Manifest Contracts with Intersection Types

Yuki Nishida and Atsushi Igarashi[0000−0002−5143−9764]

Graduate School of Informatics, Kyoto University, Kyoto, Japan
{nishida,igarashi}@fos.kuis.kyoto-u.ac.jp

Abstract. We present a *manifest contract system* PCFvΔ_{H} with *intersection types*. A manifest contract system is a typed functional calculus in which software contracts are integrated into a refinement type system and consistency of contracts is checked by combination of compile- and run-time type checking. Intersection types naturally arise when a contract is expressed by a conjunction of smaller contracts. Run-time contract checking for conjunctive higher-order contracts in an untyped language has been studied but our typed setting poses an additional challenge due to the fact that an expression of an intersection type $\tau_1 \wedge \tau_2$ may have to perform different run-time checking whether it is used as τ_1 or τ_2.
We build PCFvΔ_{H} on top of the Δ-calculus, a Church-style intersection type system by Liquori and Stolze. In the Δ-calculus, a canonical expression of an intersection type is a *strong pair*, whose elements are the same expressions except for type annotations. To address the challenge above, we relax strong pairs so that expressions in a pair are the same except for type annotations and casts, which are a construct for run-time checking. We give a formal definition of PCFvΔ_{H} and show its basic properties as a manifest contract system: preservation, progress, and value inversion. Furthermore, we show that run-time checking does not affect essential computation.

1 Introduction

Manifest contract systems [1, 10–13, 15, 19, 24–26, 31], which are typed functional calculi, are one discipline handling *software contracts* [18]. The distinguishing feature of manifest contract systems is that they integrate contracts into a type system and guarantee some sort of satisfiability against contracts in a program as type soundness. Specifically, a contract is embedded into a type by means of *refinement types* of the form $\{x{:}\tau \mid M\}$, which represents the subset of the *underlying type* τ such that the values in the subset satisfy the *predicate* M, which can be an arbitrary Boolean expression in the programming language. Using the refinement types, for example, we can express the contract of a division function, which would say "... the divisor shall not be zero ...", by the type $\texttt{int} \to \{x{:}\texttt{int} \mid x \neq 0\} \to \texttt{int}$. In addition to the refinement types, manifest contract systems are often equipped with *dependent function types* in order to express more detailed contracts. A dependent function type, written $(x{:}\sigma) \to \tau$ in this paper, is a type of a function which takes one argument of the type σ and returns a value of the type τ; the distinguished point from ordinary

function types is that τ can refer to the given argument represented by x. Hence, for example, the type of a division function can be made more specific like $(x{:}\mathtt{int}) \to (y{:}\{x'{:}\mathtt{int} \mid x' \neq 0\}) \to \{z{:}\mathtt{int} \mid x = z \times y\}$. (Here, for simplicity, we ignore the case where devision involves a remainder, though it can be taken account into by writing a more sophisticated predicate.)

A manifest contract system checks a contract dynamically to achieve its goal—as many *correct* programs as possible can be compiled and run; while some studies [16, 23, 27, 28, 30, 33], which also use a refinement type system, check contract satisfaction statically but with false positives and/or restriction on predicates. The checks are done in the form of explicit casts of the form $(M : \sigma \Rightarrow \tau)$; where M is a subject, σ is a source type (namely the type of M), and τ is a target type.[1] A cast checks whether the value of M can have the type τ. If the check fails, the cast throws an uncatchable exception called *blame*, which stands for contract violation. So, the system does not guarantee the absence of contract violations statically, but it guarantees that the result of successful execution satisfies the predicate of a refinement type in the program's type. This property follows subject reduction and a property called *value inversion* [26]—*if a value V has a type $\{x{:}\tau \mid M\}$, then the expression obtained by substituting V for x in M is always evaluated into* `true`.

1.1 Motivation

The motivation of the integration of intersection types is to enrich the expressiveness of contracts by types. It naturally arises when we consider a contract stated in a conjunctive form [3, 9, 14]. Considering parities (even/odd) of integers, for example, we can state a contract of the addition as a conjunctive form; that is

> "An even integer is returned if both given arguments are even integers; **and** an odd integer is returned if the first given argument is even integer and the second given argument is odd integer; **and** ..."

Using intersection types, we can write the contract as the following type.[2]

$$(\mathtt{even} \to \mathtt{even} \to \mathtt{even}) \wedge (\mathtt{even} \to \mathtt{odd} \to \mathtt{odd}) \\ \wedge (\mathtt{odd} \to \mathtt{even} \to \mathtt{odd}) \wedge (\mathtt{odd} \to \mathtt{odd} \to \mathtt{even})$$

In fact, a semantically equivalent contract could be expressed by using dependent function types found in existing systems as follows, where $\mathtt{evenp} := \lambda x{:}\mathtt{nat}.x \bmod 2 = 0$ and $\mathtt{oddp} := \lambda x{:}\mathtt{nat}.x \bmod 2 = 1$.

$$(x{:}\mathtt{nat}) \to (y{:}\mathtt{nat}) \to \{z{:}\mathtt{nat} \mid \mathtt{if}\ \mathtt{evenp}\, x \\ \mathtt{then}\ (\mathtt{if}\ \mathtt{evenp}\, y\ \mathtt{then}\ \mathtt{evenp}\, z\ \mathtt{else}\ \mathtt{oddp}\, z) \\ \mathtt{else}\ (\mathtt{if}\ \mathtt{evenp}\, y\ \mathtt{then}\ \mathtt{oddp}\, z\ \mathtt{else}\ \mathtt{evenp}\, z)\}$$

[1] Many manifest contract systems put a unique label on each cast to distinguish which cast fails, but we omit them for simplicity.

[2] $\mathtt{even} := \{x{:}\mathtt{nat} \mid x \bmod 2 = 0\}$ $\mathtt{odd} := \{x{:}\mathtt{nat} \mid x \bmod 2 = 1\}$

Thus, one might think it is just a matter of taste in how contracts are represented. However, intersection types are more expressive, that is, there are contracts that are hard to express in existing manifest contract systems. Consider the following (a bit contrived) contract for a higher-order function.

$$((\mathtt{int} \to \{x{:}\mathtt{int} \mid x \neq 0\}) \to \{z{:}\mathtt{int} \mid z = 1\}) \wedge ((\mathtt{int} \to \mathtt{int}) \to \{z{:}\mathtt{int} \mid z = 0\})$$

The result type depends on input as the parity contract does. This time, however, it cannot be written with a dependent function type; there is no obvious way to write a predicate corresponding to `evenp` (or `oddp`). Such a predicate must check that a given function returns non-zero for all integers, but this is simply not computable.

1.2 Our Work

We develop a formal calculus PCFvΔ_{H}, a manifest contract system with intersection types. The goal of this paper is to prove its desirable properties: preservation, progress, value inversion; and one that guarantees that the existence of dynamic checking does not change the "essence" of computation.

There are several tasks in constructing a manifest contract system, but a specific challenge for PCFvΔ_{H} arises from the fact—manifest contract systems are intended as an intermediate language for *hybrid type checking* [10]. Firstly, consider the following definition with a parity contract in a surface language.

$$\mathtt{let}\ \mathit{succ}'{:}\mathtt{odd} \to \mathtt{even} = \lambda x.\mathtt{succ}(x).$$

Supposing the primitive operator $\mathtt{succ}(x)$ has the type $\mathtt{nat} \to \mathtt{nat}$, we need to check subtyping relation $\mathtt{odd} <: \mathtt{nat}$ and $\mathtt{nat} <: \mathtt{even}$ to check well-typedness of the definition. As we have mentioned, however, this kind of subtyping checking is undecidable in general. So, (when the checking is impossible) we insert casts to check the contract at run-time and obtain the following compiled definition.

$$\mathtt{let}\ \mathit{succ}'{:}\mathtt{odd} \to \mathtt{even} = \lambda x{:}\mathtt{odd}.(\mathtt{succ}((x : \mathtt{odd} \Rightarrow \mathtt{nat})) : \mathtt{nat} \Rightarrow \mathtt{even}).$$

A problem arises when we consider the following definition equipped with a more complicated parity contract.

$$\mathtt{let}\ \mathit{succ}'{:}(\mathtt{odd} \to \mathtt{even}) \wedge (\mathtt{even} \to \mathtt{odd}) = \lambda x.\mathtt{succ}(x).$$

The problem is that we need to insert different casts into code according to how the code is typed; and one piece of code might be typed in several essentially different ways in an intersection type system since it is a polymorphic type system. For instance, in the example above, $\lambda x{:}\mathtt{odd}.(\mathtt{succ}((x : \mathtt{odd} \Rightarrow \mathtt{nat})) : \mathtt{nat} \Rightarrow \mathtt{even})$ is obtained by cast insertion if the function is typed as $\mathtt{odd} \to \mathtt{even}$; while $\lambda x{:}\mathtt{even}.(\mathtt{succ}((x : \mathtt{even} \Rightarrow \mathtt{nat})) : \mathtt{nat} \Rightarrow \mathtt{odd})$ is obtained when the body is typed as $\mathtt{even} \to \mathtt{odd}$. However, the function must have both types to have the intersection type. It may seem sufficient to just cast the body itself, that is, $((\lambda x{:}\mathtt{nat}.\mathtt{succ}(x)) : \mathtt{nat} \to \mathtt{nat} \Rightarrow (\mathtt{odd} \to \mathtt{even}) \wedge (\mathtt{even} \to \mathtt{odd}))$. However, this just shelves the problem: Intuitively, to check if the subject has the target intersection type, we need to check if the subject has both types in the conjunction. This brings us back to the same original question.

