Constructive Data Refinement in Typed Lambda Calculus

Furio Honsell[1,2], John Longley[1], Donald Sannella[1], and Andrzej Tarlecki[3,4]

[1] Laboratory for Foundations of Computer Science, University of Edinburgh
[2] Dipartimento di Matematica e Informatica, Università di Udine
[3] Institute of Informatics, Warsaw University
[4] Institute of Computer Science, Polish Academy of Sciences

Abstract. A new treatment of data refinement in typed lambda calculus is proposed, phrased in terms of *pre-logical relations* [HS99] rather than logical relations, and incorporating a constructive element. Constructive data refinement is shown to have desirable properties, and a substantial example of refinement is presented.

1 Introduction

One of the activities involved in developing programs from specifications is the transformation of "abstract programs" involving types of data that are not normally available as primitive in programming languages (graphs, sets, etc.) into "concrete programs" in which a representation of these in terms of simpler types of data is provided. Apart from the change to data representation, such *data refinement* should have no effect on the results computed by the program: the concrete program should be equivalent to the abstract program in the sense that all computational observations should return the same results in both cases.

The standard treatment of data refinement in the context of typed lambda calculus, originating with Reynolds in [Rey81, Rey83] but described most clearly in [Ten94], cf. Sect. 8.5 of [Mit96], uses *logical relations* to prove the correctness of refinements. This work has its roots in [Hoa72], which proposes that the correctness of the concrete program be verified using an invariant on the domain of concrete values together with a function mapping concrete values (that satisfy the invariant) to abstract values. In algebraic terms, what is required is a homomorphism from a subalgebra of the concrete algebra to the abstract algebra. A strictly more general method is to take a homomorphic relation (a so-called *correspondence* [Sch90], cf. [Mil71]) in place of a homomorphism from a subalgebra. Logical relations extend these ideas to deal with higher-order functions.

Proof method ([Ten94]). *Let $\mathcal{A}$ and $\mathcal{B}$ be Σ-Henkin models and let OBS, the* observable types, *be a subset of Types(Σ). To show that $\mathcal{B}$ is a refinement of $\mathcal{A}$, find a logical relation $\mathcal{R}$ over $\mathcal{A}$ and $\mathcal{B}$ such that R^σ is the identity relation for each $\sigma \in OBS$. We then say that $\mathcal{B}$ is a* logical refinement *of $\mathcal{A}$ and write $\mathcal{A} \rightsquigarrow \mathcal{B}$, or $\mathcal{A} \overset{\mathcal{R}}{\rightsquigarrow} \mathcal{B}$ when we want to make $\mathcal{R}$ explicit.*

J. Tiuryn (Ed.): FOSSACS 2000, LNCS 1784, pp. 161–176, 2000.

It is well-known that the composition of two logical relations is not in general a logical relation. It follows that, given logical refinements $\mathcal{A} \overset{\mathcal{R}}{\rightsquigarrow} \mathcal{B}$ and $\mathcal{B} \overset{\mathcal{S}}{\rightsquigarrow} \mathcal{C}$, the composition $\mathcal{S} \circ \mathcal{R}$ cannot in general be used as a witness for the composed refinement $\mathcal{A} \rightsquigarrow \mathcal{C}$. (In fact, the problem is more serious than it appears at first: sometimes there is no witness for $\mathcal{A} \rightsquigarrow \mathcal{C}$ at all, see Sect. 3.) This is at odds with the *stepwise* nature of refinement, and the transitivity of the underlying concept of refinement expressed in terms of observational equivalence. It is one source of examples demonstrating the incompleteness of the above proof method; there are other examples that do not involve composition of refinement steps, see e.g. Sect. 5. The proof method is complete in the absence of higher-order term constants [Mit96].

In [HS99], a weakening of the notion of logical relations called *pre-logical relations* was studied; cf. [PPST00]. Pre-logical relations are closed under composition; in fact, they are the minimal weakening of logical relations with this property. They completely characterize observational equivalence of Henkin models, without restriction to first-order signatures. Replacing logical relations with pre-logical relations in the above gives a notion of *pre-logical refinement* which is in pleasing harmony with stepwise refinement. Indeed, it is equivalent to the underlying concept of refinement, i.e. sound and complete as a proof method.

This is an improvement but pre-logical refinement still does not entirely accord with our intuition concerning data refinement and stepwise development of programs. For one thing, like logical refinement it is a symmetric relation. We will consider a more elaborate notion of data refinement, called *constructive pre-logical refinement* (Sect. 4). This is a relation between specifications, written $SP \underset{\delta}{\overset{OBS}{\rightsquigarrow\!\!\!\rightsquigarrow}} SP'$, which incorporates a *construction* in the form of a *derived signature morphism* δ taking models of SP' to Henkin models over the signature of SP. Derived signature morphisms define the types and constants in one signature by giving terms over another signature, and this corresponds directly to the code in an ML functor body. It follows that the result of a complete chain of constructive refinements is a Henkin model, corresponding to a modular ML program, which is a solution to the original programming task. We give an extended example of constructive data refinement in the context of exact real number computation, and show that it is not a (constructive) logical refinement (Sect. 5).

Some recent accounts of data refinement in typed lambda calculus have employed variants of logical relations that are related to pre-logical relations, for instance [KOPTT97]. Our inclusion of a constructive element in the relation is new, and our example appears to be the first non-trivial concrete example of data refinement in the lambda calculus literature.

The idea of constructive pre-logical refinement comes from the world of algebraic specifications, where it is called *abstractor implementation* [ST88] or *behavioural implementation* [ST97]. This paper is an attempt to explain this idea in lambda calculus terms, since it is a substantial improvement on current accounts of data refinement in that context. One novelty with respect to existing work on abstractor implementations concerns the connection with pre-logical relations, which generalizes Schoett's characterization of observational equivalence via correspondences and makes a bridge with work on data refinement in lambda calculus based on logical relations. Another novelty concerns the use of derived signature morphisms in the typed lambda calculus for defining constructions. In order for abstractor implementations to compose, constructions are required to preserve observational equivalence, a property known as *stability* [Sch87]. This requirement is normally imposed as an assumption on the language used for defining constructions, which is left unspecified. Here, stability follows easily from the Basic Lemma of pre-logical relations. Finally, the example in Sect. 5 goes considerably beyond the simple examples of refinement of data representation that have been considered previously.

2 Preliminaries: Syntax and Semantics

For the sake of simplicity of the exposition we restrict attention to $\lambda^{\rightarrow}$, the simply-typed lambda calculus having $\rightarrow$ as its only type constructor.

Definition 2.1. *The set of* types *over a set B of* base types *(or* type constants*) is given by the grammar $\sigma ::= b \mid \sigma \rightarrow \sigma$ where b ranges over B. A* signature *Σ consists of a set B of type constants and a collection C of typed* term constants *$c : \sigma$. $Types(\Sigma)$ denotes the set of types over B.*

In a *Σ-context* $\Gamma = x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n$, we require that $x_i \neq x_j$ for all $1 \leq i < j \leq n$ and $\sigma_i \in Types(\Sigma)$ for all $1 \leq i \leq n$. *Σ-terms* are given by the grammar $M ::= x \mid c \mid \lambda x{:}\sigma.M \mid M\ M$ where x ranges over variables and c over term constants. The usual typing rules associate each well-formed term M in context Γ with a type $\sigma \in Types(\Sigma)$, written $\Gamma \rhd M : \sigma$ (or $\Gamma \rhd_{\Sigma} M : \sigma$ when we need to make Σ explicit). If Γ is empty then we write simply $M : \sigma$ or $\rhd_{\Sigma} M : \sigma$.

