Math. Struct. in Comp. Science (1998), *vol.* 8, *pp.* 321–349. Printed in the United Kingdom
© 1998 Cambridge University Press

A computationally adequate model for overloading via domain-valued functors

HIDEKI TSUIKI

Department of Fundamental Science, Kyoto University, Yoshida-Nihonmatsu, Sakyo-ku 606-8316, Kyoto, Japan.

Email: tsuiki@i.h.kyoto-u.ac.jp

Received 4 December 1995; revised 15 November 1997

We give a denotational semantics to a calculus $\lambda_{\otimes}$ with overloading and subtyping. In $\lambda_{\otimes}$, the interaction between overloading and subtyping causes self application, and non-normalizing terms exist for each type. Moreover, the semantics of a type depends not on that type alone, but also on infinitely many others. Thus, we need to consider infinitely many domains, which are related by an infinite number of mutually recursive equations. We solve this by considering a functor category from the poset of types modulo equivalence to a category in which each type is interpreted. We introduce a categorical constructor corresponding to overloading, and formalize the equations as a single equation in the functor category. A semantics of $\lambda_{\otimes}$ is then expressed in terms of the minimal solution of this equation. We prove the adequacy theorem for $\lambda_{\otimes}$ following the construction in Pitts (1994) and use it to derive some syntactic properties.

1. Introduction

In theoretical studies, the notion of overloading has often been treated as something *ad hoc* and trivial; an overloaded function is simply a finite product of functions from which one component is selected at compile time, so it does not complicate the computational behaviour of the calculus. However, recent studies have shown that when the simply typed lambda calculus is extended with both overloading and subtyping, a kind of self application occurs and it becomes non-normalizing (Tsuiki 1992; Tsuiki 1994; Castagna *et al.* 1995). This suggests that overloading cannot be interpreted as a simple product when subtyping exists, and we need to consider a richer structure.

In this paper, we construct a domain theoretic model of $\lambda_{\otimes}$, an extension of the simply typed lambda calculus with subtyping and the merge operator. The merge operator provides a way of defining an overloaded function. Although $\lambda_{\otimes}$ has neither the recursion operator nor the notion of recursive types, an overloaded function may be applicable to itself due to the interaction between subtyping and overloading. This means that we cannot define the semantics of a type inductively, as in the case of the simply typed lambda calculus and PCF, but rather we need to consider recursive equations between domains. Moreover, we will find that the semantics of a type depends not only on itself but

https://doi.org/10.1017/S0960129598002539 Published online by Cambridge University Press

on infinitely many types including itself. This means that we need to construct infinitely many domains, which are related by an infinite number of mutually recursive equations.

Let $(\mathbf{T}, \leqslant)$ be the poset of all types modulo equivalence extended with a top element, and $\mathbf{C}$ be a category in which each type is interpreted. Each object of $\mathbf{C}$ should have a pointed order structure in order to interpret non-normalizing terms. Since a subtype relation is interpreted as a strict coercion function from a subtype to a supertype, the function D that assigns a domain to each type can be considered as a functor from $\mathbf{T}$ to $\mathbf{C}_!$. Here, the poset $\mathbf{T}$ is considered as a category and $\mathbf{C}_!$ is the subcategory of $\mathbf{C}$ with strict morphisms only. In this setting, the conditions we need to solve can be expressed as the following single recursive equation in the functor category $\mathbf{Fun}(\mathbf{T}, \mathbf{C}_!)$:

$$D = D_{\mathbf{B}} \times [D \Rightarrow D]. \tag{1}$$

Here, $[D \Rightarrow D]$ is a functor from $\mathbf{T}$ to $\mathbf{C}_!$ corresponding to the overloaded function space.

Dana Scott solved the domain equation $U = U^U$ in the category of complete lattices and continuous functions. We need to do the same kind of construction in a functor category. Smyth and Plotkin (1982) generalized Scott's construction to arbitrary categories and gave conditions under which such a domain equation is solvable. We use their results to give conditions on the category $\mathbf{C}$ under which the above equation in $\mathbf{Fun}(\mathbf{T}, \mathbf{C}_!)$ is solvable. Following that, we give a semantics to $\lambda_\otimes$ for the case in which $\mathbf{C}$ is the category of pointed cpo's and continuous functions.

In Plotkin (1977), the computational adequacy property of the language PCF with respect to the standard cpo model is shown, and thus some equivalences between terms are proved through the semantics. We will do an analogous study for $\lambda_\otimes$ with the model constructed over the minimal solution of (1). The adequacy property of PCF is usually proved using the notion of computability or the notion of formal approximation. However, they cannot be applied to $\lambda_\otimes$ directly because the semantics of types is not defined inductively. Recently, A. M. Pitts (Pitts 94; Pitts 1996) has developed a technique for proving the adequacy of a language with respect to the model constructed over the minimal invariant of a recursive domain equation, by defining a formal approximation relation as a fixed point of a constructor of mixed variance over relations. We show the adequacy of $\lambda_\otimes$ by following that construction. Using adequacy, we obtain some syntactic properties of the calculus: transitivity of coercions; equivalence between terms of equivalent types; and the relation between the notion of merge and overloading.

We give the intuitive idea behind the notion of merge in the next section, and introduce the calculus $\lambda_\otimes$ and its normalization-related properties in Section 3. In Section 4, we construct the semantic functor D by solving an equation in a functor category, and in Section 5, we give a denotational semantics to $\lambda_\otimes$. Finally, we prove the adequacy theorem and some syntactic properties in Section 6.

2. The merge operator as an overloaded function constructor

2.1. *Calculi with overloading and subtyping*

First, we give the fundamental idea of calculi with overloading and subtyping. Figure 1 gives us an intuitive meaning of an overloaded function over a subtype structure. An

https://doi.org/10.1017/S0960129598002539 Published online by Cambridge University Press

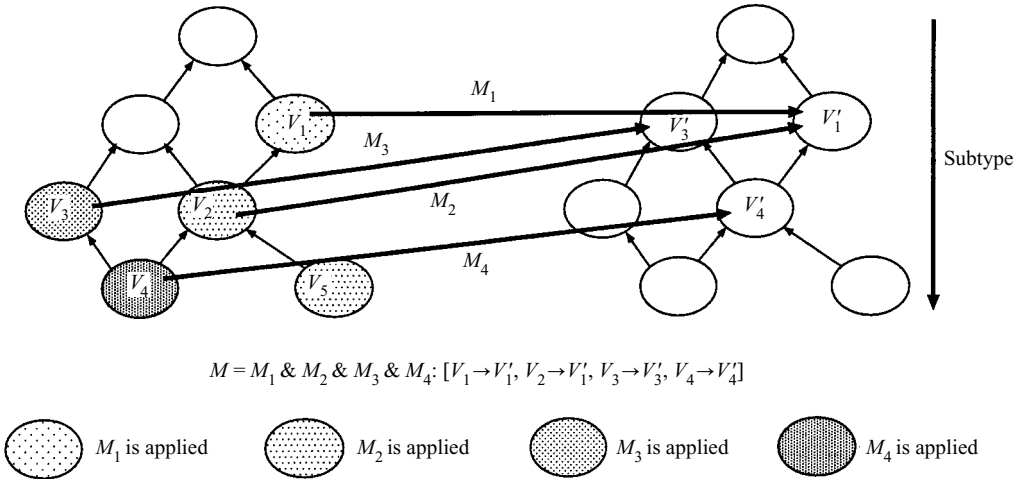


Fig. 1. Intuitive meaning of an overloaded function.

overloaded function is defined as a collection of monomorphic functions, and its type is expressed as a list of component function types. When it is applied, the branch defined on the closest super type is selected, and applied to the argument. In this example, M is applicable to subtypes of V_i $(i = 1, \ldots, 4)$. When M is applied to an argument N of type, for example, V_5, M_2 is selected and is applied to N. Note that the type of a term is used at runtime to determine the most appropriate branch. This is in contrast to the simply typed lambda calculus and its extensions, which ignore type information at runtime.

This kind of calculus has been studied with the aim of providing a foundation for typed object-oriented languages. It models an object as a record that contains only instance variables, a class as a record type, a message as an overloaded function defined by combining all the methods with the same name, and a message-sending as an application of an overloaded function. Thus, when a message is sent to an object, the method defined on the closest superclass is selected and applied to the argument. The $\lambda\&$-calculus (Castagna *et al.* 1995) is an example of a calculus with this kind of overloading.

2.2. *The merge operator*

The calculus $\lambda_{\otimes}$ uses a different constructor in defining an overloaded function, which allows us to define the behaviour of an overloaded function on a subtype as a modification of those of the supertypes, and which has a simpler formalization.

It has the 'merge' operator $\otimes$, which takes two overloaded functions and produces an overloaded function. Instead of selecting only one branch, a merge of functions computes the result by selecting all the component functions applicable to the argument, applying them, and merging their results. That is, $(M_1 \otimes M_2)\ N$ is reduced to $M_1\ N$ if M_1 is applicable to N, $M_2\ N$ if M_2 is applicable to N, and to $(M_1\ N) \otimes (M_2\ N)$ if both are

https://doi.org/10.1017/S0960129598002539 Published online by Cambridge University Press

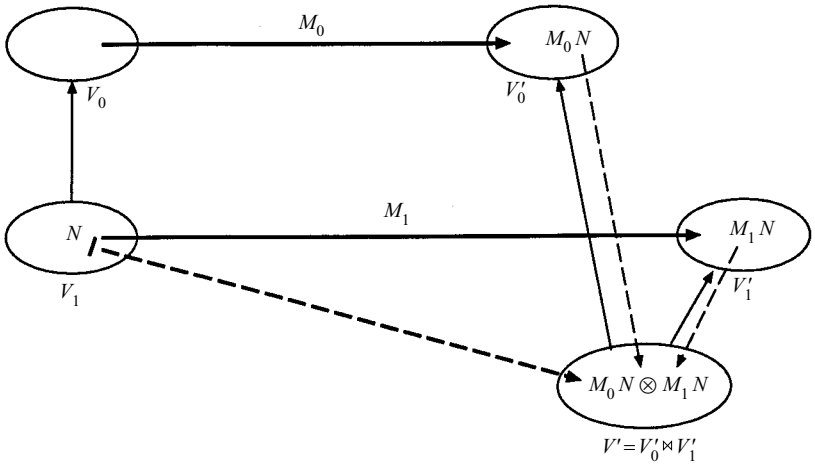


Fig. 2. An example of the use of the merge operator.

applicable. This is in contrast to the $\lambda\&$-calculus in which only one function component is selected.

For this definition, we need to define the merge of the result values. In $\lambda_\otimes$, we only consider basic types and overloaded function types. We have already mentioned the meaning of a merge of overloaded functions. As for the merge of two terms of the same basic type, we have many choices. Actually, we can use any set of functions $\{f_B : B \times B \to B \mid B \text{ is a basic type}\}$ to form a calculus of this kind. We have made our choice so that a merge $M \otimes N$ of basic type terms M and N is reduced to N. That is, we adopt the rule that the right-hand value overrides the value on the left. We do not allow the merge of an overloaded function and a basic type value, or the merge of two basic type values of different types; we cannot find any practical meaning in such expressions.

The notion of merge is designed as a new way of defining a subclass method in object-oriented programming. In most cases, we define a subclass method as a modification of supertype methods by first activating them by, for example, sending a message to *super*, and then writing the part that is peculiar to the subtype. We formalize in $\lambda_\otimes$ this idea of writing only the difference in a subclass. As an example, suppose that M_0 is an overloaded function with only one branch defined on V_0 with the result type V_0', and that we want to extend it with a branch defined on a subtype V_1 of V_0 to form an overloaded function M (Figure 2). We assume that V', the result type of M to V_1, is a subtype of V_0', since the return value of a subtype method is usually more specific than that of a supertype method. Instead of giving the full description of M to V_1 as the branch for V_1, we only specify as $M_1 : [V_1 \to V_1']$ the difference from the behaviour of M_0 to V_1 and construct the merge term $M = M_0 \otimes M_1$. Here, V_1' is an appropriate type to specify the difference, which satisfies $V' = V_0' \bowtie V_1'$ (the merge type of V_0' and V_1', which we will explain below). When M is applied to $N : V_1$, both M_0 and M_1 are activated and the results are merged to form a value $M_0\ N \otimes M_1\ N$ in V'.

To give more intuition to merge, let the result values $M_0\ N$ and $M_1\ N$ be records.

https://doi.org/10.1017/S0960129598002539 Published online by Cambridge University Press

As we shall see in Section 3.3, records can be expressed as overloaded functions in $\lambda_{\otimes}$. Assume $M_0\ N$ to be a record $\{\texttt{one} = N_1, \texttt{two} = N_2\}$ and $M_1\ N$ a record $\{\texttt{two} = N_2', \texttt{three} = N_3'\}$, with N_2 and N_2' of the same type. Then $M_0\ N \otimes M_1\ N$ is equivalent to $\{\texttt{one} = N_1, \texttt{two} = N_2', \texttt{three} = N_3'\}$ in the sense of Definition 3.5 below, because records are merged recursively by the reduction rule of an overloaded function application and the right-hand value overrides on basic type values. In this way, the result of the subtype method inherits some parts of the supertype method (`one` slot), overrides some parts (`two` slot), and extends some parts (`three` slot).

When we do not want to inherit the behaviour of the supertype methods, we can put $V_1' = V'$ and completely override the supertype methods. In this case, a merge expression behaves just like overloading, as we shall prove formally in Section 6.3. Therefore, our notion of merge is a generalization of the notion of overloading. At another extreme, we can completely inherit the supertype methods by not specifying the subtype component. In this sense, the notion of merge gives us a way of controlling the extent to which a subtype method overrides.

With this notion of merge, the syntax and the semantics of the calculus becomes simpler than in calculi with the conventional overloading operator. In the $\lambda\&$-calculus, in order to ensure that there exists only one branch defined on the closest super type, there is a condition on the form of an overloaded type, which does not exist in $\lambda_{\otimes}$. The $\lambda\&$-calculus has the property that equivalent types do not have the same meaning, and thus the type structure must be treated as a pre-ordered set, not a partially ordered set, whereas equivalent types have the same meaning in $\lambda_{\otimes}$.

In Tsuiki (1994), we introduced *generic function* as a new term for functions that are composed by merge. However, we will use the word overloaded function (or simply 'function' because we only treat overloaded functions in our calculus) here, because our semantic construction applies to calculi with overloading in general.

2.3. *Late binding vs. early binding*

One of the difficulties in considering a calculus with subtyping is that the type of a term is not preserved by β-reduction. For example, when $U \leqslant V$ and $M : U$, we have that $(\lambda x^V.\ x)\ M$ has type V, whereas it has type U when it is β-reduced. Usually, this is avoided by introducing the subsumption rule, and considering a term of type U as also belonging to type V. However, it is not easy to adopt this rule in the presence of overloading, because the type of a term is used in the reduction and needs to be uniquely defined.

