A Modality for Recursion *

Hiroshi Nakano
Ryukoku University, Seta, Otsu 520-2194, Japan
E-mail: nakano@math.ryukoku.ac.jp

Abstract

We propose a modal logic that enables us to handle self-referential formulae, including ones with negative self-references, which on one hand, would introduce a logical contradiction, namely Russell's paradox, in the conventional setting, while on the other hand, are necessary to capture a certain class of programs such as fixed point combinators and objects with so-called binary methods in object-oriented programming. Our logic provides a basis for axiomatic semantics of such a wider range of programs and a new framework for natural construction of recursive programs in the proofs-as-programs paradigm.

1. Introduction

Even though recursion, or self-reference, is an indispensable concept in both programs and their specifications, it is still far from obvious how to capture it in an axiomatic semantics such as the formulae-as-types notion of construction [17]. Only a rather restricted class of recursive programs (and specifications) has been captured in this direction as (co)inductive proofs over the (co)inductive data structures (see e.g., [9, 14, 23, 19, 26]), and, for example, negative self-references, which would be necessary to handle a certain range of programs such as fixed point combinators and objects with so-called binary methods in object-oriented programming, still remain out of the scope.

In this paper, we propose a modal logic that provides a basis for capturing such a wider range of programs in the proofs-as-programs paradigm. We give the logic as a modal typing system with recursive types for the purpose of presentation, and show its soundness with respect to a realizability interpretation which implies the convergence of well-typed programs according to their types.

Difficulty in binary-methods. Consider, for example, the specification $\mathbf{Nat}(n)$ of objects that represent a natural number n with a *method* which returns an object of $\mathbf{Nat}(n+m)$ when one of $\mathbf{Nat}(m)$ is given. It could be represented by a self-referential specification such as:

$$\mathbf{Nat}(n) \equiv ((n=0) + (n>0 \wedge \mathbf{Nat}(n-1)) \times (\forall m.\, \mathbf{Nat}(m) \to \mathbf{Nat}(n+m))),$$

where we assume that n and m range over the set of natural numbers; $+$, $\times$ and $\to$ are type constructors for direct sums, direct products and function spaces, respectively; $\wedge$ and $\forall$ have standard logical (annotative) meanings. Although it is not obvious whether this self-referential specification is meaningful in a certain mathematical sense, it could be a first approximation of the specification we want since this can be regarded as a refined version of recursive types which have been widely adopted as a basis for object-oriented type systems [1, 6]. At any rate, if we define an object $\mathbf{0}$ as:

$$\mathbf{0} \equiv <\mathbf{i_1}\, *, \lambda x.\, x>,$$

then it would satisfy $\mathbf{Nat}(0)$, where $\mathbf{i_1}$ is the injection into the first summand of direct sums and $*$ is a constant. We assume that any program satisfies annotative formulae such as $n=0$ whenever they are true. We can easily define a function that satisfies $\forall n.\, \forall m.\, \mathbf{Nat}(n) \to \mathbf{Nat}(m) \to \mathbf{Nat}(n+m)$ as:

$$\mathbf{add}\ x\ y \equiv \mathbf{p_2}\ x\ y,$$

or

$$\mathbf{add}'\ x\ y \equiv \mathbf{p_2}\ y\ x,$$

where $\mathbf{p_2}$ extracts second components, i.e., the method of addition in this particular case, from pairs. We could also define the successor function as a recursive program as:

$$\mathbf{s}\ x \equiv <\mathbf{i_2}\ x, \lambda y.\ \mathbf{add}\ x\ (\mathbf{s}\ y)>$$

or

$$\mathbf{s}'\ x \equiv <\mathbf{i_2}\ x, \lambda y.\ \mathbf{add}'\ x\ (\mathbf{s}'\ y)>.$$

In spite of the apparent symmetry between **add** and **add**′, which are both supposed to satisfy the same specification, the computational behaviors of **s** and **s**′ are completely different. We can observe that **s** works as expected, but **s**′

*Research supported in part by 1999 Overseas Researcher Program of Ryukoku University.

does not. For example, $\mathbf{p_2}\,(\mathbf{s}\;\mathbf{0})\;\mathbf{0}$ would be evaluated as: $\mathbf{p_2}\,(\mathbf{s}\;\mathbf{0})\;\mathbf{0} \to (\lambda y.\;\mathbf{add}\;\mathbf{0}\;(\mathbf{s}\;y))\;\mathbf{0} \to \mathbf{add}\;\mathbf{0}\;(\mathbf{s}\;\mathbf{0}) \to \mathbf{p_2}\,\mathbf{0}\;(\mathbf{s}\;\mathbf{0}) \to (\lambda x.\;x)\;(\mathbf{s}\;\mathbf{0}) \to \mathbf{s}\;\mathbf{0}$, whereas $\mathbf{p_2}\,(\mathbf{s}'\;\mathbf{0})\;\mathbf{0} \to (\lambda y.\,\mathbf{add}'\,\mathbf{0}\,(\mathbf{s}'\,y))\,\mathbf{0} \to \mathbf{add}'\,\mathbf{0}\,(\mathbf{s}'\,\mathbf{0}) \to \mathbf{p_2}\,(\mathbf{s}'\,\mathbf{0})\,\mathbf{0} \to \ldots$, and more generally, for any objects x and y of $\mathbf{Nat}(n)$ (for some n), $\mathbf{p_2}\,(\mathbf{s}'\;x)\;y \to \ldots \to \mathbf{p_2}\,(\mathbf{s}'\;y)\;x \to \ldots \to \mathbf{p_2}\,(\mathbf{s}'\;x)\;y \to \ldots$.

It should be noted that this sort of divergence would also be quite common in (careless) recursive definitions of programs even if we did not have to handle object-oriented specifications like $\mathbf{Nat}(n)$. The peculiarity here is the fact that the divergence is caused by a program, $\mathbf{add}'$, which is supposed to satisfy the same specification as **add**. This example shows such a loss of the compositionality of programs with respect to the specifications that imply their termination, or convergence. It also suggests that, to overcome this difficulty, **add** and $\mathbf{add}'$ should have different specifications, and accordingly the definition of $\mathbf{Nat}(n)$ should be revised in some way in order to force it.

$\lambda\mu$ and its logical inconsistency. The typing system $\lambda\mu$ (see [4], and Section 2 of the present paper for a summary) is a simply-typed lambda calculus with recursive types, where any form of self-references, including negative ones, is permitted. A non-trivial model for such unrestricted recursive types was developed by MacQueen, Plotkin and Sethi [22], and has been widely adopted as a theoretical basis for object-oriented type systems [1, 6].

On the other hand, it is well known that logical formulae with such unrestricted self-references would introduce a contradiction (variant of Russell's paradox). Therefore, logical systems must have certain restrictions on the forms of self-references (if ever allowed) in order to keep themselves sound; for example, μ-calculus [24, 20] does not allow negative self-references (see also [13]).

Through the formulae-as-types notion, this paradox corresponds to the fact that every type of $\lambda\mu$ is inhabited by a diverging program which does not produce any information; for example, the λ-term $(\lambda x.\,x\,x)(\lambda x.\,x\,x)$ can be typed with every type in $\lambda\mu$. Therefore, even with the model mentioned above, types can be regarded only as partial specifications of programs, and that is considered the reason why we lost the compositionality of programs in the $\mathbf{Nat}(n)$ case, where we regarded convergence of programs as a part of their specifications. This shows a contrast with the success of $\lambda\mu$ as a basis for type systems of object-oriented program languages, where the primary purpose of types, i.e., coarse specifications, is to prevent run-time type errors, and termination of programs is out of the scope.

The logical inconsistency of $\lambda\mu$ also implies that no mater how much types, or specifications, are refined, convergence of programs can not be expressed by them, and must be handled by endowing the typing system with some facilities for discussing computational properties of programs. For example, Constable et al. adopted this approach in their pioneering works to incorporate recursive definitions and partial objects into constructive type theory [10, 11]. However, in this paper, we will pursue another approach such that types themselves can express convergence of programs.

Towards the approximation modality. Suppose that we have a recursive program f defined by:

$$f \equiv F(f),$$

and want to show that f satisfies a certain specification S. Since the denotational meaning of f is given as the least fixed point of F, i.e., $f = \sup_{n<\omega} F^n(\bot)$, a possible way to do that would be to apply Scott's fixed point induction [25] by showing that:

- $\bot$ satisfies S,
- $F(x)$ satisfies S provided that x satisfies S, and
- S is chain closed.

However, this does not suffice for our purpose if S includes some requirement about the convergence of f, because obviously $\bot$, or even $F^n(\bot)$, could not satisfy the requirement. So we need more refined approach. The failure of the naive fixed point induction above suggests that the specification to be satisfied by each $F^n(\bot)$ inherently depends on n, and the requirement concerning its convergence must become stronger when n increases. This leads us to a layered version of the fixed-point induction scheme as follows: in order to show that f satisfies S, it suffices to find an infinite sequence S_0, S_1, S_2, ... of properties, or (virtual) specifications, such that:

(1) $S = \bigcap_{n<\omega} S_n$,
(2) $S_{n+1} \subset S_n$,
(3) $\bot$ satisfies S_0,
(4) $F(x)$ satisfies S_{n+1} provided that x satisfies S_n, and
(5) S_n is chain closed.

