Formal Reasoning about Modules, Reuse and their Correctness*

Christoph Kreitz,[1] Kung-Kiu Lau[2] and Mario Ornaghi[3]

[1] Fachgebiet Intellektik, Fachbereich Informatik
Technische Hochschule Darmstadt
Alexanderstr. 10, D-64283 Darmstadt, Germany
kreitz@intellektik.informatik.th-darmstadt.de

[2] Department of Computer Science
University of Manchester, Manchester M13 9PL, UK
kung-kiu@cs.man.ac.uk

[3] Dipartimento di Scienze dell'Informazione
Universita' degli studi di Milano, Via Comelico 39/41, Milano, Italy
ornaghi@hermes. mc.dsi.unimi.it

Abstract. We present a formalisation of modules that are *correct*, and (*correctly*) *reusable* in the sense that composition of modules preserves both correctness and reusability. We also introduce a calculus for formally reasoning about the construction of such modules.

1 Introduction

Modular programming has been around for a long time, and has more recently evolved into object-oriented programming (e.g. [11]). Various forms of *modules* and *objects* can be found in a variety of modern programming languages. They are important because they facilitate structured design as well as code *reuse*. However, for formal program development, i.e. developing programs that are *formally correct* wrt their (formal) specifications, current modular and object-oriented programming languages lack a suitable formal semantics in our view, even though some of them do have type-system based rules for program composition (see e.g. [3, 13]).

In this paper, we define modules as first-order theories (with isoinitial semantics) that contain logic programs. We express reuse in model-theoretic terms. Our formalisation provides a logical basis for formal reasoning about modules, reuse, and their *correctness*; it is therefore suitable for formal program development in an object-oriented fashion. We shall use the logic programming paradigm for simplicity, but our approach is very general, and should be applicable to any programming paradigm with suitable logical semantics.

The paper is organised as follows. In Section 2, we introduce the background theory together with our notation and terminology. In Section 3, we briefly define our notion of a module, its correctness and reusability. Then in Section 4, which is the bulk of the paper, we show rules to construct modules in such a way that correctness and reusability are both preserved. The rules are based on the notion of dependency type, and refine and expand our first results in earlier sections.

* This work was done during the second author's visit to TH Darmstadt, supported by the European Union HCM project LPST, contract no. 93/414. He wishes to thank Prof Wolfgang Bibel for his invitation and hospitality.

2 Background

2.1 Many-sorted First-order Languages and Interpretations

We will denote a many-sorted *signature* by $\Sigma = \langle S, F, R\rangle$, where S is a set of *sort* symbols, F is a set of *function* declarations and R is a set of *relation* declarations. Signatures will be introduced as shown in the following example.

Example 1. The signature of stacks is defined thus:

$$\begin{array}{ll} \text{SORTS:} & Elem, Stacks; \\ \text{FUNCTIONS:} & empty : \to Stacks; \\ & \;push\; : (Elem, Stacks) \to Stacks; \\ \text{RELATIONS:} & = : (Elem, Elem); \\ & = : (Stacks, Stacks); \end{array}$$

Each function declaration has the form $f : a \to s$, where f is the declared function symbol, $a = (s_{i_1}, \ldots, s_{i_n})$ is its arity, and s is its sort, and each relation declaration has the form $r : a$, where r is the declared relation symbol, and $a = (s_{k_1}, \ldots, s_{k_m})$ is its arity. Constants will be functions with empty arity, i.e. $c : \to s$. We allow *overloaded* relation symbols, i.e. symbols that have many declarations, like overloaded identity $= : (Elem, Elem)$, $= : (Stacks, Stacks)$ in the example.

A Σ*-interpretation* $\mathcal{I}$ interprets the symbols of the signature as follows:

- Every sort symbol s is interpreted as a domain $s^{\mathcal{I}}$.
- Every function declaration $f : a \to s$ is interpreted as a function $f^{\mathcal{I}} : a^{\mathcal{I}} \to s^{\mathcal{I}}$, where, if $a = (s_1, \ldots, s_n)$, then $a^{\mathcal{I}}$ is $s_1^{\mathcal{I}} \times \cdots \times s_n^{\mathcal{I}}$.
- Every relation declaration $r : a$ is interpreted as a relation $r^{\mathcal{I}} \subseteq a^{\mathcal{I}}$.

The first-order language L_Σ generated by a signature Σ (and by some set of sorted variables) is defined in the usual way, as are terms and formulas. We will write $\tau : s$ to indicate that the sort of τ is s. Formulas of L_Σ will also be called Σ-formulas.

An *assignment* $\mathbf{a}$ over an interpretation $\mathcal{I}$ is a map that associates with every variable $x : s$ an element $\mathbf{a}(x)$ of $s^{\mathcal{I}}$. In a given $\mathcal{I}$ and $\mathbf{a}$, we have that:

- Every term $\tau : s$ denotes an element of $s^{\mathcal{I}}$, that we will call the value of τ and denote by $val_{\mathcal{I}}(\tau, \mathbf{a})$. Since the value of a ground term τ does not depend on $\mathbf{a}$ it will be denoted by $val_{\mathcal{I}}(\tau)$.
- Every formula evaluates to *true* or *false*. We will write $\mathcal{I} \models_{\mathbf{a}} F$ to indicate that the formula F evaluates to *true* (in $(\mathcal{I},\mathbf{a})$). If F is closed, we will simply write $\mathcal{I} \models F$.

A Σ-homomorphism $h : \mathcal{I}_1 \to \mathcal{I}_2$, for Σ-interpretations $\mathcal{I}_1$ and $\mathcal{I}_2$, is an S-indexed family of maps $h_s : s^{\mathcal{I}_1} \to s^{\mathcal{I}_2}$ that preserve functions and relations, i.e. $h(f^{\mathcal{I}_1}(\alpha)) = f^{\mathcal{I}_2}(h(\alpha))$[4] and $\alpha \in r^{\mathcal{I}_1} \Rightarrow h(\alpha) \in r^{\mathcal{I}_2}$. Starting from (many-sorted) homomorphisms, isomorphism and isomorphic embeddings can be defined in the usual way. To indicate that two Σ-interpretations $\mathcal{I}_1$ and $\mathcal{I}_2$ are isomorphic, we will write $\mathcal{I}_1 \sim \mathcal{I}_2$.

---

[4] We omit the subscript s in h_s. Moreover, if $\alpha = \alpha_1, \ldots, \alpha_n$, then $h(\alpha_1, \ldots, \alpha_n)$ means $h(\alpha_1), \ldots, h(\alpha_n)$.

2.2 Theories and Logic Programs

A Σ-theory $\mathcal{T}$ is a set of axioms. As usual, a model $\mathcal{M}$ of a theory $\mathcal{T}$ is an interpretation such that $\mathcal{I} \models \mathcal{T}$. We say that $\mathcal{M}$ is *reachable* iff, for every sort s and $\alpha \in s^{\mathcal{M}}$, there is a ground term τ of sort s such that $\alpha = val_{\mathcal{M}}(\tau)$. $\mathcal{M}$ is an *isoinitial model* of $\mathcal{T}$ iff, for every other model $\mathcal{N}$ of $\mathcal{T}$, there is a unique isomorphic embedding $\mu : \mathcal{M} \rightarrow \mathcal{N}$.

