B.5 The definition of λ_m^*

λ_m^* is different from λ_m only in that the reduction rule (E-APPC1) is replaced by (E-APPC1*).

(E-APPC1*) $(e|_F)\ f \ \triangleright\ (e\ f|_{AT(F,S)})|_{AP(F,S)} \quad (f : S)$

(T-VAR) $\dfrac{}{x^S : S}$

(T-FUN) $\dfrac{e : S'}{\lambda x^S.\ e : [S \longrightarrow S']}$

(T-APP1) $\dfrac{e : F = [S_1 \longrightarrow S'_1, \ldots, S_n \longrightarrow S'_n] \quad f : S \quad AT(F,S) \text{ exists}}{e\ f : AP(F,S)}$

(T-LABEL1) $\dfrac{e_i : S_i \quad (i = 1, \ldots, n)}{\{l_1 = e_1, \ldots, l_n = e_n\} : \{l_1 : S_1, \ldots, l_n : S_n\}}$

(T-LABEL2) $\dfrac{e : \{l_1 : S_1, \ldots, l_n : S_n\}}{e.l_i : S_i}$

(T-COE) $\dfrac{e : S \quad S > S'}{e|_{S'} : S'}$

(T-LUB) $\dfrac{e : S \quad e' : S' \quad S \uparrow S'}{e \vee e' : S \vee S'}$

Here, $AT([S_1 \longrightarrow S'_1, \ldots, S_n \longrightarrow S'_n], T)$ is $S_{\phi(1)} \vee \ldots \vee S_{\phi(l)}$, where $\phi(i)$ satisfies $S_{\phi(i)} < T$. If no S_j satisfies $S_j < T$, it is undefined.

$AP([S_1 \longrightarrow S'_1, \ldots, S_n \longrightarrow S'_n], T)$ is $S'_{\phi(1)} \vee \ldots \vee S'_{\phi(l)}$, where $\phi(i)$ satisfies $S_{\phi(i)} < T$. If no S_j satisfies $S_j < T$, it is undefined.

B.4 Reduction Rules

(E-LABEL) $\{l_1 = e_1, \ldots, l_n = e_n\}.l_i \ \triangleright\ e_i$

(E-APP) $(\lambda x^S.\ e)\ f \ \triangleright\ e[x^S := f|_S]$

(E-APPL) $(e_1 \vee \ldots \vee e_n)\ f \ \triangleright\ e_{\phi(1)}\ f \vee \ldots \vee e_{\phi(l)}\ f$
(Where $e_{\phi(i)}$ are selected so that $AT(F_{\phi(i)}, S)$ exists for $F_{\phi(i)}$, the type of $e_{\phi(i)}$ and S, the type of f)

(E-APPC1) $(e|_F)\ f \ \triangleright\ (e\ f)|_{AP(F,S)}$ (Here S is the type of f)

(E-COE1) $\{l_1 = e_1, \ldots, l_n = e_n\}|_{\{l_1:S_1,\ldots,l_m:S_m\}} \ \triangleright\ \{l_1 = e_1|_{S_1}, \ldots, l_m = e_m|_{S_m}\}$

(E-COE2) $e|_N \ \triangleright\ e \quad (N = int, bool)$

(E-LUBR) $\{l_1{=}e_1, \ldots, l_m{=}e_m, \hat{l_1}{=}\hat{e_1}, \ldots \hat{l_k}{=}\hat{e_k}\} \vee \{l_1{=}f_1, \ldots, l_m{=}f_m, \check{l_1}{=}\check{f_1}, \ldots, \check{l_n}{=}\check{f_n}\} \ \triangleright$
$\{l_1 = e_1 \vee f_1, \ldots, l_m = e_m \vee f_m, \hat{l_1} = \hat{e_1}, \ldots \hat{l_k} = \hat{e_k}, \check{l_1} = \check{f_1}, \ldots, \check{l_n} = \check{f_n}\}$

(E-CONST1) $n \vee n \ \triangleright\ n$ (n is a constant in int or $bool$)

(E-CONST2) $n_1 \vee n_2 \ \triangleright\ E_{int} \quad (n_1 \neq n_2,\ n_1, n_2$ are constants in int.)

(E-CONST3) $n_1 \vee n_2 \ \triangleright\ E_{bool} \quad (n_1 \neq n_2,\ n_1, n_2$ are constants in $bool$.)

$int \vee int = int$

$bool \vee bool = bool$

$\{l_1 : S_1, \ldots, l_m : S_m, \hat{l}_1 : \hat{S}_1, \ldots, \hat{l}_n : \hat{S}_n\} \vee \{l_1 : T_1, \ldots, l_m : T_m, \check{l}_1 : \check{T}_1, \ldots \check{l}_k : \check{T}_k\} =$

$\{l_1 : S_1 \vee T_1, \ldots, l_m : S_m \vee T_m, \hat{l}_1 : \hat{S}_1, \ldots, \hat{l}_n : \hat{S}_n, \check{l}_1 : \check{T}_1, \ldots \check{l}_k : \check{T}_k\}$

$[S_1 \longrightarrow S'_1, \ldots, S_m \longrightarrow S'_m] \vee [S_{m+1} \longrightarrow S'_{m+1}, \ldots, S_n \longrightarrow S'_n] = [S_1 \longrightarrow S'_1, \ldots, S_n \longrightarrow S'_n].$

B.2 The Coercible Order

(I-REFL) $\dfrac{}{S > S}$

(I-TRANS) $\dfrac{S > S' \quad S' > S''}{S > S'''}$

(I-LABEL) $\dfrac{S_i > S'_i \quad (i = 1, \ldots, m) \quad n > m}{\{l_1 : S_1, \ldots, l_n : S_n\} > \{l : S'_1, \ldots, l_m : S'_m\}}$

(I-FUN) $\dfrac{T < S \quad T' > S'}{[T \longrightarrow T'] > [S \longrightarrow S']}$

(I-FLIST1) $\dfrac{}{[P_1, \ldots, P_n] > [P_i]}$

(I-FLIST2) $\dfrac{}{[S_1 \longrightarrow S'_1, S_2 \longrightarrow S'_2] > [S_1 \vee S_2 \longrightarrow S'_1 \vee S'_2]}$

(I-FLIST3) $\dfrac{[P_1, \ldots, P_n] > [P'_i] \quad (i = 1, \ldots, n)}{[P_1, \ldots, P_n] > [P'_1, \ldots, P'_m]}$

B.3 Expressions and Type Assignment

$$
\begin{array}{rcl}
e & ::= & x^S \mid c^{\top}_{int} \mid c^{\top}_{bool} \\
& & \mid \lambda x^S.\ e \mid e\ f \\
& & \mid \{l_1 = e_1, \ldots, l_n = e_n\} \mid e.l\ . \\
& & \mid e|_S \\
& & \mid e_1 \vee e_2
\end{array}
$$

Appendix B

The Definition of λ_m

B.1 Type Expressions

$$
\begin{array}{lcl}
S & ::= & int \mid bool \mid \{l_1 : S_1, \ldots, l_n : S_n\} \\
 & & \mid [P_1, \ldots, P_n] \quad (P_1, \ldots, P_n \text{ are pairwise compatible } (\uparrow).) \\
P & ::= & S \rightarrow S'
\end{array}
$$

(M-REFL) $\dfrac{}{N \uparrow N}$ (N = int, bool)

(M-LABEL) $\dfrac{S_i \uparrow T_i (i = 1, \ldots, m)}{\{l_1 : S_1, \ldots, l_m : S_m, \hat{l}_1 : \hat{S}_1, \ldots, \hat{l}_n : \hat{S}_n\} \uparrow \{l_1 : T_1, \ldots, l_m : T_m, \check{l}_1 : \check{T}_1, \ldots \check{l}_k : \check{T}_k\}}$

$(l_i\ (1 \leq i \leq m), \hat{l}_j\ (1 \leq j \leq n), \check{l}_h\ (1 \leq h \leq k)$ are distinct)

(M-FUN1) $\dfrac{\text{not } S_1 \uparrow S_2}{S_1 \rightarrow S_1' \uparrow S_2 \rightarrow S_2'}$

(M-FUN2) $\dfrac{S_1 \uparrow S_2 \text{ and } S_1' \uparrow S_2'}{S_1 \rightarrow S_1' \uparrow S_2 \rightarrow S_2'}$

(M-FLIST) $\dfrac{P_i \uparrow P_j' (i = 1, \ldots, n,\ j = 1, \ldots, m)}{[P_1, \ldots, P_n] \uparrow [P_1', \ldots, P_m']}$

When $S_1 \uparrow S_2$, we define $S_1 \vee S_2$, the merge of S_1 and S_2 as an abbreviation for a type expression as follows:

(T-VAR) $$\frac{}{x^S : S}$$

(T-FUN) $$\frac{e : S'}{\lambda x^S.\ e : S \longrightarrow S'}$$

(T-APP) $$\frac{e : S \longrightarrow S' \quad f : T \quad T > S}{e\ f : S'}$$

(T-LABEL1) $$\frac{e_i : S_i}{\{l_1 = e_1, \ldots, l_n = e_n\} : \{l_1 : S_1, \ldots, l_n : S_n\}}$$

(T-LABEL2) $$\frac{e : \{l_1 : S_1, \ldots, l_n : S_n\}}{e.l_i : S_i}$$

(T-COE) $$\frac{e : T \quad T > S}{e|_S : S}$$

A.4 Reduction Rules

(E-LABEL) $\{l_1 = e_1, \ldots, l_n = e_n\}.l_i \ \triangleright\ e_i$

(E-APP) $(\lambda x^S.\ e)\ f \ \triangleright\ e[x^S := f|_S]$

(E-APPC) $(e|_{S \longrightarrow S'})\ f \ \triangleright\ (e\ f)|_{S'}$

(E-COE1) $\{l_1 = e_1, \ldots, l_n = e_n\}|_{\{l_1:S_1,\ldots,l_m:S_m\}} \ \triangleright\ \{l_1 = e_1|_{S_1}, \ldots, l_m = e_m|_{S_m}\}$

(E-COE2) $e|_N \ \triangleright\ e \quad (N = int, bool)$

A.5 The Definition of λ_c^*

λ_c^* is different from λ_c only in that the reduction rule (E-APPC) is replaced by (E-APPC*).

(E-APPC*) $(e|_{S \longrightarrow S'})\ f \ \triangleright\ (e\ f|_S)|_{S'}$.

Appendix A

The Definition of $\lambda_{\mathcal{C}}$

A.1 Type Expressions

$$S ::= int \mid bool \mid \{l_1 : S_1, \ldots, l_n : S_n\} \mid S \rightarrow S'$$

A.2 Coercible Order

(I-REFL) $$\frac{}{S > S}$$

(I-TRANS) $$\frac{S > S' \quad S' > S''}{S > S''}$$

(I-LABEL) $$\frac{S_i > S'_i \ (i = 1, \ldots, m) \quad n > m}{\{l_1 : S_1, \ldots, l_n : S_n\} > \{l : S'_1, \ldots, l_m : S'_m\}}$$

(I-FUN) $$\frac{T < S \quad T' > S'}{T \rightarrow T' > S \rightarrow S'}$$

A.3 Expressions and Type Assignment

$$\begin{array}{rcl} e & ::= & x^S \mid c_{int} \mid c_{bool} \\ & & \mid \lambda x^S.\ e \mid e\ f \\ & & \mid \{l_1 = e_1, \ldots, l_n = e_n\} \mid e.l \\ & & \mid e|_S \end{array}$$

[Rey81] J. C. Reynolds. Using category theory to design implicit conversions and generic operators. In *Lecture Notes in Computer Science, 94*. Springer-Verlag, 1981.

[Rey83] J. C. Reynolds. Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pp. 513–523. North-Holland, 1983.

[Rey91] John C. Reynolds. The coherence of languages with intersection types. In *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pp. 657–700, 1991.

[Sco76] D. Scott. Data types as lattices. *SIAM Journal of Computing*, Vol. 5, pp. 522–587, 1976.

[SP82] M. Smyth and G. Plotokin. The category-theoretic solution of recursive domain equations. *SIAM J. Comput.*, Vol. 11, No. 4, pp. 761–783, 1982.

[Sto77] J. Stoy. *Denotational Semantics*. MIT Press, 1977.

[Tak92] Masako Takahashi. Parallel reductions in λ-calculus. Technical Report C-103, Tokyo Institute of Technology, 1992.

[TBG91] A. Tarlecki, R. Burstall, and J. Goguen. Some fundamental algebraic tools for the semantics of computation: Part3. indexed categories. *Theoretical Computer Science*, Vol. 91, pp. 239–264, 1991.

[Tsu90] Hideki Tsuiki. A denotational model of type inheritance and higher-order generics. *Computer Software*, Vol. 7, No. 4, pp. 60–78, 1990. (In Japanese.).

[Tsu92a] Hideki Tsuiki. Fixed points and the theory of programming langueges. *Journal of Information Processing Society of Japan*, Vol. 33, No. 4,, 1992. (In Japanese).

[Tsu92b] Hideki Tsuiki. A record calculus with a merge operator. (Submitted. In Japanese.), 1992.

[Tsu92c] Hideki Tsuiki. A reflective domain construction for type inheritance and higher-order generics. *Advances in Software Science and Technology*, Vol. 4,, 1992. English version of [Tsu90].

[Wan87] Mitchell Wand. Complete type inference for simple objects. In *Proceedings, IEEE Symposium on Logic in Computer Science*, pp. 37–44, 1987.

[Wan89] Mitchell Wand. Type inference for record concatenation and multiple inheritance. In *Proceedings, Fourth IEEE Annual Symposium on Logic in Computer Science*, pp. 92–97, 1989.

[HP91] R. Harper and B. Pierce. A record calculus based on symmetric concatenation. In *Proceedings, The Eighteenth Annual Symposium on Principles of Programming Languages*, pp. 131–142, 1991.

[Mil77] R. Milner. Fully abstract models of typed λ-calculus. *Theoret. Comput. Sci*, Vol. 4, pp. 1–22, 1977.

[MR91] QingMing Ma and J. C. Reynolds. Types, abstraction, and parametric polymorphism, part 2. In S. Brookes, M. Main, A. Melton, M. Mislove, and D.A. Schmidt, editors, *Mathematical Foundations of Programming Semantics, LNCS.* Springer-Verlag, 1991.

[OB88] A. Ohori and P. Buneman. Type inference in a database programming language. In *Proc. ACM Conf. on LISP and Functional Programming*, pp. 174–183, 1988.

[OB89] A. Ohori and P. Buneman. Static type inference for parametric classes. In *Proceedings, OOPSLA89: ACM conference on Object Oriented Programming Systems,Languages, and Applications*, pp. 445–455, 1989.

[OBBT89] A. Ohori, P. Buneman, and V. Breazu-Tannen. Database programming in machiavelli - a polymorphic language with static type inference. In *Proc. of ACM SIGMOD Conference*, pp. 46–57, Portland,Oregon, 1989.

[Oho90] A. Ohori. Semantics of types for database objects. *Theoretical Computer Science*, Vol. 76, pp. 53–91, 1990.

[PC89] J. Palsberg and W. Cook. A denotational semantics of inheritance and its correctness. In *Proceedings, OOPSLA89: ACM conference on Object Oriented Programming Systems,Languages, and Applications*, pp. 433–444, 1989.

[Pho92] W. Phoa. An introduction to fibrations, topos theory, the effective topos and modest sets. Technical Report ECS-LFCS-92-208, Edinburgh University, 1992.

[Pie92] B. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, CMU, 1992.

[Plo77] G. Plotokin. Lcf considered as a programming language. *Theoretical Computer Science*, Vol. 5, pp. 223–255, 1977.

[Rem89] D. Rémy. Typechecking records and variants in a natural extension of ml. In *Proceedings, The Sixteenth Annual Symposium on Principles of Programming Languages*, pp. 77–88, 1989.

[Rey74] J. C. Reynolds. Towards a theory of type structure. In *Proceedings, Colloque sur la Programmation, LNCS Vol19*, pp. 408–425, 1974.

[Car92] L. Cardelli. Extensible records in a pure calculus of subtyping. Technical Report 81, DEC SRC Report, 1992.

[CC90] F. Cardone and M. Coppo. Two extensions of curry's type inference system. In *Logic and Computer Science*, pp. 19–66. Academic Press, 1990.

[CCH+89] P. Channing, W. Cook, W. Hill, W. Olthoff, and J. Mitchell. F-bounded quantification for object-oriented programming. In *Proceedings, Fourth International Conference on Functional Programming Languages and Computer Architecture*, pp. 273–280, 1989.

[CDCV81] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Functional characters of solvable terms. *Z. Math. Logik Grundlagen Math.*, Vol. 27, pp. 45–58, 1981.

[CGL92] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. In *Proc. ACM Conf. on LISP and Functional Programming*, 1992.

[CGW89] T. Coquand, C. Gunter, and G. Winskel. Domain theoretic models of polymorphism. *Information and Computation*, Vol. 81, pp. 123–167, 1989.

[CHC89] W. Cook, W. Hill, and P. Channing. Inheritance is not subtyping. In *Proceedings, The Sixteenth Annual Symposium on Principles of Programming Languages*, 1989.

[CM91] L. Cardelli and J. Mitchell. Operations on records. *Mathematical Structures in Computer Science*, Vol. 1, No. 1, pp. 3–48, 1991.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, Vol. 17, No. 4, pp. 471–522, 1985.

[Gir71] J. Y. Girard. Une extension de l'interprétation de gödel á l'analyse et son application á l'élimination des coupures dans l'analyse et la théorie des types. In *Proc. 2nd Scandinavian Logic Symposium*, pp. 63–92. North Holland, 1971.

[GLT89] J. Y. Girard, Y. Lafont, and P. Taylor. *Proof and Types.* Cambridge University Press, 1989.

[Has90] R. Hasegawa. Categorical datatypes in parametric polymorphism. In *Fourth Asian Logic Conference*, Tokyo, 1990.

[Has91] R. Hasegawa. Parametricity of extensionally collapsed term models of polymorphism and their categorical properties. In *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pp. 495–512, 1991.

Bibliography

[Bar84] H. P. Barendregt. *The Lambda Calculus*. North Holland, 1984.

[BC90] G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings, OOPSLA90: ACM conference on Object Oriented Programming Systems, Languages, and Applications*, pp. 303–311, 1990.

[BCDC83] H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic*, Vol. 48, pp. 931–940, 1983.

[BDG+88] D. Bobrow, L. DiMichiel, R. Gabriel, S. Keene G., Kiczales D., and Moon. Common lisp object system specification: X3j13 document 88-002r. *SIGPLAN Notices*, Vol. 23,, 1988.

[Blo90] Bard Bloom. Can lcf be topped?, flat lattice models of typed λ-calculus. *Information and Computation*, Vol. 87, pp. 264–301, 1990.

[BMM88] K. B. Bruce, A. R. Meyer, and J. C. Mitchell. The semantics of second-order lambda calculus. *Information and Computation*, Vol. 59, pp. 76–134, 1988.

[Bru92] K. Bruce. A paradigmatic object-oriented language: Design, static typing and semantics. Technical Report CS-92-01, Williams College, 1992.

[BTGS90] V. Breazu-Tannen, C. A. Gunter, and A. Scedrov. Computing with coercions. In *Proceedings, ACM Conference on Lisp and Functional Programming*, pp. 44–59, 1990.

[BTGS91] V. Breazu-Tannen, C. A. Gunter, and A. Scedrov. Inheritance as implicit coercion. *Information and Computation*, Vol. 93, pp. 172–221, 1991.

[BW90] M. Barr and C. Wells. *Category Theory for Computing Science*. Prentice Hall, 1990.

[Car84] L. Cardelli. A semantics of multiple inheritance. In *Lecture Notes in Computer Science, 173*, pp. 51–67. Springer-Verlag, 1984.

one generic function. Our calculus λ_m was based on this idea, and it is expected that a practical object oriented language will be designed in the near future, which is based on generic functions and which ensures that the behavior of a generic function on each types has the same functionality.

	Unicity of Type	Subject Reduction	Church Rosser	Strong Normalization	Transitivity of Coercions
λ_c	◯	◯	◯	◯	◯
λ_c^*	◯	◯	◯	◯	◯
λ_m	◯	◯	◯	×	◯
λ_m^*	◯	◯	◯	◯	×

Figure 6.1: Properties of Calculi.

passed as an argument to another message.

There are two major extensions of λ_m to make it a practical programming language. One is to add the Y operator on each type and allow recursive programs. It is easy, syntactically, to add the Y operator to the calculus. However, it causes another kind of non-terminating programs, and we need to study the relation between these two kinds of nonterminations.

The other extension is to add polymorphism. Polymorphism is one of the most important features of object oriented languages, and there have been two directions of works on studying polymorphism of object oriented languages.

One is to use second order quantification called bounded polymorphism ([CW85]) to quantify over all the subtypes of a type, or to use F-bounded polymorphism ([CCH$^+$89]) to quantify over all the types that inherit a type. These methods are very powerful and allow polymorphic function arguments. However, in order to construct a consistent type system, the subsumption rule is needed and therefore exact type checking is impossible.

The other direction is to restrict polymorphism to record extensions and use row variables ([Wan87],[Wan89],[Rem89],[OB89], [OBBT89]). These systems do not use the subsumption rule, and therefore exact type checking is possible.

The author is trying to add record extension to λ_m. That is, λ_m serves as a language on which polymorphic system should be constructed. This just corresponds to simply typed lambda calculus on which polymorphism is usually considered. The author conjectures that this will succeed to an extent; with some restrictions on the pattern a row variable is used, a polymorphic function in an extension of λ_m can be viewed as an infinite merge of functions. Thus a generic function is a merge of a finite set of functions, and a polymorphic function is a merge of an infinite set of functions under some restrictions.

Most of the previous works on the foundation of object oriented computing formalized objects as self-contained entities, which include inside it all the methods applicable to itself. However, when we write a program using multiple inheritance, we frequently encounter name conflicts, that is, methods with the same name exist in two superclasses and they have different behavior. This causes program errors which are difficult to detect. In order to avoid such difficulties, all the methods with the same name need to have the same functionality. For this purpose, it is natural to consider that all the methods with the same name define

Chapter 6

Conclusions and Future Works

We have defined and studied a calculus λ_m with record types, implicit coercions, and a merge operator. A merge of functions corresponds to a "good natured" generic function whose behavior on a subtype is a refinement of that on a supertype.

λ_m has Subject Reduction, Unicity of Types, and Church-Rosser properties, and coercion is transitive in λ_m. A function type of λ_m has the property that the behavior of a function of type $[S \longrightarrow S']$ is not determined by its behavior on the type S; it may change its behavior on some type T bigger than S. Another point to note is that there is a non-normalizing term in λ_m. λ_m has a mathematically natural model, which is defined using a categorical structure called opfibration, in which a merge is modeled by a least upper bound.

We have also defined a calculus λ_m^*, which is obtained from λ_m by replacing one reduction rule on the application of a function defined through coercing another function. λ_m^* behaves differently to λ_m; it has the strong normalization property, and the behavior of a function of type $[S \longrightarrow S']$ is determined by its behavior on the type S. However, coercion is not transitive in λ_m^*, which is against our intuition on the notion of coercions. λ_m^* has Subject Reduction, Unicity of Types, and Church-Rosser properties.

Before introducing λ_m and λ_m^*, we defined subsystems λ_c and λ_c^*respectively, which do not have the notion of merge. λ_c and λ_c^* are equivalent and they have all good properties expected for a typed calculus with hierarchical type structure.

All the properties obtained are listed in Figure 6.1.

This is actually a negative result; when a simply typed lambda calculus is extended with coercion and the merge operator, we cannot expect both of the transitivity of coercions and strong normalization. Since λ_m and λ_m^* differs only in the reduction rule of the coercion of a function, when one restricts the language and prohibits functional arguments then one can obtain a language which is a sublanguage of both. Therefore, it has both properties. This just corresponds to object oriented languages, in which a message or a generic function cannot be passed as an argument. Therefore, this negative result shows the difficulty which one may encounter when extending an object oriented language so that a message can be

Proof: Prove the equality for all the rules of (E-λ_m). ∎

Proposition 88 (Identity of a coercion to itself) *When* $e : T$, $\mathcal{E}[\![e]\!](H) = \mathcal{E}[\![e|_T]\!](H)$ *for any environment* H.

Proof: Obvious from the structure of an M-domain. ∎

Proposition 89 (Transitivity of coercion) *When* $e : U, U > T > S$, $\mathcal{E}[\![e|_T|_S]\!](H) = \mathcal{E}[\![e|_S]\!](H)$ *for any environment* H.

Proof: Obvious from the structure of an M-domain. ∎

5.2.4 Discussion on the Reflectiveness of the Semantic Domain

Thus, we have given denotational semantics to λ_m. The difficult part of this semantics is that we need to construct a reflective M-domain. We need some explanation why we need a reflective domain. This fact is explained as follows. As we have studied, the meaning of a type S is given in $\mathcal{T}_m$ where m is the degree of S. Therefore, we expect that the meaning of $e : S$ is given in $\mathcal{A}^{-1}(\mathcal{E}_m^{\mathcal{T}}[\![S]\!])$. However, it is different from $\mathcal{A}^{-1}(\mathcal{E}^{\mathcal{T}}[\![S]\!])$, in which the meaning of e is given; $\mathcal{A}^{-1}(\mathcal{E}_m^{\mathcal{T}}[\![S]\!])$ is embedded into $\mathcal{A}^{-1}(\mathcal{E}_n^{\mathcal{T}}[\![S]\!])$ $(n > m)$ through the embedding $\phi_{m,n}$, but they are not equal. Actually, the meaning of e cannot be given in $\mathcal{A}^{-1}(\mathcal{E}_m^{\mathcal{T}}[\![S]\!])$; $\Phi_m(\Psi_m(\mathcal{E}[\![e]\!](H)))$ is not equal to $\mathcal{E}[\![e]\!](H)$.

This occurs because when $e : [S_1 \longrightarrow S_1', \ldots, S_n \longrightarrow S_n']$, e also acts to a type T which satisfy $T > S_i$ for some i, and T may have bigger degree than S_i. Since $e\ f$, $f : T$ may not be equal to $e\ f|_{S_i}$, $\Psi_m(\mathcal{E}[\![e]\!](H))$, which is, roughly, the meaning of the restriction of e to types less than degree m, does not fully explain the behavior of e.

This is surely related to the failure of strong normalization. The non-normalizing term $e\ e$ given in Section 3.3.5 has this kind of structure. Though $e : T = [S \longrightarrow S_1', \ldots, S_n \longrightarrow S_n']$ is applicable to itself with $T > S$, $e\ e$ and $e\ e|_S$ behave differently.

Proof: Let V be a directed subset of $\mathcal{D}$, with the least upper bound x. If $\mathcal{A}(x) \not> X$, then obviously $\vee(g(V)) = g(\vee(V))$. Suppose that $\mathcal{A}(x) > X$. Since $\mathcal{A}$ is continuous, $\vee\mathcal{A}(V) = \mathcal{A}(x) > X$. Since X is finite, $\mathcal{A}(y) > X$ for a $y \in V$. More generally, we can take a directed subset V' of V with the least upper bound x and $\mathcal{A}(y) > X$ for all $y \in V'$.

$$\begin{aligned} g(x) &= g(\vee_{y\in V'}\ y) \\ &= f((\vee_{y\in V'}\ y)|_X) \\ &= f(\vee_{y\in V'}\ y|_X) \quad (|_X \text{ is continuous }) \\ &= \vee_{y\in V'}\ f(y|_X) \\ &= \vee_{y\in V'}\ g(y) \\ &= \vee_{y\in V}\ g(y). \end{aligned}$$

∎

Proposition 82 *When e is an expression of type T which may include a variable x^S, the function from $\mathcal{A}^{-1}(\mathcal{E}^{\mathcal{T}}[\![S]\!])$ to $\mathcal{E}^{\mathcal{T}}[\![T]\!]$ defined as lambda$(d : \mathcal{A}^{-1}(\mathcal{E}^{\mathcal{T}}[\![S]\!])).\mathcal{E}[\![e]\!](H[d/x^S])$ is continuous.*

Proof: Induction on the structure of e. ∎

By composing the above three propositions, we have

Theorem 83 *lambda(d). if$(\mathcal{A}(d) > \mathcal{E}^{\mathcal{T}}[\![S]\!])$ then $\mathcal{E}[\![e]\!](H[d|_{\mathcal{E}^{\mathcal{T}}[\![S]\!]}/x^S])$ else $\perp_{\mathcal{D}}$ is a continuous function.*

Theorem 84 *If $e : S$, then $\mathcal{E}[\![e]\!](H) : \mathcal{E}^{\mathcal{T}}[\![S]\!]$ for every environment H.*

Proof: Induction on the structure of e. ∎

Theorem 85 *If $e : S$, then $\mathcal{E}[\![e]\!](H)$ is not $\perp_{\mathcal{D}}$.*

Proof: From the above theorem, we only need to show that $\mathcal{E}^{\mathcal{T}}[\![S]\!]$ is not $\perp_{\mathcal{T}}$, which is guaranteed by Theorem 80. ∎

Lemma 86 *$\mathcal{E}[\![e]\!](H[\mathcal{E}[\![f]\!](H)/x^S])$ is equal to $\mathcal{E}[\![e[x^S := f]]\!](H)$ for an expression f of type S.*

Proof: Induction on the structure of e ∎

Theorem 87 (Soundness of semantics) *If $e : S$ and $e \triangleright e'$, then $\mathcal{E}[\![e]\!](H) = \mathcal{E}[\![e']\!](H)$ for every environment.*

Theorem 80 *$\mathcal{E}^{\mathcal{T}}[\![S]\!]$ is not $\perp_{\mathcal{T}}$.*

Proof: It can be proved by induction on types. It can also be immediate from the above completeness theorem and the fact that the least type expression with respect to the coercible order does not exist. For example, int is only compared with int. ∎

5.2.3 Semantics of Expressions

An environment H is an assignment to each free variable x^S an element of $\mathcal{A}^{-1}(\mathcal{E}^{\mathcal{T}}[\![S]\!])$.

When $e : S$ in λ_m, we define $\mathcal{E}[\![e]\!](H) \in \mathcal{D}$ for which $\mathcal{E}[\![e]\!](H) : \mathcal{E}^{\mathcal{T}}[\![S]\!]$.

$$
\begin{aligned}
\mathcal{E}[\![x^S]\!](H) &= H(x^S) \\
\mathcal{E}[\![c_{int}]\!](H) &= \Psi(c_{int} \in M_B) \ \ (c_{int} = 0, 1, 2 \ldots) \\
\mathcal{E}[\![E_{int}]\!](H) &= \Psi(\top_{int}) \\
\mathcal{E}[\![c_{bool}]\!](H) &= \Psi(c_{bool} \in M_B) \ \ (c_{bool} = true, false \ldots) \\
\mathcal{E}[\![E_{bool}]\!](H) &= \Psi(\top_{bool}) \\
\mathcal{E}[\![\lambda x^S.\ e]\!](H) &= \Psi(lambda(d).\ if(\mathcal{A}(d) > \mathcal{E}^{\mathcal{T}}[\![S]\!])\ then\ \mathcal{E}[\![e]\!](H[d|_{\mathcal{E}^{\mathcal{T}}[\![S]\!]}/x^S]) \\
&\quad\ else\ \perp_{\mathcal{D}}) \\
\mathcal{E}[\![e\ f]\!](H) &= \Phi(\mathcal{E}[\![e]\!](H))\ \mathcal{E}[\![f]\!](H) \\
\mathcal{E}[\![\{l_1 = e_1, \ldots, l_n = e_n\}]\!](H) &= \Psi(lambda(l).\ if(l = l_i)\ then\ \mathcal{E}[\![e_i]\!](H)\ (i = 1, \ldots, n) \\
&\quad\ otherwise\ \perp_{\mathcal{D}}) \\
\mathcal{E}[\![e.l]\!](H) &= \Phi(\mathcal{E}[\![e]\!](H))(l) \\
\mathcal{E}[\![e|_S]\!](H) &= \mathcal{E}[\![e]\!](H)|_{\mathcal{E}^{\mathcal{T}}[\![S]\!]} \\
\mathcal{E}[\![e \vee f]\!](H) &= \mathcal{E}[\![e]\!](H) \vee \mathcal{E}[\![f]\!](H)
\end{aligned}
$$

Here, $H[d/x^S]$ is an environment which is identical to H except that the value of x^S is d.

For this definition, one needs that $lambda(d).\ if(\mathcal{A}(d) > \mathcal{E}^{\mathcal{T}}[\![S]\!])\ then\ \mathcal{E}[\![e]\!](H[d|_{\mathcal{E}^{\mathcal{T}}[\![S]\!]}/x^S])$ $else\ \perp_{\mathcal{D}}$ is a continuous function. This is proved as follows:

Proposition 81 *Let $M : \mathcal{D} \underset{\mathcal{S}}{\overset{\mathcal{A}}{\rightleftarrows}} \mathcal{T}$ be an M-domain, $X \in \mathcal{T}$ be a finite element of $\mathcal{T}$, and f be a continuous function from $\mathcal{A}^{-1}(X)$ to a cpo U. Then the function g from $\mathcal{D}$ to U defined as*

$$g = lambda(d).\ if(\mathcal{A}(d) > X)\ then\ f(d|_X) else \perp_{\mathcal{D}}$$

is continuous.

In this definition, we also used the fact that $\mathcal{E}_i^{\mathcal{T}}[\![S]\!]$ is a finite element when $i > \delta(S)$. This is ensured by the following lemma.

Lemma 75 *If (ϕ, ψ) is an embedding-projection pair from a cpo D to another cpo D', then ϕ preserves finite elements. Especially, $\phi_{m,i} : \mathcal{T}_m \longrightarrow \mathcal{T}_i$ and $\Phi_i^{\mathcal{T}} : \mathcal{T}_i \longrightarrow \mathcal{T}$ preserve finite elements.*

Proof: Let x be a finite element in D and U be a directed subset of D' which has the least upper bound $\phi(x)$. Then $\psi(U)$ is a directed set with the least upper bound $\psi(\phi(x)) = x$. Since x is finite, there exists a $y \in U$ for which $\psi(y) = x$. Since $y < \phi(x)$, $\phi(\psi(y)) = \phi(x) > y$. On the other hand, $\phi(\psi(y)) < y$. Therefore, $\phi(x) = y$. ∎

Thus, we have defined $\mathcal{E}_i^{\mathcal{T}}[\![S]\!]$ when S has degree i. Since $\mathcal{T}_i$ can be embedded into $\mathcal{T}$, $\mathcal{E}_i^{\mathcal{T}}$ can be extended to the semantics function $\mathcal{E}^{\mathcal{T}}$.

Proposition 76 *$\mathcal{E}^{\mathcal{T}}[\![S]\!]$ is a finite element of $\mathcal{T}$.*

Proof: It follows from the above two lemmas. ∎

5.2.2 Semantics of the Coercible Order

Theorem 77 *When $T > S$, then $\mathcal{E}^{\mathcal{T}}[\![T]\!] > \mathcal{E}^{\mathcal{T}}[\![S]\!]$ in $\mathcal{T}$.*

Proof: This is proved by checking each rule of the coercible order of λ_m. ∎

What is characteristic is that this semantics is complete with respect to the coercible order. That is, when $\mathcal{E}^{\mathcal{T}}[\![T]\!] > \mathcal{E}^{\mathcal{T}}[\![S]\!]$ is true in $\mathcal{T}$, $T > S$ is derivable in λ_m.

Lemma 78 *Let G be a function type expression. $\mathcal{E}_m^{\mathcal{T}}[\![G]\!](X) = \mathcal{E}_{m-1}^{\mathcal{T}}[\![AP(G,S)]\!]$ if $X = \mathcal{E}_{m-1}^{\mathcal{T}}[\![S]\!]$.*

Theorem 79 *When $\mathcal{E}^{\mathcal{T}}[\![T]\!] > \mathcal{E}^{\mathcal{T}}[\![S]\!]$ in $\mathcal{T}$, then $T > S$.*

Proof: Induction on $max(Deg(T), Deg(S))$. The only difficult case is when S and T are function types F and G. Suppose that $\mathcal{E}^{\mathcal{T}}[\![G]\!] > \mathcal{E}^{\mathcal{T}}[\![F]\!]$. $\mathcal{E}_i^{\mathcal{T}}[\![G]\!]$ and $\mathcal{E}_i^{\mathcal{T}}[\![F]\!]$ are step functions in $\mathcal{T}_i$ for an i bigger than $max(Deg(F), Deg(G))$. Because $\mathcal{E}_i^{\mathcal{T}}[\![G]\!] > \mathcal{E}_i^{\mathcal{T}}[\![F]\!]$, $\mathcal{E}_i^{\mathcal{T}}[\![G]\!](X) > \mathcal{E}_i^{\mathcal{T}}[\![F]\!](X)$ for every $X \in \mathcal{T}_{i-1}$. Especially, it holds for $X = \mathcal{E}_{i-1}^{\mathcal{T}}[\![S]\!]$ with $S \longrightarrow S' \in M(F)$. $\mathcal{E}_i^{\mathcal{T}}[\![F]\!](X) = \mathcal{E}_{i-1}^{\mathcal{T}}[\![S']\!]$ and $\mathcal{E}_i^{\mathcal{T}}[\![G]\!](X) = \mathcal{E}_{i-1}^{\mathcal{T}}[\![AP(G,S)]\!]$ from Lemma 78. Thus, we have $\mathcal{E}_{i-1}^{\mathcal{T}}[\![AP(G,S)]\!] > \mathcal{E}_{i-1}^{\mathcal{T}}[\![S']\!]$. Since $AP(G,S)$ and S' have degree less than i, from induction hypothesis, we have $AP(G,S) > S'$. Therefore, by (I-FUN'), $G > F$. ∎

Because the coercible order is used in reduction of λ_m, the coercible order must not be a simple collection of rules which are intuitively correct, but a collection of rules which will produce all the intuitively correct order relations. This completeness of type order to a semantics is justifies the rule of coercible order.

$$\text{(M-LABEL)}\ \frac{S_i \uparrow T_i (i=1,\ldots,j)}{\{l_1:S_1,\ldots,l_j:S_j,\hat{l}_1:\hat{S}_1,\ldots,\hat{l}_n:\hat{S}_n\} \uparrow \{l_1:T_1,\ldots,l_j:T_j,\check{l}_1:\check{T}_1,\ldots\check{l}_k:\check{T}_k\}}$$

Since S_i and T_i have degree less than m, $\mathcal{E}^{\mathcal{T}}_{m-1}[\![S_i]\!]$ and $\mathcal{E}^{\mathcal{T}}_{m-1}[\![T_i]\!]$ have the least upper bound $\mathcal{E}^{\mathcal{T}}_{m-1}[\![S_i \vee T_i]\!]$ for $i=1,\ldots,j$. Thus, the least upper bound $\mathcal{E}^{\mathcal{T}}_m[\![\{l_1:S_1,\ldots,l_j:S_j,\hat{l}_1:\hat{S}_1,\ldots,\hat{l}_n:\hat{S}_n\}]\!] \vee \mathcal{E}^{\mathcal{T}}_m[\![\{l_1:T_1,\ldots,l_j:T_j,\check{l}_1:\check{T}_1,\ldots\check{l}_k:\check{T}_k\}]\!]$ is the function R which maps

$$\begin{aligned}
R(l_i) &= \mathcal{E}^{\mathcal{T}}_{m-1}[\![S_i \vee T_i]\!] \ \ (i=1,\ldots,j)\\
R(\hat{l}_i) &= \mathcal{E}^{\mathcal{T}}_{m-1}[\![\hat{S}_i]\!] \ \ (i=i,\ldots,n)\\
R(\check{l}_i) &= \mathcal{E}^{\mathcal{T}}_{m-1}[\![\check{T}_i]\!] \ \ (i=1,\ldots,k)\\
R(l_i) &= \bot_{\mathcal{T}_{m-1}} \ \ (else)
\end{aligned}$$

Suppose that $P \uparrow P'$ by

$$\text{(M-FUN1)}\ \frac{\text{not } S_1 \uparrow S_2}{S_1 \to S_1' \uparrow S_2 \to S_2'}.$$

or

$$\text{(M-FUN2)}\ \frac{S_1 \uparrow S_2 \text{ and } S_1' \uparrow S_2'}{S_1 \to S_1' \uparrow S_2 \to S_2'}$$

When it is proved by (M-FUN2), since S_1, S_2, S_1', S_2' have degree less than m, $\mathcal{E}^{\mathcal{T}}_{m-1}[\![S_1 \vee S_2]\!]$ and $\mathcal{E}^{\mathcal{T}}_{m-1}[\![S_1' \vee S_2']\!]$ exists in this case. Thus the least upper bound of $\mathcal{E}^{\mathcal{T}}_m[\![S_1 \to S_1']\!]$ and $\mathcal{E}^{\mathcal{T}}_m[\![S_2 \to S_2']\!]$ is the function $B \in [\mathcal{T}_{m-1} \to \mathcal{T}_{m-1}]$ which maps

$$\begin{aligned}
B(X) &= \mathcal{E}^{\mathcal{T}}_{m-1}[\![S_1' \vee S_2']\!] \ \ (X > \mathcal{E}^{\mathcal{T}}_{m-1}[\![S_1 \vee S_2]\!])\\
B(X) &= \mathcal{E}^{\mathcal{T}}_{m-1}[\![S_1']\!] \ \ (X > \mathcal{E}^{\mathcal{T}}_{m-1}[\![S_1]\!])\\
B(X) &= \mathcal{E}^{\mathcal{T}}_{m-1}[\![S_2']\!] \ \ (X > \mathcal{E}^{\mathcal{T}}_{m-1}[\![S_2]\!])\\
B(X) &= \bot_{\mathcal{T}_{m-1}} \ \ (else)
\end{aligned}$$

Suppose that $S \uparrow T$ by

$$\text{(M-FLIST)}\ \frac{P_i \uparrow P_j' (i=1,\ldots,n,\ j=1,\ldots,k)}{[P_1,\ldots,P_n] \uparrow [P_1',\ldots,P_k']}$$

Then $\mathcal{E}^{\mathcal{T}}_m[\![P_i]\!]$ $(i=1,\ldots,n)$ and $\mathcal{E}^{\mathcal{T}}_m[P_j']$ $(j=1,\ldots,k)$ are all pairwise compatible. Because $\mathcal{T}_m$ is coherent, their upper bound exists, which is the function we want to construct. ∎

Note that $\mathcal{E}^{\mathcal{T}}_m[\![[S_1 \to S_1',\ldots,S_n \to S_n']]\!]$ is the function B which maps

$$\begin{aligned}
B(X) &= \mathcal{E}^{\mathcal{T}}_{m-1}[\![S_{\phi(1)'}]\!] \vee \ldots \vee \mathcal{E}^{\mathcal{T}}_{m-1}[\![S_{\phi(l)'}]\!]\\
&\quad \text{(where } \phi(i) \text{ are selected so that } X > S_{\phi(i)}).\\
B(X) &= \bot_{\mathcal{T}_{m-1}}\\
&\quad \text{(When no } S_j \text{ satisfies } X > S_j).
\end{aligned}$$

$$\begin{aligned}\mathcal{E}_0^{\mathcal{T}}[\![int]\!] &= int \\ \mathcal{E}_0^{\mathcal{T}}[\![bool]\!] &= bool.\end{aligned}$$

In the following, since a type in $\mathcal{T}_i$ can be embedded into $\mathcal{T}_m$ through $\phi_{i,m}^{\mathcal{T}}$ when $i < m$, we will also write $\mathcal{E}_m^{\mathcal{T}}[\![S]\!]$ for $\phi_{i,m}(\mathcal{E}_i^{\mathcal{T}}[\![S]\!])$ when $\delta(S) = i$ and $i < m$.