For, since $F^n(\bot) \in S_n$ for every n by (3) and (4), we get $F^k(\bot) \in S_n$ for every $k \geq n$ by (2). This and (5) imply $f \in S_n$ for every n, and consequently $f \in S$ by (1).

In this scheme, the sequence S_0, S_1, S_2, ... can be regarded as a successive approximation of S, and F a (higher-order) program which constructs a program that satisfies S_{n+1} from one that satisfies S_n. It should be also noted that F works independently of n. This uniformity of F over n leads us to consider a formalization of this scheme in a modal logic, where the set of possible worlds (in the sense of Kripke semantics) consists of all non-negative integers,

and S_n in the induction scheme above corresponds to the interpretation of S in the world n. We now write $x\ \mathbf{r}_k\ S$ to denote the fact that x satisfies the interpretation of S in the world k, and define a modality, say $\bullet$, as:

$$x\ \mathbf{r}_k\ \bullet S \quad \text{iff} \quad k = 0\ \text{ or }\ x\ \mathbf{r}_{k-1}\ S.$$

The condition (2) of the induction scheme says that $x\ \mathbf{r}_k\ S$ implies $x\ \mathbf{r}_l\ S$ for every $l \leq k$; in other words, the interpretation of specifications should be *hereditary* with respect to the accessibility relation $>$. In such a modal framework, the specification to be satisfied by F can be represented by $\bullet S \to S$ provided that the $\to$-connective is interpreted in the standard way in each world, and our induction scheme can be rewritten as:

> if $\bot\ \mathbf{r}_0\ S$ and $F\ \mathbf{r}_k\ \bullet S \to S$ for every $k > 0$, then $f\ \mathbf{r}_k\ S$ for every k.

Furthermore, if we assume that S_0 is a trivial specification which is satisfiable by any program, then, shifting the possible worlds downwards by one, we can simplify this to:

$(*)$ if $F\ \mathbf{r}_k\ \bullet S \to S$ for every k, then $f\ \mathbf{r}_k\ S$ for every k.

Although this assumption about S_0 somewhat restricts our choice of the sequence $S_0, S_1, S_2, \ldots$, it could be thought rather reasonable because, at any rate, S_0 must be an almost trivial specification that is even satisfiable by $\bot$. Note that S_{n+1} occurring in the induction now corresponds to the interpretation of S in the world n, and S_0 corresponds to the interpretation of $\bullet S$ in the world 0.

From this interpretation, we can extract some fundamental properties concerning the $\bullet$-modality, which introduce a subsumption, or subtyping, relation over specifications into our modal framework. First, the hereditary interpretation of specifications implies the following property:

- $x\ \mathbf{r}_k\ S$ implies $x\ \mathbf{r}_k\ \bullet S$.

Second, this and the standard interpretations of $\to$ imply the following two properties:

- $x\ \mathbf{r}_k\ S \to T$ implies $x\ \mathbf{r}_k\ \bullet S \to \bullet T$, and
- $x\ \mathbf{r}_k\ \bullet S \to \bullet T$ implies $x\ \mathbf{r}_k\ \bullet(S \to T)$.

Furthermore, if $x\ \mathbf{r}_k\ \top \to \top$ for every x and k, where $\top$ is the trivial specification which is satisfiable by any program, i.e., the universe of (meanings of) programs, then the converse of the second one is also true, that is:

- $x\ \mathbf{r}_k\ \bullet(S \to T)$ implies $x\ \mathbf{r}_k\ \bullet S \to \bullet T$.

Note that this is not always the case because we could consider non-extensional interpretations, e.g., F-semantics [15], in which $\lambda x.\ \bot\ \mathbf{r}_k\ \top \to \top$ holds, but $\bot\ \mathbf{r}_k\ \top \to \top$ does not.

Specification-level self-references. This modal framework introduced for program-level self-references also provides a basis for specification-level self-references. Suppose that we have a self-referential specification such as:

$$S = \phi(S).$$

As we saw in the **Nat**(n) case, negative reference to S in ϕ can introduce a contradiction in the conventional setting, and this is still true in our modal framework. However, in the world n, we can now refer to the interpretation of S in any world $k < n$ without worrying about the contradiction. That is, as long as S occurs only in scopes of the modal operator $\bullet$ in ϕ, the interpretation of S is well-defined and given as a fixed point of ϕ, which is actually shown to be unique. For example, if S is defined as $S = \bullet S \to T$, then S could be interpreted in each world as follows:

$$\begin{aligned}
S_0 &= \top \to T_0 \\
S_1 &= S_0 \cap ((\top \to T_0) \to T_1) \\
S_2 &= S_1 \cap ((S_0 \cap ((\top \to T_0) \to T_1)) \to T_2) \\
&\vdots \\
S_{n+1} &= S_n \cap (S_n \to T_{n+1}) \\
&\vdots
\end{aligned}$$

where S_k and T_k are the interpretations of S and T in the world k, respectively, and the notations such as $\top$ and $\to$ are abused to denote their expected interpretations also. Note that this kind of self-references provides us a method to define the sequence $S_0, S_1, S_2, \ldots$ for the refined induction scheme when we derive properties of recursive programs, and the induction scheme would be useless if we did not have such a method.

In the following sections, we will see that this form of specification-level self-references is quite powerful, and captures a wide range of specifications including those which are not representable in the conventional setting such as ones for **add** and **add**$'$ in the **Nat**(n) case. Furthermore, the modal version $(*)$ of the induction scheme turns out to be derivable from other properties of the $\bullet$-modality and such self-referential specifications, where the derivation corresponds to fixed point combinators, such as Curry's **Y**. This also gives us a way to construct recursive programs based on the proofs-as-programs notion.

2. A brief review of $\lambda\mu$

To fix notation, we begin with a brief review of $\lambda\mu$, which is a simply typed lambda calculus with recursive types. We first assume a set **Exp** of untyped λ-terms possibly containing *individual constants* $(c, d, \ldots)$. We use $M, N, K, L, \ldots$ to denote λ-terms. Free and bound

$$\frac{}{\Gamma \cup \{x : A\} \ \vdash_{\lambda\mu} \ x : A} \ (\text{var}) \qquad \frac{}{\Gamma \ \vdash_{\lambda\mu} \ c : \tau(c)} \ (\text{const}) \qquad \frac{\Gamma \ \vdash_{\lambda\mu} \ M : A}{\Gamma \ \vdash_{\lambda\mu} \ M : A'} \ (\simeq_{\lambda\mu}) \quad (A \simeq_{\lambda\mu} A')$$

$$\frac{\Gamma \cup \{x : A\} \ \vdash_{\lambda\mu} \ M : B}{\Gamma \ \vdash_{\lambda\mu} \ \lambda x.\, M : A \to B} \ (\to \mathrm{I}_{\lambda\mu}) \qquad \frac{\Gamma_1 \ \vdash_{\lambda\mu} \ M : A \to B \quad \Gamma_2 \ \vdash_{\lambda\mu} \ N : A}{\Gamma_1 \cup \Gamma_2 \ \vdash_{\lambda\mu} \ M\,N : B} \ (\to \mathrm{E}_{\lambda\mu})$$

Figure 1. The typing system $\lambda\mu$

occurrences of individual variables and the notion of α-convertibility are defined in the standard manner. Hereafter, we identify λ-terms by this α-convertibility. We denote the set of individual variables occurring freely in M by $FV(M)$, and use $M[N_1/x_1, \ldots, N_n/x_n]$ to denote the λ-term obtained from a λ-term M by substituting $N_1, \ldots, N_n$ for each free occurrence of individual variables $x_1, \ldots, x_n$, respectively, with necessary α-conversion to avoid accidental capture of free variables.

β-reduction is also defined in the standard manner, and treated as a binary relation $\underset{\beta}{\to}$ over **Exp**. We denote the transitive and reflexive closure of $\underset{\beta}{\to}$ by $\underset{\beta}{\overset{*}{\to}}$, and the symmetric closure of $\underset{\beta}{\to}$ by $\underset{\beta}{\leftrightarrow}$. We define the equivalence relation $\underset{\beta}{=}$ as the transitive and reflexive closure of $\underset{\beta}{\leftrightarrow}$. Our intended semantics for untyped λ-terms is summarized as the following, where we do not require extensionality with respect to their interpretations.