Definition 2.2. *A Σ-*Henkin model *$\mathcal{A}$ consists of:*

- *a* carrier *set $[\![\sigma]\!]^{\mathcal{A}}$ for each $\sigma \in Types(\Sigma)$;*
- *a function $App_{\mathcal{A}}^{\sigma,\tau} : [\![\sigma \rightarrow \tau]\!]^{\mathcal{A}} \rightarrow [\![\sigma]\!]^{\mathcal{A}} \rightarrow [\![\tau]\!]^{\mathcal{A}}$ for each $\sigma, \tau \in Types(\Sigma)$;*
- *an element $[\![c]\!]^{\mathcal{A}} \in [\![\sigma]\!]^{\mathcal{A}}$ for each term constant $c : \sigma$ in Σ; and*
- *elements $K_{\mathcal{A}}^{\sigma,\tau} \in [\![\sigma \rightarrow (\tau \rightarrow \sigma)]\!]^{\mathcal{A}}$ and $S_{\mathcal{A}}^{\rho,\sigma,\tau} \in [\![(\rho \rightarrow \sigma \rightarrow \tau) \rightarrow (\rho \rightarrow \sigma) \rightarrow \rho \rightarrow \tau]\!]^{\mathcal{A}}$ for each $\rho, \sigma, \tau \in Types(\Sigma)$*

such that

- *$K_{\mathcal{A}}^{\sigma,\tau}\, x\, y = x$ and $S_{\mathcal{A}}^{\rho,\sigma,\tau}\, x\, y\, z = (x\, z)(y\, z)$; and*
- *(extensionality) if $App_{\mathcal{A}}^{\sigma,\tau}\, f\, x = App_{\mathcal{A}}^{\sigma,\tau}\, g\, x$ for every $x \in [\![\sigma]\!]^{\mathcal{A}}$, then $f = g$.*

The class of all Σ-Henkin models is denoted $Mod(\Sigma)$.

Term constants of functional type are interpreted as *total* functions. Allowing partial functions does not seem to introduce problems, but we have not checked the details. Moreover, the use of Henkin models in this paper is not essential; the definitions and results in [HS99] that we will need later also apply to non-extensional models, for instance combinatory algebras.

A Γ*-environment* η on a Henkin model $\mathcal{A}$ assigns elements of $\mathcal{A}$ to variables, with $\eta(x) \in [\![\sigma]\!]^{\mathcal{A}}$ for $x : \sigma$ in Γ. A Σ-term $\Gamma \rhd M : \sigma$ is interpreted in $\mathcal{A}$ under a Γ-environment η in the usual way with λ-abstraction interpreted via translation to combinators, written $[\![\Gamma \rhd M : \sigma]\!]^{\mathcal{A}}_{\eta}$, and this is an element of $[\![\sigma]\!]^{\mathcal{A}}$. If M is closed then we write simply $[\![M : \sigma]\!]^{\mathcal{A}}$.

We can allow terms to contain the fixed-point combinator Y (viewed as a term constant). To interpret such terms in a Henkin model $\mathcal{A}$, we need to additionally require elements $Y^{\sigma}_{\mathcal{A}} \in [\![(\sigma \to \sigma) \to \sigma]\!]^{\mathcal{A}}$ for each $\sigma \in \mathit{Types}(\Sigma)$ such that $f\ (Y^{\sigma}_{\mathcal{A}}\ f) = Y^{\sigma}_{\mathcal{A}}\ f$. We will assume that this additional structure is present whenever we consider such terms.

Definition 2.3. *A* logical relation $\mathcal{R}$ *over* Σ*-Henkin models* $\mathcal{A}$ *and* $\mathcal{B}$ *is a family of relations* $\{R^{\sigma} \subseteq [\![\sigma]\!]^{\mathcal{A}} \times [\![\sigma]\!]^{\mathcal{B}}\}_{\sigma \in \mathit{Types}(\Sigma)}$ *such that:*

- $R^{\sigma \to \tau}(f, g)$ *iff* $\forall a \in [\![\sigma]\!]^{\mathcal{A}}.\forall b \in [\![\sigma]\!]^{\mathcal{B}}.R^{\sigma}(a, b) \Rightarrow R^{\tau}(\mathit{App}_{\mathcal{A}}\ f\ a, \mathit{App}_{\mathcal{B}}\ g\ b)$.
- $R^{\sigma}([\![c]\!]^{\mathcal{A}}, [\![c]\!]^{\mathcal{B}})$ *for every term constant* $c : \sigma$ *in* Σ.

Definition 2.4. *If* $\Gamma \rhd_{\Sigma} M : \sigma$ *and* $\Gamma \rhd_{\Sigma} M' : \sigma$ *then* $\forall \Gamma.M =_{\sigma} M'$ *is a* Σ-equation. *The subscript* σ *is omitted when it is obvious. A* Σ*-Henkin model* $\mathcal{A}$ satisfies *a* Σ*-equation* $\forall \Gamma.M =_{\sigma} M'$ *if* $[\![\Gamma \rhd M : \sigma]\!]^{\mathcal{A}}_{\eta} = [\![\Gamma \rhd M' : \sigma]\!]^{\mathcal{A}}_{\eta}$ *for all* Γ*-environments* η*. It is easy to add connectives and quantifiers, giving sentences of predicate logic with equality. A* specification SP *consists of a signature* Σ *and a set* Φ *of* Σ*-sentences. Then* $\mathit{Sig}(SP) = \Sigma$*, and* $\mathit{Mod}(SP)$ *(the* models *of* SP*) is the class of all* Σ*-Henkin models satisfying all the sentences in* Φ*.*

3 Data Refinement

We begin with an analysis of the failure of composition of logical relations and its impact on composition of logical refinements.

Example 3.1. Let Σ contain two type constants, b and b', and no term constants. Consider Σ-Henkin models $\mathcal{A}, \mathcal{B}, \mathcal{C}$ which interpret b and b' as follows, and interpret function types using full set-theoretic function spaces: $[\![b]\!]^{\mathcal{A}} = \{*\} = [\![b']\!]^{\mathcal{A}}$; $[\![b]\!]^{\mathcal{B}} = \{*\}$ and $[\![b']\!]^{\mathcal{B}} = \{\circ, \bullet\}$; $[\![b]\!]^{\mathcal{C}} = \{\circ, \bullet\} = [\![b']\!]^{\mathcal{C}}$. Let $\mathcal{R}$ be the logical relation over $\mathcal{A}$ and $\mathcal{B}$ induced by $R^{b} = \{\langle *, * \rangle\}$ and $R^{b'} = \{\langle *, \circ \rangle, \langle *, \bullet \rangle\}$ and let $\mathcal{S}$ be the logical relation over $\mathcal{B}$ and $\mathcal{C}$ induced by $S^{b} = \{\langle *, \circ \rangle, \langle *, \bullet \rangle\}$ and $S^{b'} = \{\langle \circ, \circ \rangle, \langle \bullet, \bullet \rangle\}$. $\mathcal{S} \circ \mathcal{R}$ is not a logical relation because it does not relate the identity function in $[\![b]\!]^{\mathcal{A}} \to [\![b']\!]^{\mathcal{A}}$ to the identity function in $[\![b]\!]^{\mathcal{C}} \to [\![b']\!]^{\mathcal{C}}$. The problem is that the only two functions in $[\![b]\!]^{\mathcal{B}} \to [\![b']\!]^{\mathcal{B}}$ are $\{* \mapsto \circ\}$ and $\{* \mapsto \bullet\}$, and $\mathcal{S}$ does not relate these to the identity in $\mathcal{C}$. □

This simple example shows that we may have logical refinements $\mathcal{A} \overset{\mathcal{R}}{\rightsquigarrow} \mathcal{B}$ and $\mathcal{B} \overset{\mathcal{S}}{\rightsquigarrow} \mathcal{C}$ (where we take $OBS = \emptyset$ in both cases), where $\mathcal{S} \circ \mathcal{R}$ is not a logical relation and so cannot act as witness to $\mathcal{A} \rightsquigarrow \mathcal{C}$.