For theories with reachable models, the following very useful theorem holds:

Theorem 1. *Let $\mathcal{T}$ be a theory that has a reachable model $\mathcal{M}$. Then $\mathcal{M}$ is an isoinitial model of $\mathcal{T}$ iff, for every closed atomic formula A:*

$$\mathcal{T} \vdash A \quad \text{or} \quad \mathcal{T} \vdash \neg A \tag{1}$$

A theory $\mathcal{T}$ is *atomically complete* if it satisfies (1). For such a theory any two reachable models are isomorphic, since they are both isoinitial.[5]

We will often use theories related to logic programs (see e.g. [10] for a general introduction to logic programming). We say that a predicate symbol is *defined* by a logic program P, if it occurs in the head of at least one clause of P. If a predicate is not defined by P, it is said to be *open* (in P).

We associate with a program P the theory $\mathcal{T}(P) = \mathit{free}(P) \cup \mathit{Ocomp}(P)$, where $\mathit{free}(P)$ is the set of freeness axioms for the constant and function symbols of P; and $\mathit{Ocomp}(P)$, the *open completion* of P, is the set of completed definitions of the predicates defined by P.[6]

2.3 Signature and Theory Morphisms

First-order theories are an institution [6]. Here we briefly introduce relevant properties of an institution, and some terminology.

A signature $\Sigma = \langle S, F, R \rangle$ is a *subsignature* of $\Delta = \langle S', F', R' \rangle$, written $\Sigma \preceq \Delta$, iff $S \subseteq S'$, $F \subseteq F'$ and $R \subseteq R'$.

Let Σ be a signature and ρ be a function that maps the symbols of Σ into another set of symbols. Then ρ *generates* $\rho(\Sigma) = \langle \rho(S), \rho(F), \rho(R) \rangle$, where $\rho(S)$ is the image of S, $\rho(F)$ contains the declarations $\rho(f) : \rho(a) \rightarrow \rho(s)$ such that $f : a \rightarrow s \in F$, and $\rho(R)$ the declarations $\rho(r) : \rho(a)$ such that $r : a \in R$.

If ρ is a map from the symbols of Σ to those of another signature Δ, and if $\rho(\Sigma) \preceq \Delta$, then we say that ρ is a *signature morphism* from Σ to Δ. If it is an injective map, then ρ is called a *signature extension*. If bijective, it is called a *renaming*.

Let $\rho : \Sigma \rightarrow \Delta$ be a signature morphism. The interpretations $int(\Delta)$ of Δ are related to the interpretations $int(\Sigma)$ of Σ by the *reduct* operation $|\rho : int(\Delta) \rightarrow int(\Sigma)$ defined thus:

$$\begin{aligned} s^{\mathcal{I}|\rho} & = \rho(s)^{\mathcal{I}} \\ f^{\mathcal{I}|\rho} : a \rightarrow s & = \rho(f)^{\mathcal{I}} : \rho(a) \rightarrow \rho(s) \\ r^{\mathcal{I}|\rho} : a & = \rho(r)^{\mathcal{I}} : \rho(a) \end{aligned}$$

$\mathcal{I}|\rho$ will be called the ρ-*reduct* of $\mathcal{I}$.

---

[5] [1] gives a proof of Theorem 1. Isoinitial semantics is closely related to initial semantics used in algebraic ADTs [14] (see also [7, 12]).

[6] We can also use induction principles in $\mathcal{T}(P)$.

If $\rho(\sigma)=\sigma$ for every symbol σ, i.e. $\Sigma \preceq \Delta$, we get the usual notion of reduct to a subsignature (see e.g. [5]). In this case, the reduct will also be denoted by $\mathcal{I}|\Sigma$.

It is immediate to extend a morphism $\rho : \Sigma \rightarrow \Delta$ to a map $\rho : L_\Sigma \rightarrow L_\Delta$ and the following satisfaction condition (see [6]) can be easily proved:

Theorem 2. *For every closed Σ-formula F and every Δ-interpretation $\mathcal{I}$:*

$$\mathcal{I} \models \rho(F) \Leftrightarrow \mathcal{I}|\rho \models F \tag{2}$$

Finally, given a signature morphism $\rho : \Sigma \rightarrow \Delta$ and a theory $\mathcal{T} \subseteq L_\Sigma$, we denote the *$\rho$-image* of $\mathcal{T}$ by $\rho(\mathcal{T})$.

3 Modules

Our notion of a module is that it consists of a (first-order) theory $\mathcal{F}$ which axiomatises a problem domain, together with logic programs prog_1, prog_2, $\ldots$, that can be specified and derived in $\mathcal{F}$. It can be visualised as follows:

Framework:- $\mathcal{F} : \Delta \Leftarrow \Pi$

Program:- $\mathrm{prog}_1 : \delta_1 \Leftarrow \pi_1$	Program:- $\mathrm{prog}_2 : \delta_2 \Leftarrow \pi_2$	$\ldots$
Program 1	Program 2	

We call $\mathcal{F}$ a *framework*. It contains axiomatisations of abstract data types (ADTs), *any* relevant axioms for reasoning about these ADTs (e.g. induction axioms), as well as other axioms for reasoning about the domain (e.g. axioms in a library framework stating that each book has a title, a list of authors, and so on). In particular $\mathcal{F}$ axiomatises function and relation symbols Δ.

$\mathcal{F}$ may be *open*, i.e. it may have *parameters* Π. In this case, $\mathcal{F}$ may axiomatise the symbols in Δ in terms of Π, and we write $\mathcal{F} : \Delta \Leftarrow \Pi$, where $\Delta \Leftarrow \Pi$ is called $\mathcal{F}$'s *dependency type*.

Similarly, a program prog_i in $\mathcal{F}$ may be open, and computes relations in δ in terms of its parameters π, where δ are the defined relations of P and π are its open relations (see Section 2). In this case, we write $\mathrm{prog}_i : \delta \Leftarrow \pi$, and $\delta \Leftarrow \pi$ is also called prog_i's dependency type.

A module M therefore contains two kinds of parameters: *framework parameters*, viz. Π, and *program parameters*, viz. π. This means that the issues of *reuse* and *correctness* occur at two levels. M can be reused with different instantiations of both Π and π, i.e. M can be composed with other modules that (partially or completely) define Π and π in different ways.

The correctness of a module M refers to the consistency of its framework $\mathcal{F}$, as well as the correctness of each program prog_i wrt its specification in $\mathcal{F}$. Moreover, when M is composed with another module, the composition must preserve both the consistency of $\mathcal{F}$ and the correctness of all the prog_i's. By contrast, current object-oriented programming lacks a formalisation of reuse and correctness.

Our notion of a module is in spirit similar to that of a *class* in object-oriented programming, which is an ADT implementation together with procedures for the ADT. Indeed we can also have an *object* by endowing a module with an internal state.