In $\lambda_{\otimes}$, this problem is solved by considering $U \leqslant V$ as showing the existence of an inherent coercion function from U to V instead of using the subsumption rule and considering subtyping as set inclusion. We use the following rule instead of the β-rule:

$$(E - APP) \quad (\lambda x^V.M)N \ \triangleright\ M[x^V := N|_V]\ .$$

Here, $N|_V$ is a term denoting the coercion of N to V. The intuitive meaning of $N|_V$ is that it is N but the functionality is restricted to V, and thus behaves as a value of type V. In this way, the calculus we treat here has the unicity of types and the subject reduction properties. This rule also means that a branch selection of a function application to $N|_V$

https://doi.org/10.1017/S0960129598002539 Published online by Cambridge University Press

inside the term $M[x^V := N|_V]$ is based on V instead of the type of N. This is called 'early binding' in object-oriented programming.

This is in contrast to the $\lambda\&$-calculus, in which $(\lambda x^V.M)\ N$ is reduced using the usual β rule and therefore the type of N is used in the subsequent branch selections. This just corresponds to late-binding in object-oriented programming. This is accomplished by allowing the type of a term to change to a subtype by reductions, and by restricting the reduction of an overloaded application so that it is reduced only when the argument is in normal form. Thus, each term has two notions of types: the static type, which is given by type checking; and the dynamic type on which branch selection is based. Though late-binding is one of the characteristics of object-orientedness, it complicates both the formation and the semantics of the calculus. Since the aim of this paper is to investigate the semantic structure behind the circularity caused by overloading and subtyping, which occur also in calculi with early binding, we treat the simpler case.

3. The calculus $\lambda_\otimes$

3.1. *The type system of* $\lambda_\otimes$

In our language, we only consider basic types and (overloaded) function types. Suppose that a countable set of basic types (ranged over by B) is given. To give concrete examples, we assume that `Int` and `Bool` are in this set. We first define pretypes (ranged over by U and V) and function precomponents (ranged over by H) as follows:

$$\begin{aligned} V &::= B \mid [\overbrace{H, \ldots, H}^{\text{at least one}}], \\ H &::= V \to V\ . \end{aligned}$$

We call pretypes of the form $[H, \ldots, H]$ function pretypes. In our language, not all lists of function components are allowed as function types. For example, when V and V' have a common subtype, $[V \to \texttt{Int}, V' \to \texttt{Bool}]$ is not allowed because both branches are activated on an argument whose type is a subtype of both V and V', and thus we need to merge the results. Therefore, we define the following compatibility relation $\uparrow$ on pretypes and function precomponents, and define types by imposing a compatibility condition on the components of a function type:

$$\text{(M-REFL)}\quad \frac{}{B \uparrow B} \qquad \text{(M-FUN)}\quad \frac{(\text{not } V_1 \uparrow V_2) \text{ or } (V_1 \uparrow V_2 \text{ and } V_1' \uparrow V_2')}{V_1 \to V_1' \uparrow V_2 \to V_2'}$$

$$\text{(M-FLIST)}\quad \frac{H_i \uparrow H_j' \quad (i = 1, \ldots, n,\ j = 1, \ldots, m)}{[H_1, \ldots, H_n] \uparrow [H_1', \ldots, H_m']}$$

This definition is by structural induction on type expression, and thus correct in spite of the negative premise in rule (M-FUN).

The meaning of compatibility is that U and V are compatible iff $M : U$ and $N : V$ are mergeable. As we will see in Proposition 3.1 below, $U \uparrow V$ also means that U and V have a common subtype. Thus, the precondition of (M-FUN), which can also be restated

https://doi.org/10.1017/S0960129598002539 Published online by Cambridge University Press

as '$V_1 \uparrow V_2$ implies $V_1' \uparrow V_2'$', means that if V_1 and V_2 have a common subtype (and thus both components may be activated on some argument), then elements of V_1' and V_2' must be mergeable.

Now we define types:

— a pretype B is a type,
— if V and V' are types, then $V \to V'$ is a function component,
— if $H_1,\ldots,H_n$ are function components and are pairwise compatible (that is, $H_i \uparrow H_j$ for $i, j = 1,\ldots,n$), then $[H_1,\ldots,H_n]$ is a type.

Hereafter, we redefine that U and V range over types and H ranges over function components. We also define that F and G range over function types. We use TExp to denote the set of all types.

When $V_1 \uparrow V_2$, we define $V_1 \bowtie V_2$, the merge type of V_1 and V_2, as an abbreviation for a type as follows:

$$B \bowtie B = B,$$
$$[H_1,\ldots,H_m] \bowtie [H_{m+1},\ldots,H_n] = [H_1,\ldots,H_n].$$

Next, we define subtype relation on TExp as follows:

$$\text{(I-REFL)}\ \frac{}{V \leqslant V} \qquad \text{(I-FLIST1)}\ \frac{}{[H_1,\ldots,H_n] \leqslant [H_i]}$$

$$\text{(I-TRANS)}\ \frac{V \leqslant V' \quad V' \leqslant V''}{V \leqslant V''} \qquad \text{(I-FLIST2)}\ \frac{V_1 \uparrow V_2}{[V_1 \to V_1', V_2 \to V_2'] \leqslant [V_1 \bowtie V_2 \to V_1' \bowtie V_2']}$$

$$\text{(I-FUN)}\ \frac{V \leqslant U \quad U' \leqslant V'}{[U \to U'] \leqslant [V \to V']} \qquad \text{(I-FLIST3)}\ \frac{[H_1,\ldots,H_n] \leqslant [H_i'] \quad (i = 1,\ldots,m)}{[H_1,\ldots,H_n] \leqslant [H_1',\ldots,H_m']}$$

Note that a basic type is only comparable with itself, and we do not consider a subtype relation between basic types like `Int` $\leqslant$ `Real`. We will explain the reason for this restriction in Section 7.

We call V a subtype of U when $V \leqslant U$, and we say that V and U are equivalent and write $V \simeq U$ when $V \leqslant U$ and $U \leqslant V$. Though the same domain is assigned to equivalent types in the semantics, we will treat them as distinct syntactic objects of the calculus.

Now, $\uparrow$ and $\bowtie$ are characterized on the pre-ordered set $(\mathsf{TExp}, \leqslant)$ by the following proposition.

Proposition 3.1.

1 $V_1 \uparrow V_2$ iff V_1 and V_2 have a lower bound (*i.e.*, a common subtype).
2 $V_1 \bowtie V_2$ is the greatest lower bound of V_1 and V_2 with this preorder.

The proof is given in Appendix A.

https://doi.org/10.1017/S0960129598002539 Published online by Cambridge University Press

Definition 3.2. Let F be $[V_1 \to V_1', \ldots, V_n \to V_n']$.

1 We say that F is applicable to U if at least one of the V_i $(i = 1, \ldots, n)$ satisfies $U \leqslant V_i$.
2 When F is applicable to U, define $cod(F, U)$ as $V'_{\phi(1)} \bowtie \ldots \bowtie V'_{\phi(l)}$, where $\phi(i)$ $(i = 1, \ldots, l)$ satisfy $U \leqslant V_{\phi(i)}$.
3 When F is applicable to U, define $dom(F, U)$ as $V_{\phi(1)} \bowtie \ldots \bowtie V_{\phi(l)}$, where $\phi(i)$ $(i = 1, \ldots, l)$ satisfy $U \leqslant V_{\phi(i)}$.

From a function type F, we can form a partial function from TExp to itself that maps a type V to $cod(F, V)$ when F is applicable to V. This is actually the function that maps the argument type to the result type. The order $F \leqslant G$ on function types corresponds to the pointwise order on this function space. That is, $F \leqslant G$ iff $cod(F, V) \leqslant cod(G, V)$ for all V to which F is applicable. We shall explain the detail in Section 4.1.

We show the decidability of this order relation, which is not trivial because there exists the rule (I-TRANS), which is not structural.

Definition 3.3. For a function type F, $expand(F)$ is the set of all function components of the form $dom(F, V) \to cod(F, V)$ for $V \in \mathsf{TExp}$.

For example, when $U \uparrow V$, $U \not\leqslant V$, and $V \not\leqslant U$, $expand([U \to U', V \to V']) = \{U \to U', V \to V', U \bowtie V \to U' \bowtie V'\}$. Note that $expand(F)$ is a finite set for a function type F.

Theorem 3.4. $F \leqslant G$ iff for every $U \to U' \in expand(G)$, there exists $V \to V' \in expand(F)$ such that $V' \leqslant U'$ and $U \leqslant V$. In particular, $F \leqslant G$ is decidable.

The proof is given in Appendix B. Note that our type structure is essentially the same as a variant of an intersection type system $\mathbf{T}_{\#}$ in Cardone and Coppo (1990).

3.2. *The terms of* $\lambda_{\otimes}$

The preterms (ranged over by M and N) are defined as follows:

$$M \quad ::= \quad K^B \mid x^V \mid \lambda x^V.\, M \mid M\ M \mid M|_V \mid M \otimes M.$$

Here, K^B means a constant of a basic type B and x^V means a variable of type V. We assume that each basic type has at least one constant. $M|_V$ is the coercion of M to V and $M_1 \otimes M_2$ is the merge expression of M_1 and M_2.

We define the typing rules of $\lambda_{\otimes}$ as follows:

$$\text{(T-CONST)}\ \frac{}{K^B : B} \qquad \text{(T-APP)}\ \frac{M : F \quad N : V \quad F \text{ is applicable to } V}{M\ N : cod(F, V)}$$

$$\text{(T-VAR)}\ \frac{}{x^V : V} \qquad \text{(T-COE)}\ \frac{M : U \quad U \leqslant V}{M|_V : V}$$

$$\text{(T-FUN)}\ \frac{M : V'}{\lambda x^V.\, M : [V \to V']} \qquad \text{(T-MERGE)}\ \frac{M_1 : V_1 \quad M_2 : V_2 \quad V_1 \uparrow V_2}{M_1 \otimes M_2 : V_1 \bowtie V_2}$$

We call a preterm M a term when we can infer $M : V$ for some type V. Hereafter, we redefine that M and N range over terms. Recall that $V_1 \bowtie V_2$ is an abbreviation for a type.

https://doi.org/10.1017/S0960129598002539 Published online by Cambridge University Press

Therefore, all types have terms of the form $M_1 \otimes M_2$. As examples of terms, $1 \otimes 1$ and $1 \otimes 2$ are terms of type $\mathtt{Int} \bowtie \mathtt{Int} = \mathtt{Int}$, and $\lambda y^{[\mathtt{Int}\to\mathtt{Int},\mathtt{Bool}\to\mathtt{Bool}]}.(\lambda x^{\mathtt{Int}}.y\ (y\ x) \otimes \lambda x^{\mathtt{Bool}}.y\ (y\ x))$ is a term of type $[[\mathtt{Int} \to \mathtt{Int}, \mathtt{Bool} \to \mathtt{Bool}] \to [\mathtt{Int} \to \mathtt{Int}, \mathtt{Bool} \to \mathtt{Bool}]]$. Note also that (T-APP) includes the simple case

$$\text{(T-APP}') \ \frac{M : [V \to V'] \quad N : U \quad U \leqslant V}{M\ N : V'}.$$

It is clear that $\lambda_\otimes$ has the unicity of types property.

We define the reduction rules of $\lambda_\otimes$ as follows:

(E-APP) $(\lambda x^V.\ M)\ N \ \triangleright\ M[x^V := N|_V]$

(E-APPL) $(M_1 \otimes M_2)\ N \ \triangleright$
$$\left\{\begin{array}{l} M_1\ N \quad (F_1 \text{ is applicable to } V \text{ and } F_2 \text{ is not }) \\ M_2\ N \quad (F_2 \text{ is applicable to } V \text{ and } F_1 \text{ is not}) \\ M_1\ N \otimes M_2\ N \text{ (both } F_1 \text{ and } F_2 \text{ are applicable to } V) \end{array}\right\}$$
$(M_1 : F_1, M_2 : F_2 \text{ and } N : V)$

(E-CONST) $M \otimes N \ \triangleright\ N$ (M and N have the same basic type)

(E-COE) $M|_B \ \triangleright\ M$ (B is a basic type)

(E-APPC) $(M|_F)\ N \ \triangleright\ (M\ N)|_{cod(F,V)}$ (V is the type of N).

Note that the typing rule (T-APP) ensures that either M_1 or M_2 is applicable in (E-APPL). We have already explained the intuitive meanings of (E-APP), (E-APPL), and (E-CONST). The reduction rules (E-COE) and (E-APPC) determine the meaning of $M|_V$. Since a basic type is only comparable with itself, M also has the type B in (E-COE) and the coercion is from B to B. (E-COE) means that we define this coercion function to be the identity function. For a coercion of a function, (E-APPC) means that $M|_F$ has the same functionality as M, but the domain is restricted to $\{V \mid F \text{ is applicable to } V\}$ and the type of the return value is adjusted to $cod(F, V)$ by coercion.

It is easy to show the Subject Reduction property of $\lambda_\otimes$. The Church–Rosser Property is proved in the standard way, for example, by using the notion of parallel reduction (Takahashi 1989).

3.3. *Examples*

First, we show that records can be expressed in $\lambda_\otimes$ as overloaded functions. Let $\mathtt{L}_1, \ldots, \mathtt{L}_n, \ldots$ be the set of slot names used in our language. We use basic types $L_1, \ldots, L_n, \ldots$ and constants $l_1^{L_1}, \ldots, l_n^{L_n}, \ldots$ corresponding to $\mathtt{L}_1, \ldots, \mathtt{L}_n, \ldots$ for this encoding. A record type $\{\mathtt{L}_1 : V_1, \ldots, \mathtt{L}_n : V_n\}$ is expressed as $[L_1 \to V_1, \ldots, L_n \to V_n]$, a record value $\{\mathtt{L}_1 = M_1, \ldots, \mathtt{L}_n = M_n\}$ is expressed as $\lambda x^{L_1}.M_1 \otimes \ldots \otimes \lambda x^{L_n}.M_n$, and a record selection $M.\mathtt{L}_i$ is expressed as an application $M\ l_i^{L_i}$. With this encoding, we have $\{\mathtt{L}_1 = M_1, \ldots, \mathtt{L}_n = M_n\}.\mathtt{L}_i \ \triangleright^+\ M_i$. This is the same way records are encoded in $\lambda\&$-calculus (Castagna *et al.* 1995).

https://doi.org/10.1017/S0960129598002539 Published online by Cambridge University Press

Now, we give a simple example to show how the computational mechanism of $\lambda_{\otimes}$ works.