One possible solution might be to construct the relations at higher types from the composite relations at base types. This works if Σ contains only first-order term constants (guaranteeing that the restriction to base types lifts to a logical relation) and if *OBS* contains only base types (guaranteeing that the resulting logical relation is the identity relation for each $\sigma \in OBS$). The following example shows how this idea fails in the presence of second-order term constants.

Example 3.2. In the previous example, add a type constant *bool* and a term constant $c : (b \to b') \to bool$ to Σ. Let $[\![bool]\!]^{\mathcal{A}} = [\![bool]\!]^{\mathcal{B}} = [\![bool]\!]^{\mathcal{C}} = \{true, false\}$ and take $\mathcal{R}^{bool}$ and $\mathcal{S}^{bool}$ to be the identity. In each model, let the interpretation of c take constant functions to *true* and all other functions to *false*. The resulting $\mathcal{R}$ and $\mathcal{S}$ are logical relations. As before, $\mathcal{S} \circ \mathcal{R}$ is not a logical relation but now the restriction of $\mathcal{S} \circ \mathcal{R}$ to base types cannot be lifted to a logical relation either: this would relate the identity function in $[\![b]\!]^{\mathcal{A}} \to [\![b']\!]^{\mathcal{A}}$ (which is a constant function) to every function in $[\![b]\!]^{\mathcal{C}} \to [\![b']\!]^{\mathcal{C}}$, but then the constant function in $\mathcal{A}$ would be related to non-constant functions in $\mathcal{C}$, and so $[\![c]\!]^{\mathcal{A}}$ could not be related to $[\![c]\!]^{\mathcal{C}}$, otherwise *true* would be related to *false*. □

These two examples show that certain ways of composing the logical relations witnessing $\mathcal{A} \rightsquigarrow \mathcal{B}$ and $\mathcal{B} \rightsquigarrow \mathcal{C}$ do not yield a logical relation witnessing $\mathcal{A} \rightsquigarrow \mathcal{C}$. Such a witness may exist, however, and in the above example it does. For $OBS = \emptyset$, the full relation suffices; for $OBS = \{bool\}$, the full relation on b together with the empty relation on b' and the identity on *bool* lifts to a logical relation. But if we add constants $b1, b2 : b$ and $b1', b2' : b'$ with $[\![b1]\!]^{\mathcal{A}} = [\![b2]\!]^{\mathcal{A}} = [\![b1']\!]^{\mathcal{A}} = [\![b2']\!]^{\mathcal{A}} = *$, $[\![b1]\!]^{\mathcal{C}} = [\![b1']\!]^{\mathcal{C}} = \circ$ and $[\![b2]\!]^{\mathcal{C}} = [\![b2']\!]^{\mathcal{C}} = \bullet$ then there is no logical relation over $\mathcal{A}$ and $\mathcal{C}$ which is the identity on *bool* so $\mathcal{A} \not\rightsquigarrow \mathcal{C}$ for $OBS = \{bool\}$. The following proposition summarizes the situation.

Proposition 3.3. *$\mathcal{A} \rightsquigarrow \mathcal{B}$ and $\mathcal{B} \rightsquigarrow \mathcal{C}$ does not in general imply $\mathcal{A} \rightsquigarrow \mathcal{C}$.* □

Ultimately, the justification for the definition of logical refinement lies in the notion of observational equivalence, in terms of which the underlying concept of data refinement is formulated.

Definition 3.4. *Let $\mathcal{A}$ and $\mathcal{B}$ be Σ-Henkin models and let $OBS \subseteq Types(\Sigma)$. Then $\mathcal{A}$ is* observationally equivalent *to $\mathcal{B}$ with respect to OBS, written $\mathcal{A} \equiv_{OBS} \mathcal{B}$, if for any two closed Σ-terms $M, N : \sigma$ for $\sigma \in OBS$, $[\![M : \sigma]\!]^{\mathcal{A}} = [\![N : \sigma]\!]^{\mathcal{A}}$ iff $[\![M : \sigma]\!]^{\mathcal{B}} = [\![N : \sigma]\!]^{\mathcal{B}}$.*

It is usual to take *OBS* to be the "built-in" types for which equality is decidable, for instance *bool* and/or *nat*. Then $\mathcal{A}$ and $\mathcal{B}$ are observationally equivalent iff it is not possible to distinguish between them by performing computational experiments. Note that $OBS \subseteq OBS'$ implies $\equiv_{OBS} \supseteq \equiv'_{OBS}$.

For $OBS = \{nat\}$, the connection between logical refinement and observational equivalence is given by Mitchell's representation independence theorem.

Theorem 3.5 ([Mit96]). *Let Σ be a signature that includes a type constant nat, and let $\mathcal{A}$ and $\mathcal{B}$ be Σ-Henkin models, with $[\![nat]\!]^{\mathcal{A}} = [\![nat]\!]^{\mathcal{B}} = \mathbb{N}$. If there is a logical relation $\mathcal{R}$ over $\mathcal{A}$ and $\mathcal{B}$ with R^{nat} the identity relation on natural numbers, then $\mathcal{A} \equiv_{\{nat\}} \mathcal{B}$. Conversely, if $\mathcal{A} \equiv_{\{nat\}} \mathcal{B}$, Σ provides a closed term* *for each element of* $\mathbb{N}$, *and* Σ *contains only first-order term constants, then there is a logical relation* $\mathcal{R}$ *over* $\mathcal{A}$ *and* $\mathcal{B}$ *with* R^{nat} *the identity relation.* □

The restriction to signatures with first-order term constants in the second part of the theorem is necessary, and this is the key to the incompleteness of logical refinements as a proof method and the problem with composability of logical refinements. If $\mathcal{A} \rightsquigarrow \mathcal{B} \rightsquigarrow \mathcal{C}$ then $\mathcal{A} \equiv_{OBS} \mathcal{B} \equiv_{OBS} \mathcal{C}$, and so $\mathcal{A} \equiv_{OBS} \mathcal{C}$ since $\equiv_{OBS}$ is an equivalence relation. But then it follows that $\mathcal{A} \rightsquigarrow \mathcal{C}$ only for signatures without higher-order term constants.

An improved version of the above theorem, without the restriction to first-order signatures, holds if logical relations are replaced by *pre-logical relations.*

Definition 3.6 ([HS99]). *A* pre-logical relation $\mathcal{R}$ *over* Σ*-Henkin models* $\mathcal{A}$ *and* $\mathcal{B}$ *is a family of relations* $\{R^\sigma \subseteq [\![\sigma]\!]^{\mathcal{A}} \times [\![\sigma]\!]^{\mathcal{B}}\}_{\sigma \in Types(\Sigma)}$ *such that:*

- *If* $R^{\sigma \to \tau}(f, g)$ *then* $\forall a \in [\![\sigma]\!]^{\mathcal{A}}. \forall b \in [\![\sigma]\!]^{\mathcal{B}}. R^\sigma(a, b) \Rightarrow R^\tau(App_{\mathcal{A}}\, f\, a, App_{\mathcal{B}}\, g\, b)$.
- $R^\sigma([\![c]\!]^{\mathcal{A}}, [\![c]\!]^{\mathcal{B}})$ *for every term constant* $c : \sigma$ *in* Σ.
- $R(S_{\mathcal{A}}^{\rho,\sigma,\tau}, S_{\mathcal{B}}^{\rho,\sigma,\tau})$ *and* $R(K_{\mathcal{A}}^{\sigma,\tau}, K_{\mathcal{B}}^{\sigma,\tau})$ *for all* $\rho, \sigma, \tau \in Types(\Sigma)$.

Theorem 3.7 ([HS99]). *Let* $\mathcal{A}$ *and* $\mathcal{B}$ *be* Σ*-Henkin models and let* $OBS \subseteq Types(\Sigma)$. *Then* $\mathcal{A} \equiv_{OBS} \mathcal{B}$ *iff there exists a pre-logical relation over* $\mathcal{A}$ *and* $\mathcal{B}$ *which is a partial injection on* OBS. □

This suggests the following. (We switch to a notation that makes the set of observable types explicit.)