3.1 Frameworks

A framework is a first-order theory that axiomatises an ADT, or a problem domain, or both. A framework is *closed* iff it axiomatises an isoinitial model (that is unique up to isomorphism). Otherwise, it is *open*. An open framework axiomatises a class of isoinitial models.

Example 2. The following is an example of a *closed* framework:

Framework $\mathcal{K}_{a,b}$;
SORTS: $Elab$;
FUNCTIONS: $a, b :\rightarrow Elab$;
D-AXIOMS: $\neg a = b$;

The interpretation $\mathcal{I}$ such that $Elab^{\mathcal{I}} = \{a, b\}$ is the isoinitial model of $\mathcal{K}_{a,b}$.

Example 3. As an example of an *open* framework, the theory of stacks can be axiomatised as follows.

Framework $\mathcal{STACK}(Elem)$;
SORTS: $Elem, Stacks$;
FUNCTIONS: $empty : \rightarrow Stacks$;
$push \; : (Elem, Stacks) \rightarrow Stacks$;
D-AXIOMS: $\neg empty = push(x, S)$;
$push(x_1, S_1) = push(x_2, S_2) \rightarrow x_1 = x_2 \wedge S_1 = S_2$;
P-AXIOMS: $\exists x, y\, (\neg x = y)$;

where we have omitted the universal closure of the axioms, upper-case variables are of sort *Stacks*, and lower-case variables are of sort *Elem*.

We have a parameter *Elem* and two kinds of axioms. The p-axioms, that state some assumptions on the parameter *Elem* (in this case we require that *Elem* contains at least two distinct elements). The d-axioms completely characterise the non-parametric symbols in terms of the parameter. This means that via *Elem* the framework $\mathcal{STACK}$ axiomatises a class of isoinitial models. That is, whenever we choose a particular set of elements (fixed by a closed framework), we get an axiomatisation of stacks with the chosen elements. For example, if we instantiate *Elem* by the sort *Elab* of $\mathcal{K}_{a,b}$, we get a closed (reachable and atomically complete) framework that has the isoinitial model $\mathcal{I}$ such that $Elem^{\mathcal{I}}$ is $\{a, b\}$, $Stacks^{\mathcal{I}}$ is the set of stacks with elements from $\{a, b\}$, and $push^{\mathcal{I}}$ is the usual push operation on stacks.

A framework also contains a list of *theorems*. For example, in $\mathcal{STACK}(Elem)$ we can prove $\forall S \exists x, y(\neg push(x, S) = push(y, S))$, and add it as a theorem to the framework.

In the sequel, a framework will be indicated by $\mathcal{F}(\Pi) = \langle \Sigma, \mathcal{T}, \mathcal{TH}, \Pi \rangle$, where Π is the set of parameters, Σ the signature, $\mathcal{T}$ the theory (a set of axioms), and $\mathcal{TH}$ the current set of proved theorems. We will write $\mathcal{F}$ if no confusion arises.

In a closed framework, Π is empty. In an open one, any sort symbol s or relation declaration $r : a$ can be used as a parameter. Identity $=: (s, s)$ is a parameter iff s is a parameter too. The set of p-axioms is simply the restriction $\mathcal{T}|\Pi$, namely the set of axioms that contain *only* symbols of Π. The other axioms are the d-axioms. Note that in closed frameworks there are no p-axioms.

3.2 Programs

In a module M, the logic programs prog_i are not just ordinary programs. They are open programs that must be correct in all possible instances of M. We call this kind of correctness *steadfastness*, and a model-theoretic formalisation can be found in [9]. Here we give a brief explanation so as to make clear what kind of programs we can find in a module.

If $\mathcal{F}$ in M is a closed framework, then a closed program $P : \delta$ (that is, a program without open relations) is correct in M iff its minimum Herbrand model is isomorphic to the intended model of $\mathcal{F}$ restricted to the signature of P. Steadfastness of an open program can be defined as follows:

Definition 3. A program $P : \delta \Leftarrow \pi$ is *steadfast* in a closed framework $\mathcal{C}$ iff, for every closed program $Q : \delta_1$ that is correct in $\mathcal{C}$, if $\pi \subseteq \delta_1$ and $\delta_1 \cap \delta = \emptyset$ then $P \cup Q : \delta \cup \delta_1$ is correct in $\mathcal{C}$.

A program $P : \delta \Leftarrow \pi$ is *steadfast* in an open framework $\mathcal{F}$ iff it is steadfast in every closed instance $\mathcal{F} \cup \mathcal{C}$, where $\mathcal{C}$ is a closed framework.

This means that $P : \delta \Leftarrow \pi$ is (closed and) correct wrt to its specification whenever $\mathcal{F} : \Delta \Leftarrow \Pi$ becomes closed, and correct closed programs for computing π are supplied. The programs for computing the framework parameters in π can be supplied only after the instantiation $\mathcal{F} \cup \mathcal{C}$ by a closed framework $\mathcal{C}$, while those for the rest of π can be supplied either before or after such an instantiation.

Example 4. In $\mathcal{LIST} : \Delta \Leftarrow Elem, \lhd$, the framework of lists which axiomatises the usual list relations Δ, e.g. $ord, perm, \ldots$, with parametric element type $Elem$ and ordering relation $\lhd$, sorting can be defined in the usual way:[7]

$$sort(x, y) \leftrightarrow perm(x, y) \wedge ord(y) \tag{3}$$

From (3) we can synthesise the following open program for *sort*:[8]

Framework:- $\mathcal{LIST} : \Delta \Leftarrow Elem, \lhd$

Program:- merge sort : $sort \Leftarrow merge, split$

$$\begin{array}{rl} sort(nil, nil) \leftarrow & \\ sort(x.nil, x.nil) \leftarrow & \\ sort(x.y.A, W) \leftarrow & split(x.y.A, I, J) \wedge sort(I, U) \wedge sort(J, V) \wedge \\ & merge(U, V, W) \end{array} \tag{4}$$

The meaning of the program parameters *split* and *merge* can be partially specified. For example, to avoid an overdetermined specification, $merge(U, V, W)$ may be required to give an ordered W if U and V are ordered, whilst its meaning may be left open for unordered U and V. Such partially specified relations are called *internal parameters*. They are used only in program derivation, but not as parameters of the framework. By a suitable partial characterisation of *split* and *merge*, we can prove that the program (4) is steadfast wrt the specification (3), that is, it is correct for every interpretation of the internal parameters that

[7] This definition is adequate, see Section 4.2 later.

[8] The synthesis of steadfast programs is discussed in [8].

satisfies their partial definitions, as well as in every closed instance of $\mathcal{LIST}$. Thus it can be composed with different programs for *merge* and *split* to yield different sorting algorithms based on merging, e.g. insertion sort, and with every program for deciding the framework parameter $\triangleleft$, in any possible instance of the parameters *Elem* and $\triangleleft$.
Thus a (steadfast) program prog_i is not only correct wrt to its specification within its parent module M, but it can also be reused correctly in different instances of M, i.e. in different instances of the framework $\mathcal{F}$ in M.

Finally, it is important to note that the correctness of prog_i in M is defined in model-theoretic terms ([9]), by comparing the (minimum Herbrand) model of prog_i with the (isoinitial) model of $\mathcal{F}$.

