Contractive Signatures with Recursive Types, Type Parameters, and Abstract Types*

Hyeonseung Im[1], Keiko Nakata[2], and Sungwoo Park[3]

[1] LRI - Université Paris-Sud 11, Orsay, France
[2] Institute of Cybernetics at Tallinn University of Technology, Estonia
[3] Pohang University of Science and Technology, Republic of Korea

Abstract. Although theories of equivalence or subtyping for recursive types have been extensively investigated, sophisticated interaction between recursive types and abstract types has gained little attention. The key idea behind type theories for recursive types is to use syntactic contractiveness, meaning every μ-bound variable occurs only under a type constructor such as $\to$ or $*$. This syntactic contractiveness guarantees the existence of the unique solution of recursive equations and thus has been considered necessary for designing a sound theory for recursive types. However, in an advanced type system, such as OCaml, with recursive types, type parameters, and abstract types, we cannot easily define the syntactic contractiveness of types. In this paper, we investigate a sound type system for recursive types, type parameters, and abstract types. In particular, we develop a new semantic notion of contractiveness for types and signatures using mixed induction and coinduction, and show that our type system is sound with respect to the standard call-by-value operational semantics, which eliminates signature sealings. Moreover we show that while non-contractive types in signatures lead to unsoundness of the type system, they may be allowed in modules. We have also formalized the whole system and its type soundness proof in Coq.

1 Introduction

Recursive types are widely used features in most programming languages and the key constructs to exploit recursively defined data structures such as lists and trees. In type theory, there are two ways to exploit recursive types, namely by using the *iso-recursive* or *equi-recursive* formulation.

In the iso-recursive formulation, a recursive type $\mu X.\tau$ is considered isomorphic but not equal to its one-step unfolding $\{X \mapsto \mu X.\tau\}\tau$. Correspondingly the term language provides built-in coercion functions called fold and unfold, witnessing this isomorphism.

$$\begin{aligned} \mathsf{fold} &: \{X \mapsto \mu X.\tau\}\tau \to \mu X.\tau \\ \mathsf{unfold} &: \mu X.\tau \to \{X \mapsto \mu X.\tau\}\tau \end{aligned}$$

* An expanded version of this paper, containing detailed proofs and omitted definitions, and the Coq development are available at `http://toccata.lri.fr/~im`.

Although in the iso-recursive formulation programs have to be annotated with fold and unfold coercion functions, this formulation usually simplifies typechecking in the presence of recursive types. For example, datatypes in SML [11] are a special form of iso-recursive types.

In contrast, the equi-recursive formulation defines a recursive type $\mu X.\tau$ to be equal to its one-step unfolding $\{X \mapsto \mu X.\tau\}\tau$ and does not require any coercion functions as in the iso-recursive formulation. For example, polymorphic variant and object types as implemented in OCaml [1] require structural recursive types and thus equi-recursive types. To use equi-recursive types, it suffices to add either of the two rules below into the type system:

$$\frac{\Gamma \vdash e : \tau \quad \tau \equiv \sigma}{\Gamma \vdash e : \sigma}\ \textsf{typ-eq} \qquad \frac{\Gamma \vdash e : \tau \quad \tau \leq \sigma}{\Gamma \vdash e : \sigma}\ \textsf{typ-sub}$$

Here the rule typ-eq exploits the equivalence relation $\equiv$ between recursive types, and the rule typ-sub the subtyping relation $\leq$. The equi-recursive formulation makes it easier to write programs using recursive types, but it raises a tricky problem in typechecking: we need to decide when two recursive types are in the equivalence or subtyping relation. In response, several authors have investigated theories of equivalence or subtyping for equi-recursive types [2, 3, 6, 18].

The key idea behind type theories for equi-recursive types is to use syntactic contractiveness [9], meaning that given a recursive type $\mu X.\tau$, the use of the recursion variable X in τ must occur under a type constructor such as $\to$ or $*$. In other words, non-contractive types such as $\mu X.X$ and $\mathsf{unit} * \mu X.X$ are rejected at the syntax level. For example, most previous work considers variants of the following type language:

$$\tau, \sigma \quad ::= \quad X \mid \tau \to \sigma \mid \mu X.(\tau \to \sigma)$$

The main reason for employing this syntactic contractiveness is to guarantee the existence of the unique solution of recursive equations introduced by equi-recursive types and obtain a sound type theory.