Let `[Window]` and `[FWindow]` be the following record types:

```
[Window] = {x:Int, y:Int,      ; x, y position
            w:Int, h:Int},     ; width and height
[FWindow] = {x:Int, y:Int,      ; x, y position
             w:Int, h:Int,      ; width and height
             f:Int}.            ; the width of the frame of the window
```

Suppose that the move function on `[Window]` is defined as follows:

$$\texttt{move1} = \lambda r^{\texttt{[Window]}}.\lambda dx^{\texttt{Int}}.\lambda dy^{\texttt{Int}}.r \otimes \{\texttt{x} = r.\texttt{x} + dx, \texttt{y} = r.\texttt{y} + dy\}.$$

Here, we assume that our language is extended with $+ : [\texttt{Int} \to \texttt{Int}]$ and reduction rules for $+$. Though the type of the return value is not `[Window]`, it is equivalent to `[Window]`. Therefore, the return value can be treated as if it is a `[Window]` value (see the next subsection). Consider that one tries to write the move function on `[FWindow]` such that it does not change the frame size and enlarges the height of the window by 1. One only needs to write the difference from the behaviour of `move1` to a `[FWindow]`-value as

$$\texttt{move2} = \lambda r^{\texttt{[FWindow]}}.\lambda dx^{\texttt{Int}}.\lambda dy^{\texttt{Int}}.\{\texttt{h} = r.\texttt{h} + 1, \texttt{f} = r.\texttt{f}\},$$

and define `move` as `move1` $\otimes$ `move2`. In this way, in a subtype method definition, one can extend the result of the supertype methods (in this example, the `f` slot), and override some part (in this example, the `h` slot). Tsuiki (1996) gives more examples of the use of the merge operator based on related calculi.

3.4. *Non-normalizing terms in* $\lambda_{\otimes}$

Though the fundamental idea of a calculus with overloading and subtyping looks very simple, its behaviour is rather complicated in that every type has a non-normalizing term, though types are defined inductively.

In $\lambda_{\otimes}$, an overloaded function is treated as a first class object that can be passed to another overloaded function as an argument. Moreover, a type $V = [V_1 \to V_1', \ldots, V_n \to V_n']$ may be a subtype of one of V_i $(i = 1, \ldots, n)$, and thus an overloaded function may be applicable to itself. This causes non-normalization in $\lambda_{\otimes}$.

As an example, we construct a non-normalizing term of type `Int`. First, note that $(\lambda x^{\texttt{S}}.x\ x)$ is well-typed for $\texttt{S} = [\texttt{Int} \to \texttt{Int},\ [\texttt{Int} \to \texttt{Int}] \to \texttt{Int}]$; `S` is applicable to `S` because `S` is a subtype of $[\texttt{Int} \to \texttt{Int}]$. Now, consider the type

$$\texttt{T} = \begin{bmatrix} \texttt{Int}\to\texttt{Int}, \\ [\texttt{Int} \to \texttt{Int}]\to\texttt{Int}, \\ \texttt{S}\to\texttt{Int} \end{bmatrix}.$$

`T` is a subtype of `S`. In particular, it is equivalent with `S` (see Proposition 4.1 below).

https://doi.org/10.1017/S0960129598002539 Published online by Cambridge University Press

Consider the term $M = (\lambda x^{\mathtt{Int}}.x) \otimes (\lambda x^{[\mathtt{Int}\to\mathtt{Int}]}.1) \otimes (\lambda x^{\mathtt{S}}.x\ x)$ of type $\mathtt{T}$. M is applicable to itself because $\mathtt{T} \leqslant \mathtt{S}$, and $M\ M$ has type $\mathtt{Int}$. It reduces infinitely as follows:

$$
\begin{aligned}
M\ M &= ((\lambda x^{\mathtt{Int}}.x) \otimes (\lambda x^{[\mathtt{Int}\to\mathtt{Int}]}.1) \otimes (\lambda x^{\mathtt{S}}.x\ x))\ M \\
&\rhd^{+} (\lambda x^{[\mathtt{Int}\to\mathtt{Int}]}.1)\ M \otimes (\lambda x^{\mathtt{S}}.x\ x)\ M \\
&\rhd (\lambda x^{\mathtt{S}}.x\ x)\ M \\
&\rhd M|_{\mathtt{S}}\ M|_{\mathtt{S}} \\
&\rhd (M\ M|_{\mathtt{S}})|_{\mathtt{Int}} \\
&\rhd^{+} ((\lambda x^{[\mathtt{Int}\to\mathtt{Int}]}.1)\ M|_{\mathtt{S}} \otimes (\lambda x^{\mathtt{S}}.x\ x)\ M|_{\mathtt{S}})|_{\mathtt{Int}} \\
&\rhd^{+} M|_{\mathtt{S}}|_{\mathtt{S}}\ M|_{\mathtt{S}}|_{\mathtt{S}} \\
&\ldots.
\end{aligned}
$$

Here, $\rhd^{+}$ means one or more reduction steps. Note that the same kind of non-normalizing term can be constructed in any type. Besides, we can define a term $Y : [[V \to V] \to V]$ for each V in a similar way for which $Y\ M$ is reduced to a term equivalent to $M\ (Y\ M)$ in the following sense.

Definition 3.5. (Plotkin 1977) Two terms M and N of equivalent types are (operationally) equivalent if in any closed context $C[]$ of a basic type, $C[M]$ reduces to a normal form K iff $C[N]$ reduces to K. We write $M \approx N$ when M and N are equivalent.

Note that we define equivalence not to terms of the same type but to terms of equivalent types. This definition is correct because when $U \simeq V$, $M : U$, and $N : V$, $C[M]$ is well-typed iff $C[N]$ is well-typed for any context $C[]$.

One thing to note is that though a term may be applicable to itself, it cannot return itself. Moreover, each function has a maximal number of arguments. This is shown by defining the degree of a type as follows:

$$
\begin{aligned}
\mathit{degree}(B) &= 1, \\
\mathit{degree}([V_1 \to V_1', \ldots, V_n \to V_n']) &= \mathit{max}(\mathit{degree}(V_1'), \ldots, \mathit{degree}(V_n')) + 1.
\end{aligned}
$$

The degree of the type of $M\ N$ is less than that of M and thus when the degree of the type of M is n, there are no terms $N_1, \ldots, N_n$ that make $M\ N_1\ N_2\ \ldots\ N_n$ well-typed. This fact means that we can consider a closed normal form of a basic type (which is a constant of $\lambda_{\otimes}$) as a value, and a closed term of a function type as a function returning a value on termination when enough number of arguments are given. This is one explanation why we can use the equivalence based on basic type contexts.

Non-normalization itself is not a desirable property when we consider the calculus as a practical programming language. Actually, practical studies (Castagna *et al.* 1995; Tsuiki 1996) use fragments of overloaded calculi which are normalizing. However, non-normalization poses a question on the mathematical structure of the calculus. That is, though types are defined inductively, we cannot give their semantics in a compositional manner, and we need to construct domains that reflect the circular structure of the calculus. The following sections are devoted to this construction.

https://doi.org/10.1017/S0960129598002539 Published online by Cambridge University Press

4. Construction of the semantic functor

4.1. *Semantics of types*

We first investigate the structure of the set of types, in particular, the poset of types modulo equivalence. We use $(\mathbf{T}, \leqslant)$ to denote the poset of all type expressions modulo equivalence extended with the top element $\top$. $\top$ can be considered a special type with only one element denoting type-error, to which every term can be coerced. We use $\overline{V}$ to denote the equivalence class that V belongs to. We use $\mathbf{T_B}$ and $\mathbf{T_F}$ to denote the subsets of $\mathbf{T}$ corresponding to basic types and function types, respectively, extended with $\top$. Thus the poset $\mathbf{T}$ satisfies $\mathbf{T} = \mathbf{T_B} + \mathbf{T_F}$, with $+$ the sum of two posets with the top elements identified. We use variables s and t for elements of $\mathbf{T}$.

$\mathbf{T_B}$ is a flat co-cpo since there is no ordering between basic types. We will investigate the structure $\mathbf{T_F}$ has. From a function type $F = [V_1 \to V_1', \ldots, V_n \to V_n']$, we can construct a monotonic function $\mathscr{G}(F)$ from $\mathbf{T}$ to $\mathbf{T}$ that maps an argument type to the result type. That is, $\mathscr{G}(F)(\overline{U}) = \overline{cod(F, U)}$ when F is applicable to U, and $\mathscr{G}(F)(\overline{U}) = \top$ when F is not. It is well-defined, because $cod(F, U) \simeq cod(F, V)$ when $U \simeq V$. Note that it is opposite to the standard treatment of a partial function where the least element is assigned when the value is undefined. Actually, the construction in this subsection will fit the usual domain theory if we change the direction of the subtype ordering. We use the prefix co- for notions that hold on the opposite order.

When $s, t \in \mathbf{T}$ satisfy $s, t \neq \top$, we define a co-step function $\mathsf{step}_{s,t}$ from $\mathbf{T}$ to $\mathbf{T}$ as follows:

$$\begin{aligned} \mathsf{step}_{s,t}(u) &= t \ (\text{if } u \leqslant s), \\ \mathsf{step}_{s,t}(u) &= \top \ (\text{otherwise}). \end{aligned}$$

Further, we define a co-finite function to be a function expressed as the greatest lower bound of a (possibly empty) finite set of co-step functions. Let $[\mathbf{T} \xrightarrow{\text{Cofin}} \mathbf{T}]$ be the set of co-finite functions from $\mathbf{T}$ to $\mathbf{T}$. Then, it is obvious that $\mathscr{G}([U \to V]) = \mathsf{step}_{\overline{U},\overline{V}}$, and $\mathscr{G}([U_1 \to U_1', \ldots, U_n \to U_n'])$ is equal to the greatest lower bound of $\mathsf{step}_{\overline{U_1},\overline{U_1'}}, \ldots, \mathsf{step}_{\overline{U_n},\overline{U_n'}}$. Thus, $\mathscr{G}$ is a function to $[\mathbf{T} \xrightarrow{\text{Cofin}} \mathbf{T}]$. On the other hand, every element of $[\mathbf{T} \xrightarrow{\text{Cofin}} \mathbf{T}]$ except the top element can be expressed in this form. This shows that $\mathscr{G}$ is an onto-function from $\mathsf{TExp} + \{\text{the top element}\}$ to $[\mathbf{T} \xrightarrow{\text{Cofin}} \mathbf{T}]$. Moreover, the following proposition holds.

Proposition 4.1. For two function types F and G, $F \leqslant G$ iff $\mathscr{G}(F) \leqslant \mathscr{G}(G)$ by the pointwise order on $[\mathbf{T} \xrightarrow{\text{Cofin}} \mathbf{T}]$. In particular, $F \simeq G$ iff $\mathscr{G}(F) = \mathscr{G}(G)$.

Proof. $\Rightarrow$: We can check that each subtyping rule satisfies this.
$\Leftarrow$: We have $\mathscr{G}(F)(s) \leqslant \mathscr{G}(G)(s)$ for every $s \in \mathbf{T}$ by definition. Assigning $s = \overline{dom(G, V)}$, we obtain $cod(F, dom(G, V)) \leqslant cod(G, dom(G, V))$. The right-hand side is equal to $cod(G, V)$. Thus, for each function component $dom(G, V) \to cod(G, V) \in expand(G)$, there is a function component $dom(F, dom(G, V)) \to cod(F, dom(G, V)) \in expand(F)$ such that $cod(F, dom(G, V)) \leqslant cod(G, V)$, and, from the definition of dom, we have $dom(F, dom(G, V)) \geqslant dom(G, V)$. Hence, by Theorem 3.4, we obtain $F \leqslant G$. □

Thus, from now on, we view $\mathscr{G}$ as a function from $\mathbf{T_F}$ to $[\mathbf{T} \xrightarrow{\text{Cofin}} \mathbf{T}]$. This proposition says that $\mathbf{T_F}$ is isomorphic to $[\mathbf{T} \xrightarrow{\text{Cofin}} \mathbf{T}]$ through $\mathscr{G}$. Thus, the poset $\mathbf{T}$ satisfies the following

https://doi.org/10.1017/S0960129598002539 Published online by Cambridge University Press

isomorphism:

$$\mathbf{T} \xleftrightarrow{\quad} \mathbf{T_B} + [\mathbf{T} \xrightarrow{\text{Cofin}} \mathbf{T}]. \tag{2}$$

Proposition 4.2. *degree*$(U) \geqslant$ *degree*(V) when $U \leqslant V$. In particular, *degree*$(U) =$ *degree*(V) when $U \simeq V$.

Proof. The proof is by structural induction on the proof of $U \leqslant V$. □

Therefore, by putting *degree*$(\top) = 0$ and *degree*$(\overline{V}) =$ *degree*(V), we redefine *degree* on **T**. We also redefine $\bowtie$ on **T**.

4.2. *An intuitive description of the model*

Before going on to the mathematical construction, we give the conceptual model of this calculus.

Since subject reduction holds, we consider a domain $D(V)$ for each type V so that $M : V$ denotes an element of $D(V)$. As a consequence of unicity of types, we can assume all the domains $\{D(V) \mid V \in \mathsf{TExp}\}$ to be disjoint. Each $D(V)$ must have some pointed order structure so that it reflects the normalization order. For example, $D(\mathtt{Int})$ is the flat cpo of integers in which a non-normalizing term is interpreted as $\perp_{\mathtt{Int}}$.

When $V \leqslant V'$, there needs to be a coercion function $\mathbf{coerce}_{V,V'}$ from $D(V)$ to $D(V')$ so that a term $M|_{V'}$ is interpreted as an element of $D(V')$ when $M : V$. The $\mathbf{coerce}_{V,V'}$ are expected to be transitive (that is, $\mathbf{coerce}_{V',V''} \cdot \mathbf{coerce}_{V,V'} = \mathbf{coerce}_{V,V''}$ when $V \leqslant V' \leqslant V''$) and reflexive (that is, $\mathbf{coerce}_{V,V}$ is the identity on V), and thus $D(V)$ and $D(U)$ are expected to be isomorphic when $V \simeq U$. However, these are only expected properties at this point; we have not proved $M|_{V'}|_{V'} \approx M|_{V'}$ when $M : V$ and $V \leqslant V' \leqslant V''$, nor do we have the equivalence of M and $M|_V$ when $M : V$. Though these equivalences can be proved by purely syntactic techniques, the proofs are a bit complicated. In this paper, we prove them differently. We first construct a model in which the $\mathbf{coerce}_{V,V'}$ are transitive and reflexive, and then prove that this model is computationally adequate. We then obtain these equivalences as corollaries. In the same way, we can also expect $M \approx M|_V$ when $U \simeq V$ and $M : U$. Therefore, we shall assign the same semantic domain to equivalent types, and prove these equivalences through the model.

For these reasons, we consider the semantic function D as a functor from **T** to **C**, which assigns a domain $D(t)$ to $t \in \mathbf{T}$, and assigns a coercion function $\mathbf{coerce}_{s,t}$ to every $s \leqslant t$. Here, **C** is a suitable category whose objects are pointed ordered sets. For the following construction, we also consider the conditions that **C** be a cartesian closed category with infinite products.

One desirable property of a coercion function $\mathbf{coerce}_{s,t}$ is that it is strict, *i.e.*, it preserves the bottom element. Therefore, we construct D as a functor to $\mathbf{C}_!$, which is a subcategory of **C** with strict morphisms only. The choice to work in the category $\mathbf{Fun}(\mathbf{T}, \mathbf{C}_!)$ instead of $\mathbf{Fun}(\mathbf{T}, \mathbf{C})$ is essential for the construction in Section 4.4.

Since $\mathbf{T} = \mathbf{T_B} + \mathbf{T_F}$, we study the structures of $D(t)$ ($t \in \mathbf{T}$) for the following three cases.