Definition 3.8. *Let* $\mathcal{A}$ *and* $\mathcal{B}$ *be* Σ*-Henkin models and* $OBS \subseteq Types(\Sigma)$. *Then* $\mathcal{B}$ *is a* pre-logical refinement *of* $\mathcal{A}$, *written* $\mathcal{A} \overset{OBS}{\rightsquigarrow} \mathcal{B}$, *if there is a pre-logical relation* $\mathcal{R}$ *over* $\mathcal{A}$ *and* $\mathcal{B}$ *such that* R^σ *is a partial injection for each* $\sigma \in OBS$.

We phrase this as a definition, rather than as a proof method for the underlying notion of data refinement, in contrast to logical refinements. As a proof method it is sound and complete, and therefore equivalent to this underlying notion.

Pre-logical relations compose [HS99], so pre-logical refinements compose, and this explains why stepwise refinement is sound. Another explanation goes via Theorem 3.7: $\mathcal{A} \overset{OBS}{\rightsquigarrow} \mathcal{B} \overset{OBS}{\rightsquigarrow} \mathcal{C} \Rightarrow \mathcal{A} \equiv_{OBS} \mathcal{B} \equiv_{OBS} \mathcal{C} \Rightarrow \mathcal{A} \equiv_{OBS} \mathcal{C} \Rightarrow \mathcal{A} \overset{OBS}{\rightsquigarrow} \mathcal{C}$. The set of observable types need not be the same in both steps, as the following result spells out.

Proposition 3.9. *If* $\mathcal{A} \overset{OBS}{\rightsquigarrow} \mathcal{B}$ *and* $\mathcal{B} \overset{OBS'}{\rightsquigarrow} \mathcal{C}$ *then* $\mathcal{A} \overset{OBS}{\rightsquigarrow} \mathcal{C}$ *provided* $OBS \subseteq OBS'$. □

The definition of observational equivalence may be extended to allow experiments to include the fixed-point combinator by requiring Henkin models to include elements $Y_{\mathcal{A}}^\sigma \in [\![(\sigma \to \sigma) \to \sigma]\!]^{\mathcal{A}}$ for each $\sigma \in Types(\Sigma)$ as indicated above. Theorem 3.7 still holds provided pre-logical relations are required to relate $Y_{\mathcal{A}}^\sigma$ with $Y_{\mathcal{B}}^\sigma$ for all σ.

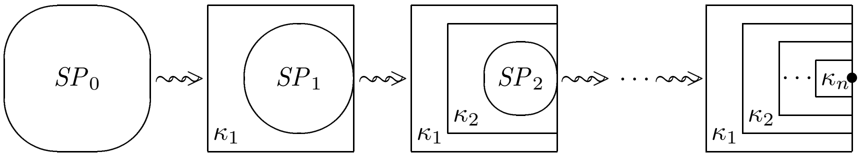
4 Constructive Data Refinement

Pre-logical refinement, like logical refinement, is a symmetric relation. This does not fit the intuition that refinement is about going from an abstract, high-level description of a program to a concrete, more detailed description. There are at least two basic defects of the notion of pre-logical refinement of which the symmetry of the relation is merely a symptom.

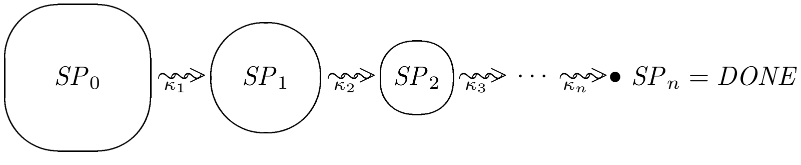
First, it is a relation between Henkin models. The intuition behind stepwise refinement suggests that it should be rather a relation between *descriptions* of Henkin models, i.e. between *specifications*. The original specification of a problem rarely determines a single permissible behaviour: some of the details of the behaviour are normally left open to the implementor. So at this stage one starts with an assortment of models, corresponding to all the possible choices of behaviours. (Some of these will be isomorphic to one another, given a suitable notion of isomorphism, but if the specification permits more than one externally-visible behaviour then there will be non-isomorphic models.) The final program, on the other hand, corresponds to a single Henkin model. So the refinement process involves not just replacement of abstract data representations by more concrete ones, but also selection between permitted behaviours.

Definition 4.1. *Let SP and SP' be specifications with $\Sigma = Sig(SP) = Sig(SP')$ and $OBS \subseteq Types(\Sigma)$. Then SP' is a* pre-logical refinement *of SP, written $SP \overset{OBS}{\rightsquigarrow} SP'$, if for any $\mathcal{B} \in Mod(SP')$ there is some $\mathcal{A} \in Mod(SP)$ with a pre-logical relation $\mathcal{R}$ over $\mathcal{A}$ and $\mathcal{B}$ such that R^σ is a partial injection for each $\sigma \in OBS$.*

Second, the idea that refinement is a *reduction* of one as-yet-unsolved problem to another is not explicit. Intuitively, each refinement step reduces the current problem to a smaller problem, such that any solution to the smaller problem gives rise to a solution to the original problem. In pre-logical refinement of specifications, one models this by having the successive specifications accumulate more and more details arising from successive design decisions. Some parts become fully determined, and remain unchanged as a part of the specification until the final program is obtained. The parts that are not yet fully determined correspond to the unsolved parts of the original problem. (To avoid clutter, we omit the OBS decorations in the following diagrams.)



It is much cleaner to separate the finished parts from the specification, proceeding with the development of the unresolved parts only, giving



where *DONE* is a specification having a ready realisation (e.g. a specification of the built-ins of the programming language in use). The finished parts $\kappa_1, \ldots, \kappa_n$ are *constructions* extending any solution (model) of a reduced problem (specification) to a solution of the previous problem, and so we will refer to this relation as *constructive data refinement.* The signatures of successive specifications may be different, in contrast to the earlier refinement relations.

Constructive data refinement will be defined below. As constructions we will take "δ-reducts" of Σ'-Henkin models induced by "derived signature morphisms" $\delta : \Sigma \to \Sigma'$, where Σ and Σ' are the signatures before and after refinement, respectively. This amounts to giving an interpretation of the type constants and term constants in Σ as types and terms over Σ'.

Definition 4.2. *Let Σ and Σ' be signatures. A* derived signature morphism *$\delta : \Sigma \to \Sigma'$ consists of:*

- *a mapping from base types in Σ to types over Σ': for every base type b in Σ, $\delta(b) \in Types(\Sigma')$. This induces a mapping (also called δ) from $Types(\Sigma)$ to $Types(\Sigma')$, using $\delta(\sigma \to \tau) = \delta(\sigma) \to \delta(\tau)$.*
- *a type-preserving mapping from term constants in Σ to closed terms over Σ': for every $c : \sigma$ in Σ, $\rhd_{\Sigma'} \delta(c) : \delta(\sigma)$.*

This induces a mapping (also called δ) from terms over Σ to terms over Σ', using $\delta(x) = x$, $\delta(\lambda x{:}\sigma.M) = \lambda x{:}\delta(\sigma).\delta(M)$, $\delta(M\,M') = \delta(M)\,\delta(M')$, and (if we are using the Y combinator) $\delta(Y) = Y$. Composition is obvious.

Proposition 4.3. *If $\delta : \Sigma \to \Sigma'$ and $\Gamma \rhd_{\Sigma} M : \sigma$ then $\delta(\Gamma) \rhd_{\Sigma'} \delta(M) : \delta(\sigma)$ where $\delta(x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n) = x_1{:}\delta(\sigma_1), \ldots, x_n{:}\delta(\sigma_n)$.* □

A derived signature morphism corresponds exactly to a *functor* in ML terminology, or a *parameterised program* [Gog84]: the functor parameter is a Σ'-Henkin model, and the functor body contains code which defines the components of Σ using the components of Σ'. If the fixed-point combinator is available then this code may involve recursive functions. (Recursively-defined *types* are not allowed since we are working in $\lambda^{\to}$, but see Sect. 6.)