For a record type $\{l_1 : S_1, \ldots, l_n : S_n\}$, suppose $\delta(S_i) = d_i$, and $m = Max(d_i) + 1 = \delta(\{l_1 : S_1, \ldots, l_n : S_n\})$. Then $\mathcal{E}_m^{\mathcal{T}}[\![\{l_1 : S_1, \ldots, l_n : S_n\}]\!] \in [L \longrightarrow \mathcal{T}_{m-1}]$ is defined as the function R from L to $\mathcal{T}_{m-1}$ which maps

$$\begin{aligned}R(l_i) &= \mathcal{E}_{m-1}^{\mathcal{T}}[\![S_i]\!] \ \ (i = 1, \ldots, n) \\ R(l) &= \perp_{\mathcal{T}_{m-1}} \ \ (otherwise).\end{aligned}$$

R maps only finitely many elements to non-bottom elements. Therefore, it belongs to $[L \longrightarrow \mathcal{T}_{m-1}]$, and is obviously a finite element.

For function types, first consider the semantics of a function component $S \longrightarrow S'$. Here again, we put $\delta(S) = d$, $\delta(S') = d'$, $m = max(d, d') + 1$, Then $\mathcal{E}_m^{\mathcal{T}}[\![S \longrightarrow S']\!]$ is given as the function B in $[\mathcal{T}_{m-1} \longrightarrow \mathcal{T}_{m-1}]$ which maps

$$\begin{aligned}B(X) &= \mathcal{E}_{m-1}^{\mathcal{T}}[\![S']\!] \quad (\text{if } X > \mathcal{E}_{m-1}^{\mathcal{T}}[\![S]\!]) \\ B(X) &= \perp_{\mathcal{T}_{m-1}} \quad (\text{if } X \not> \mathcal{E}_{m-1}^{\mathcal{T}}[\![S]\!])\end{aligned}$$

Note that these step functions are bases of the poset $[\mathcal{T}_{m-1} \longrightarrow \mathcal{T}_{m-1}]$, and are finite elements. Note also that B is a continuous function and thus an element of $[\mathcal{T}_{m-1} \longrightarrow \mathcal{T}_{m-1}]$, because $\mathcal{E}_{m-1}^{\mathcal{T}}[\![S']\!]$ is a finite element. For function type expressions of the form $[P_1, \ldots, P_n]$, We define that

$$\mathcal{E}_m^{\mathcal{T}}[\![\ [P_1, \ldots, P_n]\ ]\!] = \mathcal{E}_m^{\mathcal{T}}[\![P_1]\!] \vee \ldots \vee \mathcal{E}_m^{\mathcal{T}}[\![P_n]\!].$$

For this definition, we need to prove that the least upper bound exists in $\mathcal{T}_m$.

Proposition 74 *If S and T are compatible, then $\mathcal{E}_m^{\mathcal{T}}[\![S]\!]$ and $\mathcal{E}_m^{\mathcal{T}}[\![T]\!]$ have the least upper bound, which is equal to $\mathcal{E}_m^{\mathcal{T}}[\![S \vee T]\!]$. Here, $m > Max(\delta(S), \delta(T))$.*

Note that the definition of $\mathcal{E}_i^{\mathcal{T}}$ and this proposition mutually depends.

Proof: Induction on the proof of the compatibility. Suppose that $S \uparrow T$ by (M-REFL), then trivial. Suppose that $S \uparrow T$ by

1. First coerce x_1 and x_2 to the common subtype Z.
2. Compare them and merge them in that type to obtain z, and adjust the difference of x_1 and x_2.
3. Reflect the result to x_1 and x_2 through fibration to obtain y_1 and y_2.
4. Because they share the same information with respect to their intersection, they can be merged.

This construction can be understood easily when x_i are record types. In that case, this construction says first merge the common slots, and then merge the records.

Actually, it seems that the fact that each coercion is a fibration is the essence of the existence of the merge in our model.

In order to construct an M-domain using the notion of coindexed category, we need to prove that the above properties are preserved in constructing record space and function space. The details are left open.

5.2 Semantics of λ_m

In this section, we give denotational semantics to λ_m.

5.2.1 Semantics of Type expressions

We define the degree δ of type expressions as follows:

$$\begin{aligned}
&\delta(int) = \delta(bool) = 0\\
&\delta(\{l_1 : S_1, \ldots, l_n : S_n\}) = Max(\delta(S_i)) + 1\\
&\delta([S_1 \longrightarrow S'_1, \ldots, S_n \longrightarrow S'_n]) = Max(\delta(S_i \longrightarrow S'_i))\\
&\delta(S \longrightarrow S') = Max(\delta(S), \delta(S')) + 1,
\end{aligned}$$

Note that this δ is different from Deg in 3.3.1 in that we increase the degree also in constructing a record type.

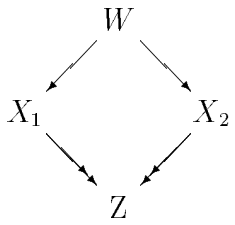
We will give the semantics of type expressions of degree i in $\mathcal{T}_i$. Recall that

$$\begin{array}{ccccccccc}
\mathcal{D}_0 & \underset{\psi^{\mathcal{D}}_{0,1}}{\overset{\phi^{\mathcal{D}}_{0,1}}{\rightleftarrows}} & \mathcal{D}_1 & \underset{\psi^{\mathcal{D}}_{1,2}}{\overset{\phi^{\mathcal{D}}_{1,2}}{\rightleftarrows}} & \ldots & \underset{\psi^{\mathcal{D}}_{n-1,n}}{\overset{\phi^{\mathcal{D}}_{n-1,n}}{\rightleftarrows}} & \mathcal{D}_n & \underset{\psi^{\mathcal{D}}_{n,n+1}}{\overset{\phi^{\mathcal{D}}_{n,n+1}}{\rightleftarrows}} & \ldots \\
\downarrow \mathcal{T}_0 & & \downarrow \mathcal{T}_1 & & \ldots & & \downarrow \mathcal{T}_n & & \ldots \\
\mathcal{T}_0 & \underset{\psi^{\mathcal{T}}_{0,1}}{\overset{\phi^{\mathcal{T}}_{0,1}}{\rightleftarrows}} & \mathcal{T}_1 & \underset{\psi^{\mathcal{T}}_{1,2}}{\overset{\phi^{\mathcal{T}}_{1,2}}{\rightleftarrows}} & \ldots & \underset{\psi^{\mathcal{T}}_{n-1,n}}{\overset{\phi^{\mathcal{T}}_{n-1,n}}{\rightleftarrows}} & \mathcal{T}_n & \underset{\psi^{\mathcal{T}}_{n,n+1}}{\overset{\phi^{\mathcal{T}}_{n,n+1}}{\rightleftarrows}} & \ldots
\end{array}$$

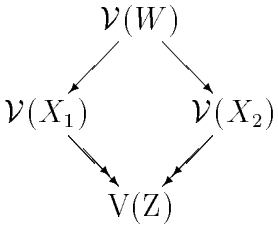
We first give the semantic functions $\mathcal{E}^{\mathcal{T}}_i$ $(i = 0, 1, 2, \ldots)$ which assign a finite element in $\mathcal{T}_i$ to a type expression of degree i. First, we define $\mathcal{E}^{\mathcal{T}}_0$ as follows:

Here, we do not give the definition of a fibration. See [Pho92] or [Pho92] for the definition. Though this theorem holds even the case $\mathcal{V}(X)$ is not an ordered set, we only prove the case $\mathcal{V}(X)$ are all ordered sets for simplicity.

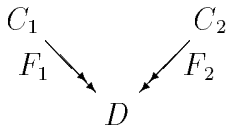
Proof: Let $x_1 \in \mathcal{V}(X_1)$ and $x_2 \in \mathcal{V}(X_2)$, and $W = X_1 \vee X_2$ be the least upper bound of X_1 and X_2. Since **Ind** is a lower semi-lattice, $Z = X_1 \wedge X_2$ exists. We put $x_i' = x_i|_Z$ $(i = 1, 2)$. In the following, i ranges over $\{1,\ 2\}$. Since $\mathcal{V}(Z)$ is complete, the product of x_1' and x_2' exists, which we write $z = x_1' \vee x_2'$. Since $\mathcal{V}(\sigma)$ for σ the order relation between X_i and Z is an opfibration, there is a y_i in X_i which satisfy $y_i|_Z = z$ and $y_i > x_i$, and has the universality that y_i is the least element in X_i which is coerced to z and which is bigger than x_i. Now, recall that $\mathcal{V}$ preserves pullbacks. The following is a pullback diagram.



Therefore,



is also a pullback diagram in the category **Cat**. A pullback of



in **Cat** is the category whose objects are $\{(x_1, x_2)|x_i \in obj(C_i), F_1(x_1) = F_2(x_2)\}$, and whose arrows are $\{(f_1, f_2)|f_i : x_i \longrightarrow y_i, F_1(f_1) = F_2(f_2)\}$. Since $y_1|_Z = y_2|_Z$, there is an element $w : \mathcal{V}(W)$, which satisfy $w|_{X_i} = y_i$.

w is the least upper bound of x_1 and x_2 in $\mathbf{FLAT}(\mathcal{V})$. First, $w > x_i$ in $\mathbf{FLAT}(\mathcal{V})$ because $w|_{X_i} > x_i$. Suppose that for $v \in \mathcal{V}(V)$, $v > x_i$ in $\mathbf{FLAT}(\mathcal{V})$. That is, $V > X_i$ and $v|_{X_i} > x_i$ in $\mathcal{V}(X_i)$. Since W is the least upper bound of X_i, $V > W$. Let $w' = v|_W$. We show that $w' > w$. First, $w' > x_i$ from the universality of $v|_W$. Let $y_i' = w'|_{X_i}$. $y_i' > x_i$. $y_i'|_Z > x_i'$. $y_1'|_Z = y_2'|_Z$ because both are equal to $w'|_Z$. From this, $y_i'|_Z > z$ Therefore, from the universality of y_i, $y_i' > y_i$. Note that $w' = (y_1', y_2')$ from the construction of W. Thus we get $w' > w$. ∎

This proof gives us a natural way how the merge is constructed. That is,

5. $\mathcal{V}(X)$ has the top element for each X, which is preserved by $\mathcal{V}(\sigma)$ for each morphism σ in **Ind**.

We will consider the fifth condition in detail, and find a sufficient condition under which this requirement holds. That is, let $\mathbf{FLAT}(\mathcal{V}) = \mathcal{A} : \mathcal{D} \longrightarrow \mathcal{T}$. When $\mathcal{A}(x) \vee \mathcal{A}(x')$ exists in **Ind**, then $x \vee x'$ exists in $\mathcal{D}$.

It says the existence of a limit in $\mathcal{D}$ under some conditions. As for colimit, there is a general theory which guarantee the existence of colimits in $\mathbf{FLAT}(\mathcal{V})$ ([TBG91]). That is, if every colimit exists in **Ind** and $\mathcal{V}(X)$ for every X, and $\mathcal{V}(\sigma)$ preserves colimit for each $\sigma : X \longrightarrow Y$, then colimit exists in $\mathbf{FLAT}(\mathcal{V})$. However, unfortunately, the corresponding condition for limit do not hold. Instead, we can find the following condition in [TBG91].

Fact 5 (Theorem 2 of [TBG91]) *If* $\mathcal{V} : \mathbf{Ind} \longrightarrow \mathbf{Cat}$ *is a coindexed category such that*

1. **Ind** *is complete,*
2. $\mathcal{V}(X)$ *is complete for all* $X \in \mathbf{Ind}$*,*
3. $\mathcal{V}(\sigma)$ *has right adjoint for each* $\sigma : X \longrightarrow Y$ *in* **Ind***,*

then $FLAT(\mathcal{V})$ *is complete.*

We cannot use this theorem, because in our situation, right adjoint does not exist to $\mathcal{V}(\sigma)$. For example, consider the coindexed category $\mathcal{V}_B : \mathcal{T}_B \longrightarrow \mathbf{Cat}$ in the previous section. $\mathcal{T}_B$ includes $int > null$, and the corresponding coercion function from $\mathcal{V}(int)$ to $\mathcal{V}(null)$ maps every element to nil. If there is a right adjoint to this coercion function, then it needs to map the only element in $\mathcal{V}(null)$ to the least element of $\mathcal{V}(int)$. However, $\mathcal{V}(int)$ does not have a least element. Therefore, the coercion function does not have the right adjoint. We need to consider another condition which guarantee that there is a least upper bound in $\mathcal{D}$.

Theorem 73 *If* $\mathcal{V} : \mathbf{Ind} \longrightarrow \mathbf{Cat}$ *is a coindexed category such that,*

1. **Ind** *is a lower semi-lattice,*
2. $\mathcal{V}(X)$ *is complete for all* $X \in \mathbf{Ind}$*,*
3. $\mathcal{V}(\sigma)$ *is a fibration for all* $\sigma : X \longrightarrow Y$*,*
4. *The functor* $\mathcal{V}$ *preserves pullback,*

then, in $\mathbf{FLAT}(\mathcal{V}) = \mathcal{F} : \mathbf{C} \longrightarrow \mathcal{T}$*,*

1. *if* $\mathcal{F}(x)$ *and* $\mathcal{F}(y)$ *have a product (i.e. least upper bound)* U *in* **Ind***, then* x *and* y *have a product* z *in* **C***, which satisfy* $\mathcal{F}(z) = U$*.*
2. **C** *has all equalizer.*

Proof: From Fact 3. ∎

Thus, we have constructed a reflective domain. This construction is similar to that of Scott domain $\mathcal{D}^\infty$ ([Sco76],[Sto77]). But we need to construct the limit in parallel; both in $\mathcal{T}$ and $\mathcal{D}$.

Note that the sequence constructed in 72 has the form

$$\begin{array}{ccccccccc}
M_0 & \underset{\psi_{0,1}}{\overset{\phi_{0,1}}{\rightleftarrows}} & M_1 & \underset{\psi_{1,2}}{\overset{\phi_{1,2}}{\rightleftarrows}} & \cdots & \underset{\psi_{n-1,n}}{\overset{\phi_{n-1,n}}{\rightleftarrows}} & M_n & \underset{\psi_{n,n+1}}{\overset{\phi_{n,n+1}}{\rightleftarrows}} & \cdots \\
\vdots & & \vdots & & & & \vdots & & \\
\mathcal{D}_0 & \underset{\psi^{\mathcal{D}}_{0,1}}{\overset{\phi^{\mathcal{D}}_{0,1}}{\rightleftarrows}} & \mathcal{D}_1 & \underset{\psi^{\mathcal{D}}_{1,2}}{\overset{\phi^{\mathcal{D}}_{1,2}}{\rightleftarrows}} & \cdots & \underset{\psi^{\mathcal{D}}_{n-1,n}}{\overset{\phi^{\mathcal{D}}_{n-1,n}}{\rightleftarrows}} & \mathcal{D}_n & \underset{\psi^{\mathcal{D}}_{n,n+1}}{\overset{\phi^{\mathcal{D}}_{n,n+1}}{\rightleftarrows}} & \cdots \\
\downarrow \mathcal{T}_0 & & \downarrow \mathcal{T}_1 & & \cdots & & \downarrow \mathcal{T}_n & & \cdots \\
\mathcal{T}_0 & \underset{\psi^{\mathcal{T}}_{0,1}}{\overset{\phi^{\mathcal{T}}_{0,1}}{\rightleftarrows}} & \mathcal{T}_1 & \underset{\psi^{\mathcal{T}}_{1,2}}{\overset{\phi^{\mathcal{T}}_{1,2}}{\rightleftarrows}} & \cdots & \underset{\psi^{\mathcal{T}}_{n-1,n}}{\overset{\phi^{\mathcal{T}}_{n-1,n}}{\rightleftarrows}} & \mathcal{T}_n & \underset{\psi^{\mathcal{T}}_{n,n+1}}{\overset{\phi^{\mathcal{T}}_{n,n+1}}{\rightleftarrows}} & \cdots
\end{array}$$

with $M_0 = M_\perp$, and $M_{i+1} = F(M_i)$. $\psi_{0,1}$ the unique arrow to M_0, and $\phi_{0,1}$ its corresponding embedding, and $\psi_{i+1} = F(\psi_i), \phi_{i+1} = F(\phi_i)$. We will also write $\phi_{i,j}$ for the composition of $\phi_{i,i+1}, \phi_{i+1,i+2}, \ldots, \phi_{j-1,j}$, and so on.

M is the limit of the sequence $M_0 \overset{\psi_{0,1}}{\longleftarrow} M_1 \overset{\psi_{1,2}}{\longleftarrow} M_2 \overset{\psi_2}{\longleftarrow} \cdots$ in the category **Mad**, which is equal to the colimit of the sequence $M_0 \xrightarrow[\phi_{0,1}]{} M_1 \xrightarrow[\phi_{1,2}]{} M_2 \xrightarrow[\phi_2]{} \cdots$ in the category $\mathbf{Mad}^E$.

We write $\Phi_i : M_i \longrightarrow M$ and $\Psi_i : M \longrightarrow M_i$ for the homomorphisms equipped with the colimit.

5.1.7 Construction using Coindexed Categories

We have thus constructed the semantic domain M as an opfibration. However, at least to my intuition, a coindexed category gives us much straightforward understanding of types and values than an opfibration. Therefore, it is interesting to find a condition on a coindexed category under which the domain constructed through Grothendieck construction satisfy the conditions of an M-domain.

Let $\mathcal{V} : \mathbf{Ind} \longrightarrow \mathbf{Cat}$ be a coindexed category. If $\mathcal{V}$ satisfies the following five conditions, then $\mathbf{FLAT}(\mathcal{V})$ is a split opfibration which satisfies the first four conditions of an M-domains.

1. **Ind** is a poset with the bottom element $\perp^{\mathbf{Ind}}$.
2. $\mathcal{V}(X)$ is a poset for each object X of **Ind**.
3. $\mathcal{V}(\perp^{\mathbf{Ind}})$ contains only one element.
4. **Ind** is coherent.

Proposition 69 *1)The functor HOM* : $\mathbf{Mad}^{OP} \times \mathbf{Mad} \longrightarrow \mathbf{Mad}$ *defined above is an* **O***-functor.*

2)The product functor PROD : $\mathbf{Mad} \times \mathbf{Mad} \longrightarrow \mathbf{Mad}$ *is an* **O***-functor.*

3)The sum functor SUM : $\mathbf{Mad} \times \mathbf{Mad} \longrightarrow \mathbf{Mad}$ *is an* **O***-functor.*

4)The record functor $RCD_L : \mathbf{Mad} \longrightarrow \mathbf{Mad}$ *defined as* $RCD_L(M) = [L \longrightarrow M]$ *is an* **O***-functor.*

Proof: 1) Let $f_n = (f_n^{\mathcal{D}}, f_n^{\mathcal{T}}) : M_1 \longrightarrow M_2$ be an increasing ω-sequence in $\mathbf{Mad}^{OP}$, and $g_n = (g_n^{\mathcal{D}}, g_n^{\mathcal{T}}) : M_3 \longrightarrow M_4$ be one in **Mad**, We only have to prove the equality of $\mathcal{D}$-components, that is, $\sqcup_n (g_n^{\mathcal{D}} \cdot d \cdot f_n^{\mathcal{D}}) = (\sqcup_n g_n^{\mathcal{D}}) \cdot d \cdot (\sqcup_n f_n^{\mathcal{D}})$, which is obvious.

2),3) Easy. ∎

Corollary 70 *The functors* $RCD^E, HOM^E, PROD^E, SUM^E : \mathbf{Mad}^E \times \mathbf{Mad}^E \longrightarrow \mathbf{Mad}^E$ *are* ω*-continuous.*

Proof: HOM^E is a direct consequence of the above proposition and Fact 4. We can consider *PROD* be a functor from $\mathbf{ID}^{OP} \times (\mathbf{Mad} \times \mathbf{Mad})$ to **Mad**, and *SUM* be one from $\mathbf{ID}^{OP} \times (\mathbf{Mad} \times \mathbf{Mad})$ to **Mad**, where **ID** is the category with one object and one arrow. By Fact 4, $PROD^E$ is a functor from $\mathbf{Mad}^E \times \mathbf{Mad}^E \longrightarrow \mathbf{Mad}^E$, and SUM^E is one $\mathbf{Mad}^E \times \mathbf{Mad}^E \longrightarrow \mathbf{Mad}^E$. ∎

Proposition 71 *The functor* $F : \mathbf{Mad}^E \longrightarrow \mathbf{Mad}^E$ *defined as:*

$$F(M) = M_B + [M \to M] + [L \longrightarrow M]$$
$$F(f : M_1 \longrightarrow M_2) = SUM^E(SUM^E(id_{M_B},\ HOM^E(f, f)),\ [L \longrightarrow f])$$

is ω*-continuous.*

Theorem 72 *Let* $M_\perp$ *be the initial object of* $\mathbf{Mad}^E$*, and* f *be the unique morphism from* $\perp_{\mathbf{Mad}}$ *to* $F(\perp_{\mathbf{Mad}})$*. Then,* M*, the colimit of* $\Delta = (F^n(\perp_{\mathbf{Mad}}), F^n(f))$ *is isomorphic to* $M_B + [M \to M] + [L \longrightarrow M]$*; that is, there are additive homomorphisms*

$$\begin{array}{lrcl} \Phi = (\Phi^{\mathcal{D}}, \Phi^{\mathcal{T}}) : & M & \longrightarrow & M_B + [M \to M] + [L \longrightarrow M] \\ \Psi = (\Psi^{\mathcal{D}}, \Psi^{\mathcal{T}}) : & M_B + [M \to M] + [L \longrightarrow M] & \longrightarrow & M \end{array}$$

which satisfy

$$\begin{array}{ll} \Phi \cdot \Psi & = id_{(M_B + [M \to M] + [L \longrightarrow M])} \\ \Psi \cdot \Phi & = id_M. \end{array}$$

Definition 16 (Definition 10 of [SP82]) *Let* **K, L, M** *be* **O***-categories. A covariant functor* $T : \mathbf{K}^{OP} \times \mathbf{L} \longrightarrow \mathbf{M}$ *is an* **O***-functor if and only if it is* ω*-continuous on the hom-sets; that is, if* $f_n : A \longrightarrow B$ *is an increasing* ω*-sequence in* $\mathbf{K}^{OP}$ *and* $g_n : C \longrightarrow D$ *is one in* $\mathbf{L}$ *then* $T(\sqcup_n f_n, \sqcup_n g_n) = \sqcup_n T(f_n, g_n)$.

Fact 4 (Derived from Theorem 3 and Corollary to Theorem 2 of [SP82]) *Suppose a covariant functor* $T : \mathbf{K}^{OP} \times \mathbf{L} \longrightarrow \mathbf{M}$ *is an* **O***-functor, and every* ω^{OP}*-chain has a limit both in* $\mathbf{K}$ *and* $\mathbf{L}$*. Then the functor* $T^E : \mathbf{K}^E \times \mathbf{L}^E \longrightarrow \mathbf{M}^E$ *defined as* $T^E(A, B) = T(A, B)$ *for objects and* $T^E(f, g) = T((f^P)^{OP}, g)$ *for morphisms is* ω*-continuous.*

We will prove that a functor $F : \mathbf{Mad}^E \longrightarrow \mathbf{Mad}^E$ defined for objects as $F(M) = M_B + [M \to M]$ is ω-continuous using Fact 4, and that there exists an M-domain M which is isomorphic to $F(M)$ in $\mathbf{Mad}^E$ using Fact 3.

Proposition 68 *Let* $M_i : \mathcal{D}_i \underset{\mathcal{S}_i}{\overset{\mathcal{A}_i}{\rightleftarrows}} \mathcal{T}_i$ $(i = 1,2,3,4)$ *be M-domains and* $f = (f^{\mathcal{D}}, f^{\mathcal{T}}) : M_1 \longrightarrow M_2$, $g = (g^{\mathcal{D}}, g^{\mathcal{T}}) : M_3 \longrightarrow M_4$ *be additive homomorphisms.*

1)Define $\phi^{\mathcal{D}} : [\mathcal{D}_2 \to \mathcal{D}_3]_{M_2,M_3} \longrightarrow [\mathcal{D}_1 \to \mathcal{D}_4]_{M_1,M_4}$ *and* $\phi^{\mathcal{T}} : [\mathcal{T}_2 \to \mathcal{T}_3] \longrightarrow [\mathcal{T}_1 \to \mathcal{T}_4]$ *as follows:*

$$\begin{aligned} \phi^{\mathcal{D}}(d : [\mathcal{D}_2 \to \mathcal{D}_3]_{M_2,M_3}) &= g^{\mathcal{D}} \cdot d \cdot f^{\mathcal{D}} \\ \phi^{\mathcal{T}}(u : [\mathcal{T}_2 \to \mathcal{T}_3]) &= g^{\mathcal{T}} \cdot u \cdot f^{\mathcal{T}}. \end{aligned}$$

Then, $\phi = (\phi^{\mathcal{D}}, \phi^{\mathcal{T}})$ *is an additive homomorphism from* $[M_2 \to M_3]$ *to* $[M_1 \to M_4]$.

2)We can define a functor $HOM : \mathbf{Mad}^{OP} \times \mathbf{Mad} \longrightarrow \mathbf{Mad}$ *as follows:*

$$\begin{aligned} HOM(M_1, M_2) &= [M_1 \to M_2] \\ HOM(f, g) &= \phi. \end{aligned}$$

Proof: 1)Since $\phi^{\mathcal{D}}$ is continuous, we only have to prove that $\mathcal{A}_1\mathcal{A}_4^* \cdot \phi^{\mathcal{D}} = \phi^{\mathcal{T}} \cdot \mathcal{A}_2\mathcal{A}_3^*$. Suppose $d \in [\mathcal{D}_2 \to \mathcal{D}_3]_{M_2,M_3}$, $t \in \mathcal{T}_1$.

$$\begin{aligned} (\mathcal{A}_1\mathcal{A}_4^* \cdot \phi^{\mathcal{D}})(d)(t) &= \mathcal{A}_4(\phi^{\mathcal{D}}(d)(x)) && ;;\textit{Definition of } \mathcal{A}_1\mathcal{A}_4^* \\ &\quad \textit{where } x \in \mathcal{D}_1 \textit{ such that } \mathcal{A}_1(x) = t \\ &= \mathcal{A}_4(g^{\mathcal{D}} \cdot d \cdot f^{\mathcal{D}}(x)) && ;;\textit{Definition of } \phi^{\mathcal{D}} \\ &= (g^{\mathcal{T}} \cdot \mathcal{A}_3 \cdot d \cdot f^{\mathcal{D}})(x) && ;;g \textit{ is a homomorphism.} \end{aligned}$$

$$\begin{aligned} (\phi^{\mathcal{T}} \cdot \mathcal{A}_2\mathcal{A}_3^*)(d)(t) &= (g^{\mathcal{T}} \cdot \mathcal{A}_2\mathcal{A}_3^*(d) \cdot f^{\mathcal{T}})(t) && ;;\textit{Definition of } \phi^{\mathcal{T}} \\ &= (g^{\mathcal{T}} \cdot \mathcal{A}_2\mathcal{A}_3^*(d) \cdot \mathcal{A}_2)(f^{\mathcal{D}}(x)) && ;;f \textit{ is a homomorphism.} \\ &\quad \textit{where } x \in \mathcal{D}_1 \textit{ such that } \mathcal{A}_1(x) = t \\ &= (g^{\mathcal{T}} \cdot \mathcal{A}_3)(d(f^{\mathcal{D}}(x))) && ;;\textit{Definition of } \mathcal{A}_2\mathcal{A}_3^*. \end{aligned}$$

It is easy to check that $\phi^{\mathcal{D}}$ is additive. 2)Easy. ∎

Proof: 1) Let $M : \mathcal{D} \underset{\mathcal{S}}{\overset{\mathcal{A}}{\rightleftarrows}} \mathcal{T}$, $M' : \mathcal{D}' \underset{\mathcal{S}'}{\overset{\mathcal{A}'}{\rightleftarrows}} \mathcal{T}'$ be M-domains. The hom-set from M to M' is isomorphic to a subset of additive functions from $\mathcal{D}$ to $\mathcal{D}'$, which is a cpo. Composition of morphisms is ω-continuous because of the definition of the order. ∎

Definition 13 (Definition 6 of [SP82]) *Let* $\mathbf{K}$ *be an* $\mathbf{O}$*-category and let* $f : A \longrightarrow B, g : B \longrightarrow A$ *be morphisms such that* $g \cdot f = id_A$ *and* $f \cdot g \preceq id_B$*. Then, we call* (f, g) *a projection pair from* A *to* B*,* f *an embedding and* g *a projection.*

It can be proved that one half of a projection pair determines the other. When f is an embedding, we write f^P for the corresponding projection. We write $\mathbf{K}^E$ for the subcategory of $\mathbf{K}$ which has the same objects as $\mathbf{K}$ and which has only embeddings as morphisms.

Fact 1 (Theorem 1 of [SP82]) *Let* $\mathbf{K}$ *be an* $\mathbf{O}$*-category which has a terminal object,* $\perp$*, and in which every hom-set* $hom(A, B)$ *has a least element,* $\perp_{A,B}$*. Suppose too that composition is left-strict in the sense that for any* $f : A \longrightarrow B$ *we have* $\perp_{B,C} \cdot f = \perp_{A,C}$*. Then,* $\perp$ *is the initial object of* $\mathbf{K}^E$*.*

Proposition 66 $\mathbf{Mad}^E$ *has an initial object.*

Proof: $M_\perp$ is the terminal object of **Mad**. Easy using Fact 1. ∎

Fact 2 (Theorem 2 of [SP82]) *Let* $\mathbf{K}$ *be an* $\mathbf{O}$*-category and* $\Delta = (A_i, f_i)$ *be an* ω*-chain in* $\mathbf{K}^E$*. Then,* $\Delta^{Rev} = (A_i, f_i^P)$ *is an* ω^{op}*-chain in* $\mathbf{K}$*. If* Δ^{Rev} *has a limit in* $\mathbf{K}$*, then* Δ *has a colimit in* $\mathbf{K}^E$*.*

Definition 14 (Definition 3 of [SP82]) *A category* $\mathbf{K}$ *is an* ω*-category if and only if it has an initial object and every* ω*-chain has a colimit.*

Proposition 67 $\mathbf{Mad}^E$ *is an* ω*-category.*

Definition 15 (Definition 4 of [SP82]) *Let* $F : \mathbf{K} \longrightarrow \mathbf{L}$ *be a functor. It is* ω*-continuous if and only if it preserves* ω*-colimits, that is, when* $\Delta = (A_i, f_i)$ *is an* ω*-chain and* (A, g_i) *is its limit, then* $(F(A), F(g_i))$ *is the limit of* $F(\Delta) = (F(A_i), F(f_i))$*.*

Fact 3 (Derived from Lemma1, Lemma2 of [SP82]) *Let* $\mathbf{K}$ *be an* ω*-category and let* $F : \mathbf{K} \longrightarrow \mathbf{K}$ *be an* ω*-continuous functor. Let* $1_\mathbf{K}$ *be the initial object of* $\mathbf{K}$*, and* f *be the unique morphism from* $1_\mathbf{K}$ *to* $F(1_\mathbf{K})$*. Define the* ω*-chain* Δ *to be* $(F^n(1_\mathbf{K}), F^n(f))$ *and* (A, g_i) *to be the colimit of* Δ*. Then* $F(A) \cong A$*.*

$$1_\mathbf{K} \xrightarrow{\quad f \quad} F(1_\mathbf{K}) \xrightarrow{\quad F(f) \quad} F^2(1_\mathbf{K}) \xrightarrow{\quad F^2(f) \quad} \cdots\cdots \longrightarrow A$$

$M_i, i = 0,1,2,\ldots)$ be an ω^{op}-chain with f_i additive homomorphisms. Then Δ has the limit $(M, g_i : M \longrightarrow M_i)$.

Proof: $(\mathcal{D}_i, f_i^{\mathcal{D}}), (\mathcal{T}_i, f_i^{\mathcal{T}})$ are ω^{op}-chains in the category of cpo. Therefore, there exist their limits $(\mathcal{D}, g_i^{\mathcal{D}} : \mathcal{D} \longrightarrow \mathcal{D}_i)$ and $(\mathcal{T}, g_i^{\mathcal{T}} : \mathcal{T} \longrightarrow \mathcal{T}_i)$. $\mathcal{D}$ and $\mathcal{T}$ are actually constructed as follows:

$$\begin{aligned}\mathcal{D} &= \left\{< d_0, d_1, \ldots > | d_n \in \mathcal{D}_n,\ f_n^{\mathcal{D}}(d_{n+1}) = d_n\ (n = 0,1,2,\ldots)\right\}\\ \mathcal{T} &= \left\{< t_0, t_1, \ldots > | t_n \in \mathcal{T}_n,\ f_n^{\mathcal{T}}(t_{n+1}) = t_n\ (n = 0,1,2,\ldots)\right\}.\end{aligned}$$

Since $(\mathcal{A}_i)$ is a function between ω^{op}-chains, there exists its limit $\mathcal{A}$ from $\mathcal{D}$ to $\mathcal{T}$. $\mathcal{A}$ is actually defined as follows:

$$\mathcal{A}(< d_0, d_1, \ldots >) =< \mathcal{A}_0(d_0), \mathcal{A}_1(d_1), \ldots > .$$

Because sections form a function $(\mathcal{S}_i)$ between ω^{op}-chains, there exists their limit $\mathcal{S}$ from $\mathcal{T}$ to $\mathcal{D}$. From the universality of limit, $\mathcal{A} \circ \mathcal{S}$ is identity.

Because f_i are additive, every least upper bound is preserved by f_i. Suppose that A and $B \in \mathcal{T}$ has the least upper bound, $a : A$, and $b : B$. $g_i^{\mathcal{D}}(a) : g_i^{\mathcal{T}}(A)$, and $g_i^{\mathcal{D}}(b) : g_i^{\mathcal{T}}(B)$. Since A and B has the least upper bound, $g_i^{\mathcal{T}}(A)$ and $g_i^{\mathcal{T}}(B)$ have the least upper bound. Therefore, $g_i^{\mathcal{D}}(a) \vee g_i^{\mathcal{D}}(b)$ exists. Since least upper bounds are preserved by f_i, $f_i(g_{i+1}^{\mathcal{D}}(a) \vee g_{i+1}^{\mathcal{D}}(b)) = g_i^{\mathcal{D}}(a) \vee g_i^{\mathcal{D}}(b)$. Therefore, their limit $c \in \mathcal{D}$ exist. $c : A \vee B$. It is easy to show that c is the least upper bound of a and b. Other conditions can be checked easily. ∎

We define the category of M-domains with additive homomorphisms **Mad**.

5.1.6 The Construction of a Reflective M-domain

In this section, we construct an M-domain M which satisfies

$$M = M_B + [M \to M],$$

where $M_B : \mathcal{D}_B \underset{\mathcal{S}_B}{\overset{\mathcal{A}_B}{\rightleftarrows}} \mathcal{T}_B$ is a given M-domain. [SP82] gives general conditions under which such recursive domain equations are solvable. We construct an M-domain M using their results. Here, we refer to their results as "Facts" and omit proofs. See [SP82] for their proofs.

Definition 12 (Definition 5 of [SP82]) *A category,* **K**, *is an* **O**-*category if and only if (i) every hom-set is a poset in which every ascending ω-sequence has the l.u.b. and (ii)composition of morphisms is an ω-continuous operation with respect to this partial order.*

Proposition 65 **Mad** *is an* **O**-*category.*

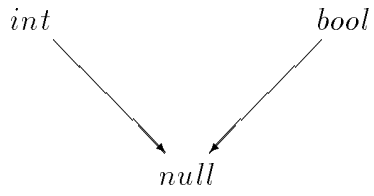
base change (see [TBG91]), $\mathcal{V}' \circ G$ is also a coindexed category over **Ind**. Since $\mathcal{V}$ and $\mathcal{V}' \circ G$ are two coindexed categories over **Ind**, we can consider a natural transformation g from $\mathcal{V}$ to $\mathcal{V}' \circ G$. In [TBG91], they call such a natural transformation a indexed functor. If we construct opfibrations from $\mathcal{V}$ and $\mathcal{V}' \circ G$ through Grothendieck construction, a coindexed functor extends to a functor from $\mathbf{FLAT}(\mathcal{V})$ to $\mathbf{FLAT}(\mathcal{V}')$. From that functor, we can construct a pair (g, G). However, the g constructed in this way preserves coercions. Therefore, the function space constructed in this way is too limited to interpret λ_m.

