Kyoto University
Department of Communications and Computer Engineering
Graduate School of Informatics

Doctoral Thesis

Typed Software Contracts with Intersection and Nondeterminism

Yuki Nishida

April 30, 2020

Supervisor: Professor Atsushi Igarashi

Abstract

Program verification is one of the important subjects of software science, and there is a vast amount of work dealing with the subject. Static type systems are one of the most successful static verification methods, but they can deal with rather simple specifications in practice. Meanwhile, some languages which do not have a static type system have a *software contract* system, which monitors the behavior of a program at run-time and checks if an embedded specification, namely contract, is not violated. Contracts are usually described as Boolean expressions, and so, more expressive than types.

Software contracts have been extended for several directions. One important extension is *functional contracts* that can describe a relation between input and output of a function. Another is a smooth integration with a static type system, as known as *hybrid type checking*, which combines compile- and run-time checking.

Our goal is to develop a theory for software contracts on typed functional programming languages—formalizing a mathematical model of a typed functional programming language equipped with software contract system; stating desirable properties that the language should have; and proving the properties. In this thesis, we try to combine two well-known programming features with software contract systems.

The first one is a contract in conjunctive form. A conjunctive contract is a contract consisting of smaller contracts by connecting with "and" conjunction. Such a contract naturally arises when we naively think program specifications. Furthermore, it is quite useful when we add more contracts to an existing contract. A challenge comes from functional contracts. As for contracts about first-order values—integers, booleans, etc.; it is obvious how we handle conjunctive contracts—we can just see the contracts as Boolean conjunction. However, it is not trivial for functional contracts, which are not simple Boolean predicates anymore.

The other is nondeterminism. Nondeterminism is widely used in logic programming languages, like Prolog; where evaluation order of code has freedom, and even more, a result of evaluation could vary. An interesting point is that contracts could involve nondeterminism if we allow arbitrary code to describe a contract. This design choice blurs the meaning of a contract because an evaluation of a predicate could succeed or fail randomly.

To archive the goal we develop two *manifest contract systems* and supplemental discussion. Manifest contract systems are one group of formal calculi for typed software contract systems. In a manifest contract system, contracts are integrated into types by using *refinement types* of the form $\{x{:}\tau \mid M\}$, where the predicate M restricts values belonging to the refinement types; and so for instance, $\{x{:}\texttt{int} \mid x > 0\}$ denotes positive integers. A manifest contract system still checks contracts at run-time by means of *casts* of the form $(M : \sigma \Rightarrow \tau)$, which checks if the value obtained by M of type σ can have type τ at run-time. One advantage of a manifest contract system to traditional software contracts system is whether a value has been checked against some contracts is guaranteed as a formal property of the system, that is, if a value has the refinement type $\{x{:}\tau \mid M\}$, the value satisfies the predicate M.

The first manifest contract system that we have proposed is one equipped with *intersection*

types. An intersection type $\sigma \wedge \tau$ gives both types σ and τ to code. So, a conjunctive contract can be expressed by just connecting types that express each contract constituting the conjunctive contract by $\wedge$. We have formalized the system as PCFvΔ_{H}. Concretely we have shown if a value has an intersection type of refinement types, namely $\{x{:}\tau_1 \mid M_1\} \wedge \{x{:}\tau_2 \mid M_2\}$, the value satisfies both predicates M_1 and M_2.

The second manifest contract system that we have proposed is one equipped with nondeterministic choices. A nondeterministic choice is a pair of code, written $(M \parallel N)$ in this thesis, which is randomly evaluated into M or N in every execution. We formalize the system as $\lambda^{H\parallel^{\Phi}}$. As we have mentioned, it is non-trivial what the fact that a value has a refinement type means. To give a strict meaning, we have proposed *coordinated choices*, which are equipped with names, in $\lambda^{H\parallel^{\Phi}}$ instead of usual nondeterministic choices, which we have just seen. Using coordinated choices, we have shown if a value has a refinement type, the value *deterministically* satisfies the predicate of the refinement type even if the predicate involves nondeterministic code.

After developing the two systems, we give an additional discussion to give a proper name for each coordinated choice automatically. We are motivated by two reasons. One is a naive motivation, that is, giving a proper name is a bit complicated and bother work for a programmer. Another is related to PCFvΔ_{H}. In fact, PCFvΔ_{H} uses nondeterminism to check some contracts, but because of the difficulty solved by $\lambda^{H\parallel^{\Phi}}$, PCFv$\Delta_{\mathrm{H}}$ has no *dependent function types*—more expressive types equipped in most manifest contract systems. The discussion could help to recover dependent function types for PCFvΔ_{H}. Formally, we have given a compilation algorithm from a simply typed lambda calculus equipped with usual nondeterministic choices into one equipped with coordinated choices; and shown source code and its compiled code behave in the same way.

As a result of our development, we have had a theoretical base for a typed software contract system with conjunctive contracts, at least under the absence of dependent function types. As for dependent function types, we have obtained interesting results and insight. We hope that our result leads us to a fully-integrated system.

Acknowledgment

First of all, I would like to thank Prof. Atsushi Igarashi for supervising me. He suggested software contracts as my research topic. It finally constitutes my Ph.D. thesis. He gave me a lot of advice and help but not meddling. I recognize that his education raises my research skills and leads me to the completion of this thesis.

I also thank Prof. Akihiro Yamamoto and Prof. Shinichi Minato for accepting my Ph.D. examiner role. They give me useful comments from the perspective of a third party.

Active and past staffs of Computer Software Group, Dr. Kohei Suenaga, Dr. Seiji Umatani, Dr. Koji Nakazawa, Dr. Kensuke Kojima, Dr. Kanae Tsushima, and Ms. Kaori Takei, also help my study and assist my laboratory life. Dr. Kohei Suenaga, especially, give me a chance and advise for another research work that is not included in this thesis, though.

I am happy to thank laboratory students who study and work together. It is a very pleasant time during my school days.

Lastly, I am very grateful to my parents, who support me for a long time.

List of Publications

Chapter 3 consists of

> Yuki Nishida and Atsushi Igarashi. "Manifest Contracts with Intersection Types". In: *Programming Languages and Systems - 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1-4, 2019, Proceedings*. Ed. by Anthony Widjaja Lin. Vol. 11893. Lecture Notes in Computer Science. Springer, 2019, pp. 33–52. ISBN: 978-3-030-34174-9. DOI: `10.1007/978-3-030-34175-6_3`. URL: `https://doi.org/10.1007/978-3-030-34175-6_3`

Chapter 4 consists of

> Yuki Nishida and Atsushi Igarashi. "Nondeterministic Manifest Contracts". In: *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming, PPDP 2018, Frankfurt am Main, Germany, September 03-05, 2018*. Ed. by David Sabel and Peter Thiemann. ACM, 2018, 16:1–16:13. DOI: `10.1145/3236950.3236964`. URL: `https://doi.org/10.1145/3236950.3236964`

Chapter 5 consists of

> Yuki Nishida and Atsushi Igarashi. "Compilation of Coordinated Choice". In: *CoRR* abs/2004.14084 (2020). arXiv: `2004.14084 [cs.PL]`. URL: `https://arxiv.org/abs/2004.14084`

Contents

1 Introduction **1**
- 1.1 Background . . . 2
  - 1.1.1 Software Contracts . . . 2
  - 1.1.2 Refinement Type Systems . . . 4
  - 1.1.3 Hybrid Type Checking . . . 5
- 1.2 This Thesis . . . 6
  - 1.2.1 Contributions . . . 8
- 1.3 Organization . . . 9

2 Manifest Contract System **11**
- 2.1 Language . . . 12
  - 2.1.1 Syntax . . . 12
  - 2.1.2 Semantics . . . 17
  - 2.1.3 Type System . . . 18
- 2.2 Properties . . . 23
- 2.3 Summary . . . 23

3 Manifest Contracts with Intersection Types **25**
- 3.1 Overview of Language: PCFvΔ_{H} . . . 26
  - 3.1.1 The Δ-calculus . . . 26
  - 3.1.2 Cast Semantics for Intersection Types . . . 27
- 3.2 Language: PCFvΔ_{H} . . . 28
  - 3.2.1 PCFv . . . 28
  - 3.2.2 PCFvΔ_{H} . . . 28
  - 3.2.3 Operational Semantics of PCFvΔ_{H} . . . 30
  - 3.2.4 Type System of PCFvΔ_{H} . . . 35
- 3.3 Properties . . . 35
  - 3.3.1 Type Soundness . . . 37
- 3.4 Summary . . . 39

4 Nondeterministic Manifest Contracts **41**
- 4.1 Coordinated Choice . . . 43
  - 4.1.1 Orthant . . . 43
  - 4.1.2 Orthant-local Reduction . . . 44
  - 4.1.3 Programming with Names . . . 45
- 4.2 Language . . . 46
  - 4.2.1 Syntax . . . 46
  - 4.2.2 Semantics . . . 48
  - 4.2.3 Type System . . . 50

4.3 Properties 54
4.3.1 Co-termination 56
4.3.2 Type Soundness 61
4.4 Summary 63

5 Compilation of Coordinated Choice **65**
5.1 Compilation 65
5.1.1 Correctness of compilation 66
5.2 Formal System 67
5.2.1 Source Language: $\lambda^{\|}$ 67
5.2.2 Target Language: $\lambda^{\|\omega}$ 69
5.2.3 Effect System 69
5.3 Property 73
5.3.1 Type Soundness of $\lambda^{\|\omega}$ 74
5.3.2 Soundness of Compilation 74
5.3.3 Bisimulation 75
5.4 Summary 77

6 Related Work **79**

7 Conclusion **83**

List of Figures

2.1 Syntax of $\mathrm{PCF_H}$ . . . 12
2.2 Free variables of $\mathrm{PCF_H}$ terms . . . 14
2.3 Capture-avoiding substitution for $\mathrm{PCF_H}$ terms . . . 15
2.4 Operational semantics of $\mathrm{PCF_H}$ . . . 16
2.5 Type compatibility of $\mathrm{PCF_H}$ . . . 19
2.6 Type equivalence of $\mathrm{PCF_H}$ . . . 19
2.7 Type system of $\mathrm{PCF_H}$ (1): well-formedness rules . . . 20
2.8 Type system of $\mathrm{PCF_H}$ (2): compile-time typing rules . . . 21
2.9 Type system of $\mathrm{PCF_H}$ (3): run-time typing rules . . . 22

3.1 Syntax of PCFv . . . 28
3.2 Operational semantics of PCFv . . . 29
3.3 Syntax of $\mathrm{PCFv\Delta_H}$ . . . 29
3.4 Essence of a $\mathrm{PCFv\Delta_H}$ term . . . 30
3.5 Operational semantics of $\mathrm{PCFv\Delta_H}$ (1): essential evaluation . . . 31
3.6 Operational semantics of $\mathrm{PCFv\Delta_H}$ (2): reduction rules for dynamic checking 33
3.7 Operational semantics of $\mathrm{PCFv\Delta_H}$ (3): contextual rules for dynamic checking 34
3.8 Type system of $\mathrm{PCFv\Delta_H}$ (1): well-formedness rules . . . 35
3.9 Type system of $\mathrm{PCFv\Delta_H}$ (2): compile-time typing rules . . . 36
3.10 Type system of $\mathrm{PCFv\Delta_H}$ (3): run-time typing rules . . . 37

4.1 Examples of an informal interpretation of expressions. . . . 44
4.2 Syntax of $\lambda^{H\|^{\Phi}}$. . . . 47
4.3 Operational semantics of $\lambda^{H\|^{\Phi}}$ (1): deterministic part . . . 49
4.4 Operational semantics of $\lambda^{H\|^{\Phi}}$ (2): nondeterministic part . . . 50
4.5 Results . . . 50
4.6 Type compatibility of $\lambda^{H\|^{\Phi}}$ . . . 51
4.7 Type equivalence of $\lambda^{H\|^{\Phi}}$ . . . 51
4.8 Type system of $\lambda^{H\|^{\Phi}}$ (1): well-formedness rules . . . 52
4.9 Type system of $\lambda^{H\|^{\Phi}}$ (2): typing rules . . . 53
4.10 Type system of $\lambda^{H\|^{\Phi}}$ (3): run-time typing rules . . . 54
4.11 Example of typing capturing a free variable in an orthant in the middle . . . 55
4.12 Example of typing for an expression after distributing an function argument 55
4.13 Deterministic evaluation (1) . . . 57
4.14 Deterministic evaluation (2) . . . 58

5.1 Compilation of expressions . . . 66
5.2 Syntax of $\lambda^{\|}$ . . . 67
5.3 Operational semantics of $\lambda^{\|}$ . . . 68
5.4 Typing rules for $\lambda^{\|}$ . . . 68

5.5 Syntax of $\lambda^{\|\omega}$ . . . 69
5.6 Operational semantics of $\lambda^{\|\omega}$ . . . 70
5.7 Effect system of $\lambda^{\|\omega}$ (1): subtyping rules . . . 70
5.8 Effect system of $\lambda^{\|\omega}$ (2): well-formedness rules . . . 71
5.9 Effect system of $\lambda^{\|\omega}$ (3): typing rules . . . 72
5.10 Name erasure function . . . 75
5.11 Pseudo compilation . . . 76

1 Introduction

In this thesis, we deal with a software verification method, which helps to develop *correct* software. No one would doubt that it is pleasant that software works correctly. "Correctly" means software works as users *think* informally, but it is not hard to think how we formally define the correctness. Passing through this thesis, we assume that correct software means one which follows specifications. Furthermore, we assume that specifications themselves are correct. Consequently, a software verification method means a method that checks if software follows its specifications.

Although software consists of many parts, we mainly focus on correctness of each program code. For example, consider a division program whose specification is as follows.

> This program takes two natural numbers: one is called *dividend* and another is called *divisor*. A divisor must not be zero. As a result, this program returns a natural number called *quotient*. The number obtained by subtracting the product of the divisor and the quotient from the dividend is less than the divisor.

There are several things to take care: a program that uses this division program never passes zero as a divisor; this program calculates a number satisfies the condition in the last sentence; etc. In the old days, these kinds of things are left to programmers to write a correct program.

Nevertheless, automated verification methods emerged in the earlier history of software development since human beings make mistakes. Traditionally, the methods are roughly categorized into *run-time verification* and *compile-time verification*.

Run-time verification would be a better noticeable form to check correctness. For instance, C language [27] has an `assert` statement which checks a property in each program point at run-time. So, a division program with run-time verification could be implemented as follows in C language.

```
1  int div(int a, int b) {
2    int i;
3    int a_;
4    a_ = a;
5    assert (a >= 0);
6    assert (b > 0);
7    while (a_ > b) {
8      i++;
9      a_ -= b;
10   }
11   assert (0 <= a - b * i < b);
12   return i;
13 }
```

Of course, `assert` is very primitive—it just terminates a software execution at the point in which `assert` is placed if described properties do not hold, and hopefully it would put the location of the failed assertion on a display, a log file, or somewhere else.

Compile-time verification, in contrast, tends to be used unawares since it is highly integrated into a compiler and used automatically and forcibly. The most famous method of this category would be *type systems*, which are widely adopted in many programming languages and help developers everyday. For instance, the example code above already could receive a benefit of a type checker. A compiler reports an error at compile-time, which means before running a program, if a program that uses `div` function passes: only one argument; characters (not integers); and/or so on. What a type checker can detect would be very trivial things especially classical ones (like C language's one), but the long and wide use proves the usefulness.

In the following, we introduce more recent (and still developed) work for each verification method—*software contracts* for run-time verification and *refinement type systems* for compile-time verification; and even more, those are getting integrated. The reason we choose these topics as a representative for each is intentional because those are related more closely to this thesis; there is a lot of work taking a different approach though—e.g., model checking [30], abstract interpretation [12], test-based verification [8], certification by mathematical proofs [33], etc.

1.1 Background

1.1.1 Software Contracts

Software contracts are originally coined by Meyer [37, 38] in his methodology "Design by Contract"—specifications of software should be accompanied with source code because whether software is correct or not is only determined by the specifications. Specifications specify the duties for developers, what functions should be implemented, and the rights for users, what functions can be used and/or assumed. This is analogous to ordinary contracts and why we call software specifications software contracts.

Eiffel [37] should be the first programming language equipped with a full-fledged software contract system. It was used when Design by Contract methodology was introduced. In Eiffel, contracts mainly specify *pre*-conditions and *post*-conditions of methods. The following is an example of Eiffel code for a division function.

```
1 div (a, b: INTEGER) : INTEGER is
2   require
3     a >= 0
4     b > 0
5   do
6     ...
7   ensure
8     0 <= a - b * Result < b
9   end
```

Contracts for the division function is better organized than ones in the example code in C language. Pre-conditions, placed between `require` and `do`, specify the conditions held before the function call, and so a caller must keep the contracts. Post-condition(s), placed between `ensure` and `end`, specify the conditions held after the function call, and so a function provider must keep the contracts. Implementation code is placed between `do` and `ensure`, where no assertions anymore. This organization leads to more readable code and better error messages when a contract is violated.

Accompanied contracts are actually utilized for software verification, not just as documents. In origin (Eiffel), it is a bit obscure how contracts are checked; run-time checking would be the first candidate, though. However, in the later system in which a software contract system is supported (e.g., Java, C#, Racket [18]), it becomes clear that contracts are the one written in a runnable code and checked at run-time. So, for instance, a contract system checks if given integers for `div` function satisfy the pre-conditions by evaluating the pre-conditions as Boolean expressions; and if a returned value (which is stored in `Result`) satisfies the post-conditions by a similar manner. In this sense, we also consider a dynamic type system, used in Perl, Ruby, Javascript, etc., is a software contract system.

How contracts are expressed is one major issue about software contract systems. In Eiffel, it seems enough that contracts are just an enumeration of Boolean expressions since Eiffel only has first-order values. One interesting breakthrough in this perspective is *higher-order contracts* [17], which is implemented in Racket. In functional programming languages (including Racket), functions are also first-class values. So, it is a natural demand to be able to write contracts for functional values. In Racket, we can write a division function as follows.

```
1  (provide (contract-out
2            [div (->i ([a (>=/c 0)]
3                       [b (>/c 0)])
4                      [result (a b)
5                             (lambda (res)
6                               (and (<= 0 (- a (* b res)))
7                                    (< (- a (* b res)) b)))])]))
8  (define (div a b) ...)
```

This program provides `div` function, which is defined in the last line, with the accompanying contracts. The contracts are given in line 2–6 as one unified contract. `(->i P Q)` expresses a contract for functions, where `P` denotes contracts for arguments and `Q` denotes contracts for a result. "Provide" means that `div` function is carried around everywhere as a value. So, the contract is checked when the function is imported/exported. At a glance, it is impossible to check a functional contract even at run-time in general because we need to observe that all inputs and outputs follow the given contract. For example, to check if a given function between numbers always returns a positive number, we need to give all numbers and check if every returned value is positive. This impossibility is solved by using monitoring instead of actually checking. That means contract checking for a functional value makes a new function instead of examining the functional value. The new function is obtained by wrapping the functional value with checking code for input and output. As a drawback, contract violation cannot be detected until the new function is applied for a *good*, which means the corresponding returned value is against the result part of the functional contracts, argument.

Someone might wonder if Racket style code loses some information, comparing Eiffel style code; namely what is a pre-conditions and what is a post-conditions? Actually, it does not; it can be found by examining where a contract occurs—in short, contracts for arguments are pre-conditions and contracts for a result are post-conditions. In general, it is difficult to determine who is blamed (users or providers?) when a contract is violated; and so it is also a subject of study for software contract systems as known as a blame assignment problem [3, 14, 26].

1.1.2 Refinement Type Systems

Most type systems widely used can only check a syntactic property of program code—like inconsistency of the number of parameters between call-side and caller-side, invalid method call which does not exist, etc. To make more properties be able to check, type systems are still studied actively for many directions.

