots \times \mathcal{U}^{\mathbf{A}}}_{n \text{ times}} \to \{\mathsf{T}, \mathsf{F}, \mathsf{N}\}$;
5. a total map $=^{\mathbf{A}} : \mathcal{U}^{\mathbf{A}} \times \mathcal{U}^{\mathbf{A}} \to \{\mathsf{T}, \mathsf{F}, \mathsf{N}\}$ such that *for all* $d, d' \in \mathcal{U}^{\mathbf{A}}$,

$$=^{\mathbf{A}}(d, d') = \begin{cases} \mathsf{T} & \text{if } d \neq \bot \text{ and } d' \neq \bot \text{ and } d = d' \\ \mathsf{F} & \text{if } d \neq \bot \text{ and } d' \neq \bot \text{ and } d \neq d' \\ \mathsf{N} & \text{otherwise.} \end{cases}$$

Instead of $w^{\mathbf{A}}$ we write w when it is clear from the context that the interpretation of symbol w in structure **A** is meant.

Assignments

An assignment in a structure $\mathbf{A}$ with signature Σ assigns to variables elements in the domain of $\mathbf{A}$. However, variables are never mapped to $\perp$. This restriction is in accordance with the treatment of variables: both free and bound variables always denote. The interpretation of terms and formulae of LPF(Σ) in $\mathbf{A}$ is given with respect to an assignment α in $\mathbf{A}$.

Let $\mathbf{A}$ be a structure with signature Σ. Then an *assignment* in $\mathbf{A}$ is a function $\alpha : \mathit{VAR} \to \mathcal{U}^{\mathbf{A}} - \{\perp\}$.

For every assignment α in $\mathbf{A}$, variable symbol x and element $d \in \mathcal{U}^{\mathbf{A}} - \{\perp\}$, we write $\alpha(x \to d)$ for the assignment α' such that $\alpha'(y) = \alpha(y)$ if $y \not\equiv x$ and $\alpha'(x) = d$.

Interpretation

The interpretation of terms is given by a function mapping term t, structure $\mathbf{A}$ and assignment α in $\mathbf{A}$ to the element of $\mathcal{U}^{\mathbf{A}}$ that is the value of t in $\mathbf{A}$ under assignment α. Similarly, the interpretation of formulae is given by a function mapping formula A, structure $\mathbf{A}$ and assignment α in $\mathbf{A}$ to the element of $\{\mathsf{T}, \mathsf{F}, \mathsf{N}\}$ that is the truth value of A in $\mathbf{A}$ under assignment α. We write $[\![t]\!]^{\mathbf{A}}_{\alpha}$ and $[\![A]\!]^{\mathbf{A}}_{\alpha}$ for these interpretations. The superscripts are omitted when it is clear from the context which structure is meant.

The interpretation functions for terms and formulae are inductively defined by

$$\begin{aligned} [\![x]\!]^{\mathbf{A}}_{\alpha} &= \alpha(x)\,, \\ [\![f(t_1,\ldots,t_n)]\!]^{\mathbf{A}}_{\alpha} &= f^{\mathbf{A}}([\![t_1]\!]^{\mathbf{A}}_{\alpha},\ldots,[\![t_n]\!]^{\mathbf{A}}_{\alpha}) \end{aligned}$$

and

$$\begin{aligned} [\![*]\!]^{\mathbf{A}}_{\alpha}, &= \mathsf{N}\,, \\ [\![P(t_1,\ldots,t_n)]\!]^{\mathbf{A}}_{\alpha} &= P^{\mathbf{A}}([\![t_1]\!]^{\mathbf{A}}_{\alpha},\ldots,[\![t_n]\!]^{\mathbf{A}}_{\alpha})\,, \\ [\![t_1 = t_2]\!]^{\mathbf{A}}_{\alpha} &= =^{\mathbf{A}}([\![t_1]\!]^{\mathbf{A}}_{\alpha},[\![t_2]\!]^{\mathbf{A}}_{\alpha})\,, \\ [\![t : T]\!]^{\mathbf{A}}_{\alpha} &= \begin{cases} \mathsf{T} & \text{if } [\![t]\!]^{\mathbf{A}}_{\alpha} \neq \perp \textit{ and } [\![t]\!]^{\mathbf{A}}_{\alpha} \in T^{\mathbf{A}} \\ \mathsf{F} & \text{if } [\![t]\!]^{\mathbf{A}}_{\alpha} \neq \perp \textit{ and } [\![t]\!]^{\mathbf{A}}_{\alpha} \notin T^{\mathbf{A}} \\ \mathsf{N} & \text{otherwise,} \end{cases} \\ [\![\triangle\, A]\!]^{\mathbf{A}}_{\alpha} &= \begin{cases} \mathsf{T} & \text{if } [\![A]\!]^{\mathbf{A}}_{\alpha} = \mathsf{T} \textit{ or } [\![A]\!]^{\mathbf{A}}_{\alpha} = \mathsf{F} \\ \mathsf{F} & \text{otherwise,} \end{cases} \\ [\![\neg\, A]\!]^{\mathbf{A}}_{\alpha} &= \begin{cases} \mathsf{T} & \text{if } [\![A]\!]^{\mathbf{A}}_{\alpha} = \mathsf{F} \\ \mathsf{F} & \text{if } [\![A]\!]^{\mathbf{A}}_{\alpha} = \mathsf{T} \\ \mathsf{N} & \text{otherwise,} \end{cases} \\ [\![A_1 \wedge A_2]\!]^{\mathbf{A}}_{\alpha} &= \begin{cases} \mathsf{T} & \text{if } [\![A_1]\!]^{\mathbf{A}}_{\alpha} = \mathsf{T} \textit{ and } [\![A_2]\!]^{\mathbf{A}}_{\alpha} = \mathsf{T} \\ \mathsf{F} & \text{if } [\![A_1]\!]^{\mathbf{A}}_{\alpha} = \mathsf{F} \textit{ or } [\![A_2]\!]^{\mathbf{A}}_{\alpha} = \mathsf{F} \\ \mathsf{N} & \text{otherwise,} \end{cases} \\ [\![\forall x : T \cdot A]\!]^{\mathbf{A}}_{\alpha} &= \begin{cases} \mathsf{T} & \text{if } \textit{for all } d \in T^{\mathbf{A}},\ [\![A]\!]^{\mathbf{A}}_{\alpha(x\to d)} = \mathsf{T} \\ \mathsf{F} & \text{if } \textit{for some } d \in T^{\mathbf{A}},\ [\![A]\!]^{\mathbf{A}}_{\alpha(x\to d)} = \mathsf{F} \\ \mathsf{N} & \text{otherwise.} \end{cases} \end{aligned}$$

We write $\mathbf{A} \models A[\alpha]$ for $[\![A]\!]^{\mathbf{A}}_{\alpha} = \mathsf{T}$.

Notice that the above interpretation makes conjunction non-strict in both of its arguments and gives the truth value F for a universally quantified formula $\forall x : T \cdot A$ in some cases where the interpretation of A is neither T nor F for some assignments.

For a finite set Γ of formulae of LPF(Σ) and a formula A of LPF(Σ), A is a *consequence* of Γ, written $\Gamma \models A$, iff for all structures **A** with signature Σ, for all assignments α in **A**, if $\mathbf{A} \models A'[\alpha]$ for all $A' \in \Gamma$ then $\mathbf{A} \models A[\alpha]$.

Theorem 1. *The proof system given above for LPF has the following soundness and completeness properties:*

$$\begin{array}{ll} \text{soundness :} & \text{if } \Gamma \vdash A, \text{ then } \Gamma \models A \ ; \\ \text{completeness :} & \text{if } \Gamma \models A, \text{ then } \Gamma \vdash A \ . \end{array}$$

Proof. The proof for the untyped case in [Che86] extends directly to the typed case.

It is a consequence of the compositional style adopted for constructing a completeness proof that – in case of incompleteness – the failed proof attempt indicates the origin(s) of the incompleteness. In fact, the rules (δ :-I), (δ :-E), and (x-den), which are needed for typing assertions, were only discovered when we tried to construct the completeness proof.

2.5. Embedding LPF into L_ω

In this subsection, the relationship between LPF and classical infinitary logic is characterized. The terms, formulae and sequents of LPF are translated to terms, formulae and sequents, respectively, of L_ω. The mappings concerned provide a uniform embedding of LPF into L_ω. The translation has the property that what can be proved in LPF remains the same after translation. It provides an illuminating classical explanation of LPF and justifies the inference rules of LPF logically. Later, extensions of LPF concerning the base types and type formers used in VDM, subtypes via invariants and recursively defined types and functions are presented. The inference rules concerned are also justified by an embedding into L_ω.

Translation

In the translation, a canonical mapping from symbols of LPF to symbols of L_ω is assumed. More precisely, we assume a total mapping from $\mathcal{V}_{\mathrm{LPF}}$ to $\mathcal{V}_{\mathrm{L}_\omega}$; for each $w \in \mathcal{V}_{\mathrm{LPF}}$, we write $\boldsymbol{w}$ for the symbol to which w is mapped. Furthermore, the mapping is assumed to be injective and such that

each type symbol T is mapped to a predicate symbol $\boldsymbol{T}$
with *arity*($\boldsymbol{T}$) = 1,
each function symbol f is mapped to a function symbol $\boldsymbol{f}$
with *arity*($\boldsymbol{f}$) = *arity*(f),
each predicate symbol P is mapped to a function symbol $\boldsymbol{P}$
with *arity*($\boldsymbol{P}$) = *arity*(P),
each variable symbol x is mapped to a variable symbol $\boldsymbol{x}$.

We also use the notation $\boldsymbol{W}$ for the image of W ($W \subseteq \mathcal{V}_{\mathrm{LPF}}$) under this mapping. We write:

$$\begin{array}{ll} \mathcal{T}_{\mathrm{LPF}} & \text{for } \bigcup\{\mathcal{T}_{\mathrm{LPF}}(\Sigma) \mid \Sigma \in \mathit{SIG}\}, \\ \mathcal{L}_{\mathrm{LPF}} & \text{for } \bigcup\{\mathcal{L}_{\mathrm{LPF}}(\Sigma) \mid \Sigma \in \mathit{SIG}\}, \\ \mathcal{T}_{\mathrm{L}_\omega} & \text{for } \bigcup\{\mathcal{T}_{\mathrm{L}_\omega}(\Sigma) \mid \Sigma \in \boldsymbol{SIG}\}, \\ \mathcal{L}_{\mathrm{L}_\omega} & \text{for } \bigcup\{\mathcal{L}_{\mathrm{L}_\omega}(\Sigma) \mid \Sigma \in \boldsymbol{SIG}\}. \end{array}$$

The terms and formulae of LPF are translated by mappings:

$$(\!|\bullet|\!) : \mathcal{T}_{\mathrm{LPF}} \to \mathcal{T}_{\mathrm{L}_\omega}, \qquad (\!|\bullet|\!)^{\mathsf{t}} : \mathcal{L}_{\mathrm{LPF}} \to \mathcal{L}_{\mathrm{L}_\omega}.$$

For the translation of formulae, an auxiliary mapping is used as well:

$$(\!|\bullet|\!)^{\mathsf{f}} : \mathcal{L}_{\mathrm{LPF}} \to \mathcal{L}_{\mathrm{L}_\omega}.$$

For a term t of LPF, the term $(\!|t|\!)$ is the translation of t to L_ω. For a formula A of LPF, the formula $(\!|A|\!)^{\mathsf{t}}$ is the translation of A to L_ω. Intuitively, $(\!|A|\!)^{\mathsf{t}}$ is a formula of L_ω stating that the formula A of LPF is true in LPF. Likewise, $(\!|A|\!)^{\mathsf{f}}$ is a formula of L_ω stating that the formula A of LPF is false in LPF. In case both $(\!|A|\!)^{\mathsf{t}}$ and $(\!|A|\!)^{\mathsf{f}}$ are false in L_ω, A is neither-true-nor-false in LPF.