The semantics of these programs as functions on Henkin models is given by the notion of δ-reduct.

Definition 4.4. *Let $\delta : \Sigma \to \Sigma'$ and let $\mathcal{A}'$ be a Σ'-Henkin model. The δ-*reduct *of $\mathcal{A}'$ is the Σ-Henkin model $\mathcal{A}'|_\delta$ defined as follows:*

- *$[\![\sigma]\!]^{\mathcal{A}'|_\delta} = [\![\delta(\sigma)]\!]^{\mathcal{A}'}$ for each $\sigma \in Types(\Sigma)$;*
- *$App^{\sigma,\tau}_{\mathcal{A}'|_\delta} = App^{\delta(\sigma),\delta(\tau)}_{\mathcal{A}'}$ for each $\sigma, \tau \in Types(\Sigma)$;*

- $[\![c]\!]^{\mathcal{A}'|_\delta} = [\![\delta(c)]\!]^{\mathcal{A}'}$ *for each term constant* $c : \sigma$ *in* Σ*; and*
- $K^{\sigma,\tau}_{\mathcal{A}'|_\delta} = K^{\delta(\sigma),\delta(\tau)}_{\mathcal{A}'}$, $S^{\rho,\sigma,\tau}_{\mathcal{A}'|_\delta} = S^{\delta(\rho),\delta(\sigma),\delta(\tau)}_{\mathcal{A}'}$ *and (if we are using the* Y *combinator)* $Y^{\sigma}_{\mathcal{A}'|_\delta} = Y^{\delta(\sigma)}_{\mathcal{A}'}$, *for each* $\rho, \sigma, \tau \in \mathit{Types}(\Sigma)$.

Proposition 4.5. $[\![\Gamma \rhd_\Sigma M : \sigma]\!]^{\mathcal{A}'|_\delta}_\eta = [\![\delta(\Gamma) \rhd_{\Sigma'} \delta(M) : \delta(\sigma)]\!]^{\mathcal{A}'}_\eta$. □

Σ-Henkin models and pre-logical relations between such models form a category $\mathbf{Mod}(\Sigma)$. The following property is intimately related to the concept of *stability* in [Sch87].

Proposition 4.6 (Stability). *For any* $\delta : \Sigma \to \Sigma'$, *the mapping* $\cdot|_\delta$ *extends to a functor* $\cdot|_\delta : \mathbf{Mod}(\Sigma') \to \mathbf{Mod}(\Sigma)$. *If a pre-logical relation* $\mathcal{R}'$ *in* $\mathbf{Mod}(\Sigma')$ *is a partial injection on* $\mathit{OBS}' \subseteq \mathit{Types}(\Sigma')$, *then* $\mathcal{R}'|_\delta$ *is a partial injection on* $\delta^{-1}(\mathit{OBS}')$. *Thus* $\mathcal{A}' \equiv_{\mathit{OBS}'} \mathcal{B}'$ *implies* $\mathcal{A}'|_\delta \equiv_{\mathit{OBS}} \mathcal{B}'|_\delta$ *for any* $\mathit{OBS} \subseteq \mathit{Types}(\Sigma)$ *such that* $\delta(\mathit{OBS}) \subseteq \mathit{OBS}'$.

Proof. *Take* $(R|_\delta)^\sigma = R^{\delta(\sigma)}$. *It follows from the Basic Lemma for pre-logical relations (see [HS99]) that this yields a pre-logical relation.* □

Now we are ready to give a formal definition of constructive data refinement.

Definition 4.7. *Let* SP *and* SP' *be specifications,* $\delta : \mathit{Sig}(SP) \to \mathit{Sig}(SP')$ *be a derived signature morphism, and let* $\mathit{OBS} \subseteq \mathit{Types}(\mathit{Sig}(SP))$. *Then* SP' *is a* constructive pre-logical refinement *of* SP via δ, *written* $SP \overset{\mathit{OBS}}{\underset{\delta}{\rightsquigarrow}} SP'$, *if for any* $\mathcal{B} \in \mathit{Mod}(SP')$ *there is some* $\mathcal{A} \in \mathit{Mod}(SP)$ *with a pre-logical relation* $\mathcal{R}$ *over* $\mathcal{A}$ *and* $\mathcal{B}|_\delta$ *such that* R^σ *is a partial injection for each* $\sigma \in \mathit{OBS}$.

It is easy to modify this definition to give a notion of constructive logical refinement, written $\underset{\delta}{\rightsquigarrow}$. The correspondence between derived signature morphisms as defined above and ML functors justifies the use of the word "constructive". In Sect. 5 below we give an example of constructive pre-logical refinement and show that it is not a constructive logical refinement.

Constructive pre-logical refinements compose via the composition of their underlying derived signature morphisms:

Proposition 4.8. *If* $SP \overset{\mathit{OBS}}{\underset{\delta}{\rightsquigarrow}} SP'$ *and* $SP' \overset{\mathit{OBS}'}{\underset{\delta'}{\rightsquigarrow}} SP''$ *then* $SP \overset{\mathit{OBS}}{\underset{\delta' \circ \delta}{\rightsquigarrow}} SP''$ *provided* $\delta(\mathit{OBS}) \subseteq \mathit{OBS}'$. □

The required relationship between OBS and OBS' is just what one would expect: as refinement progresses, the successive specifications become increasingly less abstract and so the number of non-observable types tends to decrease, while the overall task of implementing SP with observable types OBS remains the same.

As suggested above, a chain of constructive refinements is complete when the original problem has been reduced to a specification DONE with a given (implemented) model $\mathcal{D}$:

$$SP_0 \overset{\mathit{OBS}_1}{\underset{\delta_1}{\rightsquigarrow}} SP_1 \overset{\mathit{OBS}_2}{\underset{\delta_2}{\rightsquigarrow}} \cdots \overset{\mathit{OBS}_n}{\underset{\delta_n}{\rightsquigarrow}} SP_n = \mathit{DONE}$$

Then, by Prop. 4.8, if the condition on $OBS_1, \ldots, OBS_n$ is satisfied, $DONE$ is a constructive pre-logical refinement of SP_0 via $\delta_n \circ \cdots \circ \delta_2 \circ \delta_1$ with respect to OBS_1: the Henkin model $\mathcal{D}|_{\delta_n \circ \cdots \circ \delta_2 \circ \delta_1}$ is observationally equivalent to some model of SP_0 with respect to OBS_1. In other words, $\delta_n \circ \cdots \circ \delta_2 \circ \delta_1$ is a program that is a solution to the original programming task.

5 An Example from Real Number Computation

We now present an extended example of constructive data refinement in the context of exact real number computation. The point of this example is that the desired refinement can be expressed in terms of pre-logical relations, but not in terms of logical relations.

We will describe a specification SP involving real numbers and some operations on them, and a specification SP' which provides a means of implementing SP using higher-type functions. We will then present a constructive pre-logical refinement $SP \overset{OBS}{\underset{\delta}{\leadsto}} SP'$ that captures this implementation; however, we will show that there is no constructive *logical* refinement $SP \underset{\delta}{\leadsto} SP'$.

5.1 A Specification for Real Number Operations

The specification SP has an underlying signature Σ consisting of the type constants *real* and *bool* and the following term constants:

$$\begin{array}{rlrl} 0, 1 : & \mathit{real} & \mathit{sup}_{[0,1]} : & (\mathit{real} \to \mathit{real}) \to \mathit{real} \\ - : & \mathit{real} \to \mathit{real} & \mathit{true}, \mathit{false}, \bot : & \mathit{bool} \\ +, *, \mathit{max} : & \mathit{real} \to \mathit{real} \to \mathit{real} & < : & \mathit{real} \to \mathit{real} \to \mathit{bool} \end{array}$$

We declare *bool* (only) to be an observable type. As usual, we treat $+, *$ and $<$ as infixes. One could of course consider richer signatures (e.g. with division), but the signature above has the technical advantage that all the above operations are *total* functions in the intended models (see below regarding the interpretation of $\mathit{sup}_{[0,1]}$).

