A Syntactic Approach to Modularity in Denotational Semantics

Pietro Cenciarelli
Dept. of Comp. Sci., Univ. of Edinburgh, UK, email: pic@dcs.ed.ac.uk
Eugenio Moggi*
DISI, Univ. di Genova, ITALY, email: moggi@disi.unige.it

Abstract

This paper proposes a syntactic reformulation of the modular approach to Denotational Semantics in [Mog89a, Mog91a]. This reformulation is based on a *duality* between model constructions and translations of theories (often called relative interpretations), analogous to Gabriel-Ulmer duality. To demonstrate the simplicity and usability of the syntactic reformulation, we give a sample of theories and translations, which can be used to give semantics to concurrent languages (via translation into suitable metalanguages).

Introduction

The main objective of Denotational Semantics is to provide mathematical models of programming languages. These models can be used both as formal descriptions of programming languages and also to validate reasoning principles. However, to make Denotational Semantics usable one must address two key issues:

- **Modularity**. A key issue in the applicability of formalisms to large-scale examples is the ability to *structure* theories so as to be able to reuse components.
- **Simplicity**. In fact, one barrier to the wide-spread use of mathematical models and formal methods is the level of expertise required.

[Mog89a, Mog91a] propose a way of tackling the problem of *modularizing* the semantic specification of programming languages, based on the monadic approach and the use of monad transformers. This paper addresses not only the issue of modularity, but also that of simplicity. Simplicity is achieved by hiding the category-theoretic concepts and sophistication of the original approach in suitable metalanguages. We give a sample of metalanguage theories and translations, which can be used to give semantics to concurrent languages without the ad hoc treatment still present in the original proposal. Finally, we have investigated how these theories and translations relate to lex-categories and lex functors, in the hope that they may suggest more general notions of translation.

Metalanguages for Denotational Semantics. Over the past 20 years the work on Domain Theory and Denotational Semantics by Scott, Plotkin, Milner et al. has provided elegant formalisms (like LCF) based on the typed λ-calculus for dealing with higher-order, recursively defined partial functions on data-types. A more recent advance is the use of computational monads as a structuring principle for semantics (see [Mog91b]): *computational features* are encapsulated in a strong monad T over a category of *values*. LCF is replaced by a metalanguage $ML_T(\Sigma)$ featuring

*This work is supported by ESPRIT BRA 6811 (Categorical Logic In Computer Science II) and EC SCIENCE twinning ERBSC1*CT920795 (Progr. Lang. Semantics and Program Logics).

a type constructor T (mapping A to the type TA of programs *computing* values in A) and T related operations specified in a signature Σ. Then, the semantics of a programming language PL is given via a translation mapping PL programs into $ML_T(\Sigma)$ terms of computational type. By a suitable choice of Σ, one may avoid most of the complexity involved in translating PL directly into LCF, since $ML_T(\Sigma)$ hides the interpretation of T and Σ, just as an interface hides the implementation details of an abstract data-type.

The issue of modularity in semantic specifications. Lack of *modularity* is a primary obstacle to making the techniques of Denotational Semantics applicable to complex programming languages. Finding good structuring principles is one aspect of this problem: large, unstructured, monolithic semantic specifications are just as unusable as large, unstructured, monolithic programs. Another aspect is *generality*: because of the subtle interactions that may occur between different language features, it is hard to build semantics incrementally, combining well-studied features in a safe fashion. In particular, when a programming language is extended, its semantics may need to be extensively redefined. [Mos90] identifies this problem very clearly, and stresses the role of auxiliary notation (our Σ) in making semantic specifications more reusable.

Monad transformers and incremental approach to semantics. The monadic approach to semantics consists of three steps, i.e. given a (complex) programming language PL: first identify a suitable metalanguage $ML_T(\Sigma)$, then define a translation of PL into $ML_T(\Sigma)$, and finally construct a model of $ML_T(\Sigma)$. Usually, the first two steps are straightforward, and most of the difficulties are in the third step. To overcome these difficulties [Mog89a, Mog91a] proposes an *incremental approach* to constructing models of $ML_T(\Sigma)$.
The idea is to mimic the stepwise methodology for program development. If we consider specifications as theories and implementations as models, then we can reduce the problem of finding an implementation of a complex specification Th' to that of finding an implementation of a simpler specification Th, provided there is a *construction* $F\colon Mod(Th) \to Mod(Th')$, where $Mod(Th)$ is the class of models of Th (see [ST87]). If a specification is given by a metalanguage $ML_T(\Sigma)$ (and possibly a theory Th over it), then an implementation is an interpretation of $ML_T(\Sigma)$ in a category with a strong monad (satisfying the axioms of Th) and a construction is a function $F\colon Mod_T(\Sigma) \to Mod_T(\Sigma')$. When signatures are ignored, F amounts to a *monad transformer*, i.e. a function mapping monads to monads. Therefore, starting from a model of a metalanguage $ML_T(\Sigma_0)$, we can incrementally construct a model of $ML_T(\Sigma_n)$ by providing a sequence of signatures $\Sigma_i \subset \Sigma_{i+1}$ $(0 \leq i < n)$ and constructions $F_i\colon Mod_T(\Sigma_i) \to Mod_T(\Sigma_{i+1})$. The constructions we are interested in are not *persistent*, since they involve a reinterpretation of the type constructor T (i.e. a change of monad) and of the operations in Σ_i.

1 The Syntactic Approach to Modularity

Constructions as defined above are far too general; in practice one is interested only in *implementable* ones. For a wide range of logics one can associate to a translation (also called relative interpretation) I of a theory Th' into Th a construction $Mod(I)\colon Mod(Th) \to Mod(Th')$. Therefore, one may manipulate (finite presentations) of translations instead of working directly with constructions. In some cases, e.g. for algebraic theories, one might even establish a *duality* between translations and constructions enjoying certain additional properties. Most constructions $F_i\colon Mod_T(\Sigma_i) \to Mod_T(\Sigma_{i+1})$ used in the incremental approach correspond to translations $I_i\colon ML_T(\Sigma_{i+1}) \to ML_T(\Sigma_i)$. In general, these translations are of the form $I_\Sigma\colon ML_T(\Sigma_{par} + \Sigma + \Sigma_{new}) \to ML_T(\Sigma_{par} + \Sigma)$, where Σ_{new} are the new operations defined by I_Σ, Σ are the old operations *redefined* by I_Σ and Σ_{par} are the parameters of the construction, which are unaffected by I_Σ. Moreover, I_Σ is *natural* w.r.t. Σ ranging over a category Sig_I of *redefinable* signatures and signature morphisms:

$$\begin{tikzcd}
\Sigma \arrow[d, "\sigma"'] & & ML_T(\Sigma_{par}+\Sigma+\Sigma_{new}) \arrow[r, "I_\Sigma"] \arrow[d, "ML_T(\Sigma_{par}+\sigma+\Sigma_{new})"'] & ML_T(\Sigma_{par}+\Sigma) \arrow[d, "ML_T(\Sigma_{par}+\sigma)"] \\
\Sigma' & \text{in } Sig_I \quad \text{implies} & ML_T(\Sigma'+\Sigma_{new}+\Sigma_{par}) \arrow[r, "I_{\Sigma'}"'] & ML_T(\Sigma'+\Sigma_{par})
\end{tikzcd}$$

Because of naturality, I is determined by the translation $I_\emptyset\colon ML_T(\Sigma_{par}+\Sigma_{new}) \to ML_T(\Sigma_{par})$ defining the new symbols, and translations $I_\tau\colon ML_T(\Sigma_{par}+\Sigma_{c:\tau}) \to ML_T(\Sigma_{par}+\Sigma_{c:\tau})$, called *uniform redefinitions*, redefining a generic symbol c of type τ in *isolation*. Uniform redefinition may be possible only for certain τ, which depend on I.