However, in an advanced type system, such as OCaml, with recursive types, type parameters, abstract types, and modules, we cannot easily define the syntactic contractiveness of types. To illustrate, consider the following code fragment which is allowed in OCaml using the "-rectypes" option. The "-rectypes" option allows arbitrary equi-recursive types.

```
module M : T = struct          module type T = sig
  type 'a t = 'a                 type 'a t
  type s = int and u = bool      type s = s t and u = u t
  let f x = x                    val f : int -> s
  let g x = x                    val g : u -> bool
end                            end
let h x = M.g (M.f x)
let y = h 3                    (* run-time error *)
```

Under the usual interpretation of ML signature sealings, the module `M` correctly implements, or satisfies, the signature `T`. Moreover the types `s` and `u` in `T` are

considered contractive in OCaml since the type cycles are guarded by the parameterized abstract type `t` in `T`; hence the signature `T` is well-formed. Furthermore the types `s` and `u` in `T` are structurally equivalent, and thus the values `h` and `y` are well-formed with types `int -> bool` and `bool`, respectively. At run-time, however, the evaluation of `y`, *i.e.,* `h 3`, leads to an unknown constructor `3` of type `bool`, breaking type soundness.[4]

In this paper, we investigate a type system for equi-recursive types, type parameters, and abstract types. In our system, recursive types may be declared by using type definitions of the form `type` $\alpha\ t = \tau$ where both the type parameter α and the recursive type t may appear in the type τ.[5] Abstract types may be declared by using the usual ML-style signature sealing operations (Section 2.1). For this system, we develop a new notion of semantic contractiveness for types and signatures using mixed induction and coinduction (Section 2.3). Our semantic contractiveness determines the types `s` and `u` in the signature `T` above to be non-contractive, and our type system rejects `T`. We then show that our type system with semantic contractiveness is sound with respect to the standard call-by-value operational semantics, which eliminates signature sealings (Section 2.4).

Another notable result is that even in the presence of non-contractive types in modules, we can develop a sound type system where well-typed programs cannot go wrong. This is particularly important since our type soundness result may give a strong hint about the soundness of OCaml, which allows us to define non-contractive types using recursive modules and signature sealings.

Our contributions are summarized as follows.

- To our knowledge, we are the first to consider a type system for equi-recursive types, type parameters, and abstract types, and define a type sound semantic notion of contractiveness.
- Since the OCaml type system allows both recursive types and abstract types, and non-contractive types in modules, our type soundness result gives a strong hint about how to establish the soundness of OCaml.
- We have formalized the whole system and its type soundness proof in Coq version 8.4. Our formalization extensively uses mixed induction and coinduction, so it may act as a good reference for using mixed induction and coinduction in Coq.

The remainder of the paper is organized as follows. Section 2 presents a type system for recursive types, type parameters, and abstract types. In particular, we consider a simple module system with a signature sealing operation and define a structural type equivalence and semantic contractiveness using mixed induction and coinduction. Section 3 discusses Coq mechanization and an algorithmic type equivalence and contractiveness. Section 4 discusses related work and Section 5 concludes.

[4] We discovered the above bug together with Jacques Garrigue and it has been fixed in the development version of OCaml (available in the OCaml svn repository).

[5] We do not use the usual μ-notation because encoding mutually recursive type definitions into μ-types requires type-level pairs and projection, complicating the theory. Moreover the use of type definitions better reflects the OCaml implementation.

Syntax

$$
\begin{array}{lrll}
\text{types} & \tau, \sigma & ::= & \mathsf{unit} \mid \alpha \mid \tau \to \sigma \mid \tau_1 * \tau_2 \mid \tau\ t \\
\text{expressions} & e & ::= & () \mid x \mid \lambda x : \tau.\ e \mid e_1\ e_2 \mid (e_1, e_2) \mid \pi_i(e) \mid \mathsf{fix}\ x : \tau.\ e \mid l \\
\text{specifications} & D & ::= & \mathsf{type}\ \alpha\ t \mid \mathsf{type}\ \alpha\ t = \tau \mid \mathsf{val}\ l : \tau \\
\text{definitions} & d_\tau & ::= & \mathsf{type}\ \alpha\ t = \tau \qquad \text{(type definitions)} \\
 & d_e & ::= & \mathsf{let}\ l = e \qquad \text{(value definitions)} \\
\text{signatures} & S & ::= & \cdot \mid D, S \\
\text{modules} & M & ::= & (\overline{d_\tau}, \overline{d_e}) \\
\text{programs} & P & ::= & (M, S, e) \mid (M, e)
\end{array}
$$

Well-formed types $\boxed{S; \Sigma \vdash \tau\ \mathsf{type}}$

type variable sets $\Sigma ::= \cdot \mid \{\alpha\}$

Standard well-formedness rules for types are omitted.

$$\frac{S \ni \mathsf{type}\ \alpha\ t \quad S; \Sigma \vdash \tau\ \mathsf{type}}{S; \Sigma \vdash \tau\ t\ \mathsf{type}}\ \textsf{wft-abs} \qquad \frac{S \ni \mathsf{type}\ \alpha\ t = \sigma \quad S; \Sigma \vdash \tau\ \mathsf{type}}{S; \Sigma \vdash \tau\ t\ \mathsf{type}}\ \textsf{wft-app}$$