Case $t = \top$. As we have said, $\top$ is a special type extended to denote type error. We define $D(\top)$ as the terminal object **1** of **C**.

https://doi.org/10.1017/S0960129598002539 Published online by Cambridge University Press

Case $t \in \mathbf{T_B}, t \neq \top$. We define $D(t)$ as a given object corresponding to t, on which we can interpret constants of type t. For example, we can define $D(\overline{\texttt{Int}})$ to be the flat cpo of integers and $D(\overline{\texttt{Bool}})$ the flat cpo of boolean values.

Case $t \in \mathbf{T_F}, t \neq \top$. One might expect that an overloaded function type is a finite product of component function types. That is, $D(\overline{F})$ is $D(\overline{V'_1})^{D(\overline{V_1})} \times \ldots \times D(\overline{V'_n})^{D(\overline{V_n})}$ when $F = [V_1 \to V'_1, \ldots, V_n \to V'_n]$. However, this would depend on the choice of the representative type F, and is false. As an example, consider $U \leqslant V$, $F = [V \to \texttt{Int}]$, $G = [V \to \texttt{Int}, U \to \texttt{Int}]$, $M = (\lambda x^V.1) \otimes (\lambda x^U.2)$, and $N : U$. In this case, we have $F \simeq G$ and $M|_F\, N \triangleright (M\, N)|_{\texttt{Int}} \triangleright^+ 2$, whereas $M|_F\, N|_V \triangleright 1$. In this way, $M|_F : [V \to \texttt{Int}]$ behaves just like M, which cannot be expressed as a function from $D(\overline{V})$ to $D(\overline{\texttt{Int}})$.

Each function M of type F maps a type V to a type in $\mathcal{G}(\overline{F})(\overline{V})$. Therefore, one of the natural interpretations of M is a collection of monomorphic functions from $D(s)$ to $D(\mathcal{G}(t)(s))$ for all $s \in \mathbf{T}$. Therefore, we set

$$D(t) = \Pi_{s \in \mathbf{T}} D(\mathcal{G}(t)(s))^{D(s)}.$$

Note that when $\mathcal{G}(t)(s)$ is $\top$, we have $D(\mathcal{G}(t)(s))$ is $\mathbf{1}$, and therefore, $D(\mathcal{G}(t)(s))^{D(s)}$ is also isomorphic to $\mathbf{1}$. This means that this infinite product is isomorphic to the product over $\{s \in \mathbf{T} | \mathcal{G}(t)(s) \text{ is not } \top\}$. As for coercions, since $(M|_G)\ N \triangleright (M\ N)|_{cod(G,V)}$ when $N : V$, the branch of $M|_G$ to V is that of M with the result coerced from $cod(F, V)$ to $cod(G, V)$. Therefore, we define coercions pointwise. When $t \leqslant t'$ with $t \in \mathbf{T_F}$, we have $\mathcal{G}(t)(s) \leqslant \mathcal{G}(t')(s)$, and thus there is a morphism from $D(\mathcal{G}(t)(s))$ to $D(\mathcal{G}(t')(s))$. This induces a morphism from $D(\mathcal{G}(t)(s))^{D(s)}$ to $D(\mathcal{G}(t')(s))^{D(s)}$ for all $s \in \mathbf{T}$, and thus we have a morphism from $D(t)$ to $D(t')$, which we take as $\mathbf{coerce}_{t,t'}$.

Note that s also ranges over t itself in this product. Therefore, we cannot use this as the definition of D. Rather, it is a condition on D so that we can make this kind of interpretation.

4.3. *A formalization in* $\mathbf{Fun}(\mathbf{T}, \mathbf{C}_!)$.

We formalize the above idea in $\mathbf{Fun}(\mathbf{T}, \mathbf{C}_!)$.

Notation When $e_s \in D(s)$ for all $s \in \mathbf{T}$, we write $\langle s \in \mathbf{T}.e_s \rangle$ for the product in $\Pi_{s \in \mathbf{T}} D(s)$, and π_s for the projection from $\Pi_{s \in \mathbf{T}} D(s)$ to $D(s)$. To simplify the presentation, we use informal lambda notation for an arrow in $\mathbf{C}$ and for an element of an exponential object of $\mathbf{C}$.

For basic types, we define $D_{\mathbf{B}} \in \mathbf{Fun}(\mathbf{T}, \mathbf{C}_!)$ as follows:

$$\begin{array}{lcll} D_{\mathbf{B}}(t) & = & \text{a given object corresponding to } t & (t \in \mathbf{T_B}, t \neq \top), \\ D_{\mathbf{B}}(t) & = & \mathbf{1} & (t \in \mathbf{T_F}). \end{array}$$

Next, we define $[D \Rightarrow D']$, an overloaded function space from D to D'. When $D, D' \in \mathbf{Fun}(\mathbf{T}, \mathbf{C}_!)$, we define $[D \Rightarrow D'] \in \mathbf{Fun}(\mathbf{T}, \mathbf{C}_!)$, which operates on objects as follows:

$$\begin{array}{lcll} [D \Rightarrow D'](t) & = & \mathbf{1} & (t \in \mathbf{T_B}), \\ [D \Rightarrow D'](t) & = & \Pi_{s \in \mathbf{T}} D'(\mathcal{G}(t)(s))^{D(s)} & (t \in \mathbf{T_F}, t \neq \top), \end{array}$$

https://doi.org/10.1017/S0960129598002539 Published online by Cambridge University Press

and on morphisms as follows:

$$\begin{array}{lcll} [D \Rightarrow D'](t \leqslant t') & = & id_{\mathbf{1}} & (t, t' \in \mathbf{T_B}), \\ [D \Rightarrow D'](t \leqslant t') & = & \lambda f : \Pi_{s \in \mathbf{T}} D'(\mathscr{G}(t)(s))^{D(s)}. & \\ & & \langle s \in \mathbf{T}. D'(\mathscr{G}(t)(s) \leqslant \mathscr{G}(t')(s)) \circ \pi_s(f) \rangle & (t, t' \in \mathbf{T_F}). \end{array}$$

In particular, $[_ \Rightarrow _]$ is a functor from $\mathbf{Fun}(\mathbf{T}, \mathbf{C}_!)^{OP} \times \mathbf{Fun}(\mathbf{T}, \mathbf{C}_!)$ to $\mathbf{Fun}(\mathbf{T}, \mathbf{C}_!)$. It operates on natural transformations in the following way. Suppose that $\mu : D_2 \to D_1$, $\nu : D_1' \to D_2'$ are natural transformations. A natural transformation $[\mu \Rightarrow \nu] : [D_1 \Rightarrow D_1'] \to [D_2 \Rightarrow D_2']$ is defined as follows:

$$\begin{array}{ll} [\mu \Rightarrow \nu]_t = id_{\mathbf{1}} & (t \in \mathbf{T_B}), \\ [\mu \Rightarrow \nu]_t = \lambda f : \Pi_{s \in \mathbf{T}} D_1'(\mathscr{G}(t)(s))^{D_1(s)}. \langle s \in \mathbf{T}. \nu_{\mathscr{G}(t)(s)} \circ \pi_s(f) \circ \mu_s \rangle & (t \in \mathbf{T_F}, t \neq \top). \end{array}$$

We also define a product of functors $D \times D'$ as

$$(D \times D')(t) = D(t) \times D'(t).$$

$_ \times _$ is a functor from $\mathbf{Fun}(\mathbf{T}, \mathbf{C}_!) \times \mathbf{Fun}(\mathbf{T}, \mathbf{C}_!)$ to $\mathbf{Fun}(\mathbf{T}, \mathbf{C}_!)$.

Using these functors, the condition on D in the previous section can be stated as a natural isomorphism between functors (*i.e.*, an isomorphism in the category $\mathbf{Fun}(\mathbf{T}, \mathbf{C}_!)$) as follows:

$$D \underset{\Psi}{\overset{\Phi}{\rightleftarrows}} D_{\mathbf{B}} \times [D \Rightarrow D]. \tag{3}$$

Note that for each t, only one of the two components of $\times$ is not $\mathbf{1}$. Therefore, this product is used to select the appropriate component.

4.4. *The conditions for the equation to be solvable*

In this subsection, we give the condition under which Equation (3) is solvable. Smyth and Plotkin (1982) gives general conditions under which recursive domain equations are solvable by using the notion of order-enriched categories. Applying their results to the functor category $\mathbf{Fun}(\mathbf{T}, \mathbf{C}_!)$, we give conditions on $\mathbf{C}$.

Definition 4.3. A category is an **O**-*category* if and only if

(i) every hom-set is a poset in which every ascending ω-sequence has a l.u.b.
(ii) composition of morphisms is an ω-continuous operation with respect to this partial order.

Definition 4.4. Let $\mathbf{K}$ be an **O**-category and let $f : A \to B, g : B \to A$ be morphisms such that $g \circ f = id_A$ and $f \circ g \preceq id_B$. 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 $\mathbf{K}$ that has the same objects as $\mathbf{K}$ and that has only embeddings as morphisms.

Definition 4.5. Let **K, L, M** be **O**-categories. A functor $T : \mathbf{K}^{OP} \times \mathbf{L} \to \mathbf{M}$ is a locally continuous functor if and only if it is ω-continuous on the hom-sets; that is, if $f_n : A \to$

https://doi.org/10.1017/S0960129598002539 Published online by Cambridge University Press

B $(n = 1, 2, \ldots)$ is an increasing ω-sequence in $\mathbf{K}^{OP}$ and $g_n : C \to D$ $(n = 1, 2, \ldots)$ is one in $\mathbf{L}$, then $T(\sqcup_n f_n, \sqcup_n g_n) = \sqcup_n T(f_n, g_n)$.

By collecting the conditions on a category so that an equation is solvable, we give the following definition of $\mathbf{O}_\perp$-categories.

Definition 4.6. An $\mathbf{O}$-category is an $\mathbf{O}_\perp$-*category* if

(i) it has a terminal object,
(ii) every hom-set $hom(A, B)$ has a least element, $\perp_{A,B}$,
(iii) composition is left-strict in the sense that for any $f : A \to B$ we have $\perp_{B,C} \circ f = \perp_{A,C}$, and
(iv) every ω^{OP}-chain has a limit.

Then the results of Smyth and Plotkin (1982) can be stated by the following theorem.

Theorem 4.7. When $\mathbf{K}$ is an $\mathbf{O}_\perp$-category and $T : \mathbf{K}^{OP} \times \mathbf{K} \to \mathbf{K}$ is a locally continuous functor, there exists a minimal invariant of T. That is, there exists an object A together with an isomorphism $\Phi : A \to T(A, A)$ such that the l.u.b. of the morphisms $p_n : A \to A$ defined by $p_0 = \perp_{A,A}$, $p_{n+1} = \Phi^{-1} \circ T(p_n, p_n) \circ \Phi$ is the identity on A.

Proof. See Theorem 1, 2, 3 and Lemma 1, 2 of Smyth and Plotkin (1982). □

Using this, we consider conditions on $\mathbf{C}$ so that Equation (3) is solvable in $\mathbf{Fun}(\mathbf{T}, \mathbf{C}_!)$. A key fact is that limits in functor categories are computed pointwise. That is, when every ω^{op}-chain has a limit in $\mathbf{K}$, the same is true in $\mathbf{Fun}(\mathbf{L}, \mathbf{K})$ for any category $\mathbf{L}$, and when D is the limit of an ω^{op}-chain $\Delta = (D^i, \mu^i)$ in $\mathbf{Fun}(\mathbf{L}, \mathbf{K})$, $D(C)$ is the limit of the ω^{op}-chain $(D^i(C), \mu^i_C)$ in $\mathbf{K}$ for each object C of $\mathbf{L}$. Another key fact is that when $\mathbf{K}$ is an $\mathbf{O}$-category, $\mathbf{Fun}(\mathbf{L}, \mathbf{K})$ is also an $\mathbf{O}$-category with respect to the pointwise ordering on natural transformations, and the l.u.b. of a hom-set $\mathbf{Nat}(D, D')$ is computed pointwise. That is, when μ is the l.u.b. of an ascending ω-sequence $\langle \mu^0, \mu^1, \ldots \rangle$ in $\mathbf{Nat}(D, D')$, we have that μ_C is the l.u.b. of $\langle \mu^0_C, \mu^1_C, \ldots \rangle$ in $Hom(D(C), D'(C))$.

First, we redefine precisely the notion of strictness for the case that $\mathbf{C}$ is an $\mathbf{O}_\perp$-category.

Definition 4.8. Let $\mathbf{C}$ be an $\mathbf{O}_\perp$-category. We call a morphism $f : B \to C$ strict if we have $f \circ \perp_{A,B} = \perp_{A,C}$ for all A.

Proposition 4.9. If $\mathbf{C}$ is an $\mathbf{O}_\perp$-category, then $\mathbf{Fun}(\mathbf{T}, \mathbf{C}_!)$ is also an $\mathbf{O}_\perp$-category.

Proof. First, note that $\sqcup_i f_i$ is strict when f_i is. It means that $\mathbf{C}_!$ is also an $\mathbf{O}$-category, and therefore, so is $\mathbf{Fun}(\mathbf{T}, \mathbf{C}_!)$.

Since a terminal arrow is strict, $\mathbf{C}_!$ has a terminal object, and so does $\mathbf{Fun}(\mathbf{T}, \mathbf{C}_!)$. As for Condition (ii) of Definition 4.6, the least element of $\mathbf{Nat}(D, D')$ is the natural transformation that assigns $\perp_{D(s),D'(s)}$ to $s \in \mathbf{T}$. Composition is obviously left strict in $\mathbf{Fun}(\mathbf{T}, \mathbf{C}_!)$. For (iv), first we prove that if $(A, g_i : A \to A_i)$ is the limit of an ω^{op}-chain $\Delta = (A_i, f_i : A_{i+1} \to A_i)$ $(i = 1, \ldots)$ with f_i strict, then g_i are also strict. We show that $g_i \circ \perp_{C,A} = \perp_{C,A_i}$ for every C. Since f_i are strict, $(C, \perp_{C,A_i})$ is a cone. Therefore, there is a morphism $h : C \to A$ so that $g_i \circ h = \perp_{C,A_i}$. Since $\perp_{C,A} \preceq h$, we have $g_i \circ \perp_{C,A} \preceq \perp_{C,A_i}$. Therefore, $g_i \circ \perp_{C,A} = \perp_{C,A_i}$. From this, every ω^{OP}-chain has a limit in $\mathbf{C}_!$. Since limits in functor categories can be computed pointwise, an ω^{op}-chain in $\mathbf{Fun}(\mathbf{T}, \mathbf{C}_!)$ also has a limit. □

https://doi.org/10.1017/S0960129598002539 Published online by Cambridge University Press

Note that $\mathbf{Fun}(\mathbf{T}, \mathbf{C})$ fails to be an $\mathbf{O}_\perp$-category even when $\mathbf{C}$ is a $\mathbf{O}_\perp$-category because there may not be the bottom element of $\mathbf{Nat}(D, D')$. This is why we work in the category $\mathbf{Fun}(\mathbf{T}, \mathbf{C}_!)$ instead of $\mathbf{Fun}(\mathbf{T}, \mathbf{C})$.