The syntactic variables that are used in the definition of these mappings, range over syntactic objects as follows (subscripts and primes are not shown):

$$\begin{array}{llll} T & \text{ranges over } \mathit{TYPE}, & x & \text{ranges over } \mathit{VAR}, \\ f & \text{ranges over } \mathit{FUNC}, & t & \text{ranges over } \mathcal{T}_{\mathrm{LPF}}, \\ P & \text{ranges over } \mathit{PRED}, & A & \text{ranges over } \mathcal{L}_{\mathrm{LPF}}. \end{array}$$

It is assumed that $\mathsf{t}, \mathsf{f}, \uparrow \in \boldsymbol{FUNC}$, $\mathsf{U}, \mathsf{B} \in \boldsymbol{PRED}$, $y, y_1, \ldots, y_n \in \boldsymbol{VAR}$, $\mathsf{t}, \mathsf{f}, \uparrow$ of arity 0 and U, B of arity 1.

The symbol $=\!\!=$ is used for equality in L_ω. This (classical) equality is explained in Appendix A.

The translation mapping for terms is inductively defined by

$$\begin{array}{rcl} (\!|x|\!) & = & \boldsymbol{x}, \\ (\!|f(t_1, \ldots, t_n)|\!) & = & \boldsymbol{f}((\!|t_1|\!), \ldots, (\!|t_n|\!)). \end{array}$$

The translation mapping for formulae and the auxiliary mapping are simultaneously and inductively defined by

$$\begin{array}{rcl} (\!|*|\!)^{\mathsf{t}} & = & \mathsf{false}, \\ (\!|P(t_1, \ldots, t_n)|\!)^{\mathsf{t}} & = & \boldsymbol{P}((\!|t_1|\!), \ldots, (\!|t_n|\!)) =\!\!= \mathsf{t}, \\ (\!|t_1 = t_2|\!)^{\mathsf{t}} & = & (\!|t_1|\!) \neq\!\!= \uparrow \wedge (\!|t_2|\!) \neq\!\!= \uparrow \wedge (\!|t_1|\!) =\!\!= (\!|t_2|\!), \\ (\!|t : T|\!)^{\mathsf{t}} & = & (\!|t|\!) \neq\!\!= \uparrow \wedge \boldsymbol{T}((\!|t|\!)), \\ (\!|\triangle A|\!)^{\mathsf{t}} & = & (\!|A|\!)^{\mathsf{t}} \vee (\!|A|\!)^{\mathsf{f}}, \\ (\!|\neg A|\!)^{\mathsf{t}} & = & (\!|A|\!)^{\mathsf{f}}, \\ (\!|A_1 \wedge A_2|\!)^{\mathsf{t}} & = & (\!|A_1|\!)^{\mathsf{t}} \wedge (\!|A_2|\!)^{\mathsf{t}}, \\ (\!|\forall x : T \cdot A|\!)^{\mathsf{t}} & = & \forall \boldsymbol{x} \cdot \boldsymbol{T}(\boldsymbol{x}) \Rightarrow (\!|A|\!)^{\mathsf{t}}, \end{array}$$

$$
\begin{aligned}
(\!|*|\!)^{\mathsf{f}} &= \mathsf{false}\ , \\
(\!|P(t_1,\ldots,t_n)|\!)^{\mathsf{f}} &= \boldsymbol{P}((\!|t_1|\!),\ldots,(\!|t_n|\!)) = \mathsf{f}\ , \\
(\!|t_1 = t_2|\!)^{\mathsf{f}} &= (\!|t_1|\!) \neq \uparrow \wedge (\!|t_2|\!) \neq \uparrow \wedge (\!|t_1|\!) \neq (\!|t_2|\!)\ , \\
(\!|t : T|\!)^{\mathsf{f}} &= (\!|t|\!) \neq \uparrow \wedge \neg\ \boldsymbol{T}((\!|t|\!))\ , \\
(\!|\triangle\ A|\!)^{\mathsf{f}} &= \neg\ ((\!|A|\!)^{\mathsf{t}} \vee (\!|A|\!)^{\mathsf{f}})\ , \\
(\!|\neg\ A|\!)^{\mathsf{f}} &= (\!|A|\!)^{\mathsf{t}}\ , \\
(\!|A_1 \wedge A_2|\!)^{\mathsf{f}} &= (\!|A_1|\!)^{\mathsf{f}} \vee (\!|A_2|\!)^{\mathsf{f}}\ , \\
(\!|\forall x : T \cdot A|\!)^{\mathsf{f}} &= \exists \boldsymbol{x} \cdot \boldsymbol{T}(\boldsymbol{x}) \ \wedge\ (\!|A|\!)^{\mathsf{f}}\ .
\end{aligned}
$$

These translation rules strongly resemble the interpretation rules of LPF that are given in Sect. 2.4: the rules for the mapping $(\!|\bullet|\!)^{\mathsf{t}}$ correspond to the truth-conditions and the rules for the mapping $(\!|\bullet|\!)^{\mathsf{f}}$ correspond to the falsehood-conditions.

A translation for sequents of LPF(Σ) can also be devised:

$$(\!|\Gamma \vdash A|\!) \quad := \quad \mathrm{Ax}(\Sigma, \Gamma \cup \{A\}) \cup \{(\!|A'|\!)^{\mathsf{t}} \mid A' \in \Gamma\} \vdash (\!|A|\!)^{\mathsf{t}}\ ,$$

where

$$
\begin{aligned}
&\mathrm{Ax}(\Sigma, \Gamma') = \\
&\quad \{\mathsf{U}(\uparrow) \wedge \exists y \cdot \mathsf{U}(y) \wedge y \neq \uparrow\} \cup \\
&\quad \{\mathsf{t} \neq \mathsf{f} \wedge \mathsf{t} \neq \uparrow \wedge \mathsf{f} \neq \uparrow \wedge (\mathsf{B}(b) \ \Leftrightarrow\ b = \mathsf{t} \vee b = \mathsf{f} \vee b = \uparrow)\} \cup \\
&\quad \{\boldsymbol{T}(y) \ \Rightarrow\ \mathsf{U}(y) \wedge y \neq \uparrow \mid T \in \mathrm{T}(\Sigma)\} \cup \\
&\quad \{\mathsf{U}(y_1) \wedge \ldots \wedge \mathsf{U}(y_n) \Rightarrow \mathsf{U}(\boldsymbol{f}(y_1,\ldots,y_n)) \mid f \in \mathrm{F}(\Sigma), \mathit{arity}(f) = n\} \cup \\
&\quad \{\mathsf{U}(y_1) \wedge \ldots \wedge \mathsf{U}(y_n) \Rightarrow \mathsf{B}(\boldsymbol{P}(y_1,\ldots,y_n)) \mid P \in \mathrm{P}(\Sigma), \mathit{arity}(P) = n\} \cup \\
&\quad \{\mathsf{U}(\boldsymbol{x}) \wedge \boldsymbol{x} \neq \uparrow \mid x \in \mathit{free}(\Gamma')\}\ .
\end{aligned}
$$

$\mathrm{Ax}(\Sigma, \Gamma')$ contains a formula asserting that the domain of values contains at least one value in addition to the special element used as the interpretation of non-denoting terms and a formula asserting that the domain of truth values contains exactly two distinct truth values in addition to the special element used as the interpretation of non-denoting formulae. It also contains formulae asserting that the types concerned do not contain the special element used as the interpretation of non-denoting terms. It further contains formulae asserting that application of the functions concerned yields values from the domain of values and formulae asserting that application of the predicates concerned yields truth values. Finally, it contains formulae asserting that the free variables are always denoting.

Note that the finite fragment of L_ω suffices for the embedding of LPF. L_ω is used because its countably infinite disjunctions are needed for the embedding of the extensions for recursive definitions of functions and types in Sects. 3 and 5.

Reducibility

Roughly speaking, LPF can be reduced to L_ω in the sense that what can be proved in LPF remains the same after translation.

Theorem 2. *LPF can be reduced to L_ω, i.e.*

$$\mathrm{LPF}(\Sigma) :\ \Gamma \vdash A \quad \text{iff} \quad \mathrm{L}_\omega(\boldsymbol{\Sigma} \cup \{\mathsf{U}, \mathsf{B}, \mathsf{t}, \mathsf{f}, \uparrow\}) :\ (\!|\Gamma \vdash A|\!)\ .$$

Proof. $\Rightarrow$ is proved by induction over the length of a proof of $\Gamma \vdash A$. For $\Leftarrow$, it suffices to show that for some structure **A** of LPF with signature Σ that is a counter-model for $\Gamma \vdash A$, there exists a structure $\mathbf{A}^*$ of L_ω with signature $\boldsymbol{\Sigma} \cup \{\mathsf{U}, \mathsf{B}, \mathsf{t}, \mathsf{f}, \uparrow\}$ that is a counter-model for $(\!|\Gamma \vdash A|\!)$.

It is assumed that the translation of sequents is extended to inference rules in the obvious way.

Corollary. *The translation of the inference rules of LPF are derived rules in* L_ω.

3. Recursively defined functions

In the previous section, LPF was embedded into L_ω. Recursive function definitions can be represented in L_ω. This permits the rules used for reasoning about recursively defined functions in LPF to become derived rules of L_ω.

In this section the extension of LPF for recursive function definitions is described. First, the additional formation rules, inference rules and interpretation rules for recursive function definitions are given. Thereafter, their embedding into L_ω is defined.

3.1. LPF and recursive function definitions

The logic LPF is used in VDM to reason about recursively defined functions. The treatment of recursive function definitions in VDM is made precise below by defining a conservative extension of LPF.

The following additional formation rule for terms is required:

3. if A is a formula and t_1 and t_2 are terms, then if A then t_1 else t_2 is a term.

Terms of this form are called *conditionals*. In [BCJ84], conditionals are also regarded as terms of an extension of LPF.

The following additional formation rule for formulae is required:

8. if $f \in \mathrm{F}(\Sigma)$, *arity*(f) = n, $x_1, \ldots, x_n$ are distinct variable symbols, $T_1, \ldots, T_n$ are (not necessarily distinct) types and t is a term with *free*(t) $\subseteq \{x_1, \ldots, x_n\}$, then $f(x_1 : T_1, \ldots, x_n : T_n)\, T \triangleq t$ is a formula.