5.1.5 Some Properties of M-domains

When $M : \mathcal{D} \underset{\mathcal{S}}{\overset{\mathcal{A}}{\rightleftarrows}} \mathcal{T}$ and $M' : \mathcal{D}' \underset{\mathcal{S}'}{\overset{\mathcal{A}'}{\rightleftarrows}} \mathcal{T}'$ are M-domains, we will call a pair of continuous functions (g, G) a homomorphism from M to M' when it commutes with $\mathcal{A}$ and $\mathcal{A}'$, and also commutes with $\mathcal{S}$ and $\mathcal{S}'$. M-domains with homomorphisms form a category, which we call **Mdom**. It is actually a cartesian closed category with the exponent $[M \longrightarrow M]$.

We call a homomorphism (g, G) strict if g is strict. We call a homomorphism (g, G) additive if g is additive. When (g, G) is strict, G is also strict. When (g, G) is additive, G is also additive.

Let $\mathcal{T}_B = \{int, bool, null\}$ be the following ordered set.



Let $\mathcal{V}_B(int)$ and $\mathcal{V}_B(bool)$ be the upper semi-lattices in Section 5.1.1, and $\mathcal{V}_B(null)$ be the one-point semi-lattice with the element nil. Then, there are coercion functions from $\mathcal{V}_B(int)$ to $\mathcal{V}_B(null)$ and from $\mathcal{V}_B(bool)$ to $\mathcal{V}_B(null)$, which map all the elements to nil. Therefore, $\mathcal{V}_B : \mathcal{T}_B \longrightarrow \mathbf{Cat}$ is a coindexed category. Let $M_B : \mathcal{D}_B \underset{\mathcal{S}_B}{\overset{\mathcal{A}_B}{\rightleftarrows}} \mathcal{T}_B$ be the split opfibration $\mathbf{FLAT}(\mathcal{V}_B)$. M_B is an M-domain.

When M and M' are two M-domains, we can easily construct their smashed sum, the sum with the least elements of their value domains and type domains are identified. Let $M + M'$ be the smashed sum.

Let $M_\perp : \mathcal{D}_\perp \underset{\mathcal{S}_\perp}{\overset{\mathcal{A}_\perp}{\rightleftarrows}} \mathcal{T}_\perp$ denote the M-domain whose value domain $\mathcal{D}_\perp$ consists of only one element, and whose type domain $\mathcal{T}_\perp$ consists of only one element. $M_\perp$ is the terminal object of **Mdom**.

Next, we will see that every additive ω^{op}-chain $M_0 \xleftarrow{f_0} M_1 \xleftarrow{f_1} M_2 \xleftarrow{f_2} \cdots$ has a limit.

Proposition 64 *Let* $\Delta = (M_i, f_i)$ $(M_i : \mathcal{D}_i \underset{\mathcal{S}_i}{\overset{\mathcal{A}_i}{\rightleftarrows}} \mathcal{T}_i), f_i = (f_i^{\mathcal{D}}, f_i^{\mathcal{T}}), f_i : M_{i+1} \longrightarrow$

and F also commutes with the sections.

$$\begin{array}{ccc} \mathcal{D} & \xrightarrow{f} & \mathcal{D}' \\ \mathcal{S}\uparrow & & \uparrow\mathcal{S}' \\ \mathcal{T} & \xrightarrow[F]{} & \mathcal{T}' \end{array} ,$$

Suppose that g is a continuous function from $\mathcal{D}$ to $\mathcal{D}'$, G and F be continuous functions from $\mathcal{T}$ to $\mathcal{T}'$, and

$$\begin{array}{ccc} \mathcal{D} & \xrightarrow{g} & \mathcal{D}' \\ \mathcal{A}\downarrow & & \downarrow\mathcal{A}' \\ \mathcal{T} & \xrightarrow[G]{} & \mathcal{T}' \end{array} ,$$

commutes. That is, $(g, G) : G$ in an element of the opfibration $[M \longrightarrow M']$. Because there is a section $\mathcal{S}$ from $\mathcal{T}$ to $\mathcal{D}$, $\mathcal{A}$ is a surjective map on objects. We have that $G = G \circ \mathcal{A} \circ \mathcal{S} = \mathcal{A}' \circ g \circ \mathcal{S}$. It means that the G which commutes with g is uniquely determined. Thus, we simply write (g, G) as g.

$[\mathcal{D} \longrightarrow \mathcal{D}']_{M,M'}$ is a poset with the bottom element the constant function to $\perp^{\mathcal{D}'}$, $[\mathcal{T} \longrightarrow \mathcal{T}']$ is a poset with the bottom element the constant function to $\perp^{\mathcal{T}'}$. $\mathcal{A}^{-1}(\perp^{[\mathcal{T} \longrightarrow \mathcal{T}']})$ only contains $\perp^{[\mathcal{D} \longrightarrow \mathcal{D}']_{M,M'}}$. $[\mathcal{T} \longrightarrow \mathcal{T}']$ is evidently coherent. There is a section $\mathcal{S}\mathcal{S}'^*$ which maps G to $\mathcal{S}' \circ G \circ \mathcal{A}$. $\mathcal{S}' \circ G \circ \mathcal{A}$ is continuous and thus is an element of $[\mathcal{D} \longrightarrow \mathcal{D}']_{M,M'}$. $\mathcal{S}\mathcal{S}'^*$ is also continuous.

For the least upper bound condition, suppose that $f : F$, $g : G$, and $F \vee G$ exists in $[\mathcal{T} \longrightarrow \mathcal{T}']$. For each S,$(F \vee G)(S) > F(S)$ and $(F \vee G)(S) > G(S)$. Therefore, $F(S) \vee G(S)$ exists because $\mathcal{T}'$ is coherent. Then $H(S) = F(S) \vee G(S)$ is a continuous function such that $F \vee G > H$. It follows that $(F \vee G)(S) = F(S) \vee G(S)$. Therefore, $f(x) \vee g(x)$ exists for each x, and it has type $F(S) \vee G(S)$. It is obvious that $h(x) = f(x) \vee g(x)$ is continuous. Therefore, $f \vee g$ exists.

Thus, when $M : \mathcal{D} \underset{\mathcal{S}}{\overset{\mathcal{A}}{\rightleftarrows}} \mathcal{T}$ and $M' : \mathcal{D}' \underset{\mathcal{S}'}{\overset{\mathcal{A}'}{\rightleftarrows}} \mathcal{T}'$ are M-domains, we have constructed a function space M-domain $[M \longrightarrow M'] = [\mathcal{D} \longrightarrow \mathcal{D}']_{M,M'} \underset{\mathcal{S}\mathcal{S}'^*}{\overset{\mathcal{A}\mathcal{A}'^*}{\rightleftarrows}} [\mathcal{T} \longrightarrow \mathcal{T}']$.

One thing to note about this definition is that a function does not need to preserve coercions. That is, when $f : F$ is an element of $[M \longrightarrow M']$, $f(x)|_{F(S)}$ need not be equal to $f(x|_S)$. The only thing we have is $f(x)|_{F(S)} > f(x|_S)$. This is actually the case in λ_m. See the example in Section 3.3.3. In this sense, for $(f, F) \in [M \longrightarrow M']$, f is not a cartesian functor from M to $F^\sharp(M')$ over **Ind**. The definition of a cartesian functor and $F^\sharp$ is found in [Pho92].

One can construct a similar notion using coindexed categories. Let $\mathcal{V} : \mathbf{Ind} \longrightarrow \mathbf{Cat}$ and $\mathcal{V}' : \mathbf{Ind}' \longrightarrow \mathbf{Cat}'$ be coindexed categories, and G be a functor from **Ind** to **Ind**$'$. Then by

We will construct it using the universality of the cleavage $g(x)|_{F(\mathcal{F}(x)),\nu_{\mathcal{F}(x)}}$. $g(y)|_{F(\mathcal{F}(y)),\nu_{\mathcal{F}(y)}} \circ g(k)$ is a morphism from $g(x)$ to n. $\mathcal{F}'(m) = F(\mathcal{F}(y))$. $\mathcal{F}'(n) = F(\mathcal{F}(x))$. $\mathcal{F}'(g(y)|_{F(\mathcal{F}(y)),\nu_{\mathcal{F}(y)}} \circ g(k)) = \nu_{\mathcal{F}(y)} \circ \mathcal{F}'(g(k)) = \nu_{\mathcal{F}(y)} \circ G(\mathcal{F}(k))$. On the other hand, $F(\mathcal{F}(k)) \circ \mathcal{F}'(g(x)|_{F(\mathcal{F}(x)),\nu_{\mathcal{F}(x)}}) = F(\mathcal{F}(k)) \circ \nu_{\mathcal{F}(x)}$. Since ν is a natural transformation from G to F,

$$\begin{array}{ccc} G(\mathcal{F}(x)) & \xrightarrow{\nu_{\mathcal{F}(x)}} & F(\mathcal{F}(x)) \\ {\scriptstyle G(\mathcal{F}(k))}\downarrow & & \downarrow{\scriptstyle F(\mathcal{F}(k))} \\ G(\mathcal{F}(y)) & \xrightarrow[\nu_{\mathcal{F}(y)}]{} & F(\mathcal{F}(y)) \end{array}$$

commutes. Thus, $\mathcal{F}'(g(y)|_{F(\mathcal{F}(y)),\nu_{\mathcal{F}(y)}} \circ g(k))$ factors through $\mathcal{F}'(g(x)|_{F(\mathcal{F}(x)),\nu_{\mathcal{F}(x)}})$. From the universality of $g(x)|_{F(\mathcal{F}(x)),\nu_{\mathcal{F}(x)}}$, There is a morphism from m to n, which we define $f(k)$.

f is a functor from $\mathbf{C}$ to $\mathbf{C}'$ from the second property of a cleavage and the cleavages $g(x)|_{F(\mathcal{F}(x)),\nu_{\mathcal{F}(x)}}$ form a natural transformation from g to f. With this natural transformation as a cleavage of g through ν, $[M \longrightarrow M'] = \mathcal{F}\mathcal{F}'^{*} : [\mathbf{C} \longrightarrow \mathbf{C}']_{M,M'} \longrightarrow [\mathbf{Ind} \longrightarrow \mathbf{Ind}']$ is a split opfibration. ∎

Let us consider the case $M : \mathcal{D} \underset{\mathcal{S}}{\overset{\mathcal{A}}{\rightleftarrows}} \mathcal{T}$ and $M' : \mathcal{D}' \underset{\mathcal{S}'}{\overset{\mathcal{A}'}{\rightleftarrows}} \mathcal{T}'$ are M-domains. $[M \longrightarrow M'] : \mathcal{A}\mathcal{A}'^{*} : [\mathcal{D} \longrightarrow \mathcal{D}']_{M,M'} \longrightarrow [\mathcal{T} \longrightarrow \mathcal{T}']$ is an opfibration from the above theorem. Since a functor from a poset to anothe poset is a monotonic function, $[\mathcal{T} \longrightarrow \mathcal{T}']$ is the poset of monotonic functions from $\mathcal{T}$ to $\mathcal{T}'$, $[\mathcal{D} \longrightarrow \mathcal{D}']_{M,M'}$ is the poset of pairs of monotonic functions $(f : \mathcal{D} \longrightarrow \mathcal{D}', F : \mathcal{T} \longrightarrow \mathcal{T}')$, for which

$$\begin{array}{ccc} \mathcal{D} & \xrightarrow{f} & \mathcal{D}' \\ {\scriptstyle \mathcal{A}}\downarrow & & \downarrow{\scriptstyle \mathcal{A}'} \\ \mathcal{T} & \xrightarrow[F]{} & \mathcal{T}' \end{array}$$

commutes. The fact that $[M \longrightarrow M']$ is an opfibration says that, a function $f : F$ coerced to G exists. This function $f|_G$ maps $x : \mathcal{D}$ to $f(x)|_{G(\mathcal{A}(x))}$. That is, first apply f and then coerce the result to the appropriate type determined by the function G.

For our purpose, we will also restrict that $[\mathcal{D} \longrightarrow \mathcal{D}']_{M,M'}$ and $[\mathcal{T} \longrightarrow \mathcal{T}]$ contain only continuous functions. This restriction does not change the fact that $[M \longrightarrow M']$ is an opfibration, because it is easily shown that every function $f|_G$ is also continuous. We can easily show that $|_G$ is continuous. We will also require that for $(f, F) \in [\mathcal{D} \longrightarrow \mathcal{D}']_{M,M'}$, f

is a monotonic function from $\mathcal{T}$ to $\mathcal{T}$. We also note that functions expressible in λ_m are all monotonic functions from $\mathcal{D}$ to $\mathcal{D}$. Thus, when two M-domains $M : \mathcal{D} \underset{\mathcal{S}}{\overset{\mathcal{A}}{\rightleftarrows}} \mathcal{T}$ and $M' : \mathcal{D}' \underset{\mathcal{S}'}{\overset{\mathcal{A}'}{\rightleftarrows}} \mathcal{T}'$ are given, we construct an M-domain $[M \longrightarrow M']$ whose type domain $[\mathcal{T} \longrightarrow \mathcal{T}']$ is, roughly, the set of monotonic functions from $\mathcal{T}$ to $\mathcal{T}'$ and whose value domain $[\mathcal{D} \longrightarrow \mathcal{D}']_{M,M'}$ is, roughly, the set of monotonic functions from $\mathcal{D}$ to $\mathcal{D}'$ which commutes with $\mathcal{A}$ and $\mathcal{A}'$. Since the corresponding construction in category theory for opfibrations does not seem to have appeared anywhere, we first generally define categorically the function space for opfibrations. Because this construction is so complicated, one may skip it if one is not interested in category theory.

Theorem 63 *Suppose that* $M = \mathcal{F} : \mathbf{C} \longrightarrow \mathbf{Ind}$ *and* $M' = \mathcal{F}' : \mathbf{C}' \longrightarrow \mathbf{Ind}'$ *be split opfibrations. Let* $[\mathbf{Ind} \longrightarrow \mathbf{Ind}']$ *be the functor category. Let* $[\mathbf{C} \longrightarrow \mathbf{C}']_{M,M'}$ *be the category whose objects are pairs of functors* $(f : \mathbf{C} \longrightarrow \mathbf{C}', F : \mathbf{Ind} \longrightarrow \mathbf{Ind}')$*, for which the following diagram commutes:*

$$\begin{array}{ccc} \mathbf{C} & \xrightarrow{f} & \mathbf{C}' \\ {\scriptstyle \mathcal{F}}\downarrow & & \downarrow{\scriptstyle \mathcal{F}'} \\ \mathbf{Ind} & \xrightarrow[F]{} & \mathbf{Ind}' \end{array},$$

and whose arrows are pairs of natural transformations $(\mu : g \longrightarrow f, \nu : G \longrightarrow F)$ *which satisfies* $\mathcal{F}'(\mu_x) = \nu_{\mathcal{F}(x)}$ *for all objects* x *in* $\mathbf{C}$*.*

$$\begin{array}{ccc} g(x) & \xrightarrow{\mu_x} & f(x) \\ {\scriptstyle \mathcal{F}'}\downarrow & & \downarrow{\scriptstyle \mathcal{F}'} \\ G(\mathcal{F}(x)) & \xrightarrow[\nu_{\mathcal{F}(x)}]{} & F(\mathcal{F}(x)), \end{array}$$

Let $\mathcal{F}\mathcal{F}'^* : [\mathbf{C} \longrightarrow \mathbf{C}']_{M,M'} \longrightarrow [\mathbf{Ind} \longrightarrow \mathbf{Ind}']$ *be the functor which takes the second component of a pair. Then* $[M \longrightarrow M'] = \mathcal{F}\mathcal{F}'^* : [\mathbf{C} \longrightarrow \mathbf{C}']_{M,M'} \longrightarrow [\mathbf{Ind} \longrightarrow \mathbf{Ind}']$ *is a split opfibration.*

Proof: Let (g, G) be an object of $[\mathbf{C} \longrightarrow \mathbf{C}']_{M,M'}$, and $\nu : G \longrightarrow F$ be a natural transformation. Then the functor f for which (f, F) is the target of the cleavage of (g, G) through ν is defined as follows.

Action on objects: $\nu_{\mathcal{F}(x)}$ is a morphism from $G(\mathcal{F}(x))$ to $F(\mathcal{F}(x))$. Since $G(\mathcal{F}(x))$ is equal to $\mathcal{F}'(g(x))$, we define $f(x)$ to be the target of the cleavage $g(x)|_{F(\mathcal{F}(x)),\nu_{\mathcal{F}(x)}}$.

Action on morphisms: Let $k : x \longrightarrow y$ in $\mathbf{C}$. We will construct a morphism from m the target of $g(x)|_{F(\mathcal{F}(x)),\nu_{\mathcal{F}(x)}}$, to n the target of $g(y)|_{F(\mathcal{F}(y)),\nu_{\mathcal{F}(y)}}$, which is mapped by $\mathcal{F}'$ to $F(\mathcal{F}(k))$.

Definition 11 *An M-domain M is a split opfibration $\mathcal{A} : \mathcal{D} \longrightarrow \mathcal{T}$ with the conditions:*

1. *$\mathcal{D}$ and $\mathcal{T}$ are posets with bottom elements $\bot^{\mathcal{D}}$ and $\bot^{\mathcal{T}}$ respectively.*
2. *$\mathcal{A}^{-1}(\bot^{\mathcal{T}})$ only contains $\bot^{\mathcal{D}}$.*
3. *$\mathcal{T}$ is coherent.*
4. *There is a continuous section $\mathcal{S}$ from $\mathcal{T}$ to $\mathcal{D}$.*
5. *$x \vee x'$ exists in $\mathcal{D}$ when $\mathcal{A}(x) \vee \mathcal{A}(x')$ exists in $\mathcal{T}$.*
6. *$\mathcal{D}$ is a complete partial order.*
7. *For each $X \in \mathcal{T}$, the function $|_X$ from $\cup_{\{Y | Y > X\}} \mathcal{A}^{-1}(Y)$ to $\mathcal{A}^{-1}(X)$ is continuous.*

The last two requirements are related to the reflectiveness of the domain we need to construct. $\mathcal{D}$ is a cpo in order to solve reflective domain equations. From this and the existence of a section, it can easily be shown that $\mathcal{T}$ is also complete. Coercion functions are continuous in order to interpret a function expression as a continuous function.

5.1.4 Record Space and Function Space

Suppose that an M-domain $M : \mathcal{D} \underset{\mathcal{S}}{\overset{\mathcal{A}}{\rightleftarrows}} \mathcal{T}$ is given. We construct a record space M-domain and a function space M-domain from this.

First, we consider the record space. Let L be an infinite set of labels. When O is a poset with a bottom element, we write $[L \longrightarrow O]$ for the poset of functions from L to O with the restriction that it maps only finitely many elements of L to non-bottom elements of O. The order on $[L \longrightarrow O]$ is induced by that of O.

Theorem 62 *Suppose that $M : \mathcal{D} \underset{\mathcal{S}}{\overset{\mathcal{A}}{\rightleftarrows}} \mathcal{T}$ is an M-domain. Then $[L \longrightarrow M] = [L \longrightarrow \mathcal{D}] \underset{[L \longrightarrow \mathcal{S}]}{\overset{[L \longrightarrow \mathcal{A}]}{\rightleftarrows}} [L \longrightarrow \mathcal{T}]$ is also an M-domain. Here, $[L \longrightarrow \mathcal{A}]$ maps $r \in [L \longrightarrow \mathcal{D}]$ to $\mathcal{A} \circ r$.*

Proof: (sketch.) Suppose that $Q, R \in [L \longrightarrow \mathcal{T}]$ with $Q < R$, $r : R$. The cleavage $r|_Q$ is defined to be the function $(r|_Q)(l) = r(l)|_{Q(l)}$. ∎

Next we consider the function space in detail. Recall that an element of a function type $F = [S_1 \longrightarrow S'_1, \ldots, S_n \longrightarrow S'_n]$ is a function which is applicable to a type T which is bigger than one of the S_i, and when $e : T$ is applied, then returns a value in $AP(F, T)$. We interpret the type $F = [S_1 \longrightarrow S'_1, \ldots, S_n \longrightarrow S'_n]$ as a step function from $\mathcal{T}$ to $\mathcal{T}$ which is given by this correspondence between T and $AP(F, T)$, and an element of this type as a function from $\mathcal{D}$ to $\mathcal{D}$ which follows this type correspondence. Note that the function defined by F

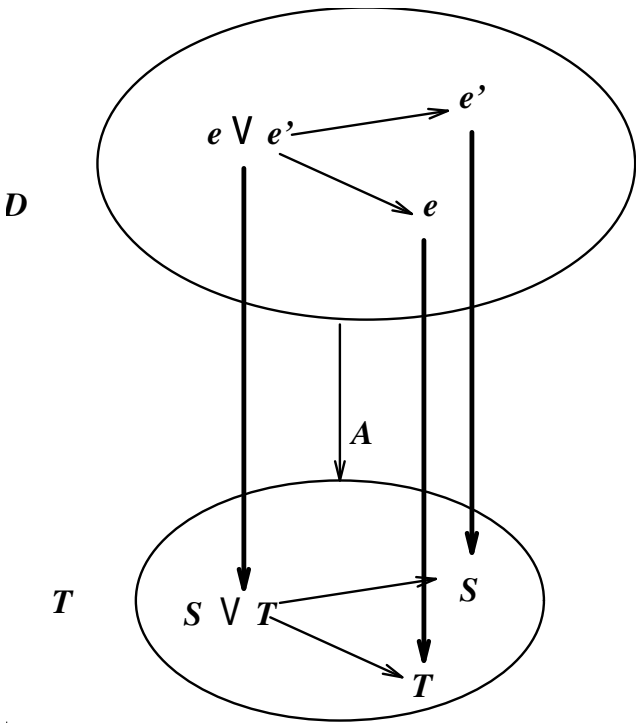


Figure 5.3: A Merge viewed as the least upper bound.

consistently complete, that is, if a set of types has an upper bound, then it has the least upper bound. For the latter, we require that when $\mathcal{A}(x) \vee \mathcal{A}(y)$ exists, then $x \vee y$ exists. We also need that $\mathcal{A}(x \vee y) = \mathcal{A}(x) \vee \mathcal{A}(y)$, which is guaranteed by the following proposition.

Proposition 61 *Let $\mathcal{A} : \mathcal{D} \longrightarrow \mathcal{T}$ be a split opfibration in which $\mathcal{D}$ and $\mathcal{T}$ are posets, and $\mathcal{T}$ is consistently complete. Then $\mathcal{A}$ is additive. That is, if $\vee_{\{t \in T\}}\ x_t$ exists in $\mathcal{D}$, then $\vee_{\{t \in T\}} \mathcal{A}(x_t)$ exists and is equal to $\mathcal{A}(\vee_{\{t \in T\}}\ x_t)$. As a special case, if $x \vee x'$ exists, then $\mathcal{A}(x \vee x') = \mathcal{A}(x) \vee \mathcal{A}(x')$. Also as a special case, $\mathcal{A}$ is continuous.*

Proof: Let $y = \vee_{\{t \in T\}} x_t$, $\mathcal{A}(x_t) = X_t (t \in T)$, and $\mathcal{A}(y) = Y$. $Y > X_t$ $(t \in T)$ because $\mathcal{A}$ is a functor. Thus, $\{X_t \mid t \in T\}$ has the least upper bound V. Since $Y > V$, $y > y|_V$. From the universality of $y|_V$, $y|_V > x_t$ $(t \in T)$. Because y is the least upper bound, $y|_V > y$. Therefore $y|_V = y$. This means that $Y = V$. ∎

Thus, we define an opfibration which has the concept of merge. In order to construct a model of λ_m, we need to define function types and record types. For this purpose, we also require the existence of the bottom elements in both $\mathcal{T}$ and $\mathcal{D}$, and a functor $\mathcal{S}$ from $\mathcal{T}$ to $\mathcal{D}$ for which $\mathcal{A} \circ \mathcal{S}$ is identity. The former is used to handle partial functions, and the latter is used to ensure that every type is non-empty. Thus, we define an M-domain, a domain which contains the structures required to model λ_m, as follows:

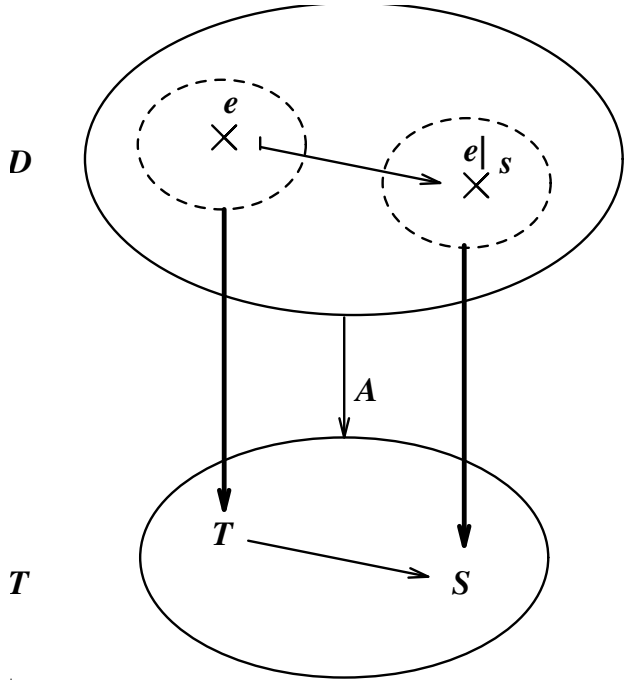


Figure 5.2: Data and value domain considered as an opfibration.

5.1.3 A Merge as the Least Upper Bound

Now, we consider the kind of structure needed to interpret the merge of two values. Recall that the typing rule of $\vee$ in λ_m was defined as follows:

$$\text{(T-LUB)}\ \frac{e : S \quad e' : S' \quad S \uparrow S'}{e \vee e' : S \vee S'}.$$

We interpret $S \vee S'$ as the least upper bound in $\mathcal{T}$, and $e \vee e'$ as the least upper boud in $\mathcal{D}$ (Figure 5.3). Here again, we use the notation of λ_m for an opfibration, and write the least upper bound in $\mathcal{T}$ and $\mathcal{D}$ by $\vee$. This definition is consistent with the case x and x' both have the same type. In this case, we have defined that $x \vee x'$ is the least upper bound in $\mathcal{V}(\mathcal{A}(x))$, which is also the least upper bound in $\mathcal{D}$.

For this interpretation of a merge as the least upper bound, we need two things; one is the existence of the least upper bound $X \vee Y$ when X and Y are "compatible", and the other is the existence of the least upper bound of x and y when $\mathcal{A}(x)$ and $\mathcal{A}(y)$ are compatible. We interpret the compatibility ($\uparrow$) of types by the compatibility of elements of a poset. That is, X and Y are compatible when X and Y have an upper bound.

For the former, we require that $\mathcal{T}$ is *coherent*([Bar84]), that is, if a set of types are pairwise compatible, then it has the least upper bound. It is a stronger condition than being

General category theory says that we can construct a split opfibration $FLAT(\mathcal{V}) : \mathcal{D} \longrightarrow$ **Ind** from a coindexed category $\mathcal{V} :$ **Ind** $\longrightarrow$ **Cat**. This construction is called the Grothendieck construction.

Definition 10 *Let $\mathcal{V} :$ **Ind** $\longrightarrow$ **Cat** be a coindexed category. Let $\mathcal{D}$ be the category whose objects are pairs (x, X), with X an object of* **Ind** *and x an object of $\mathcal{V}(X)$, and whose morphisms $(x, X) \longrightarrow (y, Y)$ are pairs (f, σ) with $\sigma : X \longrightarrow Y$ in* **Ind**, *and $f : \mathcal{V}(\sigma)(x) \longrightarrow y$ in $\mathcal{V}(Y)$. The composition of (f, σ) and (g, τ) in $\mathcal{D}$ is the pair $(g \circ \mathcal{V}(\tau)(f), \tau \circ \sigma)$. The functor $\mathcal{F} : \mathcal{D} \longrightarrow$* **Ind** *taking the second component of a pair is an opfibration. We write* **FLAT**$(\mathcal{V})$ *for this opfibration. It is actually a split opfibration with the cleavage of $(x, X) \in \mathcal{D}$ through $\sigma : X \longrightarrow Y$ defined as $(id_{\mathcal{V}(\sigma)x}, \sigma) : (x, X) \longrightarrow (\mathcal{V}(\sigma)x, Y)$.*

You can find the details of this construction in [BW90], [CGW89], [TBG91], and [Pho92]. Conversely to the Grothendieck construction, we can generally construct a coindexed category from a split opfibration. Thus, these two notions are equivalent ([BW90]).

As we have said in the previous section, we are only interested in the case when $\mathcal{T}$ is a poset and $\mathcal{V}(X)$ for each X are all posets. In this case, the opfibration constructed through Grothendieck construction is also a poset. That is, it is an opfibration $\mathcal{A} : \mathcal{D} \longrightarrow \mathcal{T}$ in which $\mathcal{D}$ and $\mathcal{T}$ are both posets (Figure 5.2). $\mathcal{D}$ is the set $\{(x, X) |\ X \in \mathcal{T} \text{ and } x \in \mathcal{V}(X)\}$, which is the union of the values of all types, with the order $(x, X) > (y, Y)$ if $x|_{Y,\sigma} > y$ in $\mathcal{V}(Y)$. Here, σ is the unique morphism from X to Y. Especially, $(x, X) > (x|_{Y,\sigma}, Y)$.

In the following, we only consider opfibrations whose domain and codomain are posets as the semantic domain. We will use the notation of $\lambda_m$for an opfibration of this nature. We write $x : X$ when $\mathcal{A}(x) = X$, and say that X is the type of x. We will also say that $x : X$ is an element of the opfibration. We will call $\mathcal{D}$ and $\mathcal{T}$ the value and type domain of an opfibration. In the following, we will use uppercase characters $X, Y, \ldots$ for objects of $\mathcal{T}$, and lower case characters $x, y, z, \ldots$ for objects of $\mathcal{D}$.

Let $\mathcal{A} : \mathcal{D} \longrightarrow \mathcal{T}$ be a split opfibration in which $\mathcal{D}$ and $\mathcal{T}$ are posets. In this case, the cleavage does not depend on the choice of a morphism in $\mathcal{T}$. Therefore, we only need to specify the target of a cleavage morphism to give a cleavage. Thus, we write the target of the cleavage as $x|_Y$. The requirement for a cleavage says that

1. $x|_X = x$ if $x : X$.
2. $x|_X|_Y = x|_Y$ for $x : Z, Z > X > Y$.

The universality of an opfibration says that when $X > Y$ and $x : X$, $x|_Y$ satisfies,

1. $x|_Y : Y$.
2. $x > x|_Y$.
3. for each $y : Z$ such that $Y > Z$ and $x > y$, $x|_Y > y$ holds.

From the third condition, by taking Z as Y, $x|_Y$ is the greatest element in Y which is less than x. That is, $x|_Y$ is the least upper bound of the set $\{y | \mathcal{A}(y) = Y \text{ and } x > y\}$.

When T is bigger than S by the coercible order, we can construct a term $e|_S$ from $e : T$. Therefore, we require the existence of a coercion function from $\mathcal{V}(T)$ to $\mathcal{V}(S)$ when $T > S$ as in Figure 5.1. This coercion function is denoted by $\mathcal{V}_{T,S}$. $\mathcal{V}_{T,S}$ is a monotonic function from $\mathcal{V}(T)$ to $\mathcal{V}(S)$. $\mathcal{V}_{T,T}$ must be the identity function on $\mathcal{V}(T)$. $\mathcal{V}_{*,*}$ must be transitive, that is, $\mathcal{V}_{T,S} \circ \mathcal{V}_{U,T}$ is equal to $\mathcal{V}_{U,S}$. We will also write $x|_S$ for $\mathcal{V}_{T,S}(x)$.

This gives us the view that $\mathcal{V}$ is a functor from $\mathcal{T}$, the set of types with the coercible order viewed as a category, to the category of upper semi lattices. More generally, we can view it as a coindexed category with the index set $\mathcal{T}$.

Definition 7 *Let* **Ind** *be a category and* **Cat** *be the category of small categories. A contravariant functor from* **Ind** *to* **Cat** *is called an indexed category. A covariant functor from* **Ind** *to* **Cat** *is called a coindexed category.*

In our case, **Ind** is the ordered set $\mathcal{T}$ viewed as a category, and the codomain of $\mathcal{V}$ is restricted to upper semi lattices viewed as small categories. A monotonic function from an ordered set to another ordered set can be considered as a functor between them. Thus, $\mathcal{V}$ is a coindexed category.

5.1.2 Grothendieck Construction

A coindexed category gives us an intuitive understanding of the structure we are going to construct. However, in order to consider the merge operator which acts on values of different types, we need a large domain which contains all the values.

Definition 8 *An opfibration is a functor $\mathcal{F}$ from a category* **C** *to another category* **Ind**, *such that, for an object $x \in \mathbf{C}$ and a morphism $\sigma : \mathcal{F}(x) \longrightarrow Y$, there exists an object $y \in \mathbf{C}$ with $\mathcal{F}(y) = Y$ and a morphism $g : x \longrightarrow y$ with $\mathcal{F}(g) = \sigma$ with the following universality. For any $f : x \longrightarrow z$ and a morphism $\alpha : Y \longrightarrow \mathcal{F}(z)$ which satisfies $\mathcal{F}(f) = \alpha \circ \sigma$, there exists a morphism $h : y \longrightarrow z$ with $\mathcal{F}(h) = \alpha$.*

We will consider an opfibration in which the choice of y in the definition is fixed for each $x \in \mathbf{C}$ and $\sigma : \mathcal{F}(x) \longrightarrow Y$.

Definition 9 *A cleavage of an opfibration is a function which returns the g in the above definition to $x \in \mathbf{C}$ and $\sigma : \mathcal{F}(x) \longrightarrow Y$ and satisfy the following conditions. In the following, we write $x|_{Y,\sigma}$ for the value of a cleavage, and call it the cleavage of x through σ.*

1. *the cleavage of x through identity is identical to x.*
2. *the composition of the cleavage of x through σ from x to y, and the cleavage of y through τ, is equal to the cleavage of x through $\tau \circ \sigma$.*

An opfibration $\mathcal{F} : \mathbf{C} \longrightarrow$ **Ind** *is splitting if it is equipped with a cleavage.*

the set of types form a poset (partially ordered set) instead of a pre-ordered set. Note that it is not the case for λ_m^* in which e and $e|_S$ behave differently.

We first study the structure required for each $\mathcal{V}(S)$. When $e : S$ and $f : S$, then their merge $e \vee f$ is defined in the type $S \vee S$, which is equal to S or equivalent to S. Therefore, for $x, y \in \mathcal{V}(S)$, $x \vee y \in \mathcal{V}(S)$ must be defined on each type S. For this, we give an order structure to each type, and define $x \vee y$ as the least upper bound of x and y. Since $e \vee f$ is a valid expression for all $e, f : S$, $\mathcal{V}(S)$ is required to be an upper semi-lattice; that is, a partially ordered set in which every non-empty finite subset has an upper bound.

In the basic types int and $bool$, all closed normal forms are constants. Therefore, we define $\mathcal{V}(int)$ and $\mathcal{V}(bool)$ be the flat upper semi-lattices defined as follows.

$$\begin{array}{lccccc} & & \top_{int} & & & \\ \mathcal{V}(int): & / & | & \ldots & \\ & 0 & 1 & 2 \ldots & \end{array} \qquad \begin{array}{lccc} & & \top_{bool} & \\ \mathcal{V}(bool): & / & & \backslash \\ & true & & false \end{array}$$

Note that these domains have the dual structure to the domains used in Scott semantics ([Sco76], [Sto77]), and the semantics of the language PCF([Plo77], [BTGS90]). In these semantics, a bottom element which means a non-terminating program is introduced to each type, and this element is considered as an undefined value. Here, a top element is considered as an overdefined element which cause contradiction, like both 1 and 2 at the same time.

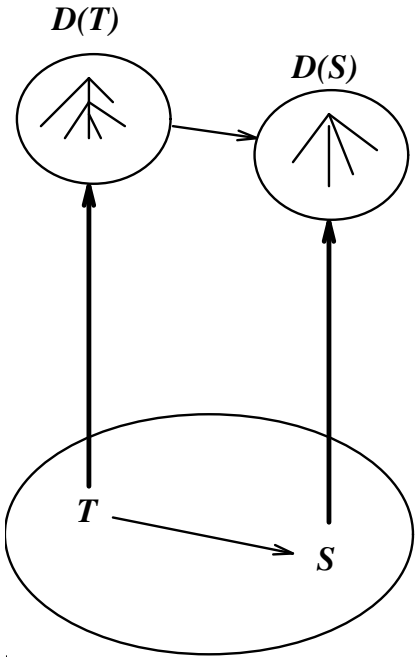


Figure 5.1: Coercion functions attached to the coercible order.

Chapter 5

Denotational Semantics of λ_m

In this chapter, we construct a semantic domain and give semantics to λ_m. The important notions of λ_m are the coercion operator and the merge operator. First, we show that coindexed category or opfibration is a natural structure to interpret the coercion operator. Then we show that merge operator can be interpreted as the least upper bound on an opfibration. After that, we show that these structures, opfibration with least upper bounds, are preserved by the record space and function space constructors, and construct the semantic domain.

Though the structure of the model can be expressed only using order relations, we sometimes explain it categorically considering an ordered set as a special case of a category. We use categorical explanation for two reasons. One is that it is important to note that the structure we construct is a special case of a well-known categorical structure. The other is that, by giving some properties of our model as categorical theorems, the results become applicable to other cases. One can read this chapter skipping all the categorical explanation. We use categorical tools like coindexed categories, Grothendieck construction, and opfibration. Note that these tools are also used in constructing a domain theoretic model of the higher order lambda calculus([CGW89]).

5.1 M-domain: A Semantic Domain for λ_m

5.1.1 Coercion as a Coindexed Category

We start with the intuition that a type is a set of values which belong to the type. We write $\mathcal{V}(S)$ for the set of values belonging to the type S. We use x, y for elements of $\mathcal{V}(S)$.

$\mathcal{V}(S)$ and $\mathcal{V}(T)$ are disjoint sets when $S \neq T$, because the unicity of type holds.

We identify equivalent types. This is justified as follows. When $e : T$ and $T \simeq S$, define $f = e|_S$. $f|_T$ and e are equivalent by theorem 41 and 40. We identify equivalent terms in the semantics, because they have the same functionality. Thus, the correspondence between e and f is one to one, and thus we can identify them through this coercion function. Thus,

From this proposition, it is expected that an element of type $S \vee T$ is always composed of an element of type S and an element of type T, and a type $[S_1 \to S'_1, \ldots, S_n \to S'_n]$ has the intuitive meaning of a list of monomorphic functions of types $S_1 \to S'_1, \ldots, S_n \to S'_n$. This must be the most intuitive meaning of a merge of types. However, the author leaves it an open problem whether this holds. Especially, the author leaves it an open problem whether $e|_S$ is congruent with e when $e : S$.

4.5 $\lambda_m^\sharp$: Yet Another Calculus with a Merge Operator

There is another alternative to (E-APPC1). That is, to express directly the function obtained by coercing another function as a merge of lambda expressions. When the target type is a simple function type, reduce $e|_{[S \longrightarrow S']}$ to $\lambda(y^S).(e\ y)|'_S$. In general, when the target type F is not a simple function type, coerce the function e to each simple function types which compose F, and merge them. that is,

$$\text{(E-APPC1}^\sharp\text{)}\quad e|_{[S_1 \longrightarrow S'_1,\ldots,S_n \longrightarrow S'_n]} \ \triangleright\ \lambda y^{S_1}.(e\ y)|_{S'_1} \vee \ldots \vee \lambda y^{S_n}.(e\ y)|_{S'_n}.$$

The left hand side of this reduction rule actually has type $[S_1 \longrightarrow S'_1, \ldots, S_n \longrightarrow S'_n]$. Therefore, this rule is type correct and does not change the subject reduction property.

We write $\lambda_m^\sharp$ for the calculus obtained from λ_m by replacing (E-APPC1) with the rule (E-APPC1$^\sharp$).