Well-formed specifications and signatures $\boxed{S \vdash D\ \mathsf{ok}}$ $\boxed{S\ \mathsf{ok}}$

$$\frac{}{S \vdash \mathsf{type}\ \alpha\ t\ \mathsf{ok}}\ \textsf{wfs-abs} \qquad \frac{S; \{\alpha\} \vdash \tau\ \mathsf{type}}{S \vdash \mathsf{type}\ \alpha\ t = \tau\ \mathsf{ok}}\ \textsf{wfs-type} \qquad \frac{S; \cdot \vdash \tau\ \mathsf{type}}{S \vdash \mathsf{val}\ l : \tau\ \mathsf{ok}}\ \textsf{wfs-val}$$

$$\frac{\mathrm{BN}(S)\ \text{distinct} \quad \forall D \in S,\ S \vdash D\ \mathsf{ok}}{S\ \mathsf{ok}}\ \textsf{wf-sig}$$

Fig. 1. Syntax and well-formedness

2 A Type System $\lambda^{\mathsf{rec}}_{\mathsf{abs}}$

This section presents a type system $\lambda^{\mathsf{rec}}_{\mathsf{abs}}$ for recursive types, type parameters, and abstract types, permitting non-contractive types in modules. Section 2.1 presents the syntax and inference rules for well-formedness. Section 2.2 defines a structural type equivalence and Section 2.3 defines a semantic contractiveness using mixed induction and coinduction. Finally Section 2.4 presents a type soundness result with respect to the standard call-by-value operational semantics.

2.1 Syntax and Well-formedness

Figure 1 shows the syntax for $\lambda^{\mathsf{rec}}_{\mathsf{abs}}$ and inference rules for well-formedness. We use meta-variables α, β for type variables, s, t, u for type names, x, y, z for value variables, and l for value names. Both types and expressions are defined in the standard way as in the simply-typed λ-calculus except that they may refer to type definitions and value definitions in modules. For simplicity, we include simple signatures and modules only, which suffice to introduce abstract types, and exclude nested modules and functors. Abstract types are then introduced by a program of the form (M, S, e), called a signature sealing, which hides the implementation details of the module M behind the signature S.

Recursive types are introduced by type definitions of the form $\mathsf{type}\ \alpha\ t = \tau$ where τ may refer to the name t with no restriction. For example, we may define recursive types such as $\mathsf{type}\ \alpha\ t = \alpha\ t \to \alpha\ t$ and $\mathsf{type}\ \alpha\ t = \alpha\ t \to (\alpha * \alpha)\ t$. We also permit a non-contractive type definition such as $\mathsf{type}\ \alpha\ t = \alpha\ t$ in a module but reject a non-contractive type specification in a signature, since the latter breaks type soundness. Any sequence $\overline{d_\tau}$ of type definitions (or type specifications) in modules (or signatures) may be mutually recursive, whereas no sequence $\overline{d_e}$ of value definitions are mutually recursive. The main reason for this design choice is that our focus in this paper is to investigate the interaction between non-contractive recursive types and abstract types. Moreover, to simplify the discussion, we consider only those type constructors with a single parameter. We can easily add into the system nullary or multi-parameter type constructors.

As for well-formedness of types, we use a judgment $S; \Sigma \vdash \tau\ \mathsf{type}$ to mean that type τ is well-formed under context (S, Σ). Here we use a type variable set Σ to denote either an empty set or a singleton set. We also use judgments $S \vdash D\ \mathsf{ok}$ and $S\ \mathsf{ok}$ to mean that specification D and signature S are well-formed, respectively. Most of the rules are standard, and we only remark that a signature S is well-formed only if all bound type names in S are distinct from each other and each type definition is well-formed under S (rule wf-sig). These well-formedness conditions for signatures allow us to define arbitrarily mutually recursive type definitions. In the remainder of the paper, we assume that every type and signature that we mention is well-formed, without explicitly saying so.

2.2 Type Equivalence

In this section, we define a type equivalence relation in terms of unfolding type definitions. The usual β-equivalence is then embedded into our type equivalence relation by means of unfolding. We use a judgment $S \vdash \tau \rightharpoonup \sigma$ to mean that type τ unfolds into σ by expanding a type name in τ into its definition under S. The rule unfold below, which is the only rule for unfolding type definitions, implements this idea and given a type application $\tau\ t$, it replaces type variable α with argument τ in definition σ of t.

$$\frac{S \ni \mathsf{type}\ \alpha\ t = \sigma}{S \vdash \tau\ t \rightharpoonup \{\alpha \mapsto \tau\}\sigma}\ \mathsf{unfold}$$

We write $S \vdash \tau \rightharpoonup^* \sigma$ for the reflexive and transitive closure of unfolding. We say that a type τ is *vacuous* under signature S, if $\forall \tau',\ S \vdash \tau \rightharpoonup^* \tau'$ implies $\exists \tau'',\ S \vdash \tau' \rightharpoonup \tau''$. In other words, vacuous types are those types that allow us to unfold type definitions infinitely using the rule unfold.