Remark 1.1 The notion of translation used in this paper is general enough to describe the traditional techniques of denotational semantics, but is not applicable to a "change of category", such as $\mathcal{C} \mapsto \mathcal{C}^W$. Indeed, it is unclear whether monad transformers may help in structuring more complex denotational semantics, such as those for modeling local variables (see [OT92]).

Instead of working directly with ML_T, we prefer to use a general purpose metalanguage HML (similar to HML of [Mog89b]), which is not biased towards monads, but is expressive enough to easily axiomatize desirable metalanguage features (including monads) and to describe concisely uniform redefinitions. HML is given by a set of rules (see Appendix A) for deriving the following judgements: well-formedness of signatures $\vdash \Sigma$ sig, theories $\vdash_\Sigma Th$ theory, contexts $\vdash_\Sigma \Gamma$ context, kinds $\vdash_\Sigma$ k: kind, constructors $\Gamma \vdash_\Sigma u\colon \mathrm{k}$, type schemes $\Gamma \vdash_\Sigma \sigma\colon$scheme, terms $\Gamma \vdash_\Sigma e\colon\sigma$ and propositions $\Gamma \vdash_\Sigma \phi\colon$prop; equality of constructors, type schemes and terms; and sequents.
A signature Σ is a sequence for declaring constants at the level of kinds K: kind, constructors $C\colon$k, type schemes S: k$\Rightarrow$scheme, terms $c\colon\sigma$ and propositions P: $\forall v\colon \mathrm{k}.\sigma\Rightarrow$prop. A context Γ is a sequence for declaring variables at the level of constructors $v\colon$k and terms $x\colon\sigma$. Except for term equality judgements $\Gamma \vdash_{\Sigma,Th} e_1 =_\sigma e_2$ and sequents $\Gamma \vdash_{\Sigma,Th} \Phi \Longrightarrow \phi$ (which depend on a theory Th), it is decidable whether a judgement is derivable.
We use a notion of translation, corresponding to signature morphisms mapping primitive operations to derived ones (for simplicity we restrict to the case when propositions are simply equations).

Definition 1.2 (Raw translation) *Given a partial function F from constants to raw expressions, we say that F is a raw translation iff whenever defined*

- $F(\mathrm{K}) \in Kind$
- $F(C) \in Constr$ *and has no free variables*
- $F(\mathrm{S}) \in Schema$ *and has at most one free variable, say v*
- $F(c) \in Term$ *and has no free variables.*

A raw translation F induces a total function F^* defined by induction on raw expressions:

- $F^*(\mathrm{K}) = F(\mathrm{K})$ *if* $F(\mathrm{K})$ *is defined,* $F^*(\mathrm{K}) = \mathrm{K}$ *otherwise*
- $F^*(C) = F(C)$ *if* $F(C)$ *is defined,* $F^*(C) = C$ *otherwise*
- $F^*(\mathrm{S}(u)) = [F^*(u)/v]F(\mathrm{S})$ *if* $F(\mathrm{S})$ *is defined,* $F^*(\mathrm{S}(u)) = \mathrm{S}(F^*(u))$ *otherwise*
- $F^*(c) = F(c)$ *if* $F(c)$ *is defined,* $F^*(c) = c$ *otherwise*
- *in all other cases F^* commutes with the top-level form.*

Definition 1.3 (Translation) *Given two signatures Σ_1 and Σ_2, we say that F is a* **translation** *from Σ_1 to Σ_2 (and write $F\colon\Sigma_1 \to \Sigma_2$) iff F is a raw translation with domain* $\mathrm{DC}(\Sigma_1)$ *and*

- $\vdash_{\Sigma_2} F(\mathrm{K})\colon$kind, *if* $\Sigma_1(\mathrm{K}) =$ kind

- $\emptyset \vdash_{\Sigma_2} F(C){:}\, F^*(\mathrm{k})$, *if* $\Sigma_1(C) = \mathrm{k}$
- $v{:}\, F^*(\mathrm{k}) \vdash_{\Sigma_2} F(\mathrm{S}){:}\,\mathrm{scheme}$, *if* $\Sigma_1(\mathrm{S}) = \mathrm{k}{\Rightarrow}\mathrm{scheme}$
- $\emptyset \vdash_{\Sigma_2} F(c){:}\, F^*(\sigma)$, *if* $\Sigma_1(c) = \sigma$

*Given a Σ_2-theory Th, we say that two translations $F, G{:}\, \Sigma_1 \to \Sigma_2$ are Th-**equivalent** iff*

- $F(\mathrm{K}) = G(\mathrm{K})$, *if* $\Sigma_1(\mathrm{K}) = \mathrm{kind}$
- $\emptyset \vdash_{\Sigma_2} F(C) = G(C){:}\, F^*(\mathrm{k})$, *if* $\Sigma_1(C) = \mathrm{k}$
- $v{:}\, F^*(\mathrm{k}) \vdash_{\Sigma_2} F(\mathrm{S}) = G(\mathrm{S}){:}\,\mathrm{scheme}$, *if* $\Sigma_1(\mathrm{S}) = \mathrm{k}{\Rightarrow}\mathrm{scheme}$
- $\emptyset \vdash_{\Sigma_2,Th} F(c) = G(c){:}\, F^*(\sigma)$, *if* $\Sigma_1(c) = \sigma$

Definition 1.4 (Interpretation) *Given a Σ_1-theory Th_1 and a Σ_2-theory Th_2, we say that F is an* **interpretation** *of Th_1 in Th_2 (and write $F{:}\, (\Sigma_1, Th_1) \to (\Sigma_2, Th_2)$) iff $F{:}\, \Sigma_1 \to \Sigma_2$ and*

- $\emptyset \vdash_{\Sigma_2,Th_2} F^*(\phi)$, *for every assertion $\phi \in Th_1$.*

where F^* is extended to assertions in the obvious way.

It is decidable whether a raw translation is a translation from Σ_1 to Σ_2, but it may be undecidable whether a translation is an interpretation, or whether two translations are Th-equivalent.

Proposition 1.5 *If $F{:}\, \Sigma_1 \to \Sigma_2$ is a translation, then*

- $F^*(\Phi) \vdash_{\Sigma_2} F^*(J)$ *is derivable, when* $\Phi \vdash_{\Sigma_1} J$ *is*
- $F^*(\Phi) \vdash_{\Sigma_2,F^*(Th)} F^*(J)$ *is derivable, when* $\Phi \vdash_{\Sigma_1,Th} J$ *is*

Corollary 1.6 (Composition) *If $F{:}\, \Sigma_1 \to \Sigma_2$ and $G{:}\, \Sigma_2 \to \Sigma_3$ are translations, then the raw translation $H = F; G^*$ is a translation $H{:}\, \Sigma_1 \to \Sigma_3$, called the composite of F followed by G.*

2 Case studies

This Section gives a sample of HML-theories and -translations for describing metalanguages $ML_T(\Sigma)$ and translations $I_\Sigma{:}\, ML_T(\Sigma_{par} + \Sigma + \Sigma_{new}) \to ML_T(\Sigma_{par} + \Sigma)$. For instance, consider the translation I_Σ corresponding to the monad transformer $FTX = T(X + E)$ for exceptions: the only parameter is the type E, the new symbols are the (polymorphic) operations $raise_X{:}\, E{\Rightarrow}TX$ and $handle_X{:}\, TX, (E{\Rightarrow}TX){\Rightarrow}TX$, while the old symbols might be either types or operations $c_X{:}\, \tau_1{\times}(\tau_2{\Rightarrow}TX){\Rightarrow}TX$, where T is neither in τ_1 nor in τ_2. To describe I_Σ in HML we use two translations:

- $HML(\Sigma_+ + \Sigma_{par} + \Sigma_T + \Sigma_{new}) \to HML(\Sigma_+ + \Sigma_{par} + \Sigma_T)$ corresponds to $I_\emptyset$, where the HML-signature must make explicit not only the parameters Σ_{par} of the construction, but also the relevant features of ML_T (in this case sums and computational types);
- $HML(\Sigma_\mathrm{S}) \to HML(\Sigma_\mathrm{S})$, where $\Sigma_\mathrm{S} \equiv G{:}\, (\Omega{\Rightarrow}\Omega){\times}\Omega{\Rightarrow}\Omega, \mathrm{S}{:}\, \Omega{\Rightarrow}\mathrm{scheme}, T{:}\, \Omega{\Rightarrow}\Omega, c{:}\, \forall X{:}\, \Omega.\mathrm{S}(TX)$, corresponds to uniform redefinition $I_{\mathrm{S}(TX)}$ for a monad transformer $FTX = T(GTX)$ of a generic operation $c_X{:}\, \mathrm{S}(TX)$. By a suitable choice of G and S we can specialize $I_{\mathrm{S}(TX)}$ to the case $FTX = T(X + E)$ and $c_X{:}\, \tau_1{\times}(\tau_2{\Rightarrow}TX){\Rightarrow}TX$, but it may be specialized to other monad transformers, e.g.: $T(X{\times}M)$ for complexity and $\mu X'.T(X + X')$ for resumptions.

We investigate also the translation I_Σ corresponding to the monad transformer $FTX = \mu X'.T(X + X')$ for resumptions, and show that many forms of parallel composition can be expressed using the new symbols τ_H and C_H defined by I_Σ.

Remark 2.1 For describing uniform redefinitions it seems essential to have type variables and signatures introducing also kinds and type schemes besides type constructors and polymorphic operations. When we stay in the fragment of HML corresponding to Standard ML, we may improve conciseness by dropping type information which can be recovered by type inference. So far we did not need the additional expressiveness given by dependent types, which are so important in logical frameworks such as LF.
In what follows type information may be erased from well-formed terms and equations, when it can be recovered (uniquely up to provable type equality) from the type information in signatures and contexts.

2.1 Examples of theories

Metalanguages can be represented in HML like logics are represented in LF (see [HHP87]). More precisely, we represent a metalanguage ML by a pair (Σ, Th), consisting of a HML-signature Σ and a Σ-theory Th, such that the formal system $HML(\Sigma, Th)$ (obtained by fixing signature and theory) is a conservative extension of ML, after some suitable translation of ML into $HML(\Sigma, Th)$. For instance, a metalanguage ML_T for categories with finite products and a strong monad can be represented by taking $Th = Th_\times + Th_T$, and if we want also coproducts and exponentials of the form $(TY)^X$, then we should use $Th = Th_\times + Th_+ + Th_T + Th_\Rightarrow$.

Example 2.2 [Theory for monads]

- signature Σ_T

  $T\colon \Omega\Rightarrow\Omega$,
  $val\colon \forall v\colon \Omega.v\Rightarrow Tv$,
  $let\colon \forall v_1, v_2\colon \Omega.Tv_1, (v_1\Rightarrow Tv_2)\Rightarrow Tv_2$

- theory Th_T

1. $v\colon \Omega, c\colon Tv \vdash c = let(c, val)$
2. $v_1, v_2\colon \Omega, x\colon v_1, f\colon v_1\Rightarrow Tv_2 \vdash let(val(x), f) = f(x)$
3. $v_1, v_2, v_3\colon \Omega, c\colon Tv_1, f\colon v_1\Rightarrow Tv_2, g\colon v_2\Rightarrow Tv_3 \vdash let(let(c, f), g) = let(c, \lambda x.let(f(x), g))$

Example 2.3 [theory for reflection of products]

- signature $\Sigma_\times$

  $unit\colon \Omega$,
  $In_1\colon 1\Rightarrow unit$,
  $Out_1\colon unit\Rightarrow 1$,
  $prod\colon \Omega, \Omega\Rightarrow\Omega$,
  $In_\times\colon \forall v_1, v_2\colon \Omega.(v_1\times v_2)\Rightarrow prod(v_1, v_2)$,
  $Out_\times\colon \forall v_1, v_2\colon \Omega.prod(v_1, v_2)\Rightarrow(v_1\times v_2)$

- theory $Th_\times$

  $x\colon 1 \vdash Out_1(In_1(x)) = x$

  $x\colon unit \vdash In_1(Out_1(x)) = x$

  $v_1, v_2\colon \Omega, x\colon v_1\times v_2 \vdash Out_\times(In_\times(x)) = x$

  $v_1, v_2\colon \Omega, x\colon prod(v_1, v_2) \vdash In_\times(Out_\times(x)) = x$

Example 2.4 [Theory for sums]

- signature Σ_+

  $0\colon \Omega$,
  $+\colon \Omega, \Omega\Rightarrow\Omega$,
  $Z\colon \forall v\colon \Omega.0\Rightarrow v$,
  $in_i\colon \forall v_1, v_2\colon \Omega.v_i\Rightarrow v_1 + v_2,\ (i = 1, 2)$
  $case\colon \forall v_1, v_2, v\colon \Omega.v_1 + v_2, (v_1\Rightarrow v), (v_2\Rightarrow v)\Rightarrow v$

- theory Th_+

1. $v{:}\,\Omega, f{:}\,0{\Rightarrow}v, x{:}\,0 \vdash f(x) = Z(x)$
2. $v_1, v_2, v{:}\,\Omega, x{:}\,v_i, f_1{:}\,v_1{\Rightarrow}v, f_2{:}\,v_2{\Rightarrow}v \vdash \mathit{case}(in_i(x), f_1, f_2) = f_i(x)\ (i = 1, 2)$
3. $v_1, v_2, v{:}\,\Omega, x{:}\,v_1 + v_2, f{:}\,(v_1 + v_2){\Rightarrow}v \vdash \mathit{case}(x, \lambda x_1.f(in_1(x_1)), \lambda x_2.f(in_2(x_2))) = f(x)$

Example 2.5 [theory for reflection of T-exponentials]

- signature $\Sigma_{\Rightarrow} = \Sigma_T +$
  $Tfun{:}\,\Omega, \Omega{\Rightarrow}\Omega,$
  $In_{\Rightarrow}{:}\,\forall v_1, v_2{:}\,\Omega.(v_1{\Rightarrow}Tv_2){\Rightarrow}Tfun(v_1, v_2),$
  $Out_{\Rightarrow}{:}\,\forall v_1, v_2{:}\,\Omega.Tfun(v_1, v_2){\Rightarrow}(v_1{\Rightarrow}Tv_2)$

- theory $Th_{\Rightarrow}$
  $v_1, v_2{:}\,\Omega, x{:}\,v_1{\Rightarrow}Tv_2 \vdash Out_{\Rightarrow}(In_{\Rightarrow}(x)) = x$
  $v_1, v_2{:}\,\Omega, x{:}\,Tfun(v_1, v_2) \vdash In_{\Rightarrow}(Out_{\Rightarrow}(x)) = x$

Example 2.6 [Theory for fixed point operator]

- signature $\Sigma_Y = \Sigma_T +$
  $Y{:}\,\forall v_1, v_2{:}\,\Omega.((v_1{\Rightarrow}Tv_2), v_1{\Rightarrow}Tv_2), v_1{\Rightarrow}Tv_2$