Formulae of this form are called *recursive function definitions*. A recursive function definition $f(x_1 : T_1, \ldots, x_n : T_n)\, T \triangleq t$ defines f directly in terms of a defining term t in which the function being defined may be recursively used. It corresponds to the direct definition of f written in the VDM notation as

$$f : T_1 \times \cdots \times T_n \to T$$
$$f(x_1, \ldots, x_n) \triangleq t$$

The following are additional inference rules for conditionals and recursive function definitions:[3]

[3] The first hypothesis of the rule (Func-ind) could be replaced by the simpler $[f(x_1, \ldots, x_n) := *]A$ if $*$ was also regarded as a (non-denoting) term of the extension of LPF.

$$\text{if-1}\ \frac{A}{\text{if } A \text{ then } t_1 \text{ else } t_2 == t_1} \qquad \text{if-2}\ \frac{\neg\ A}{\text{if } A \text{ then } t_1 \text{ else } t_2 == t_2}$$

$$\text{if-3}\ \frac{\neg\ \triangle\ A}{\neg\ ((\text{if } A \text{ then } t_1 \text{ else } t_2)\downarrow)}$$

$$\text{Func-def}\ \frac{x_1 : T_1 \quad \ldots \quad x_n : T_n \quad t : T}{f(x_1 : T_1, \ldots, x_n : T_n)\, T \triangleq t \vdash f(x_1, \ldots, x_n) = t}$$

$$\text{Func-ind}\ \frac{\neg\,(u\downarrow)\ \Rightarrow\ [f(x_1, \ldots, x_n) := u]A \quad A \vdash [f(x_1, \ldots, x_n) := t]A}{f(x_1 : T_1, \ldots, x_n : T_n)\, T \triangleq t \vdash A}\ t \text{ continuous in } f,\ A \text{ admissible in } f$$

Here $[f(x_1, \ldots, x_n) := t]A$ is the result of simultaneously replacing the occurrences of the substitution instances of $f(x_1, \ldots, x_n)$ in A by the corresponding substitution instances of t. The function definition hypothesis is usually dropped when it is clear from the context which definition is meant.

We say that t is *continuous* in f iff the mapping from functions to functions that maps f to (the function that maps $x_1, \ldots, x_n$ to) t is continuous with respect to the 'less defined than' ordering given below. A sufficient syntactic condition for continuity is: in every term of the form if A' then t_1 else t_2 occurring in t, f does not occur in A'.

We say that A is *admissible* in f iff, for every chain of functions $F_0 \sqsubseteq F_1 \sqsubseteq F_2 \sqsubseteq \ldots$ (where $\sqsubseteq$ is the 'less defined than' ordering) contained in the set of all functions f satisfying A, its least upper bound is also in that set. The following syntactic properties characterize a large class of admissible formulae. Formulae of the forms $P(t_1, \ldots, t_n)$, $t_1 = t_2$, $t' : T'$, $\neg\,(t' : T')$, $t'\downarrow$ and $\neg\,(t'\downarrow)$ are admissible if in every term of the form if A' then t_1 else t_2 occurring in the formula concerned, f does not occur in A'; so are formulae of the forms $\neg\ P(t_1, \ldots, t_n)$ and $t_1 \neq t_2$ if additionally f occurs in at most one of the terms t_i (where $i \in \{1, \ldots, n\}$ and $i \in \{1, 2\}$, respectively). Also admissible are formulae in which f does not occur. Furthermore, if A', A_1 and A_2 are admissible formulae, then so are $A_1 \wedge A_2$, $A_1 \vee A_2$ and $\forall x : T' \cdot A'$. So is $\exists x : T' \cdot A'$ if additionally T' is a finite type. If A' is an admissible formula, then so are all formulae obtained by replacing one or more occurrences of a subformula A'' by $\neg\ \neg\ A''$ or vice versa. From these properties it follows among other things that a formula of the form $A_1 \Rightarrow A_2$ is admissible if $\neg\ A_1$ and A_2 are admissible.

Strong equality (==) is used instead of weak equality (=) in the rules (if-1) and (if-2) for the sake of conciseness and simplicity of the collection of primitive inference rules for conditionals. However, rules involving strong equality, which can only be defined in LPF by means of the uncommon connective $\triangle$, can mostly be dispensed with when reasoning about specifications. The following derived rules for conditionals are more often used in practice:

$$\text{if-1}'\ \frac{t_1 = t_1 \quad A}{\text{if } A \text{ then } t_1 \text{ else } t_2 = t_1} \qquad \text{if-2}'\ \frac{t_2 = t_2 \quad \neg\ A}{\text{if } A \text{ then } t_1 \text{ else } t_2 = t_2}$$

Moreover, the method of reasoning about recursive functions discussed below often circumvents the need to argue about conditionals directly.

In the structures used for interpretation, a partial function is modelled by a total map whose argument domains and result domain contain $\bot$. An argument tuple is mapped to $\bot$ if the function concerned is undefined for that argument tuple. This suggests the following definition, which is used in the additional interpretation rules given below.

For total maps $F, G : \underbrace{\mathcal{U}^{\mathbf{A}} \times \cdots \times \mathcal{U}^{\mathbf{A}}}_{n \text{ times}} \to \mathcal{U}^{\mathbf{A}}$, where $\mathbf{A}$ is a given structure, F is *less defined than* G iff

$$\textit{for all } d_1, \ldots, d_n \in \mathcal{U}^{\mathbf{A}}, F(d_1, \ldots, d_n) \neq \bot \Rightarrow F(d_1, \ldots, d_n) = G(d_1, \ldots, d_n) \ .$$

The following are the additional interpretation rules for conditionals and recursive function definitions:

$$[\![\textsf{if } A \textsf{ then } t_1 \textsf{ else } t_2]\!]^{\mathbf{A}}_{\alpha} \quad = \quad \begin{cases} [\![t_1]\!]^{\mathbf{A}}_{\alpha} & \text{if } [\![A]\!]^{\mathbf{A}}_{\alpha} = \mathsf{T} \\ [\![t_2]\!]^{\mathbf{A}}_{\alpha} & \text{if } [\![A]\!]^{\mathbf{A}}_{\alpha} = \mathsf{F} \\ \bot & \text{otherwise}, \end{cases}$$

$$[\![f(x_1 : T_1, \ldots, x_n : T_n)\, T \triangleq t]\!]^{\mathbf{A}}_{\alpha} =$$

$$\begin{cases} \mathsf{T} & \text{if } \ f^{\mathbf{A}} \text{ is the least defined } F : \underbrace{\mathcal{U}^{\mathbf{A}} \times \cdots \times \mathcal{U}^{\mathbf{A}}}_{n \text{ times}} \to \mathcal{U}^{\mathbf{A}} \text{ such that} \\ & \quad \textit{for all } d_1 \in T_1^{\mathbf{A}}, \ldots, d_n \in T_n^{\mathbf{A}}, d \in T^{\mathbf{A}}, \\ & \quad [\![t]\!]^{\mathbf{A}'}_{\alpha(x_1 \to d_1) \cdots (x_n \to d_n)} = d \Rightarrow F(d_1, \ldots, d_n) = d \\ \mathsf{F} & \text{otherwise}, \end{cases}$$

where $\mathbf{A}'$ is the structure with signature Σ such that $w^{\mathbf{A}'} = w^{\mathbf{A}}$ if $w \not\equiv f$ and $f^{\mathbf{A}'} = F$ ($w \in \Sigma$).

Note that the interpretation of $f(x_1 : T_1, \ldots, x_n : T_n)\, T \triangleq t$ is not a set of models in which f corresponds to the function being defined. Instead it is essentially the characteristic function of the set concerned. This interpretation is taken for technical reasons: function definition hypotheses and other hypotheses can thus be treated alike.

The soundness of the rules (if-1), (if-2), (if-3) and (Func-def) with respect to this interpretation is obvious. The hypotheses of the rule (Func-ind) imply that A holds for a countable sequence of approximations of the function f where each approximation is less defined than the next one: the first approximation is the totally undefined function and each of the following approximations relies on the previous approximation for the recursive uses of f in t. If t is continuous in f, then this sequence converges to the function being defined according to the interpretation of recursive function definitions given above. If additionally A is admissible in f, A holds for that function as well.

In [Jon90], it is informally explained how a recursive definition of a partial function can be rendered into inference rules. The inference rules concerned resemble the appropriate rules of an inductive definition of the function (for partial functions, such rules usually need to be of a particular form). Given the recursive definition, the inference rules can also be regarded as derived rules of this extension of LPF. For example,

$$\begin{array}{l} \mathit{fac} \ : \ \mathbb{Z} \to \mathbb{Z} \\ \mathit{fac}(n) \triangleq \textsf{if } n = 0 \textsf{ then } 1 \textsf{ else } n * \mathit{fac}(n-1) \end{array}$$

is a recursive definition of a function on integers which yields the factorial of non-negative integers and is undefined otherwise. The corresponding inference rules according to [Jon90] are

$$\boxed{\mathit{fac}\text{-b}}\ \frac{}{\mathit{fac}(0) = 1} \qquad \boxed{\mathit{fac}\text{-i}}\ \frac{t : \mathbb{Z} \quad t \neq 0 \quad \mathit{fac}(t-1) = t'}{\mathit{fac}(t) = t * t'}$$

They are derived rules of LPF with the extension for recursive function definitions. The

rules (*fac*-b) and (*fac*-i) are allowing any fixpoint of the definition instead of requiring the least fixpoint.[4] They do not suffice to show that *fac* is only defined for non-negative integers, i.e.

$$\forall x : \mathbb{Z} \cdot \mathit{fac}(x)\downarrow \;\Rightarrow\; x \geq 0 \,.$$

The justification of this leastness result depends among other things upon the rule (Func-ind). Note that the uncommon connective $\triangle$ has to be used – at least indirectly – to formulate leastness results. However, such results are not often needed when reasoning about specifications in practice.

3.2. Embedding recursive function definitions into L_ω

Just like formulae of LPF, recursive function definitions can be mapped to formulae of L_ω. The rules (if-1), (if-2), (if-3), (Func-def) and (Func-ind) become derived rules of L_ω after translation. So the translation justifies these additional rules as well. Consequently, it also justifies the generation of rules from recursive function definitions according to [Jon90].

Recursive definitions in L_ω

In L_ω, a large class of recursive definitions can be expressed as formulae.[5] To describe the formulae concerned, we use the following notation:

- *defined predicates* $\{x_1, \ldots, x_n \mid A\}$, with the meaning given by $\{x_1, \ldots, x_n \mid A\}(t_1, \ldots, t_n) \;\Leftrightarrow\; [x_1 := t_1, \ldots, x_n := t_n]A$;
- *predicate operators* $\Lambda P.\{x_1, \ldots, x_n \mid A\}$, with the meaning given by $(\Lambda P.\{x_1, \ldots, x_n \mid A\})(D) = \{x_1, \ldots, x_n \mid [P := D]A\}$;
- a *fixpoint operator* Fix: Fix(Φ) is the least fixpoint of Φ for *continuous* predicate operators $\Phi = \Lambda P.\{x_1, \ldots, x_n \mid A\}$ with *arity*(P) = n.