Now, we move on to functors.

Proposition 4.10. Suppose that $\mathbf{C}$ is an $\mathbf{O}$-category that is cartesian closed and that has infinite products. Suppose also that the product preserves the order structure in that when $f, g : A \to \Pi_i.B_i$, $f \preceq g$ iff $\pi_i \circ f \preceq \pi_i \circ g$ for all i. Suppose also that the isomorphism between hom-sets $A \to C^B$ and $A \times B \to C$ preserves the order structure. Then,

1 The functor $[_ \Rightarrow _] : \mathbf{Fun}(\mathbf{T}, \mathbf{C}_!)^{OP} \times \mathbf{Fun}(\mathbf{T}, \mathbf{C}_!) \to \mathbf{Fun}(\mathbf{T}, \mathbf{C}_!)$ is locally continuous.
2 The functor $\times : \mathbf{Fun}(\mathbf{T}, \mathbf{C}_!) \times \mathbf{Fun}(\mathbf{T}, \mathbf{C}_!) \to \mathbf{Fun}(\mathbf{T}, \mathbf{C}_!)$ is locally continuous.

Proof. (1) First, note that $\mathbf{Fun}(\mathbf{T}, \mathbf{C}_!)$ is also an $\mathbf{O}$-category. From the assumption and the second condition of the definition of an $\mathbf{O}$-category, we can show that the exponential functor from $\mathbf{C}^{OP} \times \mathbf{C}$ to $\mathbf{C}$ is also locally continuous. Since the l.u.b. of a hom-set $\mathbf{Nat}(D, D')$ is computed pointwise, and products preserve the order structure, we have the result.
(2) This is easy. □

Thus, using Theorem 4.7 and Propositions 4.9 and 4.10, we can solve Equation (3) in many categories. For example, we can solve this in the category of pointed cpo's and continuous functions. In the following sections, we shall assume $\mathbf{C}$ to be the category of pointed cpo's and continuous functions.

5. A semantics of $\lambda_\otimes$

Let D be a functor from $\mathbf{T}$ to $\mathbf{C}$ satisfying Equation (3). We give a denotational semantics of $\lambda_\otimes$ in which a closed term M of type V is interpreted as an element of $D(\overline{V})$.

In order to simplify the presentation, we hereafter identify $D(t)$ and $D_{\mathbf{B}}(t)$ when $t \in \mathbf{T_B}$, and identify $D(t)$ and $[D \Rightarrow D](t)$ when $t \in \mathbf{T_F}$. We define $*$ as the only element of $\mathbf{1}$, and $\mathbf{coerce}_{s,t}$ as the morphism $D(s \leqslant t)$. We define two morphisms $\mathbf{expand}_{t,t'}$ and $\mathbf{merge}_{t_1,t_2}$, which will be used in giving the semantics. The morphism $\mathbf{expand}_{t,t'} : D(t')^{D(t)} \to D(\mathscr{G}^{-1}(\mathsf{step}_{t,t'}))$ enables us to view a monomorphic function from $D(t)$ to $D(t')$ as a function applicable to subtypes of $D(t)$ through coercions. We define

$$\mathbf{expand}_{t,t'}(f) = \langle s \in \mathbf{T}.\ \text{if } (s \leqslant t) \quad \text{then } \lambda d : D(s).f(\mathbf{coerce}_{s,t}(d)) \quad \text{else } \lambda d : D(s).* \rangle.$$

The morphism $\mathbf{merge}_{t_1,t_2} : D(t_1) \times D(t_2) \to D(t_1 \bowtie t_2)$ is defined only when t_1 and t_2 are compatible, and it corresponds to the merge operator $\otimes$. First, when $t_1, t_2 \in \mathbf{T_B}$ and $t_2 \neq \top$, we have $t_2 = t_1 \bowtie t_2$. In this case, we define $\mathbf{merge}_{t_1,t_2}(d_1, d_2) = d_2$. Next, when $t_1 \in \mathbf{T_B}$ and $t_2 = \top$, we have $t_1 = t_1 \bowtie t_2$. In this case, we define $\mathbf{merge}_{t_1,t_2}(d_1, d_2) = d_1$. Finally, when $t_1, t_2 \in \mathbf{T_F}$, $D(t)$ is isomorphic to $\Pi_{s \in \mathbf{T}} D(\mathscr{G}(t)(s))^{D(s)}$ for $t = t_1, t_2$, and $t_1 \bowtie t_2$. Moreover, since $\mathscr{G}(t_1 \bowtie t_2)(s) = \mathscr{G}(t_1)(s) \bowtie \mathscr{G}(t_2)(s)$, $D(t_1 \bowtie t_2)$ is isomorphic to $\Pi_{s \in \mathbf{T}} D(\mathscr{G}(t_1)(s) \bowtie \mathscr{G}(t_2)(s))^{D(s)}$. Thus, in this case, we define

$$\mathbf{merge}_{t_1,t_2}(f, g) = \langle s \in \mathbf{T}.\ \lambda d : D(s).\ \mathbf{merge}_{\mathscr{G}(t_1)(s),\mathscr{G}(t_2)(s)}(\pi_s(f)(d), \pi_s(g)(d)) \rangle\ .$$

https://doi.org/10.1017/S0960129598002539 Published online by Cambridge University Press

This definition needs care as **merge** also appears on the right-hand side of the definition and one may think of it as a circular definition. However, since the degree of $\mathscr{G}(t)(s)$ is smaller than that of t, $\mathbf{merge}_{t_1,t_2}$ is defined inductively on $max(degree(t_1), degree(t_2))$ and this definition is correct.

Let M be an expression of type V whose free variables are in the set $\vec{x} = \{x_1^{V_1}, \ldots, x_n^{V_n}\}$. An $\vec{x}$-environment ρ is a function that assigns to each variable $x_i^{V_i}$ in $\vec{x}$ a value $\rho(x_i^{V_i})$ in $D(\overline{V_i})$. The semantics $\mathscr{E}_{\vec{x}}[\![M]\!]$ of M is a function that assigns to each $\vec{x}$-environment ρ a value $\mathscr{E}_{\vec{x}}[\![M]\!]\rho \in D(\overline{V})$. Suppose that the semantics $\mathscr{E}_{\mathbf{B}}[\![K^B]\!] \in D(\overline{B})$ of basic type constants is given.

$\mathscr{E}_{\vec{x}}[\![M]\!]\rho$ is defined as follows:

$$
\begin{aligned}
&\mathscr{E}_{\vec{x}}[\![x_i^{V_i}]\!]\rho && = \rho(x_i^{V_i}),\\
&\mathscr{E}_{\vec{x}}[\![K^B]\!]\rho && = \mathscr{E}_{\mathbf{B}}[\![K^B]\!],\\
&\mathscr{E}_{\vec{x}}[\![\lambda x^V.M^U]\!]\rho && = \mathbf{expand}_{\overline{V},\overline{U}}(\lambda d : D(\overline{V}).\mathscr{E}_{\vec{x}\cup\{x^V\}}[\![M]\!]\rho[x^V := d]),\\
&\mathscr{E}_{\vec{x}}[\![M^U \otimes N^V]\!]\rho && = \mathbf{merge}_{\overline{U},\overline{V}}(\mathscr{E}_{\vec{x}}[\![M]\!]\rho, \mathscr{E}_{\vec{x}}[\![N]\!]\rho),\\
&\mathscr{E}_{\vec{x}}[\![M^F\ N^V]\!]\rho && = \pi_{\overline{V}}(\mathscr{E}_{\vec{x}}[\![M]\!]\rho)(\mathscr{E}_{\vec{x}}[\![N]\!]\rho),\\
&\mathscr{E}_{\vec{x}}[\![M^U|_V]\!]\rho && = \mathbf{coerce}_{\overline{U},\overline{V}}(\mathscr{E}_{\vec{x}}[\![M]\!]\rho).
\end{aligned}
$$

Here, we put the type of each subterm as a superscript. For an $\vec{x}$-environment ρ, $\rho[x^V := d]$ is an $\vec{x}\cup\{x^V\}$-environment that assigns d to x^V. When M is closed, we simply write $\mathscr{E}_{\vec{x}}[\![M]\!]$ for $\mathscr{E}_{\vec{x}}[\![M]\!]\rho$ because it does not depend on ρ.

Theorem 5.1. (Soundness) $\mathscr{E}_{\vec{x}}[\![M]\!]\rho = \mathscr{E}_{\vec{x}}[\![M']\!]\rho$ when $M \triangleright M'$.

Proof. We will only show the case M is reduced to M' by (E-APP). That is, $M = (\lambda x^V.\ M_1^U)\ N^{V'}$ and $M' = M_1[x^V := N|_V]$.

$$
\begin{aligned}
&\mathscr{E}_{\vec{x}}[\![(\lambda x^V.\ M_1^U)\ N^{V'}]\!]\rho\\
&\quad = \pi_{\overline{V'}}(\mathbf{expand}_{\overline{V},\overline{U}}(\lambda d : D(\overline{V}).\mathscr{E}_{\vec{x}\cup\{x^V\}}[\![M_1]\!]\rho[x^V := d]))(\mathscr{E}_{\vec{x}}[\![N]\!]\rho)\\
&\quad = (\lambda e : D(\overline{V'}).(\lambda d : D(\overline{V}).\mathscr{E}_{\vec{x}\cup\{x^V\}}[\![M_1]\!]\rho[x^V := d])(\mathbf{coerce}_{\overline{V'},\overline{V}}(e)))(\mathscr{E}_{\vec{x}}[\![N]\!]\rho)\\
&\quad = (\lambda e : D(\overline{V'}).\mathscr{E}_{\vec{x}\cup\{x^V\}}[\![M_1]\!]\rho[x^V := \mathbf{coerce}_{\overline{V'},\overline{V}}(e)])(\mathscr{E}_{\vec{x}}[\![N]\!]\rho)\\
&\quad = \mathscr{E}_{\vec{x}\cup\{x^V\}}[\![M_1]\!]\rho[x^V := \mathbf{coerce}_{\overline{V'},\overline{V}}(\mathscr{E}_{\vec{x}}[\![N]\!]\rho)]\\
&\quad = \mathscr{E}_{\vec{x}}[\![M_1[x^V := N|_V]]\!]\rho.
\end{aligned}
$$

The other cases are shown similarly. For the cases of (E-COE) and (E-APPC), note that $\mathbf{coerce}_{s,t} = D(s \leqslant t)$ is identified with $(D_{\mathbf{B}} \times [D \Rightarrow D])(s \leqslant t)$ through Equation (3). It is also identified with the identity on $D(s)$ when $s = t \in \mathbf{T_B}$, and with $[D \Rightarrow D](s \leqslant t)$ when $s, t \in \mathbf{T_F}$. □

6. Computational adequacy and syntactic properties

6.1. *Computational adequacy*

Now we prove the adequacy theorem. Computational adequacy means that when M is a closed term of a basic type B, $\mathscr{E}[\![M]\!]$ is the least element of $D(\overline{B})$ iff M does not have a normal form.

The language PCF, which has the Y operator as a primitive, also has this kind of property. Usually, this property is proved for a typed calculus like PCF by using the notion

https://doi.org/10.1017/S0960129598002539 Published online by Cambridge University Press

of computability or the formal approximation relations. However, these techniques cannot be applied to our language directly because computability and formal approximation relations are defined compositionally on the formation of types, whereas the corresponding notions need circular definitions in our calculus.

Recently, A. M. Pitts (Pitts 94) introduced a new way of proving adequacy for the case in which the domain is given as the minimal invariant of a domain constructor of mixed variance. It constructs the formal approximation relation as a fixed point of an operator over relations. We prove the adequacy of $\lambda_{\otimes}$ by following his construction.

Let $D \in \mathbf{Fun}(\mathbf{T}, \mathbf{C}_!)$ be the minimal invariant of Equation (3). That is, D satisfies Equation (3) and the l.u.b. of the natural transformations $p_n : D \to D$ defined by $p_0 = \bot_{D,D}$, $p_{n+1} = \Phi^{-1} \circ id_{\mathbf{B}} \times [p_n \Rightarrow p_n] \circ \Phi$ is the identity on D. Here, $id_{\mathbf{B}}$ is the identity natural transformation on $D_{\mathbf{B}}$. A functor constructed in Section 4.1 satisfies this.

We use $Prog(V)$ to denote the set of closed expressions of type V. We use meta variables P and Q for closed expressions.

Definition 6.1. We define $\mathscr{R}$ as the set of all TExp-indexed collections of binary relations $\{R_V \subseteq D(\overline{V}) \times Prog(V) \mid V \in \mathsf{TExp}\}$, satisfying the following:

1 $\bot_{\overline{V}} \; R_V \; P$ for all $V \in \mathsf{TExp}$ and $P \in Prog(V)$,
2 when $d_0 \sqsubseteq d_1 \ldots \sqsubseteq d_n \ldots$ is an increasing ω-sequence in $D(\overline{V})$ for which $d_i \; R_V \; P$ holds, then $(\sqcup_n d_n) \; R_V \; P$.

We define a formal approximation relation as follows:

Definition 6.2. $\lesssim \in \mathscr{R}$ is a formal approximation if the following hold:

1 $d \lesssim_B P$ iff $d = \bot_B$ or $P \rhd^+ K^B$ and $\mathscr{E}_{\mathbf{B}}[\![K^B]\!] = d$.
2 $d \lesssim_F P$ iff $d = \bot_F$ or $\pi_{\overline{V}}(d)(d') \lesssim_{cod(F,V)} P \; P'$ for all V applicable to F and for all (d', P') such that $d' \lesssim_V P'$.

Note that this defintion is simpler than the corresponding one in Pitts (1994). We first show that the adequacy property holds if there exists a formal approximation relation. In the following, we assume that $\lesssim$ is a formal approximation.

Lemma 6.3. When $P \rhd^+ P'$ with $P \in Prog(V)$, $d \lesssim_V P$ iff $d \lesssim_V P'$.

Proof. The proof is by induction on the degree of V. When V is a basic type, it is immediate because $P \rhd^+ K$ iff $P' \rhd^+ K$ from the Church–Rosser Property. Next, the case V is a function type. Suppose that d is not $\bot_V$. Let V' be a type applicable to V and let (e, Q) satisfy $e \lesssim_{V'} Q$. From the induction hypothesis, we can infer $\pi_{\overline{V'}}(d)(e) \lesssim_{cod(V,V')} P \; Q$ iff $\pi_{\overline{V'}}(d)(e) \lesssim_{cod(V,V')} P' \; Q$ because $P \; Q \rhd^+ P' \; Q$ and $cod(V, V')$ has a smaller degree than V has. Therefore, by definition, we have $d \lesssim_V P$ iff $d \lesssim_V P'$. □

Lemma 6.4. Suppose that F is applicable to V, $d \lesssim_F P$, and $e \lesssim_V Q$. Then we have $\pi_{\overline{V}}(d) \; (e) \lesssim_{cod(F,V)} P \; Q$.