Note that $\lambda_m^\sharp$ is not equivalent to λ_m, nor λ_m^*. For example, the example in Section 4.4.1 reduces as follows:

$$\begin{aligned}
f|_F\ \{l_1 = 1, l_2 = 1\} &\triangleright ((\lambda x^{\{l_1:int\}}.(f\ x)|_{int}) \vee (\lambda x^{\{l_2:int\}}.(f\ x)|_{int}))\ \{l_1 = 1, l_2 = 1\} \\
&\triangleright ((\lambda x^{\{l_1:int\}}.(f\ x)|_{int})\ \{l_1 = 1, l_2 = 1\}) \vee ((\lambda x^{\{l_2:int\}}.(f\ x)|_{int})\ \{l_1 = 1, l_2 = 1\}) \\
&\triangleright^* (f\ \{l_1 = 1, l_2 = 1\}|_{\{l_1:int\}}) \vee (f\ \{l_1 = 1, l_2 = 1\}|_{\{l_2:int\}}) \\
&\triangleright^* (f\ \{l_1 = 1\}) \vee (f\ \{l_2 = 1\}) \\
&\triangleright^* 1 \vee 1 \\
&\triangleright 1.
\end{aligned}$$

Similarly to λ_m^*, $\lambda_m^\sharp$does not have the transitivity of coercions. That is, let $U > G > F$ and $f : U$. $(f|_G)|_F\ g$ is not, in general, equal to $f|_F\ g$. A counter example is same as the one for λ_m^* in Section 43.

$\lambda_m^\sharp$ has strong normalization and Church-Rosser properties. The proof of strong normalization is a bit simpler than that of λ_m^*, in that an expression of the form $e|_S$ can be added to neutral terms when we consider a system corresponding to λ'_m and λ''_m. Church-Rosser is proved much the same way as for λ_m. It is also proved that $\lambda_c^\sharp$, which is a subsystem of $\lambda_m^\sharp$ defined in the same way as λ_c^*, is equivalent to λ_c and λ_c^*.

This reduction rule has the drawback that it is uneconomical and unnatural that it duplicates the expression $e\ y$, and produces new lambda expressions. This corresponds to a kind of η-expansion.

$\lambda_m^\sharp$ has the following good property which λ_m and λ_m^* did not have.

Proposition 60 *$e|_{S \vee T}$ is congruent with $e|_S \vee e|_T$.*

Proof: Proved easily by induction on the degree of $S \vee T$. ∎

Suppose that e has the form $e_1 \vee \ldots \vee e_n$ with $e_i : F_i$ $(i = 1, \ldots, n)$. $(e_1 \vee \ldots \vee e_n)\ f \triangleright (e_{\phi(1)}\ f) \vee \ldots \vee (e_{\phi(l)}\ f)$. On the other hand, $(e_1 \vee \ldots \vee e_n)\ f|_{AT(F,T)} \triangleright (e_{\phi(1)}\ f|_{AT(F,T)}) \vee \ldots \vee (e_{\phi(l)}\ f|_{AT(F,T)})$. By induction hypothesis, $e_{\phi(i)}\ f|_{AT(F,T)}$ is congruent with $e_{\phi(i)}\ f|_{AT(F,T)}|_{AT(F_1,T)}$, and $e_{\phi(l)}\ f$ is congruent with $e_{\phi(i)}\ f|_{AT(F_1,T)}$. Here, we used the fact $AT(F_1, AT(F,T)) = AT(F_1,T)$. From the lemma below, they are congruent.

Suppose that e has the form $e_0|_F$. $e\ f$ reduces to $e_0\ f|_{AT(F,T)}$. On the other hand, $e\ f|_{AT(F,T)}$ reduces to $e_0\ f|_{AT(F,T)}|_{AT(F,T)}$, which are congruent by the lemma below. ∎

Lemma 59 *In λ^*_m, if S is the merge of T and other types, then $e|_S|_T$ and $e|_T$ are congruent.*

Proof: We need to prove that $C[e|_S|_T]$ has a basic type and reduces to a normal form c iff $C[e|_T]$ reduces to c. Since λ^*_m and λ''_m have strong normalization and Church-Rosser properties, every expression has a unique normal form in these systems. Therefore, $e \triangleright^* c$ in λ^*_m iff $e^\diamond \triangleright^* c^\diamond$ in λ''_m. Since $c = c^\diamond$, $e \triangleright^* c$ in λ^*_m iff $e^\diamond \triangleright^* c$ in λ''_m. Therefore, we only need to prove that $C^\diamond[e^\diamond|_S|_T]$ reduces to c iff $C^\diamond[e^\diamond|_T]$ reduces to c. Here, $C^\diamond[\cdot]$ is a translation of $C[\cdot]$ which may include a coercion to the type expected by T on the hole. This is obvious because $e^\diamond|_S|_T$ reduces to $e^\diamond|_T$ in λ''_m, and λ''_m has Church-Rosser property. ∎

4.4.2 A Note on the Rule (E-APPL)

As we have noted, $e : F$ in λ^*_m has the intuitive meaning of a list of functions of types $M(F)$, and the application $e\ f$ selects one function $S \longrightarrow S'$ from $M(F)$, coerce the argument to S, and apply it to f.

From this meaning, it is natural also to replace the rule (E-APPL)

$$\text{(E-APPL)} \quad (e_1 \vee \ldots \vee e_n)\ f \ \triangleright\ e_{\phi(1)}\ f \vee \ldots \vee e_{\phi(l)}\ f$$

by the following rule (E-APPL*)

$$\text{(E-APPL}^*\text{)} \quad (e_1 \vee \ldots \vee e_n)\ f \ \triangleright\ e_{\phi(1)}\ f|_{AT(F,S)} \vee \ldots \vee e_{\phi(l)}\ f|_{AT(F,S)},$$

where, $f : S$, $e_i : F_i$, and $F = F_1 \vee \ldots \vee F_n$. $e_{\phi(i)}$ are selected so that $AT(F_{\phi(i)}, S)$ exist. That is, since $(e_1 \vee \ldots \vee e_n)\ f$ is an application of a function of type F, f must be coerced to the type $AT(F,S)$ and then $(e_1 \vee \ldots \vee e_n)$ is applied. If we use the rule (E-APPL) instead, then it will lose the information that $(e_1 \vee \ldots \vee e_n)$ is once applied before each e_i is applied.

It seems that (E-APPL) and (E-APPL*) cause different results because one coercion is inserted in (E-APPL*) which may cause different result because coercions are not transitive. However, from the Lemma 59, coercions are transitive for the special case we are concerned with here, and these two rules form equivalent systems. That is, in reducing $e_{\phi(l)}\ f|_{AT(F,S)}$, the argument $f|_{AT(F,S)}$ will be coerced to $AT(F_{\phi(l)}, AT(F,S))$ which is equal to $AT(F_{\phi(l)}, S)$, and the above lemma says that $f|_{AT(F,S)}|_{AT(F_{\phi(l)},S)}$ is equal to $f|_{AT(F_{\phi(l)},S)}$. This fact justifies the rule (E-APPL).

The precise proof of this equivalence can be given easily by translating the two systems to λ''_m.

function with the type $AT(F,T) \longrightarrow AP(F,T)$ from this list, and then applying it to f. This application has the same meaning as in λ_c. That is, f is coerced to $AT(F,T)$, and then e is applied.

Now, suppose that $G > F$. Since $AP(G,S) > AP(F,S)$ for all S, an intuitive meaning of the coercion of a function $e : G$ to F is the list of monomorphic functions e_i $(i = 1, \ldots, n)$, where the behavior of $e_i : T_i \longrightarrow T'_i \in M(F)$ is the same as that of e to T_i, except that the result is coerced from $AP(G,T_i)$ to $AP(F,T_i) = T'_i$. Therefore, when $e|_F$ is applied to $f : S$, the argument is coerced to the type $AT(F,S)$, e is applied, and the result is coerced to $AP(F,S)$.

Note that the meaning of $[S_1 \longrightarrow S'_1, \ldots, S_n \longrightarrow S'_n]$ is different from a list of monomorphic functions of types $S_1 \longrightarrow S'_1, \ldots, S_n \longrightarrow S'_n$, and thus $f|_F$ is different from $f|_{S_1 \longrightarrow S'_1} \vee \ldots \vee f|_{S_n \longrightarrow S'_n}$ as is given in the following example.

Example Consider the type $F = [\{l_1 : int\} \longrightarrow int, \{l_2 : int\} \longrightarrow int]$. and consider the functions $f_1 = \lambda x^{\{l_1:int\}}.1$, $f_2 = \lambda x^{\{l_2:int\}}.1$, $f_3 = \lambda x^{\{l_1:int, l_2:int\}}.E_{int}$.

Let $f = (f_1 \vee f_2 \vee f_3)$. $f|_{[\{l_1:int\} \longrightarrow int]}(\{l_1 = 1, l_2 = 1\}) \triangleright^* 1$, $f|_{[\{l_2:int\} \longrightarrow int]}(\{l_2 = 1, l_2 = 1\}) \triangleright^* 1$. However,

$$
\begin{aligned}
f|_F(\{l_1 = 1, l_2 = 1\}) &\triangleright (f\ \{l_1 = 1, l_2 = 1\}|_{\{l_1:int, l_2:int\}})|_{int} \\
&\triangleright (f\ \{l_1 = 1, l_2 = 1\})|_{int} \\
&\triangleright^* (f_1\ \{l_1 = 1, l_2 = 1\}) \vee (f_2\ \{l_1 = 1, l_2 = 1\}) \vee (f_3\ \{l_1 = 1, l_2 = 1\}) \\
&\triangleright^* 1 \vee 1 \vee E_{int} \\
&\triangleright^* E_{int}.
\end{aligned}
$$

This example also says that $e|_{S \vee T}$ is not equivalent to $e|_S \vee e|_T$.

Therefore, in λ^*_m, a function is composed from only finite number of monomorphic functions whose type is determined by the type of the function. Note that $AT(F,T)$ has less degree than F and therefore F does not appear in $AT(F,T)$. Therefore, though there is a term e which is applicable to itself, the result depends not on e but on $e|_S$ for a type S, which has little degree than e has. This is an intuitive explanation why strong normalization holds in λ^*_m.

This view of a function type is justified by the following proposition.

Proposition 58 *Let $e : F$ and $f : T$, for which $AT(F,T)$ exists. Then $e\ f$ is congruent with $e\ f|_{AT(F,T)}$.*

For the definition of congruence, see Section 3.3.6.

Proof: Prove it by induction on the form of e. Since λ^*_m is strong normalizing and Church-Rosser property, we can safely assume that e has a normal form.

Suppose that e has the form $\lambda x^S.e_0$. In this case, $e\ f$ reduces to $e_0[x^S := f|_S]$, and $e\ f|_{AT(F,S)}$ reduces to $e_0[x^S := f|_S|_S]$, because $AT(F,S) = S$ for this case. This is congruent with $e_0[x^S := f|_S]$ by the lemma below.

On the other hand, $(e_1 \vee \ldots \vee e_n)\ f$ reduces in λ^*_m to $(e_{\phi(1)}\ f) \vee \ldots \vee (e_{\phi(l)}\ f)$, which is mapped by $^\diamond$ to the same expression.

For (E-APPC1*), let $e : G$ and $f : S$. $e|_f\ f \triangleright (e\ f|_{AT(F,S)})|_{ap(F,S)}$ by (e-appc1*). $(e|_f\ f)^\diamond = e^\diamond|_F\ f^\diamond|_{AT(F,S)}$ and it reduces in λ''_m to $(e^\diamond\ f^\diamond|_{AT(F,S)}|_{AT(G,AT(F,S))})|_{AP(F,AT(F,S))}$. On the other hand, $((e\ f|_{AT(F,S)})|_{AP(F,S)})^\diamond$ is equal to $(e^\diamond\ f^\diamond|_{AT(F,S)}|_{AT(G,AT(F,S))})|_{AP(F,S)}$. Note that $AP(F, AT(F,S)) = AP(F,S)$ by Lemma 34. ∎

Compared to the Lemma 18 for λ_c, this proof is simpler for the case (E-APPC1), because $(e|_F\ f)^\diamond$ reduces to $((e\ f|_{AT(F,S)})|_{AP(F,S)})^\diamond$ directly without the help of (E-CC). This is because we are proving it not for λ_m but for λ^*_m. Actually, if we try to prove the strong normalization of λ_m in this way, we need more general (E-CC) rule in λ'_m, which makes the system inconsistent. Thus, the non-normalization of reductions in λ_m and non-transitivity of coercions in λ'_m are closely related.

Theorem 57 *The system λ^*_m is strong normalizable.*

Proof: Let $\Delta = f_1 \triangleright \ldots \triangleright f_n \triangleright \ldots$ be an infinite reduction sequence in λ^*_m. Then $\Delta' = {f_1}^\diamond \triangleright^* \ldots \triangleright^* {f_n}^\diamond \triangleright^* \ldots$ is also a reduction sequence in λ''_m. ∎

4.4 Other Syntactic Properties of λ^*_m

4.4.1 The Meaning of a Function Type in λ^*_m

In λ_m, we used the rule (E-APPC1).

$$\text{(E-APPC1)} \quad (e|_F)\ f \triangleright (e\ f)|_{AP(F,S)}. \quad (f : S)$$

This rule calculates the application of $e|_F$ to f by calculating the application of e to f and then adjust the result to the expected type. This rule is based on the view that (1) a type $G = [S_1 \longrightarrow S'_1, \ldots, S_n \longrightarrow S'_n]$ determines a function from the set of types $\mathcal{T}$ to itself which is the correspondence of the argument type and the result type, (2) $G > F$ if the function defined by G is bigger than F in the monotonic function space $[\mathcal{T} \longrightarrow \mathcal{T}]$, and (3) the coercion of $e : G$ to F is the function pointwisely coercing the result of e to the type expected by F.

In λ^*_m, we used the rule

$$\text{(E-APPC1*)} \quad (e|_F)\ f \triangleright (e\ f|_{AT(F,S)})|_{AP(F,S)} \quad (f : S)$$

which results in a calculus with different behavior.

We explain this rule in detail. We view in $\lambda^*_m$the type $F = [S_1 \longrightarrow S'_1, \ldots, S_n \longrightarrow S'_n]$ as a list of monomorphic functions of types $M(F)$. For instance, we view the type $[S_1 \longrightarrow S'_1, S_2 \longrightarrow S'_2]$ with $S_1 \uparrow S_2$ as a list of monomorphic functions of types $S_1 \longrightarrow S'_1$, $S_2 \longrightarrow S'_2$, and $S_1 \vee S_2 \longrightarrow S'_1 \vee S'_2$. And the application $e\ f$ with $f : T$ as first selecting one monomorphic

For the (E-CC2) step, we need to show that S is a merge of $AP(F,S)$ and other types. Because $e_1|_F|_G\ f$ is typable in λ'_m, $M(G)$ includes $S \longrightarrow S'$ for some S'. Therefore, when we write $G = [S_1 \longrightarrow S'_1, \ldots, S_n \longrightarrow S'_n]$, $S = S_{\phi(1)} \vee \ldots \vee S_{\phi(l)}$ for some $\phi(1), \ldots, \phi(l)$. Since F includes $S_i \longrightarrow S'_i$ as its components, $AT(F,S)$ is the merge of $S_{\phi(i)}$ $(i = 1.\ldots, l)$ and other types.

Next, we study the case (E-CC) is (E-CC2). F has all the function components of U, e reduces in the same way as above, except that we use (E-CC1) instead of (E-CC2), and (E-CC2) instead of (E-CC1). $AT(U, AT(F,S)) = AT(U,S)$ for the same reason.

For the (E-CC2) step, we need to show that $AP(F, AT(F,S))$ is the merge of $AP(U, AT(U, AT(F,S)))$ and other types. The former is equal to $AP(F,S)$, and the latter is equal to $AP(U, AT(F,S))$. Since $S > AT(F,S)$, a component of $AP(U, AT(F,S))$ is a component of $AP(U,S)$, which is included in $AT(F,S)$.

For the (E-CC1) step, we need to show that $AT(F,S)$ has all the components of $AT(U, AT(F,S))$. This is proved similarly. ∎

From this Lemma, the SN of λ''_m follows in the same way as the SN of λ''_c (Section 2.2.3).

Proposition 54 *λ''_m is strongly normalizable.*

We need to prove that λ''_m has the Church-Rosser property to justify the (E-CC) rules in λ''_m.

Proposition 55 *λ''_m has the Church-Rosser property.*

Proof: Since λ''_m is strongly normalizing and the Lemma 53 holds, it is proved in the same way as in Subsection 2.2.4 for λ''_c. ∎

4.3.4 Strong Normalization of λ^*_m

We will subsume the SN of λ_m to the SN of λ''_m.

Lemma 56 *If $e \triangleright e'$ in λ^*_m, then $e^\diamond \triangleright^* e'^\diamond$ in λ''_m in one or two steps.*

Proof: It is trivial for (E-LABEL), (E-COE1), (E-COE2), (E-LUBR), (E-CONST1), (E-CONST2) and (E-CONST3). For (E-APP), the proof is same as in Lemma 18 because $AT([S \longrightarrow S'], T) = S$ if $T > S$.

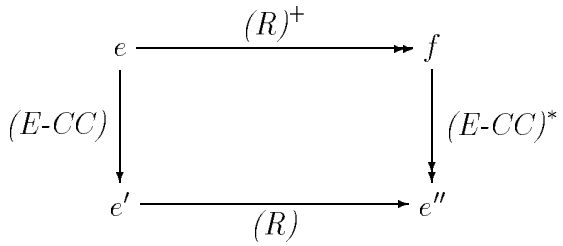
For (E-APPL), Let $e_1 : F_1, \ldots, e_n : F_n, f : S$, and $F = F_1 \vee \ldots \vee F_n$. $(e_1 \vee \ldots \vee e_n)\ f \triangleright e_{\phi(1)}\ f \vee \ldots \vee e_{\phi(l)}\ f$ by (E-APPL), where $e_{\phi(i)}$ are selected so that $AT(F_{\phi(i)}, S)$ exists for $F_{\phi(i)}$.

$((e_1 \vee \ldots \vee e_n)\ f)^\diamond = (e_1{}^\diamond \vee \ldots \vee e_n{}^\diamond)\ f^\diamond|_{AT(F,S)}$. It reduces in λ''_m to $e_{\phi(1)}{}^\diamond\ (f^\diamond|_{AT(F,S)}|_{AT(F_{\phi(1)},S)}) \vee \ldots \vee e_{\phi(l)}{}^\diamond (f^\diamond|_{AT(F,S)}|_{AT(F_{\phi(l)},S)})$ because $AT(F_{\phi(i)}, AT(F,S)) = AT(F_{\phi(i)}, S)$ by lemma 34. Since $AT(F,S)$ is the merge of $AT(F_{\phi(1)}, S)$ and other types, (E-CC1) applies and thus it reduces to $e_{\phi(1)}{}^\diamond\ (f^\diamond|_{AT(F_{\phi(1)},S)}) \vee \ldots \vee e_{\phi(l)}{}^\diamond (f^\diamond|_{AT(F_{\phi(l)},S)})$.

The condition S is a merge of T needs more explanation. The precise meaning is that $S = S_1 \vee S_2 \vee \ldots S_n$ and $T = T_0 \vee S_1 \vee T_1 \vee S_2 \vee T_2 \vee \ldots \vee S_n \vee T_n$ for some $S_1, \ldots, S_n$ and $T_0, \ldots, T_n$ where T_0 and T_n may not exist. Note that $=$ does not mean equivalence but syntactic equality. For function types, it means that all the function components of T appear in S. Thus, for function types, (E-CC1) says that coercion is transitive in the case that S is composed from T by adding function components. (E-CC2) says that coercion is transitive in the case that the type of e, U, is composed from S by adding function components, which are less than U. We will generally call (E-CC1) and (E-CC2) the (E-CC) rules. We define λ''_m as the extension of λ'_m with the (E-CC) rules.

Since coercion is not transitive in λ'_m, we need to show that coercion is transitive for these special cases, which will be done by proving the Church-Rosser property of λ''_m. Before that, we prove the following lemma which corresponds to Lemma 10 of λ''_c.

Lemma 53 *In λ''_m, if $e \triangleright e'$ by a (E-CC) rule and $e' \triangleright e''$ by a rule (R) which is not a (E-CC) rule, then there is a term f to which e reduces one or two steps by the rule (R), and f is equal to e'' or f reduces to e'' by several (E-CC) rules. In a diagram,*



Here, $(R)^+$ denotes one or two (R) rules and (E-CC)* denotes zero or more (E-CC) rules.

Proof: The proof is similar to that of Lemma 10. We will only prove the case (R) is (E-APPC1'). First study the case (E-CC) is (E-CC1). F has all the function components of G, e has the form $e_1|_F|_G\ f$ with $e_1 : U$ and $f : S$, $e' = e_1|_G\ f$, and $e'' = (e_1\ f|_{AT(U,S)})|_{AP(G,S)}$. On the other hand, e reduces as follows:

$$
\begin{array}{lll}
e_1|_F|_G\ f & \triangleright (e_1|_F\ f|_{AT(F,S)})|_{AP(G,S)} & \text{(E-APP1')} \\
& \triangleright (e_1\ f|_{AT(F,S)}|_{AT(U,AT(F,S))})|_{AP(F,AT(F,S))}|_{AP(G,S)} & \text{(E-APPC1')} \\
& \triangleright (e_1\ f|_{AT(F,S)}|_{AT(U,AT(F,S))})|_{AP(G,S)} & \text{(E-CC1)} \\
& \triangleright (e_1\ f|_{AT(U,AT(F,S))})|_{AP(G,S)}. & \text{(E-CC2)}
\end{array}
$$

Note that $AT(G,S) = S$ because $e_1|_F|_G\ f$ is well typed. On the other hand, $S > AT(F,S) > AT(G,S)$. Thus, $AT(F,S) \simeq S$. Therefore, $AT(U, AT(F,S)) = AT(U,S)$.

For the (E-CC1) step, we need to show that $AP(F, AT(F,S))$ is a merge of $AP(G,S)$ and other type expressions. First note that $AP(F, AT(F,S)) = AP(F,S)$. From the construction of $AP(F,S)$, it is obvious that $AP(F,S)$ is a merge of $AP(G,S)$ and other type expressions.

is reducible. Since $e \in Red(T)$ and $f \in Red(S)$, they are strongly normalizable. Thus, we prove it by induction on $\nu(e) + \nu(f)$, the sum of the max length of reductions starting e and f. Let e' and f' be one step reduction from e and f respectively. $(e|_F)f$ one step reduces to $(e'|_F)f$ and $(e|_F)f'$, which are reducible from induction hypothesis. It also reduces to $(e\ f|_{AT(G,S)})|_{AP(F,AT(G,S))}$. Since $S > AT(G,S)$ is proved in the proof of $G > F$, from induction hypothesis, $f|_{AT(G,S)}$ is reducible. Because e is reducible, $e\ f|_{AT(G,S)}$ is a reducible expression of type $AP(G,S)$. Since $AP(G,S) > AP(F,AT(G,S))$ also needs to be proved in the proof of $G > F$, $(e\ f|_{AT(G,S)})|_{AP(F,AT(G,S))}$ is reducible. ∎

Lemma 50 *$e \in Red(S)$ and $f \in Red(T)$ implies $e \vee f \in Red(S \vee T)$.*

Proof: Induction on the structure of S and T. When both are basic types, simply induction on $\nu(e) + \nu(f)$. Suppose that both are record types. Induction on $\nu(e) + \nu(f)$ to prove that $(e \vee f).l_i$ are reducible for all slots l_i. It one step reduces to $(e' \vee f).l_i$ and $(e \vee f').l_i$, both of them are reducible by induction hypothesis on $\nu(e) + \nu(f)$. If both e and f are of the form of records, it also reduces to another record, with each slots of the form e_i, f_i, or $e_i \vee f_i$; all of them are reducible by induction on the structure of S_1 and S_2. The last case is that both are function types. Then $e \vee f$ applied to a reducible term d reduces to the forms $(e' \vee f)\ d$, $(e \vee f')\ d$, $(e \vee f)\ d'$, $e\ d|_U$, $f\ d|_V$, $e\ d|_U \vee f\ d|_V$, for some types U and V. which are all reducible by induction hypothesis, the above lemma, and the definition of $Red([S_1 \longrightarrow S'_1, \ldots, S_n \longrightarrow S'_n])$. ∎

As in λ'_c, the following theorem and corollary are proved.

Theorem 51 *If $e : S'$ is a term with free variables x_i of type S_i, then e satisfies $e[x_i := f_i] \in Red(S')$ for all $f_i \in Red(S_i)$.*

Corollary 52 *All the terms of λ'_m are strongly normalizable.*

4.3.3 Strong Normalization of λ''_m

As in Section 2.2.3, we will define an extension of λ'_m with the rule of transitivity of coercions. However, as we have shown, coercion is not, in general, transitive in λ^*_m nor in λ'_m. Therefore, in order to obtain a consistent system, we need to restrict the (E-CC) rule in λ''_c only to some cases.

$$\text{(E-CC1)}\quad \frac{S \text{ is a merge of } T \text{ and other types}}{(e|_S)|_T \ \triangleright\ e|_T}$$

$$\text{(E-CC2)}\quad \frac{e : U,\ U,\ S \text{ are equivalent types and } S \text{ is a merge of } U \text{ and other types}}{(e|_S)|_T \ \triangleright\ e|_T}.$$

- $Red([S_1 \longrightarrow S'_1, \ldots, S_n \longrightarrow S'_n]) = \{e \mid e\ f \in Red(S')\ for\ all\ S \longrightarrow S' \in M([S_1 \longrightarrow S'_1, \ldots, S_n \longrightarrow S'_n])\ and\ f \in Red(S)\}$.

As in Section 2.2.2, we define the set of neutral terms as follows:

$$x^S \mid c^{\top}_{int} \mid c^{\top}_{bool} \mid e\ f \mid e.l.$$

As in Section 2.2.2, $Red(S)$ has the following three properties.

Lemma 46

- *If $e \in Red(S)$ then e is strongly normalizable.*
- *If $e \in Red(S)$ and $e \triangleright e'$ then $e' \in Red(S)$.*
- *If e is neutral and every e' which is one step reduction from e satisfies $e' \in Red(S)$, then $e \in Red(S)$.*

From the last property, a variable x^S belongs to $Red(S)$.

The following two lemmas are proved much the same way as in λ'_c.

Lemma 47 *If e_i belongs to $Red(S_i)$ $(i = 1, \ldots, n)$, then $\{l_1 = e_1, \ldots, l_n = e_n\} \in Red(\{l_1 : S_1, \ldots, l_n : S_n\})$.*

Lemma 48 *Let $e : S'$ be a term which may contain a free variable x^S of type S. If e satisfies $e[x^S := f] \in Red(S')$ for all $f \in Red(S)$, then $\lambda x^S.\ e \in Red(S \longrightarrow S')$.*

Lemma 49 *IF $e \in Red(T)$, then $e|_S \in Red(S)$ $(T > S)$*

Proof: Induction on the proof of $T > S$ using the modified version of the coercible order defined in 3.3.1. In order to apply induction, we replace (I-FUN') with the following rule

$$\text{(I-FUN")}\quad \frac{AT(G,S) \text{ exists for each } S \longrightarrow S' \in M(F),\ AT(G,S) < S, \text{ and } AP(G, AT(G,S)) > S'}{G > F}$$

The condition $AT(G,S) < S$ was proved in the Lemma 33, and is alway true. From the discussion in 3.3.1, the proof of $AT(G,S) < S$ does not depend on $G > F$. Thus, this rule does not add circularity to the system. Therefore, this modification does not change the coercible order. The proof is similar to that for λ'_c. The only case we need to consider is that $T > S$ is proved using (I-FUN").

Suppose that the last rule is (I-FUN"). For notational consistency, we use G for T and F for S. We need to prove that for all $S \longrightarrow S' \in M(F)$, and for all $f \in Red(S)$, $(e|_F)f$

$(E-APP')$ $(\lambda x^S.\ e)\ f \ \triangleright\ e[x^S := f]$

$(E-APPL')$ $(e_1 \vee \ldots \vee e_n)\ f \ \triangleright\ (e_{\phi(1)}\ f|_{AT(F_{\phi(1)},S)}) \vee \ldots \vee (e_{\phi(l)}\ f|_{AT(F_{\phi(l)},S)})$
(Where $e_{\phi(i)}$ are selected so that $AT(F_{\phi(i)}, S)$ exist for $F_{\phi(i)}$ the type of $e_{\phi(i)}$, and S the type of f.)

$(E-APPC1')$ $(e|_F)\ f \ \triangleright\ (e\ f|_{AT(G,S)})|_{AP(F,S)}$ $(e : G, f : S)$

The rest of the rules need not be changed. We call this modified system λ'_m.

Proposition 44 *λ'_m has subject reduction property.*

As in Section 2.2.1, we define a translation $e^\diamond$ from λ_m to λ'_m.

$$
\begin{array}{lcl}
(x^S)^\diamond & = & x^S \\
{c^\top_{int}}^\diamond & = & c^\top_{int} \\
{c^\top_{bool}}^\diamond & = & c^\top_{bool} \\
(\lambda x^S.\ e)^\diamond & = & \lambda x^S.\ e^\diamond \\
(e\ f)^\diamond & = & e^\diamond\ f^\diamond|_{AT(F,T)}, (\text{Where, } e : F, f : T) \\
\{l_1 = e_1, \ldots, l_n = e_n\}^\diamond & = & \{l_1 = {e_1}^\diamond, \ldots, l_n = {e_n}^\diamond\} \\
e.l^\diamond & = & e^\diamond.l \\
(e|_S)^\diamond & = & e^\diamond|_S \\
(e \vee e')^\diamond & = & e^\diamond \vee e'^\diamond
\end{array}
$$

This translation is almost same as the translation for λ_c in Subsection 2.2.1. The difference is that $(e\ f)^\diamond$ is extended to the function type of λ_m, and the definition for a merge expression is added.

Theorem 45 *When $e : S$ in λ_m, $e^\diamond$ is well typed and has type S in λ'_m.*

Proof: Induction on the structure of e. The only essential case is that e has the form $e\ f$. Let $e : F$ and $f : T$ in λ_m. Then $e^\diamond : F$ and $f^\diamond : T$ in λ'_m from induction hypothesis. $M(F)$ includes $AT(F,T) \longrightarrow AP(F,T)$, and $f^\diamond|_{AT(F,T)}$ has type $AT(F,T)$. Therefore, from (T-APP1'), $e^\diamond\ f^\diamond|_{AT(F,T)}$ is well typed. ∎

4.3.2 Strong Normalization of λ'_m

We prove the strong normalization property of λ'_m extending the proof for λ'_c in Section 2.2.2.

The definition of $Red(S)$, which is a subset of expressions of type S, is as follows.

- $Red(int)$, $Red(bool)$ are sets of expressions which are strongly normalizable.
- $Red(\{l_1 : S_1, \ldots, l_n : S_n\}) = \{e \mid e.l_i \in Red(S_i)\}$.

- $(E_{int})|_{int}$
- E_{int}.

■

The system λ^*_m includes two kinds of orders on functions. One is the order between two monomorphic functions, and the other is the order induced by merging functions. The intuitive reason coercion is not transitive is that the coercions attached to these two kinds of orders have different meaning; the former is to restrict the range of the function, and the later is to select some function components.

4.3 Strong Normalization of λ^*_m

In this section, we prove the strong normalization (in short, SN) of λ^*_m. The proof essentially follows the technique used for λ_c. We will first prove the SN of a system λ'_m which corresponds to λ'_c, and of a system λ''_m which corresponds to λ'_c, then subsume the SN of λ_m to the SN of λ'_m.

4.3.1 λ'_m: A Calculus with Strict Type Agreement

The system λ'_m is defined by modifying the type inference rule and the reduction rule of λ^*_m. It corresponds to λ'_c used in the SN proof λ_c. λ'_c was a calculus which had strict type agreement between the argument type and the domain type of a function, and almost have the flavor of the simply typed lambda calculus. We need to extend the notion of strict type agreement if there exists the merge operator. For example, in the translation from λ_c to λ'_c, we translated the expression $e\ f$ to $e\ f|_S$ when $e : S \longrightarrow S', f : T, T > S$. Now consider the expression $(e_1 \vee e_2)f$ with types $e_1 : S_1 \longrightarrow S'_1, e_2 : S_2 \longrightarrow S'_2, f : T, T > S_1, T > S_2$. In this case, we cannot obtain strict type agreement by coercing f to S_1 nor S_2. Instead of this, we consider the set $M(F)$ as the set of types of F, and regulate that strict type agreement holds for one of them.

Thus, for each function application, we select a function component $S \longrightarrow S'$, an appropriate element of $M(F)$, and require that the argument type and S agree. That is, the type assignment system of λ'_m has the following rule (T-APP1') instead of (T-APP):

$$\text{(T-APP1')}\quad \frac{e : F = [S_1 \longrightarrow S'_1, \ldots, S_n \longrightarrow S'_n] \quad f : T \quad T \longrightarrow T' \in M(F)}{e\ f : T'}.$$

With this modification, the reduction rules also need to be modified. First, we can use the simple rule (E-APP') instead of (E-APP) because the type of f is always equal to S from (T-APP1'). Contrary to this, the rule (E-APPL) is replaced by a more sophisticated rule (E-APPL') in order to preserve the type conformity. The rule (E-APPC1) is also replaced by the rule (E-APPC1').

We will write $\vee_{\{i|S_i<U\}}S_i$ for this term. Thus there is a S_i such that $S_i < AT(F,U)$. Since $G > F$, $\exists V \rightarrow V' \in M(G)$ such that $S_i \rightarrow S_i' < V \rightarrow V'$. Therefore, $S_i > V$. Since V is a least upper bound of some T_j $(j = 1,\ldots,m)$, $AT(F,U) > T_j$ for some j by composing them. Therefore, $AT(G,AT(F,U))$ exists.

$AP(G,AT(F,U)) = \vee_{\{j|T_j<\vee_{\{i|S_i<U\}}S_i\}}T_j'$. $AP(F,U) = \vee_{\{i|S_i<U\}}S_i'$. We only need to prove that if $S_i < U$, then $S_i' < AP(G,AT(F,U))$. Suppose that $S_i < U$, and write S and S' for S_i and S_i'. As we have noted, $\exists V \rightarrow V' \in M(G)$ such that $S \rightarrow S' < V \rightarrow V'$. Therefore, $S > T_{\phi(1)}\vee\ldots\vee T_{\phi(l)}$ and $S' < T'_{\phi(1)}\vee\ldots\vee T'_{\phi(l)}$. Since $T_{\phi(j)} < S, T_{\phi(j)} < \vee_{\{i|S_i<U\}}S_i$ $(j = 1,\ldots,l)$. Therefore, $T'_{\phi(1)}\vee\ldots\vee T'_{\phi(l)} < AP(G,AT(F,U))$. ∎

As we have noted in Chapter 2, the corresponding modification to λ_c does not change the meaning of the reduction system. However, λ_m changes completely with this slight modification. As we will see, $F = [S_1 \rightarrow S_1',\ldots,S_n \rightarrow S_n']$ has the intuitive meaning of a list of monomorphic functions of types in $M(F)$, and λ_m^* has the strong normalization property. However, it does not have the transitivity of coercions, which is one of the most fundamental properties of a calculus with coercions.

4.2 Non-transitivity of Coercions

Proposition 43 *Let $U > G > F$ and $f : U$. $f|_G|_F\ g$ is not, in general, equal to $f|_F\ g$ in λ_m^*.*

Proof: Let F be $\{l_1 : int, l_2 : int\} \rightarrow int$, G be $\{l_1 : int\} \rightarrow int$, and U be $F \vee G$. Then $U > G > F$. Let f_1 be $\lambda x^{\{l_1:int,l_2:int\}}.x.l_2$ and f_2 be $\lambda x^{\{l_1:int\}}.x.l_1$. $f_1 : F$ and $f_2 : G$. Therefore, $f = f_1 \vee f_2 : U$. Let g be $\{l_1 = 1, l_2 = 2\}$ of type $\{l_1 : int, l_2 : int\}$.

$(f|_G)|_F\ g$ reduces as follows:

$$
\begin{aligned}
f|_G|_F\ g &\triangleright (f|_G\ g|_{\{l_1:int,l_2:int\}})|_{int} \\
&\triangleright (f\ g|_{\{l_1:int,l_2:int\}}|_{\{l_1:int\}})|_{int}|_{int} \\
&\triangleright^* (f\ \{l_1 = 1\})|_{int}|_{int} \\
&\triangleright (f_2\ \{l_1 = 1\})|_{int}|_{int} \\
&\triangleright^* ((\{l_1 = 1\}.l_1)|_{int}|_{int} \\
&\triangleright (1|_{int})|_{int} \\
&\triangleright (1|_{int}) \\
&\triangleright 1.
\end{aligned}
$$

On the other hand,

$$
\begin{aligned}
(f|_F)\ g &\triangleright (f\ g|_{\{l_1:int,l_2:int\}})|_{int} \\
&\triangleright (f\ \{l_1 = 1, l_2 = 2\})|_{int} \\
&\triangleright ((f_1\ \{l_1 = 1, l_2 = 2\})\vee(f_2\ \{l_1 = 1, l_2 = 2\}))|_{int} \\
&\triangleright^* (2\vee 1)|_{int}
\end{aligned}
$$

Chapter 4

λ_m^*: Another Calculus with a Merge Operator

As we have studied, the system λ_m did not have the strong normalization property, which many typed calculi have. Moreover, the meaning of a type is different from conventional type systems. In this chapter, we study another calculus with a merge operator. This calculus, λ_m^*, has the strong normalization property. And the meaning of each type is similar to conventional type systems. However, this calculus has the serious problem that coercion functions are not transitive.

4.1 Definition of λ_m^*

Non-standard properties of λ_m are caused by the reduction rule (E-APPC1).

$$\text{(E-APPC1)} \quad (e|_F)\ f \ \triangleright\ (e\ f)|_{AP(F,S)}. \quad (f : S)$$

This rule does not have the notion that only the part of e specified by F is applied. We replace this rule by the rule

$$\text{(E-APPC1}^*\text{)} \quad (e|_F)\ f \ \triangleright\ (e\ f|_{AT(F,S)})|_{AP(F,S)} \quad (f : S)$$

and define a calculus λ_m^*. (E-APPC1*) selects a part of the function e by coercing the argument f to $f|_{AT(F,S)}$.

We need to prove the subject reduction property of λ_m^*.

Proposition 42 *If $e|_F\ f$ is well typed, then $(e\ f|_{AT(F,U)})|_{AP(F,U)}$ is also well typed for U the type of f.*

Proof: Let $e : G = [T_1 \to T_1', \ldots, T_m \to T_m']$, and $F = [S_1 \to S_1', \ldots, S_n \to S_n']$. Since $e|_F$ is well typed, $G > F$. We need to prove that $AT(G, AT(F,U))$ exists and $AP(G, AT(F,U)) > AP(F,U)$. $AT(F,U) = S_{\phi(1)} \vee S_{\phi(l)}$, where $S_{\phi(i)} < U$ $(i = 1, \ldots, l)$.

$f|_F\ e\ \vec{g}$ reduces to c, and vice versa. Suppose that $f\ e\ \vec{g}$ reduces to c. $f|_F\ e\ \vec{g}$ reduces to $(f\ e)|_{AP(F,S)}\ \vec{g}$. Since $AP(F,S)$ has degree less than F, and $f\ e$ is a closed term of type $AP(F,S)$, induction hypothesis applies and $(f\ e)|_{AP(F,S)}\ \vec{g}$ reduces to c. As for the converse, suppose that $f|_F\ e\ \vec{g}$ reduces to c. By Church-Rosser property, we can first reduce $f|_F\ e\ \vec{g}$ to $(f\ e)|_{AP(F,S)}\ \vec{g}$ and then reduce it to c. Therefore, by induction hypothesis, $(f\ e)\ \vec{g}$ reduces to c. ∎

Theorem 41 *When $f : F$ is a closed term and $F > G > U$, then $f|_G|_U$ and $f|_U$ are applicably congruent.*

Proof: By induction on the degree of F. The proof is similar to Theorem 40, and we omit it. ∎

We will leave it an open problem whether $f|_G|_U$ and $f|_U$ are congruent.

It is obvious that $e \approx f$ implies $e \approx_{AP} f$.