A class of intended models for SP may be given via some logical axioms, as follows. For $0, 1, -, +, *$, we take the usual axioms for a field; we also add axioms saying that the type *real* is totally ordered by $\leq$, where $t \leq u$ abbreviates the logical formula $\exists z{:}\mathit{real}. u = t + (z * z)$. For *max* and $\mathit{sup}_{[0,1]}$, we add the axioms

$$\forall x, y{:}\mathit{real}.\, (x \leq y \Rightarrow \mathit{max}\ x\ y = y) \wedge (y \leq x \Rightarrow \mathit{max}\ x\ y = x)$$

$$\begin{array}{ll} \forall f : \mathit{real} \to \mathit{real}. & (\exists z{:}\mathit{real}.\, \forall x{:}\mathit{real}.\, 0 \leq x \wedge x \leq 1 \Rightarrow f(x) \leq z) \Rightarrow \\ & (\forall z{:}\mathit{real}.\, \mathit{sup}_{[0,1]} f \leq z \Leftrightarrow \forall x{:}\mathit{real}.\, 0 \leq x \wedge x \leq 1 \Rightarrow f(x) \leq z) \end{array}$$

An important logical consequence of these axioms (which we shall use later) is the formula $\mathit{sup}_{[0,1]}(\lambda x{:}\mathit{real}.\, 0) = 0$.

The language we have defined is surprisingly expressive. For instance, every algebraic real number is definable by a closed term, and so any model for SP must contain a copy of at least the algebraic reals. In fact, the models we have in mind contain all the computable or *recursive* reals (though not every computable real is definable by a closed term).

The only purpose of including the type *bool* in *SP* is to allow us to make observations on real numbers. In general we do not expect to be able to tell when two real numbers are the same, but we can tell when they are different. It suffices for our purposes to include the order relation $<$ in our signature. The axioms for $<$ are:

$$\begin{array}{l} \forall x,y{:}\mathit{real}.\, (\neg y \leq x) \Rightarrow x < y = \mathit{true} \\ \forall x,y{:}\mathit{real}.\, (\neg x \leq y) \Rightarrow x < y = \mathit{false} \\ \forall x,y{:}\mathit{real}.\, x = y \Rightarrow x < y = \bot \end{array}$$

This completes the definition of *SP*.

Some brief remarks on models for *SP* may be helpful. The full set-theoretic type structure over $\mathbb{R}$ gives a model of *SP*, though we need to assign arbitrary values to the interpretation of $\mathit{sup}_{[0,1]}$ on functions $f : \mathbb{R} \to \mathbb{R}$ which are unbounded on $[0, 1]$. There are also natural models in which the interpretation of $\mathit{real} \to \mathit{real}$ is constrained to include only *continuous* functions (see e.g. [Nor98]).

5.2 A Specification for PCF Computations

We now present a specification SP' corresponding to the familiar functional language PCF [Plo77]. A constructive refinement $SP \overset{OBS}{\underset{\delta}{\rightsquigarrow}} SP'$ for $OBS = \{\mathit{bool}\}$ then amounts to a way of implementing *SP* in PCF via a "program" δ.

The signature Σ' for SP' will consist of the single type constant *nat* and:

$$\begin{array}{rlrl} 0 : & \mathit{nat} & \mathit{ifzero} : & \mathit{nat} \to \mathit{nat} \to \mathit{nat} \to \mathit{nat} \\ \mathit{succ}, \mathit{pred} : & \mathit{nat} \to \mathit{nat} & Y^{\sigma} : & (\sigma \to \sigma) \to \sigma \qquad (\sigma \in \mathit{Types}(\Sigma')) \end{array}$$

This is exactly the language for (a version of) PCF. The intention is that *nat* stands for the *lifted* natural numbers, with the term $\bot \equiv Y^{nat}(\lambda z{:}nat.z)$ denoting the bottom element. We freely employ syntactic sugar in PCF terms where the meaning is evident.

We now wish to add axioms to ensure that any model for SP' is a model of PCF in some reasonable sense. We do not know whether all the axioms below are strictly necessary for our purposes, but they correspond to a well-understood class of models of PCF. Let us write $t \downarrow$ as an abbreviation for the formula $\mathit{ifzero}\ t\ 0\ 0 = 0$ (we may read this as "t terminates"). First we have an axiom saying there is only one non-terminating element of type *nat*:

$$\forall x{:}nat.\, \neg(x \downarrow) \Leftrightarrow x = \bot$$

For 0 and *succ*, we take the usual first-order Peano axioms for the *terminating* elements. For the remaining constants, we take the axioms

$$\begin{array}{ll} \mathit{pred}\ 0 = 0 & \forall x{:}nat.\ \mathit{pred}(\mathit{succ}\ x) = x \\ \forall y,z{:}nat.\ \mathit{ifzero}\ 0\ y\ z = y & \forall x,y,z{:}nat.\ \mathit{ifzero}(\mathit{succ}\ x)\ y\ z = z \\ \forall y,z{:}nat.\ \mathit{ifzero}\ \bot\ y\ z = \bot & \\ \forall f : \sigma \to \sigma.\ Y^{\sigma} f = f(Y^{\sigma} f) & \forall f : \sigma \to \sigma,\ z{:}\sigma.\ z = f\ z \Rightarrow Y^{\sigma} f \sqsubseteq_{\sigma} z \end{array}$$

where $t \sqsubseteq_{\sigma} u$ abbreviates $\forall P : \sigma \to nat.\ (P\ t \downarrow) \Rightarrow (P\ u \downarrow)$.

Note that the full set-theoretic type structure over $\mathbb{N}_\perp$ is *not* a model, because not every set-theoretic function $\mathbb{N}_\perp \rightarrow \mathbb{N}_\perp$ has a fixed point. However, the usual Scott model based on CPOs (see [Plo77]) and the game models of e.g. [AJM96] do provide models of SP', as do their recursive analogues. The extensional closed term model of PCF also provides a model (which in fact is isomorphic to the recursive game models).

5.3 A Constructive Pre-logical Refinement

We now describe a constructive refinement from SP to SP'. The basic idea is that we will represent a real number r by an infinite sequence $\boldsymbol{d} = d_0 d_1 d_2 \ldots$ of natural numbers, which in turn is represented by the function $f : \mathbb{N}_\perp \rightarrow \mathbb{N}_\perp$ given by $f(i) = d_i$. (More generally: in any model $\mathcal{B}$ of SP', including non-standard ones, there will be an inclusion from $\mathbb{N}_\perp$ to $[\![nat]\!]^{\mathcal{B}}$; for simplicity of notation we take $\mathbb{N}_\perp \subseteq [\![nat]\!]^{\mathcal{B}}$. Then we represent $\boldsymbol{d}$ by any function $f \in [\![nat \rightarrow nat]\!]^{\mathcal{B}}$ such that $f(i) = d_i$ for all $i \in \mathbb{N}$.) Operations on reals are then represented by higher-type operations on such functions. There are many ways to choose a suitable representation, and the differences between them do not matter much. For definiteness, we will work with sequences $\boldsymbol{d}$ such that $d_i \leq 2$ for all $i \geq 2$; such a sequence will represent the real number $d_0 - d_1 + \sum_{i=2}^{\infty} 2^{1-i}(d_i - 1)$. We will use the meta-notation IsReal(f) to mean that $f \in [\![nat \rightarrow nat]\!]^{\mathcal{B}}$ represents a real number in this way, and write Val(f) to denote the real number it represents. Note that there will be many sequences representing any given real number — this is in fact an essential feature of any representation of reals for which even the most basic arithmetical operations are computable. The above choice is essentially a *signed binary* representation involving infinite sequences of digits $-1, 0, 1$ (coded in PCF by $0, 1, 2$ respectively).