The type equivalence relation is defined by nesting induction into coinduction. In order to structurally compare types for their equivalence, we should be able to check equivalence for vacuous types. While using induction to compare structures of types, we use coinduction to compare infinite unfoldings of types and thus to check equivalence for vacuous types. Figure 2 shows inference rules for type equivalence, defined using the rule unfold. We use a judgment

Coinductive type equivalence $\boxed{S; \Sigma \vdash \tau_1 \equiv \tau_2}$

$$\frac{S; \Sigma \vdash \tau \overset{\equiv}{=} \sigma}{S; \Sigma \vdash \tau \equiv \sigma}\ \textsf{eq-ind} \qquad \frac{S \vdash \tau \rightharpoonup \tau' \quad S \vdash \sigma \rightharpoonup \sigma' \quad S; \Sigma \vdash \tau' \equiv \sigma'}{S; \Sigma \vdash \tau \equiv \sigma}\ \textsf{eq-coind}$$

Inductive type equivalence $\boxed{S; \Sigma \vdash \tau_1 \overset{R}{=} \tau_2}$

$$\frac{}{S; \Sigma \vdash \mathsf{unit} \overset{R}{=} \mathsf{unit}}\ \textsf{eq-unit} \qquad \frac{}{S; \{\alpha\} \vdash \alpha \overset{R}{=} \alpha}\ \textsf{eq-var} \qquad \frac{S; \Sigma \vdash \tau_i\ R\ \sigma_i \quad (i = 1, 2)}{S; \Sigma \vdash \tau_1 \to \tau_2 \overset{R}{=} \sigma_1 \to \sigma_2}\ \textsf{eq-fun}$$

$$\frac{S; \Sigma \vdash \tau_i\ R\ \sigma_i \quad (i = 1, 2)}{S; \Sigma \vdash \tau_1 * \tau_2 \overset{R}{=} \sigma_1 * \sigma_2}\ \textsf{eq-prod} \qquad \frac{S \ni \mathsf{type}\ \alpha\ t \quad S; \Sigma \vdash \tau\ R\ \sigma}{S; \Sigma \vdash \tau\ t \overset{R}{=} \sigma\ t}\ \textsf{eq-abs}$$

$$\frac{S \vdash \tau \rightharpoonup \tau' \quad S; \Sigma \vdash \tau' \overset{R}{=} \sigma}{S; \Sigma \vdash \tau \overset{R}{=} \sigma}\ \textsf{eq-lunfold} \qquad \frac{S \vdash \sigma \rightharpoonup \sigma' \quad S; \Sigma \vdash \tau \overset{R}{=} \sigma'}{S; \Sigma \vdash \tau \overset{R}{=} \sigma}\ \textsf{eq-runfold}$$

Fig. 2. Type equivalence

$S; \Sigma \vdash \tau_1 \equiv \tau_2$ to mean that τ_1 and τ_2 are coinductively equivalent under context (S, Σ) and a judgment $S; \Sigma \vdash \tau_1 \overset{R}{=} \tau_2$ to mean that τ_1 and τ_2 are inductively equivalent. Note that the inductive equivalence relation $\overset{R}{=}$ is parameterized over a relation R, which is instantiated with the coinductive equivalence relation $\equiv$ in the rule eq-ind. This way, we nest the inductive equivalence relation into the coinductive equivalence relation[6]. We use a double horizontal line for a coinductive rule and a single horizontal line for an inductive rule.

The rule eq-coind is a coinductive rule for checking equivalence between vacuous types. To show that two vacuous types τ and σ are equivalent, that is, $S; \Sigma \vdash \tau \equiv \sigma$, we repeatedly apply the rule eq-coind. When we get the very same proposition to be proved in the premise, the proof is completed by coinduction. Notably vacuous types are only equivalent to vacuous types. As for equivalence for types other than vacuous types, we use the rule eq-ind, which nests induction into coinduction, to compare their structures.

The inductive type equivalence compares structures of types. Given a pair of types, we apply the rule eq-lunfold or eq-runfold a finite number of times, unfolding type definitions, until we get a pair of the unit type, type variables, function types, or product types. Then we structurally compare them. Note that the rules eq-lunfold and eq-runfold are the only rules where induction plays a role. It is crucial that these rules are defined inductively; if we allow them to be used coinductively, a vacuous type becomes equivalent to any type. The rules eq-unit for the unit type and eq-var for type variables are standard. The rules eq-fun for function types, eq-prod for product types, and eq-abs for abstract types are where the inductive equivalence goes back to the coinductive equivalence.

[6] A definition of the form $\nu X.F(X, \mu Y.G(X, Y))$.