All this is precisely defined as abbreviations in Appendix A.

This will do to describe the formulae corresponding to recursive predicate definitions. In case of recursive function definitions, the definition concerned has first to be replaced by a recursive definition of a predicate that uniquely determines the function concerned. The replacement is also given by the mapping σ defined in Appendix A.

Embedding into L_ω

Conditionals require that terms are translated to formulae of L_ω by a mapping

$$(\!|\bullet|\!)^\bullet : \mathcal{T}_{\mathrm{LPF}'} \times \mathcal{T}_{\mathrm{L}_\omega} \to \mathcal{L}_{\mathrm{L}_\omega} \,,$$

where $\mathcal{T}_{\mathrm{LPF}'}$ denotes the set of all terms of LPF extended for recursive function definitions. Intuitively, $(\!|t|\!)^u$ is a formula stating that the value of t is u. The required

[4] In general, such inference rules are allowing *almost* any fixpoint. However this qualification applies only to very pathological cases.

[5] The recursive definitions concerned are exactly the definitions $f(x_1 : T_1, \ldots, x_n : T_n)\, T \mathrel{\underline{\triangle}} t$ for which t is continuous in f.

adaptations of the translation rules for the terms and formulae of LPF are trivial; e.g. the rule for the translation of function applications becomes:

$$(\!|f(t_1,\ldots,t_n)|\!)^u =$$
$$\exists y_1,\ldots,y_n \cdot \mathsf{U}(y_1) \wedge \ldots \wedge \mathsf{U}(y_n) \wedge (\!|t_1|\!)^{y_1} \wedge \ldots \wedge (\!|t_n|\!)^{y_n} \wedge \boldsymbol{f}(y_1,\ldots,y_n) \mathrel{\boldsymbol{=}} u \ .$$

The following rule is for the translation of conditionals to formulae of L_ω:

$$(\!|\mathsf{if}\ A\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2|\!)^u =$$
$$((\!|A|\!)^{\mathsf{t}} \wedge (\!|t_1|\!)^u) \vee ((\!|\neg\ A|\!)^{\mathsf{t}} \wedge (\!|t_2|\!)^u) \vee ((\!|\neg\ \triangle\ A|\!)^{\mathsf{t}} \wedge u \mathrel{\boldsymbol{=}} \uparrow) \ .$$

The following rules are for the translation of recursive function definitions to formulae of L_ω:

$$(\!|f(x_1 : T_1,\ldots,x_n : T_n)\ T \mathrel{\underline{\triangle}} t|\!)^{\mathsf{t}} =$$
$$\forall \boldsymbol{x}_1,\ldots,\boldsymbol{x}_n, y \cdot \mathsf{U}(\boldsymbol{x}_1) \wedge \ldots \wedge \mathsf{U}(\boldsymbol{x}_n) \wedge \mathsf{U}(y) \ \Rightarrow$$
$$(\boldsymbol{f}(\boldsymbol{x}_1,\ldots,\boldsymbol{x}_n) \mathrel{\boldsymbol{=}} y \ \Leftrightarrow$$
$$(y \not\mathrel{\boldsymbol{=}} \uparrow \wedge\ D(\boldsymbol{x}_1,\ldots,\boldsymbol{x}_n,y)) \vee$$
$$(y \mathrel{\boldsymbol{=}} \uparrow \wedge \neg\ \exists y' \cdot \mathsf{U}(y') \wedge D(\boldsymbol{x}_1,\ldots,\boldsymbol{x}_n,y'))) \ ,$$

where

$$D \quad := \quad \mathrm{Fix}(\Lambda F.\{\boldsymbol{x}_1,\ldots,\boldsymbol{x}_n,y \mid \boldsymbol{T}_1(\boldsymbol{x}_1) \wedge \ldots \wedge \boldsymbol{T}_n(\boldsymbol{x}_n) \wedge \boldsymbol{T}(y) \wedge \sigma((\!|t|\!)^y)\}) \ ,$$

and

$$(\!|f(x_1 : T_1,\ldots,x_n : T_n)\ T \mathrel{\underline{\triangle}} t|\!)^{\mathsf{f}} \quad = \quad \neg\,(\!|f(x_1 : T_1,\ldots,x_n : T_n)\ T \mathrel{\underline{\triangle}} t|\!)^{\mathsf{t}} \ .$$

The inference rules (if-1), (if-2), (if-3), (Func-def) and (Func-ind) become derived rules of L_ω after translation.

For the function *fac* defined above, the translation of the body of the definition, $(\!|t|\!)^y$, is logically equivalent to

$$(\boldsymbol{n} \mathrel{\boldsymbol{=}} 0 \wedge y \mathrel{\boldsymbol{=}} 1) \vee (\boldsymbol{n} \not\mathrel{\boldsymbol{=}} 0 \wedge y \mathrel{\boldsymbol{=}} \boldsymbol{n} * \boldsymbol{fac}(\boldsymbol{n} - 1))$$

under the assumption that $n : \mathbb{Z}$. After applying the mapping σ, we obtain the following recursive definition of the corresponding predicate:

$$\boldsymbol{Fac} =$$
$$\{\boldsymbol{n}, y \mid (\!|\mathbb{Z}|\!)(\boldsymbol{n}) \wedge (\!|\mathbb{Z}|\!)(y) \wedge$$
$$((\boldsymbol{n} \mathrel{\boldsymbol{=}} 0 \wedge y \mathrel{\boldsymbol{=}} 1) \vee (\boldsymbol{n} \not\mathrel{\boldsymbol{=}} 0 \wedge \exists z \cdot \boldsymbol{Fac}(\boldsymbol{n} - 1, z) \wedge y \mathrel{\boldsymbol{=}} \boldsymbol{n} * z))\} \ .$$

After applying Fix to the corresponding predicate operator, we obtain a defining formula logically equivalent to

$$\forall \boldsymbol{n}, y \cdot \mathsf{U}(\boldsymbol{n}) \wedge \mathsf{U}(y) \ \Rightarrow$$
$$(\boldsymbol{fac}(\boldsymbol{n}) \mathrel{\boldsymbol{=}} y \ \Leftrightarrow$$
$$(\neg\,(\!|\mathbb{Z}|\!)(\boldsymbol{n}) \wedge y \mathrel{\boldsymbol{=}} \uparrow) \vee$$
$$(\boldsymbol{n} < 0 \wedge y \mathrel{\boldsymbol{=}} \uparrow) \vee$$
$$(\boldsymbol{n} \mathrel{\boldsymbol{=}} 0 \wedge y \mathrel{\boldsymbol{=}} 1) \vee$$
$$(\boldsymbol{n} \mathrel{\boldsymbol{=}} 1 \wedge y \mathrel{\boldsymbol{=}} \boldsymbol{n}) \vee$$
$$(\boldsymbol{n} \mathrel{\boldsymbol{=}} 2 \wedge y \mathrel{\boldsymbol{=}} \boldsymbol{n} * (\boldsymbol{n} - 1)) \vee$$
$$(\boldsymbol{n} \mathrel{\boldsymbol{=}} 3 \wedge y \mathrel{\boldsymbol{=}} \boldsymbol{n} * (\boldsymbol{n} - 1) * (\boldsymbol{n} - 2)) \vee$$
$$\vdots$$
$$) \ .$$

4. Base types and type formers

In the VDM notation, one has base types such as the *boolean* type $\mathbb{B}$, whose elements are the truth values, and the *natural* type $\mathbb{N}$, whose elements are the natural numbers. Other types can be constructed from the base types by means of type formers such as the *set* type former $\bullet$-set and the *sequence* type former $\bullet^*$. The elements of τ-set are the finite sets with elements of type τ and the elements of τ^* are the finite sequences with elements of type τ. Another useful type former is the *union* type former $\bullet|\bullet$. Its use is necessary in recursive type definitions (treated in Sect. 5). The elements of $\tau_1|\tau_2$ are the values that are elements of τ_1 or τ_2.

Instead of describing the extension of LPF for base types and type formers fully, only the adaptations for the natural type, the sequence type former and the union type former are described in this section. Other base types can be treated in the same vein as the natural type and other type formers can be treated in the same vein as the sequence type former. The union type former is quite different from the other type formers.

First, the additional formation rules, inference rules and interpretation rules for the natural type, the sequence type former and the union type former are given. Thereafter, their embedding into L_ω is defined. A meta-rule for the creation of induction rules for inductively defined types is also given.

4.1. LPF and types

The logic LPF is also used in VDM to reason about VDM's base types and the types constructed from them by means of VDM's type formers. The treatment of these types can be made precise by defining another conservative extension of LPF. It requires the introduction of type expressions. The required adaptations of the formation rules, inference rules, etc. of LPF and the extension for recursive function definitions to the introduction of this syntactic category are trivial: type symbols are simply identified with type expressions. However, the current extension requires more.

The following formation rules for type expressions are required:

1. type symbols are type expressions;
2. $\mathbb{N}$ is a type expression;
3. if τ is a type expression, then τ^* is a type expression;
4. if τ_1 and τ_2 are type expressions, then $\tau_1|\tau_2$ is a type expression.

The following are additional inference rules concerning the natural type and the sequence types:

$$\mathbb{N}\text{-gen-b}\ \frac{}{0 : \mathbb{N}} \qquad \mathbb{N}\text{-gen-i}\ \frac{t : \mathbb{N}}{succ(t) : \mathbb{N}}$$

$$\mathbb{N}\text{-ind}\ \frac{[x := 0]A \quad x : \mathbb{N}, A \vdash [x := succ(x)]A}{x : \mathbb{N} \vdash A}$$

$$\text{Seq-gen-b}\ \frac{}{[\,] : \tau^*} \qquad \text{Seq-gen-i}\ \frac{t_1 : \tau \quad t_2 : \tau^*}{cons(t_1, t_2) : \tau^*}$$

$$\text{Seq-ind}\ \frac{[x_2 := [\,]]A \quad x_1 : \tau, x_2 : \tau^*, A \vdash [x_2 := cons(x_1, x_2)]A}{x_2 : \tau^* \vdash A}$$

The rules ($\mathbb{N}$-ind) and (Seq-ind) are induction rules for natural numbers and finite sequences, respectively.

The following are additional rules of inference concerning union types:

$$\boxed{|\text{-I}}\ \frac{t : \tau_1 \vee t : \tau_2}{t : \tau_1 | \tau_2} \qquad \boxed{|\text{-E}}\ \frac{t : \tau_1 | \tau_2}{t : \tau_1 \vee t : \tau_2}$$

A structure $\mathbf{A}$ with signature Σ has the following additional restrictions on $\mathcal{U}^{\mathbf{A}}$:

1a. $\mathcal{N} \subseteq \mathcal{U}^{\mathbf{A}} - \{\bot\}$;
1b. for every $S \subseteq \mathcal{U}^{\mathbf{A}} - \{\bot\}$, $S^* \subseteq \mathcal{U}^{\mathbf{A}} - \{\bot\}$.