We can make precise the idea of implementing SP in terms of SP' by means of a derived signature morphism $\delta : \Sigma \rightarrow \Sigma'$. For the basic types, we take

$$\delta(real) = nat \rightarrow nat, \qquad \delta(bool) = nat.$$

Next, for each term constant $c : \sigma$ of Σ we need to give a term $\delta(c) : \delta(\sigma)$ in Σ'. For the constants 0 and 1, this can be done just by choosing one particular representing sequence for these real numbers, e.g.

$$\delta(0) = \lambda i{:}nat.\ 1, \qquad \delta(1) = \lambda i{:}nat.\ \mathit{ifzero}\ i\ 2\ 1.$$

For the booleans, we take $\delta(true) = 0$, $\delta(false) = 1$ and $\delta(\perp) = \perp$. It is also straightforward to write PCF programs *Minus*, *Plus*, *Times*, *Max*, *Less* for $\delta(-)$, $\delta(+)$, $\delta(*)$, $\delta(max)$ and $\delta(<)$ respectively. For example, we may take

$$\begin{aligned} \mathit{Minus} &= \lambda f : nat \rightarrow nat,\ i{:}nat. \\ &\quad \text{if } i = 0 \text{ then } f(1) \text{ else if } i = 1 \text{ then } f(0) \text{ else } 2 \frown f(i) \end{aligned}$$

where $\frown$ implements truncated subtraction. In any model $\mathcal{B}$, this satisfies the following condition (which should be understood as a meta-level assertion):

$$\begin{aligned} &\forall f \in [\![nat \rightarrow nat]\!]^{\mathcal{B}}.\ \text{IsReal}(f) \Rightarrow \\ &\qquad\qquad \text{IsReal}([\![\mathit{Minus}]\!]^{\mathcal{B}} f) \wedge \text{Val}([\![\mathit{Minus}]\!]^{\mathcal{B}} f) = -\text{Val}(f). \end{aligned}$$

Coding details for the other operations are given e.g. in [Plu98]. What is more surprising is that the operation $sup_{[0,1]}$ can be represented in PCF by a third-order function *Sup*, by means of a clever use of higher-type recursion (a detailed account of the algorithm with code is given in [Sim98]).

Proposition 5.1. *SP' is a constructive pre-logical refinement of SP via δ.*

Proof sketch. Starting from any $\mathcal{B} \in Mod(SP')$, we will obtain a model $\mathcal{A} \in Mod(SP)$ and a pre-logical relation $\mathcal{R}$.

The correct definition of $\mathcal{A}$ is slightly subtle — the whole point is that the obvious definition via a logical relation on $\mathcal{B}$ does not work (see below). First, we embed $\mathcal{B}$ in its chain completion *$\bar{\mathcal{B}}$ via an inclusion ι (we omit the definition). The main purpose of this step is to throw into the model all monotone functions of type $nat \to nat$ — this ensures that in $\bar{\mathcal{B}}$ we can represent all the classical reals and not just the computable ones (cf. Section 5.4 below). One can check that if $\mathcal{B}$ is a model of SP' then so is $\bar{\mathcal{B}}$. Next we define partial equivalence relations $\bar{E}^\sigma$ on $[\![\delta(\sigma)]\!]^{\bar{\mathcal{B}}}$ for each $\sigma \in Types(\Sigma)$. For the base cases, we take*

$$\bar{E}^{real}(f,g) \text{ iff } IsReal(f) \wedge IsReal(g) \wedge Val(f) = Val(g),$$
$$\bar{E}^{bool}(x,y) \text{ iff } x = y \wedge x \in \{[\![0]\!]^{\mathcal{B}}, [\![1]\!]^{\mathcal{B}}, [\![\bot]\!]^{\mathcal{B}}\}.$$

(The latter clause means that $\bar{E}$ behaves as a partial injection for observable types.) We lift this to higher types as a binary logical relation on $\bar{\mathcal{B}}$. One can show that for each constant c of Σ we have $\bar{E}(\iota[\![\delta(c)]\!]^{\mathcal{B}}, \iota[\![\delta(c)]\!]^{\mathcal{B}})$.

We now construct the required model $\mathcal{A}$ by taking $[\![\sigma]\!]^{\mathcal{A}}$ to be the set of equivalence classes of $\bar{E}^\sigma$; one can check that $\mathcal{A}$ yields a Henkin model for SP. Finally, we define relations R^σ from $[\![\sigma]\!]^{\mathcal{A}}$ to $[\![\delta(\sigma)]\!]^{\mathcal{B}}$ by $R^\sigma(a,b)$ iff $\iota(b) \in a$; clearly this defines a pre-logical relation $\mathcal{R}$ as required. □

5.4 Lack of a Constructive Logical Refinement

We now explain why SP' is not a constructive *logical* refinement of SP via δ. Intuitively, the idea is that a logical relation $\mathcal{R}$ is completely determined once we have fixed the relation at basic types — we have no freedom of choice for higher types. For certain models $\mathcal{B}$ of SP', this means that we are forced to include in the relation $R^{real\to real}$ some highly pathological elements of $[\![\delta(real \to real)]\!]^{\mathcal{B}}$, and our PCF implementation of $sup_{[0,1]}$ will fail to work for these pathological elements. This leads to a contradiction since we require $R([\![sup_{[0,1]}]\!]^{\mathcal{A}}, [\![Sup]\!]^{\mathcal{B}})$ for some model $\mathcal{A}$ of SP.

More precisely, let us take $\mathcal{B}$ to be some *effective* model of SP', such as the effective Scott domain model [Plo77] or the term model for PCF. All that we really require is that the elements of $[\![nat \to nat]\!]^{\mathcal{B}}$ correspond to just the partial recursive functions $\mathbb{N}_\bot \to \mathbb{N}_\bot$. We will show the following:

Theorem 5.2. *There is no model $\mathcal{A} \in Mod(SP)$ admitting a* logical *relation $\mathcal{R}$ over $\mathcal{A}$ and $\mathcal{B}|_\delta$ which is a partial injection on bool.*

Proof sketch. The proof of the theorem hinges on the existence of a pathological PCF implementation of the constant zero function: that is, a term Funny $: (nat \to nat) \to (nat \to nat)$ *such that* $R^{real\to real}([\![\lambda x{:}real.\,0]\!]^{\mathcal{A}}, [\![Funny]\!]^{\mathcal{B}})$*, but such that* $[\![Sup\ Funny]\!]^{\mathcal{B}} = [\![Bad]\!]^{\mathcal{B}}$*, where* $Bad = \lambda y{:}nat.\ if\ y < 2\ then\ 1\ else\ \bot$*. Since SP entails that* $sup_{[0,1]}(\lambda x{:}real.\,0) = 0$*, we have* $R^{real}([\![0]\!]^{\mathcal{A}}, [\![Bad]\!]^{\mathcal{B}})$*, which can be shown to be impossible.*

The idea behind Funny is based on the Kleene tree*, a well-known counterexample from recursion theory (see e.g. [Bee85]). Intuitively, Funny*(f) *gives 0 whenever* f *represents a* recursive *real number but diverges for certain non-recursive reals.* □

Notice how the pre-logical relation $\mathbb{R}$ in the proof of Proposition 5.1 avoids this problem: the interpretation of *Funny* in $\bar{\mathcal{B}}$ is not included in the partial equivalence relation $E^{real\to real}$, since the model contains representations of non-recursive reals on which *Funny* diverges.

The above example is robust in the sense that it is not just a feature of the particular implementation *Sup* we have chosen — it can be shown that there is *no* PCF program *Sup* that computes suprema for all relevant functions including *Funny*. Indeed, we believe that the above theorem should hold for all possible representations of the reals and all choices of the terms $\delta(c)$: the only condition on δ we require is that $\delta(real) = nat \to nat$.