4 Module Construction

To construct a module M, we need to construct the framework $\mathcal{F}$ and the programs prog_i that it contains. We want the programs developed in M to be *correct* and (*correctly*) *reusable*, that is to remain correct in larger modules built up by composition with M. To achieve this, we must construct $\mathcal{F}$ and prog_i in special ways: prog_i must be correct in M, and the framework $\mathcal{F}$ must be *adequate*, i.e. it must behave in a sound way with respect to composition.

Note that we have two levels of rules. The first level contains the rules for program synthesis, that build up programs that are formally correct wrt a framework, where correctness has a precise formal definition. The second level contains rules to build up frameworks, in order to build a formalisation that intuitively meets some informal problem specification. For this second level, we can only have informal correctness: the intended model that we incrementally construct intuitively agrees with the problem domain and the specification.

In the preceding section, we have briefly discussed the construction of correct and reusable prog_i. In the rest of the paper, we will concentrate on the construction of such a framework $\mathcal{F}$.

4.1 Operations on Frameworks

First we introduce basic operations on frameworks that will be employed in framework construction.
Renaming Let $\mathcal{F} = \langle \Sigma, \mathcal{T}, \mathcal{TH}, \Pi \rangle$ be a framework. Let ρ be a bijective map from the symbols of Σ to another set of symbols. We define $\rho(\mathcal{F}) = \langle \rho(\Sigma), \rho(\mathcal{T}), \rho(\mathcal{TH}), \rho(\Pi) \rangle$. We will introduce renaming by the syntax shown in the following example:

Framework $\mathcal{LIST}_0(Elem)$;
RENAMES $\mathcal{STACK}_0(Elem)$;
RENAMING: $Stacks \mapsto List$;
$empty \mapsto nil$;
$push \mapsto \cdot$;

Composition Let $\mathcal{F}(\Pi) = \langle \Sigma, \mathcal{T}, \mathcal{TH}, \Pi \rangle$ be an open framework. We say that a sort symbol is used in the parameters Π iff it belongs to Π or if it occurs in some declaration of Π, whilst a declaration is used iff it belongs to Π.

The composition of $\mathcal{F}(\Pi)$ with $\mathcal{G}(\Pi_1) = \langle \Sigma_1, \mathcal{T}_1, \mathcal{TH}_1, \Pi_1 \rangle$ is performed through a Π-renaming ρ defined thus: $\rho(\Sigma)$ is a *Π-renaming for Σ* and Σ_1 if for every σ used in Π, $\rho(\sigma)$ belongs to Σ_1, whilst the sort symbols and declarations not used in Π are mapped into names that do not occur in Σ_1. For a Π-renaming ρ, we can build ρ-amalgamation of the signatures

$$\rho(\Sigma) + \Sigma_1 = \langle \rho(S) \cup S_1, \rho(F) \cup F_1, \rho(R) \cup R_1 \rangle$$

where $\Sigma = \langle S, R, F \rangle$ and $\Sigma_1 = \langle S_1, R_1, F_1 \rangle$.

Now, let ρ be a Π-renaming. The composition operation $\mathcal{F}[\rho, \mathcal{F}_1]$ is defined if the following proof obligation is satisfied:

$$\rho(\mathcal{T}|\Pi) \subseteq \mathcal{T}_1 \cup \mathcal{TH}_1 \tag{5}$$

That is, we require that the p-axioms become theorems or axioms of $\mathcal{F}_1$.

If $\mathcal{F}[\rho, \mathcal{F}_1]$ is defined, the resulting composite $\mathcal{F}[\rho, \mathcal{F}_1](\Pi_c) = \langle \Sigma_c, \mathcal{T}_c, \mathcal{TH}_c, \Pi_c \rangle$ is defined as follows:

- $\Pi_c = \Pi_1$, i.e. the parameters of the composite are the (possible) ones of $\mathcal{F}_1$.
- $\Sigma_c = \rho(\Sigma) \cup \Sigma_1$, i.e. the new signature enriches Σ_1 by $\rho(\Sigma)$.
- $\mathcal{T}_c = \mathcal{T}_1 \cup (\rho(\mathcal{T}) - \mathcal{TH}_1)$, i.e. we consider the union of the axioms, and we eliminate possible axioms of $\rho(\mathcal{T})$ that are theorems of $\mathcal{F}_1$.[9]
- $\mathcal{TH}_c = \mathcal{TH}_1 \cup (\rho(\mathcal{TH}) - \mathcal{T}_1)$

If ρ is the identity, we indicate the corresponding composition by $\mathcal{F}[\mathcal{F}_1]$.

Example 5. $\mathcal{STACK}(Elem)$ can be composed with $\mathcal{K}_{a,b}$ by the renaming $Elem \mapsto Elab$ ($\rho(\sigma) = \sigma$ for the other symbols). We will use the syntax:

Framework $\mathcal{STACK}_{a,b}$;
COMPOSES: $\mathcal{STACK}(Elem)[\mathcal{K}_{a,b}]$
WITH: $Elem \mapsto Elab$;
OBLIGATION: $\exists x, y\, (\neg x = y)$;

The composition is defined. Indeed, the proof obligation $\exists x, y\, (\neg x = y)$ immediately follows from the axiom $\neg a = b$ of $\mathcal{K}_{a,b}$.

Composition is associative, i.e.

Theorem 4. *If $(\mathcal{F}_0[\rho_1, \mathcal{F}_1])[\rho_2, \mathcal{F}_2]$ is defined, then $\mathcal{F}_0[\rho_2\rho_1, \mathcal{F}_1[\rho_2, \mathcal{F}_2]]$ is defined and*

$$(\mathcal{F}_0[\rho_1, \mathcal{F}_1])[\rho_2, \mathcal{F}_2] = \mathcal{F}_0[\rho_2\rho_1, \mathcal{F}_1[\rho_2, \mathcal{F}_2]]$$

where $\rho_2\rho_1$ is the composition of the two renamings.

Proof. The proof follows from the definition of composition. □

Moreover, it is related to framework extension.

Definition 5. Let $\mathcal{F}(\Pi) = \langle \Sigma, \mathcal{T}, \mathcal{TH}, \Pi \rangle$ and $\mathcal{G}(\Pi') = \langle \Sigma', \mathcal{T}', \mathcal{TH}', \Pi' \rangle$ be two frameworks, and $\rho : \Sigma \to \Sigma'$ be a signature extension. $\mathcal{G}(\Pi')$ is a *ρ-extension* of $\mathcal{F}$, written $\mathcal{F}(\Pi) \preceq_\rho \mathcal{G}(\Pi')$,[10] iff

- $\rho(\mathcal{T}) \subseteq \mathcal{T}' \cup \mathcal{TH}'$;
- for every σ not used in Π, $\rho(\sigma)$ does not occur in Π'.

We can easily prove the following theorem:

[9] This does not mean that axioms are independent, but simply that they remain disjoint from theorems, to avoid redundancies.

[10] If ρ is the identity, we write $\mathcal{F}(\Pi) \preceq \mathcal{G}(\Pi')$.