Contributions. Our contributions are summarized as follows:

- we design a manifest contracts calculus with *refinement intersection types* [27, 33], a restricted form of intersection types.
- we formalize the calculus PCFvΔ_{H}; and
- we state and prove type soundness, value inversion, and dynamic soundness.

The whole system including proofs is mechanized with Coq.[3] We use locally nameless representation and cofinite quantification [5] for the mechanization.

Disclaimer. To concentrate on the PCFvΔ_{H}-specific problems, we put the following restrictions for PCFvΔ_{H} in this paper compared to a system one would imagine from the phrase "a manifest contract system with intersection types".

- PCFvΔ_{H} does not support dependent function types. As we will see, PCFvΔ_{H} uses nondeterminism for dynamic checking. The combination of dependent function types and nondeterminism poses a considerable challenge [19].
- We use *refinement intersection types* rather than general ones. Roughly speaking, $\sigma \wedge \tau$ is a refinement intersection type if both σ and τ refine the same type. So, for example, $(\texttt{even} \to \texttt{even}) \wedge (\texttt{odd} \to \texttt{odd})$ is a refinement intersection types since types of both sides refine the same type $\texttt{nat} \to \texttt{nat}$, while $(\texttt{nat} \to \texttt{nat}) \wedge (\texttt{float} \to \texttt{float})$ is not.

2 Overview of Our Language: PCFvΔ_{H}

Our language PCFvΔ_{H} is a call-by-value dialect of PCF [20], extended with intersection types (derived from the Δ-calculus [17]) and manifest contracts (derived from λ_{H} [10, 12]). So, the baseline is that any *valid* PCF program is also a valid PCFvΔ_{H} program; and a PCFvΔ_{H} program should behave as the same way as (call-by-value) PCF. In other words, PCFvΔ_{H} is a conservative extension of call-by-value PCF.

2.1 The Δ-calculus

To address the challenge discussed in Section 1, PCFvΔ_{H} is strongly influenced by the Δ-*calculus* by Liquori and Stolze [17], an intersection type system à la Church. Their novel idea is a new form called *strong pair*, written $\langle M, N \rangle$. It is a kind of pair and used as a constructor for expressions of intersection types. So, using the strong pair, for example, we can write an identity function having type $(\texttt{even} \to \texttt{even}) \wedge (\texttt{odd} \to \texttt{odd})$ as follows.

$$\langle \lambda x{:}\texttt{even}.x, \lambda x{:}\texttt{odd}.x \rangle$$

[3] The Coq scripts are available through the following URL: `https://www.fos.kuis.kyoto-u.ac.jp/~igarashi/papers/manifest-intersection.html`.

Unlike product types, however, M and N in a strong pair cannot be arbitrarily chosen. A strong pair requires that the *essence* of both expressions in a pair be the same. An essence $\wr M \wr$ of a typed expression M is the untyped skeleton of M. For instance, $\wr \lambda x{:}\tau.x \wr = \lambda x.x$. So, the requirement justifies strong pairs as the introduction of intersection types: that is, computation represented by the two expressions is the same and so the system still follows a Curry-style intersection type system. Strong pairs just give a way to annotate expressions with a different type in a different context.

We adapt their idea into PCFvΔ_{H} by letting an essence represent the *contract-irrelevant part* of an expression, rather than an untyped skeleton. For instance, the essence of $\lambda x{:}\mathtt{odd}.(\mathtt{succ}((x : \mathtt{odd} \Rightarrow \mathtt{nat})) : \mathtt{nat} \Rightarrow \mathtt{even})$ is $\lambda x{:}\mathtt{nat}.\mathtt{succ}(x)$ (the erased contract-relevant parts are casts and predicates of refinement types). Now, we can (ideally automatically) compile the *succ*′ definition in Section 1 into the following PCFvΔ_{H} expression.

$$\begin{aligned}&\mathtt{let}\ \mathit{succ}'{:}(\mathtt{odd} \to \mathtt{even}) \wedge (\mathtt{even} \to \mathtt{odd}) = \\ &\qquad \langle \lambda x{:}\mathtt{odd}.(\mathtt{succ}((x : \mathtt{odd} \Rightarrow \mathtt{nat})) : \mathtt{nat} \Rightarrow \mathtt{even}), \\ &\qquad\qquad \lambda x{:}\mathtt{even}.(\mathtt{succ}((x : \mathtt{even} \Rightarrow \mathtt{nat})) : \mathtt{nat} \Rightarrow \mathtt{odd}) \rangle\end{aligned}$$

This strong pair satisfies the condition, that is, both expressions have the same essence.

2.2 Cast Semantics for Intersection Types

Having introduced intersection types, we have to extend the semantics of casts so that they handle contracts written with intersection types. Following Keil and Thiemann [14], who studied intersection (and union) contract checking in the "latent" style [12] for an untyped language, we give the semantics of a cast *to* an intersection type by the following rule:

$$(V : \sigma \Rightarrow \tau_1 \wedge \tau_2) \longrightarrow \langle (V : \sigma \Rightarrow \tau_1), (V : \sigma \Rightarrow \tau_2) \rangle$$

The reduction rule should not be surprising: V has to have both τ_1 and τ_2 and a strong pair introduces an intersection type $\tau_1 \wedge \tau_2$ from τ_1 and τ_2. For the original cast to succeed, both of the split casts have to succeed.

A basic strategy of a cast *from* an intersection type is expressed by the following two rules.

$$\begin{aligned}(V : \sigma_1 \wedge \sigma_2 \Rightarrow \tau) &\longrightarrow (\pi_1(V) : \sigma_1 \Rightarrow \tau) \\ (V : \sigma_1 \wedge \sigma_2 \Rightarrow \tau) &\longrightarrow (\pi_2(V) : \sigma_2 \Rightarrow \tau)\end{aligned}$$

The cast tests whether a nondeterministically chosen element in a (possibly nested) strong pair can be cast to τ.

One problem, however, arises when a function type is involved. Consider the following expression.

$$(\lambda f{:}\mathtt{nat} \to \mathtt{nat}.f\,0 + f\,1)\,M_{\mathrm{cast}}$$

$$
\begin{aligned}
\sigma, \tau &::= \mathtt{nat} \mid \mathtt{bool} \mid \sigma \to \tau \\
L, M, N &::= \mathtt{0} \mid \mathtt{succ}(M) \mid \mathtt{pred}(M) \mid \mathtt{iszero}(M) \mid \mathtt{true} \mid \mathtt{false} \mid \mathtt{if}\, L\, \mathtt{then}\, M\, \mathtt{else}\, N \mid \\
&\quad x \mid M\, N \mid \lambda x{:}\tau.M \mid \mu f{:}\sigma_1 \to \sigma_2.\lambda x{:}\tau.M \\
\overline{n} &::= \mathtt{0} \mid \mathtt{succ}(\overline{n}) \\
V &::= \overline{n} \mid \mathtt{true} \mid \mathtt{false} \mid \lambda x{:}\tau.M \\
\mathcal{E} &::= \mathtt{succ}(\Box) \mid \mathtt{pred}(\Box) \mid \mathtt{iszero}(\Box) \mid \mathtt{if}\, \Box\, \mathtt{then}\, M\, \mathtt{else}\, N \mid \Box\, M \mid V\, \Box
\end{aligned}
$$

Fig. 1. Syntax of PCFv.

where

$$M_{\mathrm{cast}} := (V : (\mathtt{even} \to \mathtt{nat}) \wedge (\mathtt{odd} \to \mathtt{nat}) \Rightarrow \mathtt{nat} \to \mathtt{nat}).$$

V can be used as both $\mathtt{even} \to \mathtt{nat}$ and $\mathtt{odd} \to \mathtt{nat}$. This means V can handle arbitrary natural numbers. Thus, this cast should be valid and evaluation of the expression above should not fail. However, with the reduction rules presented above, evaluation results in blame in both branches: the choice is made before calling $\lambda f : \mathtt{nat} \to \mathtt{nat}.\cdots$, the function being assigned into f only can handle either $\mathtt{even}$ or $\mathtt{odd}$, leading to failure at either $f\,1$ or $f\,0$, respectively.