Refinement type systems [51, 58, 31, 57, 66, 61] are one group extending a simple type system. The main device characterizing refinement type systems is *refinement types*, which are types equipped with a predicate describing a property of values belonging to the type. Refinement types are usually denoted as $\{x\mathtt{:}\tau \mid M\}$—like a set comprehension form, where τ is called *underlying type* and M is a predicate describing a property by using a bound variable x. For instance, $\{x\mathtt{:}\mathtt{int} \mid x > 0\}$ represents a type of positive numbers. A refinement type system is usually a *dependently typed system* [35], where types can depend on run-time values. Dependent function types, we denote $(x\mathtt{:}\sigma) \to \tau$ in this thesis, are another characteristic types of a dependently typed system. A dependent function type is a type of functions whose parameter type is σ and result type is τ; but the distinguishing point is what the result type, which typically involves a refinement type, can argue a given argument by using a bound variable x. Using types that we have just introduced, the contracts of a division function can be expressed as the following type.

$$(a\mathtt{:}\{x\mathtt{:}\mathtt{int} \mid x \geq 0\}) \to (b\mathtt{:}\{y\mathtt{:}\mathtt{int} \mid y > 0\}) \to \{z\mathtt{:}\mathtt{int} \mid 0 \leq a - b \times z < b\}$$

The first two refinement types express the contracts for the parameters of a division function—the former is a type of natural numbers and the latter is a one of positive numbers. The result type depends on given arguments. If a function of this type is applied for 4 and 2, the result type becomes $\{z\mathtt{:}\mathtt{int} \mid 0 \leq 4 - 2 \times z < 2\}$, which express the singleton set only containing 2. Hence, the type expresses the contracts quite precisely.

The following is example code[1] for `div` function in F* language [56], which is a ML dialect equipped with a refinement type system.

```
1  val div: a:int{a>=0} -> b:int{b>0} -> z:int{0<=a-b*z && a-b*z<b}
2  let div a b = ...
```

The first two lines describe the contract and the last line defines the function. So it follows the same style as Racket code. However, there is an important difference, that is, the contract is checked at compile-time as type checking and no cost happens at run-time. Unfortunately, this code will be rejected by F* compiler even when the definition is correct because it could not decided automatically the body of `div` function has the result type. Putting the sad result aside, consider the type checking for the caller code as follows.

```
1  let r = div 0 0
```

A type checker synthesizes the type of `div`, which is given by a programmer as a type annotation as we have seen because only a programmer knows specifications. So it is an easy task. Next, the type checker synthesize a type of 0, which will be the singleton type $\{x\mathtt{:}\mathtt{int} \mid x = 0\}$. This is also an easy task since 0 is a constant, and it is just a syntactic operation for any integers n to synthesize the corresponding singleton type $\{x\mathtt{:}\mathtt{int} \mid x = n\}$. Difficulty comes in the next step. A simple type system next checks if a parameter type and

[1] Syntax of types are a bit different from one in this thesis, but the meaning could be reasonable.

an argument type is the same. However, in this case, those are different ($\{x\colon \texttt{int} \mid x \geq 0\}$ and $\{x\colon \texttt{int} \mid x = 0\}$ for the first parameter). So, should the type checker reject the program? Of course, no (as far as the first argument). To accept this difference, a refinement type system has a typing rule called *subsumption rule*—an expression can have a more *large* type. In this case, the type checker checks if $\forall x : \texttt{int}, x = 0 \Rightarrow x \leq 0$ holds, which shows the subset relation between the two types. A recent theorem prover [41] can prove the formula.[2] Thanks to the result, 0 can have the type $\{x\colon \texttt{int} \mid x \geq 0\}$; and the first application is accepted as a simple type system accepts. Contrary, the second application is rejected (and thus the program is rejected) because $\{x\colon \texttt{int} \mid x = 0\}$ is not a subset of $\{x\colon \texttt{int} \mid x > 0\}$, which could be also shown automatically by proving a corresponding formula.

There is another main difficulty for which many studies contribute. That is it sometimes happens that a type checker synthesize contracts which not explicitly appear in a program to check written contracts. It is analogous to what we are forced to consider weaker and/or stronger statements than one written in a mathematical problem to show that.

At the beginning of studies, the form of predicates are carefully chosen so that a type system becomes usable. Fortunately, active work for theorem provers make it possible that more general predicates could be used [51, 58, 31, 57, 66, 61]; and types are getting closer to general software contracts.

1.1.3 Hybrid Type Checking

Getting a use of sophisticated static verifier more realistic, which could prove various properties but is incomplete—some have false-positives and some still impose restrictions on predicates, verification methods which combines compile- and run-time checking has been appeared to compensate for the incompleteness.

Hybrid type checking [19, 28] is one of the combined methods for a refinement type system, and therefore, it can be seen as one milestone combining software contracts and static type systems. In the system, following the software contracts way, we can use arbitrary predicate, which is written in the programming language itself, for refinement types. The following is an ideal example source code passed to the hybrid type checking.

```
1  let div (a : {x:int|x ≥ 0}) (b : {y:int|y > 0})
2    : {z:int|0 ≤ a - b * z < b} =
3    ...
```

As we have mentioned, current automated solver can hardly decide the body can have the result type, and so the code rejected. At the same time, it is also true that an automated solver can hardly decide the body *cannot* have the result type. So this is a different situation from what the caller code in the previous subsection is rejected (which actually violates a contract). In this situation, hybrid type checking inserts a *cast* and accept the code as follows since the code would be correct.

```
1  let div (a : {x:int|x ≥ 0}) (b : {y:int|y > 0})
2    : {z:int|0 ≤ a - b * z < b} =
3    (... : int ⇒ {z:int|0 ≤ a - b * z < b})
```

[2]Actually many refinement type system just rely on an existing theorem prover after synthesizing a required formula.

A cast ($M : \sigma \Rightarrow \tau$) checks whether the evaluated value of M of type σ can have the type τ at run-time. (So the code above represents the worst case—the body has just `int` type.) If the check fails—in this case, the implementation of `div` function is incorrect and returns a wrong number, 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 (unless no casts are inserted during compilation), but it guarantees that the result of successful execution satisfies the predicate of a refinement type in the program's type.

Manifest Contracts

Recently, a target language of the translation of hybrid type checking is split out as a *manifest contract system* [53, 52, 54, 1, 20, 45] because of technical reasons. The type system of a manifest contract system is similar to one used in hybrid type checking, but the type system never examine the meaning of predicates; which means the type system has no subsumption rule, and therefore, casts are mandatory everywhere required. That means the caller code for `div` function must be as follows, and the following code accepted by the type system of a manifest contracts system while the second application violates a contract, which will be detected at run-time.

```
1  div (0 : {x:int|x = 0} ⇒ {x:int|x >= 0})
2      (0 : {x:int|x = 0} ⇒ {x:int|x > 0})
```

Nevertheless, static contract checking is still established in a manifest contract system as a post optimization, which is done after type checking, called *up-cast elimination*—a cast into a super type can be removed without changing a program behavior.

1.2 This Thesis

Our goal is to develop a theory for software contracts on typed functional programming languages with keeping in mind hybrid type checking. Concretely, we enrich manifest contracts for: *conjunctive contracts* and *nondeterministic contracts*.

Conjunctive contracts It is natural to think that we consider a contract stated in a conjunctive form. For example, one would want to give a contract for a variant of a division function as follows.

> .... This program also can take negative numbers as a divisor. In that case, the number obtained by subtracting dividend from the product of divisor and quotient is grater than divisor and less than or equal to zero.

In fact, this contract can also be expressed by using dependent function types as follows.

$$(a{:}\{x{:}\texttt{int} \mid x \geq 0\}) \rightarrow (b{:}\{y{:}\texttt{int} \mid y \neq 0\}) \\ \rightarrow \{z{:}\texttt{int} \mid \texttt{if}\ b > 0\ \texttt{then}\ 0 \leq a - b \times z < b\ \texttt{else}\ b < b \times z - a \leq 0\}$$

However, this method has a shortcoming. That is there is a gap between the contracts in natural language and the representation in types, which becomes a burden for programmers

and will produce an error when we write contracts.[3] More seriously, this method cannot be applied for some contracts of higher-order functions. Consider the following higher-order program.

```
1  let g f x = - (f (- x))
```

The function g respects the following two contracts.

$$(\{x{:}\texttt{int} \mid x > 0\} \to \{x{:}\texttt{int} \mid x > 0\}) \to \{x{:}\texttt{int} \mid x < 0\} \to \{x{:}\texttt{int} \mid x < 0\}$$
$$(\{x{:}\texttt{int} \mid x < 0\} \to \{x{:}\texttt{int} \mid x < 0\}) \to \{x{:}\texttt{int} \mid x > 0\} \to \{x{:}\texttt{int} \mid x > 0\}$$

Now we cannot express the conjunctive contract of those as a dependent function type.

Carefully examining the meaning of the contracts, some experts still wonder if the conjunctive contracts can be expressed as follows.

$$\begin{aligned} &(f{:}(a{:}\{x{:}\texttt{int} \mid x \neq 0\}) \to \{y{:}\texttt{int} \mid \texttt{if}\ a > 0\ \texttt{then}\ y > 0\ \texttt{else}\ y < 0\}) \\ \to\ &(x{:}\{x{:}\texttt{int} \mid x \neq 0\}) \\ \to\ &\{z{:}\texttt{int} \mid \texttt{if}\ x < 0\ \&\&\ f\,(-x) < 0\ \texttt{then}\ z < 0 \\ &\qquad\quad \texttt{else if}\ x > 0\ \&\&\ f\,(-x) > 0\ \texttt{then}\ z > 0 \\ &\qquad\quad \texttt{else False}\} \end{aligned}$$

This type does not express the conjunctive contract precisely. Actually, the contract for the first parameter means that a caller must give a function which can deal with non-zero integers, while giving a function which can deal with only positive or negative integers suffices in the conjunctive contract.[4]

Recently, *intersection contracts* had been proposed [26, 64] in a software contracts area to express complicated contracts like one introduced above. They provide a new binary contract combinator $\cap$ to express the conjunction of two contracts and how to monitor such contracts. Similarly, *intersection types*, which can give two types to one code, are used at some refinement type systems [57, 66] to handle the situation in which complicated contracts like one introduced above are required during verification. So it is a natural and important demand to integrate intersection contracts into a manifest contract system.

Nondeterministic contracts *Nondeterminism* is a powerful tool for programming. It is especially useful when we know *what* we are going to implement but not *how* we do. For example, consider implementing a function that takes a list of natural numbers and returns a pair of natural numbers that are taken from the list and whose sum is a prime number.

```
1  let prime_sum_pair l =
2    let a1 = choose l and a2 = choose l in
3    require (isprime (a1 + a2));
4    (a1, a2)
```

Here, `choose` returns an element of a given list nondeterministically, `require` M aborts an execution unless the condition M holds, and `isprime` is a Boolean function that returns whether its argument is prime or not. So, the defined function nondeterministically returns

[3] We have assumed that contracts themselves are correct though.

[4] A user can choose one of contracts constituting a conjunctive contract.

a pair of the expected property. The remarkable point is that we are almost free from considering how to choose proper elements from the given list and just write down the specifications into the code.

An interesting point of nondeterminism under a manifest contracts system is how we construct a dependently typed system. That is because nondeterminism is a kind of side-effect, which cannot be naively integrated into a dependently typed system. A usual solution [52, 9] for an integration is imposing some restriction on program code so that types only depend on *pure* code that is always evaluated into the same value under any environment. However, we challenge an unrestricted integration in this thesis so that we do not kill an advantage of software contracts—programmers can use any code for writing contracts.

1.2.1 Contributions

To achieve the goal, we study two manifest contract systems and give a supplemental discussion which gives an investigation for filling a gap between the two systems and the goal.

Manifest Contracts with Intersection Types *Intersection types*, denoted as $\sigma \wedge \tau$ in this thesis, are one kind of polymorphic types, which can give various types to one code. More specifically, code of $\sigma \wedge \tau$ can be used as of both σ and τ; and code must have both σ and τ so that the code has $\sigma \wedge \tau$. So, using intersection types, we can express a conjunctive contract by just connecting each contract with $\wedge$, no matter how complicated it is, for instance, as follows.

$$\begin{aligned}&(\{x{:}\mathtt{int} \mid x > 0\} \to \{x{:}\mathtt{int} \mid x > 0\}) \to \{x{:}\mathtt{int} \mid x < 0\} \to \{x{:}\mathtt{int} \mid x < 0\}\\ \wedge&(\{x{:}\mathtt{int} \mid x < 0\} \to \{x{:}\mathtt{int} \mid x < 0\}) \to \{x{:}\mathtt{int} \mid x > 0\} \to \{x{:}\mathtt{int} \mid x > 0\}\end{aligned}$$

We present a manifest contract system $\mathrm{PCFv}\Delta_{\mathrm{H}}$, which equipped with intersection types. Intersection types naturally arise when a contract is expressed by a conjunction of smaller contracts. As we mentioned, run-time contract checking for conjunctive higher-order contracts in an untyped language has been studied [26, 64] but our typed setting poses an additional challenge because 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.

Nondeterministic Manifest Contracts We study a manifest contract system with *nondeterministic choice*. The extension is not trivial, especially in the presence of dependent function types, because a naive extension would lead to inconsistent type equivalence, which makes contract information in refinement types meaningless.

To solve the problem, we propose a new kind of nondeterministic choice called *coordinated choice*, in which each occurrence of a choice operator is given a name and choices of the same name coordinately take the same branch. We introduce the notion of *orthant* that helps both intuitive understanding and the development of formal semantics of the new choice.

We formalize a manifest contract system $\lambda^{H\parallel^{\Phi}}$ using the coordinated choice and show its basic properties of progress, type preservation, and contract satisfaction, the last of which states correctness of contracts in refinement types.

Compilation of Coordinated Choice As we will see, PCFvΔ_{H} lacks dependent function types because of the problem caused by its nondeterministic semantics. The problem is solved by $\lambda^{H\parallel^{\Phi}}$. However, $\lambda^{H\parallel^{\Phi}}$ uses coordinated choice for nondeterminism, which requires careful treatment. To fill the gap, we give the compilation method from a language equipped with usual nondeterministic choices into one equipped with coordinated choices.

The main contribution of this discussion is the compilation method and its correctness, which means the run-time behavior of code before and after compilation corresponds.

1.3 Organization

The rest of the thesis organized as follows: In Chapter 2, we review a typical manifest contract system. In Chapter 3, we integrate intersection types into manifest contracts to give an ability to write conjunctive contracts. In Chapter 4, we build a manifest contract system on a nondeterministic calculus. In the last technical chapter, Chapter 5, we demonstrate how a usual nondeterministic choice is simulated by a *coordinated choice*, which introduced in Chapter 4. Lastly, in Chapter 6, we introduce related work; and we conclude this thesis and put future work in Chapter 7.

Chapter 3, Chapter 4, and Chapter 5 constitute the original work of this thesis, the first two have been submitted as papers and presented at APLAS 2019 [44] and PPDP 2018 [45], respectively.

Most of meta-properties (the whole part of Chapter 3 and Chapter 4; and a large part of Chapter 5) are mechanized by Coq—a proof assistant system. So, we only show important lemmas and proof sketches obtained by the proof code in this thesis. Proof scripts could be obtained by the following URL, respectively.

- `https://gitlab.com/westpaddy/coq-proof-deltah`
- `https://gitlab.com/westpaddy/coq-proof-nmc`
- `https://gitlab.com/westpaddy/coq-proof-cochoice`

2 Manifest Contract System

Manifest contract systems [21, 62, 53, 52, 54, 1, 20, 22, 19, 28, 45], which are typed functional calculi, are one discipline handling *software contracts* [38]. 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 $\mathtt{int} \rightarrow \{x{:}\mathtt{int} \mid x \neq 0\} \rightarrow \mathtt{int}$. 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 $(x{:}\sigma) \rightarrow \tau$ in this thesis, 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 $(x{:}\mathtt{int}) \rightarrow (y{:}\{x'{:}\mathtt{int} \mid x' \neq 0\}) \rightarrow \{z{:}\mathtt{int} \mid x = z \times y\}$. (Here, for simplicity, we ignore the case where division involves a remainder, though it can be taken account into by writing a more sophisticated predicate.)

As we have discussed, all contracts are dynamically checked by 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] The type system of a manifest contract system only checks *syntactic consistency*. The following code shows various casts required.

```
1  let x : {x:int|x>0} = ... in
2  let y : {x:int|0<x} = ... in
3  let f (x:{x:int|x>0}) = ... in
4  f (1 : {x:int|x=1} => {x:int|x>0});
5  f (0 : {x:int|x=0} => {x:int|x>0});
6  f (y : {x:int|0<x} => {x:int|x>0});
7  f x
```

Every cast in line 4 to 6 is required in a manifest contract system since the source types and target types are syntactically different, while recent static verifier could find the first and third casts never fail and the second fails. Only the application in line 7 does not require a cast since the type of `x` and the type of the parameter of `f` are syntactically the same. 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.

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

$$
\begin{array}{rcl}
\sigma,\tau & ::= & \texttt{bool} \mid \texttt{nat} \mid (x{:}\sigma) \to \tau \mid \{x{:}\tau \mid M\} \\
L,M,N & ::= & x \mid \texttt{True} \mid \texttt{False} \mid \texttt{0} \mid \texttt{succ}(M) \mid \texttt{pred}(M) \mid \texttt{iszero}(M) \mid \\
& & \texttt{if}\, L\, \texttt{then}\, M\, \texttt{else}\, N \mid M\,N \mid \lambda x{:}\tau.M \mid \mu f{:}\tau.M \mid (M : \sigma \Rightarrow \tau) \mid \\
& & \langle\langle M \mathbin{?} \{x{:}\tau \mid N\}\rangle\rangle \mid \langle\langle M \Longrightarrow V : \{x{:}\tau \mid N\}\rangle\rangle \mid \texttt{blame} \\
\overline{n} & ::= & \texttt{0} \mid \texttt{succ}(\overline{n}) \\
U,V & ::= & \texttt{True} \mid \texttt{False} \mid \overline{n} \mid \lambda x{:}\tau.M \\
\mathcal{E} & ::= & \texttt{succ}(\Box) \mid \texttt{pred}(\Box) \mid \texttt{iszero}(\Box) \mid \texttt{if}\,\Box\,\texttt{then}\,M\,\texttt{else}\,N \mid \Box M \mid \\
& & V\Box \mid (\Box : \sigma \Rightarrow \tau) \mid \langle\langle \Box \mathbin{?} \{x{:}\tau \mid M\}\rangle\rangle \mid \langle\langle \Box \Longrightarrow V : \{x{:}\tau \mid M\}\rangle\rangle \\
\Gamma & ::= & \emptyset \mid \Gamma, x{:}\tau
\end{array}
$$

Figure 2.1: Syntax of $\mathrm{PCF_H}$

This property follows from subject reduction and a property called *value inversion* [54]—*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* $\texttt{True}$.

In this chapter, we demonstrate a typical manifest contract system as $\mathrm{PCF_H}$ by simplifying and reforming an existing manifest contract system $\lambda^{H}_{\mathrm{dt}}$ [54]. Passing through this chapter, we will see how run-time checking is done in a manifest contract system and what kind of properties are considered for a manifest contract system.

2.1 Language

$\mathrm{PCF_H}$ is a simple manifest contract system for PCF [47], a simply typed lambda calculus with Boolean, natural numbers, and a fix-point operator. Types are extended with refinement types and dependent function types, and expressions are extended with casts and run-time expressions used for run-time checking—those are considered as minimal extensions to implement a manifest contract system.

2.1.1 Syntax

The syntax of $\mathrm{PCF_H}$ is shown in Figure 2.1. Meta-variables x, y, z, f, g range over term variables, σ, τ range over *types*, L, M, N range over *expressions*, $\overline{n}$ ranges over *numeral values*, U, V range over *values*, $\mathcal{E}$ ranges over *evaluation contexts*, and Γ ranges over *typing environments*. We also use meta-variables primed and/or indexed by number, e.g., x', τ_1, etc.