Definition 1 (β-model). A β-model of **Exp** is a tuple $<\mathcal{V}, \cdot, \sigma, [\![\ ]\!]^{\mathcal{V}}>$ such that:

1. $\mathcal{V}$: a non-empty set.
2. σ : **Const** $\to \mathcal{V}$.
3. $- \cdot - : \mathcal{V} \times \mathcal{V} \to \mathcal{V}$.
4. $[\![-]\!]^{\mathcal{V}}_{-}$: **Exp** $\to$ (**Var** $\to \mathcal{V}$) $\to \mathcal{V}$.
5. $[\![x]\!]^{\mathcal{V}}_{\rho} = \rho(x)$.
6. $[\![c]\!]^{\mathcal{V}}_{\rho} = \sigma(c)$.
7. $[\![M\,N]\!]^{\mathcal{V}}_{\rho} = [\![M]\!]^{\mathcal{V}}_{\rho} \cdot [\![N]\!]^{\mathcal{V}}_{\rho}$.
8. $[\![\lambda x.\, M]\!]^{\mathcal{V}}_{\rho} \cdot v = [\![M]\!]^{\mathcal{V}}_{\rho[v/x]}$, where $\rho[v/x](x) = v$ and $\rho[v/x](y) = \rho(y)$ if $y \neq x$.
9. If $M \underset{\beta}{=} N$, then $[\![M]\!]^{\mathcal{V}}_{\rho} = [\![N]\!]^{\mathcal{V}}_{\rho}$.

Definition 2 (Type expressions of $\lambda\mu$). The syntax of the type expressions of $\lambda\mu$ is defined relatively to the following two sets: **TConst** of *type constants* ($P, Q, R, \ldots$) and **TVar** of countably infinite *type variables* ($X, Y, Z, \ldots$). The set **TExp**$_{\lambda\mu}$ of *type expressions* of $\lambda\mu$ is defined as follows:

TExp$_{\lambda\mu}$	::=	**TConst**	(type constants)
	$\mid$	**TVar**	(type variables)
	$\mid$	**TExp**$_{\lambda\mu} \to$ **TExp**$_{\lambda\mu}$	(function types)
	$\mid$	μ**TVar**. **TExp**$_{\lambda\mu}$	(recursive types).

We use $A, B, C, D, \ldots$ to denote type expressions of $\lambda\mu$. We regard a type variable X as bound in $\mu X.A$. We use $A[B_1/X_1, \ldots, B_n/X_n]$ to denote the type expression obtained from A by substituting $B_1, \ldots, B_n$ for each free occurrence of $X_1, \ldots, X_n$, respectively. We denote the set of type variables occurring freely in A by $FTV(A)$. We regard α-convertible type expressions as identical; for example, $\mu X.X \to Y = \mu Z.Z \to Y$. We define the equivalence relation $\simeq_{\lambda\mu}$ over **TExp**$_{\lambda\mu}$ considering two type expressions of $\lambda\mu$ equivalent modulo $\simeq_{\lambda\mu}$ if they have the same (possibly infinite) type expression obtained by unfolding recursive types $\mu X.A$ occurring in them to $A[\mu X.A/X]$ indefinitely. This equivalence relation is known to be decidable (see [8] and [3]).

Definition 3 (Typing contexts). A *typing context*, or a *context* for short, of $\lambda\mu$ is a finite mapping that assigns a type expression of $\lambda\mu$ to each individual variable of its domain. We use Γ, Γ', ... to denote contexts, and $\{x_1 : A_1, \ldots, x_m : A_m\}$ to denote a context that assigns A_i to x_i $(i = 1, \ldots, m)$.

Definition 4 ($\lambda\mu$). Let τ be a mapping that assigns a type constant $\tau(c)$ to each individual constant c. The typing system $\lambda\mu$ is defined relatively to this τ by the derivation rules shown in Figure 1.

Definition 5 (Realizability models of $\lambda\mu$). A realizability model of $\lambda\mu$ is a tuple $<\mathcal{V}, \cdot, \sigma, [\![\ ]\!]^{\mathcal{V}}, \mathcal{T}, \delta, [\![\ ]\!]^{\mathcal{T}}>$ such that:

1. $<\mathcal{V}, \cdot, \sigma, [\![\ ]\!]^{\mathcal{V}}>$ is a β-model of **Exp**.
2. $\mathcal{T} \subset \mathcal{P}(\mathcal{V}) \quad (= \{\ S \mid S \subset \mathcal{V}\ \})$
3. δ : **TConst** $\to \mathcal{T}$
4. $\sigma(c) \in \delta(\tau(c))$

5. $[\![-]\!]^{\mathcal{T}}_{-} : \mathbf{TExp}_{\lambda\mu} \to (\mathbf{TVar} \to \mathcal{T}) \to \mathcal{T}$
6. $[\![X]\!]^{\mathcal{T}}_{\eta} = \eta(X)$
7. $[\![P]\!]^{\mathcal{T}}_{\eta} = \delta(P)$
8. $[\![A \to B]\!]^{\mathcal{T}}_{\eta} = \{\, v \mid v \cdot u \in [\![B]\!]^{\mathcal{T}}_{\eta} \text{ for every } u \in [\![A]\!]^{\mathcal{T}}_{\eta} \,\}$
9. If $A \simeq_{\lambda\mu} B$, then $[\![A]\!]^{\mathcal{T}}_{\eta} = [\![B]\!]^{\mathcal{T}}_{\eta}$.

It is not straightforward to construct a non-trivial realizability model of $\lambda\mu$. The first non-trivial model was developed by MacQueen, Plotkin and Sethi [22], by constructing a complete metric space of types and by interpreting recursive types as the fixed points of contractive type constructors (see also [2, 8]).

Proposition 1 (Soundness of $\lambda\mu$). *Let $<\mathcal{V}, \cdot, \sigma, [\![\,]\!]^{\mathcal{V}}, \mathcal{T}, \delta, [\![\,]\!]^{\mathcal{T}}>$ be a realizability model of $\lambda\mu$. If $\{x_1 : A_1, \ldots, x_n : A_n\} \vdash_{\lambda\mu} M : B$ is derivable, then $[\![M]\!]^{\mathcal{V}}_{\rho} \in [\![B]\!]^{\mathcal{T}}_{\eta}$ for every η and ρ provided $\rho(x_i) \in [\![A_i]\!]^{\mathcal{T}}_{\eta}$ $(i = 1, 2, \ldots, n)$.*

Nevertheless, as mentioned in Introduction, unrestricted self-references allowed in $\lambda\mu$ cause a logical contradiction as follows (Curry's paradox):

Proposition 2. *$\vdash_{\lambda\mu} (\lambda x.\, xx)\,(\lambda x.\, xx) : A$ is derivable for any type expression A.*

Proof. Let $C = \mu X.\, X \to A$, and Π as follows:

$$\Pi = \dfrac{\dfrac{\dfrac{x : C \vdash_{\lambda\mu} x : C}{x : C \vdash_{\lambda\mu} x : C \to A}\,(\simeq_{\lambda\mu}) \quad x : C \vdash_{\lambda\mu} x : C}{x : C \vdash_{\lambda\mu} xx : A}\,(\to E)}{\vdash_{\lambda\mu} \lambda x.\, xx : C \to A}\,(\to I)$$

Then, we can derive it as follows:

$$\dfrac{\dfrac{\vdots\ \Pi}{\vdash_{\lambda\mu} \lambda x.\, xx : C \to A} \quad \dfrac{\dfrac{\vdots\ \Pi}{\vdash_{\lambda\mu} \lambda x.\, xx : C \to A}}{\vdash_{\lambda\mu} \lambda x.\, xx : C}\,(\simeq_{\lambda\mu})}{\vdash_{\lambda\mu} (\lambda x.\, xx)\,(\lambda x.\, xx) : A}\,(\to E)$$

□

3. The typing system $\lambda\bullet\mu$

We now define a modal typing system, which is denoted by $\lambda\bullet\mu$, based on the idea discussed in Introduction. First, as a preparation for introducing the syntax of type expressions, we give the one of pseudo type expressions, which are obtained by adding a unary type constructor $\bullet$ to the one of $\mathbf{TExp}_{\lambda\mu}$.

Definition 6. We define the set **PTExp** of *pseudo type expressions* as follows:

PTExp	::= **TConst**	(type constants)
	| **TVar**	(type variables)
	| **PTExp** $\to$ **PTExp**	(function types)
	| $\bullet$**PTExp**	(approximative types)
	| μ**TVar**.**PTExp**	(recursive types).

We assume that $\to$ associates to the right as usual, and each (pseudo) type constructor associates according to the following priority:

$$\text{(Low)} \qquad \mu X. \;<\; \to \;<\; \bullet \qquad \text{(High)}.$$

For example, $\bullet\,\mu X.\bullet X \to Y \to Z$ is the same as $\bullet(\mu X.\,((\bullet X) \to (Y \to Z)))$. We use $\top$ as an abbreviation for $\mu X.\bullet X$ and use $\bullet^n A$ to denote a (pseudo) type expression $\underbrace{\bullet \ldots \bullet}_{n \text{ times}} A$, where $n \geq 0$.