Contractive types and signatures $\boxed{S \Downarrow \tau}$ $\boxed{S \downarrow_C \tau}$ $\boxed{S \Downarrow}$

$$\frac{S \downarrow_{\Downarrow} \tau}{S \Downarrow \tau}\ \textsf{c-coind} \qquad \frac{}{S \downarrow_C \mathsf{unit}}\ \textsf{c-unit} \qquad \frac{}{S \downarrow_C \alpha}\ \textsf{c-var} \qquad \frac{(S, \tau_i) \in C \quad (i = 1, 2)}{S \downarrow_C \tau_1 \to \tau_2}\ \textsf{c-fun}$$

$$\frac{(S, \tau_i) \in C \quad (i = 1, 2)}{S \downarrow_C \tau_1 * \tau_2}\ \textsf{c-prod} \qquad \frac{S \ni \mathsf{type}\ \alpha\ t \quad S \downarrow_C \tau}{S \downarrow_C \tau\ t}\ \textsf{c-abs} \qquad \frac{S \vdash \tau \rightharpoonup \sigma \quad S \downarrow_C \sigma}{S \downarrow_C \tau}\ \textsf{c-type}$$

$$\frac{\forall(\mathsf{type}\ \alpha\ t = \tau) \in S,\ S \Downarrow \tau}{S \Downarrow}\ \textsf{c-sig}$$

Fig. 3. Contractive types and signatures

With this definition of type equivalence, for example, now we prove that the types `s` and `u` in the signature `T` in the introduction are equivalent as follows:

$$\dfrac{\dfrac{\mathtt{T} \ni \mathtt{type\ 'a\ t} \qquad \overline{\mathtt{T}; \cdot \vdash \mathtt{s} \equiv \mathtt{u}}\ \text{coinduction hypothesis}}{\mathtt{T}; \cdot \vdash \mathtt{s\ t} \mathrel{\overline{\equiv}} \mathtt{u\ t}}\ \textsf{eq-abs}}{\mathtt{T}; \cdot \vdash \mathtt{s} \equiv \mathtt{u}}\ \textsf{eq-ind}, \textsf{eq-lunfold}, \textsf{eq-runfold}$$

Our type equivalence is indeed an equivalence relation, *i.e.*, reflexive, symmetric, and transitive (see the expanded version for the proof).

2.3 Contractive Types and Signatures

Given a program (M, S, e), we restrict every type τ in a type specification $\mathsf{type}\ \alpha\ t = \tau$ in S to be contractive to obtain type soundness as illustrated in the introduction. Intuitively a type is contractive if any sequence of its unfolding eventually produces a type constructor such as unit, α, $\to$, or $*$. A subtle case is a type application $\tau\ t$ where t is an abstract type: we require that τ be contractive. For instance, assuming t is an abstract type, $\mathsf{type}\ \alpha\ s = (\alpha\ s * \alpha\ s)\ t$ and $\mathsf{type}\ \alpha\ s = ((\alpha * \alpha)\ t)\ t$ are contractive, but $\mathsf{type}\ \alpha\ s = (\alpha\ s)\ t$ is not. This way, we avoid the possibility of a type specification $\mathsf{type}\ \alpha\ s = \tau$ in a signature to degenerate into $\mathsf{type}\ \alpha\ s = \alpha\ s$ during subtyping.

Figure 3 shows inference rules for contractive types and signatures. We use two judgments $S \Downarrow \tau$ and $S \downarrow_C \tau$ to define contractive types: the former to define coinductive contractiveness and the latter inductive contractiveness. The basic idea of using nested induction into coinduction is the same as for type equivalence. Note that the rule c-coind is the only coinductive rule for checking contractiveness of a type, which nests induction into coinduction.

The inductive contractiveness is defined using six rules: two axioms, two rules going back to the coinductive contractiveness, and two inductive rules for type applications. The unit type and type variables are by definition inductively contractive (rules c-unit and c-var). In the rules c-fun and c-prod, a function type $\tau_1 \to \tau_2$ and a product type $\tau_1 * \tau_2$ are inductively contractive under signature S if each component τ_i and S are related by C, which is instantiated with the

coinductive contractiveness $\Downarrow$ in the rule c-coind. The rules c-abs and c-type are where induction plays a role. An abstract type $\tau\ t$ is inductively contractive if so is its argument τ (rule c-abs). Finally a type τ is inductively contractive if so is its unfolding σ (rule c-type).

A signature S is contractive, denoted by $S \Downarrow$, if for every type specification in S, its right hand side is contractive under S (rule c-sig). With this definition of contractiveness, for example, now the signature `T` in the introduction is not contractive because the type `s t` in the type specification `type s = s t` (or `u t` in `type u = u t`) cannot be proved to be contractive:

$$\dfrac{\dfrac{\mathtt{T} \ni \mathtt{type\ 'a\ t} \qquad \dfrac{\mathtt{T} \vdash \mathtt{s} \rightharpoonup \mathtt{s\ t}}{\mathtt{T} \downarrow_{\Downarrow} \mathtt{s\ t}}\ \textsf{c-type} \qquad \dfrac{\text{an infinite derivation}}{\mathtt{T} \downarrow_{\Downarrow} \mathtt{s}}}{\mathtt{T} \downarrow_{\Downarrow} \mathtt{s}}\ \textsf{c-abs}}{\mathtt{T} \Downarrow \mathtt{s\ t}}\ \textsf{c-coind, c-abs}$$