Proposition 39 *$Norm(S)$: The set of closed normal forms of type S are as follows:*
$Norm(int) = \{c^{\top}_{int}\} = \{E_{int}, 0, 1, \ldots\}$,
$Norm(bool) = \{c^{\top}_{bool}\} = \{E_{bool}, true, false\}$,
$Norm(\{l_1 : S_1, \ldots, l_n : S_n\}) = \{\{l_1 = c_1, \ldots, l_n = c_n\} | c_{int} \in Norm(S_i)\}$
$Norm(F = [S_1 \longrightarrow S'_1, \ldots, S_n \longrightarrow S'_n]) = \{c_1 \vee \ldots \vee c_k |\ c_1, \ldots, c_k$ *are normal forms*$\} \cup$
$\{c|_F \mid c$ *is a normal form*$\} \cup$
$\{\lambda x^S.c \mid F = [S \longrightarrow S']$ *and* $c : S'$ *is a normal form*$\}$.

Proof: Induction on the structure of a term to prove that these forms cannot be reduced any more, and closed terms of other forms can be reduced. ∎

From this proposition and the Church-Rosser theorem, every closed expression of a basic type is reduced to a unique constant, or do not have a normal form. This justifies the above definition of equality between terms.

Theorem 40 *When $f : F$ is a closed term, f and $f|_F$ are applicably congruent.*

Proof: We define the degree δ of a type as follows:

$$\begin{aligned}
&\delta(int) = \delta(bool) = 0\\
&\delta(\{l_1 : S_1, \ldots, l_n : S_n\}) = Max(\delta(S_i)) + 1\\
&\delta([P_1, \ldots, P_n]) = Max(\delta(P_i))\\
&\delta(S \longrightarrow S') = \delta(S') + 1,
\end{aligned}$$

And prove it by induction on the degree of F.

Suppose that F has degree 0. If f reduces to c, then $f|_F$ reduces to f by (E-COE2) and then reduces to c. If $f|_F$ reduces to c then, by the Church-Rosser property, we can first reduce $f|_F$ to f, and then reduce it to c. Thus, we can form a reduction sequence from f to c.

Suppose that F has the form $\{l_1 : S_1, \ldots, l_n : S_n\}$. A sequence deriving f to a normal form starts with one of the l_i $(i = 1, \ldots, n)$. Thus, we prove that if $f.l_i\ \vec{g}$ reduces to c then $f|_F.l_i\ \vec{g}$ reduces to c, and vice versa. Suppose that $f.l_i\ \vec{g}$ reduces to c by Δ. Then, by the form of c, the reduction includes a step $\{l_1 : e_1, \ldots, l_n : e_n\}.l_i \triangleright e_i$ with $f \triangleright^* \{l_1 : e_1, \ldots, l_n : e_n\}$. By the Church-Rosser property, we may safely suppose that Δ first reduce f to $\{l_1 : e_1, \ldots, l_n : e_n\}$ (Δ'), reduce $\{l_1 : e_1, \ldots, l_n : e_n\}.l_i$ to e_i by (E-LABEL), then $e_i\ \vec{g}$ is reduced to c . By induction hypothesis, $e_i|_{S_i}\ \vec{g}$ also reduces to c (Δ''). Therefore, $f|_F.l_i\ \vec{g}$ reduces to c by Δ' followed by the (E-COE1) step $\{l_1 : e_1, \ldots, l_n : e_n\}|_F \triangleright \{l_1 : e_1|_{S_1}, \ldots, l_n : e_n|_{S_n}\}$ followed by (E-LABEL) and Δ''. The converse is proved similarly.

Suppose that F has the form $[S_1 \longrightarrow S'_1, \ldots, S_n \longrightarrow S'_n]$. A sequence deriving f to a normal form starts with applying $e : S$. Thus, we prove that if $f\ e\ \vec{g}$ reduces to c then

Let $e = (\lambda x^S.x\ x) \vee (\lambda x^i.x) \vee (\lambda x^{i \longrightarrow i}.1)$. e has the type of the right hand side of (3.7). We write this type F. Consider the term $e\ e$. It is type correct because $AT(F,F) = S \vee i \longrightarrow i$. It reduces as follows:

$$\begin{aligned} e\ e &= (\lambda x^S.x\ x) \vee (\lambda x^i.x) \vee (\lambda x^{i \longrightarrow i}.1)\ e \\ &\triangleright\ (\lambda x^S.x\ x)\ e \vee (\lambda x^{i \longrightarrow i}.1)\ e \\ &\triangleright^*\ (e|_S\ e|_S) \vee 1 \\ &\triangleright\ (e\ e|_S)|_{int} \vee 1 \\ &\triangleright^*\ (e|_S|_S\ e|_S|_S \vee 1) \vee 1 \\ &\triangleright^*\ ((e|_S|_S|_S\ e|_S|_S|_S \vee 1) \vee 1) \vee 1 \\ &\ldots. \end{aligned}$$

It is an infinite reduction sequence. One can easily see that every reduction sequence starting $e\ e$ is infinite. Therefore, not only the strong normalization, but also the weak normalization fails in λ_m.

3.3.6 Transitivity of Coercions

In λ_m, when f has type F, functions f and $f|_F$ are expected to be the same function because the coercion function from F to F is identity. However, there is no reduction rule which reduces $f|_F$ to f, or f to $f|_F$. Actually, we do not expect that λ_m is extensional from the beginning because we did not even have the η-rule,

$$\eta : \lambda x.f\ x\ \triangleright\ f.$$

Though f and $f|_F$ are not equal, we expect that they behave equally. For the same reason, we expect that coercions are transitive. That is, $f|_G|_U$ and $f|_U$ behave equally. In this subsection, we give syntactic proof of the equality.

For this purpose, we define a notion of equivalence between terms, which is sometimes called applicable congruence ([Blo90], [Plo77], [Mil77]).

Definition 5 *Two closed terms e and f of a same type are applicably congruent ($e \approx_{AP} f$) if for every sequent of applications and slot selections $\vec{g}$ deriving e and f to ground types, $e\ \vec{g}$ reduces to a normal form c iff $f\ \vec{g}$ reduces to c.*

If e has the type $[\{l_1 : int\} \longrightarrow \{l_1 : int\}, \{l_2 : bool\} \longrightarrow \{l_2 : [int \longrightarrow int]\}, int \longrightarrow int]$, then (1), $(\{l_1 = 1\},\ .l_1)$, $(\{l_2 = true, l_3 = 5\},\ .l_2,\ 5)$ are all sequences of applications which derive e to a ground type.

There is a stronger notion of equivalence.

Definition 6 *Two closed terms of the same type e and f are congruent ($e \approx f$) if for every context $C[\cdot]$, $C[e]$ reduces to a normal form c iff $C[f]$ reduces to c.*

$$
\begin{aligned}
&(\lambda x^S.x\ x)\ (\lambda x^S.x\ x)\\
&\quad \triangleright\ (x\ x)[x := (\lambda x^S.x\ x)|_S]\\
&\quad = (\lambda x^S.x\ x)|_S\ (\lambda x^S.x\ x)|_S\\
&\quad \triangleright\ ((\lambda x^S.x\ x)\ (\lambda x^S.x\ x)|_S)|_G\\
&\qquad\quad (\text{for } G = AP(S \to AP(S,S), S \to AP(S,S)))\\
&\quad \triangleright\ ((x\ x)[x := (\lambda x^S.x\ x)|_S|_S])|_G\\
&\quad = ((\lambda x^S.x\ x)|_S|_S\ (\lambda x^S.x\ x)|_S|_S)|_G\\
&\quad \triangleright\ ((\lambda x^S.x\ x)|_S\ (\lambda x^S.x\ x)|_S|_S)|_{G'}|_G\\
&\quad \triangleright\ ((\lambda x^S.x\ x)\ (\lambda x^S.x\ x)|_S|_S)|_{G''}|_{G'}|_G\\
&\quad \ldots,
\end{aligned}
$$

and we can form an infinite reduction sequence.

For this to be type correct, one needs, in addition to (3.1), the existence of $AT(S \to AP(S,S), S \to AP(S,S))$. It says that $S \to AP(S,S) > S$. Note that $S = [X_1 \to X_1', \ldots, X_n \to X_n'] > X_1 \to X_1'$. Therefore, by composing them, we have $S \to AP(S,S) > X_1 \to X_1'$. Therefore, from Lemma 29, we have

$$S < X_1. \tag{3.3}$$

from this and (3.1), we need to solve

$$X_1 \simeq [X_1 \to X_1', \ldots, X_n \to X_n']. \tag{3.4}$$

This reflective type equation is actually solvable in λ_m. First, we will form a type X which satisfy

$$X > X \to X'. \tag{3.5}$$

This is a special case of (3.3). From $i \to i < [i \to i,\ (i \to i) \to i]$, we have $(i \to i) \to i > [i \to i,\ (i \to i) \to i] \to i$. Therefore,

$$\begin{bmatrix} i & \to & i, \\ i \to i & \to & i \end{bmatrix} > \begin{bmatrix} i & \to & i, \\ i \to i & \to & i \end{bmatrix} \to i \tag{3.6}$$

Since $i \to i$, $(i \to i) \to i$, $[i \to i,\ (i \to i) \to i] \to i$ have the same codomain and thus are pairwise compatible, we have

$$\begin{bmatrix} i & \to & i, \\ (i \to i) & \to & i \end{bmatrix} \simeq \begin{bmatrix} \begin{bmatrix} i & \to & i, \\ (i \to i) & \to & i \end{bmatrix} & \to & i, \\ i & \to & i, \\ (i \to i) & \to & i \end{bmatrix}. \tag{3.7}$$

This is a solution of (3.4).

This is only a necessary condition for (3.2) to be type correct. Actually, (3.2) is not type correct for $S = [i \to i,\ (i \to i) \to i]$, and we leave it as an open problem. However, this type equation suggests another expression which may cause infinite reduction.

As an example, let $e_1 = \lambda x^{\{l_1:int\}}.x.l_1 : \{l_1 : int\} \longrightarrow int$, $e_2 = \lambda x^{\{l_1:int, l_2:int\}}.x.l_2 : \{l_1 : int, l_2 : int\} \longrightarrow int$, and $e = e_1 \vee e_2$. e has the type $F = [\{l_1 : int\} \longrightarrow int, \{l_1 : int, l_2 : int\} \longrightarrow int]$. Since $F > [\{l_1 : int\} \longrightarrow int]$, we can define a function $e' = e|_{[\{l_1:int\} \longrightarrow int]}$. Though, e_1 and e' both have the same type $[\{l_1 : int\} \longrightarrow int]$, and behave equivalently to the type $\{l_1 : int\}$, they behave differently to arguments of type $\{l_1 : int, l_2 : int\}$ and bigger types. Actually, for $f = \{l_1 = 1, l_2 = 2\}$, $e'\ f$ reduces as follows:

$$
\begin{aligned}
e'\ f &= (e_1 \vee e_2)|_{\{l_1:int, l_2:int\}}\ f \\
&\triangleright\ ((e_1 \vee e_2)\ f)|_{int} \\
&\triangleright\ e_1\ f \vee e_2\ f \\
&\triangleright^*\ \{l_1 = 1\}.l_1 \vee \{l_1 = 1, l_2 = 2\}.l_2 \\
&\triangleright^*\ 1 \vee 2 \\
&\triangleright^*\ E_{int}.
\end{aligned}
$$

On the other hand, $e_1\ f$ reduces to 1. Thus, e_1 and e' are different functions though they behave equivalently to $\{l_1 : int\}$.

Therefore, we must think of $[S \longrightarrow S']$ not as the type of functions from S to T, but as the type of functions from $\cup_{\{T>S\}} T$ to S'. This new meaning of a function type makes the type system completely different from conventional type systems. For instance, the type structure no longer forms a Cartesian Closed Category because there are functions which cannot be defined by currying, and as we will see in the next subsection, strong normalization fails.

3.3.5 Non-normalizing Terms

It is rather surprising that λ_m does not have the strong or weak normalization property, though it is a typed language. Here, we will give an example of a term which does not have a normal form. In this subsection, we will write i for int, and omit parenthesis [,] for au function type with one component.

First, we show that $(\lambda x^S.x\ x)$ is a well-typed term for some S. For this term to be type correct, we need that $AT(S, S)$ exists. Let S be $[X_1 \longrightarrow X'_1, \ldots, X_n \longrightarrow X'_n]$. Then this condition says that at least one of the X_i, say X_1, is smaller than S. That is, the inequality

$$X_1 < [X_1 \longrightarrow X'_1, \ldots, X_n \longrightarrow X'_n]. \tag{3.1}$$

is solvable. We can solve this. For example, $i \longrightarrow i < [i \longrightarrow i,\ (i \longrightarrow i) \longrightarrow i]$. Thus, $\lambda x^{[i \longrightarrow i,\ (i \longrightarrow i) \longrightarrow i]}.x\ x$ is a well typed expression. It shows that some terms are self-applicable in λ_m.

Next, we try to form a term

$$(\lambda x^S.x\ x)\ (\lambda x^S.x\ x). \tag{3.2}$$

If this term is well-typed, then

3.3.3 Non-preservation of Coercions

One thing to note about λ_m is that functions do not preserve coercions. We say that a function $f : F$ preserves coercions if $(f\ e|_S)$ is equivalent to $(f\ e)|_T$ for T the type of $(f\ e|_S)$. Though we have not defined the precise notion of equivalence between terms, we can say that they are not equivalent in any way. (In 3.3.6, we will define the equivalence between terms.)

Example: Consider the function term $e = (\lambda x^{\{l_1:int\}}.x.l_1) \vee (\lambda x^{\{l_2:int\}}.x.l_2)$ and the term $f = \{l_1 = 1, l_2 = 2\} : \{l_1 : int, l_2 : int\}$. $f|_{\{l_1:int\}} = \{l_1 = 1\}$. $e\ f$ reduces to E_{int} and $e\ f|_{\{l_1:int\}}$ reduces to 1. E_{int} coerced to int is E_{int}. Therefore, it does not preserve coercions.

When we are programming in an object oriented language with multiple inheritance, we frequently encounter this kind of situation. For example, suppose that one is programming using multiple inheritance, and methods with the same name exist in two super classes. When a generic function is applied, both methods are activated and they interact to cause an unexpected result by sharing the same variable. This is not an expected behavior of a program, and such a programming error is difficult to detect. In λ_m, in order to avoid such a situation, an error value is returned when two methods return uncompatible results, and this is the reason why functions do not preserve coercions. Therefore, though functions do not preserve coercions, the results are related. Actually, they are related by the order induced by the error value, as we will show in Chapter 5.

3.3.4 The Meaning of a Function Type

As we have noted, an intuitive meaning of a function type $F = [S_1 \longrightarrow S'_1, \ldots, S_n \longrightarrow S'_n]$ is a set of functions which are applicable to types $\{T |\ T > S_i \text{ for some } i\}$ and the return values have the type $AP(F,T)$, which changes according to T. In other words, we consider the partially ordered set of types $\mathcal{T}$, and consider the monotonic function space $[\mathcal{T} \longrightarrow \mathcal{T}]$. It is also a partially ordered set. If we regard the bottom element $\bot^{\mathcal{T}}$ of $\mathcal{T}$ as the special type which means the type error, then $[S \longrightarrow S']$ is a step function in $[\mathcal{T} \longrightarrow \mathcal{T}]$. That is, it returns S' if the argument is bigger than S, and otherwise returns $\bot^{\mathcal{T}}$. And the meaning of $[S_1 \longrightarrow S'_1, \ldots, S_n \longrightarrow S'_n]$ is the least upper bound of $[S_i \longrightarrow S'_i]$ $(i = 1, \ldots, n)$.

When we apply it to a simple function type, an intuitive meaning of a type $[S \longrightarrow S']$ is a set of functions which are applicable to types $T > S$ and the return values have the type S'. On the other hand, the meaning of $S \longrightarrow S'$ in conventional typed calculi is a set of functions from S to S'. Thus it is expected that $[S \longrightarrow S']$ can also be considered conventionally. That is, though a function of this type can be applied to types $T > S$, the behavior of this function is completely determined by its behavior to the type S, by first coercing the argument to S and then applying f to the value in S. It holds if a function preserves coercions. However, as we have seen in the previous subsection, a function does not preserve coercions.

Theorem 38 *For a term g, we define g^* to be the term constructed by applying the rules of the name (P-R$*$) when possible, and the rules of the name (P-F$*$) when no (P-R$*$) can be applied. g^* is uniquely determined and if $g \Rightarrow g'$, then $g' \Rightarrow g^*$.*

Proof: g^* is uniquely determined because none of the $(P - R*)$ rules overlap. The rest of the theorem is proved using structural induction on the form of the term g.

(Case 1,2,3) g has the form x^S, $c^{\top}_{int}$, $c^{\top}_{bool}$. Then obvious.

(Case 4) g has the form $\lambda x^S.\ e$. Then g' has the form $\lambda x^S.\ e'$ for some e' to which $e \Rightarrow e'$. In the following proof, a symbol with $'$ means some term to which the symbol without $'$ parallel-reduces. For example, $e \Rightarrow e'$ and $f \Rightarrow f'$. By induction hypothesis, $e' \Rightarrow e^*$. Therefore, $\lambda x^S.\ e' \Rightarrow \lambda x^S.\ e^*$. On the other hand, from the definition, $(\lambda x^S.\ e)^* = \lambda x^S.\ e^*$.

(Case 5) g has the form $e_0\ f$. Then there are four subcases each of which has one or two subcases.

(Subcase 5.1) g has the form $(\lambda x^S.\ e)\ f$. Then g^* is $e^*[x^S := f^*|_S]$.

(Subsubcase 5.1.1) g' has the form $(\lambda x^S.\ e')\ f'$. Then, from induction hypothesis, $e' \Rightarrow e^*$ and $f' \Rightarrow f^*$. Thus, from definition, $(\lambda x^S.\ e')\ f' \Rightarrow e^*[x^S := f^*|_S]$.

(Subsubcase 5.1.2) g' has the form $e'[x^S := f'|_S]$. Then, from induction hypothesis, $e' \Rightarrow e^*$ and $f' \Rightarrow f^*$, which implies $f'|_S \Rightarrow f^*|_S$. Therefore, from Lemma 37, $e'[x^S := f'|_S] \Rightarrow e^*[x^S := f^*|_S]$.

(Subcase 5.2) g has the form $(e_1 \vee \ldots \vee e_n)\ f$. Then g^* is $e^*_{\phi(1)}\ f^* \vee \ldots \vee e^*_{\phi(l)}\ f^*$, where $\phi(i)$ are selected so that $AT(F_{\phi(i)}, S)$ exists for $F_{\phi(i)}$, the type of $e_{\phi(i)}$, and S, the type of f.

(Subsubcase 5.2.1) g' has the form $(e'_1 \vee \ldots \vee e'_n)\ f'$. Then, because the type of an expression does not change in parallel reduction, this case is proved similarly to (Subcase 5.1.2).

(Subsubcase 5.2.2) g' has the form $e'_{\phi(1)}\ f' \vee \ldots \vee e'_{\phi(l)}\ f'$. From induction hypothesis, $e'_i \Rightarrow e^*_i$ and $f' \Rightarrow f^*$. Thus, $e'_{\phi(1)}\ f' \vee \ldots \vee e'_{\phi(l)}\ f'$ parallel reduces to $e^*_{\phi(1)}\ f^* \vee \ldots \vee e^*_{\phi(l)}\ f^*$ by (P-FAPP) and (P-FCOE).

(Subcase 5.3) g has the form $(e|_F)\ f$. Then g^* is $(e^*\ f^*)|_{AP(F,S)}$, here S is the type of f. The proof is similar to (Subcase 5.1) and (Subcase 5.2).

(Subcase 5.4) g does not have the forms of (Subcase 5.1, 5.2, 5.3). Then g^* is $e^*_0\ f^*$, and g' has the form $e'_0\ f'$. This case is proved similarly to (Subcase 5.1.1).

(Case 6) g has the form $\{l_1 = e_1, \ldots, l_n = e_n\}$. This case is proved similarly to (Case 4).

(Case 7) g has the form $e.l$. This case is subdivided into the case e has the form $\{l_1 = e_1, \ldots, l_n = e_n\}$, and the case e does not have this form, and proved similarly to (Case 5).

(Case 8) g has the form $e|_S$. This case is subdivided into the case e has the form $\{l_1 = e_1, \ldots, l_n = e_n\}$, the case e has type *int* or *bool*, and the other cases, and proved similarly to (Case 5).

(Case 9) g has the form $e_1 \vee e_2$. This case is subdivided into the case e_1 and e_2 has the form $\{l_1 = e_1, \ldots, l_n = e_n\}$, the case e_1 and e_2 are constants of type *int* or *bool*, and the other cases, and proved similarly to (Case 5). ∎

	if $e_i \Rightarrow e_i'$ $(i = 1, \ldots, m), \hat{e}_i \Rightarrow \hat{e_i'}$ $(i = 1, \ldots, k)$, $f_i \Rightarrow f_i'$ $(i = 1, \ldots, m), \check{f}_i \Rightarrow \check{f}_i'$ $(i = 1, \ldots, n)$
(P-RCONST1) $n \vee n \Rightarrow n$	if n is a constant in int or $bool$
(P-RCONST2) $n_1 \vee n_2 \Rightarrow E_{int}$	if n_1 and n_2 are different constants in int
(P-RCONST3) $n_1 \vee n_2 \Rightarrow E_{bool}$	if n_1 and n_2 are different constants in $bool$

The first nine rules with names (P-F$*$) follow the structure of pseudo terms. The rest of the rules with names (P-R$*$) correspond to the reduction rules of λ_m. From these rules, the followings are obvious. (Proved using induction on the structure of a term.)

Lemma 37 *1. $e \Rightarrow e$ for any e.*

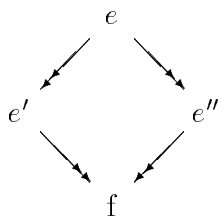
2. If $e \triangleright e'$, then $e \Rightarrow e'$.

3. If $e \Rightarrow e'$, then $e \triangleright^* e'$.

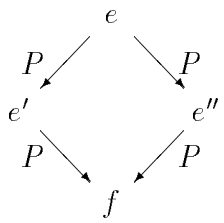
4. If $e : S$ and $e \Rightarrow e'$, then $e' : S$.

5. If $e \Rightarrow e'$, $f \Rightarrow f'$, and $f : S$, then $e[x^S := f] \Rightarrow e'[x^S := f']$.

Recall that the Church-Rosser property says that if $e \triangleright^* e'$ and $e \triangleright^* e''$, then there is a f to which $e' \triangleright^* f$ and $e'' \triangleright^* f$.



From the above lemma, Church-Rosser property of $\triangleright$ is equivalent to the Church-Rosser property of $\Rightarrow$. Church-Rosser property of $\Rightarrow$ follows from the following diamond property of $\Rightarrow$, which says if $e \Rightarrow e'$ and $e \Rightarrow e''$, then there is a f to which $e'' \Rightarrow f$ and $e'' \Rightarrow f$.



Actually, f does not depend on the choice of e' and e'', and is determined only by e. Therefore, we prove the following:

Then the order used in the definition of $M(F), AT(G,S)$, and $AP(G, AT(G,S))$ has smaller degree than $G > F$. Thus, we can avoid the circularity.

Since we only have three rules (I-REFL'), (I-LABEL), and (I-FUN'), which only applies to basic types, record types, and function types respectively, these rules give us necessary and sufficient condition. Thus, the decidability follows:

Theorem 36 *It is decidable whether $S > T$ or not.*

3.3.2 Church-Rosser Property

We prove Church-Rosser property using the notion of parallel reduction due to Tait and Martin-Löf. We follow the construction in [Tak92].

We define parallel reduction $\Rightarrow$ on terms following the structure of a term. $\triangleright$ was defined as a reduction rule; a part of a term which matches to a pattern can be changed. On the other hand, $\Rightarrow$ is defined inductively on the whole term. An intuitive meaning of $\Rightarrow$ is to reduce simultaneously many redexes which may be nesting.

(P-FVAR) $x^S \Rightarrow x^S$

(P-FCONSI) $c^{\top}_{int} \Rightarrow c^{\top}_{int}$

(P-FCONSB) $c^{\top}_{bool} \Rightarrow c^{\top}_{bool}$

(P-FABS) $\lambda x^S.\ e \Rightarrow \lambda x^S.\ e'$ if $e \Rightarrow e'$

(P-FAPP) $e\ f \Rightarrow e'\ f'$ if $e \Rightarrow e'$ and $f \Rightarrow f'$

(P-FRCD) $\{l_1 = e_1, \ldots, l_n = e_n\} \Rightarrow \{l_1 = e'_1, \ldots, l_n = e'_n\}$ if $e_i \Rightarrow e'_i (i = 1, \ldots, n)$

(P-FSELECT) $e.l \Rightarrow e'.l$ if $e \Rightarrow e'$

(P-FCOE) $e|_S \Rightarrow e'|_S$ if $e \Rightarrow e'$

(P-FLUB) $e_1 \vee e_2 \Rightarrow e'_1 \vee e'_2$ if $e_1 \Rightarrow e'_1, e_2 \Rightarrow e'_2$

(P-RLABEL) $\{l_1 = e_1, \ldots, l_n = e_n\}.l_i \Rightarrow e'_i$ if $e_i \Rightarrow e'_i$

(P-RAPP) $(\lambda x^S.\ e)\ f \Rightarrow e'[x^S := f'|_S]$ if $e \Rightarrow e', f \Rightarrow f'$

(P-RAPPL) $(e_1 \vee \ldots \vee e_n)\ f \Rightarrow e'_{\phi(1)}\ f' \vee \ldots \vee e'_{\phi(l)}\ f'$
(Where $\phi(i)$ are selected so that $AT(F_{\phi(i)}, S)$ exists for $F_{\phi(i)}$, the type of $e_{\phi(i)}$ and S, the type of f)
if $e_{\phi(i)} \Rightarrow e'_{\phi(i)} (i = 1, \ldots, l), f \Rightarrow f'$

(P-RAPPC1) $(e|_F)\ f \Rightarrow (e'\ f')|_{AP(F,S)}$
(S is the type of f) if $e \Rightarrow e'$ and $f \Rightarrow f'$

(P-RCOE1) $\{l_1 = e_1, \ldots, l_n = e_n\}|_{\{l_1:S_1, \ldots, l_m:S_m\}} \Rightarrow \{l_1 = e'_1|_{S_1}, \ldots, l_m = e'_m|_{S_m}\}$ if $e_i \Rightarrow e'_i\ (i = 1, \ldots, m)$

(P-RCOE2) $e|_N \Rightarrow e' \quad (N = int, bool)$ if $e \Rightarrow e'$

(P-RLUBR) $\{l_1{=}e_1, \ldots, l_m{=}e_m, \hat{l_1}{=}\hat{e_1}, \ldots \hat{l_k}{=}\hat{e_k}\} \vee \{l_1{=}f_1, \ldots, l_m{=}f_m, \check{l_1}{=}\check{f_1}, \ldots, \check{l_n}{=}\check{f_n}\} \Rightarrow$
$\{l_1 = e'_1 \vee f'_1, \ldots, l_m = e'_m \vee f'_m, \hat{l_1} = \hat{e'_1}, \ldots \hat{l_k} = \hat{e'_k}, \check{l_1} = \check{f'_1}, \ldots, \check{l_n} = \check{f'_n}\}$

Proof: If part: $S_i \longrightarrow AP(F, S_i) \in M(F)$. From the assumption, there exists, for each i, $U_i \longrightarrow U_i' \in M(G) s.t. [U_i \longrightarrow U_i'] > [S_i \longrightarrow AP(F, S_i)]$. From Lemma 33,(2), $G > [U_i \longrightarrow U_i']$. Therefore, $G > [S_i \longrightarrow S_i']$. Thus, from Lemma 26, $G > F$.

Only if part: For each element $S \longrightarrow AP(F, S) \in M(F)$, where $S = S_{\phi(1)} \vee \ldots \vee S_{\phi(l)}$, $AT(F, S) \simeq S$ and $AP(F, AT(F, S)) = AP(F, S)$. Thus it suffices to prove the statement when $S \longrightarrow S'$ has the form $AT(F, S) \longrightarrow AP(F, AT(F, S))$. More generally, we will prove that for all compatible subset $\{S_{\phi(1)}, \ldots, S_{\phi(l)}\}$ of $\{S_1, \ldots, S_n\}$, there exists $U \longrightarrow U' \in M(G)$ $s.t.$ $P = [S_{\phi(1)} \vee \ldots \vee S_{\phi(l)} \longrightarrow S'_{\phi(1)} \vee \ldots \vee S'_{\phi(l)}] < [U \longrightarrow U']$. From Theorem 32, it is true if $P = [S_{\phi(i)} \longrightarrow S'_{\phi(i)}]$, and put $[U_i \longrightarrow U_i']$ be the $[U \longrightarrow U']$ for this case. $[U_i \longrightarrow U_i']$ $(i = 1, \ldots, l)$ are upper bounded by G as in the proof of Theorem 32. Thus Lemma 30 applies, and we get $[S_{\phi(1)} \vee \ldots \vee S_{\phi(l)} \longrightarrow S'_{\phi(1)} \vee \ldots \vee S'_{\phi(l)}] < [U_{\phi(1)} \vee \ldots \vee U_{\phi(l)} \longrightarrow U'_{\phi(1)} \vee \ldots \vee U'_{\phi(l)}]$, which is equivalent to some $[T_{\psi(1)} \vee \ldots \vee T_{\psi(l)} \longrightarrow T'_{\psi(1)} \vee \ldots \vee T'_{\psi(l)}]$. By putting $T = T_{\psi(1)} \vee \ldots \vee T_{\psi(l)}$, $T \longrightarrow AP(F, T) \in M(G)$ is greater than this.

The maximum T: Suppose T^0 and T^1 satisfies the condition. Since $T^0 < S$ and $T^1 < S$, $T^0 \vee T^1$ exists and it is equivalent to the domain of some element of $M(G)$ by rearranging the order.

Equivalence to $AT(G, S)$: First, $AT(G, S) < S$, and $AT(G, S)$ is the maximum element which is less than S and which appears as the left hand side of a member of $M(G)$.

We need to show that $AP(G, AT(G, S)) > S'$. It holds because $AT(G, S) > T$ and $AP(G, T) > S'$ holds for the maximum T, and because AP is monotonic from the definition. ∎

From Theorem 35 and Lemma 34, We consider an alternative rule (I-FUN') for the order relation:

$$\text{(I-FUN')} \qquad \frac{\text{for each } S \longrightarrow S' \in M(F),\ AT(G, S) \text{ exists and } AP(G, S) > S'.}{G > F}$$

and consider another set of rules for the coercible order of λ_m as (I-REFL'), (I-LABEL), (I-FUN'). Here, (I-REFL') is the restriction of the (I-REFL) to *int* and *bool*. From theorem 35, it is easily verified that this system is equivalent to the original one. Since the definition of $M(F), AT(G, S)$, and $AP(G, S)$ depends on the order relation, one must care for circularity in the definition. However, there is no circularity because if we assign a degree Deg of a type as follows:

$$\begin{aligned}
&Deg(int) = Deg(bool) = 0 \\
&Deg(\{l_1 : S_1, \ldots, l_n : S_n\}) = Max(Deg(S_i)) \\
&Deg([S_1 \longrightarrow S_1', \ldots, S_n \longrightarrow S_n']) = Max(Deg([S_i \longrightarrow S_i'])) \\
&Deg([S \longrightarrow S']) = Max(Deg(S_i), Deg(S_i')) + 1,
\end{aligned}$$

and degree Deg of an order as

$$Deg(S > S') = max(Deg(S), Deg(S')).$$

$[V_{\psi(j)} \rightarrow V_{\psi'(j)}]$ $(j = 1, \ldots, l)$ are upper bounded and hence compatible from Lemma 26. Thus Lemma 30 applies, and we get $[(V_1 \vee \ldots \vee V_{\psi(l_0)}) \rightarrow (V'_1 \vee \ldots \vee V'_{\psi(l_0)})] > [(U_1 \vee \ldots \vee U_{\psi(l_0)}) \rightarrow (U'_1 \vee \ldots \vee U'_{\psi(l_0)})]$. From (I-TRANS), $[(V_1 \vee \ldots \vee V_{\psi(l_0)}) \rightarrow (V'_1 \vee \ldots \vee V'_{\psi(l_0)})] > [S \rightarrow S']$. The left hand side can be reordered to $\vee$ of T_i's using Lemma 24 and thus well proved. ∎

We define to each function type F, the set $M(F)$ of the argument-result-type pairs which describe the behavior of F.

Definition 4 *Let F be $[S_1 \rightarrow S'_1, \ldots, S_n \rightarrow S'_n]$. Define $M(F)$ to be the set of function components $\{S \rightarrow AP(F,S) \mid S = S_{\phi(1)} \vee \ldots \vee S_{\phi(l)}, S_{\phi(i)}$ $(i = 1, \ldots, l)$ are pairwise compatible.$\}$.*

For example, when $F = [\{l_1 : int\} \rightarrow int, \{l_2 : int\} \rightarrow int]$, $M(F) = \{\{l_1 : int\} \rightarrow int, \{l_2 : int\} \rightarrow int, \{l_1 : int, l_2 : int\} \rightarrow int]$.

Lemma 33 *1) $S > AT(F,S)$.*
2) $S \rightarrow S' \in M(G)$, then $G > [S \rightarrow S']$. 3)If $T > S$ then $AT(F,T) > AT(F,S)$, $AP(F,T) > AP(F,S)$.
4)If $G > F$ then $AP(G,S) > AP(F,S)$.

Proof: 1), 2), 3) are Trivial.
4) Let G be $[T_1 \rightarrow T'_1, \ldots, T_n \rightarrow T'_n]$, and F be $[S_1 \rightarrow S'_1, \ldots, S_n \rightarrow S'_n]$. Suppose that $S > S_i$. From Theorem 32, $[S_i \rightarrow S'_i] < [(T_{\phi(1)} \vee \ldots \vee T_{\phi(l)}) \rightarrow (T'_{\phi(1)} \vee \ldots \vee T'_{\phi(l)})]$, for some $\phi(1), \ldots, \phi(l)$. It means $T_{\phi(j)} < S_i < S$ $(j = 1, \ldots, l)$. Thus, $AP(G,S)$ includes $T'_{\phi(j)}$ $(j = 1, \ldots, l)$ and thus is greater than S'_i. Therefore, every component of $AP(F,S)$ is bounded by $AP(G,S)$, and thus $AP(G,S) > AP(F,S)$. ∎

Lemma 34 *1) AP(F, AT(F,S)) = AP(F,S).*
2)If $G = F \vee F'$, then $AT(F, AT(G,S)) = AT(F,S)$ and $AP(F, AT(G,S)) = AP(F,S)$.

Proof: 2) Let F be $[S_1 \rightarrow S'_1, \ldots, S_n \rightarrow S'_n]$, and F' be $[T_1 \rightarrow T'_1, \ldots, T_m \rightarrow T'_m]$. Let $S_{\phi(i)}$ $(i = 1, \ldots, l)$ satisfy $S_{\phi(i)} < S$. We only need to prove that $S_{\phi(i)} < AT(G,S)$. But it is trivial because $AT(G,S)$ has the form $AT(G,S) = S_{\phi(1)} \vee \ldots \vee S_{\phi(l)} \vee T$ for some T. ∎

Note that Lemma 34, (2) cannot be generalized to the case when $G > F$. A counter example is given in Section 3.3.3.

Theorem 35 *Let $F = [S_1 \rightarrow S'_1, \ldots, S_n \rightarrow S'_n]$ and $G = [T_1 \rightarrow T'_1, \ldots, T_n \rightarrow T'_n]$. $G > F$ if and only if $\forall S \rightarrow S' \in M(F)$, $\exists T \rightarrow T' \in M(G)$ s.t. $T < S and T' > S'$. Especially, there exists a maximum T for which $T \rightarrow T' \in M(G)$ satisfies this condition. This maximum T is equal to $AT(G,S)$.*

Lemma 30 *If* $[T_i \longrightarrow T_i'] > [S_i \longrightarrow S_i']$ $(i = 1, \ldots, n)$, S_i $(i = 1, \ldots, n)$ *are pairwise compatible, and* $T_i \longrightarrow T_i'$ $(i = 1, \ldots, n)$ *are pairwise compatible, then* T_i $(i = 1, \ldots, n)$, T_i' $(i = 1, \ldots, n)$, *and* S_i' $(i = 1, \ldots, n)$ *are pairwise compatible sets and* $[(T_1 \vee \ldots \vee T_n) \longrightarrow (T_1' \vee \ldots \vee T_n')] > [(S_1 \vee \ldots \vee S_n) \longrightarrow (S_1' \vee \ldots \vee S_n')]$.

Proof: We simply prove the case n = 2. From Lemma 29, $T_1 < S_1$ and $T_2 < S_2$. From this and $S_1 \uparrow S_2$, we have $T_1 \uparrow T_2$ and $T_1 \vee T_2 < S_1 \vee S_2$ by Lemma 26. On the other hand, $T_1 \uparrow T_2$ and $(T_1 \longrightarrow T_1') \uparrow (T_2 \longrightarrow T_2')$ implies $T_1' \uparrow T_2'$. Thus, similarly, $T_1' \vee T_2' > S_1' \vee S_2'$. From (I-FUN), $[(T_1 \vee T_2) \longrightarrow (T_1' \vee T_2')] > [(S_1 \vee S_2) \longrightarrow (S_1' \vee S_2')]$. ∎

Lemma 31 *If* S_i $(i = 1, \ldots, n)$ *are pairwise compatible and* $S_i \longrightarrow S_i'$ $(i = 1, \ldots, n)$ *are pairwise compatible, then* $S_1', \ldots, S_n'$ *are pairwise compatible and* $[S_1 \longrightarrow S_1', \ldots, S_n \longrightarrow S_n'] > [(S_1 \vee \ldots \vee S_n) \longrightarrow (S_1' \vee \ldots \vee S_n')]$.

Proof: Induction on n. From (I-FLIST2), it is true when $n = 2$. Suppose that it is true for $n - 1$. From (I-FLIST1) and (I-FLIST3), $[S_1 \longrightarrow S_1', \ldots, S_n \longrightarrow S_n'] > [S_1 \longrightarrow S_1', \ldots, S_{n-1} \longrightarrow S_{n-1}']$. From induction hypothesis, $[S_1 \longrightarrow S_1', \ldots, S_{n-1} \longrightarrow S_{n-1}'] > [(S_1 \vee \ldots \vee S_{n-1}) \longrightarrow (S_1' \vee \ldots \vee S_{n-1}')]$, and from (I-TRANS), $[S_1 \longrightarrow S_1', \ldots, S_n \longrightarrow S_n'] > [(S_1 \vee \ldots \vee S_{n-1}) \longrightarrow (S_1' \vee \ldots \vee S_{n-1}')]$. On the other hand, $[S_1 \longrightarrow S_1', \ldots, S_n \longrightarrow S_n'] > [S_n \longrightarrow S_n']$. Therefore, from Lemma 26, $[S_1 \longrightarrow S_1', \ldots, S_n \longrightarrow S_n'] > [(S_1 \vee \ldots \vee S_{n-1}) \longrightarrow (S_1' \vee \ldots \vee S_{n-1}')] \vee [S_n \longrightarrow S_n]$. Since $S_1, \ldots, S_n$ are pairwise compatible, From Lemma 23, (6), $S_1 \vee \ldots \vee S_{n-1}$ and S_n are pairwise compatible. Thus, $S_1 \vee \ldots \vee S_n$ exists, and from (I-FLIST2), $[(S_1 \vee \ldots \vee S_{n-1}) \longrightarrow (S_1' \vee \ldots \vee S_{n-1}')] \vee [S_n \longrightarrow S_n] > [(S_1 \vee \ldots \vee S_n) \longrightarrow (S_1' \vee \ldots \vee S_n')]$. From this and (I-TRANS), $[S_1 \longrightarrow S_1', \ldots, S_n \longrightarrow S_n'] > [(S_1 \vee \ldots \vee S_n) \longrightarrow (S_1' \vee \ldots \vee S_n')]$. ∎

Theorem 32 *Suppose that* $[T_1 \longrightarrow T_1', \ldots, T_n \longrightarrow T_n'] > [S_1 \longrightarrow S_1', \ldots, S_m \longrightarrow S_m']$. *Then for each* i $(i = 1, \ldots, m)$, $[(T_{\phi(1)} \vee \ldots \vee T_{\phi(l)}) \longrightarrow (T_{\phi(1)}' \vee \ldots \vee T_{\phi(l)}')] > [S_i \longrightarrow S_i']$ *for some* $\phi(1)$ *to* $\phi(l)$.