Here $\mathcal{N}$ denotes the set of all natural numbers.

The additional interpretation function for type expressions is inductively defined by

$$\begin{aligned} [\![T]\!]^{\mathbf{A}} &= T^{\mathbf{A}} , \\ [\![\mathbb{N}]\!]^{\mathbf{A}} &= \mathcal{N} , \\ [\![\tau^*]\!]^{\mathbf{A}} &= ([\![\tau]\!]^{\mathbf{A}})^* , \\ [\![\tau_1 | \tau_2]\!]^{\mathbf{A}} &= [\![\tau_1]\!]^{\mathbf{A}} \cup [\![\tau_2]\!]^{\mathbf{A}} . \end{aligned}$$

The soundness of the inference rules concerning the natural type, the sequence types and the union types with respect to this interpretation is obvious.

The VDM notation does not have dependent types. Therefore, the interpretation of any type expression remains the same under different assignments. Formulae such as $[\,] \neq 0$ are not excluded syntactically, because typing is not decidable in the VDM notation – due to its subtyping mechanism (described in Sect. 5).

4.2. Embedding types into L_ω

Type expressions can also be embedded into L_ω. They can be mapped to defined predicates. The inference rules concerning the various types become derived rules of L_ω after translation. So the translation justifies these rules as well.

Inductive definitions in L_ω

In Sect. 3, defined predicates, predicate operators and a fixpoint operator were introduced as abbreviations to facilitate expressing recursive definitions as formulae of L_ω. A large class of inductive definitions can also be expressed as formulae. To describe the formulae concerned, we use the following additional notation:

- $\overline{x_1, \ldots, x_n}$ with the meaning given by $\overline{x_1, \ldots, x_n} = \{y_1, \ldots, y_n \mid y_1 \neq x_1 \vee \ldots \vee y_n \neq x_n\}$;
- $[P^+ := D]A$ is the result of replacing the defined predicate D for the positive occurrences of the predicate symbol P in A.

In the case of an inductive definition A of a predicate P, the formula A is transformed into a continuous predicate operator Φ with the property that Fix(Φ) is the smallest P satisfying A. Under certain mild conditions, the predicate operator $\Phi = \lambda P.\{x_1, \ldots, x_n \mid \neg\, [P^+ := \overline{x_1, \ldots, x_n}]A\}$ turns out to be appropriate. This is described in detail in Appendix A.

Embedding into L_ω

Type expressions are translated to defined predicates by a mapping

$$(\!|\bullet|\!) : \mathcal{X}_{\mathrm{LPF}} \to \mathcal{D}_{L_\omega} ,$$

where $\mathcal{X}_{\mathrm{LPF}}$ denotes the set of all type expressions and $\mathcal{D}_{L_\omega}$ denotes the set of all defined predicates. Intuitively, $(\!|\tau|\!)$ is the defined predicate D such that $t : \tau$ is true in LPF is stated by $\exists y \cdot (\!|t|\!)^y \wedge D(y)$. This mapping is inductively defined by

$$\begin{aligned}
(\!|T|\!) &= \boldsymbol{T} ,\\
(\!|\mathbb{N}|\!) &= \mathrm{Fix}(\Lambda P.\{y \mid \neg[P^+ := \overline{y}](P(0) \wedge \forall y_1 \cdot P(y_1) \Rightarrow P(succ(y_1)))\}) ,\\
(\!|\tau^*|\!) &= \mathrm{Fix}(\Lambda Q.\{y \mid \neg[Q^+ := \overline{y}]\\
&\qquad (Q([\,]) \wedge \forall y_1, y_2 \cdot (\!|\tau|\!)(y_1) \wedge Q(y_2) \Rightarrow Q(cons(y_1, y_2)))\}) ,\\
(\!|\tau_1|\tau_2|\!) &= \{y \mid (\!|\tau_1|\!)(y) \vee (\!|\tau_2|\!)(y)\} .
\end{aligned}$$

Note that

$$P(0) \wedge \forall y_1 \cdot P(y_1) \Rightarrow P(succ(y_1))$$

and

$$Q([\,]) \wedge \forall y_1, y_2 \cdot (\!|\tau|\!)(y_1) \wedge Q(y_2) \Rightarrow Q(cons(y_1, y_2))$$

are the usual inductive definitions of the set of all natural numbers and the set of all finite sequences over a given set $(\!|\tau|\!)$, respectively. After replacing $\overline{y}$ for the positive occurrences of P and Q, respectively, in these formulae and taking the negation of the resulting formulae, we obtain the usual recursive definitions:

$$P = \{y \mid y = 0 \vee \exists y_1 \cdot P(y_1) \wedge y = succ(y_1)\}$$

and

$$Q = \{y \mid y = [\,] \vee \exists y_1, y_2 \cdot (\!|\tau|\!)(y_1) \wedge Q(y_2) \wedge y = cons(y_1, y_2)\} .$$

After applying Fix to the corresponding predicate operators, we obtain defined predicates $(\!|\mathbb{N}|\!)$ and $(\!|\tau^*|\!)$. One easily verifies that

$$(\!|\mathbb{N}|\!)(y) \Leftrightarrow \bigvee_n y = succ^n(0) ,$$

where

$$\begin{aligned}
succ^0(t) &:= t ,\\
succ^{n+1}(t) &:= succ(succ^n(t)) ,
\end{aligned}$$

and

$$(\!|\tau^*|\!)(y) \Leftrightarrow \bigvee_n A_n ,$$

where

$$\begin{aligned}
A_0 &:= y = [\,] ,\\
A_{n+1} &:= \exists y_1, \ldots, y_{n+1} \cdot (\!|\tau|\!)(y_1) \wedge \ldots \wedge (\!|\tau|\!)(y_{n+1}) \wedge\\
&\qquad y = cons(y_1, \ldots, cons(y_{n+1}, [\,]) \cdots) .
\end{aligned}$$

These formulae define the predicates concerned correctly. So the transformation works for the inductive definitions of $(\!|\mathbb{N}|\!)$ and $(\!|\tau^*|\!)$. This was to be expected because the form

of the inductive definitions (the Horn formulae form) guarantees that the applicability conditions for the transformation are met.

The above shows that the embedding in L_ω for other base types and types constructed by means of other type formers can be easily obtained if we know a way to generate any element of the type concerned.

It is easy to see that the inference rules concerning union types become derived rules of L_ω after translation. A corollary from one of the justifications of the meta-rule about inductive definitions given below is that it is also the case for the rules concerning the natural type and the sequence types.

4.3. A meta-rule about induction rules

All base types and types constructed by means of type formers can be defined inductively in LPF by an instance of the following schema:

$$\begin{array}{l} c_1 : \tau \wedge \ldots \wedge c_n : \tau \wedge \\ (\forall x_1^1 : \tau_1^1, \ldots, x_{n_1}^1 : \tau_{n_1}^1 \cdot f_1(x_1^1, \ldots, x_{n_1}^1) : \tau) \\ \qquad \wedge \\ \qquad \vdots \\ \qquad \wedge \\ (\forall x_1^m : \tau_1^m, \ldots, x_{n_m}^m : \tau_{n_m}^m \cdot f_m(x_1^m, \ldots, x_{n_m}^m) : \tau) \ . \end{array}$$

Fact 3. If the inductive definition of a type τ is an instance of the above schema, then the corresponding instance of the induction rule schema

$$\frac{\begin{array}{c} [x := c_1]A \quad \ldots \quad [x := c_n]A \\ x_1^1 : \tau_1^1, \ldots, x_{n_1}^1 : \tau_{n_1}^1, \{[x := x_i^1]A | \tau_i^1 \equiv \tau\} \vdash [x := f_1(x_1^1, \ldots, x_{n_1}^1)]A \\ \vdots \\ x_1^m : \tau_1^m, \ldots, x_{n_m}^m : \tau_{n_m}^m, \{[x := x_i^m]A | \tau_i^m \equiv \tau\} \vdash [x := f_m(x_1^m, \ldots, x_{n_m}^m)]A \end{array}}{x : \tau \vdash A}$$

is a sound rule of inference.

Proof. After transforming the translation of the inductive definition as described in the previous subsection, we obtain the following defining formula for $(\!|\tau|\!)$:

$$(\!|\tau|\!)(y) \ \Leftrightarrow \ \bigvee_n A_n \ ,$$

where

$$\begin{array}{rcl} A_0 & := & y = c_1 \vee \ldots \vee y = c_n \ , \\ A_{n+1} & := & \end{array}$$

$$
\begin{array}{l}
A_n \vee \\
(\exists \boldsymbol{x}^1_1, \ldots, \boldsymbol{x}^1_{n_1} \cdot \\
\quad \bigwedge_{i \in \{i \mid \tau^1_i \not\equiv \tau\}} (\!|\tau^1_i|\!)(\boldsymbol{x}^1_i) \wedge \bigwedge_{i \in \{i \mid \tau^1_i \equiv \tau\}} [y := \boldsymbol{x}^1_i]A_n \wedge y = \boldsymbol{f}_1(\boldsymbol{x}^1_1, \ldots, \boldsymbol{x}^1_{n_1})) \\
\quad \vee \\
\quad \vdots \\
\quad \vee \\
(\exists \boldsymbol{x}^m_1, \ldots, \boldsymbol{x}^m_{n_m} \cdot \\
\quad \bigwedge_{i \in \{i \mid \tau^m_i \not\equiv \tau\}} (\!|\tau^m_i|\!)(\boldsymbol{x}^m_i) \wedge \bigwedge_{i \in \{i \mid \tau^m_i \equiv \tau\}} [y := \boldsymbol{x}^m_i]A_n \wedge y = \boldsymbol{f}_m(\boldsymbol{x}^m_1, \ldots, \boldsymbol{x}^m_{n_m})) \,.
\end{array}
$$

This is the construction of the inductive closure of the set $\{c_1, \ldots, c_n\}$ under the functions $f_1, \ldots, f_m$ expressed in L_ω. The induction rule follows directly from the induction principle for inductive sets and $(\!|x : \tau|\!)^{\mathrm{t}} \Leftrightarrow (\!|\tau|\!)(\boldsymbol{x})$.

Another justification can be given by showing that the induction rule becomes a derived rule of L_ω after translation. After translation, we can infer

$$
\begin{array}{l}
(A_0 \Rightarrow [\boldsymbol{x} := y](\!|A|\!)^{\mathrm{t}}) \wedge \\
\bigwedge_n ((A_n \Rightarrow [\boldsymbol{x} := y](\!|A|\!)^{\mathrm{t}}) \Rightarrow (A_{n+1} \Rightarrow [\boldsymbol{x} := y](\!|A|\!)^{\mathrm{t}}))
\end{array}
$$

from the hypotheses of the rule. Then $\bigwedge_n (A_n \Rightarrow [\boldsymbol{x} := y](\!|A|\!)^{\mathrm{t}})$ follows by transitivity of implication. $(\!|\tau|\!)(\boldsymbol{x}) \Rightarrow (\!|A|\!)^{\mathrm{t}}$, the translation of the conclusion of the rule, is a direct consequence.