Types consist of *ground types* $\texttt{bool}$ and $\texttt{nat}$, types for Boolean and natural numbers, respectively; dependent function types $(x{:}\sigma) \to \tau$, in which x is bound in τ; and refinement types $\{x{:}\tau \mid M\}$, in which x is bound in M.

Expressions consist of usual PCF expressions, namely truth values $\texttt{True}$ and $\texttt{False}$, the numeral constant $\texttt{0}$, the successor function $\texttt{succ}(M)$ which returns the next numeral of M, the predecessor function $\texttt{pred}(M)$ which returns the previous numeral of M, Boolean predicate $\texttt{iszero}(M)$ which checks if the value of M is zero or not, if expression $\texttt{if}\ L\ \texttt{then}\ M\ \texttt{else}\ N$, function applications $M\,N$, functions $\lambda x{:}\tau.M$, and fix-point operator $\mu f{:}\tau.M$; casts $(M : \sigma \Rightarrow \tau)$, which cast the subject M of the source type σ into the target type τ; *waiting checks* $\langle\langle M \mathbin{?} \{x{:}\tau \mid N\}\rangle\rangle$, which wait the run-time check of the subject M against the predicate N until M becomes a value; *active checks* $\langle\langle M \Longrightarrow V : \{x{:}\tau \mid N\}\rangle\rangle$, which express an intermediate state of a predicate evaluation (M holds an intermediate

state of an actual evaluation of the predicate N for the subject V); and *blame* `blame`, which denotes a contract violation. The last three are called *run-time expressions*—those occur during run-time and not source code. Note that we omit labels which indicate which cast has failed, leaving blame assignment [62].

Values are as usual: Boolean values `True`, `False`; numerals $\overline{n}$; and functions $\lambda x{:}\tau.M$. Note that natural numbers usually denoted by the numerals in a PCF system as follows.

$$n \stackrel{\text{def}}{=} \underbrace{\texttt{succ}(\ldots\texttt{succ}(}_{n}\texttt{0})\ldots)$$

An evaluation context is an expression with a single hole and used to indicate which sub-expression is evaluated. We write $\mathcal{E}[M]$ for the expression obtained by replacing the hole in $\mathcal{E}$. Note that, unlike standard formulations, the definition is not recursive; a list of our evaluation contexts (which are also called *evaluation frames* in the literature [46]) corresponds to a standard evaluation context.

Typing environments (or just environments) are defined as lists of *type bindings* $x{:}\tau$ because types in an environment can depend on former type bindings of those.

Definition 2.1.1 (Terms)**.** We call the union of types and expressions *terms.*

Convention 2.1.2. We identify α-equivalent terms, which only differ in their bound variables.

Definition 2.1.3 (Free variables)**.** A variable that is not bound is called *free variable.* The set of free variables in τ and M, denoted by $\mathrm{fv}(\tau)$ and $\mathrm{fv}(M)$, respectively, is defined in Figure 2.2.

Definition 2.1.4 (Closed terms)**.** A term that has no free variables is called *closed.*

Definition 2.1.5 (Substitutions)**.** We define capture-avoiding *substitution* of L for z in τ and M, written $\tau[z := L]$ and $M[z := L]$, respectively, as Figure 2.3; assuming $x \notin \mathrm{fv}(L)$, $x \neq z$, and $y \neq z$. Thanks to Convention 2.1.2, the assumption can be satisfied for any terms.

Convention 2.1.6. We usually omit the empty environment if there is no ambiguity. We abuse commas for concatenation of environments, i.e., Γ_1, Γ_2. We denote a singleton environment by just a type binding, i.e., $x{:}\tau$.

Definition 2.1.7 (Domain of typing environment)**.** The *domain* of Γ, denoted by $\mathrm{dom}(\Gamma)$, is defined as follows.

$$\mathrm{dom}(\varnothing) = \varnothing$$
$$\mathrm{dom}(\Gamma, x{:}\tau) = \mathrm{dom}(\Gamma) \cup \{x\}$$

Definition 2.1.8 (Substitution for typing environment)**.** Capture-avoiding substitutions lift for a typing environment as follows.

$$\varnothing[z := L] = \varnothing$$
$$(\Gamma, x{:}\tau)[z := L] = \Gamma[z := L], x{:}\tau[z := L]$$

$$
\begin{aligned}
\mathrm{fv}(\mathtt{bool}) &= \varnothing \\
\mathrm{fv}(\mathtt{nat}) &= \varnothing \\
\mathrm{fv}((x{:}\sigma) \to \tau) &= \mathrm{fv}(\sigma) \cup (\mathrm{fv}(\tau) \setminus \{x\}) \\
\mathrm{fv}(\{x{:}\tau \mid M\}) &= \mathrm{fv}(\tau) \cup (\mathrm{fv}(M) \setminus \{x\}) \\
\mathrm{fv}(x) &= \{x\} \\
\mathrm{fv}(\mathtt{True}) &= \varnothing \\
\mathrm{fv}(\mathtt{False}) &= \varnothing \\
\mathrm{fv}(\mathtt{0}) &= \varnothing \\
\mathrm{fv}(\mathtt{succ}(M)) &= \mathrm{fv}(M) \\
\mathrm{fv}(\mathtt{pred}(M)) &= \mathrm{fv}(M) \\
\mathrm{fv}(\mathtt{iszero}(M)) &= \mathrm{fv}(M) \\
\mathrm{fv}(\mathtt{if}\ L\ \mathtt{then}\ M\ \mathtt{else}\ N) &= \mathrm{fv}(L) \cup \mathrm{fv}(M) \cup \mathrm{fv}(N) \\
\mathrm{fv}(M\,N) &= \mathrm{fv}(M) \cup \mathrm{fv}(N) \\
\mathrm{fv}(\lambda x{:}\tau.M) &= \mathrm{fv}(\tau) \cup (\mathrm{fv}(M) \setminus \{x\}) \\
\mathrm{fv}(\mu f{:}\tau.M) &= \mathrm{fv}(\tau) \cup (\mathrm{fv}(M) \setminus \{f\}) \\
\mathrm{fv}((M : \sigma \Rightarrow \tau)) &= \mathrm{fv}(M) \cup \mathrm{fv}(\sigma) \cup \mathrm{fv}(\tau) \\
\mathrm{fv}(\langle\!\langle M \mathrel{?} \{x{:}\tau \mid N\}\rangle\!\rangle) &= \mathrm{fv}(M) \cup \mathrm{fv}(\{x{:}\tau \mid N\}) \\
\mathrm{fv}(\langle\!\langle M \Longrightarrow V : \{x{:}\tau \mid N\}\rangle\!\rangle) &= \mathrm{fv}(M) \cup \mathrm{fv}(V) \cup \mathrm{fv}(\{x{:}\tau \mid N\}) \\
\mathrm{fv}(\mathtt{blame}) &= \varnothing
\end{aligned}
$$

Figure 2.2: Free variables of $\mathrm{PCF_H}$ terms

$$
\begin{aligned}
\texttt{bool}[z \coloneqq L] &= \texttt{bool} \\
\texttt{nat}[z \coloneqq L] &= \texttt{nat} \\
((x\!:\!\sigma) \to \tau)[z \coloneqq L] &= (x\!:\!\sigma[z \coloneqq L]) \to \tau[z \coloneqq L] \\
\{x\!:\!\tau \mid M\}[z \coloneqq L] &= \{x\!:\!\tau[z \coloneqq L] \mid M[z \coloneqq L]\} \\
y[z \coloneqq L] &= y \\
z[z \coloneqq L] &= L \\
\texttt{True}[z \coloneqq L] &= \texttt{True} \\
\texttt{False}[z \coloneqq L] &= \texttt{False} \\
\texttt{0}[z \coloneqq L] &= \texttt{0} \\
\texttt{succ}(M)[z \coloneqq L] &= \texttt{succ}(M[z \coloneqq L]) \\
\texttt{pred}(M)[z \coloneqq L] &= \texttt{pred}(M[z \coloneqq L]) \\
\texttt{iszero}(M)[z \coloneqq L] &= \texttt{iszero}(M[z \coloneqq L]) \\
(\texttt{if}\ M\ \texttt{then}\ N_1\ \texttt{else}\ N_2)[z \coloneqq L] &= \texttt{if}\ M[z \coloneqq L]\ \texttt{then}\ N_1[z \coloneqq L]\ \texttt{else}\ N_2[z \coloneqq L] \\
(M\,N)[z \coloneqq L] &= (M[z \coloneqq L])\,(N[z \coloneqq L]) \\
(\lambda x\!:\!\tau.M)[z \coloneqq L] &= \lambda x\!:\!\tau[z \coloneqq L].M[z \coloneqq L] \\
(\mu x\!:\!\tau.M)[z \coloneqq L] &= \mu x\!:\!\tau[z \coloneqq L].M[z \coloneqq L] \\
(M : \sigma \Rightarrow \tau)[z \coloneqq L] &= (M[z \coloneqq L] : \sigma[z \coloneqq L] \Rightarrow \tau[z \coloneqq L]) \\
\langle\!\langle M\ ?\ \{x\!:\!\tau \mid N\}\rangle\!\rangle[z \coloneqq L] &= \langle\!\langle M[z \coloneqq L]\ ?\ \{x\!:\!\tau[z \coloneqq L] \mid N[z \coloneqq L]\}\rangle\!\rangle \\
\langle\!\langle M \Longrightarrow V : \{x\!:\!\tau \mid N\}\rangle\!\rangle[z \coloneqq L] &= \langle\!\langle M[z \coloneqq L] \Longrightarrow V[z \coloneqq L] : \{x\!:\!\tau[z \coloneqq L] \mid N[z \coloneqq L]\}\rangle\!\rangle \\
\texttt{blame}[z \coloneqq L] &= \texttt{blame}
\end{aligned}
$$

Figure 2.3: Capture-avoiding substitution for $\mathrm{PCF_H}$ terms

$$\frac{}{\texttt{pred}(\texttt{succ}(\overline{n})) \longrightarrow \overline{n}} \quad \text{(R-Pred)}$$

$$\frac{}{\texttt{iszero}(0) \longrightarrow \texttt{True}} \quad \text{(R-IsZeroT)}$$

$$\frac{}{\texttt{iszero}(\texttt{succ}(\overline{n})) \longrightarrow \texttt{False}} \quad \text{(R-IsZeroF)}$$

$$\frac{}{\texttt{if True then } M \texttt{ else } N \longrightarrow M} \quad \text{(R-IfT)}$$

$$\frac{}{\texttt{if False then } M \texttt{ else } N \longrightarrow N} \quad \text{(R-IfF)}$$

$$\frac{}{(\lambda x{:}\tau.M)\, V \longrightarrow M[x \coloneqq V]} \quad \text{(R-Beta)}$$

$$\frac{}{\mu f{:}\sigma.\lambda x{:}\tau.M \longrightarrow (\lambda x{:}\tau.M)[f \coloneqq \mu f{:}\sigma.\lambda x{:}\tau.M]} \quad \text{(R-Fix)}$$

$$\frac{}{(V : \texttt{bool} \Rightarrow \texttt{bool}) \longrightarrow V} \quad \text{(R-CBool)}$$

$$\frac{}{(V : \texttt{nat} \Rightarrow \texttt{nat}) \longrightarrow V} \quad \text{(R-CNat)}$$

$$\frac{(x \neq y)}{(V : (x{:}\sigma_1) \to \sigma_2 \Rightarrow (y{:}\tau_1) \to \tau_2) \longrightarrow \lambda y{:}\tau_1.(\lambda x{:}\sigma_1.(V\,x : \sigma_2 \Rightarrow \tau_2))\,(y : \tau_1 \Rightarrow \sigma_1)} \quad \text{(R-CArrow)}$$

$$\frac{}{(V : \{x{:}\sigma \mid M\} \Rightarrow \tau) \longrightarrow (V : \sigma \Rightarrow \tau)} \quad \text{(R-CForget)}$$

$$\frac{(\sigma \neq \{x'{:}\sigma' \mid M'\})}{(V : \sigma \Rightarrow \{x{:}\tau \mid M\}) \longrightarrow \langle\!\langle (V : \sigma \Rightarrow \tau)\ ?\ \{x{:}\tau \mid M\} \rangle\!\rangle} \quad \text{(R-CWait)}$$

$$\frac{}{\langle\!\langle V\ ?\ \{x{:}\tau \mid M\} \rangle\!\rangle \longrightarrow \langle\!\langle M[x \coloneqq V] \Longrightarrow V : \{x{:}\tau \mid M\} \rangle\!\rangle} \quad \text{(R-CActive)}$$

$$\frac{}{\langle\!\langle \texttt{True} \Longrightarrow V : \{x{:}\tau \mid M\} \rangle\!\rangle \longrightarrow V} \quad \text{(R-CSucceed)}$$

$$\frac{}{\langle\!\langle \texttt{False} \Longrightarrow V : \{x{:}\tau \mid M\} \rangle\!\rangle \longrightarrow \texttt{blame}} \quad \text{(R-CFail)}$$

$$\frac{M \longrightarrow N}{\mathcal{E}[M] \longrightarrow \mathcal{E}[N]} \quad \text{(R-Ctx)}$$

$$\frac{}{\mathcal{E}[\texttt{blame}] \longrightarrow \texttt{blame}} \quad \text{(R-Exit)}$$

Figure 2.4: Operational semantics of $\mathrm{PCF_H}$

2.1.2 Semantics

The semantics of $\mathrm{PCF_H}$ is defined as small-step reduction relation between closed expressions M and N, written $M \longrightarrow N$ and read "M steps into N in one step", defined by the *inference rules* in Figure 2.4. Each rule consists of *premises* above the bar and *conclusion* below the bar, and says *if the premises hold the conclusion holds.* So, for example, the first rule (R-Pred), defines that $\mathtt{pred}(\mathtt{succ}(\overline{n}))$ unconditionally (since the rule has no premises) steps into $\overline{n}$ in one step. The rules are split into three categories: PCF evaluation; run-time checking; and contextural evaluation.

PCF evaluation consists of the first seven rules. Most rules are standard, but there are two notes. One is that $\mathrm{PCF_H}$ does not have a reduction rule for $\mathtt{pred}(0)$; while an usual PCF-like system has a reduction rule, e.g., $\mathtt{pred}(0) \longrightarrow 0$. $\mathrm{PCF_H}$ can deal with such a partial function by refinement types, ensuring a well-typed expression does not reach such an expression. (As we can see from the type system, applying $\mathtt{pred}$ to a natural number requires a nonzero check in terms of a cast, which may result in blame.) Another is that the body of a fixpoint operator is restricted to a lambda abstraction.

Run-time checking consists of the next eight rules. A dynamic checking starts from pealing the refinements of the source type by (R-CForget) if exists because those are useless[2] for run-time checking. Note that (R-CWait) cannot apply until the pealing has finished because of its side-condition (and therefore the semantics becomes deterministic). Next, we can use one of four rules (R-CBool), (R-CNat), (R-CArrow), and (R-CWait). By the first two rules, a cast between the same ground types is just removed. The third rule is for a higher-order contract checking, which wraps a function being cast with monitors (casts) for input and output. The side-condition exists so that lambda abstractions after the reduction do not capture unintended variables[3]—e.g., the inner abstraction $\lambda x{:}\sigma_1.\ \ldots$ will capture y in τ_2, which must be captured by the outer abstraction. The last rule does an actual evaluation of the predicate of the target type; but before the evaluation, it is checked if the subject can have the underlying type of the target refinement type since τ could be a refinement type in general. After the nested check has (successfully) finished, expression goes into a predicate evaluation state by (R-CActive); and as a result of the predicate evaluation, verified value is obtained by (R-CSucceed) if the predicate is satisfied or a contract violation is reported by (R-CFail) otherwise.[4] Note that the refinement type annotated in an active check is not used for run-time checking at all; it is just an annotation and technically used for showing meta properties.

The rest of the rules (R-Ctx) and (R-Exit) are for sub-expression evaluation and non-local exit of $\mathtt{blame}$, respectively. Taking $\mathcal{E} = \square N$ for instance, (R-Ctx) says the function part of an application is evaluated, namely $M N \longrightarrow M' N$. Taking $\mathcal{E} = V \square$ for another instance, $V M \longrightarrow V M'$. These two instances show that the evaluation contexts specify the order of evaluation between sub-expressions of an application—the function part is evaluated first and the argument part is never evaluated until the function part becomes a value.

Definition 2.1.9 (Multi-step reduction)**.** Multi-step reduction, written $M \longrightarrow^* N$, is defined as the reflexive and transitive closure of $\longrightarrow$.

[2]Honestly speaking, it depends on what kinds of contracts can be written by the predicate whether refinements are useless of not.

[3]This condition suffice since the reduction is defined for closed expressions, i.e., there are no free variables other than x and y.

[4]Actually, there is a third situation: the predicate evaluation does not terminate.

Example 2.1.10. The following two reduction sequences show how a cast for a first-order value is evaluated. Note that we use several expressions that are not given by the syntax, e.g., $x > 0$ etc., but everything can be represented by the formal ones.

$$
\begin{array}{ll}
(1 : \{x{:}\mathtt{nat} \mid x = 1\} \Rightarrow \{x{:}\mathtt{nat} \mid x > 0\}) & \\
\longrightarrow (1 : \mathtt{nat} \Rightarrow \{x{:}\mathtt{nat} \mid x > 0\}) & \text{by (R-CForget)} \\
\longrightarrow \langle\!\langle (1 : \mathtt{nat} \Rightarrow \mathtt{nat})\, ?\, \{x{:}\mathtt{nat} \mid x > 0\} \rangle\!\rangle & \text{by (R-CWait)} \\
\longrightarrow \langle\!\langle 1\, ?\, \{x{:}\mathtt{nat} \mid x > 0\} \rangle\!\rangle & \text{by (R-Ctx) and (R-CNat)} \\
\longrightarrow \langle\!\langle 1 > 0 \Longrightarrow 1 : \{x{:}\mathtt{nat} \mid x > 0\} \rangle\!\rangle & \text{by (R-CActive)} \\
\longrightarrow^{*} \langle\!\langle \mathtt{True} \Longrightarrow 1 : \{x{:}\mathtt{nat} \mid x > 0\} \rangle\!\rangle & \text{by (R-Ctx) and PCF evaluation rules} \\
\longrightarrow 1 & \text{by (R-CSucceed)}
\end{array}
$$

$$
\begin{array}{ll}
(1 : \{x{:}\mathtt{nat} \mid x = 1\} \Rightarrow \{x{:}\mathtt{nat} \mid x = 0\}) & \\
\longrightarrow^{*} \langle\!\langle \mathtt{False} \Longrightarrow 1 : \{x{:}\mathtt{nat} \mid x = 0\} \rangle\!\rangle & \text{Similar to the successful evaluation above} \\
\longrightarrow \mathtt{blame} & \text{by (R-CFail)}
\end{array}
$$

Example 2.1.11. The following shows what happens for a higher-order contract checking.

$$
\begin{array}{l}
(\lambda x{:}\mathtt{nat}.x : (x{:}\mathtt{nat}) \to \mathtt{nat} \Rightarrow (y{:}\mathtt{nat}) \to \{z{:}\mathtt{nat} \mid z > y\}) \\
\quad \longrightarrow \lambda y{:}\mathtt{nat}.(\lambda x{:}\mathtt{nat}.(\lambda x{:}\mathtt{nat}.x)\, x : \mathtt{nat} \Rightarrow \{z{:}\mathtt{nat} \mid z > y\})\, (y : \mathtt{nat} \Rightarrow \mathtt{nat}) \\
\hfill \text{by (R-CArrow)}
\end{array}
$$

The obtained expression is a value; and no more evaluation happens, that means no blame occurs even though the identity function ideally cannot satisfy the contracts represented by the target type. The violation is detected when an actual argument is given as follows.

$$
\begin{array}{l}
(\lambda y{:}\mathtt{nat}.(\lambda x{:}\mathtt{nat}.(\lambda x{:}\mathtt{nat}.x)\, x : \mathtt{nat} \Rightarrow \{z{:}\mathtt{nat} \mid z > y\})\, (y : \mathtt{nat} \Rightarrow \mathtt{nat}))\, 0 \\
\longrightarrow^{*} (0 : \mathtt{nat} \Rightarrow \{z{:}\mathtt{nat} \mid z > 0\}) \\
\longrightarrow^{*} \langle\!\langle 0 > 0 \Longrightarrow 0 : \{z{:}\mathtt{nat} \mid z > 0\} \rangle\!\rangle \\
\longrightarrow^{*} \mathtt{blame}
\end{array}
$$

2.1.3 Type System

The type system of $\mathrm{PCF_H}$ consists of *type compatibility* relation, denoted by $\sigma \simeq \tau$ and read "σ and τ are compatible"; *type equivalence* relation, denoted by $\sigma \equiv \tau$ and read "σ and τ are equivalent"; *well-formedness* relations for types, denoted by $\Gamma \Vdash \tau$ and read "τ is well-formed under Γ", (and environments, denoted by Γ ok and read "Γ is well-formed"; and *typing* relations for expressions, denoted by $\Gamma \vdash M : \tau$ and read "M has τ under Γ".