Definition 7 ($\top$-variants). A type expression A is a *$\top$-variant* if and only if $A = \bullet^{m_0}\mu X_1.\bullet^{m_1}\mu X_2.\bullet^{m_2} \ldots \mu X_n.\bullet^{m_n} X_i$ for some $n, m_0, m_1, m_2, \ldots, m_n, X_1, X_2, \ldots, X_n$ and i such that $1 \leq i \leq n$ and $m_i + m_{i+1} + m_{i+2} + \ldots + m_n \geq 1$.

Definition 8 (Properness). A pseudo type expression A is *proper* in X if and only if X occurs freely only (a) in scopes of the $\bullet$-operator in A, or (b) in a subexpression $B \to C$ of A with C being a $\top$-variant.

For example, P, $\bullet X$, $\bullet(X \to Y)$, $X \to \bullet\mu Y.\bullet Y$ and $\mu Y.\,\bullet(X \to Y)$ are proper in X, and neither X, $X \to Y$ nor $\mu Y.\,\mu Z.\,X \to Y$ is proper in X.

Definition 9 (Type expressions of $\lambda\bullet\mu$). A *type expression* is a pseudo type expression such that A is proper in X for any of its subexpressions in the form of $\mu X.\,A$. We denote the set of type expressions of $\lambda\bullet\mu$ by **TExp**.

For example, P, X, $X \to Y$, $\mu X.\bullet X \to Y$, $\mu X.\,X \to \top$ and $\mu X.\bullet\,\mu Y.\,X \to Z$ are type expressions of $\lambda\bullet\mu$, and neither $\mu X.\,X \to Y$ nor $\mu X.\,\mu Y.\,X \to Y$ is a type expression of $\lambda\bullet\mu$. We also use $A, B, C, D, \ldots$ to denote type expressions of $\lambda\bullet\mu$, and define other notations such as $FTV(A)$ and $A[B_1/X_1, \ldots, B_n/X_n]$ similarly to the case of $\lambda\mu$.

Definition 10. We define $r(A)$, the *rank* of A, as follows:

$$r(P) = r(X) = r(\bullet A) = 0$$
$$r(A \to B) = \begin{cases} 0 & (B \text{ is a } \top\text{-variant}) \\ \max(r(A),\, r(B)) + 1 & \text{(otherwise)} \end{cases}$$
$$r(\mu X.\,A) = r(A) + 1$$

Observe that $r(A[B/X]) < r(\mu X.\,A)$ for every B if A is proper in X.

$$\frac{}{\gamma \cup \{X \preceq Y\} \vdash X \preceq Y}\ (\preceq\text{-assump}) \qquad \frac{}{\gamma \vdash A \preceq \top}\ (\preceq\text{-}\top)$$

$$\frac{}{\gamma \vdash A \preceq A'}\ (\preceq\text{-reflex}) \quad (A \simeq A') \qquad \frac{\gamma_1 \vdash A \preceq B \quad \gamma_2 \vdash B \preceq C}{\gamma_1 \cup \gamma_2 \vdash A \preceq C}\ (\preceq\text{-trans})$$

$$\frac{\gamma \vdash A \preceq B}{\gamma \vdash \bullet A \preceq \bullet B}\ (\preceq\text{-}\bullet) \qquad \frac{\gamma_1 \vdash A' \preceq A \quad \gamma_2 \vdash B \preceq B'}{\gamma_1 \cup \gamma_2 \vdash A \to B \preceq A' \to B'}\ (\preceq\text{-}\to)$$

$$\frac{}{\gamma \vdash A \preceq \bullet A}\ (\preceq\text{-approx}) \qquad \frac{}{\gamma \vdash A \to B \preceq \bullet A \to \bullet B}\ (\preceq\text{-}\to\bullet) \qquad \frac{}{\gamma \vdash \bullet A \to \bullet B \preceq \bullet(A \to B)}\ (\preceq\text{-}\bullet\to)$$

$$\frac{\gamma \cup \{X \preceq Y\} \vdash A \preceq B}{\gamma \vdash \mu X.A \preceq \mu Y.B}\ (\preceq\text{-}\mu) \qquad \left(\begin{array}{l} X \notin FTV(\gamma) \cup FTV(B), Y \notin FTV(\gamma) \cup FTV(A), \\ \text{and } A \text{ and } B \text{ are proper in } X \text{ and } Y, \text{ respectively} \end{array}\right)$$

Figure 2. The subtyping rules of $\lambda\bullet\mu$

Definition 11 ($\simeq$). The equivalence relation $\simeq$ over **TExp** is defined as the smallest binary relation that satisfies:

- $A \simeq A$.
- If $A \simeq B$, then $B \simeq A$.
- If $A \simeq B$ and $B \simeq C$, then $A \simeq C$.
- If $A \simeq B$, then $\bullet A \simeq \bullet B$.
- If $A \simeq C$ and $B \simeq D$, then $A \to B \simeq C \to D$.
- $A \to \top \simeq B \to \top$.
- $\mu X.A \simeq A[\mu X.A/X]$.
- If $A \simeq C[A/X]$ and C is proper in X, then $A \simeq \mu X.C$.

Two type expressions of $\lambda\bullet\mu$ are equivalent modulo $\simeq$, if their (possibly infinite) type expression obtained by indefinite unfolding recursive types occurring in them are identical modulo the rule $A \to \top \simeq B \to \top$.

Proposition 3. *A type expression A is a $\top$-variant if and only if $A \simeq \top$.*

Definition 12 (Canonical types). We define a set **CTExp** of *canonical* type expressions as follows:

$$\begin{array}{rcl} \textbf{CTExp} & ::= & \top \mid \bullet^n\textbf{TConst} \mid \bullet^n\textbf{TVar} \\ & & \mid \bullet^n(\textbf{TExp} \to \textbf{TExp}), \end{array}$$

where n is an arbitrary non-negative integer.

Proposition 4. *There exists an effective procedure for calculating a canonical type expression A^c such that $A^c \simeq A$ from a given type expression A of $\lambda\bullet\mu$.*

Subtyping. As mentioned in Introduction, our intended interpretation of the $\bullet$-modality introduces a subtyping relation into **TExp**. We now define the subtyping relation by a set of inference rules as in [3].

Definition 13. A *subtyping assumption* is a finite set of pairs of type variables such that any type variable appears at most once in the set. We write $\{X_1 \preceq Y_1, X_2 \preceq Y_2, \ldots, X_n \preceq Y_n\}$ to denote the subtyping assumption $\{ <X_i, Y_i> \mid i = 1, 2, \ldots, n \}$. We use γ, γ', γ_1, γ_2, $\ldots$ to denote subtyping assumptions, and $FTV(\gamma)$ to denote the set of type variables occurring in γ.

Definition 14 ($\preceq$). We define the derivability of *subtyping judgment* $\gamma \vdash A \preceq B$ by the derivation rules shown in Figure 2. Note that $\gamma \cup \{X \preceq Y\}$ and $\gamma_1 \cup \gamma_2$ in the rules must be (valid) subtyping assumptions, i.e., any type variable must not have more than one occurrence in them. We also define a binary relation $\preceq$ over **TExp** as: $A \preceq B$ if and only if $\{\} \vdash A \preceq B$ is derivable.

Most of the subtyping rules are standard. The rule ($\preceq$-μ) corresponds to the "Amber rule" [7]. The rules ($\preceq$-$\top$), ($\preceq$-$\bullet$), ($\preceq$-approx), ($\preceq$-$\to\bullet$) and ($\preceq$-$\bullet\to$) reflect our intended meaning of the $\bullet$-modality discussed in Introduction.

Proposition 5 (Basic properties of $\preceq$).

1. *$\top \preceq A$ if and only if $A \simeq \top$.*
2. *$\bullet A \preceq \bullet B$ if and only if $A \preceq B$.*
3. *$\bullet A \not\preceq X$, $\bullet A \not\preceq P$, and $\bullet A \not\preceq B \to C$.*

$$\frac{}{\Gamma \cup \{x : A\} \vdash x : A}\ (\text{var}) \qquad \frac{}{\Gamma \vdash c : \tau(c)}\ (\text{const})$$

$$\frac{}{\Gamma \vdash M : \top}\ (\top) \qquad \frac{\bullet\Gamma \vdash M : \bullet A}{\Gamma \vdash M : A}\ (\bullet) \qquad \frac{\Gamma \vdash M : A \qquad \vdash A \preceq B}{\Gamma \vdash M : B}\ (\preceq)$$

$$\frac{\Gamma \cup \{x : A\} \vdash M : B}{\Gamma \vdash \lambda x.\, M : A \to B}\ (\to \text{I}) \qquad \frac{\Gamma_1 \vdash M : \bullet^n(A \to B) \quad \Gamma_2 \vdash N : \bullet^n A}{\Gamma_1 \cup \Gamma_2 \vdash M\,N : \bullet^n B}\ (\to \text{E})$$

Figure 3. The typing rules of $\lambda\bullet\mu$

The typing rules. We now define the typing rules of $\lambda\bullet\mu$. According to the intended meaning of $\bullet$, two new typing rules, $(\bullet)$ and $(\preceq)$, are added and the $(\to \text{E}_{\lambda\mu})$ rule is generalized to handle types with the $\bullet$-modality.