6 Conclusion

The main purpose of this paper was to introduce the notion of constructive pre-logical refinement and explain how it relates to the usual account of data refinement for typed lambda calculus in terms of logical relations. In a nutshell, the relationship is that for data refinement logical relations work only because they are a special case of pre-logical relations, where the additional requirement imposed by logical relations is more of a hindrance than a help.

There are many directions in which this approach could be developed.

In Sect. 4 we considered linear chains of refinement steps. Decomposition of implementation tasks into separate subtasks can be modelled using constructions that take n-tuples of Henkin models as arguments, giving tree-shaped refinement diagrams. In particular, consider $\delta : \Sigma \to (\Sigma'_1 + \ldots + \Sigma'_n)$, where $\Sigma'_1 + \ldots + \Sigma'_n$ is a coproduct of the signatures $\Sigma'_1, \ldots, \Sigma'_n$. This induces the reduct $\cdot|_\delta : Mod(\Sigma'_1 + \ldots + \Sigma'_n) \to Mod(\Sigma)$. However, this does not give an n-ary construction, since $Mod(\Sigma'_1 + \ldots + \Sigma'_n)$ and $Mod(\Sigma'_1) \times \cdots \times Mod(\Sigma'_n)$ *do not coincide* even up to isomorphism; in other words, higher-order models do not amalgamate unambiguously. However, they *weakly amalgamate*: there is a standard (injective) construction that maps $Mod(\Sigma'_1) \times \cdots \times Mod(\Sigma'_n)$ into $Mod(\Sigma'_1 + \ldots + \Sigma'_n)$ (e.g. by taking full function spaces for extra "mixed" function types). Composing this with $\cdot|_\delta$, we obtain a function from $Mod(\Sigma'_1) \times \cdots \times Mod(\Sigma'_n)$ to $Mod(\Sigma)$ as required. This still ignores one important aspect of development, namely the possibility of mutual dependencies between subtasks. One solution, discussed thoroughly in [SST92], is to use specifications of parametric models in the development process; the same ideas should apply here, but the technical implications of using higher-order models are yet to be worked out.

This paper presents a *global* view of specifications and their refinement: constructions are required to work on the "whole system" (represented as a model of the implementing specification) and produce a whole system (represented as a model of the implemented specification). Good practice suggests that there should be a way to make the refinement steps "local" — that is, to use only *part* of the system built so far to implement some remaining parts of the requirements specification, and then add the result to the whole system built so far. Details will be provided in a longer version of the paper.

In this paper we have focused on $\lambda^{\rightarrow}$ only. But of course, less elementary type structures are also of great importance in software development using data refinement. One can consider inductive/coinductive datatypes, or more generally recursive types as in ML, or impredicative types as in Girard/Reynold's System F. For instance exact real numbers as in Sect. 5 are often implemented as streams for efficiency reasons, also in purely functional contexts, and abstract data types can be understood in the context of existential types. Notions of logical relations, appropriate for each of these type disciplines, have been proposed in the literature: see e.g. [Alt98] for inductive/coinductive types and [MM85] for System F. In order to accomodate data refinement involving such datatypes we need to introduce corresponding notions of pre-logical relation. As pointed out in [HS99], there is a standard methodology here: simply require the interpretations of the "relevant" constants in the two structures to be related. Despite its simplicity, this methodology is extremely rewarding, and it allows to harvest seredipitous results also in related areas. A case in point is offered by PER models of System F, where the extra latitude and flexibility given by defining the exponential PER pre-logically allows for a number of possibly novel natural model constructions. Finally, a notion of pre-logical relation for System F would raise the intriguing question of the relationship between this framework and the one in [Han99], where data refinements in the style of [ST88] are translated into System F using existential types.

Acknowledgements: Thanks to Samson Abramsky, Jo Hannay, Peter O'Hearn, Gordon Plotkin, John Power, John Reynolds and Bob Tennent for helpful discussion. This work has been partially supported by EPSRC grants GR/K63795 and GR/L89532, the ESPRIT-funded CoFI working group, the ESPRIT- and INCO-funded CRIT-2 project and MURST'97 and MURST'99 grants.

References

[AJM96] S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF. To appear in *Information and Computation* (1996).

[Alt98] T. Altenkirch. Logical relations and inductive/coinductive types. *Proc. Computer Science Logic, CSL'98*. Springer LNCS 1584, 343–354 (1998).

[Bee85] M. Beeson. *Foundations of Constructive Mathematics*. Springer (1985).

[Gog84] J. Goguen. Parameterized programming. *IEEE Trans. on Software Engineering* SE-10(5):528–543 (1984).

[Han99] J. Hannay. Specification refinement with System F. *Proc. Computer Science Logic, CSL'99*, Madrid. Springer LNCS 1683, 530–545 (1999).

[Hoa72] C.A.R. Hoare. Correctness of data representations. *Acta Informatica* 1:271–281 (1972).

[HS99] F. Honsell and D. Sannella. Pre-logical relations. *Proc. Computer Science Logic, CSL'99*, Madrid. Springer LNCS 1683, 546–561 (1999).

[KOPTT97] Y. Kinoshita, P. O'Hearn, J. Power, M. Takeyama and R. Tennent. An axiomatic approach to binary logical relations with applications to data refinement. *Proc. TACS'97.* Springer LNCS 1281, 191–212 (1997).

[Mil71] R. Milner. An algebraic definition of simulation between programs. *Proc. 2nd Intl. Joint Conf. on Artificial Intelligence.* British Computer Society, 481–489 (1971).

[Mit96] J. Mitchell. *Foundations for Programming Languages.* MIT Press (1996).

[MM85] J. Mitchell and A. Meyer. Second-order logical relations. *Proc. Logics of Programs*, Brooklyn. Springer LNCS 193, 225-236 (1997).

[Nor98] D. Normann. The continuous functionals of finite types over the reals. Technical Report 19, Dept. of Mathematics, University of Oslo (1998).

[Plo77] G. Plotkin. LCF considered as a programming language. *Theoretical Computer Science* 5:223–255 (1977).

[PPST00] G. Plotkin, J. Power, D. Sannella and R. Tennent. Lax logical relations. Submitted for publication (2000).

[Plu98] D. Plume. A calculator for exact real number computation. B.Sc. project report, Univ. of Edinburgh; available from `ftp://ftp.tardis.ed.ac.uk/users/dbp/report.ps.gz` (1998).

[Rey81] J. Reynolds. *The Craft of Programming.* Prentice Hall (1981).

[Rey83] J. Reynolds. Types, abstraction and parametric polymorphism. *Proc. 9th IFIP World Computer Congress*, Paris. North Holland, 513–523 (1983).

[SST92] D. Sannella, S. Sokołowski and A. Tarlecki. Toward formal development of programs from algebraic specifications: parameterisation revisited. *Acta Informatica* 29:689–736 (1992).

[ST88] D. Sannella and A. Tarlecki. Toward formal development of programs from algebraic specifications: implementations revisited. *Acta Informatica* 25:233–281 (1988).

[ST97] D. Sannella and A. Tarlecki. Essential concepts of algebraic specification and program development. *Formal Aspects of Computing* 9:229–269 (1997).

[Sch87] O. Schoett. Data Abstraction and the Correctness of Modular Programming. Ph.D. thesis, report CST-42-87, Dept. of Computer Science, Univ. of Edinburgh (1987).

[Sch90] O. Schoett. Behavioural correctness of data representations. *Science of Computer Programming* 14:43–57 (1990).

[Sim98] A.K. Simpson. Lazy functional algorithms for exact real functionals. *Proc. 23rd Intl. Symp. on Mathematical Foundations of Computer Science*, Brno. Springer LNCS 1450, 456–464 (1998).

[Ten94] R. Tennent. Correctness of data representations in Algol-like languages. In: *A Classical Mind: Essays in Honour of C.A.R. Hoare.* Prentice Hall (1994).