- theory $Th_Y = Th_T +$

1. $v_1, v_2{:}\,\Omega.f{:}\,(v_1{\Rightarrow}Tv_2){\Rightarrow}(v_1{\Rightarrow}Tv_2) \vdash Y(f) = f(Yf)$

Example 2.7 [Theory for inductive types]

- signature Σ_μ
  $\mu{:}\,(\Omega{\Rightarrow}\Omega){\Rightarrow}\Omega,$
  $i{:}\,\forall F{:}\,\Omega{\Rightarrow}\Omega.F(\mu F){\Rightarrow}\mu F,$
  $I{:}\,\forall F{:}\,\Omega{\Rightarrow}\Omega, v{:}\,\Omega.(\forall v_1, v_2{:}\,\Omega.(v_1{\Rightarrow}v_2), F(v_1){\Rightarrow}F(v_2)), (F(v){\Rightarrow}v), \mu F{\Rightarrow}v$

- theory Th_μ

1. $F{:}\,\Omega{\Rightarrow}\Omega, v{:}\,\Omega, f{:}\,(\forall v_1, v_2{:}\,\Omega.(v_1{\Rightarrow}v_2), F(v_1){\Rightarrow}F(v_2)), \alpha{:}\,F(v){\Rightarrow}v \vdash$
   $f \text{ strength} \Longrightarrow \forall x{:}\,F(\mu F).I(f, \alpha, i(x)) = \alpha(f(I(f, \alpha), x))$
2. $F{:}\,\Omega{\Rightarrow}\Omega, v{:}\,\Omega, f{:}\,(\forall v_1, v_2{:}\,\Omega.(v_1{\Rightarrow}v_2), F(v_1){\Rightarrow}F(v_2)), \alpha{:}\,F(v){\Rightarrow}v, g{:}\,\mu F{\Rightarrow}v \vdash$
   $f \text{ strength}, \forall x{:}\,F(\mu F).g(i(x)) = \alpha(f(g, x)) \Longrightarrow g = I(f, \alpha)$

   where "f strength" is the assertion saying that "f is a strength for F"

Remark 2.8

- We write $\tau_1{\times}\tau_2$, $\tau_1{\Rightarrow}T\tau_2$ and $\tau_1 + \tau_2$ for $\mathit{prod}(\tau_1, \tau_2)$, $Tfun(\tau_1, \tau_2)$ and $+(\tau_1, \tau_2)$. Moreover, we do not write the conversion functions In_1, Out_1, $In_{\times}$, $Out_{\times}$ $In_{\Rightarrow}$ and $Out_{\Rightarrow}$.

- We write $\mu h.fh$ instead of $Y[v_1, v_2](f)$.

- We do not write $i[F]$, when is it clear from the context. Moreover, we write $\mu X.\tau$ for $\mu(\lambda X{:}\,\Omega.\tau)$ and $\mu h.\lambda x.\alpha(fhx)$ for $I[F, v](f, \alpha)$. The latter does not cause ambiguity with the notation introduced for fixed point operators, since $Y[\mu F, v](\lambda h, x.\alpha(fhx))$ and $I[F, Tv](f, \alpha)$ are provably equivalent in $Th_\mu + Th_Y$, when $\alpha{:}\,F(Tv){\Rightarrow}Tv$.

- The axiomatization for a fixed point operator is rather weak; it would be better to use one for the least fixed point operator.

- The theory of inductive types is consistent if we add sums(, products, fixed point operators) and T-exponentials, but it becomes inconsistent if we require also all exponentials to be reflected (e.g. consider $FX = (X \Rightarrow 2) \Rightarrow 2)$. Consistency can be proved using the following indexed category (which suggests other consistent extensions): the base consists of (large) ω-categories and ω-functors, in particular Ω is interpreted by the ω-category of small cpos and embedding-partial projections pairs, while the fiber over k is $Cpo^{|\mathrm{k}|}$, where $|\mathrm{k}|$ are the objects of k, and Cpo is the category of (large) ω-cpos and ω-continuous functions.

2.2 Examples of translations

This section exemplifies the use of HML-translations for representing the ML_T-translations $I_\emptyset$ corresponding to the monad transformers $FTX = T(X+E)$ for exceptions and $FTX = \mu Y.T(X + H(Y))$ for generalized resumptions (where H is a strong endofunctor). By a suitable choice of H, the monad transformer for generalized resumptions may be specialized to others, e.g.: $FTX = \mu Y.T(X+Y)$ for resumptions, $FTX = \mu Y.T(X + (A \times V \times Y))$ for interactive output and $FTX = \mu Y.T(X + (A \times Y^V))$ for interactive input.
In what follows, a translation $I_\emptyset \colon ML_T(\Sigma_{par} + \Sigma_{new}) \to ML_T(\Sigma_{par})$ will be represented by a HML-translation $F \colon HML(\Sigma_2) \to HML(\Sigma_1)$. In general, Σ_1 and Σ_2 are constructed through the following sequence of steps, each introducing a HML-signature extending the one defined in the previous step:

- Σ_{ML} is the signature representing the *relevant* features of ML_T not redefined by $I_\emptyset$;
- $\Sigma_0 = \Sigma_{ML} + \Sigma_{par}$, usually F restricted to Σ_0 is the identity;
- $\Sigma_1 = \Sigma_0 +$ the *relevant* features of ML_T redefined by $I_\emptyset$, usually $\Sigma_1 = \Sigma_0 + \Sigma_T$;
- $\Sigma_2 = \Sigma_1 + \Sigma_{new}$.

Example 2.9 [Translation for exceptions $FTX = T(X+E)$]

- $\Sigma_0 = \Sigma_+ + E \colon \Omega$

  E represents the type of exceptions
- $\Sigma_1 = \Sigma_0 + \Sigma_T$
- $\Sigma_2 = \Sigma_1 +$
  $raise \colon \forall X \colon \Omega . E \Rightarrow TX$,
  $handle \colon \forall X \colon \Omega . TX, (E \Rightarrow TX) \Rightarrow TX$
- $F \colon \Sigma_2 \to \Sigma_1$ is the identity over Σ_0 and

  $F(T)(X) = T(X+E)$

  $F(val)[X](x) = val(in_1(x))$

  $F(raise)[X](n) = val(in_2(n))$

  $F(handle)[X](c,f) = let(c, \lambda x.case(x, F(val), f))$

  $F(let)[X,Y](c,f) = let(c, \lambda x.case(x, f, F(raise)))$

Example 2.10 [Translation for generalized resumptions $FTX = \mu Y.T(X + HY)$]

- $\Sigma_0 = \Sigma_+ + \Sigma_\mu +$
  $H \colon \Omega \Rightarrow \Omega$,
  $st \colon (\forall X, Y \colon \Omega . (X \Rightarrow Y), HX \Rightarrow HY)$

  (H, st) represents a strong endofunctor
- $\Sigma_1 = \Sigma_0 + \Sigma_T$

- $\Sigma_2 = \Sigma_1 +$
  $\tau_H : \forall X : \Omega . H(TX) \Rightarrow TX$,
  $C_H : \forall X, Y : \Omega . (X \Rightarrow TY), (H(TX) \Rightarrow TY) \Rightarrow (TX \Rightarrow TY)$

- $F : \Sigma_2 \to \Sigma_1$ is the identity over Σ_0 and

  $F(T)(X) = \mu X'.T(X + HX')$

  $F(val)[X](x) = val(in_1(x))$

  $F(\tau_H)[X](z) = val(in_2(z))$

  $F(C_H)[X,Y](f,g,c) = let(c, \lambda x.case(x,f,g))$

  $F(let)[X,Y](c,f) = (\mu h.\lambda c.F(C_H)(c,f,\lambda z.F(\tau_H)(st(h)z)))(c)$

One may specialize (H, st) in several ways, e.g.