Type Compatibility

The type compatibility relation is defined by the rules in Figure 2.5. The relation defines types from and into which an expression can be cast, i.e., the source type and target type of a cast must be compatible. Intuitively, two types are compatible if and only if the types refine the same simple type. This expresses that a cast checks only contracts at run-time, and a cast which completely alternates the type of expression, e.g., $(M : \mathtt{bool} \Rightarrow \mathtt{nat})$, is

$$\frac{}{\texttt{bool} \simeq \texttt{bool}} \quad \text{(C-Bool)}$$

$$\frac{}{\texttt{nat} \simeq \texttt{nat}} \quad \text{(C-Nat)}$$

$$\frac{\sigma_1 \simeq \tau_1 \qquad \sigma_2 \simeq \tau_2}{(x{:}\sigma_1) \to \sigma_2 \simeq (x{:}\tau_1) \to \tau_2} \quad \text{(C-Arrow)}$$

$$\frac{\sigma \simeq \tau}{\{x{:}\sigma \mid M\} \simeq \tau} \quad \text{(C-RefineL)}$$

$$\frac{\sigma \simeq \tau}{\sigma \simeq \{x{:}\tau \mid M\}} \quad \text{(C-RefineR)}$$

Figure 2.5: Type compatibility of $\mathrm{PCF_H}$

$$\frac{}{\texttt{bool} \equiv \texttt{bool}} \quad \text{(E-Bool)}$$

$$\frac{}{\texttt{nat} \equiv \texttt{nat}} \quad \text{(E-Nat)}$$

$$\frac{\sigma_1 \equiv \sigma_2 \qquad \tau_1 \equiv \tau_2}{(x{:}\sigma_1) \to \sigma_2 \equiv (x{:}\tau_1) \to \tau_2} \quad \text{(E-Arrow)}$$

$$\frac{\sigma \equiv \tau \qquad N \longrightarrow N'}{\{x{:}\sigma \mid M[z := N]\} \equiv \{x{:}\tau \mid M[z := N']\}} \quad \text{(E-RefineLR)}$$

$$\frac{\sigma \equiv \tau \qquad N \longrightarrow N'}{\{x{:}\sigma \mid M[z := N']\} \equiv \{x{:}\tau \mid M[z := N]\}} \quad \text{(E-RefineRL)}$$

$$\frac{\tau_1 \equiv \tau_2 \qquad \tau_2 \equiv \tau_3}{\tau_1 \equiv \tau_3} \quad \text{(E-Trans)}$$

Figure 2.6: Type equivalence of $\mathrm{PCF_H}$

rejected at compile-time. Consequently, the number of expressions that can be typed in a manifest contract system is not beyond the one in a simple type system even if we try to abuse casts. In other words, a manifest contract system just extends an existing type system with a software contract system.

Type Equivalence

A dependent type system usually has a type equivalence relation to regard syntactically different types as the same type—called implicit *type conversion*, e.g., $\{x{:}\texttt{nat} \mid x = 1 + 1\}$ and $\{x{:}\texttt{nat} \mid x = 2\}$ are regarded as the same type when an expression is typed. The type equivalence relation of $\mathrm{PCF_H}$ is defined by the rules in Figure 2.6. Remembering the reduction relation is defined for closed expressions, the type equivalence relation of $\mathrm{PCF_H}$ is weaker than an usual one which is based on full-reduction of expressions, where expressions involving free variables are also evaluated. However, it suffices because type conversion is done explicitly by casts in $\mathrm{PCF_H}$, and the type equivalence relation is only used to show type soundness. So,

$$\frac{}{\emptyset\ \text{ok}} \qquad \text{(W-Empty)}$$

$$\frac{\Gamma\ \text{ok} \qquad \Gamma \Vdash \tau \qquad (x \notin \text{dom}(\Gamma))}{\Gamma, x{:}\tau\ \text{ok}} \qquad \text{(W-Push)}$$

$$\frac{\Gamma\ \text{ok}}{\Gamma \Vdash \texttt{bool}} \qquad \text{(W-Bool)}$$

$$\frac{\Gamma\ \text{ok}}{\Gamma \Vdash \texttt{nat}} \qquad \text{(W-Nat)}$$

$$\frac{\Gamma, x{:}\sigma \Vdash \tau}{\Gamma \Vdash (x{:}\sigma) \to \tau} \qquad \text{(W-Arrow)}$$

$$\frac{\Gamma, x{:}\tau \vdash M : \texttt{bool}}{\Gamma \Vdash \{x{:}\tau \mid M\}} \qquad \text{(W-Refine)}$$

Figure 2.7: Type system of PCF_H (1): well-formedness rules

it is not necessary that the equivalence relation relates many types for programmers; but it suffices that the equivalence relation only relates enough types to show type soundness—for instance, one important condition required is that $\tau[x := N] \equiv \tau[x := N']$ holds for $N \longrightarrow N'$.

Well-formed Types and Environments

The well-formedness relation is defined by the rules in Figure 2.7.[5] Type well-formedness checks if the predicate of a refinement type has the Boolean type. Environment well-formedness checks if all types in an environment are well-formed. Note that well-formedness of σ of $(x{:}\sigma) \to \tau$ and τ of $\{x{:}\tau \mid M\}$ does not checked explicitly at (W-Arrow) and (W-Refine), respectively; but the well-formedness is checked as a part of environment well-formedness in a leaf of a derivation.

Typing Relation

The typing relation is defined by the rules in Figure 2.8 and Figure 2.9. Each defines *compile-time typing*, which is used for type checking of source code, and *run-time typing*, which exists for showing meta properties, respectively. This distinction makes type checking of PCF_H decidable since all undecidable side-conditions (caused by the fixpoint operator, which makes an infinite-reduction possible) are dispelled into run-time typing. This distinction is started by Belo et al. [1] so that type checking of source code does not rely on operational semantics, which simplifies the discussion of meta properties; but, as a drawback, some source code, e.g., $(\lambda x{:}\{x{:}\texttt{nat} \mid 1+1=2\}.x)\,(0 : \texttt{nat} \Rightarrow \{x{:}\texttt{nat} \mid 2=2\})$, is rejected at compile-time; while hybrid type checking [19] could accept.

Compile-time typing rules are almost straightforward—`True` and `False` have `bool` under an arbitrary well-formed environment; $\texttt{if}\ L\ \texttt{then}\ M\ \texttt{else}\ N$ has τ when L has

[5] Actually, well-formedness relation is defined by using typing relation, and vice versa. So, Figure 2.7, Figure 2.8, and Figure 2.9 are *one* definition which is mutually defined.

$$\frac{\Gamma\ \text{ok} \qquad (x{:}\tau \in \Gamma)}{\Gamma \vdash x : \tau} \quad \text{(T-Var)}$$

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

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

$$\frac{\Gamma\ \text{ok}}{\Gamma \vdash \mathtt{0} : \mathtt{nat}} \quad \text{(T-Zero)}$$

$$\frac{\Gamma \vdash M : \mathtt{nat}}{\Gamma \vdash \mathtt{succ}(M) : \mathtt{nat}} \quad \text{(T-Succ)}$$

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

$$\frac{\Gamma \vdash M : \mathtt{nat}}{\Gamma \vdash \mathtt{iszero}(M) : \mathtt{bool}} \quad \text{(T-IsZero)}$$

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

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

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

$$\frac{\Gamma, f{:}(x{:}\sigma) \to \tau \vdash \lambda x{:}\sigma.M : (x{:}\sigma) \to \tau}{\Gamma \vdash \mu f{:}(x{:}\sigma) \to \tau.\lambda x{:}\sigma.M : (x{:}\sigma) \to \tau} \quad \text{(T-Fix)}$$

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

Figure 2.8: Type system of $\mathrm{PCF_H}$ (2): compile-time typing rules

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

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

(T-Active)

$$\frac{\Gamma\ \text{ok} \qquad \Vdash \tau}{\Gamma \vdash \mathtt{blame} : \tau} \quad \text{(T-Blame)}$$

$$\frac{\Gamma\ \text{ok} \qquad \vdash M : \sigma \qquad \Vdash \tau \qquad \sigma \equiv \tau}{\Gamma \vdash M : \tau} \quad \text{(T-Conv)}$$

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

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

Figure 2.9: Type system of PCF_H (3): run-time typing rules

$\mathtt{bool}$ and both M has τ; etc. The three rules (T-Pred), (T-App), and (T-Cast) are worth explanation. (T-Pred) claims that M of $\mathtt{pred}(M)$ must not become 0 by the refinement type of the premise, which represents the non-zero natural numbers. This is the main reason that a well-typed expression does not stuck without a reduction rule for $\mathtt{pred}(0)$ in PCF_H. (T-App) is a standard rule for function applications in a dependently typed system. The substitution in the conclusion expresses that the return type of a function depends on an actual argument. (T-Cast) just checks if the subject has the source type; the target type is well-formed;[6] and the source and the target types are compatible. Therefore, no contract checking happens at compile-time, i.e., $(0 : \mathtt{nat} \Rightarrow \{x{:}\mathtt{nat} \mid x \neq 0\})$ is accepted by a type checker (and a contract violation will be reported at run-time).

Run-time typing rules are rather specific to a manifest contract system. The first three rules are for run-time expressions. (T-Waiting) just checks if M has τ (and $\{x{:}\tau \mid N\}$ is well-formed) since a waiting check represents a state in which a validation of the predicate N against M of τ waits until M becomes a value. For (T-Active), the first four premises are straightforward, remembering M is a predicate being evaluated and V is the subject. An important premise is the last one, which guarantees that M is actually an intermediate state of the predicate validation. (T-Blame) is used to give an arbitrary type to $\mathtt{blame}$ because any well-typed expression might go to $\mathtt{blame}$ by a run-time checking failure. The last three rules exist for showing the subject reduction property—*a type of an expression is preserved by a reduction.* (T-Conv) is used when the reduction happens by (R-Ctx) with the context $V\,\square$; which will change the type from $\tau[x := N]$ into $\tau[x := N']$, where $N \longrightarrow N'$. We recover the original type by using the type equivalence relation—this is why $\tau[x := N] \equiv \tau[x := N']$ for $N \longrightarrow N'$ is an important condition for the type equivalence relation. (T-Forget) is used when the reduction happens by (R-CForget). (T-Exact) is used when the reduction happens by (R-CSucceed), where the expression before the reduction is typed by (T-Active)

[6] Well-formedness of the source type is obtained by the fact that the subject has the source type.

under the condition $M = \texttt{True}$.

Someone might wonder if the empty typing context suffices for the conclusion of run-time typing rules since those are only used for run-time expressions, i.e., closed expressions. However, the generalization is required for showing meta properties.

2.2 Properties

We introduce the desirable properties for manifest contract systems. Here, we just show statements of properties. Concrete properties and proofs will be seen in the following chapters.

First of all, as a typed system, the following well-known properties, a.k.a. type soundness, should hold.

Proposition 2.2.1 (Subject reduction)**.** *If* $\vdash M : \tau$ *and* $M \longrightarrow N$*, then* $\vdash N : \tau$*.*

Proposition 2.2.2 (Progress)**.** *If* $\vdash M : \tau$*, then* M *is a value;* $\texttt{blame}$*; or* $M \longrightarrow N$ *for some* N*.*

The characteristic property is the following, which shows a type system guarantees no necessary casts are omitted.

Proposition 2.2.3 (Value inversion)**.** *If* $\vdash V : \{x{:}\tau \mid M\}$*, then* $M[x := V] \longrightarrow^* \texttt{True}$*.*

Remark 2.2.4. One might think that the statement above is too strong since the predicate will not terminate in general. However, observing the compile-time typing rules, we can see that a refinement type cannot give directly to source code, e.g, $\vdash 0 : \{x{:}\texttt{nat} \mid x = 0\}$. We always enclose an expression by a cast if we want to give a refinement type. This is the way to establish the value inversion property. If a predicate diverges, computation diverges at the run-time checking by a mandatory cast; a program never reaches a value; and thus, it is out of scope of the value inversion property.

Summarizing the properties, type safety of a manifest contract system becomes as follows.

Proposition 2.2.5 (Type safety)**.** *If* $\vdash M : \tau$*,*

- *M is evaluated into a value V, namely $M \longrightarrow^* V$;*
- *run-time checking fails, namely $M \longrightarrow^* \texttt{blame}$; or*
- *an evaluation diverges.*

Especially, in the first case and τ is a refinement type $\{x{:}\tau' \mid N\}$, V satisfies N, namely $N[x := V] \longrightarrow^* \texttt{True}$.

2.3 Summary

We have introduced a simple manifest contract system PCF_H, which is just a manifest contract system extension of PCF. As we have seen, in a manifest contract system, contracts are written in types and checked at run-time by casts. The type system of a manifest contract system never examines contracts at compile-time but guarantees no necessary casts lack by checking syntactic consistency. The guarantee is formally stated as a value inversion property, a characteristic property of manifest contract systems.

3 Manifest Contracts with Intersection Types

In this chapter, we develop a formal calculus PCFvΔ_H, a manifest contract system with intersection types. The goal of this chapter 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. Firstly, consider the following definition with a parity contract in a surface language.

$$\mathtt{let}\, \mathit{succ}' \mathord{:} \mathrm{odd} \rightarrow \mathrm{even} = \lambda x.\mathtt{succ}(x).$$

As we have already known, casts are required in a manifest contract system as follows.

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

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

$$\mathtt{let}\, \mathit{succ}' \mathord{:} (\mathrm{odd} \rightarrow \mathrm{even}) \wedge (\mathrm{even} \rightarrow \mathrm{odd}) = \lambda x.\mathtt{succ}(x).$$

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, $\lambda x \mathord{:} \mathrm{odd}.(\mathtt{succ}((x : \mathrm{odd} \Rightarrow \mathtt{nat})) : \mathtt{nat} \Rightarrow \mathrm{even})$ is obtained by cast insertion if the function is typed as $\mathrm{odd} \rightarrow \mathrm{even}$; while $\lambda x \mathord{:} \mathrm{even}.(\mathtt{succ}((x : \mathrm{even} \Rightarrow \mathtt{nat})) : \mathtt{nat} \Rightarrow \mathrm{odd})$ is obtained when the body is typed as $\mathrm{even} \rightarrow \mathrm{odd}$. However, the function must have both types to have the intersection type. It may seem sufficient to just cast the body itself, that is, $((\lambda x \mathord{:} \mathtt{nat}.\mathtt{succ}(x)) : \mathtt{nat} \rightarrow \mathtt{nat} \Rightarrow (\mathrm{odd} \rightarrow \mathrm{even}) \wedge (\mathrm{even} \rightarrow \mathrm{odd}))$. 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.

We build PCFvΔ_H on top of the Δ-calculus, a Church-style intersection type system by Liquori and Stolze [34]. 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.

Contributions.

- we design a manifest contracts calculus with *refinement intersection types* [57, 66], a restricted form of intersection types.
- we formalize the calculus PCFvΔ_{H}; and
- we state and prove type soundness, value inversion, and dynamic soundness.

Disclaimer.

To concentrate on the PCFvΔ_{H}-specific problems, we put the following restrictions for PCFvΔ_{H} 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 dealt by the next chapter.
- 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, $(\mathtt{even} \to \mathtt{even}) \wedge (\mathtt{odd} \to \mathtt{odd})$ is a refinement intersection types since types of both sides refine the same type $\mathtt{nat} \to \mathtt{nat}$, while $(\mathtt{nat} \to \mathtt{nat}) \wedge (\mathtt{float} \to \mathtt{float})$ is not.

3.1 Overview of Language: PCFvΔ_{H}

PCFvΔ_{H} is obtained by extending PCF$_{\mathrm{H}}$ with intersection types (derived from the Δ-calculus [34]). As it is soon revealed that the dynamic checking used in PCFvΔ_{H} is more complicated than the one in PCF$_{\mathrm{H}}$, we carefully construct the system and show that any *valid* PCF program is also a valid PCFvΔ_{H} program; and a PCFvΔ_{H} program behaves as same as a call-by-value PCF. In other words, PCFvΔ_{H} is a conservative extension of call-by-value PCF.

3.1.1 The Δ-calculus

To address the challenge discussed in the beginning of this chapter, PCFvΔ_{H} is strongly influenced by the Δ*-calculus* by Liquori and Stolze [34], 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 $(\mathtt{even} \to \mathtt{even}) \wedge (\mathtt{odd} \to \mathtt{odd})$ as follows.

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

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 the beginning of this chapter into the following PCFvΔ_H expression.

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

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

3.1.2 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 [26], who studied intersection (and union) contract checking in the "latent" style [21] 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.

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

where

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

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.

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

Figure 3.1: Syntax of PCFv

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

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

similarly to higher-order casts [17]. Then, the delayed cast fires when an actual argument is given:

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

3.2 Language: PCFvΔ_{H}

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.

3.2.1 PCFv

We defines PCFv as Figure 3.1 and Figure 3.2 for syntax and semantics, respectively; which corresponds to the PCF part of PCF$_{\mathrm{H}}$.

Definition 3.2.1 (Meta operations for PCFv terms)**.** We define free variables of an expression; substition; and context application in a similar manner to ones for PCF$_{\mathrm{H}}$.

3.2.2 PCFvΔ_{H}

The syntax of PCFvΔ_{H} is shown in Figure 3.3. We introduce some more metavariables: I ranges over *interface types*, a subset of types; B ranges over *recursion bodies*, a subset of expressions; and C ranges over *commands*.

Types are extended with intersection types; the restriction that a well-formed intersection type is a refinement intersection type is enforced by the type system. 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.

$$\frac{}{\texttt{pred}(0) \longrightarrow_{\mathrm{PCF}} 0} \quad \text{(PCF-PredZ)}$$

$$\frac{}{\texttt{pred}(\texttt{succ}(\overline{n})) \longrightarrow_{\mathrm{PCF}} \overline{n}} \quad \text{(PCF-Pred)}$$

$$\frac{}{\texttt{iszero}(0) \longrightarrow_{\mathrm{PCF}} \texttt{True}} \quad \text{(PCF-IsZeroT)}$$

$$\frac{}{\texttt{iszero}(\texttt{succ}(\overline{n})) \longrightarrow_{\mathrm{PCF}} \texttt{False}} \quad \text{(PCF-IsZeroF)}$$

$$\frac{}{\texttt{if}\,\texttt{True}\,\texttt{then}\,M\,\texttt{else}\,N \longrightarrow_{\mathrm{PCF}} M} \quad \text{(PCF-IfT)}$$

$$\frac{}{\texttt{if}\,\texttt{False}\,\texttt{then}\,M\,\texttt{else}\,N \longrightarrow_{\mathrm{PCF}} N} \quad \text{(PCF-IfF)}$$

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

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

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

Figure 3.2: Operational semantics of PCFv

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

Figure 3.3: Syntax of PCFvΔ_{H}

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

Figure 3.4: Essence of a $\mathrm{PCFv}\Delta_{\mathrm{H}}$ term

Expressions are extended with strong pairs (namely, pair construction $\langle M,N\rangle$, left projection $\pi_1(M)$, and right projection $\pi_2(M)$); and delayed checks $\langle\!\langle V : \sigma\Rightarrow\tau_1\to\tau_2\rangle\!\rangle$ are added to run-time expressions as we have discussed in Section 3.1. Recursion bodies are (possibly nested strong pairs) of λ-abstractions.