Theorem 6. *Let $\mathcal{F}(\Pi)$ and $\mathcal{F}_1(\Pi_1)$ be two frameworks. If $\mathcal{F}[\rho, \mathcal{F}_1]$ is defined, then $\mathcal{F}(\Pi) \preceq_\rho \mathcal{F}[\rho, \mathcal{F}_1](\Pi_1)$ and $\mathcal{F}_1(\Pi_1) \preceq \mathcal{F}[\rho, \mathcal{F}_1](\Pi_1)$.*

These theorems will be used to introduce sound *extension rules* as compositions of a particular kind.

Theory morphism, as defined in [6], can be seen as ρ-compositions, where ρ is any signature morphism (i. e. it may be non-injective). Moreover, our definition of composition essentially coincides with that of [6]. However, the theory of institutions is not useful for our purposes, since it does not deal with the problem of preserving consistency.[11]

4.2 Constructing Adequate Frameworks

As we mentioned in Section 3, correctness of a framework $\mathcal{F}$ in a module M refers to $\mathcal{F}$'s consistency. Thus to ensure that M is correct and (correctly) reusable, we need to have rules for framework construction that preserve consistency. Moreover, as we indicated in Section 3.2, in order to keep steadfastness meaningful for programs, we also have to preserve the isoinitial model of $\mathcal{F}$. To achieve both of these targets, we introduce the notion of *adequate frameworks*, and use framework composition as the main operation to construct frameworks in such a way that adequacy is preserved. We shall first define adequacy, and then give a set of simple adequate frameworks that can be used as basic units to build composite adequate frameworks incrementally.

Definition 7. A closed framework is *adequate* iff it has a reachable isoinitial model. (In the sequel, we will simply say 'closed framework' instead of 'adequate closed framework'.)

Definition 8. $\mathcal{G}$ is an *adequate closed ρ-extension* of a closed framework $\mathcal{C}$, written $\mathcal{C} \sqsubseteq_\rho \mathcal{G}$,[12] iff $\mathcal{G}$ has a reachable isoinitial model $\mathcal{I}$,[13] i.e. it is closed, and the reduct $\mathcal{I}|\rho$ is an isoinitial model of $\mathcal{C}$.

Definition 9. An open framework $\mathcal{F}(\Pi)$ is *adequate* iff, for every closed framework $\mathcal{C}$, if $\mathcal{F}[\rho, \mathcal{C}]$ is defined, then $\mathcal{C} \sqsubseteq \mathcal{F}[\rho, \mathcal{C}]$.

Definition 10. $\mathcal{G}(\Pi')$ is an *adequate ρ-extension* of an adequate open framework $\mathcal{F}(\Pi)$, written $\mathcal{F}(\Pi) \sqsubseteq_\rho \mathcal{G}(\Pi')$, iff it is adequate, $\mathcal{F}(\Pi) \preceq \mathcal{G}(\Pi')$ and, for every closed framework $\mathcal{C}$, if $\mathcal{G}[\rho_1, \mathcal{C}]$ is defined, then $\mathcal{F}[\rho_1\rho, \mathcal{C}]$ is defined and $\mathcal{F}[\rho_1\rho, \mathcal{C}] \sqsubseteq \mathcal{G}[\rho_1, \mathcal{C}]$.

It is easy to see that renaming does not change any feature of an adequate framework. Furthermore, we have the following theorem about framework composition:

Theorem 11. *If $\mathcal{F}(\Pi)$ and $\mathcal{G}(\Pi')$ are adequate, and $\mathcal{F}[\rho, \mathcal{G}]$ is defined, then $\mathcal{F}[\rho, \mathcal{G}]$ is adequate.*

Proof. Let $\mathcal{F}[\rho, \mathcal{G}][\rho_1, \mathcal{C}]$ be defined. Since $\mathcal{F}$ and $\mathcal{G}$ are adequate, $\mathcal{C} \sqsubseteq \mathcal{G}[\rho_1, \mathcal{C}] \sqsubseteq \underline{\mathcal{F}[\rho_1\rho, \mathcal{G}[\rho_1, \mathcal{C}]}] = \mathcal{F}[\rho, \mathcal{G}][\rho_1, \mathcal{C}]$. Thus $\mathcal{F}[\rho, \mathcal{G}](\Pi')$ is adequate. □

[11] Simple forms of axioms, like equational or strict Horn axioms, are guaranteed to be consistent.

[12] If ρ is the identity, we write $\sqsubseteq$ instead of $\sqsubseteq_\rho$.

[13] Note that if $\mathcal{C} \sqsubseteq_\rho \mathcal{G}$, then $\mathcal{G}$ is consistent by definition, since it has a model.

Moreover we can prove that, if $\mathcal{F}(\Pi)$ and $\mathcal{G}(\Pi')$ are adequate, $\mathcal{F}[\rho, \mathcal{G}]$ does not introduce new constant or function symbols with sorts in $\mathcal{G}$, and $\mathcal{F}[\rho, \mathcal{G}]$ is defined, then $\mathcal{G}(\Pi') \sqsubseteq \mathcal{F}[\rho, \mathcal{G}](\Pi')$. This and Theorem 11 show that framework composition behaves in a sound way. It provides the basis for our strategy to start with small, simple, but adequate closed and open frameworks, and then use composition to construct larger, adequate frameworks incrementally.

The first example of a basic framework is *free*(K), where K is a set of constant and function symbols. K may be open or closed, depending on whether it contains open sort symbols or not. A sort s is open (wrt K) iff no declaration of the form $f : a \to s$ belongs to K. We have the following theorem on the adequacy of K.

Theorem 12. *If K is closed, then free(K) is an adequate closed framework, and the structure freely generated by K is an isoinitial model of free(K).*

If K contains open sorts $s_1, \ldots, s_n$, then free$(K)(s_1, \ldots, s_n)$ is an adequate open framework.

Proof. If K is closed, we get our result by Theorem 1, since *free*(K) is atomically complete and the structure freely generated by K is a reachable model of it.

If K is open, then let $t(u), t'(v)$ be terms with variables u, v of open sorts, *free*$(K)[\mathcal{C}]$ be a closed instance, and $t(\alpha) = t'(\beta)$ be ground. Since $\mathcal{C} \vdash \alpha = \beta$ or $\mathcal{C} \vdash \neg\alpha = \beta$, we can prove that *free*$(K)[\mathcal{C}] \vdash t(\alpha) = t'(\beta)$, or *free*$(K)[\mathcal{C}] \vdash \neg t(\alpha) = t'(\beta)$. Moreover *free*$(K)[\mathcal{C}]$ is reachable. Then, by Theorem 1, it is closed. □

The second kind of frameworks are explicit definitions.

Theorem 13. *Let Σ be a signature containing a relation declaration $r : (a, s)$, a function declaration $f : a \to s$, and the sort symbols of a, s. Then*

$$\mathcal{D}_{r(x,f(x))} = \langle \Sigma, \{\forall x\, r(x, f(x)), \forall x \exists! y\, r(x, y)\}, \emptyset, \{r : (a, s), a, s\} \rangle$$

is an adequate open framework.