- when $H(X) = E$, one gets back the translation for exceptions, and therefore Σ_μ becomes unnecessary. Actually, C_H does not specialize to *handle*, but they are *interdefinable*.

- when $H(X) = X$, one gets the translation for resumptions. In this case, τ_H *corresponds* to τ-prefixing (of CCS), while C_H does not have an obvious analogue in concurrent languages. However, several operations on concurrent programs may be *defined* in terms of τ_H, C_H and a fixed point combinator Y (see Section 2.4).

Remark 2.11 [Commutativity for generalized resumptions] Given two endofunctors H_1 and H_2, let $HX = (H_1X + H_2X)$ and F_1, F_2 and F be the translations for generalized resumptions corresponding to H_1, H_2 and H, then FTX, $F_1(F_2T)X$ and $F_2(F_1T)X$ are provably isomorphic. Moreover, τ_H and C_H are *definable* from τ_{H_1}, C_{H_1}, τ_{H_2} and C_{H_2} (and conversely). One way to establish such a definability result is by proving the equivalence of two composite translations.

2.3 Examples of uniform redefinitions

This section exemplifies the use of HML-translations for representing uniform redefinitions I_τ corresponding to a monad transformer of the form $F(T)X = T(G(T,X))$. Monad transformers of this form include: $FTX = \mu X'.T(X + HX')$ for generalized resumptions and $FTX = T(X \times M)$ for complexity (where M is a monoid).
In what follows, a uniform redefinition $I_\tau : ML_T(\Sigma_{par} + \Sigma_{c:\tau}) \to ML_T(\Sigma_{par} + \Sigma_{c:\tau})$ will be represented by a HML-translation $F : HML(\Sigma_2) \to HML(\Sigma_2)$. In general, Σ_2 is constructed through the following sequence of steps, each introducing a HML-signature extending the one defined in the previous step:

- Σ_{ML} is the signature representing the *relevant* features of ML_T not redefined by I_τ;

- $\Sigma_0 = \Sigma_{ML} + \Sigma_{par}+$ the part of $\Sigma_{c:\tau}$ not redefined by I_τ (which consists mainly of symbols occurring in τ), usually F restricted to Σ_0 is the identity;

- $\Sigma_1 = \Sigma_0+$ the *relevant* features of ML_T redefined by I_τ, e.g. $\Sigma_1 = \Sigma_0 + \Sigma_T$;

- $\Sigma_2 = \Sigma_1 + c : \tau$.

Example 2.12 [Uniform redefinition of $c_X : \mathrm{S}(TX)$]

- $\Sigma_0 = G : (\Omega \Rightarrow \Omega), \Omega \Rightarrow \Omega$,
  $\mathrm{S} : \Omega \Rightarrow \mathrm{scheme}$

  G is a parameter of the construction, while S is used for typing the symbol c

- $\Sigma_1 = \Sigma_0 + T : \Omega \Rightarrow \Omega$

  N.B.: this uniform redefinition applies also when T does not have all the structure of a monad

- $\Sigma_2 = \Sigma_1 + c : \forall X : \Omega . \mathrm{S}(TX)$

- $F\colon\Sigma_2 \to \Sigma_2$ is the identity over Σ_0 and

  $F(T)(X) = T(G(T,X))$

  $F(c)[X] = c[G(T,X)]$

Uniform redefinition of $c\colon\forall X_1,\ldots,X_n\colon\Omega.\mathrm{S}(TX_1,\ldots,TX_n)$, when $\mathrm{S}\colon\Omega^n\Rightarrow$scheme, can be defined similarly. Most of the symbols, that may need to be redefined in the incremental approach to semantics, have a type scheme of the form $\mathrm{S}(TX)$, where S is a type scheme not depending on T, or can be replaced by symbols whose type scheme is of such a form, e.g.:

- $nil_X\colon TX$, $or_X\colon(TX)^2\Rightarrow TX$
- $lookup\colon L\Rightarrow TU$, $update\colon L,U\Rightarrow T1$, $new\colon TL$
- $raise_X\colon E\Rightarrow TX$, $handle_X\colon TX,(E\Rightarrow TX)\Rightarrow TX$
- $\tau_X\colon TX\Rightarrow TX$, $?_X\colon A,(V\Rightarrow TX)\Rightarrow TX$, $!_X\colon A,V,TX\Rightarrow TX$.

But there are two important exceptions:

- $call/cc_{X,Y}\colon((X\Rightarrow TY)\Rightarrow TX)\Rightarrow TX$
- $C_{X,Y}\colon(X\Rightarrow TY),(H(TX)\Rightarrow TY)\Rightarrow(TX\Rightarrow TY)$ introduced in Example 2.10.

However, there are uniform redefinitions also for (operations of) these type schemes, although they are more involved and make further assumptions on G and T.

2.4 Applications

In this section we consider a metalanguage $ML_T(\Sigma+\Sigma_{new})$ which includes the operations τ_H and C_H of Example 2.10, and show that various operations on concurrent programs (with shared memory) can be expressed in this metalanguage.
The idea is to start with a metalanguage $ML_T(\Sigma)$ for non-deterministic (imperative) languages, where the operations declared in Σ should include at least

- $or[X]\colon TX,TX\Rightarrow TX$ non-deterministic choice, and
- $Y[X,Y]\colon((X\Rightarrow TY),X\Rightarrow TY),X\Rightarrow TY$ (least) fixed point operator.

A model of $ML_T(\Sigma)$ in the category of cpos is obtained by taking $TX=\mathcal{P}_{pl}(X\times S)^S$, where $\mathcal{P}_{pl}$ is Plotkin's powerdomain. Then, we can extend $ML_T(\Sigma)$ to $ML_T(\Sigma+\Sigma_{new})$, by adding the new operations corresponding to the monad transformer for resumptions (see Example 2.10):

- $\tau_H[X]\colon TX\Rightarrow TX$, and
- $C_H[X,Y]\colon(X\Rightarrow TY),(TX\Rightarrow TY),TX\Rightarrow TY$.

A model of $ML_T(\Sigma+\Sigma_{new})$ can be obtained via a translation into $ML_T(\Sigma)$; we use the translation in Example 2.10 (with $HX=X$) for the new operations (and T), and the uniform redefinition in Example 2.12 for the operations in Σ. In the model of $ML_T(\Sigma+\Sigma_{new})$ obtained in this way $TX=\mu Y.\mathcal{P}_{pl}((X+Y)\times S)^S$, and $T1$ is isomorphic to the cpo of resumptions $\mu R.\mathcal{P}_{pl}((S+(R\times S))^S$.
Now that we have outlined the *intended interpretation* of $ML_T(\Sigma+\Sigma_{new})$, we can give a sample of operations definable from or, Y, τ_H and C_H in ML_T:

- critical section $critical[X]\colon TX\Rightarrow TX$ is defined by primitive recursion

  $critical[X]=C_H(\lambda x.\tau_H(val(x)),critical)$
- parallel and-composition and right and-merge $pand[X_1,X_2],rand[X_1,X_2]\colon TX_1,TX_2\Rightarrow T(X_1\times X_2)$ are defined by mutual recursion
  - $pand[X_1,X_2](c_1,c_2)=or(rand(c_1,c_2),rand(c_2,c_1))$

– $rand[X_1, X_2](c) = C_H(\lambda x_2.let(c, \lambda x_1.val(\langle x_1, x_2\rangle)), \lambda c'.pand(c, c'))$

- parallel or-composition and right or-merge $por[X], ror[X]: TX, TX \Rightarrow TX$ are defined by mutual recursion

– $por[X](c_1, c_2) = or(ror(c_1, c_2), ror(c_2, c_1))$

– $ror[X](c) = C_H(\lambda x_2.val(x_2), \lambda c'.por(c, c'))$

Remark 2.13 Intuitively, *critical*(c) forces the program c to be executed in one step, and the two parallel compositions execute their arguments in parallel, but $pand(c_1, c_2)$ completes only when both c_1 and c_2 have completed, while $por(c_1, c_2)$ completes as soon as c_1 or c_2 have completed. In particular, in the intended model of $ML_T(\Sigma + \Sigma_{new})$ the following equalities hold: $pand(val(x_1), val(x_2)) = val(\langle x_1, x_2\rangle)$ and $por(val(x_1), val(x_2)) = or(val(x_1), val(x_2))$
However, to establish properties of these derived operations formally (i.e. without appealing to specific models), we should have proved that the translations defined in Examples 2.10 and 2.12 are indeed interpretations between suitable HML-theories.