We do not include `blame` in expressions, although $\mathrm{PCF}_{\mathrm{H}}$ include it among expressions. As a consequence, the evaluation relation for $\mathrm{PCFv}\Delta_{\mathrm{H}}$ is defined between commands. This distinction will turn out to be convenient in stating correspondence between the semantics of $\mathrm{PCFv}\Delta_{\mathrm{H}}$ and that of PCFv, which does not have `blame`.

Someone will notice evaluation contexts for strong pairs and active checks are lacked. The lack is intentional, i.e., sub-expression reduction rules for those are defined as separated rules without using evaluation contexts because those require special treatments.

Definition 3.2.2 (Meta operations for $\mathrm{PCFv}\Delta_{\mathrm{H}}$ terms)**.** We define free variables of an expression; substition; and context application in a similar manner to ones for $\mathrm{PCF}_{\mathrm{H}}$.

The essence of a $\mathrm{PCFv}\Delta_{\mathrm{H}}$ term is defined in Figure 3.4, which is mostly straightforward, thanks to the blame distinction. 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.

3.2.3 Operational Semantics of $\mathrm{PCFv}\Delta_{\mathrm{H}}$

The operational semantics of $\mathrm{PCFv}\Delta_{\mathrm{H}}$ consists of four relations $M \rightharpoonup_{\mathsf{p}} N$, $M \rightharpoonup_{\mathsf{c}} C$, $M \longrightarrow_{\mathsf{p}} N$, and $M \longrightarrow_{\mathsf{c}} C$. 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 p correspond to PCFv evaluation, that is, *essential evaluation*; and ones subscripted by c correspond to dynamic contract checking. Dynamic checking is nondeterministic because of (RC-WedgeL), (RC-WedgeR), (EC-PairL), and (EC-PairR).

$$\frac{}{\texttt{pred}(\texttt{succ}(\overline{n})) \rightharpoonup_{\mathsf{p}} \overline{n}} \quad \text{(RP-PRED)}$$

$$\frac{}{\texttt{iszero}(0) \rightharpoonup_{\mathsf{p}} \texttt{True}} \quad \text{(RP-ISZERO-T)}$$

$$\frac{}{\texttt{iszero}(\texttt{succ}(\overline{n})) \rightharpoonup_{\mathsf{p}} \texttt{False}} \quad \text{(RP-ISZERO-F)}$$

$$\frac{}{\texttt{if}\,\texttt{True}\,\texttt{then}\,M\,\texttt{else}\,N \rightharpoonup_{\mathsf{p}} M} \quad \text{(RP-IF-T)}$$

$$\frac{}{\texttt{if}\,\texttt{False}\,\texttt{then}\,M\,\texttt{else}\,N \rightharpoonup_{\mathsf{p}} N} \quad \text{(RP-IF-F)}$$

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

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

$$\frac{M \rightharpoonup_{\mathsf{p}} N}{M \longrightarrow_{\mathsf{p}} N} \quad \text{(RP-RED)}$$

$$\frac{M \longrightarrow_{\mathsf{p}} N}{\mathcal{E}[M] \longrightarrow_{\mathsf{p}} \mathcal{E}[N]} \quad \text{(RP-CTX)}$$

$$\frac{M \longrightarrow_{\mathsf{p}} M' \qquad N \longrightarrow_{\mathsf{p}} N'}{\langle M, N \rangle \longrightarrow_{\mathsf{p}} \langle M', N' \rangle} \quad \text{(RP-PAIRS)}$$

Figure 3.5: Operational semantics of PCFvΔ_{H} (1): essential evaluation

Essential Evaluation $\longrightarrow_{\mathsf{p}}$

The essential evaluation, defined in Figure 3.5, defines the evaluation of the essential, namely PCF, part of a program; and thus, it is similar to $\longrightarrow_{\mathrm{PCF}}$. There are just three differences, that is: there are two relations; there is no reduction rule for $\mathtt{pred}(0)$ as $\mathrm{PCF_H}$; 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.

Dynamic Checking $\longrightarrow_{\mathsf{c}}$

Dynamic checking is more complicated. Firstly, we focus on reduction rules in Figure 3.6.

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 similar to $\mathrm{PCF_H}$, but how a cast between function types is dealyed is different—in $\mathrm{PCF_H}$, it is done by using lambda abstraction. $\mathrm{PCFv}\Delta_{\mathrm{H}}$ uses a special syntax to denote the delay and deals with (RC-Delay) and (RC-Arrow). Actually, this way is not new—It is used in the original work [17] 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 3.1, 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) and (RC-WedgeR) 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 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 3.7, 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 $\mathtt{blame}$ (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 $\mathtt{blame}$ immediately—in one step—when a sub-expression evaluates to $\mathtt{blame}$. Here is an example of execution of failing dynamic checking, where in the last step immediately produces $\mathtt{blame}$; while in $\mathrm{PCF_H}$, evaluation passes through an intermediate state $\mathtt{blame} + 1$.

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

Definition 3.2.3 (Evaluation)**.** The one-step evaluation relation of $\mathrm{PCFv}\Delta_{\mathrm{H}}$, denoted by $\longrightarrow$, is defined as $\longrightarrow_{\mathsf{p}} \cup \longrightarrow_{\mathsf{c}}$. The multi-step evaluation relation of $\mathrm{PCFv}\Delta_{\mathrm{H}}$, denoted by $\longrightarrow^{*}$, is the reflexive and transitive closure of $\longrightarrow$.

$$\frac{}{\pi_1(\langle V_1, V_2\rangle) \rightharpoonup_{\mathsf{c}} V_1} \quad \text{(RC-Fst)}$$

$$\frac{}{\pi_2(\langle V_1, V_2\rangle) \rightharpoonup_{\mathsf{c}} V_2} \quad \text{(RC-Snd)}$$

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

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

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

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

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

$$\frac{}{\langle\!\langle V_1 : \sigma_1 \wedge \sigma_2 \Rightarrow \tau_1 \rightarrow \tau_2 \rangle\!\rangle\, V_2 \rightharpoonup_{\mathsf{c}} (\pi_1(V_1) : \sigma_1 \Rightarrow \tau_1 \rightarrow \tau_2)\, V_2} \quad \text{(RC-WedgeL)}$$

$$\frac{}{\langle\!\langle V_1 : \sigma_1 \wedge \sigma_2 \Rightarrow \tau_1 \rightarrow \tau_2 \rangle\!\rangle\, V_2 \rightharpoonup_{\mathsf{c}} (\pi_2(V_1) : \sigma_2 \Rightarrow \tau_1 \rightarrow \tau_2)\, V_2} \quad \text{(RC-WedgeR)}$$

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

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

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

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

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

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

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

Figure 3.6: Operational semantics of PCFvΔ_{H} (2): reduction rules for dynamic checking

$$\frac{M \rightharpoonup_{\mathsf{c}} C}{M \longrightarrow_{\mathsf{c}} C} \quad \text{(EC-Red)}$$

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

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

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

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

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

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

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

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

$$\frac{M_2 \longrightarrow_{\mathsf{c}} \mathtt{blame}}{\langle M_1, M_2\rangle \longrightarrow_{\mathsf{c}} \mathtt{blame}} \quad \text{(EB-PairR)}$$

Figure 3.7: Operational semantics of PCFvΔ_{H} (3): contextual rules for dynamic checking

$$\frac{}{\emptyset \text{ ok}} \quad \text{(W-Empty)}$$

$$\frac{\Gamma \text{ ok} \qquad \Vdash \tau \qquad (x \# \Gamma)}{\Gamma, x{:}\tau \text{ ok}} \quad \text{(W-Push)}$$

$$\frac{}{\Vdash \texttt{nat}} \quad \text{(W-Nat)}$$

$$\frac{}{\Vdash \texttt{bool}} \quad \text{(W-Bool)}$$

$$\frac{\Vdash \sigma \qquad \Vdash \tau}{\Vdash \sigma \to \tau} \quad \text{(W-Arrow)}$$

$$\frac{x{:}\tau \vdash M : \texttt{bool}}{\Vdash \{x{:}\tau \mid M\}} \quad \text{(W-Refine)}$$

$$\frac{\Vdash \sigma \qquad \Vdash \tau \qquad (\wr\sigma\wr = \wr\tau\wr)}{\Vdash \sigma \wedge \tau} \quad \text{(W-Wedge)}$$

Figure 3.8: Type system of PCFvΔ_{H} (1): well-formedness rules

3.2.4 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 3.8, 3.9 and 3.10.

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; and thus the type well-formedness relation does not involve type environments.

Extension parts comes from Liquori and Stolze [34]. 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 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.

The run-time rules have 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).

3.3 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

$$\frac{\Gamma \text{ ok}}{\Gamma \vdash \mathtt{0} : \mathtt{nat}} \quad \text{(T-Zero)}$$

$$\frac{\Gamma \vdash M : \mathtt{nat}}{\Gamma \vdash \mathtt{succ}(M) : \mathtt{nat}} \quad \text{(T-Succ)}$$

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

$$\frac{\Gamma \vdash M : \mathtt{nat}}{\Gamma \vdash \mathtt{iszero}(M) : \mathtt{bool}} \quad \text{(T-IsZero)}$$

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

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

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

$$\frac{\Gamma \text{ ok} \qquad (x{:}\tau \in \Gamma)}{\Gamma \vdash x : \tau} \quad \text{(T-Var)}$$

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

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

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

$$\frac{\Gamma \vdash M : \sigma \wedge \tau}{\Gamma \vdash \pi_1(M) : \sigma} \quad \text{(T-Fst)}$$

$$\frac{\Gamma \vdash M : \sigma \wedge \tau}{\Gamma \vdash \pi_2(M) : \tau} \quad \text{(T-Snd)}$$

$$\frac{\Gamma, f{:}I \vdash B : I}{\Gamma \vdash \mu f{:}I.B : I} \quad \text{(T-Fix)}$$

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

Figure 3.9: Type system of PCFvΔ_{H} (2): compile-time typing rules

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

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

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

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

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

Figure 3.10: Type system of PCFvΔ_H (3): run-time typing rules

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

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

Lemma 3.3.2. *If $\vdash M : \tau$ and $M \longrightarrow_{\mathsf{p}} N$, then $\wr M \wr \longrightarrow_{PCF} \wr N \wr$.*

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

The following corollary is required to prove the preservation property.

Corollary 3.3.3. *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 3.3.4. *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 3.3.2 and Lemma 3.3.4. 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 3.3.5. *If $\vdash M : \tau$ and $M \longrightarrow N$, then $\wr M \wr \longrightarrow^*_{PCF} \wr N \wr$.*

3.3.1 Type Soundness

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

Lemma 3.3.6. *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 3.3.7 (Subject reduction). *If* $\vdash M : \tau$ *and* $M \longrightarrow N$*, then* $\vdash N : \tau$*.*

Proof. We prove subject reduction 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 3.3.6 as usual. For the context evaluation for strong pairs, we use Corollary 3.3.3 and Lemma 3.3.4 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 [54, 52, 45]. The property itself is proven by using the following two, which are for strengthening an induction hypothesis.

Definition 3.3.8. We define a relation between values and types, written $V \vDash \tau$, by the following rules.

$$\frac{V \vDash \tau \qquad M[x \mapsto V] \longrightarrow^* \mathtt{True}}{V \vDash \{x{:}\tau \mid M\}} \qquad \frac{(\tau \neq \{x{:}\sigma \mid M\})}{V \vDash \tau}$$

Lemma 3.3.9. *If* $\vdash V : \tau$*, then* $V \vDash \tau$*.*

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

Theorem 3.3.10 (Value inversion). *If* $\vdash V : \{x{:}\tau \mid M\}$*, then* $M[x \mapsto V] \longrightarrow^* \mathtt{True}$*.*

Proof. Immediate from Lemma 3.3.9. □

Remark 3.3.11. 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 $M[x \mapsto V_1] \longrightarrow^* \mathtt{True}$ and $N[x \mapsto V_2] \longrightarrow^* \mathtt{True}$. 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, $\vdash V : \{x{:}\mathtt{nat} \mid M_1\} \wedge \cdots \wedge \{x{:}\mathtt{nat} \mid M_n\}$, then $M_k[x \mapsto \wr V \wr] \longrightarrow^* \mathtt{True}$ 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 $\mathtt{pred}(M)$ is partial, this theorem can be proved only after Theorem 3.3.10.

Theorem 3.3.12 (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 3.3.2 and proof by contradiction because the essence of a PCFvΔ_{H} value is a PCFv value and it is normal form. □

3.4 Summary

We design and formalize 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.

4 Nondeterministic Manifest Contracts

A well-known approach to bringing nondeterminism to a functional language is the introduction of a *choice* operator, which we write $(M \parallel N)$ in this paper. It takes two operands M, N and returns one of them nondeterministically. Using the choice, `choose` function we have seen in the example above can be defined as follows.

```
1 let rec choose l =
2   match l with
3   | [x] -> x
4   | x::xs -> (x || choose xs)
```

To deal with the case where `choose` is called with an empty list, one may want to insert `require (not (isempty l))` before **match**.

We believe that it is meaningful to introduce manifest contracts into a nondeterministic language. Let us revisit the example of `prime_sum_pair`. By using casts, `prime_sum_pair` could be written as follows:

```
1 let prime_sum_pair =
2   let aux (l: int list) =
3     let a1 = choose l and a2 = choose l in
4     (a1, a2)
5   in
6   (aux : int list → int × int
7        ⇒ int list → {x:int × int | isprime(fst x + snd x)})
```

The type of `prime_sum_pair`, which is the target type of the cast on `aux`, is as expected.

In this chapter, we develop a formal calculus $\lambda^{H\parallel^{\Phi}}$ of manifest contracts with a nondeterministic choice operator. An interesting (and perhaps controversial) point in the design space is that whether we should allow nondeterminism in predicates in refinement types. We would like to allow it so that any library function, which may be nondeterministic, can be used to describe predicates. Moreover, it is necessary if we use the result of $\lambda^{H\parallel^{\Phi}}$ to introduce dependent function types into PCFvΔ_{H}. Among variants of the semantics for choice [55], we will investigate the so-called singular semantics,[1] which corresponds to *call-time choice* [23], because we think it is easy to understand. This semantics also poses interesting technical challenges, as we will discuss shortly. The goal of the present paper is to prove basic meta-theoretic properties of progress, type preservation (subject reduction) [65], and *contract satisfaction*, which is a generalization of value inversion and means that, if M is given type $\{x{:}\tau \mid N\}$, then the result of *any* successful execution of M satisfies predicate N *nondeterministically*.

[1]Roughly speaking, in the singular semantics, every occurrence of a variable of the same name in the same scope yields the same value, although the value itself may be chosen nondeterministically. For example, the value of $(\lambda x.x + x)\,(1 \parallel 2)$ is either 2 or 4.

As far as a simple program like `prime_sum_pair` is concerned, nondeterministic choice seems to integrate smoothly into a manifest contract system. In fact, it would integrate smoothly, if it were not for dependent function types.

However, the combination of dependent function types and nondeterministic choice—in particular, call-time choice—poses a few technical challenges. Before discussing them, let us recall the typing rule for function applications, which is standard in a language with dependent function types:

$$\frac{\Gamma \vdash M : (x{:}\tau_1) \to \tau_2 \qquad \Gamma \vdash N : \tau_1}{\Gamma \vdash M N : \tau_2[x := N]}$$

The first challenge is how to define type equivalence, which is used to show type preservation. Dependent type systems have a type equivalence relation $\equiv$ and a typing rule that allows the type of a term to be converted to an equivalent one, so that the two types $\tau_2[x := N]$ and $\tau_2[x := N']$ where $N \longrightarrow N'$ are related and thereby a reduction step $V N \longrightarrow V N'$, where V is a value of type $(x{:}\tau_1) \to \tau_2$, preserves the type. In (subsumption-free) manifest contract calculi, their type equivalence relations are derived from term equivalence, which includes the reduction relation. However, standard reduction rules for choice

$$(M \parallel N) \longrightarrow M \qquad (M \parallel N) \longrightarrow N$$

would make type equivalence inconsistent because term equivalence closed under these rules would relate any two expressions M and N through $M \longleftarrow (M \parallel N) \longrightarrow N$. Such inconsistency would break value inversion because any refinement type $\{x{:}\tau \mid M\}$ can be converted to $\{x{:}\tau \mid \texttt{False}\}$.

Another challenge is how to settle down interaction between the singular semantics and substitution $\tau_2[x := N]$ that appears in the typing rule for applications above. It duplicates possibly nondeterministic computation N. However, such duplication is contradictory to the expectation in the singular semantics that x has a single value. One solution to the problem would be to restrict predicates of refinement types to *pure* ones, that is, predicates has no nondeterminism. Although the restriction seems reasonable and could be easily accomplished [59], we avoid such a restriction to give programmers full expressiveness in writing software contracts.

To address the two challenges above, we give a new kind of choice, called *coordinated choice*. For the former, the choice does not discard an alternative, but reduction retains the set of all possible executions as Kutzner and Schmidt-Schauß work does [32]. For the latter, different occurrences of choices can share nondeterministic decisions via names; thus the semantics stays singular even though syntactic duplication of a choice takes place.

Contributions.

- we propose a new kind of nondeterministic choice, called coordinated choice;
- we formalize an operational semantics of coordinated choice;
- we formalize $\lambda^{H\parallel^{\Phi}}$, a nondeterministic manifest contract system by using coordinated choice; and
- we state progress, type preservation, and contract satisfaction and sketch their proofs.

Outline. Firstly, in Section 4.1, we introduce coordinated choice. Then, in Section 4.2, we integrate the new choice into a manifest contract system. In Section 4.3, we discuss the basic properties and sketch their proofs.

4.1 Coordinated Choice

As we have discussed, the combination of dependent function types and nondeterministic choice, in particular, naive call-time choice, poses two challenges on manifest contracts. In this section, we informally describe our solution.

The first problem of inconsistency of type equivalence is avoided by not using the naive rules to choose one branch and, instead, introducing rules that distribute an evaluation context over choices [32], such as

$$V(M_1 \parallel M_2) \longrightarrow (VM_1 \parallel VM_2)$$

(where V stands for a value). For example, $(\lambda x.x)\,(1 \parallel 2)$ reduces to $(1 \parallel 2)$ via $((\lambda x.x)\,1 \parallel (\lambda x.x)\,2)$. Similarly, $(1 \parallel 2) + (3 \parallel 4)$ reduces to $((4 \parallel 5) \parallel (5 \parallel 6))$. So, our reduction relation expresses all possible nondeterministic executions at once.

The second problem of the interaction between the singular semantics and substitution is addressed by coordination of choices. Actually, our choice is given a name Φ and written $(M \parallel^{\Phi} N)$; the name can be shared by another choice. The choices of the same name "synchronize," that is, globally make the same decision: For example, expression $(1 \parallel^{\mathtt{foo}} 2) + (3 \parallel^{\mathtt{bar}} 4)$ (here `foo` and `bar` are names) nondeterministically evaluates to 4, 5, or 6, whereas $(1 \parallel^{\mathtt{foo}} 2) + (3 \parallel^{\mathtt{foo}} 4)$ cannot evaluate to 5 because if 1 is chosen by the first choice, 3 must be chosen by the other.