To solve the problem, we delay a cast into a function type even when the source type is an intersection type. In fact, M_{cast} reduces to a wrapped value V_{cast} below

$$V_{\mathrm{cast}} := \langle\!\langle V : (\mathtt{even} \to \mathtt{nat}) \wedge (\mathtt{odd} \to \mathtt{nat}) \Rightarrow \mathtt{nat} \to \mathtt{nat} \rangle\!\rangle,$$

similarly to higher-order casts [8]. Then, the delayed cast fires when an actual argument is given:

$$
\begin{aligned}
& (\lambda f{:}\mathtt{nat} \to \mathtt{nat}.f\,0 + f\,1)\, M_{\mathrm{cast}} \\
\longrightarrow\ & (\lambda f{:}\mathtt{nat} \to \mathtt{nat}.f\,0 + f\,1)\, V_{\mathrm{cast}} \\
\longrightarrow\ & V_{\mathrm{cast}}\,0 + V_{\mathrm{cast}}\,1 \\
\longrightarrow^*\ & (V : \mathtt{even} \to \mathtt{nat} \Rightarrow \mathtt{nat} \to \mathtt{nat})\,0 + (V : \mathtt{odd} \to \mathtt{nat} \Rightarrow \mathtt{nat} \to \mathtt{nat})\,1 \\
\longrightarrow^*\ & 1
\end{aligned}
$$

3 Formal Systems

In this section, we formally define two languages PCFv and PCFvΔ_{H}, an extension of PCFv as sketched in the last section. PCFv is a call-by-value PCF. We only give operational semantics and omit its type system and a type soundness proof, because we are only interested in how its behavior is related to PCFvΔ_{H}, the main language of this paper.

3.1 PCFv

The syntax of PCFv is shown in Figure 1. Metavariables x, y, z, f, and g range over term variables (f and g are intended for ones bound to functions); σ and

$$\texttt{pred}(\texttt{0}) \longrightarrow_{\mathsf{PCF}} \texttt{0} \qquad \text{(PCF-Pred-Z)}$$

$$\texttt{pred}(\texttt{succ}(\overline{n})) \longrightarrow_{\mathsf{PCF}} \overline{n} \qquad \text{(PCF-Pred)}$$

$$\texttt{iszero}(\texttt{0}) \longrightarrow_{\mathsf{PCF}} \texttt{true} \qquad \text{(PCF-IsZero-T)}$$

$$\texttt{iszero}(\texttt{succ}(\overline{n})) \longrightarrow_{\mathsf{PCF}} \texttt{false} \qquad \text{(PCF-IsZero-F)}$$

$$\texttt{if}\,\texttt{true}\,\texttt{then}\,M\,\texttt{else}\,N \longrightarrow_{\mathsf{PCF}} M \qquad \text{(PCF-If-T)}$$

$$\texttt{if}\,\texttt{false}\,\texttt{then}\,M\,\texttt{else}\,N \longrightarrow_{\mathsf{PCF}} N \qquad \text{(PCF-If-F)}$$

$$(\lambda x{:}\tau.M)\,V \longrightarrow_{\mathsf{PCF}} M[x \mapsto V] \qquad \text{(PCF-Beta)}$$

$$\mu f{:}\sigma_1 \to \sigma_2.\lambda x{:}\tau.M \longrightarrow_{\mathsf{PCF}} (\lambda x{:}\tau.M)[f \mapsto \mu f{:}\sigma_1 \to \sigma_2.\lambda x{:}\tau.M] \qquad \text{(PCF-Fix)}$$

$$\frac{M \longrightarrow_{\mathsf{PCF}} M'}{\mathcal{E}[M] \longrightarrow_{\mathsf{PCF}} \mathcal{E}[M']} \qquad \text{(PCF-Ctx)}$$

Fig. 2. Operational semantics of PCFv.

τ range over types; L, M, and N range over expressions; V ranges over values; and $\mathcal{E}$ ranges over evaluation frames. The definition is fairly standard, except for one point: instead of introducing a constant for the general `fix`-point operator, we introduce a form $\mu f{:}\sigma_1 \to \sigma_2.\lambda x{:}\tau.M$ for recursive functions.

Definition 1 (Bound and free variables). *An occurrence of x in M of $\lambda x{:}\tau.M$ and f in M of $\mu f{:}\sigma_1 \to \sigma_2.\lambda x{:}\tau.M$ is called* bound. *The set of* free variables *in M is the variables of which there are free occurrence in M. We denote the free variables by $\texttt{fv}(M)$.*

Convention. We define α-equivalence in a standard manner and identify α-equivalent expressions.

Definition 2 (Substitution). *Substitution of N for a free variable x in M, written $M[x \mapsto N]$, is defined in a standard capture-avoiding manner.*

Definition 3 (Context application). *Given an evaluation frame $\mathcal{E}$ and an expression M, $\mathcal{E}[M]$ denotes the expression obtained by just replacing the hole $\Box$ in $\mathcal{E}$ with M.*

A small-step operational semantics of PCFv is inductively defined by the rules in Figure 2. Those rules consist of standard (call-by-value) PCF axiom schemes and one rule scheme (PCF-Ctx), which expresses the call-by-value evaluation strategy using the evaluation frames.

3.2 PCFvΔ_{H}

PCFvΔ_{H} is an extension of PCFv. Through abuse of syntax, we use the metavariables of PCFv for PCFvΔ_{H}, though we are dealing with the two different languages.

The syntax of PCFvΔ_{H} is shown in Figure 3. We introduce some more metavariables: I ranges over *interface types*, a subset of types; B ranges over

$$
\begin{array}{rcl}
\sigma, \tau & ::= & \mathtt{nat} \mid \mathtt{bool} \mid \sigma \to \tau \mid \sigma \wedge \tau \mid \{x{:}\tau \mid M\} \\
I & ::= & \sigma \to \tau \mid I_1 \wedge I_2 \\
L, M, N & ::= & \mathtt{0} \mid \mathtt{succ}(M) \mid \mathtt{pred}(M) \mid \mathtt{iszero}(M) \mid \mathtt{true} \mid \mathtt{false} \mid \mathtt{if}\, L\, \mathtt{then}\, M\, \mathtt{else}\, N \mid \\
 & & x \mid M\, N \mid \lambda x{:}\tau.M \mid \mu f{:}I.B \mid \langle M, N \rangle \mid \pi_1(M) \mid \pi_2(M) \mid (M : \sigma \Rightarrow \tau) \mid \\
 & & \langle\!\langle V : \sigma \Rightarrow \tau_1 \to \tau_2 \rangle\!\rangle \mid \langle\!\langle M \mathbin{?} \{x{:}\tau \mid N\} \rangle\!\rangle \mid \langle\!\langle M \Longrightarrow V : \{x{:}\tau \mid N\} \rangle\!\rangle \\
B & ::= & \lambda x{:}\tau.M \mid \langle B_1, B_2 \rangle \\
\overline{n} & ::= & \mathtt{0} \mid \mathtt{succ}(\overline{n}) \\
V & ::= & \overline{n} \mid \mathtt{true} \mid \mathtt{false} \mid \lambda x{:}\tau.M \mid \langle V_1, V_2 \rangle \mid \langle\!\langle V : \sigma \Rightarrow \tau_1 \to \tau_2 \rangle\!\rangle \\
C & ::= & M \mid \mathtt{blame} \\
\mathcal{E} & ::= & \mathtt{succ}(\Box) \mid \mathtt{pred}(\Box) \mid \mathtt{iszero}(\Box) \mid \mathtt{if}\, \Box\, \mathtt{then}\, M\, \mathtt{else}\, N \mid \Box\, M \mid V\, \Box \mid \\
 & & \pi_1(\Box) \mid \pi_2(\Box) \mid (\Box : \sigma \Rightarrow \tau) \mid \langle\!\langle \Box \mathbin{?} \{x{:}\tau \mid M\} \rangle\!\rangle \\
\Gamma & ::= & \emptyset \mid \Gamma, x{:}\tau
\end{array}
$$

Fig. 3. Syntax of PCFvΔ_{H}.

recursion bodies, a subset of expressions; C ranges over *commands*; and Γ ranges over typing contexts. Shaded parts show differences (extensions and modifications) from PCFv. Types are extended with intersection types and refinement types; the restriction that a well-formed intersection type is a refinement intersection type is enforced by the type system. The variable x in N of $\{x{:}\tau \mid N\}$ is bound. An interface type, which is a single function type or (possibly nested) intersection over function types, is used for the type annotation for a recursive function. Expressions are extended with ones for: strong pairs (namely, pair construction, left projection, and right projection); casts; and run-time expressions of the form $\langle\!\langle \ldots \rangle\!\rangle$ that can occur at run time for dynamic checking and not in source code. Recursion bodies are (possibly nested strong pairs) of λ-abstractions.