Let Σ be the signature of a definition axiom $r(x) \leftrightarrow F(x)$, where $F(x)$ is a quantifier-free formula that does not contain r, and let Π be the signature of $F(x)$ (i.e. Π does not contain r). Then

$$\mathcal{D}_{r(x) \leftrightarrow F(x)}(\Pi) = \langle \Sigma, \{\forall x\, (r(x) \leftrightarrow F(x))\}, \emptyset, \Pi \rangle$$

is an adequate open framework.

Proof. Suppose $\mathcal{D}_{r(x,f(x))}[\mathcal{C}]$ is defined, where $\mathcal{C}$ is closed. Then $\mathcal{C}$ proves the p-axiom $\forall x \exists! y\, r(x, y)$. Hence the explicit definition $\forall x\, r(x, f(x))$ determines an expansion $\mathcal{I}_f$ of the isoinitial model $\mathcal{I}$ of $\mathcal{C}$. Since $r(x, f(x))$ is atomic, we can prove that $\mathcal{I}_f$ is an isoinitial model of $\mathcal{D}_{r(x,f(x))}[\mathcal{C}]$.

Suppose $\mathcal{D}_{r(x) \leftrightarrow F(x)}[\mathcal{C}]$ is defined. Since $\mathcal{D}_{r(x) \leftrightarrow F(x)}[\mathcal{C}]$ contains only the new axiom $\forall x (r(x) \leftrightarrow F(x))$, it is reachable and, since $F(x)$ is quantifier-free, it is atomically complete. Hence it has an isoinitial model $\mathcal{I}$. The reduct of $\mathcal{I}$ to the language of $\mathcal{C}$ is reachable, hence it is an isoinitial model of $\mathcal{C}$. □

The third basic framework is the theory $\mathcal{T}(P)$ of a program P. Note that P is open if it contains open sorts. For closed programs we have the following theorem.

Theorem 14. *If P is a definite closed program, and it terminates for every ground atomic goal $\leftarrow A$, then the theory $\mathcal{T}(P)$ is an adequate closed framework, and the minimum model of P is an isoinitial model of $\mathcal{T}(P)$.*

Proof. The minimum model of P is reachable and, by termination and completeness of finite failure, $\mathcal{T}(P)$ is atomically complete. By Theorem 1 we get our result. $\Box$

Example 6. Consider the following framework $\mathcal{PA}_0$:

Framework $\mathcal{PA}_0$;
SORTS: Nat;
FUNCTIONS: $0 : \rightarrow Nat$;
$s : (Nat) \rightarrow Nat$;
D-AXIOMS: $free(0, s : Nat)$;
$sum(x, y, z) \leftrightarrow (y = 0 \wedge x = z) \vee$
$\exists i, v\, (y = s(i) \wedge z = s(v) \wedge sum(x, i, v))$;

By Theorem 14, $\mathcal{PA}_0$ is an adequate closed framework, since it is the theory $\mathcal{T}(\mathbf{sum})$ of the usual (terminating) program **sum** for computing sum.
Now we can prove $\forall x, y\, \exists! z\, sum(x, y, z)$ (by induction) So the composition $\mathcal{PA}_1 = \mathcal{D}_{sum(x,y,x+y)}[\mathcal{PA}_0]$ is defined, and by Theorem 11, it is an adequate closed framework that extends $\mathcal{PA}_0$ by the (computable) function $+$.
Now consider the following framework $\mathcal{LIST}_0$:

Framework $\mathcal{LIST}_0(Elem)$;
SORTS: $List, Elem$;
FUNCTIONS: $nil : \rightarrow List$;
$. : (Elem, List) \rightarrow List$;
D-AXIOMS: $free(nil, .)$;

By Theorem 12, $\mathcal{LIST}_0$ is an adequate open framework. Then by Theorem 11, the composition $\mathcal{LIST}_0[\mathcal{PA}_1]$ is an adequate closed framework.

We could also associate open frameworks with open programs. However, here the situation is more complex. In particular, even non-adequate open frameworks could be used in a sound way for framework composition if suitable extra conditions are satisfied by the composition operation. To treat these cases, we will define adequacy with respect to a *dependency type*. This will yield a calculus to reason about framework composition, that extends our results so far.

4.3 Rules for Dependency Types

By Theorem 1, an equivalent definition of adequacy of an open framework $\mathcal{F}(\Pi)$ is the following: for every framework $\mathcal{C}$, if $\mathcal{C}$ is reachable and *atomically complete*, then $\mathcal{F}[\mathcal{C}]$ is reachable and *atomically complete*. We want to generalise this notion of adequacy: for every framework $\mathcal{C}$, if $\mathcal{C}$ is reachable and satisfies some property possibly stronger than atomic completeness, then $\mathcal{F}[\mathcal{C}]$ is reachable and satisfies some *other* property stronger than atomic completeness. To this end, we introduce the following notion of *constructive evaluation*.

Definition 15. Let I be a set of closed atomic or negated Σ-formulas,[14] and F be a closed formula. Then I (*constructively*) *evaluates* F in Σ, written $ev(I, F)$, iff

[14] A negated formula is any formula of the form $\neg H$, with H possibly non-atomic.

- F is atomic or negated, and $F \in I$.
- $F = A \wedge B$ and: $ev(I, A)$ and $ev(I, B)$.
- $F = A \vee B$ and: $ev(I, A)$ or $ev(I, B)$.
- $F = \exists x A(x)$ and: $ev(I, A(t))$ for some ground term t of the appropriate sort.
- $F = \forall x A(x)$ and: $ev(I, A(t))$ for every ground term t of the appropriate sort.

A (possibly closed) framework $\mathcal{F} = \langle \Sigma, \mathcal{T}, \mathcal{TH}, \Pi \rangle$ *evaluates* F iff F is evaluated by the set of closed atomic or negated formulas classically provable by $\mathcal{T}$. To indicate that $\mathcal{F}$ evaluates F, we will write $\mathcal{F} : F$.

For example, to say that $\mathcal{C}$ is atomically complete, we can write $\mathcal{C} : \forall x\,(r(x) \vee \neg r(x))$ for every relation symbol r of the signature.

We will abbreviate $\forall x\,(r(x) \vee \neg r(x))$ by $\mathbf{dec}(r : a)$, where a is the arity of r, and, if Π is a set of relation declarations, $\mathbf{dec}(\Pi)$ will contain $\mathbf{dec}(r : a)$ for every declaration $r : a \in \Pi$ and $\mathbf{dec}(=: (s, s))$ for every sort symbol used in Π.

Now adequacy can be treated by dependency types as follows.

Definition 16. We say that $\mathcal{F}(\Pi) = \langle \Sigma, \mathcal{T}, \mathcal{TH}, \Pi \rangle$ *preserves reachability* iff, for every framework $\mathcal{G}$ with at least one reachable model, $\mathcal{F}[\mathcal{G}]$ (when defined) has a reachable model.

Definition 17. We say that $\mathcal{F}(\Pi)$ has *evaluation type* $A \Rightarrow B$, written $\mathcal{F} : A \Rightarrow B$, iff, for every reachable $\mathcal{G}$ such that $\mathcal{F} \preceq \mathcal{G}$, we have $\mathcal{G} : A$ entails $\mathcal{G} : B$.