3 Categorical view

The idea of *functorial semantics* is that certain fragments of logical theories correspond to certain choices of categorical properties $\mathcal{P}$, sometimes called *doctrines* ([Law75, KR77]), so that theories of kind $\mathcal{P}$ can be identified with categories with $\mathcal{P}$ structure. In this setting, interpretations of a theory T in a category $\mathcal{C}$ with $\mathcal{P}$ structure correspond to $\mathcal{P}$-preserving functors from T to $\mathcal{C}$, that is, models of T in $\mathcal{C}$ are objects in the functor category $[T, \mathcal{C}]_{\mathcal{P}}$. According to this view, HML-theories correspond to special sorts of (split) fibrations, which we call $\lambda\overline{\omega}$-categories; they are a mild generalization of PL-categories ([See87]), Seely's categorical version of the *higher order polymorphic λ-calculus*. Moreover, one expects a correspondence between structure preserving maps between $\lambda\overline{\omega}$-categories and HML-translations.

Definition 3.1 *A $\lambda\overline{\omega}$-category T is a fibred CCC over a CCC base, which admits universal quantification along cartesian projections, and with two distinguished objects: Ω in the base and t in the fiber over Ω. $\lambda\overline{\omega}$-Cat is the category of $\lambda\overline{\omega}$-categories and structure preserving maps. The category Mod(T) of models of T is the comma category $(T \downarrow \lambda\overline{\omega}\text{-}Cat)$.*

We seek a correspondence between translations $\phi: T_2 \to T_1$ and functors $\Phi: Mod(T_1) \to Mod(T_2)$, so as to give a measure of HML's ability to express constructions. Given the above definition of models, there is a straightforward notion of *relative interpretation*: a morphism $\phi: T_2 \to T_1$, induces a functor $\Phi: Mod(T_1) \to Mod(T_2)$ by composition. We may say that Φ is *syntactically induced* because it is determined by ϕ which we think of as a translation of theories. In order to establish the categorical properties of syntactically induced functors we relate $\lambda\overline{\omega}$-categories with lex-categories:

Proposition 3.2 *a) There exists a lex-category $\Lambda\overline{\omega}$ s.t. $[\Lambda\overline{\omega}, Sets]_{lex} \cong \lambda\overline{\omega}\text{-}Cat$. b) There exists a functor $th: \lambda\overline{\omega}\text{-}Cat \to Lex$ s.t. $Mod(T) \cong [th(T), Sets]_{lex}$.*

Part a) of the proposition is proven by exhibiting a *universal Horn theory* ([Kea75]) for it. Part b) is pictured by the following diagram, where $(th, \cong)$ is a morphism of indexed categories:

$$\begin{tikzcd}
 & Lex^{op} \arrow[dr, "{[\_, Sets]_{lex}}"] & \\
\lambda\overline{\omega}\text{-}Cat^{op} \arrow[ur, "th^{op}"] \arrow[rr, "(\_ \downarrow \lambda\overline{\omega}\text{-}Cat)"'] \arrow[rr, phantom, shift left=6, "\Uparrow \cong"] & & Cat
\end{tikzcd}$$

This result tells us that the categories defined above are locally finitely presentable (by the Gabriel-Ulmer duality in [GU75]), and that functors $\mathit{Mod}(I)\colon \mathit{Mod}(T_1) \to \mathit{Mod}(T_2)$ induced by a meta-language translation $I\colon T_2 \to T_1$ (i.e. a morphism in $\lambda\overline{\omega}$-*Cat*) have left adjoints, allowing I to be recovered from $\mathit{Mod}(I)$. This view also suggests more general constructions than $\mathit{Mod}(I)$, namely the constructions corresponding to morphisms from $\mathit{th}(T_2)$ to $\mathit{th}(T_1)$ in *Lex*, which seem to include constructions corresponding to a "change of category".

References

[Gor79] M.J.C. Gordon. *The Denotational Description of Programming Languages.* Springer-Verlag, 1979.

[GU75] P. Gabriel and F. Ulmer. Lokal Präsentierbare kategorien. In *Lecture notes in mathematics, vol.445*, Berlin, 1975. Springer.

[HHP87] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. In *2nd LICS Conf.* IEEE, 1987.

[Kea75] O. Keane. Abstract horn theories. In F.W. Lawvere, C. Maurer, and G.C. Wraith, editors, *Model theory and topoi*, pages 15–50, Berlin, 1975. Springer. Lecture notes in mathematics, vol.445.

[KR77] A. Kock and G.E. Reyes. Doctrines in categorical logic. In J. Barwise, editor, *Handbook of mathematical logic*, pages 283–313. North-Holland, 1977.

[Law75] F. W. Lawvere. Introduction. In F.W. Lawvere, C. Maurer, and G.C. Wraith, editors, *Model theory and topoi*, pages 3–14, Berlin, 1975. Springer. Lecture notes in mathematics, vol.445.

[Mog89a] E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Edinburgh Univ., Dept. of Comp. Sci., 1989. Lecture Notes for course CS 359, Stanford Univ.

[Mog89b] E. Moggi. A category-theoretic account of program modules. In *Proceedings of the Conference on Category Theory and Computer Science, Paris, France, Sept. 1989*, volume 530 of *Lecture Notes in Computer Science*. Springer Verlag, 1989.

[Mog91a] E. Moggi. A modular approach to denotational semantics. Invited talk for the Conference on Category Theory and Computer Science, Paris, France, 1991.

[Mog91b] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1), 1991.

[Mos90] P. Mosses. Denotational semantics. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. North Holland, 1990.

[OT92] P.W. O'Hearn and R.D. Tennent. Semantics of local variables. In *Applications of Categories in Computer Science*, number 177 in L.M.S. Lecture Notes Series. Cambridge University Press, 1992.

[Sch86] D.A. Schmidt. *Denotational Semantics: a Methodology for Language Development.* Allyn & Bacon, 1986.

[See87] R.A.G. Seely. Categorical semantics for higher order polymorphic lambda calculus. *Journal of Symbolic Logic*, 52(2), 1987.

[ST87] D. Sannella and A. Tarlecki. Toward formal development of programs from algebraic specifications: implementations revisited. In H. Ehrig and al., editors, *TAPSOFT 87*, volume 250 of *Lecture Notes in Computer Science*. Springer Verlag, 1987.

A The metalanguage HML

For simplicity, we give the rules of HML in the case when propositions are just equations. Therefore, we ignore sequents and declarations of predicates $\mathrm{P}\colon \forall v\colon \mathrm{k}.\sigma \Rightarrow \mathrm{prop}$ in signature.