Run-time expressions deserve detailed explanation. A *delayed check* $\langle\!\langle V : \sigma \Rightarrow \tau_1 \to \tau_2 \rangle\!\rangle$ denotes a delayed cast into a function type, which is used in cases such as those discussed in Section 1 for instance. A *waiting check* $\langle\!\langle M \mathbin{?} \{x{:}\tau \mid N\} \rangle\!\rangle$ denotes a state waiting for the check M against N until M is evaluated into a value. An *active check* $\langle\!\langle M \Longrightarrow V : \{x{:}\tau \mid N\} \rangle\!\rangle$ is a state running test M to see if V satisfies N. The variable x in N of $\langle\!\langle M \mathbin{?} \{x{:}\tau \mid N\} \rangle\!\rangle$ and $\langle\!\langle M \Longrightarrow V : \{x{:}\tau \mid N\} \rangle\!\rangle$ is bound.

We do not include `blame` in expressions, although existing manifest contract systems usually include it among expressions. As a consequence, the evaluation relation for PCFvΔ_{H} is defined between commands. This distinction will turn out to be convenient in stating correspondence between the semantics of PCFvΔ_{H} and that of PCFv, which does not have `blame`.

Convention. We assume the index variable i ranges over $\{1, 2\}$ to save space.

Definition 4 (Terms). *We call the union of the sets of types and expressions as* terms.

Notation. $M \preceq N$ denotes that M *is a sub-expression of* N.

Convention. We define α-equivalence in a standard manner and identify α-equivalent terms.

$$
\begin{array}{rlrl}
\wr\mathtt{nat}\wr &= \mathtt{nat} & \wr\mathtt{if}\ L\ \mathtt{then}\ M\ \mathtt{else}\ N\wr &= \mathtt{if}\ \wr L\wr\ \mathtt{then}\ \wr M\wr\ \mathtt{else}\ \wr N\wr \\
\wr\mathtt{bool}\wr &= \mathtt{bool} & \wr x\wr &= x \\
\wr\sigma \to \tau\wr &= \wr\sigma\wr \to \wr\tau\wr & \wr M\ N\wr &= \wr M\wr\,\wr N\wr \\
\wr\sigma \wedge \tau\wr &= \wr\sigma\wr & \wr\lambda x{:}\tau.M\wr &= \lambda x{:}\wr\tau\wr.\wr M\wr \\
\wr\{x{:}\tau \mid M\}\wr &= \wr\tau\wr & \wr\langle M, N\rangle\wr &= \wr M\wr \\
\wr\mathtt{0}\wr &= \mathtt{0} & \wr\pi_i(M)\wr &= \wr M\wr \\
\wr\mathtt{succ}(M)\wr &= \mathtt{succ}(\wr M\wr) & \wr\mu f{:}I.B\wr &= \mu f{:}\wr I\wr.\wr B\wr \\
\wr\mathtt{pred}(M)\wr &= \mathtt{pred}(\wr M\wr) & \wr(M : \sigma \Rightarrow \tau)\wr &= \wr M\wr \\
\wr\mathtt{iszero}(M)\wr &= \mathtt{iszero}(\wr M\wr) & \wr\langle\!\langle V : \sigma \Rightarrow \tau_1 \to \tau_2\rangle\!\rangle\wr &= \wr V\wr \\
\wr\mathtt{true}\wr &= \mathtt{true} & \wr\langle\!\langle M\ ?\ \{x{:}\tau \mid N\}\rangle\!\rangle\wr &= \wr M\wr \\
\wr\mathtt{false}\wr &= \mathtt{false} & \wr\langle\!\langle M \Longrightarrow V : \{x{:}\tau \mid N\}\rangle\!\rangle\wr &= \wr V\wr
\end{array}
$$

Fig. 4. Essence of a PCFvΔ_{H} term.

Convention. We often omit the empty environment. We abuse a comma for the concatenation of environments like Γ_1, Γ_2. We denote a singleton environment, an environment that contains only one variable binding, by $x{:}\tau$.

Definition 5 (Free variables and substitution). *Free variables and substitution are defined similarly to PCFv; and we use the same notations. Note that since the types and expressions of PCFvΔ_{H} are mutually recursively defined, the metaoperations are inductively defined for terms.*

Definition 6 (Domain of typing context). *The* domain *of Γ, written* $\mathtt{dom}(\Gamma)$*, is defined by:* $\mathtt{dom}(\emptyset) = \emptyset$ *and* $\mathtt{dom}(\Gamma, x{:}\tau) = \mathtt{dom}(\Gamma) \cup \{x\}$*. We abbreviate $x \notin \mathtt{dom}(\Gamma)$ to $x \# \Gamma$.*

The essence of a PCFvΔ_{H} term is defined in Figure 4, which is mostly straightforward. The choice of which part we take as the essence of a strong pair is arbitrary because for a well-typed expression both parts have the same essence. Note that the essence of an active check $\langle\!\langle M \Longrightarrow V : \{x{:}\tau \mid N\}\rangle\!\rangle$ is V rather than M. This is because V is the subject of the expression.

3.3 Operational Semantics of PCFvΔ_{H}

The operational semantics of PCFvΔ_{H} consists of four relations $M \rightharpoonup_{\mathtt{p}} N$, $M \rightharpoonup_{\mathtt{c}} C$, $M \longrightarrow_{\mathtt{p}} N$, and $M \longrightarrow_{\mathtt{c}} C$. Bearing in mind the inclusion relation among syntactic categories, these relations can be regarded as binary relations between commands. The first two are basic reduction relations, and the other two are contextual evaluation relations (relations for whole programs). Furthermore, the relations subscripted by $\mathtt{p}$ correspond to PCFv evaluation, that is, *essential evaluation*; and ones subscripted by $\mathtt{c}$ correspond to dynamic contract checking. Dynamic checking is nondeterministic because of (RC-WedgeL/R), (EC-PairL), and (EC-PairR).

Essential Evaluation $\longrightarrow_{\mathtt{p}}$. The essential evaluation, defined in Figure 5, defines the evaluation of the essential part of a program; and thus, it is similar to

$$\texttt{pred}(\texttt{succ}(\overline{n})) \rightharpoonup_{\mathsf{p}} \overline{n} \qquad \text{(RP-Pred)}$$

$$\texttt{iszero}(\texttt{0}) \rightharpoonup_{\mathsf{p}} \texttt{true} \qquad \text{(RP-IsZero-T)}$$

$$\texttt{iszero}(\texttt{succ}(\overline{n})) \rightharpoonup_{\mathsf{p}} \texttt{false} \qquad \text{(RP-IsZero-F)}$$

$$\texttt{if}\,\texttt{true}\,\texttt{then}\,M\,\texttt{else}\,N \rightharpoonup_{\mathsf{p}} M \qquad \text{(RP-If-T)}$$

$$\texttt{if}\,\texttt{false}\,\texttt{then}\,M\,\texttt{else}\,N \rightharpoonup_{\mathsf{p}} N \qquad \text{(RP-If-F)}$$

$$(\lambda x{:}\tau.M)\,V \rightharpoonup_{\mathsf{p}} M[x \mapsto V] \qquad \text{(RP-Beta)}$$

$$\mu f{:}I.B \rightharpoonup_{\mathsf{p}} B[f \mapsto \mu f{:}I.B] \qquad \text{(RP-Fix)}$$

$$\frac{M \rightharpoonup_{\mathsf{p}} N}{M \longrightarrow_{\mathsf{p}} N}\ \text{(EP-Red)} \qquad \frac{M \longrightarrow_{\mathsf{p}} N}{\mathcal{E}[M] \longrightarrow_{\mathsf{p}} \mathcal{E}[N]}\ \text{(EP-Ctx)}$$

$$\frac{M \longrightarrow_{\mathsf{p}} M' \qquad N \longrightarrow_{\mathsf{p}} N'}{\langle M, N\rangle \longrightarrow_{\mathsf{p}} \langle M', N'\rangle}\ \text{(EP-PairS)}$$

Fig. 5. Operational semantics of PCFvΔ_{H} (1): essential evaluation.

$\longrightarrow_{\mathsf{PCF}}$. There are just three differences, that is: there are two relations; there is no reduction rule for $\texttt{pred}(\texttt{0})$; and there is a distinguished contextual evaluation rule (EP-PairS), which synchronizes essential reductions of the elements in a strong pair. The synchronization in (EP-PairS) is important since a strong pair requires the essences of both elements to be the same. The lack of predecessor evaluation for $\texttt{0}$ is intentional: Our type system and run-time checking guarantee that $\texttt{0}$ cannot occur as an argument to $\texttt{pred}$.

Dynamic Checking $\longrightarrow_{\mathsf{c}}$. Dynamic checking is more complicated. Firstly, we focus on reduction rules in Figure 6. The side-conditions on some rules are set so that an evaluation is less nondeterministic (for example, without the side conditions, both (RC-Forget) and (RC-Delay) could be applied to one expression).