It follows immediately from this alternative justification that the inference rules concerning the natural type and the sequence types become derived rules of L_ω after translation.

5. Subtypes and recursively defined types

As well as recursive function definitions, recursive type definitions can be represented in L_ω. So the rules used for reasoning about recursively defined types in LPF become also derived rules of L_ω. In addition to type formers and recursion, restriction of types to subtypes is used in VDM to define types.

In this section, first the extension of LPF for subtypes is described and thereafter the extension for recursive type definitions. For both extensions, the additional formation rules, inference rules and interpretation rules as well as the translation rules for the embedding into L_ω are given.

5.1. Subtypes

In the VDM notation, a type can also be a subtype of another type specified by means of an *invariant*. For example, sequences without repeating elements are defined as follows:

$$
\begin{array}{l}
\mathit{Useq} = \mathit{Elem}^* \\
\mathsf{inv}\ \mathit{inv\text{-}Useq}(s) \triangleq \mathit{is\text{-}uniques}(s)
\end{array}
$$

An obvious definition of *is-uniques* is

$$\textit{is-uniques} : Elem^* \to \mathbb{B}$$
$$\textit{is-uniques}(s) \triangleq \forall i, j : \mathbb{N}_1 \cdot i, j \in \mathsf{inds}\, s \wedge i \neq j \Rightarrow s(i) \neq s(j)$$

For a precise treatment of these subtypes in a further extension of LPF, the following additional formation rule for type expressions is required:

5. if x is a variable symbol, τ is a type expression and A is a formula with $\mathit{free}(A) \subseteq \{x\}$, then $< x : \tau \mid A >$ is a type expression.

$< x : \tau \mid A >$ corresponds to the subtype of τ denoted in the VDM notation by

$$\tau \text{ inv } \textit{inv-}T(x) \triangleq A$$

(T is a name introduced for the subtype).

The following are additional inference rules concerning subtypes:

$$\boxed{\text{subtype-I}}\ \frac{t : \tau \wedge [x := t]A}{t :< x : \tau \mid A >} \qquad \boxed{\text{subtype-E}}\ \frac{t :< x : \tau \mid A >}{t : \tau \wedge [x := t]A}$$

The following is the additional interpretation rule for subtypes:

$$[\![< x : \tau \mid A >]\!]^{\mathbf{A}} \quad = \quad \{d \in [\![\tau]\!]^{\mathbf{A}} \mid [\![A]\!]^{\mathbf{A}}_{\alpha(x \to d)}\}\ ,$$

where α is an arbitrary assignment in **A**. The soundness of the inference rules concerning subtypes with respect to this interpretation is obvious.

The following additional translation rule for type expressions makes these inference rules derived rules of L_ω after translation:

$$(\!|< x : \tau \mid A >|\!) \quad = \quad \{\boldsymbol{x} \mid (\!|\tau|\!)(\boldsymbol{x}) \wedge (\!|A|\!)^{\mathrm{t}}\}\ .$$

So subtypes can also be embedded into L_ω.

Justification of induction rules for subtypes by means of the inference rules given above generally requires proofs by induction. For sequences without repeating elements, the appropriate induction rule is:

$$\boxed{\textit{Useq}\text{-ind}}\ \frac{[x_2 := [\,]]A \qquad x_1 : Elem, x_2 : \textit{Useq}, x_1 \notin \mathsf{elems}\, x_2, A \vdash [x_2 := cons(x_1, x_2)]A}{x_2 : \textit{Useq} \vdash A}$$

is-uniques can just as well be defined as follows:

$$\begin{array}{l}\textit{is-uniques} : Elem^* \to \mathbb{B} \\ \textit{is-uniques}(s) \triangleq \\ \quad s = [\,] \vee \\ \quad \exists h : Elem, t : Elem^* \cdot \textit{is-uniques}(t) \wedge h \notin \mathsf{elems}\, t \wedge s = cons(h, t)\end{array}$$

This definition shows the restrictions under which the generation of sequences yields exactly the sequences without repeating elements. Such *constructive* definitions of invariants make it easy to create induction rules for subtypes.

We can capture the creation of an induction rule for a subtype from an associated constructively defined invariant in a meta-rule as well, because the approach described for base types and type formers generalizes to types that can be defined inductively in LPF by an instance of the following schema:

$$
\begin{array}{l}
c_1 : \tau \wedge \ldots \wedge c_n : \tau \wedge \\
(\forall x_1^1 : \tau_1^1, \ldots, x_{n_1}^1 : \tau_{n_1}^1 \cdot A_1 \;\Rightarrow\; f_1(x_1^1, \ldots, x_{n_1}^1) : \tau) \\
\qquad \wedge \\
\qquad \vdots \\
\qquad \wedge \\
(\forall x_1^m : \tau_1^m, \ldots, x_{n_m}^m : \tau_{n_m}^m \cdot A_m \;\Rightarrow\; f_m(x_1^m, \ldots, x_{n_m}^m) : \tau) \;,
\end{array}
$$

where the formulae $A_1, \ldots, A_m$ do not contain τ.

5.2. Recursive type definitions

In the VDM notation, a type can also be introduced by a recursive type definition $T = \tau$. For example, LISP-like lists can be defined by $L = \mathbb{N} | L^*$. The use of the union type former is necessary in recursive type definitions.

For a precise treatment of recursive type definitions, an additional formation rule for formulae is required:

9. if $T \in \mathrm{T}(\Sigma)$ and τ is a type expression, then $T = \tau$ is a formula.

In the rules used for reasoning about recursively defined types, recursive type definitions are used as formulae.

Additional inference rules are:[6]

$$\boxed{\text{Type-def}}\ \frac{t : \tau}{T = \tau \vdash t : T}$$

$$\boxed{\text{Type-ind}}\ \frac{[T :=\{\,\}]A \qquad A \vdash [T := \tau]A}{T = \tau \vdash A}\ \tau \text{ continuous in } T,\ A \text{ admissible in } T$$

The type definition hypothesis is usually dropped when it is clear from the context which definition is meant.

We say that τ is *continuous* in T iff the mapping from types to types that maps T to τ is continuous with respect to the 'less than' ordering given below. A sufficient syntactic condition is: in every type expression of the form $< x : \tau' \mid A' >$ occurring in τ, T does not occur in A'.

We say that A is *admissible* in T iff, for every chain of types $S_0 \subseteq S_1 \subseteq S_2 \subseteq \ldots$ (where $\subseteq$ is the 'less than' ordering) contained in the set of all types T satisfying A, its least upper bound is also in that set. The following syntactic properties characterize a large class of admissible formulae. Formulae of the forms $t' : T'$, $\neg\,(t' : T')$ and $\triangle(t' : T')$ are admissible if in every term of the form $\mathsf{if}\ A'\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2$ occurring in the formula concerned, T does not occur in A'. Also admissible are formulae in which T does not occur. The preservation properties are as in case of functions (treated in Sect. 3).

For sets $S, S' \subseteq \mathcal{U} - \{\bot\}$, S is *less than* S' iff $S \subseteq S'$.

The following is the additional interpretation rule for recursive type definitions:

$$
[\![T = \tau]\!]^{\mathbf{A}}_{\alpha} =
\begin{cases}
\mathsf{T} & \text{if } T^{\mathbf{A}} \text{ is the least } S \subseteq \mathcal{U}^{\mathbf{A}} - \{\bot\} \text{ such that } S = [\![\tau]\!]^{\mathbf{A}'}_{\alpha} \\
\mathsf{F} & \text{otherwise,}
\end{cases}
$$

[6] Here $\{\,\}$ denotes the empty type. The use of terms that denote sets as types is described in the next section.

where $\mathbf{A}'$ is the structure with signature Σ such that $w^{\mathbf{A}'} = w^{\mathbf{A}}$ if $w \not\equiv T$ and $T^{\mathbf{A}'} = S$.

The soundness of the inference rules concerning recursive type definitions with respect to this interpretation is seen in a similar way to recursive function definitions.

The following additional translation rules make them derived rules of L_ω after translation:

$$\begin{aligned} (\!|T = \tau|\!)^{\mathsf{t}} &= \forall y \cdot \boldsymbol{T}(y) \Leftrightarrow \mathrm{Fix}(\Lambda \boldsymbol{T}.\{y \mid y \neq \uparrow \wedge (\!|\tau|\!)(y)\}) , \\ (\!|T = \tau|\!)^{\mathsf{f}} &= \neg (\!|T = \tau|\!)^{\mathsf{t}} . \end{aligned}$$

So recursive type definitions can also be embedded into L_ω.

Induction rules for recursively defined types can be justified by means of the rule (Type-ind). For the LISP-like list, the induction rule

$$\boxed{L\text{-ind}}\ \frac{x : \mathbb{N} \vdash A \quad [x := [\,]]A \qquad x_1 : L, x_2 : L^*, [x := x_1]A, [x := x_2]A \vdash [x := cons(x_1, x_2)]A}{x : L \vdash A}$$

can be derived. The derivation is similar to the derivation of (structural) induction rules from the fixpoint induction rule of PPλ in [Pau87]. Note that this result is in accordance with the meta-rule about induction rules (think of the inductive definition of the type L). One might doubt the type correctness of substituting x_2 for x in A above, but $x_2 : L^*$ implies $x_2 : L$ according to the rules ($|$ -I) and (Type-def).

6. Miscellaneous matters

In the VDM notation, terms of type $\mathbb{B}$ are used as formulae and vice versa. This requires trivial additional formation rules and interpretation rules for terms and formulae as well as a restriction on the structures in which they are interpreted. The following are the inference rules concerning the interchangeability of formulae and terms of type $\mathbb{B}$:

$$\boxed{\mathbb{B}\text{ -I}}\ \frac{\delta(A)}{A : \mathbb{B}} \qquad \boxed{\mathbb{B}\text{ -E}}\ \frac{t : \mathbb{B}}{\delta(t)} \qquad \boxed{\Leftrightarrow \text{ as } =}\ \frac{A_1 \Leftrightarrow A_2}{A_1 = A_2}$$

Note that formulae t, where t is a term that is not of type $\mathbb{B}$, are not excluded syntactically – because typing is not decidable in VDM. The last rule permits derivation of the rule:

$$\boxed{\mathbb{B}\text{ -exh}}\ \frac{t : \mathbb{B}}{t = \mathsf{true} \vee t = \mathsf{false}}$$

The following additional translation rules make the inference rules concerning the interchangeability of formulae and terms of type $\mathbb{B}$ derivable in L_ω after translation:

$$\begin{aligned} (\!|A|\!)^u &= ((\!|A|\!)^{\mathsf{t}} \wedge u = \mathsf{t}) \vee ((\!|A|\!)^{\mathsf{f}} \wedge u = \mathsf{f}) \vee (\neg ((\!|A|\!)^{\mathsf{t}} \vee (\!|A|\!)^{\mathsf{f}}) \wedge u = \uparrow) , \\ (\!|t|\!)^{\mathsf{t}} &= (\!|t|\!)^{\mathsf{t}} , \\ (\!|t|\!)^{\mathsf{f}} &= (\!|t|\!)^{\mathsf{f}} . \end{aligned}$$

In the right-hand side of the last two rules, $(\!|t|\!)^{\mathsf{t}}$ and $(\!|t|\!)^{\mathsf{f}}$ are applications of the embedding function for terms.