Signatures

$\emptyset \vdash \emptyset \ \mathrm{sig}$

$$\text{add-K}\ \frac{\vdash \Sigma\ \mathrm{sig}}{\vdash \Sigma, \mathrm{K}\colon \mathrm{kind}\ \mathrm{sig}}\ \mathrm{K} \notin \mathrm{DC}(\Sigma) \quad \text{add-}C\ \frac{\vdash_\Sigma \mathrm{k}\colon \mathrm{kind}}{\vdash \Sigma, C\colon \mathrm{k}\ \mathrm{sig}}\ C \notin \mathrm{DC}(\Sigma)$$

$$\text{add-S}\ \frac{\vdash_\Sigma \mathrm{k}\colon \mathrm{kind}}{\vdash \Sigma, \mathrm{S}\colon \mathrm{k}\Rightarrow \mathrm{scheme}\ \mathrm{sig}}\ \mathrm{S} \notin \mathrm{DC}(\Sigma) \quad \text{add-}c\ \frac{\emptyset \vdash_\Sigma \sigma\colon \mathrm{scheme}}{\vdash \Sigma, c\colon \sigma\ \mathrm{sig}}\ c \notin \mathrm{DC}(\Sigma)$$

Theories

$$\emptyset\ \frac{\vdash \Sigma\ \mathrm{sig}}{\vdash_\Sigma \emptyset\ \mathrm{theory}} \quad \text{add}\ \frac{\vdash_\Sigma Th\ \mathrm{theory} \qquad \emptyset \vdash_\Sigma e_1\colon \sigma \qquad \emptyset \vdash_\Sigma e_2\colon \sigma}{\vdash_\Sigma Th, e_1 =_\sigma e_2\ \mathrm{theory}}$$

Kinds

$$\mathrm{K}\ \frac{\vdash \Sigma\ \mathrm{sig}}{\vdash_\Sigma \mathrm{K}\colon \mathrm{kind}}\ \Sigma(\mathrm{K}) = \mathrm{kind}$$

$$1\ \frac{\vdash \Sigma\ \mathrm{sig}}{\vdash_\Sigma 1\colon \mathrm{kind}} \quad \times\ \frac{\begin{array}{c}\vdash_\Sigma \mathrm{k}_1\colon \mathrm{kind}\\ \vdash_\Sigma \mathrm{k}_2\colon \mathrm{kind}\end{array}}{\vdash_\Sigma \mathrm{k}_1\times\mathrm{k}_2\colon \mathrm{kind}} \quad \Rightarrow\ \frac{\begin{array}{c}\vdash_\Sigma \mathrm{k}_1\colon \mathrm{kind}\\ \vdash_\Sigma \mathrm{k}_2\colon \mathrm{kind}\end{array}}{\vdash_\Sigma \mathrm{k}_1\Rightarrow\mathrm{k}_2\colon \mathrm{kind}}$$

Contexts

$$\emptyset\ \frac{\vdash \Sigma\ \mathrm{sig}}{\vdash_\Sigma \emptyset\ \mathrm{context}}$$

$$\text{add-}v\ \frac{\begin{array}{c}\vdash_\Sigma \Gamma\ \mathrm{context}\\ \vdash_\Sigma \mathrm{k}\colon \mathrm{kind}\end{array}}{\vdash_\Sigma \Gamma, v\colon \mathrm{k}\ \mathrm{context}}\ v \notin \mathrm{DV}(\Gamma) \quad \text{add-}x\ \frac{\begin{array}{c}\vdash_\Sigma \Gamma\ \mathrm{context}\\ \Gamma \vdash_\Sigma \sigma\colon \mathrm{scheme}\end{array}}{\vdash_\Sigma \Gamma, x\colon \sigma\ \mathrm{context}}\ x \notin \mathrm{DV}(\Gamma)$$

Constructors

$$v\ \frac{\vdash_\Sigma \Gamma\ \mathrm{context}}{\Gamma \vdash_\Sigma v\colon \mathrm{k}}\ \Gamma(v) = \mathrm{k} \quad C\ \frac{\vdash_\Sigma \Gamma\ \mathrm{context}}{\Gamma \vdash_\Sigma C\colon \mathrm{k}}\ \Sigma(C) = \mathrm{k}$$

$$1\mathrm{I}\ \frac{\vdash_\Sigma \Gamma\ \mathrm{context}}{\Gamma \vdash_\Sigma *\colon 1} \quad \times\mathrm{I}\ \frac{\Gamma \vdash_\Sigma u_1\colon \mathrm{k}_1 \qquad \Gamma \vdash_\Sigma u_2\colon \mathrm{k}_2}{\Gamma \vdash_\Sigma \langle u_1, u_2\rangle\colon \mathrm{k}_1\times\mathrm{k}_2} \quad \times\mathrm{E}\ \frac{\Gamma \vdash_\Sigma u\colon \mathrm{k}_1\times\mathrm{k}_2}{\Gamma \vdash_\Sigma \pi_i(u)\colon \mathrm{k}_i}$$

$$\Rightarrow\mathrm{I}\ \frac{\Gamma, v\colon \mathrm{k}_1 \vdash_\Sigma u\colon \mathrm{k}_2}{\Gamma \vdash_\Sigma (\lambda v\colon \mathrm{k}_1.u)\colon \mathrm{k}_1\Rightarrow\mathrm{k}_2} \quad \Rightarrow\mathrm{E}\ \frac{\Gamma \vdash_\Sigma u\colon \mathrm{k}_1\Rightarrow\mathrm{k}_2 \qquad \Gamma \vdash_\Sigma u_1\colon \mathrm{k}_1}{\Gamma \vdash_\Sigma u(u_1)\colon \mathrm{k}_2}$$

Constructor equality

Constructor equality is the congruence generated by the following rules

$$1.\eta\ \frac{\Gamma \vdash_\Sigma u\colon 1}{\Gamma \vdash_\Sigma * = u\colon 1}$$

$$\times.\beta\ \frac{\Gamma \vdash_\Sigma u_1\colon \mathrm{k}_1 \qquad \Gamma \vdash_\Sigma u_2\colon \mathrm{k}_2}{\Gamma \vdash_\Sigma \pi_i(\langle u_1, u_2\rangle) = u_i\colon \mathrm{k}_i} \quad \times.\eta\ \frac{\Gamma \vdash_\Sigma u\colon \mathrm{k}_1\times\mathrm{k}_2}{\Gamma \vdash_\Sigma \langle \pi_1(u), \pi_2(u)\rangle = u\colon \mathrm{k}_1\times\mathrm{k}_2}$$

$$\Rightarrow.\beta\ \frac{\Gamma, v\colon \mathrm{k}_1 \vdash_\Sigma u_2\colon \mathrm{k}_2 \qquad \Gamma \vdash_\Sigma u_1\colon \mathrm{k}_1}{\Gamma \vdash_\Sigma (\lambda v\colon \mathrm{k}_1.u_2)(u_1) = [u_1/v]u_2\colon \mathrm{k}_2} \quad \Rightarrow.\eta\ \frac{\Gamma \vdash_\Sigma u\colon \mathrm{k}_1\Rightarrow\mathrm{k}_2}{\Gamma \vdash_\Sigma (\lambda v\colon \mathrm{k}_1.u(v)) = u\colon \mathrm{k}_1\Rightarrow\mathrm{k}_2}$$