Proof: Induction on the structure of the proof of $[T_1 \longrightarrow T_1', \ldots, T_n \longrightarrow T_n'] > [S_1 \longrightarrow S_1', \ldots, S_m \longrightarrow S_m']$. If the last rule is (I-REFL), (I-FUN),(I-FLIST1), (I-FLIST2), or (I-FLIST3), then it is proved easily. Suppose that the last rule is (I-TRANS). That is, let Σ and Π be the proof of $[T_1 \longrightarrow T_1', \ldots, T_n \longrightarrow T_n'] > [U_1 \longrightarrow U_1', \ldots, U_k \longrightarrow U_k']$ and $[U_1 \longrightarrow U_1', \ldots, U_k \longrightarrow U_k'] > [S_1 \longrightarrow S_1', \ldots, S_m \longrightarrow S_m']$ respectively, and suppose that the proof is (I-TRANS)(Σ, Π). Fix an i $(i = 1, \ldots, m)$ and let $S = S_i, S' = S_i'$. From induction hypothesis, $[(T_{\phi(j,1)} \vee \ldots \vee T_{\phi(j,l(j))}) \longrightarrow (T_{\phi(j,1)}' \vee \ldots \vee T_{\phi(j,l(j))}')] > [U_j \longrightarrow U_j']$ for each j $(j = 1, \ldots, k)$, and $[(U_{\psi(1)} \vee \ldots \vee U_{\psi(l_0)}) \longrightarrow (U_{\psi(1)}' \vee \ldots \vee U_{\psi(l_0)}')] > [S \longrightarrow S']$. Let V_j be $(T_{\phi(j,1)} \vee \ldots \vee T_{\phi(j,l(j))})$, and V_j' be $(T_{\phi(j,1)}' \vee \ldots \vee T_{\phi(j,l(j))}')$. In order to apply Lemma 30, we must prove that $V_{\psi(j)} \longrightarrow V_{\psi'(j)}$ $(j = 1, \ldots, l)$ are pairwise compatible. $[V_{\psi(j)} \longrightarrow V_{\psi'(j)}] < [T_{\phi(j,1)} \longrightarrow T_{\phi(j,1)}', \ldots, T_{\phi(j,l(j))} \longrightarrow T_{\phi(j,l(j))}'] < [T_1 \longrightarrow T_1', \ldots, T_n \longrightarrow T_n']$ from Lemma 31. Therefore,

that the meaning of an expression does not depend the order by which the expression is reduced.

We will also prove that coercion in transitive, and that there is a term that does not have a Normal Form.

3.3.1 Decidability of the Coercible Order

The coercible order of λ_m was constructed from a set of primitive rules which can be easily understood. However, the existence of the rule (I-TRANS), which is not structural, makes the proof of $T > S$ untrivial. In order to prove that $>$ is decidable, we need some structural condition for $T > S$ which is necessary and sufficient. The condition will be given in Theorem 32.

Lemma 29 *$[T \longrightarrow T'] > [S_1 \longrightarrow S'_1, \ldots, S_n \longrightarrow S'_n]$ implies $T < S_i$ and $T' > S'_i$ $(i = 1, \ldots, n)$. As a special case, $[T \longrightarrow T'] > [S \longrightarrow S']$ implies $T < S$ and $T' > S'$.*

Proof: We write a proof with the last rule (I-TRANS) with subproofs Π and Σ as (I-TRANS)(Π, Σ). When (I-TRANS)$(\Pi 1,$ (I-TRANS)$(\Pi 2, \Pi 3))$ is a proof, then (I-TRANS)((I-TRANS)$(\Pi 1, \Pi 2)$, $\Pi 3)$ is also a proof of the same order relation. Therefore, we may safely suppose that when (I-TRANS) is used, the right premise of (I-TRANS) is proved without using (I-TRANS) as the last rule.

We prove by induction on the number of rules used in the proof of $[T \longrightarrow T'] > [S_1 \longrightarrow S'_1, \ldots, S_n \longrightarrow S'_n]$. The last step of the proof should be (I-REFL), (I-FUN), (I-FLIST3) or (I-TRANS). If it is (I-REFL) or (I-FUN), then the proof is trivial. Suppose that the last rule is (I-FLIST3). Then induction hypothesis applies to the proof of $[T \longrightarrow T'] > [S_i \longrightarrow S'_i]$, and thus $T < S_i$ and $T' > S'_i$.

Suppose that the last rule is (I-TRANS). That is, let Σ and Π be the proofs of $[T \longrightarrow T'] > [U_1 \longrightarrow U'_1, \ldots, U_m \longrightarrow U'_m]$ and $[U_1 \longrightarrow U'_1, \ldots, U_m \longrightarrow U'_m] > [S_1 \longrightarrow S'_1, \ldots, S_n \longrightarrow S'_n]$ respectively, and the present proof be (I-TRANS)(Σ, Π). By applying induction hypothesis to Σ, $T < U_i, T' > U'_i$ $(i = 1, \ldots, m)$. The last rule of Π may be (I-REFL), (I-FUN), (I-FLIST1), (I-FLIST2), and (I-FLIST3). The case (I-REFL) and (I-FLIST1) are trivial. Suppose $m = n = 1$ and the last rule of Π is (I-FUN). Then $U_1 < S_1$, and $U'_1 > S'_1$. Thus, $T < S_1$ and $T' > S'_1$. Suppose $m = 2$, $n = 1$, and the last rule of Π is (I-FLIST2). From induction hypothesis, $T < U_1$, $T < U_2$, $T' > U'_1$, and $T' > U'_2$. Thus $T < U_1 \vee U_2$, and also $T' > U'_1 \vee U'_2$ by Lemma 26. Suppose that the last rule of Π is (I-FLIST3) and $\Pi =$(I-FLIST3)$(\Pi_1, \ldots, \Pi_n)$, where Π_i is the proof of $[U_1 \longrightarrow U'_1, \ldots, U_m \longrightarrow U'_m] > [S_i \longrightarrow S'_i]$ $(i = 1, \ldots, n)$. Then consider a proof (I-TRANS)(Σ, Π_i) of $[T \longrightarrow T'] > [S_i \longrightarrow S'_i]$. Since it has fewer rules, induction hypothesis applies. Thus $T < S_i$ and $T' > S'_i$ are proved. ∎

This special case says that the coercible order of λ_m is a conservative extension of that of λ_c.

$$
\begin{aligned}
+\ 1.5\ 2 &= (+_{\mathtt{int}} \vee +_{\mathtt{real}})\ 1.5\ 2 \\
&\triangleright\ +_{\mathtt{real}}\ 1.5\ 2 \\
&\triangleright\ +'_{\mathtt{real}}\ 1.5\ 2.0 \\
&\triangleright\ 3.5.
\end{aligned}
$$

The expression $+\ 1\ 2.5$ reduce to

$$
\begin{aligned}
+\ 1\ 2.5 &= +_{\mathtt{int}} \vee +_{\mathtt{real}}\ 1\ 2.5 \\
&\triangleright\ (+_{\mathtt{int}}\ 1 \vee +_{\mathtt{real}}\ 1)\ 2.5 \\
&\triangleright\ +_{\mathtt{real}}\ 1\ 2.5 \\
&\triangleright\ +'_{\mathtt{real}}\ 1.0\ 2.5 \\
&\triangleright\ 3.5.
\end{aligned}
$$

Consider the expression $+\ 1\ 2$. It reduces to

$$
\begin{aligned}
+\ 1\ 2 &= +_{\mathtt{int}} \vee +_{\mathtt{real}}\ 1\ 2 \\
&\triangleright\ (+_{\mathtt{int}}\ 1 \vee +_{\mathtt{real}}\ 1)\ 2 \\
&\triangleright\ (+_{\mathtt{int}}\ 1\ 2) \vee (+_{\mathtt{real}}\ 1\ 2) \\
&\triangleright\ (+_{\mathtt{int}}\ 1\ 2) \vee (+'_{\mathtt{real}}\ 1.0\ 2.0) \\
&\triangleright\ 3 \vee 3.0 \\
&\triangleright\ 3.
\end{aligned}
$$

It is odd to calculate both integer addition and real addition and take their merge. However, if we can assume that $+_{\mathtt{int}}$ and $+_{\mathtt{real}}$ coincide on integer values, then, we can optimize it and omit the real addition, as we have noted in Subsection 1.2.3.

3.2.4 Union types

The merge of functions become more expressive if we have the union types ([Pie92]). For example, consider the type $int \cup bool$, whose elements are integer values or boolean values. Then a function $\mathtt{e} : [\mathtt{int} \longrightarrow \mathtt{int}, \mathtt{bool} \longrightarrow \mathtt{int}]$ applied to a value $\mathtt{f} : \mathtt{int} \cup \mathtt{bool}$ will produce a value of type int. A union type is very useful in real programming. For example, we sometimes want to define the type of a list whose elements are integers or boolean values. A merge of functions can be used as an elimination rule on union types. The details are left as a further work.

3.3 Some Syntactic Properties of λ_m

We will give some syntactic properties of λ_m. In defining λ_m, we have proved that λ_m has Unicity of Type and Subject Reduction properties. Unicity of Type says that the type of an expression is uniquely determined. Subject Reduction says that the type of an expression does not change in reduction. We will prove here the Church-Rosse Property of λ_m. It says

3.2.3 Multi-Methods

When a generic function is applied, it dispatches an appropriate function depending on the type of the arguments. This method search must be based not only on the first argument but also on all the arguments. We have been only concerned with a function with one argument, and the method search algorithm encoded in the reduction rules only handles the first argument. However, we can express a function with several arguments by encoding it to a function with one argument by Currying, and through this embedding, the reduction rules will work properly even with more than one arguments.

As an example, we consider the addition function + which is applicable to both $int \times int$ and $real \times real$, which will apply the inter addition if both arguments are integer, and applies real addition if one of the arguments are real numbers. In λ_m, we only considered two basic types int and *bool*, and did not consider the type $real$ as a basic type because operations on $real$ cannot be calculated syntactically. However, the mechanism to select the appropriate function can be applied even when the type structure is extended with $real$.

We add $real$ to the set of type expressions, the rule

(M-READ) $real \uparrow int$

to the rule of compatibility relations, and the rule

(I-REAL) $\dfrac{}{int > real}$.

to the coercible order. This rule means that integer values are coercible to real numbers. We suppose that this coercion is predefined in the machine, and suppose $e|_{real}$ is reduced to some $real$ number. As to merge, we abbreviate $int \vee real = int$. It means that int is more informative in that it has the information that it belongs to the smaller set int.

We need to define the merge of an integer expression and a real expression, which belongs to $int \vee real = int$.

(E-REAL1) $n \vee r \triangleright n$ (if $n : int, r : real, n|_{real} = r$)

(E-REAL2) $n \vee r \triangleright E_{int}$ (if $n : int, r : real, n|_{real} \neq r$)

In Section 3.2.1, we have defined + on integer values by using primitive recursion. Here, we presume that the function $+_{int}$ and $+'_{real}$ of types $int \longrightarrow int \longrightarrow int$ and $real \longrightarrow real \longrightarrow real$ are predefined, and define $+ = +_{int} \vee +_{real}$. Here, $+_{real}$ is a generic function which coerces its arguments to real and then applies $+'_{real}$. Thus, + has the type $[int \longrightarrow int \longrightarrow int,\ real \longrightarrow real \longrightarrow real]$.

Consider the expression + `1.5 2`. It reduces to

Then `move 100 200 w` is reduced as follows:

```
move 100 200 w = (move1 ∨ move2 ∨ move3) 100 200 w
          ▷   (move1 100 ∨ move2 100 ∨ move3 100) 200 w
          ▷   (move1 100 200 ∨ move2 100 200 ∨ move3 100 200) w
          ▷   (move1 100 200 w) ∨ (move2 100 200 w)
          ▷*   {x = w|[Window].x + 100,
                y = w|[Window].y + 200,
                w = w|[Window].w,
                h = w|[Window].h} ∨
               {frame = (move1 100 200 w|[Frame-Window].frame)}
          ▷*   {x = 150, y = 350, w = 40, h = 80,
                frame = { x = 148, y = 348, w = 44, h = 84}}
```

Almost in the same way, when

```
w = {x = 50, y = 150, w = 40, h = 80,
     frame = {x = 48, y = 148, w = 44, h = 84},
     title = {x = 48, y = 143, w = 10, h = 5, content:``foo''}}
```

of type `[Frame-Title-Window]`, `move 100 200 w` will be reduced to

```
{x = 150, y = 350, w = 40, h = 80,
 frame = {x = 148, y = 348, w = 44, h = 84},
 title = {x = 148, y = 343, w = 10, h = 5, content:``foo''}}.
```

3.2.2 Overloading

We will give an example of an overloaded function. The function which increments an integer value, and negate a boolean value is written as follows:

$$(\lambda \mathtt{x}^{\mathtt{int}}.\mathbf{suc}\ \mathtt{x}) \vee (\lambda \mathtt{x}^{\mathtt{bool}}.\mathbf{if}_{\mathtt{bool}}\ \mathbf{false}\ \mathbf{true}\ \mathtt{x}).$$

It has type $[\mathtt{int} \longrightarrow \mathtt{int}, \mathtt{bool} \longrightarrow \mathtt{bool}]$. When it is applied to $\mathbf{0}$, then

$$\begin{aligned}(\lambda \mathtt{x}^{\mathtt{int}}.\mathbf{suc}\ \mathtt{x} \vee \lambda \mathtt{x}^{\mathtt{bool}}.\mathbf{if}_{\mathtt{bool}}\ \mathbf{false}\ \mathbf{true}\ \mathtt{x})\ \mathbf{0} \\ \triangleright\ (\lambda \mathtt{x}^{\mathtt{int}}.\mathbf{suc}\ \mathtt{x})\ \mathbf{0} \\ \triangleright\ \mathbf{suc}\ \mathbf{0}|_{\mathtt{int}} \\ \triangleright\ \mathbf{suc}\ \mathbf{0},\end{aligned}$$

and when it is applied to **true**, then the term is reduced to **false**.

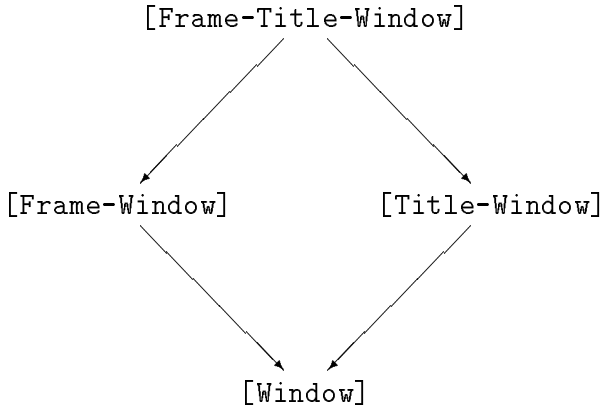
```
                h:integer       ; height
                },
```

and define window types as follows:

```
[Window] = [rectangle],
[Frame-Window] = [Window] ∨ {frame:[rectangle]},
[Title-Window] =
     [Window] ∨ {title:([rectangle] ∨ {content:string})},
[Frame-Title-Window] =
     [Frame-Window] ∨ [Title-Window].
```

Note that a name of a type only serves as an abbreviation of a record type and $\vee$ on types is also an abbreviation of a type. `[Frame-Title-Window]` is defined because `[Frame-Window]` and `[Title-Window]` are compatible.

These record types have the following inheritance relation.



The move function on `[rectangle]` (i.e. on `[window]`) is defined as follows:

$$\texttt{move1} = \lambda \texttt{dx}^{\texttt{int}}.\lambda \texttt{dy}^{\texttt{int}}.\lambda \texttt{r}^{[\texttt{rectangle}]}.\{\texttt{x} = \texttt{r.x} + \texttt{dx}, \texttt{y} = \texttt{r.y} + \texttt{dy}, \texttt{w} = \texttt{r.w}, \texttt{h} = \texttt{r.h}\}.$$

Move function on `[Frame-Window]` and `[Title-Window]` are defined as follows:

$$\texttt{move2} = \lambda \texttt{dx}^{\texttt{int}}.\lambda \texttt{dy}^{\texttt{int}}.\lambda \texttt{r}^{[\texttt{Frame-Window}]}.\{\texttt{frame} = (\texttt{move1 dx dy r.frame})\},$$

$$\texttt{move3} = \lambda \texttt{dx}^{\texttt{int}}.\lambda \texttt{dy}^{\texttt{int}}.\lambda \texttt{r}^{[\texttt{Title-Window}]}.\{\texttt{title} = (\texttt{move1 dx dy r.title}) \vee \{\texttt{content} = \texttt{r.title.content}\})\}.$$

Define `move` as `move1` $\vee$ `move2` $\vee$ `move3`. Let

```
w = {x = 50, y = 150, w = 40, h = 80,
     frame = {x = 48, y = 148, w = 44, h = 84}}:  [Frame-Window].
```

One of the drawbacks of λ_m is that, we need to use the type information in the reduction rule. This feature appears in the rules (E-APPL) and (E-APPC1). In (E-APPL), we used type information in order to find applicable functions. Actually, this is what a practical object oriented language is doing. The process (E-APPL) corresponds, in CLOS ([BDG+88]), to the process of finding the applicable methods from the methods of a generic function when a generic function is called. And the result $e_{\phi(1)}\ f \vee \ldots \vee e_{\phi(l)}\ f$ corresponds to the application of the effective method. In this sense, this calculus naturally reflects the execution mechanism of object oriented languages.

3.2 Examples

We will give some examples of λ_m terms.

3.2.1 Move function

First, we will write the move function on windows given in Section 1.2.2.

Before that, we extend λ_m with primitive recursions. We add to λ_m the infinite set of constants $\mathbf{suc} : \mathit{int} \longrightarrow \mathit{int}$, $\mathbf{rec}_S : S \longrightarrow (S \longrightarrow \mathit{int} \longrightarrow S) \longrightarrow \mathit{int} \longrightarrow S$, and $\mathbf{if}_S : S \longrightarrow S \longrightarrow \mathit{bool} \longrightarrow S$, where S varies over all the types. In this section, we omit [] from a function type when it has only one component. The intended meaning of these constants are: **suc** is the successor function, $\mathbf{rec}_S$ is the primitive recursor on the type S, and $\mathbf{if}_S$ is the **if_then_else** expression on type S. Note that we also assume the existence of constants **0** of type *int* and **true**, **false** of type *bool*.

The reduction rules of these terms are defined as follows:

$$\begin{array}{ll}
\text{(E-REC1)} & \mathbf{rec}_S\ e\ f\ 0 \ \triangleright\ e|_S \\
\text{(E-REC2)} & \mathbf{rec}_S\ e\ f\ (\mathbf{suc} n) \ \triangleright\ f|_{S \longrightarrow \mathit{int} \longrightarrow S}\ (\mathbf{rec}_S\ e\ f\ n)\ n \\
\text{(E-IF1)} & \mathbf{if}_S\ e\ f\ \mathbf{true} \ \triangleright\ e|_S \\
\text{(E-IF2)} & \mathbf{if}_S\ e\ f\ \mathbf{false} \ \triangleright\ f|_S
\end{array}$$

Now, we define "+" as a function of type $\mathit{int} \longrightarrow \mathit{int} \longrightarrow \mathit{int}$ using $\mathbf{rec}_{\mathit{int}}$ and **suc** as follows:

$$\texttt{+} = \lambda x^{\mathit{int}}.\lambda y^{\mathit{int}}.\mathbf{rec}_{\mathit{int}}\ x\ (\lambda z^{\mathit{int}}.\lambda u^{\mathit{int}}.\mathbf{suc}\ z)\ y\,.$$

For readability, we will write + in infix notation, and use `this font` for λ_m terms in the following.

Let `[rectangle]` be the following record type:

```
[rectangle] = {x:integer,      ; x position
               y:integer,      ; y position
               w:integer,      ; width
```

(E-LABEL) $\{l_1 = e_1, \ldots, l_n = e_n\}.l_i \triangleright e_i$

(E-APP) $(\lambda x^S.\ e)\ f \triangleright e[x^S := f|_S]$

(E-APPL) $(e_1 \vee \ldots \vee e_n)\ f \triangleright e_{\phi(1)}\ f \vee \ldots \vee e_{\phi(l)}\ f$
(Where $e_{\phi(i)}$ are selected so that $AT(F_{\phi(i)}, S)$ exists for $F_{\phi(i)}$, the type of $e_{\phi(i)}$ and S, the type of f)

(E-APPC1) $(e|_F)\ f \triangleright (e\ f)|_{AP(F,S)}$ (Here S is the type of f)

(E-COE1) $\{l_1 = e_1, \ldots, l_n = e_n\}|_{\{l_1:S_1,\ldots,l_m:S_m\}} \triangleright \{l_1 = e_1|_{S_1}, \ldots, l_m = e_m|_{S_m}\}$

(E-COE2) $e|_N \triangleright e$ $(N = int, bool)$

(E-LUBR) $\{l_1{=}e_1, \ldots, l_m{=}e_m, \hat{l_1}{=}\hat{e_1}, \ldots \hat{l_k}{=}\hat{e_k}\} \vee \{l_1{=}f_1, \ldots, l_m{=}f_m, \check{l_1}{=}\check{f_1}, \ldots, \check{l_n}{=}\check{f_n}\} \triangleright$
$\{l_1 = e_1 \vee f_1, \ldots, l_m = e_m \vee f_m, \hat{l_1} = \hat{e_1}, \ldots \hat{l_k} = \hat{e_k}, \check{l_1} = \check{f_1}, \ldots, \check{l_n} = \check{f_n}\}$

(E-CONST1) $n \vee n \triangleright n$ (n is a constant in int or $bool$)

(E-CONST2) $n_1 \vee n_2 \triangleright E_{int}$ $(n_1 \neq n_2, n_1, n_2$ are constants in int.)

(E-CONST3) $n_1 \vee n_2 \triangleright E_{bool}$ $(n_1 \neq n_2, n_1, n_2$ are constants in $bool$.)

(E-LABEL), (E-APP), (E-COE1), (E-COE2) are the same as in λ_c. (E-APPC) in λ_c is replaced by the rule (E-APPC1), so that it is applicable even when F is not a simple function type. It is an extension of (E-APPC) because $AP([S \longrightarrow S'], T)$ is equal to S' when $T > S$. Thus, λ_m includes λ_c as a subsystem.

(E-APPL) is the rule for an application of a merged function. The application of a merged function to a value is the merge of the result of applicable functions. The typing rule (T-APP1) ensures that at least one of the components are applicable. So the right hand side of this rule exists for typed expressions.

(E-APPC1) is the rule for applying a function defined by coercing another function. It says that first the original function is applied, and then the result is coerced to the appropriate type. This rule is related to the meaning of a coercion from a function type to another function type. As we have noted, a function type F intuitively means the correspondence of the argument type and the return type, and the coercion of a function $f : G$ to F is the function applicable to $e : S$ when $AT(F, S)$ exists, and when $f : S$ is applied, apply e to f and coerce the result pointwisely to the type expected by F.

(E-LUBR), (E-CONST1), (E-CONST2), (E-CONST3) are rules for the merge operator. (E-LUBR) says that the merge of two records e_1 and e_2 has the slots of both records, and if a slot exists both in e_1 and e_2, then the slot value is the merge of corresponding slot values of e_1 and e_2, and if the slot exists only in e_1 or e_2, then the value is used as the slot value. (E-CONST1) says that the merge of the same constant is itself. (E-CONST2) and (E-CONT3) say that the merge of different constants causes a merge error.

Theorem 28 (Subject Reduction) *The above reduction rule is well defined. That is, when e reduces to e′ and e has type S, then e′ also has type S.*

Proof: Proved almost in the same way as λ_c. ∎

Definition 2 *Let F be $[S_1 \longrightarrow S'_1, \ldots, S_n \longrightarrow S'_n]$. Define $AT(F,T)$ to be $S_{\phi(1)} \vee \ldots \vee S_{\phi(l)}$, where $\phi(i)$ satisfies $S_{\phi(i)} < T$. If no S_j satisfies $S_j < T$, it is undefined.*

From Lemma 26, $S_{\phi(1)} \vee \ldots \vee S_{\phi(l)}$ exists.

Definition 3 *Let F be $[S_1 \longrightarrow S'_1, \ldots, S_n \longrightarrow S'_n]$. Define $AP(F,T)$ to be $S'_{\phi(1)} \vee \ldots \vee S'_{\phi(l)}$, where $\phi(i)$ satisfies $S_{\phi(i)} < T$. If no S_j satisfies $S_j < T$, it is undefined.*

The detailed proof of the existence of $S'_{\phi(1)} \vee \ldots \vee S'_{\phi(l)}$ is postponed to Lemma 31.

$AT(F,T)$ is a macro on type expressions which means the result of application of a function of type F to type T.

(T-VAR), (T-FUN), (T-LABEL1), (T-LABEL2) are the same as in λ_c. (T-APP1) is the modification of (T-APP) in λ_c to fit the new function types. Since $AT([S \longrightarrow S']T)$ only exists when $T > S$ with the value S, and $AP([S \longrightarrow S']T)$ is S' when $T > S$, it is an extension of (T-APP). (T-LUB) is a new rule which says that a merged expression belongs to the merge of the component types. This expression is well formed only when the type of each component are compatible. Note that $S \vee S'$ is an abbreviation of a type expression. Therefore, many types have expressions of the form $e \vee e'$. For example, $1 \vee 1$ is an expression of type $int \vee int = int$, and $1 \vee 2$ is also an expression of type int. $\{l_1 = 1, l_2 = true\} \vee \{l_2 = true, l_3 = \{l_1 = true\}\}$ is an expression of type $\{l_1 : int, l_2 : bool, l_3 : \{l_1 : bool\}\}$, and $\lambda f^{[int \longrightarrow int, bool \longrightarrow bool]}.(\lambda x^{int}.f\ x \vee \lambda x^{\{l:bool\}}.f\ x.l)$ is an expression of type $[[int \longrightarrow int, bool \longrightarrow bool] \longrightarrow [int \longrightarrow int, \{l : bool\} \longrightarrow bool]]$.

λ_m does not include the subsumption rule. Therefore, only one rule is applicable to each expression. Therefore, the unicity of type holds.

Theorem 27 (Unicity of Type) *For each expression e, there exists at most one type expression S which satisfies $e : S$.*

3.1.4 Reduction Rules

The reduction rules of λ_m are as follows:

of S_1 and S_2 by Lemma 24. Thus $S_1 \uparrow S_2$ is the necessary and sufficient condition for S_1 and S_2 to have an upper bound. This lemma tells us that $S_1 \vee S_2$ is a least upper bound of the pre-order $<$, and the set of type expressions forms a consistently complete pre-order; that is, the set of types with an upper bound has a least upper bound.

3.1.3 Expressions and Type Assignment

To introduce expressions of each types, we first introduce pseudo expressions, and then give type inference rules. The pseudo expressions of λ_m are as follows:

$$\begin{array}{rcl} e & ::= & x^S \mid c^{\top}_{int} \mid c^{\top}_{bool} \\ & & \mid \lambda x^S.\ e \mid e\ f \\ & & \mid \{l_1 = e_1, \ldots, l_n = e_n\} \mid e.l\ . \\ & & \mid e|_S \\ & & \mid e_1 \vee e_2 \end{array}$$

Here, $c^{\top}_{int}$ and $c^{\top}_{bool}$ are constants of type int and $bool$, which are extended with the top elements E_{int} and E_{bool}, respectively. That is, $c^{\top}_{bool}$ runs over the set $\{true, false, E_{bool}\}$. Top elements are introduced in order to express merge errors.

We omit the type of a variable if the variable is bound in a lambda abstraction surrounding it and there is no ambiguity.

What is extended from λ_c are expressions of the form $e_1 \vee e_2$, which means the merge of the expressions e_1 and e_2.

The type assignment system for λ_m is as follows:

(T-VAR) $$\frac{}{x^S : S}$$

(T-FUN) $$\frac{e : S'}{\lambda x^S.\ e : [S \longrightarrow S']}$$

(T-APP1) $$\frac{e : F = [S_1 \longrightarrow S'_1, \ldots, S_n \longrightarrow S'_n] \quad f : S \quad AT(F,S) \text{ exists}}{e\ f : AP(F,S)}$$

(T-LABEL1) $$\frac{e_i : S_i \quad (i = 1, \ldots, n)}{\{l_1 = e_1, \ldots, l_n = e_n\} : \{l_1 : S_1, \ldots, l_n : S_n\}}$$

(T-LABEL2) $$\frac{e : \{l_1 : S_1, \ldots, l_n : S_n\}}{e.l_i : S_i}$$

(T-COE) $$\frac{e : S \quad S > S'}{e|_{S'} : S'}$$

(T-LUB) $$\frac{e : S \quad e' : S' \quad S \uparrow S'}{e \vee e' : S \vee S'}$$

Here, $AT(F,T)$ and $AP(F,T)$ are macros on type expressions defined as follows:

between types, these rules are essentially equivalent to the subtyping rule of intersection type. Therefore, $S_1 \vee S_2$ has similar properties to intersection types with respect to the ordering of types.

Lemma 24 *1)*$S \vee T \simeq T \vee S$.
2)$S \vee T > S, \quad S \vee T > T$.

Proof: Trivial by induction on the structure of types. ∎

Thus, $S \vee T$ is an upper bound of S and T with this order. Actually, we will prove that it is a least upper bound in Lemma 26. In order to prove this, we need a technical lemma.

Lemma 25 *$T > S$ and $T \uparrow T'$ implies $T' \uparrow S$. Especially, $T > S$ implies $T \uparrow S$.*

Proof: Induction on the proof of $T > S$. If the last rule is (I-REFL),(I-LABEL),(I-FLIST1),(I-FLIST2) or (I-FLIST3), then easily proved. For (I-FUN), we assume that $[T \rightarrow T'] > [S \rightarrow S']$ is proved using (I-FUN) as the last rule and assume that $[U_1 \rightarrow U_1', \ldots, U_n \rightarrow U_n'] \uparrow [T \rightarrow T']$, and prove that $[U_1 \rightarrow U_1', \ldots, U_n \rightarrow U_n'] \uparrow [S \rightarrow S']$. That is, we assume $T < S, T' > S', U_i \rightarrow U_i' \uparrow T \rightarrow T'$ $(i = 1, \ldots, n)$, and prove $U_i \rightarrow U_i' \uparrow S \rightarrow S'$ $(i\ 1, \ldots, n)$. Suppose $U_i \uparrow S$. Then by induction hypothesis, $U_i \uparrow T$. Therefore, $U_i' \uparrow T'$ because $U_i \rightarrow U_i' \uparrow T \rightarrow T'$. Again by induction hypothesis, $U_i' \uparrow S'$. Thus $U_i \rightarrow U_i' \uparrow S \rightarrow S'$ is proved. For (I-TRANS), we assume that $S > S''$ is proved from $S > S'$ and $S' > S''$, and assume that $T \uparrow S$. Then by induction hypothesis, $T \uparrow S'$. And again by induction hypothesis, $T \uparrow S''$. ∎

Lemma 26 *$T_1 > S_1, T_2 > S_2$, and $T_1 \uparrow T_2$ implies $S_1 \uparrow S_2$ and $T_1 \vee T_2 > S_1 \vee S_2$. Especially, $T > S_1$ and $T > S_2$ implies $S_1 \uparrow S_2$ and $T > S_1 \vee S_2$.*

Proof: From Lemma 25, $T_2 \uparrow S_1$, and again from Lemma 25, $S_1 \uparrow S_2$. We prove by induction on the structure of $T = T_1 \vee T_2$ that $T > S_1$ and $T > S_2$ implies $T > S_1 \vee S_2$. Suppose that S_1, S_2, T are all record types. It is easy to see that (I-LABEL) is a necessary and sufficient condition for two record types to be ordered. Thus, we only need to compare the types of each slots of T and $S_1 \vee S_2$, which are ordered by $T > S_1, T > S_2$, and the induction hypothesis.

Suppose that S_1, S_2, T are all function types. Let P be a function component of S_1. then From (I-FLIST1), $S_1 > [P]$ and from (I-TRANS) and Lemma 24, $T > [P]$. Thus, all the components of $S_1 \vee S_2$ are smaller than T, and therefore, from (I-FLIST3), $T > S_1 \vee S_2$.

When S_1, S_2, T are all basic types, the statement degenerates. ∎

This lemma tells us that if S_1 and S_2 has an upper bound, then S_1 and S_2 are compatible. On the other hand, if S_1 and S_2 are compatible, then $S_1 \vee S_2$ exists which is an upper bound

From 5), we omit parenthesis for $\vee$. From 6), $S_1 \vee \ldots \vee S_n$ exists when $S_1, \ldots, S_n$ are pairwise compatible.

Since we distinguish $[P, P']$ from $[P', P]$, $S \vee T$ and $T \vee S$ are different type expressions. Thus $\vee$ is not a symmetric operator.

3.1.2 The Coercible Order

The coercible order $>$ on type expressions is defined as follows:

(I-REFL) $$\frac{}{S > S}$$

(I-TRANS) $$\frac{S > S' \quad S' > S''}{S > S''}$$

(I-LABEL) $$\frac{S_i > S_i' \quad (i = 1, \ldots, m) \quad n > m}{\{l_1 : S_1, \ldots, l_n : S_n\} > \{l : S_1', \ldots, l_m : S_m'\}}$$

(I-FUN) $$\frac{T < S \quad T' > S'}{[T \longrightarrow T'] > [S \longrightarrow S']}$$

(I-FLIST1) $$\frac{}{[P_1, \ldots, P_n] > [P_i]}$$

(I-FLIST2) $$\frac{}{[S_1 \longrightarrow S_1', S_2 \longrightarrow S_2'] > [S_1 \vee S_2 \longrightarrow S_1' \vee S_2']}$$

(I-FLIST3) $$\frac{[P_1, \ldots, P_n] > [P_i'] \quad (i = 1, \ldots, n)}{[P_1, \ldots, P_n] > [P_1', \ldots, P_m']}$$

$>$ is a preorder on type expressions. When $S > T$, we call S is coercible to T. When $S > T$ and $T > S$, we say that S and T are equivalent, and write $S \simeq T$.

These rules are available only when each type expression is well formed. For example, (I-FLIST2) only says that If $S_1 \longrightarrow S_1'$ and $S_2 \longrightarrow S_2'$ are compatible, S_1 and S_2 are compatible, and S_1' and S_2' are compatible, then $[S_1 \longrightarrow S_1', S_2 \longrightarrow S_2'] > [S_1 \vee S_2 \longrightarrow S_1' \vee S_2']$.

The rules added form λ_c are (I-FLIST1), (I-FLIST2), and (I-FLIST3). As in λ_c, the meaning of $T > S$ is that there is an implicit coercion function from T to S. Recall that a function type $F = [S_1 \longrightarrow S_1', \ldots, S_n \longrightarrow S_n']$ means the correspondence of the argument type and the result type. Thus, $G > F$ intuitively means that (1) the domain of G is bigger than F, and (2) for each type S, the result types are ordered. The coercion function from G to F coerces the result of G pointwisely to the type expected by F.

This informal explanation of the meaning of a coercion function is given more formally in Chapter 5, and is realized in the reduction rules defined in Subsection 3.1.4.

$\vee$and intersection types ([CDCV81], [CC90],[BCDC83], [Rey91]) are different notions; an intersection type is a notion of the type inference problem on the untyped lambda calculus. However, if we view $S_1 \vee S_2$ as an intersection type $S_1 \wedge S_2$, and reverse the order relation

$$\text{(M-FUN)} \quad \frac{S_1 \uparrow S_2 \text{ implies } S_1' \uparrow S_2'}{S_1 \longrightarrow S_1' \uparrow S_2 \longrightarrow S_2'}$$

Lemma 22 *1.* $S \uparrow S$.

2. $S \uparrow T$ *implies* $T \uparrow S$.

3. $S_1 \uparrow S_2$ *is decidable.*

Note that $\uparrow$ is symmetric and reflective, but not a transitive relation.

When $S_1 \uparrow S_2$, we define $S_1 \vee S_2$, the merge of S_1 and S_2 as an abbreviation for a type expression as follows:

$$int \vee int = int$$

$$bool \vee bool = bool$$

$$\{l_1 : S_1, \ldots, l_m : S_m, \hat{l_1} : \hat{S_1}, \ldots, \hat{l_n} : \hat{S_n}\} \vee \{l_1 : T_1, \ldots, l_m : T_m, \check{l_1} : \check{T_1}, \ldots \check{l_k} : \check{T_k}\} =$$

$$\{l_1 : S_1 \vee T_1, \ldots, l_m : S_m \vee T_m, \hat{l_1} : \hat{S_1}, \ldots, \hat{l_n} : \hat{S_n}, \check{l_1} : \check{T_1}, \ldots \check{l_k} : \check{T_k}\}$$

$$[S_1 \longrightarrow S_1', \ldots, S_m \longrightarrow S_m'] \vee [S_{m+1} \longrightarrow S_{m+1}', \ldots, S_n \longrightarrow S_n'] = [S_1 \longrightarrow S_1', \ldots, S_n \longrightarrow S_n'].$$

This definition says that if record types R_1 and R_2 share slots $l_1, \ldots, l_m$, R_1 has other slots $\hat{l_1}, \ldots, \hat{l_n}$, and R_2 has other slots $\check{l_1}, \ldots, \check{l_k}$, then $R_1 \vee R_2$ has slots of both records with the slot types of $l_1, \ldots, l_m$ the merge of corresponding slot types of R_1 and R_2, with the slot types of $\hat{l_1}, \ldots, \hat{l_n}$ those of R_1, and with the slot types of $\check{l_1}, \ldots, \check{l_k}$ those of R_2.

From this definition, $\{l_1 : S_1, \ldots, l_n : S_n\} = \{l_1 : S_1\} \vee \ldots \vee \{l_n : S_n\}$ and $[S_1 \longrightarrow S_1', \ldots, S_n \longrightarrow S_n'] = [S_1 \longrightarrow S_1'] \vee \ldots \vee [S_n \longrightarrow S_n']$. Therefore, a record type can be constructed as a merge of records with only one slot, and a function type can be constructed as a merge of simple function types.

Lemma 23 *1) When* $S_1 \uparrow S_2$, $S_1 \vee S_2$ *is well defined.*
2) $S \vee S = S$.
3) $(S \vee T) \uparrow T$.
4) $(S \vee T) \uparrow S$.
5) When $(S_1 \vee S_2)$ *and* S_3 *are compatible,* $S_1 \uparrow S_3$, $S_2 \uparrow S_3$, $S_1 \uparrow (S_2 \vee S_3)$, *and* $(S_1 \vee S_2) \vee S_3 = S_1 \vee (S_2 \vee S_3)$ *holds.*
6) When $S_1, \ldots, S_n$ *are pairwise compatible,* $S_1 \vee \ldots \vee S_{n-1}$ *and* S_n *are compatible.*

in function applications. Function types of λ_m are natural extensions of function types in λ_c. The intuitive meaning of $[S_1 \longrightarrow S'_1, \ldots, S_n \longrightarrow S'_n]$ is the set of functions of the following properties: (1) it is applicable to those types which are bigger than one of $S_1, \ldots, S_n$ by the order we will define in the next section, (2) when the argument type is bigger than $S_{\phi(1)}, \ldots, S_{\phi(m)}$, then it returns a value in $S'_{\phi(1)} \vee \ldots \vee S'_{\phi(m)}$, here, $\vee$ is an abbreviation on types, which is defined below.

For record types, we do not care the order of record slots; therefore, we identify those types which have the same set of slot name-and-type pairs. On the other hand, we distinguish two function types only different in the order of components for two technical reasons. One is that, even if we identify those types, there are still, as we will see, many equivalent function types and we need to define later the equivalence of types. The other is that it may be possible to consider a modified version of λ_m, in which the order of components has meaning. Actually, as we will see, the subject reduction theorem holds, and therefore we do not need to identify them for the subject reduction theorem.