Proof. The proof is immediate from Lemma 6.3. □

Lemma 6.5. $\mathbf{merge}_{s,\top}(e, *) = \mathbf{merge}_{\top,s}(*, e) = e$.

Proof. The proof is by induction on the degree of s. □

https://doi.org/10.1017/S0960129598002539 Published online by Cambridge University Press

Lemma 6.6. Suppose that $d \lesssim_V P$ and $d' \lesssim_{V'} P'$ with $V \uparrow V'$. Then we have $\mathbf{merge}_{\overline{V},\overline{V'}}(d,d') \lesssim_{V \bowtie V'} P \otimes P'$.

Proof. The proof is by induction on the degree of V. When V and V' are the same basic types, $\mathbf{merge}_{\overline{V},\overline{V'}}(d,d') = d'$ and $P \otimes P' \triangleright P'$. We have the result using Lemma 6.3. When V and V' are function types, we need to show that for all U applicable to V or V' and for all (e,Q) such that $e \lesssim_U Q$, $\pi_U(\mathbf{merge}_{\overline{V},\overline{V'}}(d,d'))(e) \lesssim_{cod(V \bowtie V',U)} (P \otimes P')\ Q$. We need to consider the three cases corresponding to the branches of the (E-APPL) rule. When U is applicable to both V and V', the left-hand side is equal to $\mathbf{merge}_{\overline{cod(V,U)},\overline{cod(V',U)}}(\pi_U(d)(e), \pi_U(d')(e))$ and the right-hand side is equal to $P\ Q \otimes P'\ Q$. The result holds by Lemma 6.4 and the induction hypothesis. The other cases are shown by Lemma 6.4 and Lemma 6.5. □

Lemma 6.7. Suppose that $d \lesssim_V P$ and $V \leqslant U$. Then $\mathbf{coerce}_{\overline{V},\overline{U}}(d) \lesssim_U P|_U$.

Proof. The proof is by induction on the degree of U. It is obvious when U is a basic type. When U is a function type, we need to show that for all V' applicable to U and for all (e,Q) such that $e \lesssim_{V'} Q$, $\pi_{V'}(\mathbf{coerce}_{\overline{V},\overline{U}}(d))(e) \lesssim_{cod(U,V')} P|_U\ Q$. From the definition of $[_ \Rightarrow _]$, we have $\pi_{V'}(\mathbf{coerce}_{\overline{V},\overline{U}}(d))(e) = \mathbf{coerce}_{\overline{cod(V,V')},\overline{cod(U,V')}}(\pi_{V'}(d)(e))$. On the other hand, $P|_U\ Q$ is reduced to $(P\ Q)|_{cod(U,V')}$. Since $\pi_{V'}(d)(e) \lesssim_{cod(V,V')} P\ Q$ from Lemma 6.4, we have the result using the induction hypothesis. □

Now we can relate the existence of a formal approximation relation with the computational adequacy property.

Proposition 6.8. Suppose that $\lesssim$ is a formal approximation and $M : V$ is a term with free variables $x_1^{V_1}, \ldots, x_n^{V_n}$, and $\rho(x_i^{V_i}) \lesssim_{V_i} P_i$ for $i = 1, \ldots, n$. Then $\mathscr{E}_{\vec{x}}[\![M]\!]\rho \lesssim_V M[x_1^{V_1} := P_1, \ldots, x_n^{V_n} := P_n]$.

Proof. The proof is by induction on the formation of the term M. The cases when M is x^V and K^B are obvious. The cases $M_1\ M_2$, $M_1 \otimes M_2$, and $N|_F$ are direct consequences of Lemma 6.4, 6.6, and 6.7, respectively. Consider the case $M = \lambda x^V.N^U$. We need to show that $\pi_{\overline{V'}}(\mathscr{E}_{\vec{x}}[\![\lambda x^V.N^U]\!]\rho)(d) \lesssim_U N[x_1^{V_1} := P_1, \ldots, x_n^{V_n} := P_n, x^V := P|_V]$ for all $V' \leqslant V$ and $d \lesssim_{V'} P$. The left-hand side is equal to $\mathscr{E}_{\vec{x} \cup \{x^V\}}[\![N]\!]\rho[x^V := \mathbf{coerce}_{\overline{V'},\overline{V}}(d)]$, and since $\mathbf{coerce}_{\overline{V'},\overline{V}}(d) \lesssim_V P|_V$ by Lemma 6.7, we can use the induction hypothesis to prove the result. □

From this proposition, if M is a closed term of a basic type B, we have $\mathscr{E}[\![M]\!] \lesssim_B M$. Therefore, by definition, $\mathscr{E}[\![M]\!] = \bot_B$ or M reduces to a normal form. Thus, we have the adequacy theorem.

Proposition 6.9. If there exists a formal approximation relation, the computational adequacy property holds. That is, let M be a closed $\lambda_\otimes$-term of a basic type B. If M does not have a normal form, $\mathscr{E}[\![M]\!]$ is $\bot_B$. In other words, $M \triangleright^+ K$ iff $\mathscr{E}[\![M]\!] = \mathscr{E}_{\mathbf{B}}[\![K]\!]$.

6.2. *Construction of a formal approximation relation*

Now, we prove the existence of a formal approximation relation. The construction of this subsection is parallel to the one in Pitts (1994) though the domains to work with are different; D is a pointed cpo in Pitts (1994), whereas it is a functor from $\mathbf{T}$ to $\mathbf{C}_!$ here. First, note that $\mathscr{R}$ is closed under intersections of each component relation, and thus

https://doi.org/10.1017/S0960129598002539 Published online by Cambridge University Press

$\mathscr{R}$ forms a lattice ordered by set inclusion on each component. Consider the monotone function $\phi : \mathscr{R}^{OP} \times \mathscr{R} \to \mathscr{R}$ defined as

$$\phi(R,S)_B = \{(d,P) \in D(\overline{B}) \times Prog(B) \mid d = \bot_B \text{ or } (P \triangleright K^B \text{ and } \mathscr{E}_{\mathbf{B}}[\![K^B]\!] = d)\},$$

$$\phi(R,S)_F = \left\{ (d,P) \in D(\overline{F}) \times Prog(F) \;\middle|\; \begin{array}{l} d = \bot_F \text{ or } \pi_{\overline{V}}(d)(d')S_{cod(F,V)}P\ P' \text{ for} \\ \text{all } V \text{ applicable to } F \text{ and for all} \\ (d',P') \text{ such that } d' R_V P'. \end{array} \right\}.$$

Here, $\mathscr{R}^{OP}$ is $\mathscr{R}$ with the opposite ordering. The condition of a formal approximation is rephrased as $R = \phi(R,R)$.

Now, consider the complete lattice $\mathscr{R}^{OP} \times \mathscr{R}$ and an operator

$$\phi^\sharp : \mathscr{R}^{OP} \times \mathscr{R} \to \mathscr{R}^{OP} \times \mathscr{R}$$

defined as

$$\phi^\sharp(R,S) = (\phi(S,R), \phi(R,S)).$$

Since $\phi^\sharp$ is monotonic, we can apply Tarski's fixed point theorem to construct $\lesssim^+$ and $\lesssim^-$ such that $\phi^\sharp(\lesssim^-, \lesssim^+) = (\lesssim^-, \lesssim^+)$. Since $(\lesssim^-, \lesssim^+)$ is defined to be the intersection of $\{(R,S) | (R,S) \supseteq \phi^\sharp(R,S)\}$, we have the following lemma and theorem.

Lemma 6.10.

1 $\lesssim^+ = \phi(\lesssim^-, \lesssim^+)$ and $\lesssim^- = \phi(\lesssim^+, \lesssim^-)$.
2 For all $R, S \in \mathscr{R}$, if $R \subseteq \phi(S,R)$ and $S \supseteq \phi(R,S)$, then $R \subseteq \lesssim^-$ and $S \supseteq \lesssim^+$.

Theorem 6.11. $\lesssim^-$ and $\lesssim^+$ are equal and thus form a formal approximation relation.

Proof. We mimic the proof of Theorem 6 in Pitts (1994). By Lemma 6.10, we have $\lesssim^- \supseteq \lesssim^+$. For $\lesssim^- \subseteq \lesssim^+$, we use the minimal invariant property of D. Consider the function $\delta : Nat(D,D) \to Nat(D,D)$ mapping μ to $\Phi^{-1} \circ id_{\mathbf{B}} \times [\mu \Rightarrow \mu] \circ \Phi$. We have $p_{n+1} = \delta(p_n)$. Since δ is continuous, we have the least fixed point of δ, which is the least upper bound of p_n $(n = 0, 1, \ldots)$. The minimal invariant property says that this fixed point is equal to id_D. Now, when $\mu \in Nat(D,D)$ and $R, S \in \mathscr{R}$, we write $\mu : R \subset S$ to mean that $d\ R_V\ P$ implies $\mu_{\overline{V}}(d)\ S_V\ P$ for all $V \in \mathsf{TExp}$, $P \in Prog(V)$, and $d \in D(\overline{V})$. We can show from the definition of δ that $\mu : R \subset S$ implies

$$\delta(\mu) : \phi(S,R) \subset \phi(R,S).$$

This means,using Lemma 6.10, that δ maps the set

$$\{\mu \in Nat(D,D) \mid \mu :\lesssim^- \subset \lesssim^+\}$$

to itself. Clearly, this subset of $Nat(D,D)$ contains the least element and is closed under the limits of ω-chains from the definition of $\mathscr{R}$. Therefore, by the Principle of Fixpoint Induction (see, for example, Davey and Priestley (1990)), the least fixed point of δ is also in this subset. Thus, $id_D :\lesssim^- \subset \lesssim^+$, which means $\lesssim^- \subseteq \lesssim^+$. □

Theorem 6.12. $\lambda_\otimes$ has the computational adequacy property with respect to the model given in Section 5.

https://doi.org/10.1017/S0960129598002539 Published online by Cambridge University Press

6.3. *Syntactic properties derived from the adequacy*

Using the adequacy theorem, we can show equivalences between terms.

Lemma 6.13. Let M and N be terms of type V. If $\mathscr{E}_{\vec{x}}[\![M]\!]\rho = \mathscr{E}_{\vec{x}}[\![N]\!]\rho$ for every $\vec{x}$-environment ρ, then $M \approx N$.

Proof. We need to show that for any closed context $C[]$ of a basic type, $C[M]$ reduces to a normal form K iff $C[N]$ reduces to K. From the adequacy theorem, this is equivalent to $\mathscr{E}[\![C[M]]\!] = \mathscr{E}[\![C[N]]\!]$ for every context $C[]$ of a basic type, which follows directly from $\mathscr{E}_{\vec{x}}[\![M]\!]\rho = \mathscr{E}_{\vec{x}}[\![N]\!]\rho$ for every ρ because the semantics of a term is defined following the structure of a term. □

Corollary 6.14. (Transitivity of coercions) When $W \leqslant V \leqslant U$ and $M : W$, $M|_V|_U \approx M|_U$.

Proof. We only need to show $\mathscr{E}_{\vec{x}}[\![M|_V|_U]\!]\rho = \mathscr{E}_{\vec{x}}[\![M|_U]\!]\rho$ for all $\vec{x}$-environment ρ. This is obvious, since $\mathbf{coerce}_{\overline{V},\overline{U}} \circ \mathbf{coerce}_{\overline{W},\overline{V}} = \mathbf{coerce}_{\overline{W},\overline{U}}$. □

Corollary 6.15. (Equivalence of equivalent types) When $U \simeq V$ and $M : U$, $M \approx M|_V$.

Proof. $\mathscr{E}_{\vec{x}}[\![M|_V]\!]\rho = \mathbf{coerce}_{\overline{U},\overline{V}}(\mathscr{E}_{\vec{x}}[\![M]\!]\rho) = \mathscr{E}_{\vec{x}}[\![M]\!]\rho$ because $\overline{U} = \overline{V}$ and $\mathbf{coerce}_{\overline{V},\overline{V}} = id_{\overline{V}}$. □

Corollary 6.16. (Associativity of $\otimes$) $(M_1 \otimes M_2) \otimes M_3 \approx M_1 \otimes (M_2 \otimes M_3)$.

Corollary 6.17. (Overloading) When $M : U$, $N : V$, and $U \geqslant V$, $M \otimes N \approx N$.

Proof. First, note that $U \bowtie V \simeq V$ from Proposition 3.1. Since $\mathscr{E}_{\vec{x}}[\![M \otimes N]\!]\rho = \mathbf{merge}_{\overline{U},\overline{V}}(\mathscr{E}_{\vec{x}}[\![M]\!]\rho, \mathscr{E}_{\vec{x}}[\![N]\!]\rho)$, we show $\mathbf{merge}_{s,t}(d,e) = e$ for $d \in D(s)$ and $e \in D(t)$ when $s \geqslant t$ by induction on the degree of s. First, when $degree(s) = 0$, we have $s = \top$ and the proof is immediate from Lemma 6.5. Next, consider the case $degree(s) = 1$. Then, s and t are the same basic type and the proof is obvious from the definition of **merge**. Finally, when $degree(s) > 1$, we have

$$\mathbf{merge}_{s,t}(d,e) = \langle u \in \mathbf{T}.\ \lambda x : u.\ \mathbf{merge}_{\mathscr{G}(s)(u),\mathscr{G}(t)(u)}(\pi_u(d)(x), \pi_u(e)(x)) \rangle\ .$$

Since $\mathscr{G}(s)(u) \geqslant \mathscr{G}(t)(u)$, the right-hand side is equal to $\langle u \in \mathbf{T}.\ \lambda x : u.\ \pi_u(e)(x)) \rangle$ by the induction hypothesis, and therefore equal to $\langle u \in \mathbf{T}.\ \pi_u(e) \rangle$ and e. □

This corollary means that we can simplify $M \otimes N$ to N in the type checking phase when the type of N is a subtype of the type of M.

Corollary 6.18. (Equivalence to reduction rule for overloading) Suppose that $F = [V_1 \to V_1', \ldots, V_n \to V_n']$ satisfies the following:

1 $V_i \not\leqslant V_j$ for $1 \leqslant i < j \leqslant n$,
2 if $V_j \leqslant V_i$, then $V_j' \leqslant V_i'$ $(i, j \leqslant n)$,
3 if V_i and V_j are compatible, then there is a unique h such that $V_i \bowtie V_j \simeq V_h$.

Suppose also that $M_i : [V_i \to V_i']$ for $i = 1, \ldots, n$ and $M = M_1 \otimes \ldots \otimes M_n$ is applicable to $N : U$. Then $M\ N$ is equivalent to $M_h\ N$ for h the unique index which satisfies $V_h \simeq dom(F, U)$.