To accomplish the coordination, we will introduce rules to discard impossible results. For example, $(1 \parallel^{\mathtt{foo}} 2) + (3 \parallel^{\mathtt{foo}} 4)$ reduces to $((4 \parallel^{\mathtt{foo}} 5) \parallel^{\mathtt{foo}} (5 \parallel^{\mathtt{foo}} 6))$ (via distribution of evaluation contexts), but the two occurrences of 5 are bogus. Actually, we can see that they are bogus only from the syntactic structure of an expression: a subexpression is bogus if it is reached by choosing different sides at choices of the same name. In the example above, the first occurrence of 5 is bogus because it is reached by first taking the left side of $\parallel^{\mathtt{foo}}$ and then the right of $\parallel^{\mathtt{foo}}$. Although the general rule to discard one branch of a choice is not allowed, our reduction relation does allow a bogus expression to be discarded; so, $((4 \parallel^{\mathtt{foo}} 5) \parallel^{\mathtt{foo}} (5 \parallel^{\mathtt{foo}} 6))$ reduces to $(4 \parallel^{\mathtt{foo}} 6)$ in two steps, whereas $(1 \parallel^{\mathtt{foo}} 2) + (3 \parallel^{\mathtt{bar}} 4)$ reduces to $((4 \parallel^{\mathtt{foo}} 5) \parallel^{\mathtt{bar}} (5 \parallel^{\mathtt{foo}} 6))$, which is normal.

Somewhat surprisingly, these two ideas are sufficient for basic correctness properties to hold of a manifest contract system with nondeterministic choice.

To formalize the reduction relation as described above, we introduce the notion of *orthant*, inspired by geometry.

4.1.1 Orthant

As we have described, an expression evaluates to a set of values (combined by $\parallel^{\Phi}$). We view each element in the set inhabit an "orthant," generated by splitting the whole space by axes corresponding to names on choices.

Figure 4.1 shows an informal interpretation of expressions. The first example shows a basic interpretation: the expression $(1 \parallel^{\Phi} 2) + (3 \parallel^{\Psi} 4)$ (where $\Phi \neq \Psi$) represents four expressions

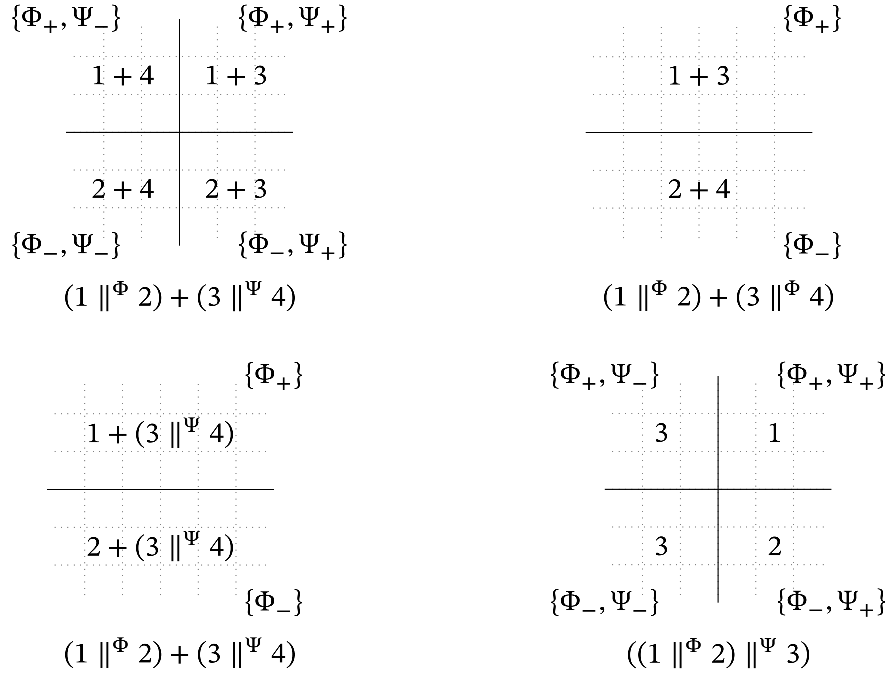

Figure 4.1: Examples of an informal interpretation of expressions.

since the names of the choices are different. Furthermore, each expression inhabits a different orthant, which is denoted by a set of signed names (Φ_+, meaning the left side of $\|^{\Phi}$, or Φ_-, meaning the right): For instance, the expression $1 + 3$ inhabits the orthant $\{\Phi_+, \Psi_+\}$ since 1 is on the left of the choice named Φ and 3 is on the left of the choice named Ψ.

In contrast, the second example shows a case in which choices have the same name. It shows that the expression $(1 \|^{\Phi} 2) + (3 \|^{\Phi} 4)$ represents only two expressions since the name is shared.

The third example shows another interpretation—which considers only axis Φ—of the expression in the first example. We do not have to decompose all the choices that occur in an expression, leaving some choices. We can even choose a name that does not occur in an expression. For example, 3, which a simple expression without coordinated choice, splits into 3 that inhabits $\{\Phi_+\}$ and 3 that inhabits $\{\Phi_-\}$. Such freedom in the choice of axes is useful to give an interpretation to the fourth example, where a choice is nested only in one branch of a choice.

These interpretations based on *orthant-local* viewpoints play an important role when we consider formal semantics.

Remark 4.1.1. Some orthants such as $\{\Phi_+, \Phi_-\}$ do not really make sense from a geometric point of view. Nevertheless, we allow such orthants for technical reasons (Lemma 4.3.1). For example, 2 in $((1 \|^{\Phi} 2) \|^{\Phi} 3)$ is considered to inhabit the nonsensical orthant $\{\Phi_+, \Phi_-\}$, which indicates 2 is bogus.

4.1.2 Orthant-local Reduction

More formally, we augment the reduction relation with an orthant Δ and write $M \xrightarrow{\Delta} N$, which intuitively means "M reduces to N in orthant Δ." The crux here is that bogus expressions

in orthant Δ can be discarded because it cannot be seen from the Δ-local viewpoint: For example, $(1 \parallel^{\mathtt{foo}} 2) \xrightarrow{\{\mathtt{foo}_+\}} 1$ holds since 2 inhabits $\{\mathtt{foo}_-\}$ and is considered bogus in this orthant. Similarly, $(1 \parallel^{\mathtt{foo}} 2) \xrightarrow{\{\mathtt{foo}_-\}} 2$ holds. Thanks to orthant, we can avoid inconsistency of equivalence discussed at the beginning of this section by making the equivalence a family of equivalences indexed by an orthant. [2] In general, reduction rules that allow discarding bogus expressions can be given as

$$(M \parallel^{\Phi} N) \xrightarrow{\Delta \cup \{\Phi_+\}} M \quad \text{and} \quad (M \parallel^{\Phi} N) \xrightarrow{\Delta \cup \{\Phi_-\}} N.$$

Along the same lines, when a subexpression under choice is reduced, the subexpression is reduced in the corresponding orthant. For example, $((1 \parallel^{\mathtt{foo}} 2) \parallel^{\mathtt{foo}} 3) \xrightarrow{\{\}} (1 \parallel^{\mathtt{foo}} 3)$ since $(1 \parallel^{\mathtt{foo}} 2)$ inhabits the orthant $\{\mathtt{foo}_+\}$ and $(1 \parallel^{\mathtt{foo}} 2) \xrightarrow{\{\mathtt{foo}_+\}} 1$. We formalize such reduction by the following two rules:

$$\frac{M \xrightarrow{\Delta \cup \{\Phi_+\}} M'}{(M \parallel^{\Phi} N) \xrightarrow{\Delta} (M' \parallel^{\Phi} N)} \qquad \frac{N \xrightarrow{\Delta \cup \{\Phi_-\}} N'}{(M \parallel^{\Phi} N) \xrightarrow{\Delta} (M \parallel^{\Phi} N')}$$

Multi-step reduction is written $M \xrightarrow{\Delta}{}^{*} N$. Note that the orthant in each reduction step is fixed to Δ. So, $M \xrightarrow{\Delta}{}^{*} N$ means $M \xrightarrow{\Delta} \cdots \xrightarrow{\Delta} N$.

Example 4.1.2. Now we can see how $(1 \parallel^{\Phi} 2) + (3 \parallel^{\Phi} 4)$ is evaluated into $(4 \parallel^{\Phi} 6)$ in the empty orthant $\{\,\}$.

$$\begin{aligned} (1 \parallel^{\Phi} 2) + (3 \parallel^{\Phi} 4) &\xrightarrow{\{\}}{}^{*} (1 + (3 \parallel^{\Phi} 4) \parallel^{\Phi} 2 + (3 \parallel^{\Phi} 4)) \\ &\xrightarrow{\{\}}{}^{*} (1 + 3 \parallel^{\Phi} 2 + 4) \\ &\xrightarrow{\{\}}{}^{*} (4 \parallel^{\Phi} 6) \end{aligned}$$

4.1.3 Programming with Names

It is cumbersome to assign appropriate names to choices in a program. Although we believe that in many cases it suffices to assign a fresh name to each choice, choices still coordinate if they are derived from the same program location. To avoid unintentional sharing, we introduce some mechanisms for programming with names below. We give the compilation method in a simpler setting in Chapter 5, which will automate the following discussions. Anyway, the following discussion will help to understand Chapter 5.

First, we consider the following program.

```
1  let choose x y = (x ||foo y) in
2    choose 1 2 + choose 3 4
```

The intention of the function `choose` is that it returns one of the given arguments nondeterministically. Thus we expect that the program results in three values: 4, 5, and 6. However, this program does not work as we expect. The result of the program is actually $(4 \parallel^{\mathtt{foo}} 6)$ because the two calls of `choose` will result in two choices of the same name and it is not possible for `choose` to return 2 and 3.

We introduce *name abstraction* and *name application* to solve the problem:

[2] Actually, 1 is considered equivalent to 2 (through $(1 \parallel^{\mathtt{foo}} 2)$) in orthant $\{\mathtt{foo}_+, \mathtt{foo}_-\}$ but this orthant is nonsensical—see Remark 4.1.1 above—meaning *both* branches are actually bogus.

```
1 let choose α x y = (x ||^α y) in
2   choose foo 1 2 + choose bar 3 4
```

Here, choose takes an additional name parameter α, which is used at the choice, and choose is called twice with different names foo and bar. As a result, each call independently returns one of the given arguments, and we can get the expected result.

Unfortunately, when a function is recursive, simple name abstraction and application do not suffice. Consider the following program:

```
1 let rec upto α x =
2   if x > 0 then (x ||^α upto Φ (x − 1)) else 0
3 in upto bar 3
```

The intention of the function upto is that it returns a random number between 0 and the given argument. Similarly to function choose above, the name of the choice is abstracted as a parameter α, which is used in the coordinated choice. Now, the question is what Φ, the name argument to the recursive call to upto, should be. Neither foo nor α is satisfactory because in either case choices of the same name will emerge after a few recursive calls. What we would like here is an ability to create a name that is different from foo and α.

We introduce name concatenation $\Phi\Psi$, which yields a name different from Φ and Ψ. Here is the final version of upto:

```
1 let rec upto α x =
2   if x > 0 then (x ||^α upto (αfoo) (x − 1)) else 0
3 in upto bar 3
```

Expanding upto bar 3 will result in $(3 \parallel^{\mathtt{foo}} (2 \parallel^{\mathtt{barfoo}} (1 \parallel^{\mathtt{barfoofoo}} 0)))$, in which the choices do not share their names.

One may wonder why we do not adopt a construct for name generation, found in process calculi such as the π-calculus [39]. Simply put, the main reason is that name generation is another effectful construct and it is not clear how to integrate it into a manifest contract calculus.

4.2 Language

We demonstrate that the coordinated choice actually solves all problems we introduced by first formalizing a manifest contract calculus $\lambda^{H\parallel^{\Phi}}$ with coordinated choice in this section and show its soundness properties, including progress, type preservation, and contract satisfaction in the next section. $\lambda^{H\parallel^{\Phi}}$ is $\mathrm{PCF_H}$.

4.2.1 Syntax

The syntax of $\lambda^{H\parallel^{\Phi}}$ is shown in Figure 4.2. We use additional meta-variables: α, β ranging over string variables; Φ, Ψ ranging over *strings*; and R ranging over *raw results*.

Formally (and unlike examples in the last section), names for choices are strings, i.e., sequences of distinguished constants $\bullet$ and $\circ$ and string variables α. $\Phi\Psi$ is concatenation of two strings.

Types are extended with forall types $\forall\alpha.\tau$ for name abstractions, in which the string variable α is bound in τ.

$$
\begin{array}{rcl}
\Phi,\Psi & ::= & \bullet \mid \circ \mid \alpha \mid \Phi\Psi \\
\sigma,\tau & ::= & \texttt{bool} \mid \texttt{nat} \mid (x{:}\sigma) \to \tau \mid \{x{:}\tau \mid M\} \mid \forall\alpha.\tau \\
L,M,N & ::= & \texttt{True} \mid \texttt{False} \mid \texttt{0} \mid x \mid MN \mid M\Phi \mid \texttt{fix}\, f(x{:}\sigma) : \tau.M \mid \\
 & & \texttt{succ}(M) \mid \texttt{pred}(M) \mid \texttt{iszero}(M) \mid \texttt{if}\, L\, \texttt{then}\, M\, \texttt{else}\, N \mid \\
 & & \lambda\alpha.M \mid (M \parallel^{\Phi} N) \mid (M : \sigma \Rightarrow \tau) \mid \langle\!\langle M\, ?\, \{x{:}\tau \mid N\}\rangle\!\rangle \mid \\
 & & \langle\!\langle M \Longrightarrow V : \{x{:}\tau \mid N\}\rangle\!\rangle \mid \texttt{blame} \\
\bar{n} & ::= & \texttt{0} \mid \texttt{succ}(\bar{n}) \\
U,V & ::= & \texttt{True} \mid \texttt{False} \mid \bar{n} \mid \texttt{fix}\, f(x{:}\sigma) : \tau.M \mid \lambda\alpha.M \\
R & ::= & V \mid \texttt{blame} \mid (R_1 \parallel^{\Phi} R_2) \\
\mathcal{E} & ::= & \texttt{succ}(\square) \mid \texttt{pred}(\square) \mid \texttt{iszero}(\square) \mid \texttt{if}\, \square\, \texttt{then}\, M\, \texttt{else}\, N \mid \square N \mid \\
 & & V\square \mid \square\Phi \mid (\square : \sigma \Rightarrow \tau) \mid \langle\!\langle \square\, ?\, \{x{:}\tau \mid M\}\rangle\!\rangle \mid \langle\!\langle \square \Longrightarrow V : \{x{:}\tau \mid M\}\rangle\!\rangle \\
\Gamma & ::= & \varnothing \mid \Gamma,\alpha \mid \Gamma, x{:}\tau
\end{array}
$$

Figure 4.2: Syntax of $\lambda^{H\parallel^{\Phi}}$.

Expressions are extended with string abstractions $\lambda\alpha.M$, in which α is bound in M; string applications $M\,\Phi$; coordinated choices $(M \parallel^{\Phi} N)$. Note that lambda abstractions and fixpoint operator are integrated into one form $\texttt{fix}\, f(x{:}\sigma) : \tau.M$; while those are separated in $\mathrm{PCF_H}$.

Values are extended with string abstractions. We present raw results to represent the result of the evaluation of an expression since the result needs to represent several possibilities rather than a simple value or blame. The raw result consists of value V; `blame`; and choice of raw results $(R_1 \parallel^{\Phi} R_2)$.

Evaluation contexts stay in the original, and sub-expression reductions of choices are defined without evaluation contexts and as separated rules because those requires special treatment.

Type environments are extended with declarations of string variable.

Definition 4.2.1 (Orthant)**.** An *orthant*, denoted by Δ, is a set of pairs of a string and a symbol $+$ or $-$, written Φ_+ or Φ_-.

We introduce string substitution $[\alpha := \Phi]$ of a string for a string variable and substitution $[M := x]$ of an expression for a variable.

Definition 4.2.2 (String substitution)**.** *String substitution* $\Phi[\alpha \coloneqq \Psi]$ of Ψ for α in Φ is defined by:

$$
\begin{array}{c}
\bullet[\alpha \coloneqq \Psi] = \bullet \quad \circ[\alpha \coloneqq \Psi] = \circ \\
\alpha[\alpha \coloneqq \Psi] = \Psi \quad \beta[\alpha \coloneqq \Psi] = \beta \text{ (if } \alpha \neq \beta) \\
(\Phi_1\Phi_2)[\alpha \coloneqq \Psi] = \Phi_1[\alpha \coloneqq \Psi]\Phi_2[\alpha \coloneqq \Psi]
\end{array}
$$

We extend string substitution to one for orthants by:

$$
\Delta[\alpha \coloneqq \Phi] = \{\Psi[\alpha \coloneqq \Phi]_+ \mid \Psi_+ \in \Delta\} \cup \{\Psi[\alpha \coloneqq \Phi]_- \mid \Psi_- \in \Delta\}
$$

and for types (written $\tau[\alpha \coloneqq \Phi]$) and expressions (written $M[\alpha \coloneqq \Phi]$) in a straightforward capture-avoiding manner. We extend to type environments $\Gamma[\alpha \coloneqq \Phi]$ as follows:

$$
\begin{array}{rcl}
\varnothing[\alpha \coloneqq \Phi] & = & \varnothing \\
(\Gamma, y{:}\tau)[\alpha \coloneqq \Phi] & = & \Gamma[\alpha \coloneqq \Phi], y{:}\tau[\alpha \coloneqq \Phi] \\
(\Gamma, \beta)[\alpha \coloneqq \Phi] & = & \Gamma[\alpha \coloneqq \Phi], \beta
\end{array}
$$

Definition 4.2.3 (Substitutions)**.** We define capture-avoiding *substitution*, written $M[x := N]$ and $\tau[x := N]$, of an expression N for a variable x in an expression M and a type τ, respectively, in a usual manner. We extend to type environments $\Gamma[x := N]$ as follows:

$$\begin{aligned} \varnothing[x := N] &= \varnothing \\ (\Gamma, y{:}\tau)[x := N] &= \Gamma[x := N], y{:}\tau[x := N] \\ (\Gamma, \alpha)[x := N] &= \Gamma[x := N], \alpha \end{aligned}$$

Definition 4.2.4 (Environment domain and string domain)**.** The domain and string domain of a type environment, written $\mathtt{dom}(\Gamma)$ and $\mathtt{sdom}(\Gamma)$, respectively, are defined as follows.

$$\begin{aligned} \mathtt{dom}(\varnothing) &= \varnothing & \mathtt{sdom}(\varnothing) &= \varnothing \\ \mathtt{dom}(\Gamma, x{:}\tau) &= \mathtt{dom}(\Gamma) \cup \{x\} & \mathtt{sdom}(\Gamma, x{:}\tau) &= \mathtt{sdom}(\Gamma) \\ \mathtt{dom}(\Gamma, \alpha) &= \mathtt{dom}(\Gamma) & \mathtt{sdom}(\Gamma, \alpha) &= \mathtt{sdom}(\Gamma) \cup \{\alpha\} \end{aligned}$$

Definition 4.2.5 (Free variables and free string variables)**.** We denote the sets of free variables and free string variables in type τ by $\mathtt{fv}(\tau)$ and $\mathtt{fsv}(\tau)$, respectively; and denote those in expression M by $\mathtt{fv}(M)$ and $\mathtt{fsv}(M)$, respectively.

4.2.2 Semantics

As we have mentioned in Section 4.1, the reduction relation of $\lambda^{H\|^{\Phi}}$ is extended with an orthant and defines how an expression is reduced in a specific orthant. The relation formally defined by the rules in Figure 4.3 and Figure 4.4. Each defines deterministic reductions and nondeterministic reductions.

Deterministic reductions are almost same as the ones of PCF_H, but there are a few exceptions. First of all, there is an orthant on $\longrightarrow$. However, it changes nothing because no rules for deterministic reductions examine the orthant. Next is that the run-time checking rule (R-CForall) is added since types are extended with forall types. It pulls out common foralls as string abstraction. Note that the forall types of $\lambda^{H\|^{\Phi}}$ are *not* ones for parametric polymorphism, and thus, this simple reduction rule does not make a problem discussed by Sekiyama et al. [53]. The last is (R-CArrow), but the difference just caused by the integration of lambda abstractions and a fixpoint operator and what the rule does is same as the original rule does.