Definition 15 (Typing rules). Typing contexts for $\lambda\bullet\mu$ are defined similarly to the case of $\lambda\mu$. Let τ be a mapping that assigns a type constant $\tau(c)$ to each individual constant c. The typing system $\lambda\bullet\mu$ is defined relatively to τ by the derivation rules shown in Figure 3, where $\bullet\,\Gamma$ denotes the typing context $\{x_1 : \bullet A_1,\ x_2 : \bullet A_2,\ \ldots,\ x_n : \bullet A_n\}$ when $\Gamma = \{\ x_1 : A_1,\ x_2 : A_2,\ \ldots,\ x_n : A_n\}$.

The $(\bullet)$-rule represents the fact that every possible world n has its successor $n{+}1$. Since the interpretation of $\Gamma \vdash M : A$ in the world n is identical to the one of $\bullet\Gamma \vdash M : \bullet A$ in the world $n{+}1$, $\Gamma \vdash M : A$ is valid whenever so is $\bullet\Gamma \vdash M : \bullet A$. The $(\to\text{E})$-rule allows us to derive $\vdash \lambda x.\, \lambda y.\, xy : \bullet(A \to B) \to \bullet A \to \bullet B$ for every A and B. Therefore, $\bullet(A \to B)$ and $\bullet A \to \bullet B$ are *logically* equivalent, even though not equivalent as sets of λ-terms. Note that $\bullet A \to \bullet B \preceq \bullet(A \to B)$, but $\bullet(A \to B) \not\preceq \bullet A \to \bullet B$.

Example 1. We can derive Curry's fixed-point combinator **Y** in $\lambda\bullet\mu$; more precisely, the following is derivable.

$$\vdash \lambda f.\, (\lambda x.\, f\,(xx))\ (\lambda x.\, f\,(xx)) \ :\ (\bullet X \to X) \to X$$

Let a formula $A = \mu Y.\bullet Y \to X$ and a derivation Π as follows:

$$\Pi = \dfrac{\dfrac{\dfrac{}{x : \bullet A \vdash x : \bullet A}\ (\text{var})}{x : \bullet A \vdash x : \bullet(\bullet A \to X)}\ (\simeq) \quad \dfrac{\dfrac{}{x : \bullet A \vdash x : \bullet A}\ (\text{var})}{x : \bullet A \vdash x : \bullet\bullet A}\ (\preceq)}{x : \bullet A \vdash xx : \bullet X}\ (\to\text{E})$$
$$\dfrac{\dfrac{\dfrac{}{f : \bullet X \to X \vdash f : \bullet X \to X}\ (\text{var}) \quad \vdots}{f : \bullet X \to X,\ x : \bullet A \vdash f\,(xx) : X}\ (\to\text{E})}{f : \bullet X \to X \vdash \lambda x.\, f\,(xx) : \bullet A \to X}\ (\to\text{I})$$

Then, let $\Gamma = \{f : \bullet X \to X\}$ and $L = \lambda x.\, f\,(xx)$. We can derive **Y** as follows:

$$\dfrac{\dfrac{\dfrac{\vdots\ \Pi}{\Gamma \vdash L : \bullet A \to X} \quad \dfrac{\dfrac{\vdots\ \Pi}{\Gamma \vdash L : \bullet A \to X}}{\Gamma \vdash L : \bullet A}\ (\preceq)}{\Gamma \vdash LL : X}\ (\to\text{E})}{\vdash \lambda f.\, LL : (\bullet X \to X) \to X}\ (\to\text{I})$$

We can also observe that Turing's fixed point combinator $(\lambda x.\lambda f.\, f\,(xxf))\ (\lambda x.\ \lambda f.\, f\,(xxf))$ has the same type. The type $(\bullet X \to X) \to X$ gives a concise axiomatic meaning to the fixed point combinators; it says that they can produce an element of X with a given function that works as an information pump from $\bullet X$ to X; in other words, they provide the induction scheme discussed in Introduction. The type thus enables us to construct recursive programs using the fixed point combinators without analyzing their computational behavior. We will see some examples of such recursive programs in Section 6.

Basic property of $\lambda\bullet\mu$. The typing system $\lambda\bullet\mu$ enjoys some basic properties such as subject reduction property.

Lemma 1. *Let Γ_1 and Γ_2 be typing contexts such that $Dom(\Gamma_1) \cap Dom(\Gamma_2) = \{\}$. If $\Gamma_1 \cup \Gamma_2 \vdash M : A$ is derivable, then so is $\bullet\,\Gamma_1 \cup \Gamma_2 \vdash M : \bullet\,A$.*

Lemma 2 (Substitution lemma). *If $\Gamma \cup \{x : A\} \vdash M : B$ and $\Gamma \vdash N : A$ are derivable, then so is $\Gamma \vdash M[N/x] : B$.*

Theorem 1 (Subject reduction). *If $\Gamma \vdash M : A$ is derivable and $M \underset{\beta}{\to} M'$, then $\Gamma \vdash M' : A$ is derivable.*

Proof. By induction on the structure of M. Use Lemmas 1 and 2. □

4. A realizability interpretation

In this section, we give a realizability interpretation of $\lambda\bullet\mu$, and show soundness of $\lambda\bullet\mu$ with respect to the interpretation.

Definition 16. Let $<\mathcal{T}, \sqsubseteq>$ be a partially ordered set. We define a set $\mathcal{A}(\mathcal{T}, \sqsubseteq)$ of infinite sequences of elements of $\mathcal{T}$ as follows:

$$\mathcal{A}(\mathcal{T}, \sqsubseteq) = \{ <t_0, t_1, t_2, \ldots t_k, \ldots> \mid t_{k+1} \sqsubseteq t_k \text{ for every } k \ \}$$

We denote the k-th element of $t \in \mathcal{A}(\mathcal{T}, \sqsubseteq)$ by t_k. Note that t starts with its 0-th element t_0.

Definition 17. A *realizability interpretation* of $\lambda\bullet\mu$ is a tuple $<\mathcal{V}, \cdot, \sigma, [\![\ ]\!]^{\mathcal{V}}, \mathcal{K}, \theta>$ such that:

1. $<\mathcal{V}, \cdot, \sigma, [\![\ ]\!]^{\mathcal{V}}>$ is a β-model of **Exp**.
2. $\mathcal{K} \subset \mathcal{V}$.
3. $u \cdot v \in \mathcal{K}$ for every $u \in \mathcal{K}$ and $v \in \mathcal{V}$.
4. $\theta : \mathbf{TConst} \to \mathcal{A}(\{ S \mid \mathcal{K} \subset S \subset \mathcal{V} \}, \subset)$.
5. $\sigma(c) \in \theta(\tau(c))_k$ for every c and k.

We call a mapping $\xi : \mathbf{TVar} \to \mathcal{A}(\{ S \mid \mathcal{K} \subset S \subset \mathcal{V} \}, \subset)$ a *type environment*.

Definition 18 (Semantics of types). Let $<\mathcal{V}, \cdot, \sigma, [\![\ ]\!]^{\mathcal{V}}, \mathcal{K}, \theta>$ be a realizability interpretation of $\lambda\bullet\mu$, ξ a type environment. We assign an element $\mathcal{I}(A)^{\xi}$ of $\mathcal{A}(\{ S \mid \mathcal{K} \subset S \subset \mathcal{V} \}, \subset)$ to each type expression A as follows:

$$\begin{aligned}
\mathcal{I}(P)^{\xi}_k &= \theta(P)_k \\
\mathcal{I}(X)^{\xi}_k &= \xi(X)_k \\
\mathcal{I}(\bullet A)^{\xi}_k &= \{ u \mid u \in \mathcal{I}(A)^{\xi}_l \text{ for every } l < k \} \\
\mathcal{I}(A \to B)^{\xi}_k &= \\
&\left\{ u \;\middle|\; \begin{array}{l} 1.\ u \in \mathcal{I}(A \to B)^{\xi}_l \text{ for every } l < k. \\ 2.\ \text{If } B \text{ is not a } \top\text{-variant, then } u \cdot v \in \\ \quad \mathcal{I}(B)^{\xi}_k \text{ for every } v \in \mathcal{I}(A)^{\xi}_k. \\ 3.\ u \in \mathcal{K} \text{ or } u = [\![\lambda x.\ M]\!]^{\mathcal{V}}_{\rho} \text{ for some } x, \\ \quad \rho \text{ and } M. \end{array} \right\} \\
\mathcal{I}(\mu X.A)^{\xi}_k &= \mathcal{I}(A[\mu X.A/X])^{\xi}_k
\end{aligned}$$