Proof. First, the existence of the unique h follows from the definition of *dom* and Condition (3). The term $M\ N$ is reduced to $M_{\phi(1)}\ N \otimes \ldots \otimes M_{\phi(l)}\ N$, where $\phi(i)$ $(i = 1, \ldots, l)$ satisfy $U \leqslant V_{\phi(i)}$. From Conditions (3) and (1), $V_{\phi(l)} \leqslant V_{\phi(i)}$ $(i = 1, \ldots, l)$,

https://doi.org/10.1017/S0960129598002539 Published online by Cambridge University Press

and from Condition (2), $V'_{\phi(l)} \leqslant V'_{\phi(i)}$ $(i = 1, \ldots, l)$. Therefore, from Corollary 6.17, $M_{\phi(1)}\ N \otimes \ldots \otimes M_{\phi(l)}\ N$ is equivalent to $M_{\phi(l)}\ N$. □

In the above corollary, Condition (1) means that we must order the components by the subtype order of domains when composing an overloaded function. This is a convention we introduce to simulate object-orientedness in $\lambda_{\otimes}$. The second and third conditions are the ones imposed on an overloaded function in calculi with overloading (Castagna *et al.* 1995). This corollary means that if one writes a function so that it satisfies these conditions, the merge operator behaves like an overloaded function constructor that selects only one component when applied to an argument. In this sense, we can say that the notion of merge is an extension of overloading when we consider calculi with early binding.

7. Conclusion

We have developed a semantics of a calculus with overloading and subtyping by formalizing overloading as a categorical constructor over a functor category from the poset $(\mathbf{T}, \leqslant)$ of types with subtype relations. We have shown that the circular structure caused by the interaction between overloading and subtyping is expressed as a single equation in this functor category, and is solved by applying the Smyth–Plotkin method to this functor category. We have also presented the adequacy property by applying the technique developed by Pitts to this functorial setting, and derived some syntactic properties. Hence, these results are new applications of well established theories.

Even though this paper deals with a calculus that uses the merge operator as an overloaded function constructor, the same kind of semantic construction can be applied to a calculus with the usual overloading operator. For example, we can add Conditions (1), (2) and (3) of Corollary 6.18 to the conditions for the function types in Section 3.1 and replace the reduction rule (E-APPL) with the following:

$$\text{(E-OVLD)} \qquad (M_1 \otimes M_2)\ N \triangleright \left\{\begin{array}{l} M_2\ N \quad (F_2 \text{ is applicable to } V) \\ M_1\ N \quad (F_1 \text{ is applicable to } V \text{ and } F_1 \text{ is not}) \end{array}\right\} \quad (M_1 : F_1, M_2 : F_2 \text{ and } N : V).$$

Then, the resulting calculus, which we shall call $\lambda'_{\otimes}$, is a subcalculus of $\lambda_{\otimes}$in which $\otimes$ behaves as an overloading operator in that only one branch is selected in each function application. One can see that the non-normalizing terms in Section 3.4 belong to $\lambda'_{\otimes}$, and thus $\lambda'_{\otimes}$ has the same circular structure as $\lambda_{\otimes}$, and we can do an analogous study of this calculus.

One reason for not using this calculus is that it has some unexpected restrictions on the way overloaded functions are created. For example, suppose that $U \leqslant V_1$ and $U \leqslant V_2$ and we need to write an overloaded function with three components $M_0 : [U \to \texttt{Int}]$, $M_1 : [V_1 \to \texttt{Int}]$ and $M_2 : [V_2 \to \texttt{Int}]$. Then, the resulting function must be written as $M_1 \otimes (M_2 \otimes M_0)$ instead of $(M_1 \otimes M_2) \otimes M_0$ because $[V_1 \to \texttt{Int}, V_2 \to \texttt{Int}]$ is not allowed as a type of $\lambda'_{\otimes}$. This contradicts our intuition, which demands that an overloaded function is created by adding a branch defined on a subtype from the right. We need more elaborate syntax to avoid this.

In our semantics, an overloaded function type is interpreted as an infinite product of

https://doi.org/10.1017/S0960129598002539 Published online by Cambridge University Press

monomorphic function spaces. This kind of formalization was also used in Castagna *et al.* (1993) to construct a PER model of a stratified version of $\lambda\&$-early, which is a calculus with overloading and subtyping based on early binding method dispatch, and which uses the usual overloading operator. In this calculus, the type structure is restricted so that no self-dependency occurs and thus the semantics of a type is given in a compositional way. Our contribution is to formalize this interpretation as a categorical constructor over a functor category, and to study the circular structure caused by overloading and subtyping.

This interpretation of an overloaded function type includes many functions not presentable as a $\lambda_{\otimes}$-term, *i.e.*, we can only express in $\lambda_{\otimes}$ a function whose behaviour is determined on a finite number of types. The author does not regard this as a defect of the semantics, but rather thinks of it as suggesting a natural way of extending the calculus. For example, the identity polymorphic function that operates as the identity function on all types is not expressible in $\lambda_{\otimes}$, partly because the type of such a function is not a finite function from **T** to **T**, and partly because its behaviour is not determined on a finite number of types. We will discuss this later.

In this paper, we have tried to define $\lambda_{\otimes}$ as the minimal necessary one in order to concentrate on the circular structure it has. One may consider an extension by allowing a subtype relation between basic types like $\mathtt{Int} \leqslant \mathtt{Real}$. However, this kind of extension is difficult because if the definition of $\bowtie$ is extended so that $U \bowtie V$ denotes the g.l.b. of U and V, then $\mathtt{Int} \bowtie \mathtt{Real} = \mathtt{Int}$, whereas if we do not modify the reduction rule, then $1 \otimes 1.5$ is reduced to 1.5.

One direction of study is to extend $\lambda_{\otimes}$ with the operators of the language PCF like $+$ and *if* and corresponding reduction rules. Since the Y operator can be encoded in our calculus, we can embed PCF into this calculus, which does not have an explicit recursion operator. Many interesting problems arise, for example, the conservativeness of this language over PCF.

Another direction is to deal with late binding and make the calculus closer to practical object-oriented languages. In Tsuiki (1996), the author considered a restriction $\lambda_{R\otimes}$ of $\lambda_{\otimes}$ in which an overloaded function is only applicable to basic type values and record values, and considered an extension $\lambda^{\forall}_{R\otimes}$ of $\lambda_{R\otimes}$ with type variables. In $\lambda^{\forall}_{R\otimes}$, one can express late binding as well as parametric polymorphic functions like the identity polymorphic function. This calculus is still complicated, and the author expects that there is a more elegant way of combining *ad hoc* polymorphism and parametric polymorphism.

Appendix A. The proof of Proposition 1

Lemma A.1. $U \bowtie V \leqslant U, \quad U \bowtie V \leqslant V$.

Proof. The proof follows from (I-FLIST1) and (I-FLIST3). □

Lemma A.2. $U \leqslant V$ and $U \uparrow W$ implies $W \uparrow V$. In particular, $U \leqslant V$ implies $U \uparrow V$.

Proof. The proof is by induction on the proof of $U \leqslant V$. If the last rule is (I-REFL), (I-FLIST1), (I-FLIST2), or (I-FLIST3), then the proof is easy. For the case

https://doi.org/10.1017/S0960129598002539 Published online by Cambridge University Press

of (I-FUN), we assume that $[U_1 \to U_1'] \leqslant [V_1 \to V_1']$ is proved using (I-FUN) as the last rule and that $[U_1 \to U_1'] \uparrow [W_1 \to W_1', \ldots, W_n \to W_n']$, and prove that $[W_1 \to W_1', \ldots, W_n \to W_n'] \uparrow [V_1 \to V_1']$ holds. That is, we assume $U_1 \geqslant V_1$, $U_1' \leqslant V_1'$ and $U_1 \to U_1' \uparrow W_i \to W_i'$ $(i = 1, \ldots, n)$, and prove $W_i \to W_i' \uparrow V_1 \to V_1'$ $(i = 1, \ldots, n)$. Suppose that $W_i \uparrow V_1$. Then, by the induction hypothesis, we have $W_i \uparrow U_1$. Therefore, $W_i' \uparrow U_1'$ because $W_i \to W_i' \uparrow U_1 \to U_1'$. Again, by the induction hypothesis, we have $W_i' \uparrow V_1'$. Thus, $W_i \to W_i' \uparrow V_1 \to V_1'$ is proved.

For the case of (I-TRANS), we assume that $V \leqslant V''$ is proved from $V \leqslant V'$ and $V' \leqslant V''$, and assume that $W \uparrow V$. Then, by the induction hypothesis, we have $W \uparrow V'$, and again by the induction hypothesis, $W \uparrow V''$. □

Lemma A.3. $W \leqslant U$ and $W \leqslant V$ implies $U \uparrow V$ and $W \leqslant U \bowtie V$.

Proof. We have $W \uparrow U$ from Lemma A.2, and $U \uparrow V$ from Lemma A.2 again. $W \leqslant U \bowtie V$ is obvious when U, V, and W are the same basic types. When they are function types, let H be a function component of U. Then, $U \leqslant [H]$ from (I-FLIST1), and $W \leqslant [H]$ from (I-TRANS). In this way, all the components of $U \bowtie V$ are greater than W. Therefore, from (I-FLIST3), we have $W \leqslant U \bowtie V$. □

From Lemma A.1 and A.3, we have Proposition 1.

Proposition A.4. (Proposition 1)

1 $V_1 \uparrow V_2$ iff V_1 and V_2 have a lower bound.
2 $V_1 \bowtie V_2$ is the greatest lower bound of V_1 and V_2.

Appendix B. The proof of Theorem 2

First, we prove that $[U \to U'] \leqslant [V \to V']$ implies $V \leqslant U$ and $U' \leqslant V'$. To use induction, we prove a more general statement.

Lemma B.1. $[U \to U'] \leqslant [V_1 \to V_1', \ldots, V_n \to V_n']$ implies $U \geqslant V_i$ and $U' \leqslant V_i'$ $(i = 1, \ldots, n)$. As a special case, $[U \to U'] \leqslant [V \to V']$ implies $U \geqslant V$ and $U' \leqslant V'$.

Proof. We use $R(\Pi_1, \ldots, \Pi_n)$ to denote a proof with the last rule R and the proofs of the premises Π_i $(i = 1, \ldots, n)$. We prove by induction on the size of the minimum proof of $[U \to U'] \leqslant [V_1 \to V_1', \ldots, V_n \to V_n']$. If (I-TRANS)($\Pi_1$, (I-TRANS)($\Pi_2, \Pi_3$)) is a proof, then (I-TRANS)((I-TRANS)(Π_1, Π_2), Π_3) is also a proof of the same subtype relation, and it has the same number of inferences. Therefore, we may safely assume that when (I-TRANS) is used the right premise of (I-TRANS) is proved without using (I-TRANS) as the last rule.

The last step of the proof is (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, the induction hypothesis applies to the proof of $[U \to U'] \leqslant [V_i \to V_i']$, and thus we have $U \geqslant V_i$ and $U' \leqslant V_i'$.

Suppose that the last rule is (I-TRANS). Let Σ and Π be the proofs of $[U \to U'] \leqslant [W_1 \to W_1', \ldots, W_m \to W_m']$ and $[W_1 \to W_1', \ldots, W_m \to W_m'] \leqslant [V_1 \to V_1', \ldots, V_n \to V_n']$, respectively, and the present proof be (I-TRANS)(Σ, Π). By applying the induction hypothesis to Σ, we have $U \geqslant W_i$ $(i = 1, \ldots, m)$ and $U' \leqslant W_i'$ $(i = 1, \ldots, m)$. The last rule

https://doi.org/10.1017/S0960129598002539 Published online by Cambridge University Press

of Π is (I-REFL), (I-FUN), (I-FLIST1), (I-FLIST2) or (I-FLIST3). The cases (I-REFL) and (I-FLIST1) are trivial. Suppose that $m = n = 1$ and that the last rule of Π is (I-FUN). Then, by the induction hypothesis, we have $W_1 \geqslant V_1$ and $W_1' \leqslant V_1'$, and thus $U \geqslant V_1$ and $U' \leqslant V_1'$. Suppose that $m = 2$, $n = 1$, $V_1 = W_1 \bowtie W_2$, $V_1' = W_1' \bowtie W_2'$, and that the last rule of Π is (I-FLIST2). Since $U \geqslant W_1$, $U \geqslant W_2$, $U' \leqslant W_1'$, and $U' \leqslant W_2'$, we have $U \geqslant W_1 \bowtie W_2 = V_1$ and $U' \leqslant W_1' \bowtie W_2' = V_1'$. 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 $[W_1 \to W_1', \ldots, W_m \to W_m'] \leqslant [V_i \to V_i']$ $(i = 1, \ldots, n)$. Then, consider a proof (I-TRANS)(Σ, Π_i) of $[U \to U'] \leqslant [V_i \to V_i']$. Since it has fewer rules, the induction hypothesis applies. Thus, $U \geqslant V_i$ and $U' \leqslant V_i'$ are proved. □

Lemma B.2. If $\{V_i \mid i = 1, \ldots, n\}$ and $\{U_i \to U_i' \mid i = 1, \ldots, n\}$ are pairwise compatible sets and $[U_i \to U_i'] \leqslant [V_i \to V_i']$ $(i = 1, \ldots, n)$, then $\{U_i \mid i = 1, \ldots, n\}$, $\{U_i' \mid i = 1, \ldots, n\}$ and $\{V_i' \mid i = 1, \ldots, n\}$ are pairwise compatible and $[(U_1 \bowtie \ldots \bowtie U_n) \to (U_1' \bowtie \ldots \bowtie U_n')] \leqslant [(V_1 \bowtie \ldots \bowtie V_n) \to (V_1' \bowtie \ldots \bowtie V_n')]$.

Proof. We only prove the case $n = 2$. From Lemma B.1, we have $U_1 \geqslant V_1$ and $U_2 \geqslant V_2$. From this and $V_1 \uparrow V_2$, we have $U_1 \uparrow U_2$ and $U_1 \bowtie U_2 \geqslant V_1 \bowtie V_2$ by Proposition 3.1. On the other hand, $U_1 \uparrow U_2$ and $(U_1 \to U_1') \uparrow (U_2 \to U_2')$ implies $U_1' \uparrow U_2'$. Thus, $U_1' \bowtie U_2'$ exists and $U_1' \bowtie U_2' \leqslant V_1' \bowtie V_2'$ because $U_1' \leqslant V_1'$ and $U_2' \leqslant V_1'$ from Lemma B.1. Therefore, from (I-FUN), $[(U_1 \bowtie U_2) \to (U_1' \bowtie U_2')] \leqslant [(V_1 \bowtie V_2) \to (V_1' \bowtie V_2')]$ holds. □

Lemma B.3. If $\{V_i \mid i = 1, \ldots, n\}$ and $\{V_i \to V_i' \mid i = 1, \ldots, n\}$ are pairwise compatible sets, then $V_1', \ldots, V_n'$ are pairwise compatible and $[V_1 \to V_1', \ldots, V_n \to V_n'] \leqslant [(V_1 \bowtie \ldots \bowtie V_n) \to (V_1' \bowtie \ldots \bowtie V_n')]$.