Theorem 18. *Let $\mathcal{F}(\Pi)$ be a parametric framework such that, for every sort s used in Π, it does not contain constant or function symbols of sort s. If $\mathcal{F}(\Pi)$ preserves reachability, and has dependency type* $\mathbf{dec}(\Pi) \Rightarrow \mathbf{dec}(\Delta)$*, where Δ contains the relation symbols not in Π, then $\mathcal{F}(\Pi)$ is adequate.*

Proof. Let $\mathcal{C}$ be closed and $\mathcal{F}[\mathcal{C}]$ be defined. Since $\mathcal{C} : \mathbf{dec}(\Pi)$ and for the sorts used in Π no new constant or function symbol is added, $\mathcal{F}[\mathcal{C}] : \mathbf{dec}(\Pi)$. We get $\mathcal{F}[\mathcal{C}] : \mathbf{dec}(\Delta)$ and hence $\mathcal{F}[\mathcal{C}]$ is atomically complete. Since $\mathcal{F}[\mathcal{C}]$ is reachable, it has an isoinitial model $\mathcal{I}$. Since $\mathcal{I}|\Sigma_{\mathcal{C}}$ is a reachable model of $\mathcal{C}$ (no new constant or function symbols of sorts in $\mathcal{C}$ are added), it is an isoinitial model of $\mathcal{C}$. □

The idea here is to extend Theorem 11 to a more general kind of dependency types, in order to get a richer calculus for module composition. Here we consider the beginnings of such a calculus, which contains rules for reasoning about dependency types and for using them in framework composition.

Rules for Dependency Types

These rules work for any kind of framework.

$$\frac{\mathcal{F}_1(\Pi_1) : A \Rightarrow B, \;\; \mathcal{F}_2(\Pi_2) : \rho(B) \Rightarrow C}{\mathcal{F}_1[\rho, \mathcal{F}_2](\Pi_2) : \rho(A) \Rightarrow C} \tag{6}$$

$$\frac{\mathcal{F}_1(\Pi_1) : B \Rightarrow C, \;\; \mathcal{F}_2(\Pi_2) : A \Rightarrow \rho(B)}{\mathcal{F}_1[\rho, \mathcal{F}_2](\Pi_2) : A \Rightarrow \rho(C)} \tag{7}$$

$$\frac{\mathcal{F}(\Pi) : A \Rightarrow B, \;\; \mathcal{F}(\Pi) : B \Rightarrow C}{\mathcal{F}(\Pi) : A \Rightarrow C} \tag{8}$$

Extension-by-composition Rules. In the following rules $\mathcal{F}(\Pi)$ is adequate, whilst $\mathcal{E}(\Pi')$, which we call an *extension framework*, may be non-adequate. Δ are the relation symbols of $\mathcal{E}$ not in Π'. We require that $\mathcal{E}(\Pi')$ preserves reachability, the (possible) function or constant symbols of sort used in Π' are parameters, and in $\mathcal{E}[\rho, \mathcal{F}](\Pi)$ and $\mathcal{F}[\mathcal{E}](\Pi)$ there are no constant or function symbols of sort in Π. Soundness follows from the previous rules, from the fact that reachability is preserved by $\mathcal{E}$ and from Theorem 18.

$$\frac{\mathcal{E}(\Pi') : B \Rightarrow \mathbf{dec}(\Delta), \;\; \mathcal{F}(\Pi) : \mathbf{dec}(\Pi) \Rightarrow \rho(B)}{\mathcal{F}(\Pi) \sqsubseteq \mathcal{E}[\rho, \mathcal{F}](\Pi)} \tag{9}$$

$$\frac{\mathcal{F}(\Pi) : \mathbf{dec}(\Pi) \Rightarrow B, \;\; \mathcal{E}(\Pi) : B \Rightarrow \mathbf{dec}(\Delta), \;\; \mathcal{T}_{\mathcal{F}}|\Pi \subseteq \mathcal{T}_{\mathcal{E}}|\Pi}{\mathcal{F}(\Pi) \sqsubseteq \mathcal{F}[\mathcal{E}](\Pi)} \tag{10}$$

We can show that if $\mathcal{F}(\Pi)$ and $\mathcal{G}(\Pi')$ preserve reachability, then $\mathcal{F}[\mathcal{G}](\Pi')$ (if defined) preserves reachability too. Therefore we can build extension frameworks by reasoning with their dependency types, starting from a few kinds of basic extension frameworks.

As an example, the theory $\mathcal{T}(P)$ of an open program $P : \delta \Leftarrow \pi$ preserves reachability and, by reasoning about termination properties of P, we can prove $\mathcal{T}(P) : A \Rightarrow B$ for suitable A and B.

Example 7. Let us consider the open program $\mathbf{iter}(iter \Leftarrow unit, op)$:

$$\begin{aligned} iter(X, 0, U) &\leftarrow unit(U) \\ iter(X, s(I), Y) &\leftarrow iter(X, I, W), op(X, W, Y) \end{aligned} \tag{11}$$

We associate **iter** with the extension framework:

Framework $\mathcal{ITER}(unit, op, 0, s, Nat)$;
SORTS: Nat;
FUNCTIONS: $0 :\to Nat$; $s : (Nat) \to Nat$;
RELATIONS: $unit \;\; : (Nat)$;
$iter, op : (Nat, Nat, Nat)$;
D-AXIOMS: $\mathbf{CD}(iter, \mathbf{iter})$
P-AXIOMS: $free(0, s : Nat)$; $construct(0, s : Nat)$;

where $\mathbf{CD}(iter, \mathbf{iter})$ is the completed definition of $iter$ in the program **iter** and $construct(0, s : Nat)$ requires that, if a composition introduces operations on Nat different from $0, s$, then for every ground term $t : Nat$ there is a numeral n (of the form $0, s(0), \ldots$) such that $n = t$ is provable. The framework $\mathcal{ITER}$ is not adequate. However we can prove that it has the following dependency types:

$$\exists ! x \, unit(x) \wedge \forall x, y \exists ! z \, op(x, y, z) \Rightarrow \forall x, y \exists ! z \, iter(x, y, z)$$
$$\forall x, y \, (x = y \vee \neg x = y) \wedge \exists ! x \, unit(x) \wedge \forall x, y \exists ! z \, op(x, y, z) \Rightarrow$$
$$\forall x, y, z \, (iter(x, y, z) \vee \neg iter(x, y, z))$$

where the second type follows from the first since $\emptyset : \forall x, y \, (x = y \vee \neg x = y) \wedge \forall x, y \exists ! z \, iter(x, y, z) \Rightarrow \forall x, y, z \, (iter(x, y, z) \vee \neg iter(x, y, z))$.

We can use this framework to introduce product in $\mathcal{PA}_1$ (see Example 6). First we introduce in $\mathcal{PA}_1$ the explicit definition $zero(X) \leftrightarrow X = 0$ (we get an adequate expansion by Theorem 13). Since $\mathcal{PA}_1 : \forall x, y \, (x = y \vee \neg x = y) \wedge \exists ! x \, zero(x) \wedge \forall x, y \exists ! z \, sum(x, y, z)$, by Rule (9) we get $\mathcal{PA}_1 \sqsubseteq \mathcal{ITER}[\rho, \mathcal{PA}_1]$, where ρ maps $unit$ to $zero$, op to sum and $iter$ to $prod$.