The rules irrelevant to intersection types ((RC-Nat), (RC-Bool), (RC-Forget), (RC-Delay), (RC-Arrow), (RC-Waiting), (RC-Activate), (RC-Succeed), and (RC-Fail)) are adopted from Sekiyama et al. [26], but there is one difference about (RC-Delay) and (RC-Arrow). In the original definition delayed checking is done by using lambda abstractions, that is,

$$(V : \sigma_1 \to \sigma_2 \Rightarrow \tau_1 \to \tau_2) \longrightarrow \lambda x{:}\tau_1.(V\,(x : \tau_1 \Rightarrow \sigma_1) : \sigma_2 \Rightarrow \tau_2).$$

The reason we adopt a different way is just it makes technical development easier. Additionally, the way we adopt is not new—It is used in the original work [8] on higher-order contract calculi.

The other rules are new ones we propose for dynamic checking of intersection types. As we have discussed in Section 2, a cast into an intersection type is reduced into a pair of casts by (RC-WedgeI). A cast from an intersection type is done by (RC-Delay), (RC-WedgeL/R) if the target type is a function type. Otherwise, if the target type is a first order type, (RC-WedgeN) and (RC-WedgeB) are used, where we arbitrarily choose the left side of the intersection

$$\pi_i(\langle V_1, V_2\rangle) \rightharpoonup_{\mathsf{c}} V_i \qquad \text{(RC-Proj)}$$

$$(V : \mathtt{nat} \Rightarrow \mathtt{nat}) \rightharpoonup_{\mathsf{c}} V \qquad \text{(RC-Nat)}$$

$$(V : \mathtt{bool} \Rightarrow \mathtt{bool}) \rightharpoonup_{\mathsf{c}} V \qquad \text{(RC-Bool)}$$

$$(V : \{x{:}\sigma \mid M\} \Rightarrow \tau) \rightharpoonup_{\mathsf{c}} (V : \sigma \Rightarrow \tau) \qquad \text{(RC-Forget)}$$

$$\frac{(\forall x\tau M.\sigma \neq \{x{:}\tau \mid M\})}{(V : \sigma \Rightarrow \tau_1 \to \tau_2) \rightharpoonup_{\mathsf{c}} \langle\!\langle V : \sigma \Rightarrow \tau_1 \to \tau_2\rangle\!\rangle} \qquad \text{(RC-Delay)}$$

$$\langle\!\langle V_1 : \sigma_1 \to \sigma_2 \Rightarrow \tau_1 \to \tau_2\rangle\!\rangle\, V_2 \rightharpoonup_{\mathsf{c}} (V_1\,(V_2 : \tau_1 \Rightarrow \sigma_1) : \sigma_2 \Rightarrow \tau_2) \qquad \text{(RC-Arrow)}$$

$$\langle\!\langle V_1 : \sigma_1 \wedge \sigma_2 \Rightarrow \tau_1 \to \tau_2\rangle\!\rangle\, V_2 \rightharpoonup_{\mathsf{c}} (\pi_i(V_1) : \sigma_i \Rightarrow \tau_1 \to \tau_2)\, V_2 \qquad \text{(RC-WedgeL/R)}$$

$$(V : \sigma_1 \wedge \sigma_2 \Rightarrow \mathtt{nat}) \rightharpoonup_{\mathsf{c}} (\pi_1(V) : \sigma_1 \Rightarrow \mathtt{nat}) \qquad \text{(RC-WedgeN)}$$

$$(V : \sigma_1 \wedge \sigma_2 \Rightarrow \mathtt{bool}) \rightharpoonup_{\mathsf{c}} (\pi_1(V) : \sigma_1 \Rightarrow \mathtt{bool}) \qquad \text{(RC-WedgeB)}$$

$$\frac{(\forall x\tau M.\sigma \neq \{x{:}\tau \mid M\})}{(V : \sigma \Rightarrow \tau_1 \wedge \tau_2) \rightharpoonup_{\mathsf{c}} \langle (V : \sigma \Rightarrow \tau_1), (V : \sigma \Rightarrow \tau_2)\rangle} \qquad \text{(RC-WedgeI)}$$

$$\frac{(\forall x\tau M.\sigma \neq \{x{:}\tau \mid M\})}{(V : \sigma \Rightarrow \{x{:}\tau \mid M\}) \rightharpoonup_{\mathsf{c}} \langle\!\langle (V : \sigma \Rightarrow \tau)\ ?\ \{x{:}\tau \mid M\}\rangle\!\rangle} \qquad \text{(RC-Waiting)}$$

$$\langle\!\langle V\ ?\ \{x{:}\tau \mid M\}\rangle\!\rangle \rightharpoonup_{\mathsf{c}} \langle\!\langle M[x \mapsto V] \Longrightarrow V : \{x{:}\tau \mid M\}\rangle\!\rangle \qquad \text{(RC-Activate)}$$

$$\langle\!\langle \mathtt{true} \Longrightarrow V : \{x{:}\tau \mid M\}\rangle\!\rangle \rightharpoonup_{\mathsf{c}} V \qquad \text{(RC-Succeed)}$$

$$\langle\!\langle \mathtt{false} \Longrightarrow V : \{x{:}\tau \mid M\}\rangle\!\rangle \rightharpoonup_{\mathsf{c}} \mathtt{blame} \qquad \text{(RC-Fail)}$$

Fig. 6. Operational semantics of PCFvΔ_{H} (2): reduction rules for dynamic checking.

type and the corresponding part of the value since the source type is not used for dynamic checking of first-order values.

The contextual evaluation rules, defined in Figure 7, are rather straightforward. Be aware of the use of metavariables, for instance, the use of N in (EC-Ctx); it implicitly means that M has not been evaluated into `blame` (so the rule does not overlap with (EB-Ctx)). The first rule lifts the reduction relation to the evaluation relation. The next six rules express the case where a sub-expression is successfully evaluated. The rules (EC-ActiveP) and (EC-ActiveC) mean that evaluation inside an active check is always considered dynamic checking, even when it involves essential evaluation. The rules (EC-PairL) and (EC-PairR) mean that dynamic checking does not synchronize because the elements in a strong pair may have different casts. The other rules express the case where dynamic checking has failed. An expression evaluates to `blame` immediately—in one step—when a sub-expression evaluates to `blame`. Here is an example of execution of failing dynamic checking.

$$\begin{aligned}(0 : \mathtt{nat} \Rightarrow \{x{:}\mathtt{nat} \mid x > 0\}) + 1 &\longrightarrow \langle\!\langle 0\ ?\ \{x{:}\mathtt{nat} \mid x > 0\}\rangle\!\rangle + 1\\ &\longrightarrow \langle\!\langle 0 > 0 \Longrightarrow 0 : \{x{:}\mathtt{nat} \mid x > 0\}\rangle\!\rangle + 1\\ &\longrightarrow \langle\!\langle \mathtt{false} \Longrightarrow 0 : \{x{:}\mathtt{nat} \mid x > 0\}\rangle\!\rangle + 1\\ &\longrightarrow \mathtt{blame}\end{aligned}$$

$$\frac{M \rightharpoonup_{\mathsf{c}} C}{M \longrightarrow_{\mathsf{c}} C}\ (\text{EC-Red}) \qquad \frac{M \longrightarrow_{\mathsf{c}} N}{\mathcal{E}[M] \longrightarrow_{\mathsf{c}} \mathcal{E}[N]}\ (\text{EC-Ctx})$$

$$\frac{M \longrightarrow_{\mathsf{p}} M'}{\langle\!\langle M \Longrightarrow V : \{x{:}\tau \mid N\}\rangle\!\rangle \longrightarrow_{\mathsf{c}} \langle\!\langle M' \Longrightarrow V : \{x{:}\tau \mid N\}\rangle\!\rangle}\ (\text{EC-ActiveP})$$

$$\frac{M \longrightarrow_{\mathsf{c}} M'}{\langle\!\langle M \Longrightarrow V : \{x{:}\tau \mid N\}\rangle\!\rangle \longrightarrow_{\mathsf{c}} \langle\!\langle M' \Longrightarrow V : \{x{:}\tau \mid N\}\rangle\!\rangle}\ (\text{EC-ActiveC})$$

$$\frac{M \longrightarrow_{\mathsf{c}} M'}{\langle M, N\rangle \longrightarrow_{\mathsf{c}} \langle M', N\rangle}\ (\text{EC-PairL}) \qquad \frac{N \longrightarrow_{\mathsf{c}} N'}{\langle M, N\rangle \longrightarrow_{\mathsf{c}} \langle M, N'\rangle}\ (\text{EC-PairR})$$

$$\frac{M \longrightarrow_{\mathsf{c}} \mathtt{blame}}{\mathcal{E}[M] \longrightarrow_{\mathsf{c}} \mathtt{blame}}\ (\text{EB-Ctx})$$

$$\frac{M \longrightarrow_{\mathsf{c}} \mathtt{blame}}{\langle\!\langle M \Longrightarrow V : \{x{:}\tau \mid N\}\rangle\!\rangle \longrightarrow_{\mathsf{c}} \mathtt{blame}}\ (\text{EB-Active})$$

$$\frac{M_i \longrightarrow_{\mathsf{c}} \mathtt{blame}}{\langle M_1, M_2\rangle \longrightarrow_{\mathsf{c}} \mathtt{blame}}\ (\text{EB-PairL/R})$$