Nondeterministic reductions are as discussed in Section 4.1. One note is that choice of $\mathcal{E}$ in (R-Branch) is deterministic since evaluation contexts are defined as single frames.

Definition 4.2.6 (Multi-step reduction)**.** For each orthant Δ, the multi-step reduction relation $M \overset{\Delta}{\longrightarrow}{}^{*} N$ is defined as the reflexive and transitive closure of $\overset{\Delta}{\longrightarrow}$.

We define the notion of *results* below. Actually, a raw result may not be a normal form, e.g., $((1 \|^{\Phi} 2) \|^{\Phi} 3) \overset{\{\}}{\longrightarrow} (1 \|^{\Phi} 3)$. In other words, it depends on an orthant whether a given raw result is a normal form.

Definition 4.2.7 (Results in an orthant)**.** A raw result R is an actual result in an orthant Δ iff $\Delta \models R$ can be derived by the rules in Figure 4.5.

$$\frac{}{(\mathtt{fix}\, f(x{:}\sigma) : \tau.M)\, V \xrightarrow{\Delta} M[x \coloneqq V][f \coloneqq \mathtt{fix}\, f(x{:}\sigma) : \tau.M]} \quad \text{(R-Beta)}$$

$$\frac{}{(\lambda\alpha.M)\,\Phi \xrightarrow{\Delta} M[\alpha \coloneqq \Phi]} \quad \text{(R-Sigma)}$$

$$\frac{}{\mathtt{pred}(\mathtt{succ}(\bar{n})) \xrightarrow{\Delta} \bar{n}} \quad \text{(R-Pred)}$$

$$\frac{}{\mathtt{iszero}(0) \xrightarrow{\Delta} \mathtt{True}} \quad \text{(R-IsZeroT)}$$

$$\frac{}{\mathtt{iszero}(\mathtt{succ}(\bar{n})) \xrightarrow{\Delta} \mathtt{False}} \quad \text{(R-IsZeroF)}$$

$$\frac{}{\mathtt{if}\,\mathtt{True}\,\mathtt{then}\,M\,\mathtt{else}\,N \xrightarrow{\Delta} M} \quad \text{(R-IfT)}$$

$$\frac{}{\mathtt{if}\,\mathtt{False}\,\mathtt{then}\,M\,\mathtt{else}\,N \xrightarrow{\Delta} N} \quad \text{(R-IfF)}$$

$$\frac{}{(V : \mathtt{bool} \Rightarrow \mathtt{bool}) \xrightarrow{\Delta} V} \quad \text{(R-CBool)}$$

$$\frac{}{(V : \mathtt{nat} \Rightarrow \mathtt{nat}) \xrightarrow{\Delta} V} \quad \text{(R-CNat)}$$

$$\frac{}{(V : \forall\alpha.\sigma \Rightarrow \forall\alpha.\tau) \xrightarrow{\Delta} \lambda\alpha.(V\alpha : \sigma \Rightarrow \tau)} \quad \text{(R-CForall)}$$

$$\frac{(x \neq y)}{\begin{array}{l}(V : (y{:}\sigma_1) \to \sigma_2 \Rightarrow (x{:}\tau_1) \to \tau_2) \xrightarrow{\Delta} \\ \quad \mathtt{fix}\, x(x{:}\tau_1) : \tau_2.(\mathtt{fix}\, y(y{:}\sigma_1) : \tau_2.(V\, y : \sigma_2 \Rightarrow \tau_2))\,(x : \tau_1 \Rightarrow \sigma_1)\end{array}} \quad \text{(R-CArrow)}$$

$$\frac{}{(V : \{x{:}\sigma \mid M\} \Rightarrow \tau) \xrightarrow{\Delta} (V : \sigma \Rightarrow \tau)} \quad \text{(R-CForget)}$$

$$\frac{(\sigma \neq \{x'{:}\tau' \mid M'\})}{(V : \sigma \Rightarrow \{x{:}\tau \mid M\}) \xrightarrow{\Delta} \langle\!\langle (V : \sigma \Rightarrow \tau) \mathrel{?} \{x{:}\tau \mid M\} \rangle\!\rangle} \quad \text{(R-CWait)}$$

$$\frac{}{\langle\!\langle V \mathrel{?} \{x{:}\tau \mid M\} \rangle\!\rangle \xrightarrow{\Delta} \langle\!\langle M[x \coloneqq V] \Longrightarrow V : \{x{:}\tau \mid M\} \rangle\!\rangle} \quad \text{(R-CActive)}$$

$$\frac{}{\langle\!\langle \mathtt{True} \Longrightarrow V : \{x{:}\tau \mid M\} \rangle\!\rangle \xrightarrow{\Delta} V} \quad \text{(R-CSucceed)}$$

$$\frac{}{\langle\!\langle \mathtt{False} \Longrightarrow V : \{x{:}\tau \mid M\} \rangle\!\rangle \xrightarrow{\Delta} \mathtt{blame}} \quad \text{(R-CFail)}$$

$$\frac{M \xrightarrow{\Delta} N}{\mathcal{E}[M] \xrightarrow{\Delta} \mathcal{E}[N]} \quad \text{(R-Ctx)}$$

$$\frac{}{\mathcal{E}[\mathtt{blame}] \xrightarrow{\Delta} \mathtt{blame}} \quad \text{(R-Exit)}$$

Figure 4.3: Operational semantics of $\lambda^{H\|^{\Phi}}$ (1): deterministic part

$$(M \parallel^{\Phi} N) \xrightarrow{\Delta \cup \{\Phi_+\}} M \qquad \text{(R-WorldL)}$$

$$(M \parallel^{\Phi} N) \xrightarrow{\Delta \cup \{\Phi_-\}} N \qquad \text{(R-WorldR)}$$

$$\mathcal{E}[(M_1 \parallel^{\Phi} M_2)] \xrightarrow{\Delta} (\mathcal{E}[M_1] \parallel^{\Phi} \mathcal{E}[M_2]) \qquad \text{(R-Branch)}$$

$$\frac{M \xrightarrow{\Delta \cup \{\Phi_+\}} M'}{(M \parallel^{\Phi} N) \xrightarrow{\Delta} (M' \parallel^{\Phi} N)} \qquad \text{(R-ChoiceL)}$$

$$\frac{N \xrightarrow{\Delta \cup \{\Phi_-\}} N'}{(M \parallel^{\Phi} N) \xrightarrow{\Delta} (M \parallel^{\Phi} N')} \qquad \text{(R-ChoiceR)}$$

Figure 4.4: Operational semantics of $\lambda^{H\parallel^{\Phi}}$ (2): nondeterministic part

$$\Delta \vDash V \qquad \text{(Res-Value)}$$

$$\Delta \vDash \texttt{blame} \qquad \text{(Res-Blame)}$$

$$\frac{\Delta \uplus \{\Phi_+\} \vDash R_1 \qquad \Delta \uplus \{\Phi_-\} \vDash R_2}{\Delta \vDash (R_1 \parallel^{\Phi} R_2)} \qquad \text{(Res-Choice)}$$

Figure 4.5: Results

4.2.3 Type System

The type system of $\lambda^{H\parallel^{\Phi}}$ consists of similar relations to $\mathrm{PCF_H}$, namely $\sigma \sim \tau$, $\sigma \overset{\Delta}{\equiv} \tau$, $\Gamma \Vdash^{\Delta} \tau$, and $\Gamma \vdash^{\Delta} M : \tau$. There are two differences from $\mathrm{PCF_H}$: the last three involve orthants, so for example, $\sigma \overset{\Delta}{\equiv} \tau$ read "σ and τ are equivalent in Δ"; and there is no explicit relation for environment well-formedness. The former comes from the fact that the reduction relation of $\lambda^{H\parallel^{\Phi}}$ involves orthants. For the latter, well-formedness of an environment is required for $\lambda^{H\parallel^{\Phi}}$, of course; but it is checked as a part of well-formedness checking and typeabilty checking. For $\lambda^{H\parallel^{\Phi}}$, we call Γ occurring in a derivation well-formed.

Type compatibility and equivalence

The type compatibility relation and the type equivalence relation are defined by the rules in Figure 4.6 and Figure 4.7, respectively. Both are straightforward extension of ones of $\mathrm{PCF_H}$ with the rules for forall types, namely (C-Forall) and (E-Forall). In the type equivalence relation, orthants are involved; but an orthant is just passed to the reduction relation used in premises and not examined by the equivalence relation itself.

Type well-formedness and typing

The type well-formedness relation and the typing relation are defined by the rules in Figure 4.8, Figure 4.9, and Figure 4.10. Well-formedness checking for environments, we has touched, are done by the explicit weakening rules (W-Weaken), (W-SWeaken), (T-Weaken), and (T-SWeaken). The checking is done in pieces and middles of a derivation since the axiom rules demand the environment used empty. Assuming the difference how well-formedness

$$\frac{}{\texttt{bool} \sim \texttt{bool}} \quad \text{(C-Bool)}$$

$$\frac{}{\texttt{nat} \sim \texttt{nat}} \quad \text{(C-Nat)}$$

$$\frac{\sigma_1 \sim \tau_1 \qquad \sigma_2 \sim \tau_2}{(x{:}\sigma_1) \to \sigma_2 \sim (x{:}\tau_1) \to \tau_2} \quad \text{(C-Arrow)}$$

$$\frac{\sigma \sim \tau}{\{x{:}\sigma \mid M\} \sim \tau} \quad \text{(C-RefineL)}$$

$$\frac{\sigma \sim \tau}{\sigma \sim \{x{:}\tau \mid M\}} \quad \text{(C-RefineR)}$$

$$\frac{\sigma \sim \tau}{\forall \alpha.\sigma \sim \forall \alpha.\tau} \quad \text{(C-Forall)}$$

Figure 4.6: Type compatibility of $\lambda^{H\|^\Phi}$

$$\frac{}{\texttt{bool} \overset{\Delta}{\equiv} \texttt{bool}} \quad \text{(E-Bool)}$$

$$\frac{}{\texttt{nat} \overset{\Delta}{\equiv} \texttt{nat}} \quad \text{(E-Nat)}$$

$$\frac{\sigma_1 \overset{\Delta}{\equiv} \tau_1 \qquad \sigma_2 \overset{\Delta}{\equiv} \tau_2}{(x{:}\sigma_1) \to \sigma_2 \overset{\Delta}{\equiv} (x{:}\tau_1) \to \tau_2} \quad \text{(E-Arrow)}$$

$$\frac{\sigma \overset{\Delta}{\equiv} \tau \qquad N \overset{\Delta}{\longrightarrow} N'}{\{x{:}\sigma \mid M[z := N]\} \overset{\Delta}{\equiv} \{x{:}\tau \mid M[z := N']\}} \quad \text{(E-RefineL)}$$

$$\frac{\sigma \overset{\Delta}{\equiv} \tau \qquad N \overset{\Delta}{\longrightarrow} N'}{\{x{:}\sigma \mid M[z := N']\} \overset{\Delta}{\equiv} \{x{:}\tau \mid M[z := N]\}} \quad \text{(E-RefineR)}$$

$$\frac{\tau_1 \overset{\Delta}{\equiv} \tau_2 \qquad \tau_2 \overset{\Delta}{\equiv} \tau_3}{\tau_1 \overset{\Delta}{\equiv} \tau_3} \quad \text{(E-Trans)}$$

$$\frac{\sigma \overset{\Delta}{\equiv} \tau}{\forall \alpha.\sigma \overset{\Delta}{\equiv} \forall \alpha.\tau} \quad \text{(E-Forall)}$$

Figure 4.7: Type equivalence of $\lambda^{H\|^\Phi}$

$$\frac{}{\Vdash^{\Delta} \texttt{bool}} \quad \text{(W-Bool)}$$

$$\frac{}{\Vdash^{\Delta} \texttt{nat}} \quad \text{(W-Nat)}$$

$$\frac{\Gamma, x{:}\sigma \Vdash^{\Delta} \tau}{\Gamma \Vdash^{\Delta} (x{:}\sigma) \to \tau} \quad \text{(W-Arrow)}$$

$$\frac{\Gamma, x{:}\tau \vdash^{\Delta} M : \texttt{bool}}{\Gamma \Vdash^{\Delta} \{x{:}\tau \mid M\}} \quad \text{(W-Refine)}$$

$$\frac{\Gamma, \alpha \Vdash^{\Delta} \tau \qquad (\alpha \notin \texttt{fsv}(\Delta))}{\Gamma \Vdash^{\Delta} \forall \alpha.\tau} \quad \text{(W-Forall)}$$

$$\frac{\Gamma \Vdash^{\Delta} \sigma \qquad \Gamma \Vdash^{\Delta} \tau \qquad (x \notin \texttt{dom}(\Gamma))}{\Gamma, x{:}\sigma \Vdash^{\Delta} \tau} \quad \text{(W-Weaken)}$$

$$\frac{\Gamma \Vdash^{\Delta} \tau \qquad (\alpha \notin \texttt{sdom}(\Gamma))}{\Gamma, \alpha \Vdash^{\Delta} \tau} \quad \text{(W-SWeaken)}$$

Figure 4.8: Type system of $\lambda^{H\|^{\Phi}}$ (1): well-formedness rules

of environments is dealt with, most rules define the same things which the corresponding ones of PCF_H do.

The additional rules are for the additional terms. (W-Forall) is for forall types, where the side-condition guarantees freshness of the bound string variable. (T-SApp) is for string applications, where $\texttt{fsv}(\Phi) \subseteq \texttt{sdom}(\Gamma)$ checks that all string variables that appear are declared in the environment, e.g., a kind of well-formedness checking for strings. (T-SFun) is for string abstractions, where the side-condition does the same thing as (W-Forall) does. (T-Choice) is the heart of the type system and reflects the following intuition: a choice is well typed if its operands have the same type in corresponding orthants. The premise $\Gamma \Vdash^{\Delta} \tau$ is required because the first two ensures only that τ is well-formed in orthants $\Delta \cup \{\Phi_+\}$ and $\Delta \cup \{\Phi_-\}$ but not necessarily in Δ.

Remark 4.2.8. An orthant appear in the judgment do not closed under a typing environment. Actually, (W-SWeaken) and (T-SWeaken) work as if an additional declaration of a free string variable captures one in an orthant. See the derivation in Figure 4.11.

Remark 4.2.9. As we have touched in Section 2.1, the type equivalence playes a crucial role to prove a subject reduction property in a dependently typed system. In $\lambda^{H\|^{\Phi}}$, where $f\,(M_1 \|^{\Phi} M_2)$ reduces to $(f\,M_1 \|^{\Phi} f\,M_2)$, showing subject reduction is more tricky. The crux here is that type $\tau[x \coloneqq (M_1 \|^{\Phi} M_2)]$ is equivalent to $\tau[x \coloneqq M_1]$ in an orthant including Φ_+ and also to $\tau[x \coloneqq M_2]$ in an orthant including Φ_- (but not equivalent to either of them in an orthant without them). So, $f\,M_1$ and $f\,M_2$ are given type $\tau[x \coloneqq (M_1 \|^{\Phi} M_2)]$ in an orthant including Φ_+ and Φ_-, respectively, and by the rule (T-Choice), $(f\,M_1 \|^{\Phi} f\,M_2)$ is given type $\tau[x \coloneqq (M_1 \|^{\Phi} M_2)]$.

Example 4.2.10 (Concrete example for the remark above). Let $\Gamma = f{:}(x{:}\texttt{bool}) \to \{z{:}\texttt{bool} \mid x\}$. Then, $\Gamma \vdash^{\{\}} f\,(\texttt{True} \|^{\text{foo}} \texttt{False}) : \{z{:}\texttt{bool} \mid (\texttt{True} \|^{\text{foo}} \texttt{False})\}$ is derived by (T-App). Since $f\,(\texttt{True} \|^{\text{foo}} \texttt{False}) \xrightarrow{\{\}} (f\,\texttt{True} \|^{\text{foo}} f\,\texttt{False})$,

$$\frac{\Gamma \Vdash^{\Delta} \tau \qquad (x \notin \mathrm{dom}(\Gamma))}{\Gamma, x{:}\tau \vdash^{\Delta} x : \tau} \quad \text{(T-Var)}$$

$$\frac{}{\vdash^{\Delta} \mathtt{True} : \mathtt{bool}} \quad \text{(T-True)}$$

$$\frac{}{\vdash^{\Delta} \mathtt{False} : \mathtt{bool}} \quad \text{(T-False)}$$

$$\frac{}{\vdash^{\Delta} \mathtt{0} : \mathtt{nat}} \quad \text{(T-Zero)}$$

$$\frac{\Gamma \vdash^{\Delta} M : \mathtt{nat}}{\Gamma \vdash^{\Delta} \mathtt{succ}(M) : \mathtt{nat}} \quad \text{(T-Succ)}$$

$$\frac{\Gamma \vdash^{\Delta} M : \{x{:}\mathtt{nat} \mid \mathtt{if}\,\mathtt{iszero}(x)\,\mathtt{then}\,\mathtt{False}\,\mathtt{else}\,\mathtt{True}\}}{\Gamma \vdash^{\Delta} \mathtt{pred}(M) : \mathtt{nat}} \quad \text{(T-Pred)}$$

$$\frac{\Gamma \vdash^{\Delta} M : \mathtt{nat}}{\Gamma \vdash^{\Delta} \mathtt{iszero}(M) : \mathtt{bool}} \quad \text{(T-IsZero)}$$

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

$$\frac{\Gamma \vdash^{\Delta} M : (x{:}\sigma) \to \tau \qquad \Gamma \vdash^{\Delta} N : \sigma}{\Gamma \vdash^{\Delta} M N : \tau[x \coloneqq N]} \quad \text{(T-App)}$$

$$\frac{\Gamma, f{:}(x{:}\sigma) \to \tau, x{:}\sigma \vdash^{\Delta} M : \tau}{\Gamma \vdash^{\Delta} \mathtt{fix}\,f(x{:}\sigma) : \tau.M : (x{:}\sigma) \to \tau} \quad \text{(T-Fun)}$$

$$\frac{\Gamma \vdash^{\Delta} M : \sigma \qquad \Gamma \Vdash^{\Delta} \tau \qquad \sigma \sim \tau}{\Gamma \vdash^{\Delta} (M : \sigma \Rightarrow \tau) : \tau} \quad \text{(T-Cast)}$$

$$\frac{\Gamma \vdash^{\Delta} M : \forall \alpha.\tau \qquad (\mathtt{fsv}(\Phi) \subseteq \mathrm{sdom}(\Gamma))}{\Gamma \vdash^{\Delta} M \Phi : \tau[\alpha \coloneqq \Phi]} \quad \text{(T-SApp)}$$

$$\frac{\Gamma, \alpha \vdash^{\Delta} M : \tau \qquad \alpha \notin \mathtt{fsv}(\Delta)}{\Gamma \vdash^{\Delta} \lambda\alpha.M : \forall \alpha.\tau} \quad \text{(T-SFun)}$$

$$\frac{\Gamma \vdash^{\Delta \cup \{\Phi_+\}} M : \tau \qquad \Gamma \vdash^{\Delta \cup \{\Phi_-\}} N : \tau \qquad \Gamma \Vdash^{\Delta} \tau \qquad (\mathtt{fsv}(\Phi) \subseteq \mathrm{sdom}(\Gamma))}{\Gamma \vdash^{\Delta} (M \parallel^{\Phi} N) : \tau} \quad \text{(T-Choice)}$$

$$\frac{\Gamma \Vdash^{\Delta} \sigma \qquad \Gamma \vdash^{\Delta} M : \tau \qquad (x \notin \mathrm{dom}(\Gamma))}{\Gamma, x{:}\sigma \vdash^{\Delta} M : \tau} \quad \text{(T-Weaken)}$$

$$\frac{\Gamma \vdash^{\Delta} M : \tau \qquad (\alpha \notin \mathrm{sdom}(\Gamma))}{\Gamma, \alpha \vdash^{\Delta} M : \tau} \quad \text{(T-SWeaken)}$$