In the VDM notation, terms of set types are used as types as well. This also requires some simple adaptations. The following additional rules of inference are needed:

$$\boxed{\in\!\!\rightarrow\!:}\ \frac{t \in t'}{t : t'} \qquad \boxed{:\!\!\rightarrow\!\in}\ \frac{t' : \tau\text{-set} \quad t : t'}{t \in t'}$$

Note that typing assertions $t : t'$ where t' is a term that is not of a set type cannot be excluded syntactically. For this reason, the first hypothesis of the second inference rule is needed.

The following is the rule for set comprehension appropriate for set types:

$$\boxed{\text{Set-compr}}\ \frac{\{x : \tau \mid A\} : \tau\text{-set}}{t \in \{x : \tau \mid A\} \ \Leftrightarrow\ t : \tau \wedge [x := t]A}$$

A direct consequence is the following derived rule:

$$\boxed{\text{Set-as-type}}\ \frac{\{x : \tau \mid A\} : \tau\text{-set}}{t : \{x : \tau \mid A\} \ \Leftrightarrow\ t :< x : \tau \mid A >}$$

The (common) hypothesis of these rules is needed because the set denoted by $\{x : \tau \mid A\}$ may be infinite.

The following additional translation rule makes the inference rules concerning the use of terms of set type as types derivable in L_ω after translation:

$$(\!|t|\!) \quad = \quad \{y \mid \exists y' \cdot (\!|t|\!)^{y'} \wedge y \in y'\} \ .$$

7. Closing remarks

This paper gives a comprehensive description of a typed version of the logic known as LPF (Sect. 2) and some extensions which are used with VDM (Sects. 3, 4 and 5). The logical justification of the inference rules concerned – by means of an embedding into classical infinitary logic – is new. Further discussion of problems of finding a proof theory for VDM can be found in [FM93]; material which shows how theories are built using the proof theory is covered in [BFL93].

The induction rules for recursively defined functions (Sect. 3) and types (Sect. 5) – which are reminiscent of the fixpoint induction principle – as well as the meta-rule about induction rules for base types and types constructed by means of type formers (Sect. 4), were not presented before. They give a firm foundation to the way in which recursive definitions of functions and types are rendered into inference rules in VDM. It is further demonstrated that constructive definitions of invariants (Sect. 5) are useful in devising induction rules for subtypes.

From the experience with VVSL [Mid93], we know that the extensions for other aspects of VDM such as implicit specification of functions and operations can be treated in the same vein. The proof obligations associated with such implicit specifications as well as the proof obligations associated with data reification and operation decomposition can also be given a logical justification. Hence it appears that VDM as described in [Jon90] can be justified entirely in classical (infinitary) logic. As a matter of course, higher-order and polymorphic functions need heavier machinery.

Acknowledgement. Our thanks go to Gerard Renardel de Lavalette for his help related to this paper. We are also grateful to an anonymous referee for his detailed and valuable comments on a draft of the paper.

References

[BCJ84] Barringer, H., Cheng, J.H., Jones, C.B.: A logic covering undefinedness in program proofs. Acta Informatica, **21**, 251–269 (1984)

[BFL93] Bicarregui, J.C., Fitzgerald, J.S., Lindsay, P.A., Moore, R., Ritchie, B.: Proof in VDM: A Practitioner's Guide. FACIT, Springer-Verlag 1993

[Che86] Cheng, J.H.: A Logic for Partial Functions. PhD thesis UMCS-86-7-1, University of Manchester, Department of Computer Science, 1986

[CJ91] Cheng, J.H., Jones, C.B.: On the usability of logics which handle partial functions. In: Morgan, C., Woodcock, J.C.P. (eds.) 3rd Refinement Workshop, pp. 51–69. Workshops in Computing Series, Springer-Verlag 1991

[FM93] Fitzgerald, J.S., Moore, R.: Experiences in developing a proof theory for VDM specifications. Technical Report TR424, University of Newcastle upon Tyne, Department of Computing Science, 1993. Also in: Andrews, D.J., Groote, J.F., Middelburg, C.A. (eds.) Semantics of Specification Languages. Workshops in Computing Series, Springer-Verlag (to appear)

[Jon90] Jones, C.B.: Systematic Software Development Using VDM (2nd edition). Prentice-Hall International Series in Computer Science, Prentice-Hall 1990

[Kei71] Keisler, H.J.: Model Theory for Infinitary Logic. Studies in Logic, vol. 62, North-Holland 1971

[KR89] Koymans, C.P.J., Renardel de Lavalette, G.R.: The logic MPL_ω. In: Wirsing, M., Bergstra, J.A. (eds.) Algebraic Methods: Theory, Tools and Applications, pp. 247–282. Lect. Notes Comput. Sci., vol. 394, Springer-Verlag 1989

[KTB88] Konikowska, B., Tarlecki, A., Blikle, A.: A three-valued logic for software specification and validation. In: Bloomfield, R., Marshall, L, Jones, R. (eds.) VDM '88, pp. 218–242. Lect. Notes Comput. Sci., vol. 328, Springer-Verlag 1988

[Mid93] Middelburg, C.A.: Logic and Specification – Extending VDM-SL for advanced formal specification. Computer Science: Research and Practice, vol. 1, Chapman & Hall 1993

[MR91] Middelburg, C.A., Renardel de Lavalette, G.R.: LPF and MPL_ω – A logical comparison of VDM-SL and COLD-K. In: Prehn, S., Toetenel, W.J. (eds.) VDM '91, vol. 1, pp. 279–308. Lect. Notes Comput. Sci., vol. 551, Springer-Verlag 1991

[Pau87] Paulson, L.C.: Logic and Computation. Cambridge Tracts in Theoretical Computer Science, vol. 2, Cambridge University Press 1987

[Ren89] Renardel de Lavalette, G.R.: COLD-K^2, the static kernel of COLD-K. Report RP/mod-89/8, Software Engineering Research Centrum, Utrecht, 1989

[Sco67] Scott, D.S.: Existence and description in formal logic. In: Schoenman, R. (ed.) Bertrand Russell, Philosopher of the Century, pp. 181–200. Allen & Unwin 1967

[Sun83] Sundhold, G.: Systems of deduction. In: Gabbay, D., Guenther, F. (eds.) Handbook of Philosophical Logic, Chap. I.2. D. Reidel Publishing Company 1983

A. L_ω

In this appendix, L_ω is reviewed in brief. For a comprehensive discussion of this logic, see e.g. [Kei71]. How recursive definitions and inductive definitions can be expressed as formulae of L_ω is also described in this appendix. The method concerned was worked out (for MPL_ω) by Renardel in [Ren89].

Introduction to L_ω

In L_ω, there are no type symbols. A signature is just a set of function symbols and predicate symbols. The formulae that contain only function symbols and predicate symbols from a signature Σ constitute the language of L_ω over Σ or the language of $L_\omega(\Sigma)$. The corresponding proof system is analogously called the proof system of $L_\omega(\Sigma)$.

The language of $L_\omega(\Sigma)$ contains terms and formulae. The terms of $L_\omega(\Sigma)$ are inductively defined by the following formation rules:

1. variable symbols are terms;
2. if $f \in \mathrm{F}(\Sigma)$, *arity*$(f) = n$ and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.

The formulae of $\mathrm{L}_\omega(\Sigma)$ are inductively defined by the following formation rules:

1. false is a formula;
2. if $P \in \mathrm{P}(\Sigma)$, *arity*$(P) = n$ and $t_1, \ldots, t_n$ are terms, then $P(t_1, \ldots, t_n)$ is a formula;
3. if t_1 and t_2 are terms, then $t_1 = t_2$ is a formula;
4. if A is formula, then $\neg\, A$ is a formula;
5. if $\langle A_n \rangle_{n<\omega} = \langle A_0, A_1, \ldots \rangle$ are formulae, then $\bigwedge_n A_n$ is a formula;
6. if A is a formula and x is a variable symbol, then $\forall x \cdot A$ is a formula.

The string representation of formulae as suggested by these formation rules can lead to syntactic ambiguities. Parentheses are used to avoid such ambiguities.

The symbol $=$ is used for equality in L_ω. It is classical equality: $t_1 = t_2$ is true if t_1 and t_2 denote the same object and $t_1 = t_2$ is false otherwise. So classical equality differs from weak equality (=) in LPF and coincides with strong equality (==) in LPF.

Countable disjunctions and binary conjunctions are defined as abbreviations as follows:

$$\begin{array}{lcl} \bigvee_n A_n & := & \neg \bigwedge_n \neg\, A_n\ , \\ A_1 \wedge A_2 & := & \bigwedge_n A'_n, \text{ where } A'_0 = A_1 \text{ and } A'_n = A_2 \text{ for } 0 < n < \omega\ . \end{array}$$

Binary disjunction, implication, equivalence and existential quantification are defined as abbreviations as for LPF.

The proof system of L_ω is formulated here as a sequent calculus for proofs in natural deduction style. It is defined by the following rules of inference:

$$\boxed{\neg\,\neg\text{-E}}\ \frac{\neg\,\neg\, A}{A} \qquad \boxed{\text{false-E}}\ \frac{\text{false}}{A}$$

$$\boxed{\neg\text{-I}}\ \frac{A_1 \vdash A_2 \quad A_1 \vdash \neg\, A_2}{\neg\, A_1} \qquad \boxed{\neg\text{-E}}\ \frac{A_1 \quad \neg\, A_1}{A_2}$$

$$\boxed{\textstyle\bigwedge\text{-I}}\ \frac{\langle A_n \rangle_{n<\omega}}{\bigwedge_n A_n} \qquad \boxed{\textstyle\bigwedge\text{-E}}\ \frac{\bigwedge_n A_n}{A_n}\ \text{for all } n$$

$$\boxed{\forall\text{-I}}\ \frac{A}{\forall x \cdot A} \qquad \boxed{\forall\text{-E}}\ \frac{\forall x \cdot A}{[x := t]A}$$

$$\boxed{=\text{-refl}}\ \frac{}{t = t} \qquad \boxed{=\text{-sub}}\ \frac{t_1 = t_2 \quad [x := t_1]A}{[x := t_2]A}$$

The following are some derived rules:

$$\boxed{\neg\text{-I}'}\ \frac{A_1 \vdash \text{false}}{\neg\, A_1} \qquad \boxed{\neg\text{-E}'}\ \frac{A_1 \quad \neg\, A_1}{\text{false}}$$

$$\boxed{\wedge\text{-I}}\ \frac{A_1 \quad A_2}{A_1 \wedge A_2} \qquad \boxed{\wedge\text{-E}}\ \frac{A_1 \wedge A_2}{A_i}\ \text{for } i = 1, 2$$

Recursive definitions in L_ω

In L_ω, a large class of recursive definitions of functions can be expressed as formulae. To show how these formulae can be obtained, we introduce some notation and abbreviations.