Fig. 7. Operational semantics of PCFvΔ_{H} (3): contextual rules for dynamic checking.

$$\emptyset\ \text{ok}\ (\text{V-Empty}) \qquad \frac{\Gamma\ \text{ok} \qquad \Vdash \tau \qquad (x \# \Gamma)}{\Gamma, x{:}\tau\ \text{ok}}\ (\text{V-Push})$$

$$\Vdash \mathtt{nat}\ (\text{W-Nat}) \qquad \Vdash \mathtt{bool}\ (\text{W-Bool}) \qquad \frac{\Vdash \sigma \qquad \Vdash \tau}{\Vdash \sigma \to \tau}\ (\text{W-Arrow})$$

$$\frac{\Vdash \sigma \qquad \Vdash \tau \qquad (\wr\sigma\wr = \wr\tau\wr)}{\Vdash \sigma \wedge \tau}\ (\text{W-Wedge}) \qquad \frac{x{:}\tau \vdash M : \mathtt{bool}}{\Vdash \{x{:}\tau \mid M\}}\ (\text{W-Refine})$$

Fig. 8. Type system of PCFvΔ_{H} (1): well-formedness rules.

Definition 7 (Evaluation). *The one-step evaluation relation of PCFvΔ_{H}, denoted by $\longrightarrow$, is defined as $\longrightarrow_{\mathsf{p}} \cup \longrightarrow_{\mathsf{c}}$. The multi-step evaluation relation of PCFvΔ_{H}, denoted by $\longrightarrow^*$, is the reflexive and transitive closure of $\longrightarrow$.*

3.4 Type System of PCFvΔ_{H}

The type system consists of three judgments: Γ ok, $\Vdash \tau$, and $\Gamma \vdash M : \tau$, read "Γ is well-formed", "τ is well-formed", and "M has τ under Γ," respectively. They are defined inductively by the rules in Figures 8, 9 and 10.

The rules for well-formed types check that an intersection type is restricted to a refinement intersection type by the side condition $\wr\sigma\wr = \wr\tau\wr$ in (W-Wedge) and that the predicate in a refinement type is a Boolean expression by (W-Refine). Note that, since PCFvΔ_{H} has no dependent function type, all types are closed and the predicate of a refinement type only depends on the parameter itself.

The typing rules, the rules for the third judgment, consist of two more sub-categories: compile-time rules and run-time rules. Compile-time rules are for checking a program a programmer writes. Run-time rules are for run-time expressions and used to prove type soundness. This distinction, which follows, Belo et al. [1], is to make compile-time type checking decidable.

$$\frac{\Gamma \text{ ok}}{\Gamma \vdash \mathtt{0} : \mathtt{nat}}\ (\text{T-Zero}) \qquad \frac{\Gamma \vdash M : \mathtt{nat}}{\Gamma \vdash \mathtt{succ}(M) : \mathtt{nat}}\ (\text{T-Succ})$$

$$\frac{\Gamma \vdash M : \{x{:}\mathtt{nat} \mid \mathtt{if}\ \mathtt{iszero}(x)\ \mathtt{then}\ \mathtt{false}\ \mathtt{else}\ \mathtt{true}\}}{\Gamma \vdash \mathtt{pred}(M) : \mathtt{nat}}\ (\text{T-Pred})$$

$$\frac{\Gamma \vdash M : \mathtt{nat}}{\Gamma \vdash \mathtt{iszero}(M) : \mathtt{bool}}\ (\text{T-IsZero}) \qquad \frac{\Gamma \text{ ok}}{\Gamma \vdash \mathtt{true} : \mathtt{bool}}\ (\text{T-True})$$

$$\frac{\Gamma \text{ ok}}{\Gamma \vdash \mathtt{false} : \mathtt{bool}}\ (\text{T-False})$$

$$\frac{\Gamma \vdash L : \mathtt{bool} \qquad \Gamma \vdash M : \tau \qquad \Gamma \vdash N : \tau}{\Gamma \vdash \mathtt{if}\ L\ \mathtt{then}\ M\ \mathtt{else}\ N : \tau}\ (\text{T-If})$$

$$\frac{\Gamma \text{ ok} \qquad (x{:}\tau \in \Gamma)}{\Gamma \vdash x : \tau}\ (\text{T-Var}) \qquad \frac{\Gamma, x{:}\sigma \vdash M : \tau}{\Gamma \vdash \lambda x{:}\sigma.M : \sigma \to \tau}\ (\text{T-Abs})$$

$$\frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash M\ N : \tau}\ (\text{T-App})$$

$$\frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash N : \tau \qquad (\wr M \wr = \wr N \wr) \qquad (\wr \sigma \wr = \wr \tau \wr)}{\Gamma \vdash \langle M, N \rangle : \sigma \wedge \tau}\ (\text{T-Pair})$$

$$\frac{\Gamma \vdash M : \sigma \wedge \tau}{\Gamma \vdash \pi_1(M) : \sigma}\ (\text{T-Fst}) \qquad \frac{\Gamma \vdash M : \sigma \wedge \tau}{\Gamma \vdash \pi_2(M) : \tau}\ (\text{T-Snd}) \qquad \frac{\Gamma, f{:}I \vdash B : I}{\Gamma \vdash \mu f{:}I.B : I}\ (\text{T-Fix})$$

$$\frac{\Gamma \vdash M : \sigma \qquad \Vdash \tau \qquad (\wr \sigma \wr = \wr \tau \wr)}{\Gamma \vdash (M : \sigma \Rightarrow \tau) : \tau}\ (\text{T-Cast})$$

Fig. 9. Type system of PCFvΔ_{H} (2): compile-time typing rules.

A large part of the compile-time rules are adapted from PCF, Sekiyama et al. [26], and Liquori and Stolze [17]. Here we explain some notable rules. As an intersection type system, (T-Pair), (T-Fst), and (T-Snd) stands for introduction and elimination rules of intersection types (or we can explicitly introduce and/or eliminate an intersection type by a cast). The rule (T-Pair) checks a strong pair is composed by essentially the same expressions by $\wr M \wr = \wr N \wr$. The rule (T-Pred) demands that the argument of predecessor shall not be zero. The premise $\wr \sigma \wr = \wr \tau \wr$ of the rule (T-Cast) for casts requires the essences of the source and target types to agree. It amounts to checking the two types σ and τ are compatible [26].

The run-time rules are from Sekiyama et al. [26] with one extra rule (T-Delayed). The rule (T-Delayed) is for a delayed checking for function types, which restrict the source type so that it respects the evaluation relation (there is no evaluation rule for a delayed checking in which source type is a refinement type), and inherits the condition on the source and target types from (T-Cast). The side condition $N[x \mapsto V] \longrightarrow^* M$ on (T-Active) is an invariant during evaluation, that is, M is an intermediate state of the predicate checking. This invariant lasts until the final (successful) run-time checking state $\langle\!\langle \mathtt{true} \Longrightarrow V : \{x{:}\tau \mid N\} \rangle\!\rangle$ and guarantees the checking result V (obtained by (RC-Succeed)) satisfies the predicate N by (T-Exact).

$$\frac{\Gamma \text{ ok} \qquad \vdash V : \sigma \qquad \Vdash \tau_1 \to \tau_2 \qquad (\forall x\tau M.\sigma \neq \{x{:}\tau \mid M\}) \qquad (\wr\sigma\wr = \wr\tau_1 \to \tau_2\wr)}{\Gamma \vdash \langle\!\langle V : \sigma \Rightarrow \tau_1 \to \tau_2 \rangle\!\rangle : \tau_1 \to \tau_2} \text{(T-Delayed)}$$

$$\frac{\Gamma \text{ ok} \qquad \vdash M : \tau \qquad \Vdash \{x{:}\tau \mid N\}}{\Gamma \vdash \langle\!\langle M \mathbin{?} \{x{:}\tau \mid N\} \rangle\!\rangle : \{x{:}\tau \mid N\}} \text{(T-Waiting)}$$

$$\frac{\Gamma \text{ ok} \qquad \vdash M : \mathtt{bool} \qquad \vdash V : \tau \qquad \Vdash \{x{:}\tau \mid N\} \qquad N[x \mapsto V] \longrightarrow^* M}{\Gamma \vdash \langle\!\langle M \Longrightarrow V : \{x{:}\tau \mid N\} \rangle\!\rangle : \{x{:}\tau \mid N\}} \text{(T-Active)}$$

$$\frac{\Gamma \text{ ok} \qquad \vdash V : \{x{:}\tau \mid N\}}{\Gamma \vdash V : \tau} \text{(T-Forget)}$$

$$\frac{\Gamma \text{ ok} \qquad \vdash V : \tau \qquad \Vdash \{x{:}\tau \mid N\} \qquad N[x \mapsto V] \longrightarrow^* \mathtt{true}}{\Gamma \vdash V : \{x{:}\tau \mid N\}} \text{(T-Exact)}$$

Fig. 10. Type system of PCFvΔ_{H} (3): run-time typing rules.