By rule (7) we also get that $\mathcal{ITER}[\rho, \mathcal{PA}_1] : \forall x, y \exists ! z \, prod(x, y, z)$, and we could now use $\mathcal{ITER}$ to obtain exponentiation, and so on.

Moreover, we can extend the notion of a steadfast program to frameworks with dependency types. In this way we get a way of composing programs within a framework whilst preserving their steadfastness (i.e. correctness) properties. Furthermore, these properties will also be preserved by framework composition. Thus we get both *correctness* within a framework and *reusability* with respect to module composition. We will not discuss this issue any further here. Rather, we will focus on a calculus of dependency types (see Section 4.4 later).

4.4 Towards a Calculus of Dependency Types

To conclude the paper, we introduce a preliminary, incomplete calculus for proving dependency types. It allows us, for example, to prove $\forall u, v\,(u = v \vee \neg u = v) \wedge \forall x \exists! y\, r(x, y) \Rightarrow \forall x, y\,(r(x, y) \vee \neg r(x, y))$, where $\exists! u\, H(u)$ is an abbreviation of $\exists u\, H(u) \wedge \neg \exists a, b\,(H(a) \wedge H(b) \wedge \neg a = b)$.

The proof system works on sequents of the form $\Gamma \vdash_{ev} F$, where Γ is a sequence of formulas and F is a formula. For simplicity, we consider the one-sorted case, but the extension to the many-sorted case should be obvious. Free variables are allowed in Γ and F.

A dependency type $A \Rightarrow B$ is *valid*[15] iff, for every signature Δ containing those of A and B, every ground instance $A^* \Rightarrow B^*$ using terms from Δ, and every Δ-theory $\mathcal{T}$, if $\mathcal{T} : A^*$, then $\mathcal{T} : B^*$. We say that $A_1, \ldots, A_n \vdash_{ev} F$ is *evaluation-valid* iff $A_1 \wedge \ldots \wedge A_n \Rightarrow F$ is a valid dependency type. We can show that, if $\Gamma, A \vdash_{ev} B$ is evaluation-valid and Γ is a set of atomic or negated closed formulas provable in a framework $\mathcal{F}(\Pi)$, then $\mathcal{F}(\Pi) : A \Rightarrow B$. This shows how we can use the calculus below for our purposes.

Now we can introduce our proof system, which is sound with respect to evaluation-validity.

(i) The axiom $A \vdash_{ev} A$, and the usual structural rules of sequent calculus.
(ii) The left and right rules for $\wedge$, $\vee$, $\exists$, $\forall$, but not those for negation and implication.
(iii) If the formulas in Γ and A are atomic or negated, and $\Gamma \vdash A$ in classical sequent calculus, then we have the axiom $\Gamma \vdash_{ev} A$.
(iv) The cut rule.

Note that (iii) introduces a recursively enumerable set of axioms, and so the set of provable sequents still remains recursively enumerable. Also, due to (iii), cut cannot be eliminated. However, it can be restricted to atomic or negated formulas. The axiom $A \vdash_{ev} A$ is trivially sound wrt evaluation-validity. The soundness of the axioms introduced by (iii) follows from the fact that a theory evaluates an atomic or closed formula iff it proves it. The inference rules ((ii) and (iv)) can be shown to be sound, by proving that if the premisses of a rule are evaluation-valid, then the consequence is evaluation-valid (we omit the proof for brevity).

This system is a subsystem of the sequent calculus for classical logic. It is not a subsystem of intuitionistic logic. Conversely, intuitionistic logic is not a subsystem of our system. Finally, our system is not a complete system for proving valid dependency types.

[15] With A and B possibly containing free variables.

5 Conclusion

We have presented a formalisation of a module that is correct, and correctly reusable in the sense that module composition preserves both correctness and reusability. We have also given a calculus for constructing modules that behave soundly with respect to composition. Our notion of a module is very general, and it should be equally applicable to any modular or object-oriented programming paradigm.

To extend our work, we intend to continue our study of dependency types, and expand the preliminary calculus we have introduced here. For instance, we could introduce $\Rightarrow$ in the formulas of a sequent $\Gamma \vdash_{ev} F$, and study how the rules for implication behave. At present the implication $A \rightarrow B$ is read as an abbreviation of $\neg(A \wedge \neg B)$ and we do not have rules for it.

Moreover, we could use stronger requirements in the definition of evaluation set, that is we could further require that all the universally quantified formulas that are involved belong to I. A calculus for this kind of strong evaluation would give a logic that is intermediate between intuitionistic logic and classical logic. However, strong principles like the Gregorzyck principle become unsound with respect to strong evaluation. Thus it is not clear if the stronger calculus is an advantage. The best choice may well be to use a mixture of different calculi.

References

1. A. Bertoni, G. Mauri and P. Miglioli. On the power of model theory in specifying abstract data types and in capturing their recursiveness. *Fundamenta Informaticae* VI(2):127–170, 1983.
2. A. Brogi, P. Mancarella, D. Pedreschi and F. Turini. Modular logic programming. *ACM TOPLAS* 16(4):1361-1398, 1994.
3. K.M. Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *J. Functional Programming* 4(2):127–206, 1994.
4. M. Bugliesi, E. Lamma and P. Mello. Modularity in logic programming. *J. Logic Programming* 19,20:443–502, 1994. Special issue: Ten years of logic programming.
5. C.C. Chang and H.J. Keisler. *Model Theory*. North-Holland, 1973.
6. J.A. Goguen and R.M. Burstall. Institutions: Abstract model theory for specification and programming. *J. ACM* 39(1):95–146, 1992.
7. K.K. Lau and M. Ornaghi. On specification frameworks and deductive synthesis of logic programs. In L. Fribourg and F. Turini, editors, *Proc. LOPSTR 94 and META 94*, *LNCS* 883, pages 104–121, Springer-Verlag, 1994.
8. K.K. Lau, M. Ornaghi and S.-Å. Tärnlund. The halting problem for deductive synthesis of logic programs. In P. van Hentenryck, editor, *Proc.* 11^{th} *Int. Conf. on Logic Programming*, pages 665–683, MIT Press, 1994.
9. K.K. Lau, M. Ornaghi and S.-Å. Tärnlund. Steadfast logic programs. Submitted
10. J.W. Lloyd. *Foundations of Logic Programming*, Springer-Verlag, 1987.
11. B. Meyer. *Eiffel the Language*. Prentice Hall, 1992.
12. P. Miglioli, U. Moscato and M. Ornaghi. Abstract parametric classes and abstract data types defined by classical and constructive logical methods. *J. Symb. Comp.* 18:41-81, 1994.
13. J. Palsberg and M.I. Schwartzbach. *Object-Oriented Type Systems*. Wiley, 1994.
14. M. Wirsing. Algebraic specification. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 675–788. Elsevier, 1990.