2.4 Type Soundness

In this section, we prove type soundness of $\lambda^{\mathsf{rec}}_{\mathsf{abs}}$ by the usual progress and preservation properties (Theorems 1 and 4). First we give an overview of typing rules and reduction rules in Figure 4 (where standard typing rules and reduction rules for expressions are omitted). Most of the typing rules are standard, and we only remark that a sealed program (M, S, e) is well-typed if the module M has some signature S', S is contractive, S' is a sub-signature of S, and e is well-typed under the sealed signature S (rule typ-prog-seal). Here we use a subtyping judgment $S' \leq S$, which is defined as for subtyping for record types, where width subtyping is allowed: S' may include more specifications than S. Furthermore any type specification is considered to be a sub-specification of an abstract type specification with the same bound name.

Reduction rules are also standard. Given a sealed program (M, S, e), we first remove the sealed signature. Then we evaluate the module M to a module value V, by sequentially evaluating the value definitions $\overline{d_e}$ in M to definition values $\overline{d_v}$. Finally using the module value V we evaluate the expression e to a value v, obtaining a program value (V, v).

Theorems 1 and 4 below now prove type soundness of $\lambda^{\mathsf{rec}}_{\mathsf{abs}}$. For the progress theorem, we also prove classification, inversion, and canonical forms lemmas as usual. In particular, the classification lemma ensures that we cannot prove equivalence of types with different constructors even in the presence of non-contractive types in modules. For example, $S; \Sigma \vdash \tau_1 * \tau_2 \not\equiv \sigma_1 \to \sigma_2$ for any types $\tau_1, \tau_2, \sigma_1, \sigma_2$.

Theorem 1 (Progress).

(1) If $\vdash (\overline{d_\tau}, \overline{d_v}) : S$ *and* $S; \cdot \vdash e : \tau$*, then either* e *is a value or* $\exists e',\ \overline{d_v} \vdash e \longmapsto e'$*.*
(2) If $\vdash M : S$*, then either* M *is a module value or* $\exists M',\ M \longmapsto M'$*.*
(3) If $\vdash P : (S, \tau)$*, then either* P *is a program value or* $\exists P',\ P \longmapsto P'$*.*

Well-typed definitions and modules $\boxed{S \vdash \overline{d_e} : S_e}$ $\boxed{\vdash M : S}$

$$\frac{}{S \vdash \cdot : \cdot}\ \textsf{typ-emp} \qquad \frac{S; \cdot \vdash e : \tau \quad S,\ \textsf{val}\ l : \tau \vdash \overline{d_e} : S_e}{S \vdash (\textsf{let}\ l = e,\ \overline{d_e}) : (\textsf{val}\ l : \tau,\ S_e)}\ \textsf{typ-val}$$

$$\frac{\overline{d_\tau}\ \textsf{ok} \quad \overline{d_\tau} \vdash \overline{d_e} : S_e \quad \mathrm{BN}(\overline{d_e})\ \text{distinct}}{\vdash (\overline{d_\tau}, \overline{d_e}) : (\overline{d_\tau}, S_e)}\ \textsf{typ-mod}$$

Well-typed programs $\boxed{\vdash P : (S, \tau)}$

$$\frac{\vdash M : S' \quad S \Downarrow \quad S' \leq S \quad S; \cdot \vdash e : \tau}{\vdash (M, S, e) : (S, \tau)}\ \textsf{typ-prog-seal} \qquad \frac{\vdash M : S \quad S; \cdot \vdash e : \tau}{\vdash (M, e) : (S, \tau)}\ \textsf{typ-prog}$$

Reduction rules

$$\begin{array}{lll} \text{values} & v & ::= () \mid \lambda x : \tau.\ e \mid (v_1, v_2) \\ \text{definition values} & d_v & ::= \textsf{let}\ l = v \\ \text{module values} & V & ::= (\overline{d_\tau}, \overline{d_v}) \\ \text{program values} & P_v & ::= (V, v) \end{array}$$

$$\frac{}{(M, S, e) \longmapsto (M, e)}\ \textsf{red-p-seal}$$

$$\frac{M \longmapsto M'}{(M, e) \longmapsto (M', e)}\ \textsf{red-p-mod} \qquad \frac{\overline{d_v} \vdash e \longmapsto e'}{(\overline{d_\tau}, \overline{d_v}, e) \longmapsto (\overline{d_\tau}, \overline{d_v}, e')}\ \textsf{red-p-exp}$$

$$\frac{\overline{d_v} \vdash e \longmapsto e'}{(\overline{d_\tau}, \overline{d_v},\ \textsf{let}\ l = e,\ \overline{d_e}) \longmapsto (\overline{d_\tau}, \overline{d_v},\ \textsf{let}\ l = e',\ \overline{d_e})}\ \textsf{red-mod} \qquad \frac{\overline{d_v} \ni \textsf{let}\ l = v}{\overline{d_v} \vdash l \longmapsto v}\ \textsf{red-name}$$