The type expression $[P_1, \ldots, P_n]$ is allowed only when $P_1, \ldots, P_n$ are pairwise compatible. We write $S \uparrow S'$ when S and S' are compatible. The definition of $S_1 \uparrow S_2$ is as follows:

(M-REFL) $\dfrac{}{N \uparrow N}$ (N = int, bool)

(M-LABEL) $\dfrac{S_i \uparrow T_i (i = 1, \ldots, m)}{\{l_1 : S_1, \ldots, l_m : S_m, \hat{l}_1 : \hat{S}_1, \ldots, \hat{l}_n : \hat{S}_n\} \uparrow \{l_1 : T_1, \ldots, l_m : T_m, \check{l}_1 : \check{T}_1, \ldots \check{l}_k : \check{T}_k\}}$

$(l_i\ (1 \leq i \leq m), \hat{l}_j\ (1 \leq j \leq n), \check{l}_h\ (1 \leq h \leq k)$ are distinct)

(M-FUN1) $\dfrac{\text{not } S_1 \uparrow S_2}{S_1 \longrightarrow S'_1 \uparrow S_2 \longrightarrow S'_2}$

(M-FUN2) $\dfrac{S_1 \uparrow S_2 \text{ and } S'_1 \uparrow S'_2}{S_1 \longrightarrow S'_1 \uparrow S_2 \longrightarrow S'_2}$

(M-FLIST) $\dfrac{P_i \uparrow P'_j (i = 1, \ldots, n,\ j = 1, \ldots, m)}{[P_1, \ldots, P_n] \uparrow [P'_1, \ldots, P'_m]}$

The intuitive meaning of $S \uparrow T$ is that S and T can be merged or there is a type expression which is bigger than both S and T. *int* is only compatible with *int*, and *bool* is only compatible with *bool*. A record type is compatible only with a record type, and a function type is compatible only with function type. (M-LABEL) means that record types R and R' are compatible only when every shared slot of R and R' has compatible types. Function components $S_1 \longrightarrow S'_1$ and $S_1 \longrightarrow S'_1$ are compatible when there is no argument which is applicable to both of them (M-FUN1) or there is an argument which is applicable to both of them and the result types are mergable (M-FUN2). We can write these two rules into the following one rule:

Chapter 3

λ_m: A Calculus with a Merge Operator

In this chapter, we study λ_m, which is the main calculus of this thesis. λ_m is an extension of λ_c with a merge operator which is applicable to both records and functions.

3.1 Definition of λ_m

3.1.1 Type Expressions

The set of type expressions of λ_m is that of λ_c, extended with the type expression $[S_1 \rightarrow S'_1, \ldots, S_n \rightarrow S'_n]$.

$$
\begin{array}{lcl}
S & ::= & \mathit{int} \mid \mathit{bool} \mid \{l_1 : S_1, \ldots, l_n : S_n\} \\
 & & \mid [P_1, \ldots, P_n] \quad (P_1, \ldots, P_n \text{ are pairwise compatible (defined below).}) \\
P & ::= & S \rightarrow S'
\end{array}
$$

As in λ_c, $l_1, \ldots, l_n$ run over an infinite set of slot names. We call int or bool a basic type expression, $\{l_1 : S_1, \ldots, l_n : S_n\}$ a record type expression, and $[P, \ldots, P_n]$ a function type expression, or, in short, basic type, record type, and function type, respectively. We call a function type expression of the form $[S_1 \rightarrow S'_1]$ a simple function type expression. We call each component of a function type expression a function component.

S and T are metavariables for type expressions, N for basic types, R for record types, and F and G for function types. P is a metavariable for function components.

A type expression $S \rightarrow S'$ of the simply typed lambda calculus and λ_c is now expressed as $[S \rightarrow S']$. $[S_1 \rightarrow S'_1, \ldots, S_n \rightarrow S'_n]$ is intuitively the type constructed by merging the types $[S_i \rightarrow S'_i]$ $(i = 1, \ldots, n)$.

In λ_c, the meaning of a function type $S \rightarrow S'$ was the set of functions which are applicable to types bigger than S and return values of type S' by implicitly inserting coercion functions

We write λ_c^* for the calculus obtained from λ_c by replacing (E-APPC) by (E-APPC*).

We need to define what does it mean that two reduction systems are equivalent. We have proved strong normalization of λ_c in the previous section. Church-Rosser property also holds for λ_c. (See Section 3.3.2.) Therefore, every term has an unique normal form. It leads to the equivalence that two reduction systems are equivalent when all the terms have the same normal forms in the two systems. However, λ_c includes different normal forms which have obviously equivalent meaning. For example, $\lambda x^{int}.x$ and $(\lambda x^{int}.x)|_{int \longrightarrow int}$ are normal forms whose intended meanings are equal. (We will study more about the equivalence of terms of λ_m in Section 3.3.6.) Therefore, the above equivalence is too strong for our aim.

For our purpose, we will restrict our attention to terms with basic types, and define the equivalence of two reduction systems as follows:

Definition 1 *Two reduction systems are equivalent if they reduce closed terms with basic types to the same normal forms.*

Note that closed normal forms of *int* and *bool* are only constants as we will see in Section 3.3.6 for λ_m. Therefore, all normal forms have different meanings. This fact justifies the above definition. We will prove the equivalence of λ_c^* and λ_c using λ_c'' as a mediating system.

Lemma 20 *If $e \triangleright e'$ in λ_c^*, then $e^\diamond \triangleright^* e'^\diamond$ in λ_c'' in one or two steps.*

Proof: The proof is similar to Lemma 18. We only need to prove the case $e \triangleright e'$ by (E-APPC*). Let $e : T \longrightarrow T', T \longrightarrow T' > S \longrightarrow S'$, and $f : U$, $U > S$. $(e|_{S \longrightarrow S'})\ f \triangleright (e\ f|_S)|_{S'}$ in λ_c^*. $(e|_{S \longrightarrow S'}\ f)^\diamond = (e^\diamond|_{S \longrightarrow S'})\ f^\diamond|_S$. It reduces in λ_c'' to $(e^\diamond\ f^\diamond|_S|_T)|_{S'}$, which is equal to $(e\ f|_S)|^\diamond_{S'}$. ∎

From this Lemma, the strong normalization of λ_c^* is proved similarly to λ_c.

Theorem 21 *λ_c and λ_c^* are equivalent.*

Proof: We have the Church-Rosser property and strong normalization of both λ_c and λ_c''. Therefore, a normal form exist and is unique in both systems. By Lemma 18, we have that if $e \triangleright e'$ in λ_c, then $e^\diamond \triangleright^* e'^\diamond$ in λ_c''. Therefore, the normal form of e in λ_c and the normal form of $e^\diamond$ in λ_c are the same.

On the other hand, λ_c^* is also strongly normalizing and Church-Rosser property. From this and Lemma 20, we have that the normal form of e in λ_c^* and the normal form of $e^\diamond$ in λ_c are the same.

By composing them, e have the same normal form in λ_c and λ_c^*. ∎

Thus, we have proved the equivalence of $\lambda_c$and λ_c^*. It is interesting to note that when we add the merge operator, the corresponding systems $\lambda_m$and $\lambda_m^*$are not equivalent.

Proof: First note that the Church-Rosser property of λ_c' can be proved easily by modifying the proof of for λ_m in Section 3.3.2. Thus, we only need to prove the case that $e \triangleright e'$ by (E-CC) and $e \triangleright e''$ by (R). In this case, it is easy to see that (R) is applicable to e' to produce f. Therefore, by Lemma 10, we can reduce e to f by applying (R) once or twice, and then applying (E-CC) zero or more times. From the proof of Lemma 10, it can be verified that the (R) step $e \triangleright e''$ is same as the first (R) step of the reduction from e to f. ∎

Theorem 16 *λ_c'' has the Church-Rosser property.*

2.2.5 Strong Normalization of λ_c

Since $e^\diamond$ was inductively defined, the following Lemma holds.

Lemma 17 $(e[x^S := f])^\diamond = e^\diamond[x^S := f^\diamond]$.

Lemma 18 *If $e \triangleright e'$ in λ_c, then $e^\diamond \triangleright^* e'^\diamond$ in λ_c'' in one or two steps.*

Proof: It is trivial for (E-LABEL), (E-COE1), and (E-COE2). For (E-APP), $(\lambda x^S.\ e)\ f \triangleright e[x^S := f|_S]$ in λ_c. $((\lambda x^S.\ e)\ f)^\diamond = (\lambda x^S.\ e^\diamond)\ f^\diamond|_S$. It reduces to $e^\diamond[x^S := f^\diamond|_S]$ in λ_c''. On the other hand, from Lemma 17, $(e[x^S := f|_S])^\diamond = e^\diamond[x^S := f^\diamond|_S]$.

For (E-APPC), Let $e : T \to T'$ and $e|_{S \to S'}\ f \triangleright (e\ f)|_{S'}$ in λ_c. $(e|_{S \to S'}\ f)^\diamond = (e^\diamond|_{S \to S'})\ f^\diamond|_S$. It reduces in λ_c'' to $(e^\diamond\ f^\diamond|_S|_T)|_{S'}$ by (E-APPC'), and then to $(e^\diamond\ f^\diamond|_T)|_{S'}$ by (E-CC). On the other hand, $((e\ f)|_{S'})^\diamond = (e^\diamond\ f^\diamond|_T)|_{S'}$. ∎

Theorem 19 (Strong Normalization of λ_c) *λ_c is strong normalizable.*

Proof: If there is an infinite reduction sequence in λ_c, then we can construct an infinite reduction sequence in λ_c'' by the above lemma. It contradicts the strong normalization of λ_c''. ∎

2.3 λ_c^*: Another Calculus with Implicit Coercion

One may feel that the reduction rule (E-APPC) is unnatural because it does not use the information that $e|_{S \to S'}$ is a function from S. In order to justify this rule, we will give a possible alternative for this rule, and prove that they are equivalent.

An alternative (E-APPC*) of (E-APP) is defined as follows:

(E-APPC*) $\quad (e|_{S \to S'})\ f \triangleright (e\ f|_S)|_{S'}$.

g_{n+1} to g_{m+1}, we can form a reduction sequence from g_{n+1} to g_{m+1} with the first rule not (E-CC). By concatenating the sequences, we can form $\Delta^{(n+1)}$.

Since $\Delta^{(n)}$ and $\Delta^{(n+1)}$ shares the first $n-1$ steps, we can form an infinite reduction sequence $\Delta' = h_1 \triangleright h_2 \triangleright \ldots \triangleright h_n \triangleright \ldots$ by taking the n-th elements of $\Delta^{(n)}$ as h_n. It does not use the (E-CC) rule. That is, Δ' is a sequence in λ'_c. ∎

Since λ'_c does not have an infinite reduction sequence by Corollary 9, we have the strong normalization of λ''_c.

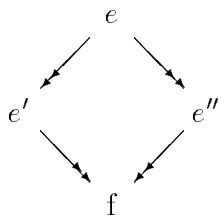
Corollary 14 *λ''_c is strong normalizable.*

2.2.4 Church-Rosser Property of λ''_c

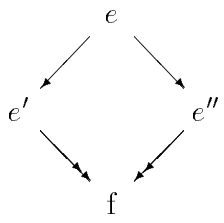
Having proved strong normalization, the Church-Rosser property of λ''_c can be proved easily.

fact *If a calculus is strong normalizable, then Church-Rosser property follows from Weak Church-Rosser property.*

The Church-Rosser property says that if $e \triangleright^* e'$ and $e \triangleright^* e''$, then there is a f to which $e' \triangleright^* f$ and $e'' \triangleright^* f$.



The Weak Church-Rosser property says that if $e \triangleright e'$ and $e \triangleright e''$, then there is a f to which $e' \triangleright^* f$ and $e'' \triangleright^* f$.



Church-Rosser property implies Weak Church-Rosser property. The above fact says that the converse is true if the calculus is strong normalizable. For the proof of this Fact, see [Bar84].

Proposition 15 *λ''_c has the Weak Church-Rosser property.*

Suppose that (R) is (E-APPC'). Then for a term $e_1 : T \longrightarrow T'$, e' has the form $(e_1|_{S \longrightarrow S'})\ f$ and e'' has the form $(e_1\ f|_T)|_{S'}$. If the reduct of (E-CC) is a subterm of e_1 or f, then we can change the order. If it is $e_1|_{S \longrightarrow S'}$, then e has the form $(e_2|_{U \longrightarrow U'})|_{S \longrightarrow S'}\ f$ for a type $U \longrightarrow U'$, which satisfy $T \longrightarrow T' > U \longrightarrow U' > S \longrightarrow S'$. It reduces by (E-APPC') to $(e_2|_{U \longrightarrow U'}\ f|_U)|_{S'}$, which reduces again by (E-APPC') to $(e_2\ f|_U|_T)|_{U'}|_{S'}$. By applying (E-CC) twice, it reduces to e''.

Suppose that (R) is (E-COE1). Then e' has the form $\{l_1 = e_1, \ldots, l_n = e_n\}|_{\{l_1:S_1,\ldots,l_m:S_m\}}$ and e'' has the form $\{l_1 = e_1|_{S_1}, \ldots, l_m = e_m|_{S_m}\}$. If the reduct of (E-CC) is a subterm of e_i, then we can change the order or reduce e directly to e'' by (E-COE1). If it is e', then e has the form $(\{l_1 = e_1, \ldots, l_n = e_n\}|_{\{l_1:U_1,\ldots,l_k:U_k\}})|_{\{l_1:S_1,\ldots,l_m:S_m\}}$. It reduces to $\{l_1 = e_1|_{U_1}, \ldots, l_k = e_k|_{U_k}\}|_{\{l_1:S_1,\ldots,l_m:S_m\}}$ by (E-COE1), and again to $\{l_1 = (e_1|_{U_1})|_{S_1}, \ldots, l_k = (e_m|_{U_m})_{S_m}\}$. It reduces to e'' by m times applying (E-CC).

Suppose that (R) is (E-COE2). Then e' has the form $e_1|_N$ and e'' has the form e_1. If the reduct is is a subterm of e_1 then trivial. If it is $e_1|_N$, then e has the form $(e_1|_N)|_N$, which reduces to e'' by applying (E-COE2) two times. ∎

Lemma 11 *Let $\Delta = f_1 \triangleright \ldots \triangleright f_n \triangleright f_{n+1}$ $(n > 1)$ be a reduction sequence with the first $n-1$ steps reduced by (E-CC), and the last step by (R) that is not (E-CC). Then we can construct a reduction sequence Δ' from f_1 to f_{n+1} with the first rule (R).*

Proof: Induction on n. Suppose that it holds to $n-1$. From the Lemma 10, there is a reduction sequence Δ'' from f_{n-1} to f_{n+1}, which first applies (R) once or twice, and then applies (E-CC) several times. Let the first step of Δ'' be $f_{n-1} \triangleright_{(R)} g$. Because the induction hypothesis applies to the sequence $f_1 \triangleright \ldots \triangleright f_{n-1} \triangleright g$, we can construct a reduction sequence from f_1 to g with the first rule (R). By appending the rest of Δ'', we can form a reduction sequence to f_{n+1}. ∎

Lemma 12 *(E-CC) does not repeat infinitely in a reduction sequence of λ_c'',*

Proof: (E-CC) decreases the number of coercions included in the expression by one. ∎

Lemma 13 *If there exists an infinite reduction sequence in λ_c'', then we can form an infinite reduction sequence in λ_c'.*

Proof: Suppose that $\Delta = f_1 \triangleright \ldots \triangleright f_n \triangleright \ldots$ is an infinite reduction sequence in λ_c''. First, we construct, for each n, an infinite reduction sequence $\Delta^{(n)}$ in λ_c'' which starts with f_1 and merges to Δ at some point, and whose first n steps are not (E-CC). Suppose that $\Delta^{(n)} = g_1 \triangleright \ldots$ is formed. If the $(n+1)$-th step of $\Delta^{(n)}$ is not (E-CC), then take $\Delta^{(n+1)} = \Delta^{(n)}$. Else, from the above lemma, there exists an m such that $g_m \triangleright g_{m+1}$ is the first step other than (E-CC) after g_n. Then applying Lemma 11 to the subsequence from

from Lemma 6 or Lemma 5, respectively. If e has the form $e_1\ e_2$ or $e_1.l$, then $e[x_i := f_i]$ is reducible from the induction hypothesis and the definition of reducibility. If e has the form $e_1|_S$, $e[x_i := f_i]$ is reducible from Lemma 7. ∎

As a special case that f_i are variables of type S_i, all the terms of λ_c' are reducible. And from Lemma 4, (1), all the terms are strong normalizable.

Corollary 9 *λ_c' is strongly normalizable.*

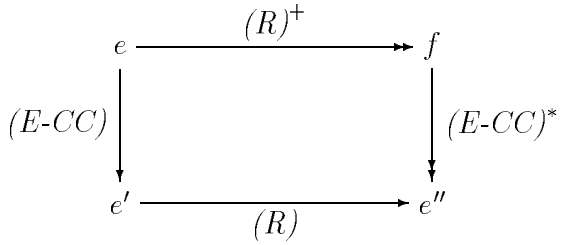
2.2.3 Strong Normalization of λ_c''

We prove strong normalization of λ_c by relating a normalization sequence in λ_c to a normalization sequence in λ_c'', which is an extension of λ_c' with the following reduction rule:

(E-CC) $\dfrac{}{(e|_S)|_T \ \triangleright\ e|_T}$.

This rule corresponds to the transitivity of the coercibleness relation.

Lemma 10 *In λ_c'', if $e \triangleright e'$ by (E-CC) and $e' \triangleright e''$ by a rule (R) which is not (E-CC), then there is a term f to which e reduces one or two steps by the rule (R), and f is equal to e'' or f reduces to e'' by several (E-CC) rules. In a diagram,*



Here, $(R)^+$ denotes one or two (R) rules and (E-CC)* denotes zero or more (E-CC) rules.

Proof: We prove by case analysis over the rule (R). When e_1 is a redex and e_1' is the result of reduction, we call e_1' the reduct. If the reduct of (E-CC) is not included as a subterm to the redex of (R), then we can change the order and apply (R) first and then apply (E-CC). Thus, we can safely assume that the reduct of (E-CC) is a subterm of the redex of (R), and assume that e' is a redex of (R).

Suppose that (R) is (E-LABEL). Then e' has the form $\{l_1 = e_1, \ldots, l_n = e_n\}.l_i$ and e'' has the form e_i. If the reduct of (E-CC) is a subterm of e_i, then we can likewise change the order of (E-CC) and (R). If the result of (E-CC) is a subterm of e_j $(j \neq i)$, then e reduces directly to e'' by (E-LABEL).

Suppose that (R) is (E-APP'). Then e' has the form $(\lambda x^S.\ e_1)\ f_1$ and e'' has the form $e_1[x^S := f_1]$. If the reduct of (E-CC) is a subterm of e_1, then we can change the order. If the reduct of (E-CC) is a subterm of f_1, then We can first apply (E-APP') and then apply (E-CC) to all the copied redexes successively.

Lemma 5 *If $e_i \in Red(S_i)$ for $i = 1,\ldots,n$, then $\{l_1 = e_1,\ldots,l_n = e_n\} \in Red(\{l_1 : S_1,\ldots,l_n : S_n\})$.*

Lemma 6 *Let $e : S'$ be a term which may contain a free variable x^S of type S. If e satisfies $e[x^S := f] \in Red(S')$ for all $f \in Red(S)$, then $\lambda x^S.\ e \in Red(S \longrightarrow S')$.*

Lemma 7 *If $e \in Red(T)$ and $T > S$, then $e|_S \in Red(S)$*

Proof: As we have noted, $T > S$ can be proved without using (I-TRANS) and without using (I-REFL) on function and record types. We prove by induction on the structure of the proof of $T > S$ of this kind. In the following proof, we write e' for a term one step reduction from e.

Suppose that S is a basic type, and $T > S$ is proved by (I-REFL). From the definition of $Red(e)$, e is strongly normalizable. Thus we prove by induction on $\nu(e)$, the maximum length of the reduction starting e, that $e|_S$ is strongly normalizable. $e|_S$ reduces to $e'|_S$ or e. The former is strongly normalizable from the induction hypothesis. The latter is also strongly normalizable. Thus $e|_S$ is strongly normalizable.

Suppose that S is a record type, and $T > S$ is proved by (I-LABEL). From Lemma 4,(1), e is strongly normalizable. Thus we prove by induction on $\nu(e)$ that $e|_S.l_i$ is reducible for all slots l_i of S. Since $e|_S.l_i$ is neutral, we need to prove, from Lemma 4,(3), that all the terms one step reduction from $e|_S.l_i$ are reducible. $e|_S.l_i$ reduces to $e'|_S.l_i$, which is reducible from induction hypothesis. It also reduces to $\{l_1 = e_1|_{S_1},\ldots,l_m = e_m|_{S_m}\}.l_i$ when e has the form $\{l_1 = e_1,\ldots,l_n = e_n\}$ and S has the form $\{l_1 : S_1,\ldots,l_m : S_m\}$.

Let T be $\{l_1 : T_1,\ldots,l_n : T_n\}$. Since $T_i > S_i$ is proved in the proof of $T > S$, $e_i|_{S_i}$ are reducible from induction hypothesis. From Lemma 5, $\{l_1 = e_1|_{S_1},\ldots,l_m = e_m|_{S_m}\}$ is reducible and from the definition of reducibility, $\{l_1 = e_1|_{S_1},\ldots,l_m = e_m|_{S_m}\}.l_i$ is reducible.

Suppose that S and T are function types of the forms $S_1 \longrightarrow S_1'$ and $T_1 \longrightarrow T_1'$ respectively, and $T > S$ is proved by (I-FUN). From induction hypothesis, $f_1|_{T_1}$ is reducible for $f_1 \in Red(S_1)$, and $g|_{S_1'}$ is reducible for $g \in Red(T_1')$. Let $f \in Red(S_1)$. From Lemma 4,(1), e and f are strongly normalizable. Thus we prove by induction on $\nu(e) + \nu(f)$ that $e|_{S_1 \longrightarrow S_1'}\ f$ is reducible for all $f \in Red(S_1)$. Since $e|_{S_1 \longrightarrow S_1'}\ f$ is neutral, we need to prove, from Lemma 4,(3), that all the terms one step reduction from $e|_{S_1 \longrightarrow S_1'}\ f$ are reducible. $e|_{S_1 \longrightarrow S_1'}\ f$ reduces to $e'|_{S_1 \longrightarrow S_1'}\ f$ or $e|_{S_1 \longrightarrow S_1'}\ f'$ which are reducible from induction hypothesis, or reduces to $(e\ f|_{T_1})|_{S_1'}$. From induction hypothesis, $f|_T$ is reducible. Since e is reducible, $e\ f|_{T_1}$ is reducible. Again from induction hypothesis, $(e\ f|_{T_1})|_{S_1'}$ is reducible. ∎

Theorem 8 *If $e : S'$ is a term with free variables x_i of type S_i, then e satisfies $e[x_i := f_i] \in Red(S')$ for all $f_i \in Red(S_i)$.*

Proof: Induction on the construction of e. If e is a constant, then it is reducible from the definition. If e has the form $\lambda x^S.\ e_1$ or $\{l_1 = e_1,\ldots,l_n = e_n\}$, $e[x_i := f_i]$ is reducible

$$
\begin{array}{lcl}
(x^S)^\diamond & = & x^S \\
c_{int}^\diamond & = & c_{int} \\
c_{bool}^\diamond & = & c_{bool} \\
(\lambda x^S.\ e)^\diamond & = & \lambda x^S.\ e^\diamond \\
(e\ f)^\diamond & = & e^\diamond\ f^\diamond|_S \quad \text{(Where, } e : S \longrightarrow S'.\text{)} \\
(\{l_1 = e_1, \ldots, l_n = e_n\})^\diamond & = & \{l_1 = e_1^\diamond, \ldots, l_n = e_n^\diamond\} \\
e.l^\diamond & = & e^\diamond.l \\
e|_S{}^\diamond & = & e^\diamond|_S
\end{array}
$$

Theorem 3 *When $e : S$ in λ_c, $e^\diamond$ is well typed and has type S in λ_c'.*

Proof: Induction on the structure of e. The only essential part is $(e\ f)^\diamond$. Let $e : S \longrightarrow S', f : T, T > S$. By induction hypothesis, $f^\diamond : T$ and $e^\diamond : S \longrightarrow S'$. Since $T > S$, $f^\diamond|_S : S$. Thus, from (T-APP'), $e^\diamond\ f^\diamond|_S : S'$ ∎

2.2.2 Strong Normalization of λ_c'

We prove the strong normalization of λ_c' extending the Tait's method for proving normalization theorem of the lambda calculus ([GLT89]).

We define, for each type S, a subset $Red(S)$ of *reducible* terms of type S inductively on the formation of S as follows:

- $Red(int)$, $Red(bool)$ are the set of expressions which are strongly normalizable.
- $Red(\{l_1 : S_1, \ldots, l_n : S_n\}) = \{e \mid e.l_i \in Red(S_i)\}$.
- $Red(S \longrightarrow S') = \{e \mid e\ f \in Red(S')$ for all $f \in Red(S)\}$.

We define the set of *neutral terms* as follows:

$$x^S \mid c_{int} \mid c_{bool} \mid e\ f \mid e.l$$

Reducible terms have the following three properties:

Lemma 4 *1. $e \in Red(S)$ then e is strongly normalizable.*

2. $e \in Red(S)$ and $e \triangleright e'$ implies $e' \in Red(S)$.

3. If e is neutral and every e' which is one step reduction from e satisfies $e' \in Red(S)$, then $e \in Red(S)$.

Proof: Prove by structural induction on types simultaneously. It is proved much the same way as in [GLT89], so we omit it. ∎

From the last property, a variable x^S belongs to $Red(S)$.

The following two lemmas are proved much the same way as in [GLT89].

put the proof of this judgement Σ. We can construct the proof of $e[x^S := f|_S] : T$ from Π' by substituting x^S with $f|_S$, and the occurrence of the rule (T-VAR) with the proof Σ. ∎

2.2 Strong Normalization of λ_c

We will study here some properties of λ_c, especially the strong normalization property.

2.2.1 λ'_c: A Calculus with Strict Type Agreement

The idea of implicit coercion enables us to write an expression which is otherwise type incorrect, and enlarges the set of expressions from the simply typed lambda calculus. However, it will be shown that an expression of λ_c can be translated into an expression which does not include implicit coercion in function application, by explicitly inserting coercion to each function applications. In this section, we define λ'_c; a system without implicit coercion, and give translation from λ_c to λ'_c. This translation is used in studying syntactic properties of λ_c such as strong normalization. Note that the relation between a calculus with subsumption rule and a calculus with explicit coercions are studied in [BTGS91],[BTGS90]. Our translation from λ_c to λ'_c corresponds to this kind of translation.

λ'_c is an extension of the simply typed lambda calculus with records and coercion expressions. The set of type expressions and pseudo expressions of λ'_c are those of λ_c. The typing rules of λ'_c are those of λ_c with the rule (T-APP) replaced by the following rule (T-APP'), which is used in the simply typed lambda calculus.

$$\text{(T-APP')} \quad \frac{e : S \longrightarrow S' \quad f : S}{e\ f : S'}$$

The reduction rules of λ'_c are those of λ_c with the rules (E-APP) and (E-APPC) replaced by the following rules (E-APP') and (E-APPC') respectively. (E-APP') is the rule of the simply typed lambda calculus, (E-APPC') is the modification of (E-APPC) so that the reduct is typed using (T-APP').

$$\begin{array}{ll} \text{(E-APP')} & (\lambda x^S.\ e)\ f \ \triangleright\ e[x^S := f] \\ \text{(E-APPC')} & (e|_{S \longrightarrow S'})\ f \ \triangleright\ (e\ f|_T)|_{S'} \quad (e : T \longrightarrow T') \end{array}$$

The rule (E-APPC') uses the type of e. Therefore, we need runtime typechecking when (E-APPC') is used, or to attach types to each subexpression, and keep them at runtime. In this sense, this calculus is not practical; only used to study some properties of λ_c.

We define a translation $e^\diamond$ from λ_c to λ'_c as follows:

A pseudo expression is called an expression when it has a type.

2.1.3 Reduction Rules

We use the notation $e \triangleright e'$ if e one step reduces to e', and $e \triangleright^* e'$ if e reduces to e' in more than one steps.

The reduction rules of λ_c are defined as follows:

$$
\begin{array}{ll}
\text{(E-LABEL)} & \{l_1 = e_1, \ldots, l_n = e_n\}.l_i \triangleright e_i \\
\text{(E-APP)} & (\lambda x^S.\ e)\ f \triangleright e[x^S := f|_S] \\
\text{(E-APPC)} & (e|_{S \longrightarrow S'})\ f \triangleright (e\ f)|_{S'} \\
\text{(E-COE1)} & \{l_1 = e_1, \ldots, l_n = e_n\}|_{\{l_1:S_1, \ldots, l_m:S_m\}} \triangleright \{l_1 = e_1|_{S_1}, \ldots, l_m = e_m|_{S_m}\} \\
\text{(E-COE2)} & e|_N \triangleright e \quad (N = int, bool)
\end{array}
$$

(E-LABEL) is the rule for slot selection. (E-APP) is the rule for function application. From this rule, the coercion function which is implicitly applied to the argument will become explicit after reduction.

The coercion $e|_S$ will be treated differently according to the type S. (E-COE2) says that if S is a basic type, then $e|_S$ is equal to e. It is justified because S is only compared with itself and e therefore has type S. (E-COE1) says that if S is a record type and e has the form $\{l_1 = e_1, \ldots, l_n = e_n\}$, then $e|_S$ is reduced to a record with slots restricted to slots of S, and slot values coerced to the types specified in S.

If S is a function type, we do not try to express the function $e|_S$ directly using a lambda expression. Instead, we introduce a reduction rule (E-APPC) of an application of a function of the form $e|_S$.

Let $T \longrightarrow T' > S \longrightarrow S'$ and $e : T \longrightarrow T'$. As we have noted, $T < S$ and $T' > S'$ because (I-FUN) is a necessary and sufficient condition. Then intended meaning of $e|_{S \longrightarrow S'}$ is a modification of the function e with the domain restricted to types bigger than S, and with the result coerced from type T' to S'. Therefore, the application $(e|_{S \longrightarrow S'})\ f$ is equivalent to $(e\ f)|_{S'}$. Here, the application $e\ f$ may be further reduced by (E-APP) or (E-APPC), and in the former case, the argument f of type greater than S will be coerced to type T.

One can easily prove the following subject reduction theorem.

Theorem 2 (Subject Reduction) *The above reduction rule is well defined. That is, when e reduces to e' and e has type S, then e' also has type S.*

Proof: It is checked rule by rule. (E-APP) must be the most difficult one. Suppose that $(\lambda x^S.\ e)\ f : T$, and let the proof of this judgement Π. Then the last rule of Π must be (T-APP). Therefore, $(\lambda x^S.\ e)$ has the type $S \longrightarrow T$, and f has the type U such that $U > S$. Then $(\lambda x^S.\ e) : S \longrightarrow T$ must be derived from the rule (T-FUN), and therefore, $e : T$. Let the proof of this judgement be Π'. From $f : U$ and $U > S$, we can prove that $f|_S : S$. We

$$
\begin{array}{rcl}
e & ::= & x^S \mid c_{int} \mid c_{bool} \\
& & \mid \lambda x^S.\ e \mid e\ f \\
& & \mid \{l_1 = e_1, \ldots, l_n = e_n\} \mid e.l \\
& & \mid e|_S
\end{array}
$$

Here, x^S are variables of type S, c_{int} are constants of type int, and c_{bool} are constants of type *bool*. l ranges over an infinite set of slot names.

The typing rules of λ_c are defined as follows:

(T-VAR) $$\frac{}{x^S : S}$$

(T-FUN) $$\frac{e : S'}{\lambda x^S.\ e : S \longrightarrow S'}$$

(T-APP) $$\frac{e : S \longrightarrow S' \quad f : T \quad T > S}{e\ f : S'}$$

(T-LABEL1) $$\frac{e_i : S_i}{\{l_1 = e_1, \ldots, l_n = e_n\} : \{l_1 : S_1, \ldots, l_n : S_n\}}$$

(T-LABEL2) $$\frac{e : \{l_1 : S_1, \ldots, l_n : S_n\}}{e.l_i : S_i}$$

(T-COE) $$\frac{e : T \quad T > S}{e|_S : S}$$

$\lambda x^S.\ e$ is a function abstraction and $e\ f$ is a function application. $\{l_1 = e_1, \ldots, l_n = e_n\}$ is a record construction, and $e.l$ is the selection of a value in slot l. Note that in the rule (T-APP), we do not require that the type of the argument and the type of the domain of the function agrees, but only require that there is an order relation between them. This is justified by the reduction rules defined in Section 2.1.3, which coerces the argument to the appropriate type when a function application expression is reduced. $e|_S$ denotes the coercion of e to type S. This expression is well formed only when e has a type T which is bigger than S.

An important characteristic of λ_c is that it does not have the following rule, which is usually called the subsumption rule.

$$\frac{e : T \quad T > S}{e : S}$$

This rule is included in most calculi with subtypes. As was written in the Introduction, this rule is not appropriate for our purpose. The absence of this rule gives us the theorem of unicity of type. That is,

Theorem 1 *For each pseudo expression e, there exists at most one type expression S which satisfies $e : S$.*

by the reduction reles in Section 2.1.3 . Note that e itself does not belong to S, as is the case in most calculi with subtypes. Note that the direction of this order is opposite to the subtype relation which is based on inclusion between types. Therefore, we call this order not the subtype relation but the coercible order. It is a kind of information order. That is, if $T > S$, then an expression of type T has, intuitively, more information than an expression of type S, and $e|_S$ is constructed by throwing off the excessive information.

A function of type $S \longrightarrow S'$ can naturally be considered as a function applicable to a value of a type T which is bigger than S in this order, by implicitly inserting a coercion function in a function application. Thus, an expression of type $S \longrightarrow S'$ works as a function applicable to types bigger than S and returns values of type S'.

With this view, it is natural to consider an order relation $T \longrightarrow S' > S \longrightarrow S'$ when $T < S$, where the coercion function from $T \longrightarrow S'$ to $S \longrightarrow S'$ is the restriction of the domain from the set of types bigger than T to the set of types bigger than S. It is also natural to consider an order relation $S \longrightarrow T' > S \longrightarrow S'$ when $T' > S'$, where a value of type $S \longrightarrow T'$ is coerced to type $S \longrightarrow S'$ by composing the coercion function from T' to S'.

Thus, we define the coercible order $>$ of λ_c as follows:

$$\text{(I-REFL)} \quad \frac{}{S > S}$$

$$\text{(I-TRANS)} \quad \frac{US > T \quad T > S}{U > S}$$

$$\text{(I-LABEL)} \quad \frac{S_i > S'_i \ (i = 1, \ldots, m) \quad n > m}{\{l_1 : S_1, \ldots, l_n : S_n\} > \{l : S'_1, \ldots, l_m : S'_m\}}$$

$$\text{(I-FUN)} \quad \frac{T < S \quad T' > S'}{T \longrightarrow T' > S \longrightarrow S'}$$

$>$ is an order relation, and the set of type expressions forms a poset with this order. It is easily shown that (I-TRANS) is needless, that is, every relation $T > S$ derived from this rule can be derived without using (I-TRANS). It is also easily shown that (I-REFL) can be restricted to basic types. Thus, a basic type can only be compared with itself, a function type can only be compared with another function type, and a record type can only be compared with another record type. Moreover, the rules (I-LABEL) and (I-FUN) give necessary and sufficient condition for the coercible order between record types and function types respectively. Thus, $>$ is decidable in λ_c.

2.1.2 Expressions and Type Assignment

To introduce expressions of each types, we first introduce pseudo expressions, and then give type inference rules. The pseudo expressions of λ_c are as follows:

Chapter 2

λ_C: A Calculus with Implicit Coercion.

In this chapter, we define and investigate the system λ_c; an extension of the simply typed lambda calculus with record types, type ordering, and implicit coercions. It is a subsystem of the system λ_m, the main system of this thesis. It can be shown that expressions in λ_c can be translated into a system without implicit coercions. In this sense, the implicit coercion in λ_c has little importance by itself. However, since the definition of the merge operator depends heavily on the fact that λ_m is based on implicit coercions, we should study how implicit coercions work before studying λ_m. Therefore, we first study a calculus with implicit coercion in this chapter, and extend it with the merge operator in the next chapter.

2.1 Definition of λ_c

2.1.1 Type System

The type system of λ_c is same as that of the simply typed lambda calculus with constant types int and $bool$, and record types.

$$S ::= int \mid bool \mid \{l_1 : S_1, \ldots, l_n : S_n\} \mid S \longrightarrow S'$$

Here, $l_1, \ldots, l_n$ are distinct slot names. We call int and $bool$ basic types, $\{l_1 : S_1, \ldots, l_n : S_n\}$ a record type, and $S \longrightarrow S'$ a function type. We identify those record types with the same set of slot name-and-type pairs.

When a calculus has record types, it is natural to introduce an order relation $T > S$ on types. The meaning of $T > S$ is that there is an implicit coercion function from T to S, and thus we can form an expression of type S from an expression of type T. For example, if e has a type $\{l_1 : S_1, \ldots, l_n : S_n\}$, and $S = \{l : S_1, \ldots, l_m : S_m\}$ $(m < n)$ is a type with a restricted set of slots of the same type, we can construct an expression of type S from e by ignoring the slots l_{m+1} to l_n. We denote this value by $e|_S$. The meaning of $e|_S$ is given

(E-APPL) $(e_1 \vee \ldots \vee e_n)\ f \ \triangleright\ e_{\phi(1)}\ f \vee \ldots \vee e_{\phi(l)}\ f$
(Where $e_{\phi(i)}$ are selected so that $AT(F_{\phi(i)}, T)$ exists for $F_{\phi(i)}$, the type of $e_{\phi(i)}$ and T, the type of f).

$AT(F,T)$ is a macro which exists when F is applicable to T.

The type of a merge of functions cannot be expressed by a pair of types: the source type and the target type. It is expressed by a list of monomorphic types: $[S_1 \longrightarrow S'_1, \ldots, S_n \longrightarrow S'_n]$. Since a function type determines the correspondence between an argument type and a result type, a function type can be regarded as a function from the set of types to itself. Precisely, this function is constructed as follows. We consider the partially ordered set $\mathcal{T}$ of types. $\mathcal{T}$ has the bottom element $\bot^{\mathcal{T}}$, which is the special type which meaning the type error. Consider the monotonic function space $[\mathcal{T} \longrightarrow \mathcal{T}]$, which is also a partially ordered set. Then $[S \longrightarrow S']$ can be considered as a step function in $[\mathcal{T} \longrightarrow \mathcal{T}]$. That is, it returns S' if the argument is bigger than S, and otherwise returns $\bot^{\mathcal{T}}$. And the meaning of $[S_1 \longrightarrow S'_1, \ldots, S_n \longrightarrow S'_n]$ can be considered as the least upper bound of $[S_i \longrightarrow S'_i]$ $(i = 1, \ldots, n)$.

Since we are forming a functional calculus, a function composed by the merge operator can be passed as an argument to another function, and coerced to an appropriate type by the (E-APP) rule. As for λ_c, there are two ways of coercing a function:

(E-APPC1) $(e|_F)\ f \ \triangleright\ (e\ f)|_{AP(F,T)}. \quad (f : T)$
(E-APPC1*) $(e|_F)\ f \ \triangleright\ (e\ f|_{AT(F,T)})|_{AP(F,T)} \quad (f : T)$

which will form calculi called λ_m and λ^*_m respectively. Here, $AP(F,T)$ is a macro on type expression which means the result of applying a function of type F to the type T, and $AT(F,T)$ is a macro on type expression which means the greatest type at which the F considered as a function on $\mathcal{T}$ changes its behavior.

Somewhat surprisingly, λ_m behaves completely differently to λ^*_m. (E-APPC1) means that the coerced function operates as applying the original function and coercing the result pointwise. This is a mathematically natural definition. However, some terms are non-normalizing with this rule.