Proof. The proof is by induction on n. From (I-FLIST2), it is true for $n = 2$. Suppose that it is true for $n - 1$. From (I-FLIST1) and (I-FLIST3), $[V_1 \to V_1', \ldots, V_n \to V_n'] \leqslant [V_1 \to V_1', \ldots, V_{n-1} \to V_{n-1}']$. From the induction hypothesis, $[V_1 \to V_1', \ldots, V_{n-1} \to V_{n-1}'] \leqslant [(V_1 \bowtie \ldots \bowtie V_{n-1}) \to (V_1' \bowtie \ldots \bowtie V_{n-1}')]$, and from (I-TRANS), $[V_1 \to V_1', \ldots, V_n \to V_n'] \leqslant [(V_1 \bowtie \ldots \bowtie V_{n-1}) \to (V_1' \bowtie \ldots \bowtie V_{n-1}')]$. On the other hand, $[V_1 \to V_1', \ldots, V_n \to V_n'] \leqslant [V_n \to V_n']$. Therefore, we have $[V_1 \to V_1', \ldots, V_n \to V_n'] \leqslant [(V_1 \bowtie \ldots \bowtie V_{n-1}) \to (V_1' \bowtie \ldots \bowtie V_{n-1}')] \bowtie [V_n \to V_n]$. Since $V_1, \ldots, V_n$ are pairwise compatible, $V_1 \bowtie \ldots \bowtie V_{n-1}$ and V_n are pairwise compatible. Thus, $V_1 \bowtie \ldots \bowtie V_n$ exists, and from (I-FLIST2), $[(V_1 \bowtie \ldots \bowtie V_{n-1}) \to (V_1' \bowtie \ldots \bowtie V_{n-1}')] \bowtie [V_n \to V_n] \leqslant [(V_1 \bowtie \ldots \bowtie V_n) \to (V_1' \bowtie \ldots \bowtie V_n')]$. From this and (I-TRANS), $[V_1 \to V_1', \ldots, V_n \to V_n'] \leqslant [(V_1 \bowtie \ldots \bowtie V_n) \to (V_1' \bowtie \ldots \bowtie V_n')]$. □

Proposition B.4. Suppose that $[U_1 \to U_1', \ldots, U_n \to U_n'] \leqslant [V_1 \to V_1', \ldots, V_m \to V_m']$. Then, for each i $(i = 1, \ldots, m)$, $[(U_{\phi(1)} \bowtie \ldots \bowtie U_{\phi(l)}) \to (U_{\phi(1)}' \bowtie \ldots \bowtie U_{\phi(l)}')] \leqslant [V_i \to V_i']$ for some $\phi(1)$ to $\phi(l)$.

Proof. The proof is by induction on the structure of the proof of $[U_1 \to U_1', \ldots, U_n \to U_n'] \leqslant [V_1 \to V_1', \ldots, V_m \to V_m']$. 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). Let Σ and Π be the proofs of $[U_1 \to U_1', \ldots, U_n \to U_n'] \leqslant [W_1 \to W_1', \ldots, W_k \to W_k']$ and $[W_1 \to W_1', \ldots, W_k \to W_k'] \leqslant [V_1 \to V_1', \ldots, V_m \to V_m']$, respectively, and that the proof is (I-TRANS)(Σ, Π). Fix an i $(i = 1, \ldots, m)$ and let $V = V_i, V' = V_i'$. From the induction hypothesis, $[(U_{\phi(j,1)} \bowtie \ldots \bowtie U_{\phi(j,l(j))}) \to (U_{\phi(j,1)}' \bowtie \ldots \bowtie U_{\phi(j,l(j))}')] \leqslant [W_j \to W_j']$

https://doi.org/10.1017/S0960129598002539 Published online by Cambridge University Press

for each j ($j = 1, \ldots, k$), and $[(W_{\psi(1)} \bowtie \ldots \bowtie W_{\psi(l_0)}) \to (W'_{\psi(1)} \bowtie \ldots \bowtie W'_{\psi(l_0)})] \leqslant [V \to V']$. Let X_j be $U_{\phi(j,1)} \bowtie \ldots \bowtie U_{\phi(j,l(j))}$ and X'_j be $U'_{\phi(j,1)} \bowtie \ldots \bowtie U'_{\phi(j,l(j))}$. In order to apply Lemma B.2, we must prove that $X_{\psi(j)} \to X'_{\psi(j)}$ ($j = 1, \ldots, l_0$) are pairwise compatible. $[X_j \to X'_j] \geqslant [U_{\phi(j,1)} \to U'_{\phi(j,1)}, \ldots, U_{\phi(j,l(j))} \to U'_{\phi(j,l(j))}] \geqslant [U_1 \to U'_1, \ldots, U_n \to U'_n]$ from Lemma B.3. Therefore, $[X_{\psi(j)} \to X'_{\psi(j)}]$ ($j = 1, \ldots, l_0$) have a common subtype and hence are compatible. Thus, Lemma B.2 applies, and we get $[(X_{\psi(1)} \bowtie \ldots \bowtie X_{\psi(l_0)}) \to (X'_{\psi(1)} \bowtie \ldots \bowtie X'_{\psi(l_0)})] \leqslant [(W_{\psi(1)} \bowtie \ldots \bowtie W_{\psi(l_0)}) \to (W'_{\psi(1)} \bowtie \ldots \bowtie W'_{\psi(l_0)})]$. From (I-TRANS), we have $[(X_{\psi(1)} \bowtie \ldots \bowtie X_{\psi(l_0)}) \to (X'_{\psi(1)} \bowtie \ldots \bowtie X'_{\psi(l_0)})] \leqslant [V \to V']$. The left-hand side can be reordered to $\bowtie$ of U_i's and we have thus proved the proposition. □

Lemma B.5.

1 $V \leqslant dom(F, V)$.
2 If $V \to V' \in expand(F)$, then $F \leqslant [V \to V']$.
3 If $U \leqslant V$ and F is applicable to V, then $dom(F, U) \leqslant dom(F, V)$ and $cod(F, U) \leqslant cod(F, V)$.
4 If $F \leqslant G$ and G is applicable to V, then $cod(F, V) \leqslant cod(G, V)$.

Proof. (1), (2) and (3) are trivial.
To prove (4), let F be $[U_1 \to U'_1, \ldots, U_n \to U'_n]$ and G be $[V_1 \to V'_1, \ldots, V_n \to V'_n]$. Suppose that $V \leqslant V_i$. From Proposition B.4, $[V_i \to V'_i] \geqslant [(U_{\phi(1)} \bowtie \ldots \bowtie U_{\phi(l)}) \to (U'_{\phi(1)} \bowtie \ldots \bowtie U'_{\phi(l)})]$ for some $\phi(1), \ldots, \phi(l)$. This means $U_{\phi(j)} \geqslant V_i \geqslant V$ ($j = 1, \ldots, l$). Thus, $cod(F, V) \leqslant U'_{\phi(j)}$ ($j = 1, \ldots, l$), and, therefore, we have $cod(F, V) \leqslant V'_i$. Since $cod(G, V)$ is the greatest lower bound of $\{V'_i \mid V \leqslant V_i\}$, we have $cod(F, V) \leqslant cod(G, V)$. □

Lemma B.6.

1 $cod(G, dom(G, V)) = cod(G, V)$.
2 If $F = G \bowtie G'$, then $dom(G, dom(F, V)) = dom(G, V)$ and $cod(G, dom(F, V)) = cod(G, V)$.

Proof. (2) Let G be $[V_1 \to V'_1, \ldots, V_n \to V'_n]$. We prove that $\{i \mid V_i \geqslant V\} = \{i \mid V_i \geqslant dom(F, V)\}$. Since $V \leqslant dom(F, V)$, we only need to show $V_i \geqslant V$ implies $V_i \geqslant dom(F, V)$. This is trivial because $dom(F, V)$ has the form $dom(F, V) = \ldots \bowtie V_i \bowtie \ldots$ when $V_i \geqslant V$. □

Note that Lemma B.6 (2) cannot be generalized to the case $F \leqslant G$.

Theorem B.7. (Theorem 3.4) $F \leqslant G$ if and only if $\forall V \to V' \in expand(G)$, $\exists U \to U' \in expand(F)$ such that $U \geqslant V$ and $U' \leqslant V'$. In particular, we can taken the function component $dom(F, V) \to cod(F, dom(F, V)) \in expand(F)$ as $U \to U'$.

Proof. (*If* part) Let $F = [U_1 \to U'_1, \ldots, U_n \to U'_n]$ and $G = [V_1 \to V'_1, \ldots, V_n \to V'_n]$. Since $dom(G, V_i) \to cod(G, V_i) \in expand(G)$, there exists, for each i, $W_i \to W'_i \in expand(F)$ such that $[W_i \to W'_i] \leqslant [dom(G, V_i) \to cod(G, V_i)]$. From Lemma B.5(2), we have $F \leqslant [W_i \to W'_i]$. Therefore, $F \leqslant [dom(G, V_i) \to cod(G, V_i)] \leqslant [V_i \to V'_i]$ because $dom(G, V_i) \simeq V_i$ and $cod(G, V_i) \leqslant V'_i$. Thus, we have $F \leqslant G$.

(*Only if* part) Each element of $expand(G)$ has the form $V_{\phi(1)} \bowtie \ldots \bowtie V_{\phi(l)} \to V'_{\phi(1)} \bowtie \ldots \bowtie V'_{\phi(l)}$. Therefore, we will prove that for all compatible subsets $\{V_{\phi(1)}, \ldots, V_{\phi(l)}\}$ of $\{V_1, \ldots, V_n\}$, there exists a function component $W \to W' \in expand(F)$ such that $[W \to W'] \leqslant [V_{\phi(1)} \bowtie \ldots \bowtie V_{\phi(l)} \to V'_{\phi(1)} \bowtie \ldots \bowtie V'_{\phi(l)}]$. From Proposition B.4, this is true if the right-hand side has the form $P = [V_i \to V'_i]$. Thus, we define $W_i \to W'_i$ ($i = 1, \ldots, n$)

https://doi.org/10.1017/S0960129598002539 Published online by Cambridge University Press

so that $[W_i \to W_i'] \leqslant [V_i \to V_i']$. $[W_i \to W_i']$ $(i = 1, \ldots, n)$ are compatible because they have a common subtype F. Therefore, we can apply Lemma B.2 and we have $[V_{\phi(1)} \bowtie \ldots \bowtie V_{\phi(l)} \to V'_{\phi(1)} \bowtie \ldots \bowtie V'_{\phi(l)}] \geqslant [W_{\phi(1)} \bowtie \ldots \bowtie W_{\phi(l)} \to W'_{\phi(1)} \bowtie \ldots \bowtie W'_{\phi(l)}]$. Note that the right-hand side is equivalent to a type of the form $[U_{\psi(1)} \bowtie \ldots \bowtie U_{\psi(l)} \to U'_{\psi(1)} \bowtie \ldots \bowtie U'_{\psi(l)}]$. This completes the proof because $dom(F, U_{\psi(1)} \bowtie \ldots \bowtie U_{\psi(l)}) \simeq U_{\psi(1)} \bowtie \ldots \bowtie U_{\psi(l)}$ and $cod(F, U_{\psi(1)} \bowtie \ldots \bowtie U_{\psi(l)}) \leqslant U'_{\psi(1)} \bowtie \ldots \bowtie U'_{\psi(l)}$.

(*In particular* part) Let $U \to U' \in expand(F)$ be the function component we composed in the previous paragraph. Then, U and U' satisfy $U \geqslant V$ and $U' \leqslant V'$. First, $dom(F, V) \geqslant V$ and $U \geqslant dom(F, V)$, since $dom(F, V)$ is the least type of the form $U_{\psi(1)} \bowtie \ldots \bowtie U_{\psi(l)}$ that is greater than V. From Lemma B.5(3), we have $U' = cod(F, U) \geqslant cod(F, dom(F, V))$. Therefore, we have $cod(F, dom(F, V)) \leqslant V'$. □

From Lemma B.6 and Theorem B.7, we can consider another set of rules for the subtype relation with two rules

$$\text{(I-FUN}') \qquad \frac{\text{for each } V \to V' \in expand(G),\ dom(F, V) \text{ exists and } cod(F, V) \leqslant V'}{F \leqslant G}$$

and (I-REFL′). Here, (I-REFL′) is the restriction of (I-REFL) to basic types. From Theorem B.7, it is easily verified that this system is equivalent to the original one. Though the definitions of $expand(F), dom(F, V)$ and $cod(F, V)$ depend on the subtype relation, we can show the correctness of this new subtype relation by considering an appropriate degree. This alternative definition also shows the decidability of our subtype relation.

Acknowledgements

The author would like to thank Andreas Knobel, Masami Hagiya, Giuseppe Castagna, Benjamin Pierce and Shinya Nishizaki for fruitful discussions on the subject and helpful comments on the paper. He also wishes to thank the referees for their invaluable comments and clarifications, in particular, on the proof of the adequacy theorem.

References

Cardone, F. and Coppo, M. (1990) Two extensions of Curry's type inference system. In *Logic and Computer Science*, Academic Press 19–66.

Castagna, G., Ghelli, G. and Longo, G. (1993) A semantics for λ&-early: a calculus with overloading and early binding. In: International Conference on Typed Lambda Calculi and Applications. *Springer-Verlag Lecture Notes in Computer Science* **664** 107–123. (A version of this work corrected and revised by the same authors appeared as Chapter 10 of Castagna, G. (1997) *Object-oriented Programming: A unified Foundation*, Birkhauser. An extended version is the report liens-94-22, Ecole Normale Supérieure.)

Castagna, G., Ghelli, G. and Longo, G. (1995) A Calculus for Overloaded Functions with Subtyping. *Information and Computation* **117** (1) 115–135.

Davey, B. A. and Priestley, H. A. (1990) *Introduction to Lattices and Order*, Cambridge University Press.

https://doi.org/10.1017/S0960129598002539 Published online by Cambridge University Press

Pitts, A. M. (1994) Computational Adequacy via 'Mixed' Inductive Definitions. In: Mathematical Foundations of Programming Semantics, Proc. 9th Int. Conf., New Orleans, LA, USA, April 1993. *Springer-Verlag Lecture Notes in Computer Science* **802** 72–82.

Pitts, A. M. (1996) Relational Properties of Domains. *Information and Computation* **127** 66–90.

Plotkin, G. (1977) LCF Considered as a Programming Language. *Theoretical Computer Science* **5** 223–255.

Smyth, M. B. and Plotkin, G. (1982) The Category-Theoretic Solution of Recursive Domain Equations. *SIAM J. Comput.* **11**(4) 761–783.

Takahashi, M. (1989) Parallel Reductions in λ-Calculus. *J. Symbolic Computation* **7** 113–123.

Tsuiki, H. (1992) *A Record Calculus with a Merge Operator*, Ph.D. thesis, Keio University.

Tsuiki, H. (1994) On typed calculi with a merge operator. In: 14th Conference on Foundations of Software Technology and Theoretical Computer Science. *Springer-Verlag Lecture Notes in Computer Science* **880** 101–112.

Tsuiki, H. (1996) Currying Multi Methods in a Merge Calculus. In: *Functional and Logic Programming: proceedings of the Second Fuji International Workshop,* World Scientific Publishing, Singapore 266–287.

https://doi.org/10.1017/S0960129598002539 Published online by Cambridge University Press