4 Properties

We start from properties of evaluation relations. As we have mentioned, $\longrightarrow_{\mathsf{p}}$ is essential evaluation, and thus, it should simulate $\longrightarrow_{\mathsf{PCF}}$; and $\longrightarrow_{\mathsf{c}}$ is dynamic checking, and therefore, it should not change the essence of the expression. We formally state and show these properties here. Note that most properties require that the expression before evaluation is well typed. This is because the condition of strong pairs is imposed by the type system.

Lemma 1. *If $M \longrightarrow_{\mathsf{PCF}} N$ and $M \longrightarrow_{\mathsf{PCF}} L$, then $N = L$.*

Proof. The proof is routine by induction on one of the given derivations. □

Lemma 2. *If $\vdash M : \tau$ and $M \longrightarrow_{\mathsf{p}} N$, then $\wr M\wr \longrightarrow_{\mathsf{PCF}} \wr N\wr$.*

Proof. The proof is by induction on the given evaluation derivation. □

The following corollary is required to prove the preservation property.

Corollary 1. *If $\vdash M : \sigma$, $\vdash N : \tau$, $M \longrightarrow_{\mathsf{p}} M'$, $N \longrightarrow_{\mathsf{p}} N'$, and $\wr M\wr = \wr N\wr$; then $\wr M'\wr = \wr N'\wr$.*

Lemma 3. *If $\vdash M : \tau$ and $M \longrightarrow_{\mathsf{c}} N$, then $\wr M\wr = \wr N\wr$.*

Proof. The proof is by induction on the given evaluation derivation. □

Now we can have the following theorem as a corollary of Lemma 2 and Lemma 3. It guarantees the essential computation in PCFvΔ_{H} is the same as the PCFv computation as far as the computation does not fail. In other words, run-time checking may introduce blame but otherwise does not affect the essential computation.

Theorem 1. *If $\vdash M : \tau$ and $M \longrightarrow N$, then $\wr M\wr \longrightarrow^*_{\mathsf{PCF}} \wr N\wr$.*

4.1 Type Soundness

We conclude this section with type soundness. Firstly, we show a substitution property; and using it, we show the preservation property.

Lemma 4. *If* $\Gamma_1, x{:}\sigma, \Gamma_2 \vdash M : \tau$ *and* $\Gamma_1 \vdash N : \sigma$*, then* $\Gamma_1, \Gamma_2 \vdash M[x \mapsto N] : \tau$*.*

Proof. The proof is by induction on the derivation for M. □

Theorem 2 (Preservation). *If* $\vdash M : \tau$ *and* $M \longrightarrow N$*, then* $\vdash N : \tau$*.*

Proof. We prove preservation properties for each $\longrightarrow_{\mathsf{p}}$ and $\longrightarrow_{\mathsf{c}}$ and combine them. Both proofs are done by induction on the given typing derivation. For the case in which substitution happens, we use Lemma 4 as usual. For the context evaluation for strong pairs, we use Corollary 1 and Lemma 3 to guarantee the side-condition of strong pairs. □

Next we show the value inversion property, which guarantees a value of a refinement type satisfies its predicate. For PCFvΔ_{H}, this property can be quite easily shown since PCFvΔ_{H} does not have dependent function types, while previous manifest contract systems need quite complicated reasoning [19, 24, 26]. The property itself is proven by using the following two, which are for strengthening an induction hypothesis.

Definition 8. *We define a relation between values and types, written* $V \models \tau$*, by the following rules.*

$$\frac{V \models \tau \qquad M[x \mapsto V] \longrightarrow^* \texttt{true}}{V \models \{x{:}\tau \mid M\}} \qquad \frac{(\tau \neq \{x{:}\sigma \mid M\})}{V \models \tau}$$

Lemma 5. *If* $\vdash V : \tau$*, then* $V \models \tau$*.*

Proof. The proof is by induction on the given derivation. □

Theorem 3 (Value inversion). *If* $\vdash V : \{x{:}\tau \mid M\}$*, then* $M[x \mapsto V] \longrightarrow^* \texttt{true}$*.*

Proof. Immediate from Lemma 5. □

Remark 1. As a corollary of value inversion, it follows that a value of an intersection type must be a strong pair and its elements satisfy the corresponding predicate in the intersection type: For example, if $\vdash \langle V_1, V_2 \rangle : \{x{:}\sigma \mid M\} \wedge \{x{:}\tau \mid N\}$, then $M[x \mapsto V_1] \longrightarrow^* \texttt{true}$ and $N[x \mapsto V_2] \longrightarrow^* \texttt{true}$. In particular, for first-order values, every element of the pair is same. That means the value satisfies all contracts concatenated by $\wedge$. For example, $\vdash V : \{x{:}\texttt{nat} \mid M_1\} \wedge \cdots \wedge \{x{:}\texttt{nat} \mid M_n\}$, then $M_k[x \mapsto \wr V \wr] \longrightarrow^* \texttt{true}$ for any $k = 1..n$. This is what we have desired for a contract written by using intersection types.

Lastly, the progress property also holds. In our setting, where $\texttt{pred}(M)$ is partial, this theorem can be proved only after Theorem 3.

Theorem 4 (Progress). *If* $\vdash M : \tau$*, then* M *is a value or* $M \longrightarrow C$ *for some* C*.*

Proof. The proof is by induction on the given derivation. Since the evaluation relation is defined as combination of $\longrightarrow_{\mathsf{p}}$ and $\longrightarrow_{\mathsf{c}}$, the proof is a bit tricky, but most cases can be proven as usual. An interesting case is (T-Pair). We need to guarantee that if one side of a strong pair is a value, another side must not be evaluated by $\longrightarrow_{\mathsf{p}}$ since a value is in normal form. This follows from Lemma 2 and proof by contradiction because the essence of a PCFvΔ_{H} value is a PCFv value and it is normal form. □

5 Related Work

Intersection types were introduced in Curry-style type assignment systems by Coppo et al. [6] and Pottinger [21] independently. In the early days, intersection types are motivated by improving a type system to make more lambda terms typeable; one important result towards this direction is that: *a lambda term has a type iff it can be strongly normalized* [21, 29]. Then, intersection types are introduced to programming languages to enrich the descriptive power of types [2, 7, 22].

Intersection Contracts for Untyped Languages. One of the first attempts at implementing intersection-like contracts is found in DrRacket [9]. It is, however, a naive implementation, which just enforces all contracts even for functional values, and thus the semantics of higher-order intersection contracts is rather different from ours.

Keil and Thiemann [14] have proposed an untyped calculus of blame assignment for a higher-order contract system with intersection and union. As we have mentioned, our run-time checking semantics is strongly influenced by their work, but there are two essential differences. On the one hand, they do not have the problem of varying run-time checking according to a typing context; they can freely put contract monitors[4] where they want since it is an untyped language. On the other hand, their operational semantics is made rather complicated due to blame assignment.

More recently, Williams et al. [32] have proposed more sorted out semantics for a higher-order contract system with intersection and union. They have mainly reformed contract checking for intersection and union "in a uniform way"; that is, each is handled by only one similar and simpler rule. As a result, their presentation becomes closer to our semantics, though complication due to blame assignment still remains. A similar level of complication will be expected if we extend our calculus with blame assignment.

It would be interesting to investigate the relationship between their calculi and PCFvΔ_{H} extended with blame labels, following Greenberg et al. [12].

[4] A kind of casts in their language.

Gradual Typing with Intersection Types. Castagna and Lanvin [3] have proposed gradual typing for set-theoretic types, which contain intersection types, as well as union and negation. A framework of gradual typing is so close to manifest contract systems that there is even a study unifying them [31]. A gradual typing system translates a program into an intermediate language that is statically typed and uses casts. Hence, they have the same problem—how casts should be inserted when intersection types are used. They solve the problem by *type-case* expressions, which dynamically dispatch behavior according to the type of a value. However, it is not clear how type-case expressions scale to a larger language. In fact, the following work [4], an extension to parametric polymorphism and type inference, removes (necessity of) type-case expressions but imposes instead a restriction on functions not to have an intersection type. Furthermore, the solution using type-case expressions relies on strong properties of set-theoretic types. So, it is an open problem if their solution can be adopted to manifest contract systems because there is not set-theoretic type theory for refinement types and, even worse, dependent function types.