On the other hand, (E-APPC1*) means that the coerced function is formed by merging monomorphic functions of types specified by the type F, and thus its behavior is determined by its behavior on a finite set of types given in F. This is a natural definition as a calculus, and strong normalization holds. However, coercion is not transitive in λ^*_m, which is essential for a basic calculus of a programming language. Both the calculi $\lambda_m$and λ^*_mhave Church-Rosser, Subject Reduction, and Unicity of Type properties.

The semantics of λ_m is given in a reflective domain in which coercible relation is modeled by a categorical structure called opfibration, and a merge is modeled by a least upper bound. This semantic domain is an extension of the domain constructed in ([Tsu92c], [Tsu90]).

We define and study λ_c in Chapter 2 λ_m in Chapter 3, and λ^*_min Chapter 3. In Chapter 5, we construct a reflective domain, and give denotational semantics to λ_m.

Here, the argument is coerced so that the reduct is type-correct without the help of the subsumption rule.

$f|_S$ is reduced to a value without $|_S$, or given the functionality of a value of type S by other rules. For example, $\{l_1 = e_1, \ldots, l_n = e_n\}|_{\{l_1=S_1,\ldots,l_m=S_m\}}$ is reduced to $\{l_1 = e_1|_{S_1}, \ldots, l_m = e_m|_{S_m}\}$, where $e_i : S_i$ and $m < n$. $e|_{S\longrightarrow S'}$ is directly given the functionality of a function from S to S' by the following rule:

(E-APPC) $\quad (e|_{S\longrightarrow S'})\ f \ \triangleright\ (e\ f)|_{S'}.$

There is an alternative for this rule. To coerce the argument to the type expected by $S \longrightarrow S'$:

(E-APPC*) $\quad (e|_{S\longrightarrow S'})\ f \ \triangleright\ (e\ f|_S)|_{S'}.$

We write λ_c^* for the calculus with this alternative rule. It can be shown that λ_c and λ_c^* form equivalent reduction systems.

1.4.2 λ_m: A Calculus with a Merge Operator

We extend λ_c with a merge operator and define λ_m. λ_m is new in two points. One is that the untyped universe is changed by introducing the merge operator, which makes the calculus completely different from conventional calculi. The other is that the type structure is also changed by introducing the type of a merge of functions.

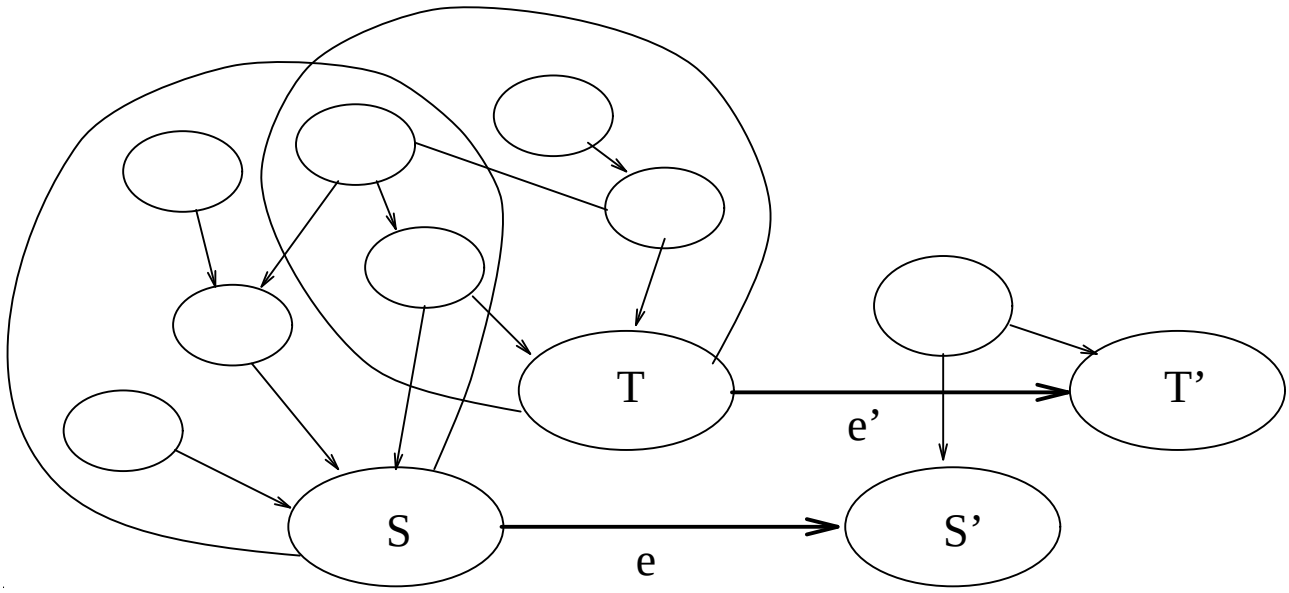


Figure 1.2: Merge of functions.

The meaning of a merge was informally given in Section 1.1; to apply all the functions applicable to the argument and merge the results. It is done by the rule (E-APPL)

runtime in contrast to the simply typed lambda calculus and its extensions which ignore type information at runtime. Therefore, the type of each expression must be uniquely determined and preserved by reduction, which is impossible with the subsumption rule. Instead, we view each function of type $S \longrightarrow S'$ as having the domain $\cup\{T|T \text{ inherits } S\}$ and codomain S', which implicitly coerces the argument to S before applying the function (Figure 1.1), and define the calculus λ_c. The implicit coercion in λ_c become explicit when it is reduced.

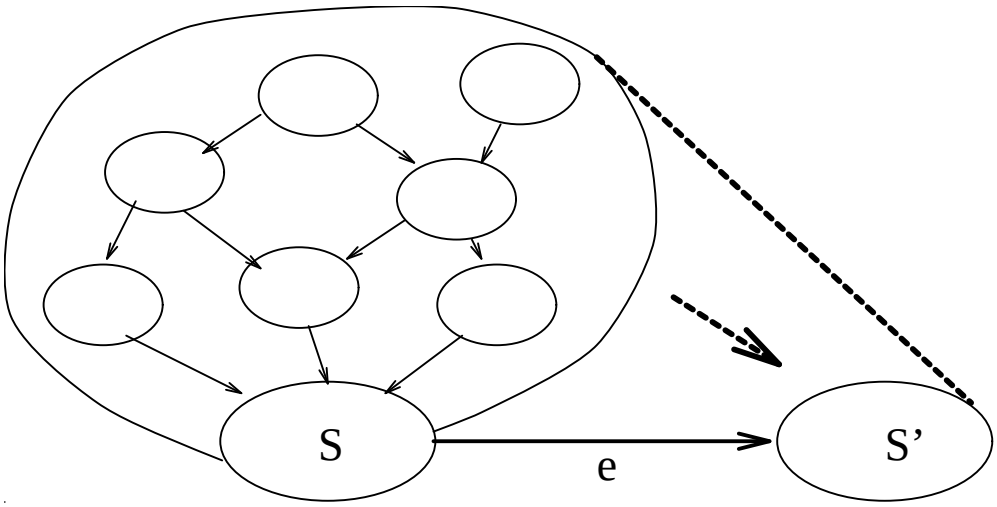


Figure 1.1: Applying a function defined on a supertype through a coercion.

When $T > S$ and $e : T$, we introduce a new expression $e|_S$ of type S. As we have noted, $T > S$ means that there is a coercion function from T to S. $e|_S$ denotes the coercion of e to type S.

We use the following typing rule which is also used in ([OBBT89]):

$$\text{(T-APP)} \quad \frac{e : S \longrightarrow S' \quad f : T \quad T > S}{e\ f : S'}$$

instead of the usual typing rule for function application

$$\frac{e : S \longrightarrow S' \quad f : S}{e\ f : S'}.$$

This new inference rule means that in each application expression $e\ f$, a coercion function from T to S is implicitly inserted around f. This implicit coercion function becomes explicit when the expression is β-reduced.

$$\text{(E-APP)} \quad (\lambda x^S.\ e)\ f \ \triangleright\ e[x^S := f|_S]$$

different notions. However, their is a similarity between these two type systems. Actually, if we view a merge of types as an intersection of types and reverse the order of types, the rules for the coercible order of λ_m are almost equivalent to the subtype relation of intersection types ([BCDC83]).

The meaning of an intersection type is the intersection of the set of terms defined by the two types. For example, if $int \longrightarrow int$ and $bool \longrightarrow bool$ are inferred to an expression, then it has the type $int \longrightarrow int \wedge bool \longrightarrow bool$. Therefore, if there is an untyped term which acts as addition to integers and negation to boolean values, then we can give the type $int \longrightarrow int \wedge bool \longrightarrow bool$. Thus, intersection typing is expected to serve as a foundation of overloading. However, it does not have the facility to construct an overloaded function by composing two lambda terms which denote an addition and negation. For this purpose, we need to extend the untyped universe. The merge of λ_m is partly motivated by offering this facility, and constructing a "true" calculus for overloading.

1.4 Overview of the Systems

We will define λ_m in two steps. We first allow the application of a function to a subtype of the argument type, and define the calculus λ_c. In λ_c, every function can be considered as a generic function whose domain is the subtypes of the argument type. Then, we add to λ_c the merge of functions, and define λ_m.

1.4.1 λ_c: A Calculus with Implicit Coercion

In λ_m, we need to apply a function defined in a supertype to a subtype. In the previous works on giving type theoretic foundations to object oriented languages, they used the following subsumption rule for this purpose ([Car84],[CW85]).

$$\text{(T-Subsumption)} \quad \frac{e : T \quad T \subset S}{e : S}$$

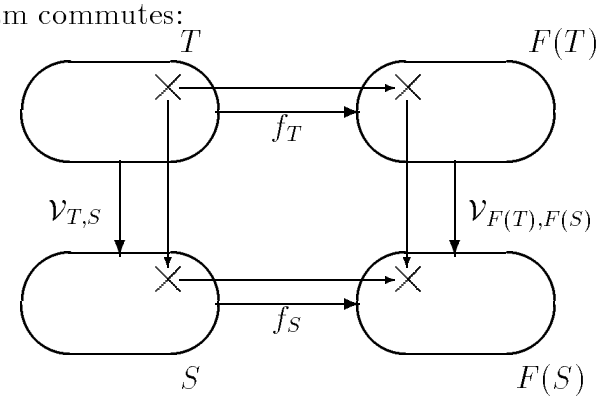
It says that when T inherits S, every element of T can be considered as an element of S. Thus, every function defined on S are applicable to elements of type T.

Though this rule is very natural, it contradicts our intuition that every object has its exact type in a pragmatic programming language ([OBBT89]). That is, e can be viewed as a value of type S, but e only belongs to T. With the subsumption rule, the exact type cannot be assigned to an expression. For example, the type of the expression $\lambda x^S.x$ is $S \longrightarrow S$. Therefore, if $e : T$ and $T \subset S$, then the type of $(\lambda x^S.x)e$ is inferred as S. However, this expression reduces to e. Of course it is type correct because $e : S$. But it is not the exact type of e.

For this reason, we cannot use the subsumption rule in our calculus. In order to dispatch appropriate functions depending on the type of the arguments, λ_m uses type information at

- F is a partial monotonic function.
- If $T > S$, $x : T$ is coercible to $y : S$, and F_S exists then $f_T\ x$ is coercible to $f_S\ y$.

That is, the following diagram commutes:



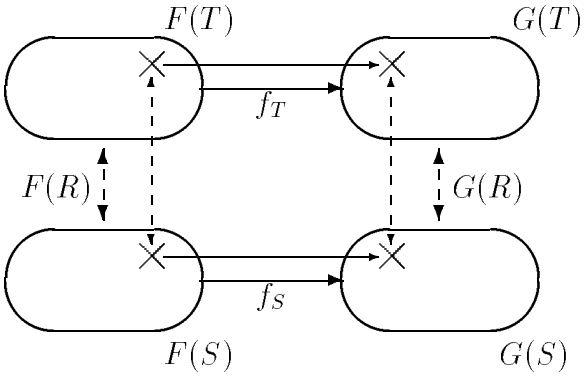
Here, $\mathcal{V}_{T,S}$ is the coercion function from T to S. It is a generalization of parametric polymorphism with the restriction that the type has the form $T \longrightarrow F(T)$ with F monotonic.

Coercion polymorphism is not a new idea; only stating that coercion and function application commutes. For example, consider the implicit conversion from *int* to *real* in an algol like language. Assignment of the sum of two integers to a real variable can be written `x := m + n`, which has two possible meanings `x := int-to-real(m +`$_{int}$ `n)` and `x := int-to-real(m) +`$_{real}$ `int-to-real(n)`. Coercion polymorphism says that the two expressions produce the same results. This is a natural condition for two functions to be assigned the same symbol ([Rey81]).

It seems that the property of a merge of function that the behavior on a subtype is a refinement of that on supertypes, means coercion polymorphism. However, contrary to our expectation, a function of λ_m does not, in general, preserve coercions. Actually, in some cases, $\mathcal{V}_{F(T),F(S)} \cdot f_T$ means a merge error, though $f_S \cdot \mathcal{V}_{T,S}$ does not. Though they are not equal, it can be said that the former is bigger than the latter by the order in $F(S)$ which is induced by the existence of the merge error. That is, they differ only in the possibility of merge errors. This non-commutativity of function applications and coercions may be a drawback of λ_m. However, it still has the nature that a function defined on a subtype is a refinement of that to a supertype. Thus, though the polymorphism of λ_m is weaker than coercion polymorphism, it is not an ad-hoc polymorphism. In Chapter 5, we give a denotational semantics of λ_m, and characterize more formally the polymorphism of λ_m.

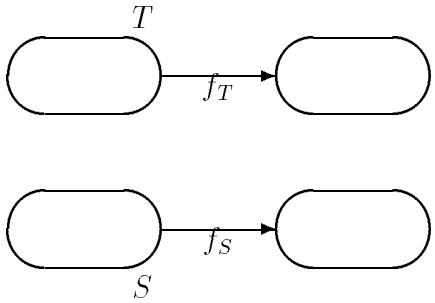
1.3.2 The Relation to Intersection Typing

With respect to the relation to type theory, there is a similarity between our merge type and intersection typing([BCDC83],[CC90], [CDCV81], [Rey91]). Intersection typing is an extension of Curry's system; the universe of untyped lambda terms is first given on which types are inferred to each term. On the other hand, λ_m is an extension of Church's system; types are first given and terms of the type are then defined. Therefore, they are completely



Reynolds characterized polymorphism using this in the first order case and gave an idea to extend it to the second order case ([Rey83]). This property have been later proved to be a natural condition for a model to have natural categorical properties ([Has91], [MR91]).

On the contrary, an ad-hoc polymorphic function is characterized in a model as a collection of unrelated functions.



One can see that there is a wide gap between these two notions. This study is to bridge the gap by studying merge of functions.

We suppose that an order relation exists on types. We view the subtype relation $T \subset S$ not as saying that every element of type T can also be considered as an element of S, but saying that every expression of type T can be coerced to an expression of type S. For instance, an expression of type $\{l_1 : S_1, l_2 : S_2\}$ can be coerced to an element of type $\{l_1 : S_1\}$ by ignoring the slot l_2. In this way, a subtype has more information than a supertype and a coercion function can be viewed as cutting off the excessive information. In this sense, we will change the notation to $T > S$ when T is a subtype of S, and we call this order $T > S$ the coercible order. You may find the direction of this order odd. However, when we consider the semantics, this order become very natural, and therefore we dare to change the direction of the order.

The coercible order is defined inductively by a set of rules, and coercion functions are defined along the order. When $T > S$, the coercion function can be considered as a relation between the types T and S.

Now we define that a polymorphic function is coercion polymorphic if it preserves the relations defined by the coercion functions. More precisely, a coercion polymorphic function is a collection of functions $f_t : t \longrightarrow F(t)$ which satisfy

- The range of f_t is upper closed; that is, if f_S is defined and $T > S$ then f_T is defined.

to one function when two functions are defined in two supertypes of the same level. And the other is that it is not natural to view a collection of unrelated operations on each type as one function.

There is another possible extension for a practical language. That is, to select other ways of combining the results. For example, it may be desirable to calculate the sum of all the integer values returned by all the functions. If the function to combine the results can be defined declaratively, then such a mechanism may be acceptable in a practical language based on our calculus.

One important restriction compared with the existing object oriented languages is that a method in a subtype cannot receive the result of super method and use it afterwards. For example, it is impossible to duplicate a slot value determined in a supertype function to a slot in a subtype. We will also discuss other directions of extensions in chapter 6 as future works.

1.3 Motivation from Type Theory

λ_m is also motivated by the study of polymorphism. Since a generic function is applicable to more than one types, it is a kind of polymorphic function. Here, we informally characterize the polymorphism formed by generic functions, or merge of functions, and explain the relation to parametric polymorphism.

1.3.1 A Generic Function as a Polymorphic Function

As to Strachey, polymorphism is classified into parametric polymorphism and ad-hoc polymorphism ([Rey83]). A parametric polymorphic function is a function which is defined uniformly over types. That is, though it can be viewed as a collection of monomorphic functions, they have the same algorithm written as a single lambda expression. Girard-Reynolds second order lambda calculus([Gir71],[Rey74]) enables one to treat parametric polymorphic functions as first order objects. On the other hand, an ad-hoc polymorphic function is a function whose effects on different argument types are unrelated. An example of such a function is a function which is written by type case.

A parametric polymorphic function is also characterized in a model as a function which preserves all the relations between all the types ([Rey83], [MR91], [Has90], [Has91], [BMM88]). To explain it in more detail, suppose that $F(t) \longrightarrow G(t)$ is a type of a polymorphic function with a type variable t. When there exists a relation R between two types S and T, then we can form a relation $F(R)$ between the types $F(S)$ and $F(T)$, and $G(R)$ between the types $G(S)$ and $G(T)$. A parametric polymorphic function is a collection of functions $\{f_t : F(t) \longrightarrow G(t) | t : type\}$ indexed by types such that for each relation R between S and T, when $x \in F(S)$ and $y \in F(T)$ are related by $F(R)$, then $f_S\ x$ in $G(S)$ and $f_T\ y$ in $G(T)$ are related by $G(R)$.

It is more declarative since we can judge which component is called without following the execution. The effect to a subtype is always a refinement of that to a supertype, since all the methods in the supertypes are also executed. It is a very naive realization of the idea that when defining a function to a subtype we only need to define the difference to the behavior on supertypes. This seems to work very well at least in the functional treatment, in which the result of a function is only represented in the return value.

For this to work, we need to "merge" the return values. We can rule out the merge of values of uncompatible types such as the merge of 1 and $\{l : 2\}$ using the type system. We add merge-error values to *int* and *bool*, to express the merge of, say, 1 and 2. A merge error occurs if one component of a generic function returns 1 and another component returns 2. We consider such a program erroneous; merge error is not the intended behavior of a program, which must be avoided by the programmer. Having extended basic types with error-values, we can define the merge of records, which has slots of both records and whose slot values are merged recursively.

1.2.3 Possible Improvements for a More Practical Language

Possible criticisms on our formalization are that it is not practical because:

- it is an unexpected result of a program that a merge error occurs in runtime;
- it restricts the expressive power that methods in supertypes are always activated;
- it is inefficient that methods in supertypes are always activated.

However, with a slight modification, we can answer these questions without losing the flavor that a generic function is a merge of functions. By reducing the merge of 1 and 2 to 1, that is, to set priority to the left value, and always putting the function defined on a subtype on the left, it is possible to override the result for a supertype in a subtype. It can be done by only modifying two reduction rules of λ_m. The resulting calculus does not contain a merge error.

It may seem that it is too restrictive that methods in supertypes are always activated. However, if we allow the overriding of a slot in a subtype, the only thing that cannot be done is to delete a slot which exists in a supertype. This is in fact not allowed in type systems with subtypes which adopt the subsumption rule ([Car84], [CW85], [CM91], [Car92], [CHC89],[CCH$^+$89],[Bru92]), and seems to be unnecessary in practical situations.

It may seem that it is inefficient to call the methods in supertypes when all the slots are overridden. However, we can judge through typechecking whether slots are overridden. Thus it is a matter of optimization to eliminate unnecessary function calls.

Thus, the overriding facility solves the above problems. It is much more natural from the practical viewpoint, much more easy to give semantics, and much more expressive. However, we do not adopt such a rule for three reasons. One is that this flexibility allows dirty programs as we have discussed. Another is that, it is difficult to give, without ad-hoc manner, priority

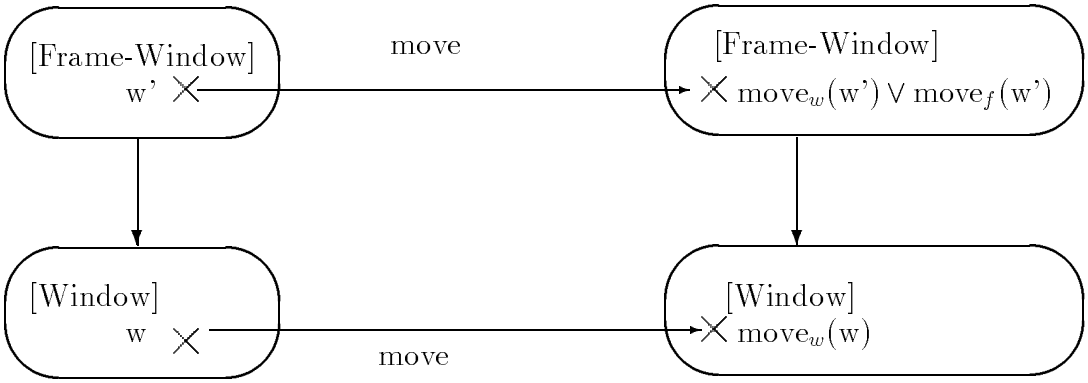
inside which `move-f` is called when it encounters `call-next-method`, and `move-t` is called nestedly from `move-f` when it encounters `call-next-method`. As a result, it will move all the three parts. Note that `move-f` is called from `move-t`, nevertheless `[Title-Window]` does not inherit `[Frame-Window]`. Therefore, the programmer cannot assume what is called by `call-next-method` when programming `move-t`. This method dispatch algorithm become more complicated if the function has more than one arguments.

This is a typical way of reusing methods in an object oriented language with multiple inheritance. However, it is too algorithmic, and semantically not clear in that the effect of `call-next-method` cannot be determined statically. Moreover, it allows unrestricted use of inheritance. Inheritance itself is a very declarative notion only stating that "`[Frame-Window]`" is a refinement of "`[Window]`". However, there is no notion that the effect of *move* to "`[Frame-Window]`" is a refinement of that to "`[Window]`". It is the programmer's responsibility to relate the effect of these methods.

Our calculus λ_m proposes a way of defining a method as a refinement of those in supertypes. In λ_m, we define to each type only the difference to the supertypes, and define a generic function as a "merge" of these functions. And when it is applied, call all the components applicable to the argument, and then "merge" the results. Thus each function do not call the same function in a supertype explicitly.

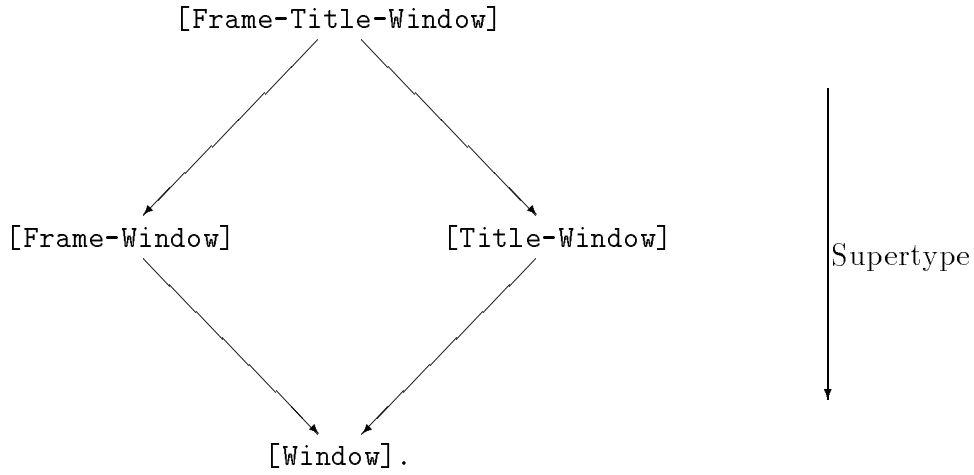
In the above example, we define `move` as the merge of $\texttt{move}_w$, $\texttt{move}_t$, and $\texttt{move}_f$. Here, $\texttt{move}_w$ is a function defined on `[Window]` which is same as `move-w`, $\texttt{move}_t$ is a function defined on `[Title-Window]` which returns only the title part of a window one unit-length on the right, and $\texttt{move}_f$ is a function defined on `[Frame-Window]` which only returns the frame part of a window one unit-length on the right. The title part and the frame part of a window are also represented as records.

When `move` is applied to a value of type `[Title-Window]`, $\texttt{move}_w$ and $\texttt{move}_t$ are applied and the "merge" of the results is returned. When `move` is applied to a value of type `[Frame-Title-Window]`, all the three functions are applied and the "merge" of a window, a frame, and a title component one unit-length on the right is returned.



However, existing object oriented programming languages allow defining a method unrelated to those in supertypes, which will often cause program errors ([Rey81]). Our calculus λ_m will focus on this problem, and propose a way of defining a method as a refinement of those in supertypes.

To start with, we explain the way of referring a method in a supertype in object oriented languages with multiple inheritance like CLOS. We consider the following types with multiple inheritance relation:



Here, `[Window]` is the type of a window, `[Frame-Window]` and `[Title-Window]` are the types of a window with a frame and with a title, respectively, and `[Frame-Title-Window]` is the type of a window with both. We suppose that these types are represented as records, and an inheritance relation exists as shown in the figure. In this figure, a supertype is given below a subtype.

On this type structure, we consider a generic function `move`, which takes one argument and moves it one unit-length right. The usual meaning of a subtype relation is if T is a subtype of S then every function applicable to S is also applicable to T. Thus, the move function `move-w` defined on the type `[Window]` can be applied to a `[Frame-Window]`. However, since it does not include the code to move the frame part, the frame is left unmoved on the original point. Therefore, the move function `move-f` on `[Frame-Window]` needs to be defined. In defining `move-f`, the code of `move-w` can be activated by sending a "super" method or calling the `call-next-method` function in CLOS terminology. In the same way, the `move-t` function on `[Title-Window]` can be defined which also refer `move-w` by `call-next-method`.

Then, the `move` function, when applied to a `[Frame-Title-Window]`, move the window, the title, and the frame simultaneously. In this case, method dispatch procedure first sorts the methods applicable to the argument, in this case, `move-t`, `move-f`, and `move-w` in this order, and apply the first one. The order between `move-t` and `move-f` is determined by the priority by which `[Frame-Title-Window]` inherits its supertypes. Thus `move-t` is first called

ject nor to each class; methods with a same name defined in all the classes are gathered into one entity called a generic function, which dispatches appropriate methods depending on the argument types. Thus, a message passing of an object oriented language can be considered as a function call to a generic function. Though it is difficult to model the encapsulation mechanism which is one of the characteristics of object orientedness, it has some advantages.

- *Multi-methods* which allow the selection of methods based on all the arguments, are available.
- The behavior of a method in a class depends on the definition of methods of the same name in other classes because of the use of *super methods*. Therefore, it is natural to consider that all the methods with the same name define one function.
- It is recommended that a method for a subclass should be a refinement of the same method in a superclass. The meaning of the term "refinement" becomes clear when we consider a generic function.
- Because a generic function is a kind of function, we can define a generic function as an extension of a usual function, and allow functional arguments. It leads to a new paradigm merging object oriented and functional programming.
- When we consider a type system, "inheritance" and "subtyping" do not coincide if we use the naive object oriented model ([CCH+89], [CHC89]). Since a method is not a part of an object in a generic function model, this problem does not exist.

Actually, Common Lisp Object System, or CLOS ([BDG+88]), which is a standard for merging object oriented programming with Common Lisp, implements message passing using generic functions.

We will use this formalization, and consider a message as a generic function, which is applicable to more than one types.

1.2.2 A Generic Function Defined by Merging Functions

It is recommended that the behaviors of a generic function to a super type and to a subtype are related. One of the distinctive features of object oriented programming is that a subtype inherits properties of supertypes, and the programmer only needs to define in a subtype the difference to supertypes. A programmer will add methods which do not exist in supertypes, or override methods defined in supertypes. Even when overriding a method, the new method will usually refer to methods in supertypes by "super" methods and only specify the part particular to the data structure added in that type. That is, a programmer will write a method in a subtype as a refinement of those in supertypes. This kind of programming only defining the difference from the same methods in supertypes is called the Mixin-based programming ([BC90]). Programs written in this style are readable, easy to maintain, and less prone to error.

We apply this idea to a typed calculus. We consider in λ_m a hierarchical type structure whose order relation is defined through the existence of a coercion function from a subtype to a supertype. Therefore, a function of type S to S' can be considered as a partial function with the domain $\cup\{T|T \text{ inherits } S\}$ and codomain S', by coercing the argument to S before applying the function. We consider the merge of such partial functions. That is, when $e_1 : S_1 \longrightarrow S'_1, \ldots, e_n : S_n \longrightarrow S'_n$ and $f : T$, $(e_1 \vee \ldots \vee e_n)\ f$ is reduced to $e_{\phi(1)}f \vee \ldots \vee e_{\phi(l)}f$, where $\phi(i)$ are selected so that $T > S_i$. Note that the applicability is checked via types of expressions.

An extension of the simply typed lambda calculus which treats a notion of merge of functions is studied in [CGL92]. It selects only one function from the component list, and thus corresponds to an overloaded function. A merge of functions activates several functions and merges their effects. Therefore, component functions work together to calculate a result; and thus the behavior of a function on each type is closely related. In this sense, it is not an overloaded function but a kind of parametric polymorphic function.

In order to select applicable functions, λ_m uses type information at runtime in contrast to the simply typed lambda calculus and its extensions which ignore type information at runtime. For this purpose, the type of each expression must be uniquely determined and preserved by reduction, which is accomplished by avoiding the use of subsumption rule and making the implicit coercion explicit at runtime. This approach has the drawbacks that it cannot express the polymorphic nature of object oriented languages. However, as is given in Chapter 6, this calculus is designed so that it serves as a basic calculus on which polymorphic calculus can be constructed, just as the simply typed lambda calculus serves as the base of polymorphic lambda calculus.

In the rest of this introduction, we first show that a merge of functions corresponds to a "good-natured" message in object oriented languages. Next, we show that a merge of functions is a kind of parametric polymorphic function. After that, we overview calculi defined in this thesis.

1.2 Motivation from Object Oriented Languages

1.2.1 An Object Oriented Model Based on Generic Functions

Previous type systems that are designed to study the foundation of object oriented programming formalized objects as records, and classes as record types([Car84], [CW85], [OB89], [OBBT89], [PC89], [Rem89], [HP91], [Car92], [CCH+89], [CHC89], [CM91], [Wan87], [Wan89]). In these works, objects are regarded as self-contained entities. That is, methods applicable to an object are contained in a slot of the record itself. This is a naive straightforward understanding of an object.

On the other hand, the same calculation mechanism can be modeled using generic functions, which will give us another viewpoint on object oriented computing. In this view too, objects are records and classes are record types. However, methods do not belong to each ob-

Chapter 1

Introduction.

1.1 Merge of Functions

It is natural to consider data objects as giving partial descriptions, and merge as a primitive operation to obtain an object which has all the information contained in each object. With this idea, a merge of records, which is also called a join or a concatenation of records in the literature, has been studied elaborately. A record value can be considered as a partial description of the real object, and records can be merged to form a more detailed description ([Oho90]). For example, $\{name : \{first : Alan\}\}$ and $\{name : \{last : Johnson\}, occupation : teacher, age : 30\}$ are two partial descriptions of a person whose merge should be $\{name : \{first : Alan, last : Johnson\}, occupation : teacher, age : 30\}$. In the area of database programming language, it is natural to represent data objects as records, and consider record merge as a primitive operation([Oho90]). Records and record merge have also been used in designing fundamental calculi of object oriented computing. [OB88] and [Wan89] present extensions of ML-like type systems which include records and a record merge operator. [HP91] presents a higher order record calculus with a merge operator.

In this paper, we extend the notion of merge to functions in the framework of typed lambda calculus, and form a calculus called λ_m. A merge of functions corresponds to a "good-natured" message in object oriented languages. It is good-natured in that the behavior on a subtype is a refinement of that on supertypes.

Partial functions have the natural notion of merge. That is, when e_1 and e_2 are partial functions with domains V and W respectively, and e_1 and e_2 operates equally to values in $V \cap W$, then their merge $e_1 \vee e_2$ is a function defined on $V \cup W$, which is same as e_1 on V, and same as e_2 on W. If the codomain of the functions also have the notion of merge, then we do not need the condition that they coincide on the intersection of their domains, and define the merge of functions $e_1, \ldots, e_n$ as the partial function whose domain is the union of the domains of the component functions, and operates as

$$(e_1 \vee \ldots \vee e_n)\ f = e_{\phi(1)} f\ \vee \ldots \vee e_{\phi(l)} f.$$

Here, $e_{\phi(i)}$ are all the functions applicable to f.

6 Conclusions and Future Works **90**

A The Definition of λ_c **97**
A.1 Type Expressions 97
A.2 Coercible Order 97
A.3 Expressions and Type Assignment 97
A.4 Reduction Rules 98
A.5 The Definition of λ_c^* 98

B The Definition of λ_m **99**
B.1 Type Expressions 99
B.2 The Coercible Order 100
B.3 Expressions and Type Assignment 100
B.4 Reduction Rules 101
B.5 The definition of λ_m^* 102

3.1.4 Reduction Rules . . . 32
3.2 Examples . . . 34
3.2.1 Move function . . . 34
3.2.2 Overloading . . . 36
3.2.3 Multi-Methods . . . 37
3.2.4 Union types . . . 38
3.3 Some Syntactic Properties of λ_m . . . 38
3.3.1 Decidability of the Coercible Order . . . 39
3.3.2 Church-Rosser Property . . . 43
3.3.3 Non-preservation of Coercions . . . 46
3.3.4 The Meaning of a Function Type . . . 46
3.3.5 Non-normalizing Terms . . . 47
3.3.6 Transitivity of Coercions . . . 49

4 λ_m^*: Another Calculus with a Merge Operator 52
4.1 Definition of λ_m^* . . . 52
4.2 Non-transitivity of Coercions . . . 53
4.3 Strong Normalization of λ_m^* . . . 54
4.3.1 λ_m': A Calculus with Strict Type Agreement . . . 54
4.3.2 Strong Normalization of λ_m' . . . 55
4.3.3 Strong Normalization of λ_m'' . . . 57
4.3.4 Strong Normalization of λ_m^* . . . 59
4.4 Other Syntactic Properties of λ_m^* . . . 60
4.4.1 The Meaning of a Function Type in λ_m^* . . . 60
4.4.2 A Note on the Rule (E-APPL) . . . 62
4.5 $\lambda_m^\sharp$: Yet Another Calculus with a Merge Operator . . . 63

5 Denotational Semantics of λ_m 65
5.1 M-domain: A Semantic Domain for λ_m . . . 65
5.1.1 Coercion as a Coindexed Category . . . 65
5.1.2 Grothendieck Construction . . . 67
5.1.3 A Merge as the Least Upper Bound . . . 69
5.1.4 Record Space and Function Space . . . 71
5.1.5 Some Properties of M-domains . . . 75
5.1.6 The Construction of a Reflective M-domain . . . 76
5.1.7 Construction using Coindexed Categories . . . 80
5.2 Semantics of λ_m . . . 83
5.2.1 Semantics of Type expressions . . . 83
5.2.2 Semantics of the Coercible Order . . . 86
5.2.3 Semantics of Expressions . . . 87
5.2.4 Discussion on the Reflectiveness of the Semantic Domain . . . 89

Contents

1 Introduction. 1

1.1 Merge of Functions 1
1.2 Motivation from Object Oriented Languages 2
1.2.1 An Object Oriented Model Based on Generic Functions 2
1.2.2 A Generic Function Defined by Merging Functions 3
1.2.3 Possible Improvements for a More Practical Language 6
1.3 Motivation from Type Theory 7
1.3.1 A Generic Function as a Polymorphic Function 7
1.3.2 The Relation to Intersection Typing 9
1.4 Overview of the Systems 10
1.4.1 λ_c: A Calculus with Implicit Coercion 10
1.4.2 λ_m: A Calculus with a Merge Operator 12

2 λ_c: A Calculus with Implicit Coercion. 14

2.1 Definition of λ_c 14
2.1.1 Type System 14
2.1.2 Expressions and Type Assignment 15
2.1.3 Reduction Rules 17
2.2 Strong Normalization of λ_c 18
2.2.1 λ_c': A Calculus with Strict Type Agreement 18
2.2.2 Strong Normalization of λ_c' 19
2.2.3 Strong Normalization of λ_c'' 21
2.2.4 Church-Rosser Property of λ_c'' 23
2.2.5 Strong Normalization of λ_c 24
2.3 λ_c^*: Another Calculus with Implicit Coercion 24

3 λ_m: A Calculus with a Merge Operator 26

3.1 Definition of λ_m 26
3.1.1 Type Expressions 26
3.1.2 The Coercible Order 29
3.1.3 Expressions and Type Assignment 31

Acknowledgements

I am deeply indebted to my advisor Professor Nobuo Saito for his consistent support and many illuminating advices over the years. The years in Shonan Fujisawa Campus was a very pleasant one due to the stuff members and to the members of Saito Laboratory.

The greatest thanks to Professor Masami Hagiya, who gave me the motivation to study the theoretical foundation of object orientedness, and gave me invaluable advices throughout this research. Many people enlightened me the techniques needed to accomplish this work. Professor Susumu Hayashi's seminor on the first course of type theory enlightened me a lot. Professor Masako Takahashi introduced me the Parallel Reduction method for proving Church-Rosser Property of a calculus. Dr. Ryu Hasegawa introduced me the naturality of models of polymorphism.

Dr. Atsushi Ohori and Professor Kuniaki Mukai gave me invaluable comments on an early version of this thesis. Many other people helped me with their comments and through discussions. I would like to thank Dr. Tatsuya Hagino, Hiroyuki Miyoshi, Takafumi Sakurai, Sinya Nishizaki, Daisuke Suzuki, and many others.

I also would like to thank Marianne Santillo for reading this thesis and supplied valuable advice on grammar and writing style.

I would especially like to thank Michie Tsuiki, my wife, for her constant encouragement and mental support.

Abstract

An extension of the simply typed lambda calculus with record types, implicit coercions, and a merge operator is defined and investigated. A merge of functions corresponds to a message in object oriented languages, or a generic function in CLOS terminology. This calculus, λ_m, is motivated by the message-passing mechanism of object oriented languages, and is intended to be the foundation of object oriented computing. λ_m restricts generic functions so that the behavior on a subtype is a refinement of that on a supertype. In this sense, a generic function of λ_m is not an overloaded function but a kind of polymorphic function.

In order to dispatch appropriate functions depending on the type of the arguments, λ_m uses type information at runtime in contrast to the simply typed lambda calculus and its extensions which ignore type information at runtime. For this purpose, the type of each expression must be uniquely determined and preserved by reduction, which is accomplished by avoiding the use of the subsumption rule and making the implicit coercion explicit at runtime.

Since we are forming a functional calculus, we need to define the coercion of a function composed by the merge operator. There are two different ways of coercing a function, resulting in two calculi, λ_m and λ_m^*, with different behavior. Though λ_m has a mathematically natural model of coercions, some terms are non-normalizing in λ_m. Though λ_m^* has a natural model of function types and has strong normalization property, it does not have the transitivity of coercions, which is required for a basic calculus of a programming language. Both λ_m and λ_m^* has Church-Rosser, Subject Reduction, and Unicity of Type properties. The semantics of λ_m is given in a reflective domain in which coercible relation is modeled by a categorical structure called opfibration, and a merge is modeled by a least upper bound.

A Record Calculus with a Merge Operator

This dissertation was submitted to the Graduate School
of Science and Technology, Keio University
in partial fulfillment of the requirements
for the Doctor of Philosophy degree.

Hideki Tsuiki
(tsuiki@sfc.keio.ac.jp)

November, 1992