Fig. 4. Typing rules and reduction rules

The key lemma for the preservation theorem is that type equivalence is preserved by subtyping. In the lemma below, the signature S_2 being contractive is crucial. For example, assuming `S` is the inferred signature of the module `M` in the introduction, although $\mathtt{S} \leq \mathtt{T}$ and $\mathtt{T}; \cdot \vdash \mathtt{s} \equiv \mathtt{t}$, we have $\mathtt{S}; \cdot \vdash \mathtt{s} \not\equiv \mathtt{t}$.

Lemma 2. *If $S_1 \leq S_2$, $S_2 \Downarrow$, and $S_2; \Sigma \vdash \tau \equiv \sigma$, then $S_1; \Sigma \vdash \tau \equiv \sigma$.*

Now using Lemma 2, we show that if a sealed program (M, S, e) is well-typed, the program (M, e) where the sealed signature S is eliminated is also well-typed (Lemma 3), which proves the most difficult case (4) of Theorem 4. We then prove other cases of Theorem 4 as usual using induction and case analysis.

Lemma 3 (Contractive signature elimination). *If $\vdash (M, S, e) : (S, \tau)$, then there exists S' such that $\vdash (M, e) : (S', \tau)$ and $S' \leq S$.*

Theorem 4 (Preservation).

(1) If $\vdash (\overline{d_\tau}, \overline{d_v}) : S$, $S; \cdot \vdash e : \tau$, and $\overline{d_v} \vdash e \longmapsto e'$, then $S; \cdot \vdash e' : \tau$.
(2) If $\vdash M : S$ and $M \longmapsto M'$, then $\vdash M' : S$.
(3) If $P = (M, e)$, $\vdash P : (S, \tau)$ and $P \longmapsto P'$, then $\vdash P' : (S, \tau)$.
(4) If $\vdash (M, S, e) : (S, \tau)$, then $\exists S'$ such that $\vdash M : S'$, $S' \leq S$, and $S'; \cdot \vdash e : \tau$.

3 Discussion

3.1 Coq Mechanization

For the Coq mechanization, we use Mendler-style [10] coinductive rules for type equivalence and contractiveness in the style of Nakata and Uustalu [14], instead of the Park-style rules in Figures 2 and 3. The reason is that Coq's syntactic guardedness condition for induction nested into coinduction is too weak to work with the Park-style rules. We cannot construct corecursive functions (coinductive proofs) that we need. For example, to enable Coq's guarded corecursion, we use the following Mendler-style coinductive rule instead of the Park-style rule eq-ind:

$$\frac{R \subseteq \equiv \quad S; \Sigma \vdash \tau \overset{R}{=} \sigma}{S; \Sigma \vdash \tau \equiv \sigma}\ \mathsf{eq\text{-}ind}'$$

The main difference is that we use in the rule eq-ind′ a relation R that is stronger than the coinductive equivalence relation $\equiv$. Hence, to build a coinductive proof, we need to find such a relation R, and in many cases we cannot just use $\equiv$ for R. With this definition, the Park-style rules are derivable.

3.2 Algorithmic Type Equivalence and Contractiveness

Strictly speaking, equality of equi-recursive types with type parameters is decidable [4]. Solomon [17] has shown it to be equivalent to the equivalence problem for deterministic pushdown automata (DPDA), which has been shown decidable by Sénizergues [16]. There is, however, no known practical algorithm for DPDA-equivalence, and it is not known whether there exists any algorithm for unification either, which is required for type inference in the core language.

One possible approach to practical type equivalence would be to reject non-regular recursive types as in OCaml. We can then also algorithmically decide contractiveness of every type in a program by enumerating all the distinct type structures that can be obtained by unfolding each type used in the program and its subterms. Still we need a sound metatheory of non-regular recursive types to prove soundness of an OCaml-style recursive module system, because such types may be hidden behind signature sealings in the presence of recursive modules.

4 Related Work

The literature on subtyping for μ-types (hence without type definitions, type parameters, and abstract types) is abundant. In this setting, contractiveness can be checked syntactically: every μ-bound variable occurs under $\to$ or $*$. We mention three landmark papers. Amadio and Cardelli [2] were the first to give a subtyping algorithm. They define subtyping in three ways, which are proved equivalent: an inclusion between unfoldings of μ-types into infinite trees, a subtyping algorithm, and an inductive axiomatization. Brandt and Henglein [3] give a new inductive axiomatization in which the underlying coinductive nature of

Amadio and Cardelli's system is internalized by allowing, informally speaking, construction of circular proofs. Gapeyev *et al.* [6] is a good self-contained introduction to subtyping for recursive types, including historical notes on theories of recursive types. They define a subtyping relation on contractive μ-types as the greatest fixed point of a suitable generating function.