A *defined predicate* is an expression of the form $\{x_1, \ldots, x_n \mid A\}$, where $x_1, \ldots, x_n$ are distinct variable symbols and A is a formula; n is called the *arity* of the defined predicate. For terms $t_1, \ldots, t_n$, $\{x_1, \ldots, x_n \mid A\}(t_1, \ldots, t_n)$ is defined as an abbreviation of a formula by

$$\{x_1, \ldots, x_n \mid A\}(t_1, \ldots, t_n) \quad := \quad [x_1 := t_1, \ldots, x_n := t_n]A \ .$$

A predicate symbol P of arity n is identified with the defined predicate

$$\{x_1, \ldots, x_n \mid P(x_1, \ldots, x_n)\} \ .$$

A recursive function definition $f(x_1 : T_1, \ldots, x_n : T_n)\, T \triangleq t$ can be expressed as a formula if there exists a defined predicate D that uniquely determines the function being defined in the sense that the value of f at $x_1, \ldots, x_n$ is y iff the formula $D(x_1, \ldots, x_n, y)$ is true.

The following abbreviations of defined predicates are used:

$$\begin{aligned} \emptyset_n \quad &:= \quad \{x_1, \ldots, x_n \mid \mathsf{false}\} \ , \\ \bigcup_{m \in \omega} \{x_1, \ldots, x_n \mid A_m\} \quad &:= \quad \{x_1, \ldots, x_n \mid \bigvee\nolimits_m A_m\} \ . \end{aligned}$$

The arity indication n as subscript of $\emptyset$ is dropped when it is clear from the context or unimportant which arity is meant.

Inclusion and extensional *equality* between defined predicates are defined as abbreviations of formulae by

$$\begin{aligned} \{x_1, \ldots, x_n \mid A_1\} \subseteq \{x_1, \ldots, x_n \mid A_2\} \quad &:= \quad \forall x_1, \ldots, x_n \cdot A_1 \Rightarrow A_2 \ , \\ \{x_1, \ldots, x_n \mid A_1\} = \{x_1, \ldots, x_n \mid A_2\} \quad &:= \quad \forall x_1, \ldots, x_n \cdot A_1 \Leftrightarrow A_2 \ . \end{aligned}$$

Substitution for predicate symbols is defined as for variable symbols. Let P be a predicate symbol, D be a defined predicate and A be a formula. Then $[P := D]A$ is the result of replacing the defined predicate D for the occurrences of the predicate symbol P in A, avoiding that free variables in D become bound.

If a predicate is recursively defined, then the definition determines a mapping from predicates to predicates. Its least fixpoint is considered to be the predicate being defined. Predicate operators correspond to mappings from predicates to predicates.

A *predicate operator* is an expression of the form $\Lambda P.D$, where P is a predicate symbol and $D = \{x_1, \ldots, x_n \mid A\}$ is a defined predicate. For a defined predicate D' of the same arity as P, $(\Lambda P.D)(D')$ is defined as an abbreviation of a defined predicate by

$$(\Lambda P.\{x_1, \ldots, x_n \mid A\})(D') \quad := \quad \{x_1, \ldots, x_n \mid [P := D']A\} \ .$$

For a predicate operator $\Phi = \Lambda P.D$ where P and D are of the same arity, Fix(Φ), the *fixpoint* of Φ, is defined as an abbreviation of a defined predicate by

$$\mathrm{Fix}(\Phi) \quad := \quad \bigcup_{m \in \omega} \Phi^m(\emptyset) \ ,$$

where

$$\begin{aligned} \Phi^0(D) &:= D\ , \\ \Phi^{m+1}(D) &:= \Phi(\Phi^m(D))\ . \end{aligned}$$

If Φ is a continuous predicate operator, then one can prove that Fix(Φ) is indeed the least fixpoint of Φ.

A predicate operator $\Phi = \Lambda P.D$ is *continuous* iff

$$\{D_m \subseteq D_{m+1} \mid m < \omega\} \vdash \Phi(\bigcup_{m\in\omega} D_m) = \bigcup_{m\in\omega} \Phi(D_m)$$

is provable for arbitrary defined predicates $D_1, D_2, \ldots$ of the same arity as P.

Fact 4. If the predicate operator $\Phi = \Lambda P.D$ is continuous and P and D are of the same arity, then Fix(Φ) is the least fixpoint of Φ, i.e.

$$\Phi(\text{Fix}(\Phi)) = \text{Fix}(\Phi) \text{ and } \Phi(P) \subseteq P \ \Rightarrow\ \text{Fix}(\Phi) \subseteq P$$

are provable.

Proof. Fix(Φ) is Kleene's least fixpoint construction (which stops at ω for a continuous operator), expressed in L_ω.

This guarantees that a large class of recursive predicate definitions can be expressed as formulae.

The following are derived rules:

$$\boxed{\text{Fix-=}}\ \frac{}{\Phi(\text{Fix}(\Phi)) = \text{Fix}(\Phi)}\ \Phi \text{ continuous}$$

$$\boxed{\text{Fix-ind}}\ \frac{[P := \emptyset]A \quad A \vdash [P := \Phi(P)]A}{[P := \text{Fix}(\Phi)]A}\ \Phi \text{ continuous, } A \text{ admissible}$$

where $\Phi = \Lambda P.D$ with P and D of the same arity. The latter rule is a ***fixpoint induction*** rule. Formula A is ***admissible*** iff

$$\{D_m \subseteq D_{m+1} \mid m < \omega\} \vdash \textstyle\bigwedge_m [P := D_m]A \ \Rightarrow\ [P := \bigcup_{m\in\omega} D_m]A$$

is provable for arbitrary defined predicates $D_1, D_2, \ldots$ of the same arity as P.

In case of a recursive function definition, the definition is first replaced by a recursive definition of a predicate that uniquely determines the function concerned. For a function f and corresponding predicate F, the replacement is given by the mapping σ defined below. It is assumed that, in a formula containing f, every occurrence of f is provided with a unique index i (to indicate this we write f_i). For each index i, x_i denotes a distinct variable symbol not free in the transformed term or formula. The mapping σ and an auxiliary mapping ϵ are simultaneously defined by the following rules:

$$
\begin{array}{lll}
\epsilon(t) & = \mathsf{true} & \text{if } f \text{ not in } t, \\
\epsilon(f_i(t_1,\ldots,t_n)) & = F(\sigma(t_1),\ldots,\sigma(t_n),x_i)\ , & \\
\epsilon(g(t_1,\ldots,t_m)) & = \epsilon(t_1)\wedge\ldots\wedge\epsilon(t_m) & \text{if } g \text{ different from } f, \\
 & & \\
\sigma(t) & = t & \text{if } f \text{ not in } t, \\
\sigma(f_i(t_1,\ldots,t_n)) & = x_i\ , & \\
\sigma(g(t_1,\ldots,t_m)) & = g(\sigma(t_1),\ldots,\sigma(t_m)) & \text{if } g \text{ different from } f, \\
\sigma(P(t_1,\ldots,t_m)) & = P(t_1,\ldots,t_m) & \text{if } f \text{ not in } t_1,\ldots,t_m, \\
\sigma(t_1 \mathrel{=\!\!=} t_2) & = t_1 \mathrel{=\!\!=} t_2 & \text{if } f \text{ not in } t_1,t_2, \\
\sigma(P(t_1,\ldots,t_m)) & = & \\
\multicolumn{3}{l}{\quad \exists x_1,\ldots,x_l\cdot\epsilon(t_1)\wedge\ldots\wedge\epsilon(t_m)\wedge P(\sigma(t_1),\ldots,\sigma(t_m)) \quad \text{if } f \text{ in } t_1,\ldots,t_m,} \\
\sigma(t_1 \mathrel{=\!\!=} t_2) & = & \\
\multicolumn{3}{l}{\quad \exists x_1,\ldots,x_l\cdot\epsilon(t_1)\wedge\epsilon(t_2)\wedge\sigma(t_1)\mathrel{=\!\!=}\sigma(t_2) \qquad \text{if } f \text{ in } t_1,t_2,}
\end{array}
$$

where $x_1,\ldots,x_l$ are the variables x_i occurring in $\epsilon(t_1)\wedge\ldots\wedge\epsilon(t_m)$ and $\epsilon(t_1)\wedge\epsilon(t_2)$, respectively,

σ commutes with the logical connectives and quantifiers.

Inductive definitions in L_ω

The previous subsection shows how recursive predicate definitions can be expressed as formulae of L_ω. In case of an inductive definition A of a predicate P, the idea is to transform the formula A into a continuous predicate operator Φ with the property that Fix(Φ) is the smallest P satisfying A. To show how the predicate operator concerned can be obtained, we introduce some additional notation and abbreviations.

The following abbreviation of defined predicates is used:

$$\overline{x_1,\ldots,x_n} \quad := \quad \{y_1,\ldots,y_n \mid y_1 \neq x_1 \vee \ldots \vee y_n \neq x_n\}\ .$$

Let P be a predicate symbol, D be a defined predicate and A be a formula. Then $[P^+ := D]A$ and $[P^- := D]A$ are the results of replacing the defined predicate D for the positive occurrences and the negative occurrences, respectively, of the predicate symbol P in A, avoiding that free variables in D become bound.

For an inductive definition A of a predicate P (of arity n), the predicate operator $\Phi = \lambda P.\{x_1,\ldots,x_n \mid \neg\,[P^+ := \overline{x_1,\ldots,x_n}]A\}$ turns out to be appropriate under certain conditions.

The formula A is *complement preserving* for P iff

$$[P := \{x_1,\ldots,x_n \mid \neg\,[P := \overline{x_1,\ldots,x_n}]A\}]A$$

is provable.

Fact 5. If $\Phi = \lambda P.\{x_1,\ldots,x_n \mid \neg\,[P^+ := \overline{x_1,\ldots,x_n}]A\}$ is a continuous predicate operator and $[P^- := Q]A$ is complement preserving for P, then Fix(Φ) is the smallest predicate P satisfying A, i.e.

$$[P := \text{Fix}(\Phi)]A \text{ and } A \ \Rightarrow\ \text{Fix}(\Phi) \subseteq P$$

are provable.

Proof. This is proved almost exactly as Theorem D.2.5. in [Ren89].

This guarantees that a large class of inductive predicate definitions can be expressed as formulae. For example, if a predicate P is inductively defined by a (finite or infinite) conjunction of formulae of the form

$$\forall x_1, \ldots, x_l \cdot A_1 \wedge \ldots \wedge A_m \;\Rightarrow\; P(t_1, \ldots, t_n) \;,$$

where every formula A_i is of the form $P(t'_1, \ldots, t'_n)$ or does not contain P, then the definition can be expressed as a formula in L_ω.

This article was processed by the author using the LaTeX style file *pljour1* from Springer-Verlag.