Nondeterminism for Dependently Typed Languages. As we have noted in Section 1, PCFvΔ_{H} has no dependent function types. In fact, no other work discussed in this section supports both dependent function contracts and intersection contracts. To extend PCFvΔ_{H} to dependent function types, we have to take care of their interaction with nondeterminism, which we studied elsewhere [19] for a manifest calculus $\lambda^{H\|\Phi}$ with a general nondeterministic choice operator.

A technical challenge in combining dependent function types and nondeterminstic choice comes from the following standard typing rule for (dependent) function applications:

$$\frac{\Gamma \vdash M : (x{:}\sigma) \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash M\, N : \tau[x \mapsto N]}$$

The problem is that the argument N, which may contain nondeterministic choice, may be duplicated in $\tau[x \mapsto N]$ and, to keep consistency of type equivalence, choices made in each occurrence of N have to be "synchronized." To control synchronization, $\lambda^{H\|\Phi}$ introduces a named choice operator so that choice operators with the same name make synchronized choice. However, $\lambda^{H\|\Phi}$ puts burden on programmers to avoid unintended synchronization caused by accidentally shared names.

If we incoporate the idea above to PCFvΔ_{H}, it will be natural to put names on casts so that necessary synchronization takes place for choices made by (RC-WedgeL) and (RC-WedgeR). It is not clear, however, how unintended synchronization can be avoided systematically, without programmers' ingenuity.

6 Conclusion

We have designed and formalized a manifest contract system PCFvΔ_{H} with refinement intersection types. As a result of our formal development, PCFvΔ_{H}

guarantees not only ordinary preservation and progress but also the property that a value of an intersection type, which can be seen as an enumeration of small contracts, satisfies all the contracts.

The characteristic point of our formalization is that we regard a manifest contract system as an extension of a more basic calculus, which has no software contract system, and investigate the relationship between the basic calculus and the manifest contract system. More specifically, essential computation and dynamic checking are separated. We believe this investigation is important for modern manifest contract systems because those become more and more complicated and the separation is no longer admissible at a glance.

Future Work. Obvious future work is to lift the restriction we have mentioned in Section 1. That aside, the subsumption-free approach is very naive and has an obvious disadvantage, that is, it requires run-time checking even for a cast like $(M : \sigma \wedge \tau \Rightarrow \sigma)$, which should be able to checked and removed at compile time. To address the disadvantage, some manifest contract systems provide the property known as *up-cast elimination* [1]—*a cast from subtype into supertype can be safely removed at compile-time*. An interesting fact is that a well-known up-cast (subtyping) relation for a traditional intersection type system is defined syntactically; while a usual up-cast relation for a manifest contract system depends on semantics. So, focusing on only the traditional subtyping relation, the property might be proven more easily.

Towards a practice language, our cast semantics using strong pairs and nondeterminism needs more investigation. For the strong pairs, it will be quite inefficient to evaluate both sides of a strong pair independently since its essence part just computes the same thing. The inefficiency might be reduced by a kind of sharing structures. For the nondeterminism, our theoretical result gives us useful information only for successful evaluation paths; but we have not given a way to pick up a successful one. One obvious way is computing every evaluation path, but of course, it is quite inefficient.

Acknowledgments

We would like to thank Peter Thiemann, John Toman, Yuya Tsuda, and anonymous reviewers for useful comments. This work was supported in part by the JSPS KAKENHI Grant Number JP17H01723.

References

1. Belo, J.F., Greenberg, M., Igarashi, A., Pierce, B.C.: Polymorphic contracts. In: Proc. of ESOP. pp. 18–37 (2011)
2. Benzaken, V., Castagna, G., Frisch, A.: CDuce: an XML-centric general-purpose language. In: Proc. of ICFP. pp. 51–63 (2003)
3. Castagna, G., Lanvin, V.: Gradual typing with union and intersection types. PACMPL **1**(ICFP), 41:1–41:28 (2017)

4. Castagna, G., Lanvin, V., Petrucciani, T., Siek, J.G.: Gradual typing: A new perspective. Proc. ACM Program. Lang. **3**(POPL), 16:1–16:32 (Jan 2019)
5. Charguéraud, A.: The locally nameless representation. J. Autom. Reasoning **49**(3), 363–408 (2012)
6. Coppo, M., Dezani-Ciancaglini, M., Venneri, B.: Functional characters of solvable terms. Math. Log. Q. **27**(2-6), 45–58 (1981)
7. Dunfield, J.: Refined typechecking with Stardust. In: Proc. of PLPV. pp. 21–32 (2007)
8. Findler, R.B., Felleisen, M.: Contracts for higher-order functions. In: Proc. of ICFP. pp. 48–59 (2002)
9. Findler, R.B., PLT: DrRacket: Programming environment. Tech. Rep. PLT-TR-2010-2, PLT Design Inc. (2010), `https://racket-lang.org/tr2/`
10. Flanagan, C.: Hybrid type checking. In: Proc. of POPL. pp. 245–256 (2006)
11. Greenberg, M.: Space-efficient manifest contracts. In: Proc. of POPL. pp. 181–194 (2015)
12. Greenberg, M., Pierce, B.C., Weirich, S.: Contracts made manifest. In: Proc. of POPL. pp. 353–364 (2010)
13. Gronski, J., Knowles, K., Tomb, A., Freund, S.N., Flanagan, C.: Sage: Hybrid checking for flexible specifications. In: Scheme and Functional Programming Workshop. pp. 93–104 (2006)
14. Keil, M., Thiemann, P.: Blame assignment for higher-order contracts with intersection and union. In: Proc. of ICFP. pp. 375–386 (2015)
15. Knowles, K., Flanagan, C.: Hybrid type checking. ACM Trans. Program. Lang. Syst. **32**(2), 6:1–6:34 (2010)
16. Kobayashi, N., Sato, R., Unno, H.: Predicate abstraction and CEGAR for higher-order model checking. In: Proc. of PLDI. pp. 222–233 (2011)
17. Liquori, L., Stolze, C.: The Δ-calculus: Syntax and types. In: Proc. of FSCD. pp. 28:1–28:20 (2018)
18. Meyer, B.: Object-Oriented Software Construction, 2nd Edition. Prentice-Hall (1997)
19. Nishida, Y., Igarashi, A.: Nondeterministic manifest contracts. In: Proc. of PPDP. pp. 16:1–16:13 (2018)
20. Plotkin, G.D.: LCF considered as a programming language. Theor. Comput. Sci. **5**(3), 223–255 (1977)
21. Pottinger, G.: A type assignment for the strongly normalizabile λ-terms. To H. B. Curry, Essays in Combinatory Logic, Lambda-Calculus and Formalism pp. 561–577 (1980)
22. Reynolds, J.C.: Preliminary design of the programming language Forsythe. Tech. Rep. CMU-CS-88-159, Carnegie Mellon University (1988)
23. Rondon, P.M., Kawaguchi, M., Jhala, R.: Liquid types. In: Proc. of PLDI. pp. 159–169 (2008)
24. Sekiyama, T., Igarashi, A.: Stateful manifest contracts. In: Proc. of POPL. pp. 530–544 (2017)
25. Sekiyama, T., Igarashi, A., Greenberg, M.: Polymorphic manifest contracts, revised and resolved. ACM Trans. Program. Lang. Syst. **39**(1), 3:1–3:36 (2017)
26. Sekiyama, T., Nishida, Y., Igarashi, A.: Manifest contracts for datatypes. In: Proc. of POPL. pp. 195–207 (2015)
27. Terauchi, T.: Dependent types from counterexamples. In: Proc. of POPL. pp. 119–130 (2010)
28. Unno, H., Kobayashi, N.: Dependent type inference with interpolants. In: Proc. of PPDP. pp. 277–288 (2009)

29. Valentini, S.: An elementary proof of strong normalization for intersection types. Arch. Math. Log. **40**(7), 475–488 (2001)
30. Vazou, N., Seidel, E.L., Jhala, R., Vytiniotis, D., Peyton-Jones, S.: Refinement types for Haskell. In: Proc. of ICFP. pp. 269–282 (2014)
31. Wadler, P., Findler, R.B.: Well-typed programs can't be blamed. In: Proc. of ESOP. pp. 1–16 (2009)
32. Williams, J., Morris, J.G., Wadler, P.: The root cause of blame: Contracts for intersection and union types. Proc. ACM Program. Lang. **2**(OOPSLA), 134:1–134:29 (Oct 2018)
33. Zhu, H., Jagannathan, S.: Compositional and lightweight dependent type inference for ML. In: Proc. of VMCAI. pp. 295–314 (2013)