Type schemes

$$\text{type}\ \frac{\Gamma \vdash_\Sigma u\colon \Omega}{\Gamma \vdash_\Sigma u} \quad 1\ \frac{\vdash_\Sigma \Gamma \text{ context}}{\Gamma \vdash_\Sigma 1} \quad \mathrm{S}\ \frac{\Gamma \vdash_\Sigma u\colon \mathrm{k}}{\Gamma \vdash_\Sigma \mathrm{S}(u)}\ \Sigma(\mathrm{S}) = \mathrm{k}{\Rightarrow}\text{scheme}$$

$$\times\ \frac{\begin{array}{c}\Gamma \vdash_\Sigma \sigma_1\colon \text{scheme}\\ \Gamma \vdash_\Sigma \sigma_2\colon \text{scheme}\end{array}}{\Gamma \vdash_\Sigma \sigma_1{\times}\sigma_2\colon \text{scheme}} \quad \Rightarrow\ \frac{\begin{array}{c}\Gamma \vdash_\Sigma \sigma_1\colon \text{scheme}\\ \Gamma \vdash_\Sigma \sigma_2\colon \text{scheme}\end{array}}{\Gamma \vdash_\Sigma \sigma_1{\Rightarrow}\sigma_2\colon \text{scheme}} \quad \forall\ \frac{\Gamma, v\colon \mathrm{k} \vdash_\Sigma \sigma\colon \text{scheme}}{\Gamma \vdash_\Sigma (\forall v\colon \mathrm{k}.\sigma)\colon \text{scheme}}$$

Type scheme equality

Type scheme equality is the congruence induced by constructor equality

Terms

$$x\ \frac{\vdash_\Sigma \Gamma \text{ context}}{\Gamma \vdash_\Sigma x\colon \sigma}\ \Gamma(x) = \sigma \quad c\ \frac{\vdash_\Sigma \Gamma \text{ context}}{\Gamma \vdash_\Sigma c\colon \sigma}\ \Sigma(c) = \sigma$$

$$1\mathrm{I}\ \frac{\vdash_\Sigma \Gamma \text{ context}}{\Gamma \vdash_\Sigma *\colon 1} \quad \times\mathrm{I}\ \frac{\begin{array}{c}\Gamma \vdash_\Sigma e_1\colon \sigma_1\\ \Gamma \vdash_\Sigma e_2\colon \sigma_2\end{array}}{\Gamma \vdash_\Sigma \langle e_1, e_2\rangle\colon \sigma_1{\times}\sigma_2} \quad \times\mathrm{E}\ \frac{\Gamma \vdash_\Sigma e\colon \sigma_1{\times}\sigma_2}{\Gamma \vdash_\Sigma \pi_i(e)\colon \sigma_i}$$

$${\Rightarrow}\mathrm{I}\ \frac{\Gamma, x\colon \sigma_1 \vdash_\Sigma e\colon \sigma_2}{\Gamma \vdash_\Sigma (\lambda x\colon \sigma_1.e)\colon \sigma_1{\Rightarrow}\sigma_2} \quad {\Rightarrow}\mathrm{E}\ \frac{\Gamma \vdash_\Sigma e\colon \sigma_1{\Rightarrow}\sigma_2 \qquad \Gamma \vdash_\Sigma e_1\colon \sigma_1}{\Gamma \vdash_\Sigma e(e_1)\colon \sigma_2}$$

$$\forall\mathrm{I}\ \frac{\Gamma, v\colon \mathrm{k} \vdash_\Sigma e\colon \sigma}{\Gamma \vdash_\Sigma (\Lambda v\colon \mathrm{k}.e)\colon (\forall v\colon \mathrm{k}.\sigma)} \quad \forall\mathrm{E}\ \frac{\Gamma \vdash_\Sigma e\colon (\forall v\colon \mathrm{k}.\sigma) \qquad \Gamma \vdash_\Sigma u\colon \mathrm{k}}{\Gamma \vdash_\Sigma e[u]\colon [u/v]\sigma}$$

$$:\text{-eq}\ \frac{\Gamma \vdash_\Sigma e\colon \sigma_1 \qquad \Gamma \vdash_\Sigma \sigma_1 = \sigma_2\colon \text{scheme}}{\Gamma \vdash_\Sigma e\colon \sigma_2}$$

Term equality

Term equality is the congruence generated by constructor equality the following rules

$$\text{axiom}\ \frac{\vdash_\Sigma \mathit{Th} \text{ theory} \qquad \vdash_\Sigma \Gamma \text{ context}}{\Gamma \vdash_{\Sigma,\mathit{Th}} e_1 =_\sigma e_2}\ (e_1 =_\sigma e_2) \in \mathit{Th}$$

$$1.\eta\ \frac{\vdash_\Sigma \mathit{Th} \text{ theory} \qquad \Gamma \vdash_\Sigma e\colon 1}{\Gamma \vdash_{\Sigma,\mathit{Th}} * =_1 e}$$

$$\times.\beta\ \frac{\vdash_\Sigma \mathit{Th} \text{ theory} \qquad \Gamma \vdash_\Sigma e_1\colon \sigma_1 \qquad \Gamma \vdash_\Sigma e_2\colon \sigma_2}{\Gamma \vdash_{\Sigma,\mathit{Th}} \pi_i(\langle e_1, e_2\rangle) =_{\sigma_i} e_i}$$

$$\times.\eta\ \frac{\vdash_\Sigma \mathit{Th} \text{ theory} \qquad \Gamma \vdash_\Sigma e\colon \sigma_1{\times}\sigma_2}{\Gamma \vdash_{\Sigma,\mathit{Th}} \langle \pi_1(e), \pi_2(e)\rangle =_{\sigma_1\times\sigma_2} e}$$

$${\Rightarrow}.\beta\ \frac{\vdash_\Sigma \mathit{Th} \text{ theory} \qquad \Gamma, x\colon \sigma_1 \vdash_\Sigma e_2\colon \sigma_2 \qquad \Gamma \vdash_\Sigma e_1\colon \sigma_1}{\Gamma \vdash_{\Sigma,\mathit{Th}} (\lambda x\colon \sigma_1.e_2)(e_1) =_{\sigma_2} [e_1/x]e_2}$$

$${\Rightarrow}.\eta\ \frac{\vdash_\Sigma \mathit{Th} \text{ theory} \qquad \Gamma \vdash_\Sigma e\colon \sigma_1{\Rightarrow}\sigma_2}{\Gamma \vdash_{\Sigma,\mathit{Th}} (\lambda x\colon \sigma_1.e(x)) =_{\sigma_1\Rightarrow\sigma_2} e}$$

$$\forall.\beta\ \frac{\vdash_\Sigma \mathit{Th} \text{ theory} \qquad \Gamma, v\colon \mathrm{k} \vdash_\Sigma e\colon \sigma \qquad \Gamma \vdash_\Sigma u\colon \mathrm{k}}{\Gamma \vdash_{\Sigma,\mathit{Th}} (\Lambda v\colon \mathrm{k}.e)[u] = [u/v]e\colon [u/v]\sigma}$$

$$\forall.\eta\ \frac{\vdash_\Sigma \mathit{Th} \text{ theory} \qquad \Gamma \vdash_\Sigma e\colon (\forall v\colon \mathrm{k}.\sigma)}{\Gamma \vdash_{\Sigma,\mathit{Th}} (\Lambda v\colon \mathrm{k}.e[v]) =_{(\forall v:\mathrm{k}.\sigma)} e}$$

$$=\text{-eq}\ \frac{\Gamma \vdash_{\Sigma,\mathit{Th}} e_1 =_{\sigma_1} e_2 \qquad \Gamma \vdash_\Sigma \sigma_1 = \sigma_2\colon \text{scheme}}{\Gamma \vdash_{\Sigma,\mathit{Th}} e_1 =_{\sigma_2} e_2}$$