Note that the $\mathcal{I}(A)^{\xi}_k$ is defined by induction on the lexicographic ordering of $<k, r(A)>$. We can easily check that $\mathcal{K} \subset \mathcal{I}(A)^{\xi}_{k+1} \subset \mathcal{I}(A)^{\xi}_k$ for every k. It should also be noted that $\mathcal{I}(A)^{\xi}_k = \mathcal{V}$ whenever A is a $\top$-variant; and therefore, the conditional "if B is not a $\top$-variant" is redundant for the clause 2 of the definition of $\mathcal{I}(A \to B)$. The set $\mathcal{K}$ takes a rather technical role (cf. [21]) in this semantics, and is only used to show head normalizability of λ-terms of certain types in the proofs of (2) and (3) of Theorem 3. It can usually be considered an empty set. The third condition for $u \in \mathcal{I}(A \to B)^{\xi}_k$ implies that we distinguish $\lambda x.\ M\,x$ from M unless $M \underset{\beta}{=} \lambda y.\ N$ for some y and N, even if M has a function type. Note that $\mathcal{I}(\bullet(A \to B))^{\xi}_k = \mathcal{I}(\bullet A \to \bullet B)^{\xi}_k$ for every $k > 0$, but $\mathcal{I}(\bullet(A \to B))^{\xi}_0 = \mathcal{V}$ and $\mathcal{I}(\bullet A \to \bullet B)^{\xi}_0 \neq \mathcal{V}$ by this condition. Thus, $\bullet(A \to B) \simeq \bullet A \to \bullet B$ is not valid in this interpretation. We can also consider a variant system of $\lambda\bullet\mu$ with this equality, where we have to drop the third condition from $u \in \mathcal{I}(A \to B)^{\xi}_k$ to get its soundness. However, we omit the details from this paper.

The typing system $\lambda\bullet\mu$ is sound with respect to this semantics.

Lemma 3. *1. If $A \simeq B$, then $\mathcal{I}(A)^{\xi} = \mathcal{I}(B)^{\xi}$.*

2. If $A \preceq B$, then $\mathcal{I}(A)^{\xi}_k \subset \mathcal{I}(B)^{\xi}_k$ for every k.

Theorem 2 (Soundness). *Let the tuple $<\mathcal{V}, \cdot, \sigma, [\![\ ]\!]^{\mathcal{V}}, \mathcal{K}, \theta>$ be a realizability interpretation of $\lambda\bullet\mu$, and ξ a type environment. If $\{x_1 : A_1, \ldots, x_n : A_n\} \vdash M : B$ is derivable in $\lambda\bullet\mu$, then $[\![M]\!]^{\mathcal{V}}_{\rho} \in \mathcal{I}(B)^{\xi}_k$ for every k, ξ and ρ provided $\rho(x_i) \in \mathcal{I}(A_i)^{\xi}_k$ for every i ($i = 1, 2, \ldots, n$).*

Proof. By induction on the derivation and by cases of the last rule used in the derivation. Most cases are straightforward. Use Lemma 3 for the case of $(\preceq)$. Prove it by induction on k in the case of $(\to\mathrm{I})$. □

5. Convergence of well-typed terms

The soundness theorem assures the convergence of well-typed λ-terms according to their types. In this section we give a summary of such results.

Definition 19. A λ-term M is a *weak head normal form* if M is either of the following forms: (1) c, (2) $\lambda x.\ N$, or (3) $x\,N_1\,N_2\,\ldots\,N_n$ $(n \geq 0)$, and is a *head normal form* if M is either of the following forms: (1) c, or (2) $\lambda x_1.\,\lambda x_2.\,\ldots\lambda x_m.\,y\,N_1\,N_2\,\ldots\,N_n$ $(m, n \geq 0)$.

We say that M *has a (weak) head normal form*, or *is (weakly) head normalizable*, if $M \underset{\beta}{\overset{*}{\to}} M'$ for some (weak) head normal form M'. Note that if M is (weakly) head normalizable and $M \underset{\beta}{=} M'$, then M' is also (weakly) head normalizable. We also define Böhm trees of λ-terms in the standard manner according to this definition of head normal forms, in which λ-terms without head normal forms are denoted by $\bot$. We say that a λ-term is *maximal* if its Böhm tree has no occurrence of $\bot$.

Definition 20 (Tail finite types). A type expression A is *tail finite* if and only if $A \simeq \bullet^{m_1}(B_1 \to \bullet^{m_2}(B_2 \to \bullet^{m_3}(B_3$

$\to \ldots \to \bullet^{m_n}(B_n \to C)\ldots))))$ for some n, m_0, m_1, m_2, $\ldots$, m_n, B_1, B_2, $\ldots$, B_n and C such that $C \not\simeq \top$ and $C \not\simeq \bullet^k(D \to E)$ for every k, D and E.

Definition 21. Let A be a type expression. We define sets $ETV^+(A)$ and $ETV^-(A)$ of type variables as follows:

$$ETV^{\pm}(P) = \{\}$$
$$ETV^{+}(X) = \{X\}, \quad ETV^{-}(X) = \{\}$$
$$ETV^{\pm}(\bullet A) = ETV^{\pm}(A)$$
$$ETV^{\pm}(A \to B) = \begin{cases} \{\} & (B \text{ is a } \top\text{-variant}) \\ ETV^{\mp}(A) \cup ETV^{\pm}(B) & (\text{otherwise}) \end{cases}$$
$$ETV^{\pm}(\mu X.A) = (ETV^{\pm}(A) - \{X\}) \cup \begin{cases} ETV^{\mp}(A) - \{X\} & (X \in ETV^{-}(A)) \\ \{\} & (\text{otherwise}) \end{cases}$$

The set $ETV^+(A)$ ($ETV^-(A)$) consists of the type variables that have free positive (negative) occurrences in A, where we ignore any subexpression $B \to C$ of A whenever C is a $\top$-variant. Note also that $ETV^{\pm}(A) \subset FTV(A)$.

Definition 22 (Positively and negatively finite types). A type expression A is *positively (negatively) finite* if and only if C is tail finite whenever $A \simeq B[C/X]$ for some B and X such that $X \in ETV^+(B)$ ($X \in ETV^-(B)$) and $X \notin ETV^-(B)$ ($X \notin ETV^+(B)$).

Note that every positively finite type expression is tail finite.

Proposition 6. *The following three properties of a given type expression A are decidable.*

(1) $A \simeq \top$.

(2) *A is tail finite.*

(3) *A is positively (negatively) finite.*

Proof (sketch). The decidability of (1) is straightforward from Proposition 3. For the property (2), let V be a set of type variables, and define $\mathbf{TF}^V$ as follows:

$$\mathbf{TF}^V ::= \mathbf{TConst} \mid X \quad (X \in \mathbf{TVar} - V) \mid \bullet\mathbf{TF}^V \mid \mathbf{TExp} \to \mathbf{TF}^V \mid \mu X.\mathbf{TF}^{V \cup \{X\}} \quad (X \in \mathbf{TVar}).$$

It suffices to show that A is tail finite if and only if $A \in \mathbf{TF}^{\{\}}$. For (3), show that A is positively (negatively) finite if and only if $A \in \mathbf{PF}$ ($\mathbf{NF}$), where $\mathbf{PF}$ and $\mathbf{NF}$ are defined as follows:

$$\mathbf{PF} ::= \mathbf{TConst} \mid \mathbf{TVar} \mid \bullet\mathbf{PF} \mid \mathbf{NF} \to \mathbf{PF} \mid \mu X.A \quad \begin{pmatrix} \mu X.A \in \mathbf{TF}^{\{\}},\ A \in \mathbf{PF}, \text{ and (a)} \\ A \in \mathbf{NF} \text{ or (b) } X \notin ETV^-(A) \end{pmatrix}.$$

$$\mathbf{NF} ::= \mathbf{TConst} \mid \mathbf{TVar} \mid \bullet\mathbf{NF} \mid \mathbf{PF} \to \mathbf{NF} \mid \mathbf{TExp} \to A \quad (A \text{ is a } \top\text{-variant}) \mid \mu X.A \quad \begin{pmatrix} A \in \mathbf{NF}, \text{ and (a) } \mu X.A \in \mathbf{PF} \\ \text{or (b) } X \notin ETV^-(A) \end{pmatrix}.$$

□

Theorem 3 (Convergence). *Let $\Gamma \vdash M : A$ be a derivable typing judgment of $\lambda\bullet\mu$.*

(1) If $A \not\simeq \top$, then M has a weak head normal form.

(2) If A is tail finite, then M has a head normal form.

(3) If A is positively finite and $\Gamma(x)$ is negatively finite for every $x \in Dom(\Gamma)$, then M is maximal.