Danielsson and Altenkirch [5] present an axiomatization of subtyping for μ-types using induction nested into coinduction. They formalized the development in Agda, which supports induction nested into coinduction as a basic form. Komendantsky [8] conducted a similar project in Coq using the Mendler-style coinduction.

Recursive types are indispensable in theories of recursive modules since recursive modules allow us to indirectly introduce recursion in types that span across module boundaries. In this setting, one has to deal with a more expressive language for recursive types, which may include, for instance, higher-order type constructors, type definitions, and abstract types. Montagu and Rémy [12, 13] investigate existential types to model modular type abstraction in the context of a structural type system. They consider its extensions with recursion (*i.e.,* equi-recursive types without type parameters) and higher-order type constructors separately but do not investigate a combination of the two extensions. Crary et al. [4] first propose a type system for recursive modules using an inductive axiomatization of (coinductive) type equivalence for equi-recursive types with higher-order type constructors, type definitions, and abstract types. However, the metatheory of their axiomatization such as type soundness is not investigated. Rossberg and Dreyer [15] use equi-recursive types with inductive type equivalence (*i.e.,* they do not have a rule equivalent to *contract* in [2] to enable coinductive reasoning) to prove soundness of their mixin-style recursive module system. They do not intend to use equi-recursive types for the surface language. Our earlier work [7] on recursive modules considers equi-recursive types with type definitions and abstract types, but without type parameters. There we define a type equivalence relation using weak bisimilarity.

5 Conclusion and Future Work

This paper studies a type system for recursive types, type parameters, and abstract types. In particular, we investigate the interaction between non-contractive types and abstract types, and show that while non-contractive types in signatures lead to unsoundness of the type system, they may be allowed in modules. Our study is mainly motivated by OCaml, which allows us to define both abstract types and equi-recursive types with type parameters (with the "-rectypes" option). To obtain a sound type system, we develop a new notion of semantic contractiveness using mixed induction and coinduction and reject non-contractive types in signatures. We show that our type system is sound with respect to the standard call-by-value operational semantics, which eliminates signature sealings. We have also formalized the whole system and its soundness proof in Coq.

Future work includes extending our type system to the full-scale module system including recursive modules, nested modules, and higher-order functors.

Acknowledgments.

This work was supported by Mid-career Researcher Program through NRF funded by the MEST (2010-0022061). Hyeonseung Im was partially supported by the ANR TYPEX project n. ANR-11-BS02-007_02. Keiko Nakata was supported by the ERDF funded EXCS project, the Estonian Ministry of Education and Research research theme no. 0140007s12, and the Estonian Science Foundation grant no. 9398.

References

1. OCaml. `http://caml.inria.fr/ocaml/`.
2. R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, September 1993.
3. M. Brandt and F. Henglein. Coinductive axiomatization of recursive type equality and subtyping. In *TLCA '97*, pages 63–81. Springer-Verlag, 1997.
4. K. Crary, R. Harper, and S. Puri. What is a recursive module? In *PLDI '99*.
5. N. A. Danielsson and T. Altenkirch. Subtyping, declaratively: an exercise in mixed induction and coinduction. In *MPC '10*, pages 100–118. Springer-Verlag, 2010.
6. V. Gapeyev, M. Y. Levin, and B. C. Pierce. Recursive subtyping revealed. *Journal of Functional Programming*, 12(6):511–548, 2002.
7. H. Im, K. Nakata, J. Garrigue, and S. Park. A syntactic type system for recursive modules. In *OOPSLA '11*.
8. V. Komendantsky. Subtyping by folding an inductive relation into a coinductive one. In *TFP '11*, pages 17–32. Springer-Verlag, 2012.
9. D. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. In *POPL '84*.
10. N. P. Mendler. Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied Logic*, 51(1-2):159–172, 1991.
11. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.
12. B. Montagu. *Programming with first-class modules in a core language with subtyping, singleton kinds and open existential types*. PhD thesis, École Polytechnique, Palaiseau, France, December 2010.
13. B. Montagu and D. Rémy. Modeling abstract types in modules with open existential types. In *POPL '09*.
14. K. Nakata and T. Uustalu. Resumptions, weak bisimilarity and big-step semantics for While with interactive I/O: An exercise in mixed induction-coinduction. In *SOS '10*, pages 57–75, 2010.
15. A. Rossberg and D. Dreyer. Mixin' up the ML module system. To appear in *ACM Transactions on Programming Languages and Systems*, 2013.
16. G. Sénizergues. The equivalence problem for deterministic pushdown automata is decidable. In *ICALP '97*, pages 671–681. Springer-Verlag, 1997.
17. M. Solomon. Type definitions with parameters (extended abstract). In *POPL '78*.
18. C. A. Stone and A. P. Schoonmaker. Equational theories with recursive types. Available at `http://www.cs.hmc.edu/~stone/publications.html`, 2005.