Proof (sketch). We consider the following β-model (term model) $<\mathcal{V}, \cdot, \sigma, [\![\ ]\!]^{\mathcal{V}}>$ of **TExp**:

- $\mathcal{V} = \mathbf{Exp}/_{=_\beta}$
- $[\![c]\!]^{\mathcal{V}}_\rho = c$
- $M \cdot N = MN$
- $[\![MN]\!]^{\mathcal{V}}_\rho = [\![M]\!]^{\mathcal{V}}_\rho\ [\![N]\!]^{\mathcal{V}}_\rho$
- $\sigma(c) = c$
- $[\![\lambda x.\ M]\!]^{\mathcal{V}}_\rho = \lambda x.\ [\![M]\!]^{\mathcal{V}}_{\rho[x/x]}$
- $[\![x]\!]^{\mathcal{V}}_\rho = \rho(x)$

Let $\mathcal{K}_0 = \{\ xN_1N_2\ldots N_n \mid x \in \mathbf{Var},\ n \geq 0 \text{ and } N_i \in \mathcal{V}\ (i = 1, 2, \ldots, n)\ \}$, and $\theta(P)_k = \mathcal{K}_0 \cup \{\ M \mid M =_\beta c \text{ for some } c \in \mathbf{Const}\ \}$ for every k and $P \in \mathbf{TConst}$. We can easily show that $<\mathcal{V}, \cdot, \sigma, [\![\ ]\!]^{\mathcal{V}}, \mathcal{K}_0, \theta>$ is a realizability interpretation of $\lambda\bullet\mu$. Fixing ρ and ξ as $\rho(x) = x$ for every x, and $\xi(X)_k = \mathcal{K}_0$ for every k and X, respectively, we get $M \in \mathcal{I}(A)^\xi_k$ for every k by Theorem 2 because $[\![M]\!]^{\mathcal{V}}_\rho = M$ and $\rho(x) = x \in \mathcal{K}_0 \subset \mathcal{I}(\Gamma(x))^\xi_k$ for every $x \in Dom(\Gamma)$. We can easily show (1) by cases of the form of A since we may assume that A is a canonical type expression. For (2), the proof proceeds by induction on the structure of A. As for (3), by induction on the depth of the Böhm tree nodes, in which we use the result of (2); however, the proof also needs some technical lemmas. □

6. $\lambda\bullet\mu$ as a basis for logic of programs

The typing system $\lambda\bullet\mu$ and its interpretation can be easily extended to cover full propositional and second-order types. For example, we can add the following rules for product types.

$$\frac{\Gamma \vdash M : A_1 \quad \Gamma \vdash N : A_2}{\Gamma \vdash <M, N> : A_1 \times A_2}\ (\times\mathrm{I})$$

$$\frac{\Gamma \vdash M : \bullet^n(A_1 \times A_2)}{\Gamma \vdash \mathbf{p}_i M : \bullet^n A_i}\ (\times\mathrm{E}) \qquad (i = 1, 2)$$

With the help of such extensions, $\lambda\bullet\mu$ can be a basis for logic of a wide range of programs. In this section we give some examples.

Streams. Streams, or infinite sequences, of data of a type X are representable by the type $\mu Y.\, X \times \bullet Y$. Since this type is positively finite, its elements are all maximal. We can derive recursive programs over streams using fixed point combinators such as **Y**; for example, the program $\lambda x.\ \mathbf{Y}(\lambda y.\ <x, y>)$, which generates a constant stream of a given value of the type X, has the type $X \to A$, and the program $\mathbf{Y}(\lambda f.\ \lambda x.\ \lambda y.\ <\mathbf{p_1}\, x, f\, y\, (\mathbf{p_2}\, x)>)$, which merges two streams, has type $A \to A \to A$, where $A = \mu Y.\, X \times \bullet Y$. More complicated programs over streams such as the prime number generator based on the sieve of Eratosthenes are also derivable by extending our logic to a predicate logic endowed with an arithmetic (powerful enough to handle, e.g., primitive recursive functions) for annotation, and by allowing type expressions such as $\bullet^t A$ as well-formed type expressions, with t being a numeric expression.

McCarthy's 91-function. Provided such an extension to predicate logic, we can construct a wide range of recursive programs with fixed point combinators assuring their termination. Consider the following recursive program, which represents McCarthy's 91-function:

$$f\ x \equiv \textbf{if}\ (x > 100)\ \textbf{then}\ x - 10\ \textbf{else}\ f\ (f\ (x + 11)).$$

We can show that f has a type, or satisfies a specification, $\forall n.\, \mathbf{nat}(n) \to \bullet^{101 \dot{-} n}\mathbf{nat}(g(n))$, where n ranges over non-negative integers, and $\mathbf{nat}(n)$ represents the implementation of the non-negative integer n; $\dot{-}$ and g are primitive recursive functions defined in the arithmetic as:

$$x \dot{-} y \equiv \begin{cases} x - y & (\text{if } x \geq y) \\ 0 & (\text{otherwise}) \end{cases}$$

$$g(x) \equiv \begin{cases} x - 10 & (\text{if } x > 100) \\ 91 & (\text{otherwise}) \end{cases}$$

and $\forall n.\, A(n)$ is interpreted as:

$$\mathcal{I}(\forall n.\, A(n))^{\xi}_{k} = \{\ u \mid u \in \mathcal{I}(A(n))^{\xi}_{k} \text{ for all } n\ \}.$$

Suppose that $f : \bullet \forall n.\, \mathbf{nat}(n) \to \bullet^{101 \dot{-} n}\mathbf{nat}(g(n))$ and $x : \mathbf{nat}(n)$. The type of **Y** assures that it suffices to show:

$$\textbf{if}\ (x > 100)\ \textbf{then}\ x - 10\ \textbf{else}\ f\ (f\ (x + 11)) : \forall n.\, \mathbf{nat}(n) \to \bullet^{101 \dot{-} n}\mathbf{nat}(g(n)).$$

We assume that $-$ and $+$ satisfy $\forall m.\, \forall n.\, \mathbf{nat}(m) \to \mathbf{nat}(n) \to \mathbf{nat}(n \dot{-} m)$ and $\forall m.\, \forall n.\, \mathbf{nat}(m) \to \mathbf{nat}(n) \to \mathbf{nat}(n+m)$, respectively. First, we get:

$$f\ (x{+}11) : \bullet\bullet^{101 \dot{-} (n+11)}\mathbf{nat}(g(n{+}11)).$$

If $n \leq 90$, then we get $f\ (x{+}11) : \bullet^{91 \dot{-} n}\, \mathbf{nat}(91)$ by the definitions of $\dot{-}$ and g; and therefore, $f\ (f\ (x{+}11)) : \bullet^{91 \dot{-} n}\bullet^{101 - 91}\mathbf{nat}(g(91))$, which is equivalent to $\bullet^{101 \dot{-} n}\mathbf{nat}(91)$. On the other hand, if $90 < n \leq 100$, then we similarly get $f\ (x{+}11) : \bullet\, \mathbf{nat}(n{+}1)$; and therefore, $f\ (f\ (x{+}11)) : \bullet\bullet^{101 \dot{-} (n+1)}\mathbf{nat}(g(n{+}1))$, which is also equivalent to $\bullet^{101 \dot{-} n}\mathbf{nat}(91)$. Otherwise, i.e., if $100 < n$, obviously $x{-}10 : \mathbf{nat}(g(n))$.

In this derivation, the fixed point combinator worked as the induction scheme discussed in Introduction with a sequence $S_0, S_1, S_2, \ldots, S_n$ as follows:

$$S_0 = \mathcal{V}$$
$$S_{k+1} = \{\ f \mid \forall n \geq 101 \dot{-} k.\ f(n) = g(n)\ \}.$$

Note also that if $\mathcal{I}(\mathbf{nat}(n))^{\xi}_{k}$ does not depend on k, then the interpretation of $\vdash\ f : \forall n.\, \mathbf{nat}(n) \to \bullet^{101 \dot{-} n}\mathbf{nat}(g(n))$ implies $f \in \mathcal{I}(\forall n.\, \mathbf{nat}(n) \to \mathbf{nat}(g(n)))^{\xi}_{k}$ for every k. Therefore, the apparent complexity of the type expression is not essential, and it can be observed that $\vdash\ f : \forall n.\, \mathbf{nat}(n) \to \mathbf{nat}(g(n))$ becomes formally derivable from $\vdash\ f : \forall n.\, \mathbf{nat}(n) \to \bullet^{101 \dot{-} n}\mathbf{nat}(g(n))$ if we introduce another modality, say $\Box$, which is interpreted as:

$$\mathcal{I}(\Box A)^{\xi}_{k} = \{\ u \mid u \in \mathcal{I}(A)^{\xi}_{l} \text{ for every } l\ \},$$

and accordingly enjoys the following subtyping relations and typing rules:

- $A \preceq B$ implies $\Box A \preceq \Box B$
- $\Box(A \to B) \preceq \Box A \to \Box B$
- $\Box A \preceq A$
- $\Box A \preceq \Box\Box A$
- $\Box\bullet^t A \preceq \Box A$
- $\mathbf{nat}(n) \preceq \Box\mathbf{nat}(n)$

$$\frac{\Gamma\ \vdash\ M : A}{\Box\Gamma\ \vdash\ M : \Box A}\ (\Box) \qquad \frac{\Box\Gamma_1 \cup \bullet\Gamma_2\ \vdash\ M : \bullet A}{\Box\Gamma_1 \cup \Gamma_2\ \vdash\ M : A}\ (\bullet)$$

The $(\bullet)$-rule supersedes the original one in Figure 3. Recursive type variables are not allowed to occur in scopes of the $\Box$-operator, and $\mathcal{I}(A)^{\xi}_{k}$ is now defined by induction on the lexicographic ordering of $<b(A),\ k,\ r(A)>$, where $b(A)$ is the depth of nesting occurrences of $\Box$ in A.

The $\mathbf{Nat}(n)$-example. We now reconsider the example of object-oriented natural numbers with an addition method. We revise the definition of $\mathbf{Nat}(n)$ as follows:

$$\mathbf{Nat}(n) \equiv ((n = 0) + (n > 0 \wedge \bullet\, \mathbf{Nat}(n{-}1)) \times (\forall m.\, \bullet\, \mathbf{Nat}(m) \to \bullet\, \mathbf{Nat}(n{+}m))).$$

Then, the specifications of **add** and **add**$'$ are now different as follows:

$$\mathbf{add} : \forall n.\, \forall m.\, \mathbf{Nat}(n) \to \bullet\, \mathbf{Nat}(m) \to \bullet\, \mathbf{Nat}(n{+}m)$$
$$\mathbf{add}' : \forall n.\, \forall m.\, \bullet\, \mathbf{Nat}(n) \to \mathbf{Nat}(m) \to \bullet\, \mathbf{Nat}(n{+}m)$$

We can show $\mathbf{s} : \forall n.\, \mathbf{Nat}(n) \to \mathbf{Nat}(n{+}1)$ by deriving $<\mathbf{i_2}\, x, \lambda y.\, \mathbf{add}\, x\, (\mathbf{s}\, y)> : \mathbf{Nat}(n{+}1)$ from $\mathbf{s} : \bullet\forall n.\, \mathbf{Nat}(n) \to \mathbf{Nat}(n{+}1)$ and $x : \mathbf{Nat}(n)$. Obviously,

$$\mathbf{i_2}\, x : (n{+}1 = 0) + (n{+}1 > 0 \wedge \bullet\, \mathbf{Nat}(n{+}1{-}1)).$$

If $y : \bullet\, \mathbf{Nat}(m)$, we get $\mathbf{s}\ y : \bullet\, \mathbf{Nat}(m{+}1)$, and consequently,

$$\mathbf{add}\ x\ (\mathbf{s}\ y) : \bullet \mathbf{Nat}(n{+}1{+}m).$$

We thus get $<\mathbf{i_2}\, x, \lambda y.\, \mathbf{add}\, x\, (\mathbf{s}\, y)> : \mathbf{Nat}(n{+}1)$. Note that, on the other hand, under similar assumptions, we can only get

$$\mathbf{add}'\ x\ (\mathbf{s}'\ y) : \bullet\bullet\, \mathbf{Nat}(n{+}1{+}m),$$

and fail to derive $\mathbf{s}' : \forall n.\, \mathbf{Nat}(n) \to \mathbf{Nat}(n{+}1)$.

7. Concluding Remarks

We have presented a modal typing system with recursive types and shown its soundness with respect to a realizability interpretation and the convergence of well-typed terms according to their types. The decidability questions for type checking, typability and inhabitation of $\lambda\bullet\mu$ types are still open. Although we presented it as a typing system, we do not intend to apply it directly to type systems of programming languages. Since our framework asserts the convergence of derived programs, typing general recursive programs naturally requires some (classical) arithmetics as seen in the case of the 91-function, which would make mechanical type checking impossible. Our goal is to capture a wider range of programs in the proofs-as-programs paradigm and give an axiomatic semantics to them preserving the compositionality of programs. We have seen that our approach is applicable to some interesting programs such as fixed point combinators and objects with binary methods, which have not been captured in the conventional frameworks.

Similar results concerning the existence of fixed points of proper type expressions (Lemma 3.1 in our case) could historically go back to the fixed point theorem of the logic of provability (see [5, 18]). The difference is that our logic is intuitionistic, and fixed points are treated as sets of realizers. Interestingly, by applying $(\preceq)$ to the type $(\bullet X \to X) \to X$ of the fixed point combinators, we can also derive $\bullet(\bullet X \to X) \to \bullet X$, i.e., Löb's axiom schema $\Box(\Box\phi \to \phi) \to \Box\phi$ representing the well-foundedness of the (classical) Kripke frame. It should be observed that $(\Box\phi \to \phi) \to \phi$ is valid based on intuitionistic frames, where $\phi \to \Box\phi$ is valid, if and only if the frame is well-founded.

Our intended semantics of $\lambda\bullet\mu$ suggests that our modal logic could be related to some temporal logic of discrete, linear time with finite past and infinite future, where the modal operator $\bullet$ corresponds to the "previous time", or "yesterday", modality. Gabbay and Hodkinson discussed such a temporal logic and its fixed point operator [12, 16]; however, the relationship with $\lambda\bullet\mu$ is not obvious since their temporal logic is based on classical logic.

Acknowledgments

The author is greatly indebted to Professor Solomon Feferman for the opportunity to develop the main part of this research in a stimulating environment at Stanford University. Thanks are also due to the anonymous referees for their helpful comments on the preliminary version of the present paper.

References

[1] M. Abadi and L. Cardelli. *A theory of objects*. Springer-Verlag, 1996.

[2] R. M. Amadio. Recursion over realizability structure. *Information and Computation*, 91(1):55–85, 1991.

[3] R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.

[4] H. P. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 118–309. Oxford University Press, 1992.

[5] G. Boolos. *The logic of provability*. Cambridge University Press, 1993.

[6] K. B. Bruce, L. Cardelli, and B. C. Pierce. Comparing object encodings. *Information and Computation*, 155:108–133, 1999.

[7] L. Cardelli. Amber. In G. Cousineau, P.-L. Curien, and B. Robinet, editors, *Combinators and functional programming languages*, volume 242 of *Lecture Notes in Computer Science*, pages 21–47. Springer-Verlag, 1986.

[8] F. Cardone and M. Coppo. Type inference with recursive types: syntax and semantics. *Information and Computation*, 92(1):48–80, 1991.

[9] R. L. Constable, S. Allen, H. Bromely, W. Cleveland, et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.

[10] R. L. Constable and N. P. Mendler. Recursive definitions in type theory. In *Logics of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 61–78. Springer-Verlag, 1985.

[11] R. L. Constable and S. F. Smith. Partial objects in constructive type theory. In *Proceedings of the 2nd IEEE Symposium on Logic in Computer Science*, pages 183–193. IEEE Computer Society Press, 1987.

[12] D. M. Gabbay. The declarative past and imperative future. In *Temporal logic in specification*, volume 398 of *Lecture Notes in Computer Science*, pages 409–448. Springer-Verlag, 1989.

[13] Y. Gurevich and S. Shelah. Fixed-point extensions of first-order logic. *Annals of Pure and Applied Logic*, 32(3):265–280, 1986.

[14] S. Hayashi and H. Nakano. *PX: A Computational Logic*. The MIT Press, 1988.

[15] R. Hindley. The completeness theorem for typing λ-terms. *Theoretical Computer Science*, 22:1–17, 1983.

[16] I. M. Hodkinson. On Gabbay's temporal fixed point operator. *Theoretical Computer Science*, 139:1–25, 1995.

[17] W. A. Howard. The formulae-as-types notion of construction. In R. J. Hindley and J. P. Seldin, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 480–490. Academic Press, 1980.

[18] G. Japaridze and D. de Jongh. The logic of provability. In *Handbook of proof theory*, pages 475–546. North Holland, 1998.

[19] S. Kobayashi and M. Tatsuta. Realizability interpretation of generalized inductive definitions. *Theoretical Computer Science*, 131(1):121–138, 1994.

[20] D. C. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27(3):333–354, 1983.

[21] D. Leivant. Typing and computational properties of lambda expressions. *Theoretical Computer Science*, 44(1):51–68, 1986.

[22] D. B. MacQueen, G. D. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. *Information and Computation*, 71:95–130, 1986.

[23] C. Paulin-Mohring. Extracting F_ω's programs from proofs in the calculus of constructions. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*, pages 89–104, 1989.

[24] V. R. Pratt. A decidable μ-calculus (preliminary report). In *Proceedings of the 22nd IEEE Symposium on Foundation of Computer Science*, pages 421–427, 1981.

[25] D. S. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science*, 121:411–440, 1993.

[26] M. Tatsuta. Realizability interpretation of coinductive definitions and program synthesis with streams. *Theoretical Computer Science*, 122:119–136, 1994.