Figure 4.9: Type system of $\lambda^{H\parallel^{\Phi}}$ (2): typing rules

$$\frac{\vdash^{\Delta} M : \tau \qquad \Vdash^{\Delta} \{x{:}\tau \mid N\}}{\vdash^{\Delta} \langle\!\langle M \mathrel{?} \{x{:}\tau \mid N\}\rangle\!\rangle : \{x{:}\tau \mid N\}} \quad \text{(T-Waiting)}$$

$$\frac{\vdash^{\Delta} M : \texttt{bool} \qquad \vdash^{\Delta} V : \tau \qquad \Vdash^{\Delta} \{x{:}\tau \mid N\} \qquad (N[x \coloneqq V] \xrightarrow{\Delta}{}^{*} M)}{\vdash^{\Delta} \langle\!\langle M \Longrightarrow V : \{x{:}\tau \mid N\}\rangle\!\rangle : \{x{:}\tau \mid N\}} \quad \text{(T-Active)}$$

$$\frac{\Vdash^{\Delta} \tau}{\vdash^{\Delta} \texttt{blame} : \tau} \quad \text{(T-Blame)}$$

$$\frac{\vdash^{\Delta} M : \sigma \qquad \Vdash^{\Delta} \tau \qquad \sigma \overset{\Delta}{\equiv} \tau}{\vdash^{\Delta} M : \tau} \quad \text{(T-Conv)}$$

$$\frac{\vdash^{\Delta} V : \{x{:}\tau \mid M\}}{\vdash^{\Delta} V : \tau} \quad \text{(T-Forget)}$$

$$\frac{\vdash^{\Delta} V : \tau \qquad \Vdash^{\Delta} \{x{:}\tau \mid M\} \qquad (M[x \coloneqq V] \xrightarrow{\Delta}{}^{*} \texttt{True})}{\vdash^{\Delta} V : \{x{:}\tau \mid M\}} \quad \text{(T-Exact)}$$

Figure 4.10: Type system of $\lambda^{H\|^{\Phi}}$ (3): run-time typing rules

$\Gamma \vdash^{\{\}} (f\,\texttt{True} \parallel^{\texttt{foo}} f\,\texttt{False}) : \{z{:}\texttt{bool} \mid (\texttt{True} \parallel^{\texttt{foo}} \texttt{False})\}$ must be derived for subject reduction. Actually, it can be derived as Figure 4.12. The point is $f\,\texttt{True}$ inhabits the orthant $\{\texttt{foo}_+\}$ and $\{z{:}\texttt{bool} \mid \texttt{True}\} \overset{\{\texttt{foo}_+\}}{\equiv} \{z{:}\texttt{bool} \mid (\texttt{True} \parallel^{\texttt{foo}} \texttt{False})\}$. This is the way our type system preserves an expression type before and after distributing a function argument.

4.3 Properties

The type soundness of $\lambda^{H\|^{\Phi}}$ is consist of the desirable properties of a manifest contract system, namely subject reduction, progress, and value inversion; and, in addition to those, *contract satisfaction.* which is a generalization of value inversion under nondeterminism.

Contract satisfaction deserves some more informal explanation. Suppose M has type $\{x{:}\tau \mid N\}$ and all of its nondeterministic executions terminate. We denote the result of the evaluation as R, which is essentially a multi set of values and blame. Roughly speaking, the contract satisfaction says R *satisfies* N—for any value, denoted V_i, in R satisfies the predicate N, i.e., $N[x \coloneqq V_i]$ *nondeterministically evaluates to* $\texttt{True}$. Note that blame in R is ignored because it denotes failed execution. The phrase "$N[x \coloneqq V_i]$ nondeterministically evaluates to $\texttt{True}$" means that $N[x \coloneqq V_i] \xrightarrow{\Delta}{}^{*} \texttt{True}$, where Δ is the orthant that V_i inhabits. One might wonder if it is too strong to require a predicate N to reduce to $\texttt{True}$, not to an expression like $(\texttt{True} \parallel^{\Phi} M)$, even when N can be nondeterministic. For example, consider type $\tau = \{x{:}\texttt{nat} \mid (x = 1 \parallel^{\Phi} x = 2)\}$, whose predicate is nondeterministic, and a constant 1; then, $(x = 1 \parallel^{\Phi} x = 2)[x \coloneqq 1] \longrightarrow^{*} (\texttt{True} \parallel^{\Phi} \texttt{False}) \neq \texttt{True}$, which may look like a counterexample to contract satisfaction. Actually, it is not. First of all, $\vdash^{\{\}} 1 : \tau$ cannot be derived because the type of 1 is only $\texttt{nat}$. We can make the expression well typed by inserting a cast. The judgment $\vdash^{\{\}} (1 : \texttt{nat} \Rightarrow \tau) : \tau$ is derivable and this cast expression is reduced to $(1 \parallel^{\Phi} \texttt{blame})$. Now, the trick is to use the orthant

$$\dfrac{\dfrac{\vdash^{\{\alpha_+\}} \texttt{True} : \texttt{bool}}{\alpha \vdash^{\{\alpha_+\}} \texttt{True} : \texttt{bool}}\ (\text{T-CWeaken}) \qquad \dfrac{\vdash^{\{\alpha_-\}} \texttt{False} : \texttt{bool}}{\alpha \vdash^{\{\alpha_-\}} \texttt{False} : \texttt{bool}}\ (\text{T-CWeaken}) \qquad \dfrac{\Vdash^{\{\}} \texttt{bool}}{\alpha \Vdash^{\{\}} \texttt{bool}}\ (\text{W-CWeaken})}{\dfrac{\alpha \vdash^{\{\}} (\texttt{True} \parallel^{\alpha} \texttt{False}) : \texttt{bool}}{\vdash^{\{\}} \lambda\alpha.(\texttt{True} \parallel^{\alpha} \texttt{False}) : \forall\alpha.\texttt{bool}}\ (\text{T-SFun})}\ (\text{T-Choice})$$

Figure 4.11: Example of typing capturing a free variable in an orthant in the middle

$$\dfrac{\dfrac{\Gamma \vdash^{\{\texttt{foo}_+\}} f\,\texttt{True} : \{z{:}\texttt{bool} \mid \texttt{True}\} \qquad \cdots}{\Gamma \vdash^{\{\texttt{foo}_+\}} f\,\texttt{True} : \{z{:}\texttt{bool} \mid (\texttt{True} \parallel^{\texttt{foo}} \texttt{False})\}}\ (\text{T-Conv}) \qquad \dfrac{\Gamma \vdash^{\{\texttt{foo}_-\}} f\,\texttt{False} : \{z{:}\texttt{bool} \mid \texttt{False}\} \qquad \cdots}{\Gamma \vdash^{\{\texttt{foo}_-\}} f\,\texttt{False} : \{z{:}\texttt{bool} \mid (\texttt{True} \parallel^{\texttt{foo}} \texttt{False})\}}\ (\text{T-Conv}) \qquad \cdots}{\Gamma \vdash^{\{\}} (f\,\texttt{True} \parallel^{\texttt{foo}} f\,\texttt{False}) : \{z{:}\texttt{bool} \mid (\texttt{True} \parallel^{\texttt{foo}} \texttt{False})\}}$$

Figure 4.12: Example of typing for an expression after distributing an function argument

that each value inhabit to run the predicate. So, the predicate is run in $\{\Phi_+\}$ for 1 and we get $(x = 1 \parallel^\Phi x = 2)[x \coloneqq 1] \xrightarrow{\{\Phi_+\}}{}^* \texttt{True}$.

We start with the following lemma, which is used everywhere implicitly in proofs and states a judgment that holds in some orthant holds in a more split orthant. This lemma is why we allow nonsensical orthants (see Remark 4.1.1).

Lemma 4.3.1 (Orthant weakening)**.** *Suppose* $\Delta \subseteq \Delta'$.

1. *If* $M \xrightarrow{\Delta} N$, *then* $M \xrightarrow{\Delta'} N$.
2. *If* $M \xrightarrow{\Delta}{}^* N$, *then* $M \xrightarrow{\Delta'}{}^* N$.
3. *If* $M \overset{\Delta}{\equiv} N$, *then* $M \overset{\Delta'}{\equiv} N$.
4. *If* $\Gamma \Vdash^\Delta \tau$, *then* $\Gamma \Vdash^{\Delta'} \tau$.
5. *If* $\Gamma \vdash^\Delta M : \tau$, *then* $\Gamma \vdash^{\Delta'} M : \tau$.

Proof of Lemma 4.3.1 (1). The proof is straightforward by the induction on the given derivation. □

Proof of Lemma 4.3.1 (2). This is a corollary of Lemma 4.3.1 (1). □

Proof of Lemma 4.3.1 (3). The proof is done by the induction on the given derivation. In the cases (E-RefineL) and (E-RefineR), we use Lemma 4.3.1 (1). □

Proof of Lemma 4.3.1 (4) and (5). The proof is done by the mutually induction on the given derivation. In the cases (T-Active) and (T-Exact), we use Lemma 4.3.1 (2). In the case (T-Conv), we use Lemma 4.3.1 (3). □

4.3.1 Co-termination

In this subsection we show the type equivalence preserves the meaning of predicates, which is required to show the value inversion property. Technically, this is guaranteed by the following lemma.

Definition 4.3.2 (Normal orthant)**.** An orthant Δ is *normal* if and only if $\{\Phi_+, \Phi_-\} \nsubseteq \Delta$ for any Φ.

Lemma 4.3.3 (Co-termination for true)**.** *Suppose* Δ *is* normal *and* $N \xrightarrow{\Delta} N'$. *Then* $M[x \coloneqq N] \xrightarrow{\Delta}{}^* \mathtt{True} \iff M[x \coloneqq N'] \xrightarrow{\Delta}{}^* \mathtt{True}$.

This is a technical reason for which we develop the orthant-based reduction of $\lambda^{H\parallel^\Phi}$. If we adopted a naive choice semantics $(M \parallel^\Phi N) \longrightarrow M$ and $(M \parallel^\Phi N) \longrightarrow N$ (regardless of orthants), we would have $(\texttt{True} \parallel^\Phi \texttt{False}) \longrightarrow \texttt{False}$ and $x[x \coloneqq (\texttt{True} \parallel^\Phi \texttt{False})] \longrightarrow \texttt{True}$ and $x[x \coloneqq \texttt{False}] \longrightarrow \texttt{False}$, breaking the property. Actually, the same problem still occurs in nonsensical orthants, so we consider only the normal orthants, excluding nonsensical orthants.

Lemma 4.3.3 is, however, not easy to prove directly since our operational semantics is nondeterministic even if Δ is restricted to be normal and moreover subexpression would be

$$\frac{}{\texttt{True} \Downarrow^{\Delta} \texttt{True}} \quad \text{(N-True)}$$

$$\frac{}{\texttt{False} \Downarrow^{\Delta} \texttt{False}} \quad \text{(N-False)}$$

$$\frac{}{\texttt{0} \Downarrow^{\Delta} \texttt{0}} \quad \text{(N-Zero)}$$

$$\frac{}{\texttt{fix}\, f(x{:}\sigma) : \tau.M \Downarrow^{\Delta} \texttt{fix}\, f(x{:}\sigma) : \tau.M} \quad \text{(N-Fun)}$$

$$\frac{}{\lambda\alpha.M \Downarrow^{\Delta} \lambda\alpha.M} \quad \text{(N-SFun)}$$

$$\frac{M \Downarrow^{\Delta} \bar{n}}{\texttt{succ}\,(M) \Downarrow^{\Delta} \texttt{succ}\,(\bar{n})} \quad \text{(N-Succ)}$$

$$\frac{M \Downarrow^{\Delta} \texttt{succ}\,(\bar{n})}{\texttt{pred}\,(M) \Downarrow^{\Delta} \bar{n}} \quad \text{(N-Pred)}$$

$$\frac{M \Downarrow^{\Delta} \texttt{0}}{\texttt{iszero}\,(M) \Downarrow^{\Delta} \texttt{True}} \quad \text{(N-IsZeroT)}$$

$$\frac{M \Downarrow^{\Delta} \texttt{succ}\,(\bar{n})}{\texttt{iszero}\,(M) \Downarrow^{\Delta} \texttt{False}} \quad \text{(N-IsZeroF)}$$

$$\frac{M \Downarrow^{\Delta} \texttt{fix}\, f(x{:}\sigma) : \tau.M_1 \qquad N \Downarrow^{\Delta} V_1 \qquad M_1[x \coloneqq V_1][f \coloneqq \texttt{fix}\, f(x{:}\sigma) : \tau.M_1] \Downarrow^{\Delta} R}{M\,N \Downarrow^{\Delta} R}$$

(N-Beta)

$$\frac{M \Downarrow^{\Delta} \lambda\alpha.M_1 \qquad M_1[\alpha \coloneqq \Phi] \Downarrow^{\Delta} R}{M\,\Phi \Downarrow^{\Delta} R} \quad \text{(N-Sigma)}$$

$$\frac{M_1 \Downarrow^{\Delta} \texttt{True} \qquad M_2 \Downarrow^{\Delta} R}{\texttt{if}\, M_1\, \texttt{then}\, M_2\, \texttt{else}\, M_3 \Downarrow^{\Delta} R} \quad \text{(N-IfT)}$$

$$\frac{M_1 \Downarrow^{\Delta} \texttt{False} \qquad M_3 \Downarrow^{\Delta} R}{\texttt{if}\, M_1\, \texttt{then}\, M_2\, \texttt{else}\, M_3 \Downarrow^{\Delta} R} \quad \text{(N-IfF)}$$

Figure 4.13: Deterministic evaluation (1)

$$\frac{M \Downarrow^{\Delta} V}{(M : \mathtt{bool} \Rightarrow \mathtt{bool}) \Downarrow^{\Delta} V} \quad \text{(N-CBool)}$$

$$\frac{M \Downarrow^{\Delta} V}{(M : \mathtt{nat} \Rightarrow \mathtt{nat}) \Downarrow^{\Delta} V} \quad \text{(N-CNat)}$$

$$\frac{M \Downarrow^{\Delta} V}{(M : \forall \alpha.\sigma \Rightarrow \forall \alpha.\tau) \Downarrow^{\Delta} \lambda \alpha.(V\,\alpha : \sigma \Rightarrow \tau)} \quad \text{(N-CForall)}$$

$$\frac{M \Downarrow^{\Delta} V \qquad (x \neq y)}{\begin{array}{l}(M : (y{:}\sigma_1) \to \sigma_2 \Rightarrow (x{:}\tau_1) \to \tau_2) \Downarrow^{\Delta} \\ \quad \mathtt{fix}\, x(x{:}\tau_1) : \tau_2.(\mathtt{fix}\, y(y{:}\sigma_1) : \tau_2.(V\,y : \sigma_2 \Rightarrow \tau_2))\,(x : \tau_1 \Rightarrow \sigma_1)\end{array}} \quad \text{(N-CArrow)}$$

$$\frac{M \Downarrow^{\Delta} V \qquad (V : \sigma \Rightarrow \tau) \Downarrow^{\Delta} R}{(M : \{x{:}\sigma \mid N\} \Rightarrow \tau) \Downarrow^{\Delta} R} \quad \text{(N-CForget)}$$

$$\frac{M \Downarrow^{\Delta} V \qquad \langle\!\langle (V : \sigma \Rightarrow \tau)\,?\,\{x{:}\tau \mid N\} \rangle\!\rangle \Downarrow^{\Delta} R \qquad (\sigma \neq \{x'{:}\tau' \mid M'\})}{(M : \sigma \Rightarrow \{x{:}\tau \mid N\}) \Downarrow^{\Delta} R} \quad \text{(N-CPreCheck)}$$

$$\frac{M \Downarrow^{\Delta} V \qquad \langle\!\langle N[x := V] \Longrightarrow V : \{x{:}\tau \mid N\} \rangle\!\rangle \Downarrow^{\Delta} R}{\langle\!\langle M\,?\,\{x{:}\tau \mid N\} \rangle\!\rangle \Downarrow^{\Delta} R} \quad \text{(N-Check)}$$

$$\frac{M \Downarrow^{\Delta} \mathtt{True}}{\langle\!\langle M \Longrightarrow V : \{x{:}\tau \mid N\} \rangle\!\rangle \Downarrow^{\Delta} V} \quad \text{(N-Good)}$$

$$\frac{M \Downarrow^{\Delta} \mathtt{False}}{\langle\!\langle M \Longrightarrow V : \{x{:}\tau \mid N\} \rangle\!\rangle \Downarrow^{\Delta} \mathtt{blame}} \quad \text{(N-Bad)}$$

$$\frac{}{\mathtt{blame} \Downarrow^{\Delta} \mathtt{blame}} \quad \text{(N-Blame)}$$

$$\frac{M \Downarrow^{\Delta} \mathtt{blame} \qquad (\mathcal{E} \neq V\,\square)}{\mathcal{E}[M] \Downarrow^{\Delta} \mathtt{blame}} \quad \text{(N-CtxBlame)}$$

$$\frac{M \Downarrow^{\Delta} (R_1 \parallel^{\Phi} R_2) \qquad (\mathcal{E}[R_1] \parallel^{\Phi} \mathcal{E}[R_2]) \Downarrow^{\Delta} R \qquad (\mathcal{E} \neq V\,\square)}{\mathcal{E}[M] \Downarrow^{\Delta} R} \quad \text{(N-Branch)}$$

$$\frac{M \Downarrow^{\Delta} V \qquad N \Downarrow^{\Delta} \mathtt{blame}}{M\,N \Downarrow^{\Delta} \mathtt{blame}} \quad \text{(N-CtxBlameA)}$$

$$\frac{M \Downarrow^{\Delta} V \qquad N \Downarrow^{\Delta} (R_1 \parallel^{\Phi} R_2) \qquad (V R_1 \parallel^{\Phi} V R_2) \Downarrow^{\Delta} R}{M\,N \Downarrow^{\Delta} R} \quad \text{(N-BranchA)}$$

$$\frac{M_1 \Downarrow^{\Delta \uplus \{\Phi_+\}} R_1 \qquad M_2 \Downarrow^{\Delta \uplus \{\Phi_-\}} R_2}{(M_1 \parallel^{\Phi} M_2) \Downarrow^{\Delta} (R_1 \parallel^{\Phi} R_2)} \quad \text{(N-Choice)}$$

$$\frac{M_1 \Downarrow^{\Delta \cup \{\Phi_+\}} R \qquad (\Phi_- \notin \Delta)}{(M_1 \parallel^{\Phi} M_2) \Downarrow^{\Delta \cup \{\Phi_+\}} R} \quad \text{(N-WorldL)}$$

$$\frac{M_2 \Downarrow^{\Delta \cup \{\Phi_-\}} R \qquad (\Phi_+ \notin \Delta)}{(M_1 \parallel^{\Phi} M_2) \Downarrow^{\Delta \cup \{\Phi_-\}} R} \quad \text{(N-WorldR)}$$

Figure 4.14: Deterministic evaluation (2)

evaluated in a nonsensical orthant by (R-ChoiceL) and (R-ChoiceR). Instead, we give an alternative semantics, which is big-step and deterministic, and prove the property via this semantics.

Figure 4.13 and Figure 4.14 define the alternative semantics for $\lambda^{H\|^{\Phi}}$ in a big-step style. Most of the rules are straightforward; the key rules are (N-Choice), (N-WorldL), and (N-WorldR), which correspond to (R-ChoiceL), (R-ChoiceR), (R-WorldL), and (R-WorldR). The point is that (N-Choice) uses disjoint union: both branches of a choice are evaluated only if they are not bogus. The following lemmas guarantee determinism and the two semantics agree with each other.

Lemma 4.3.4 (Well-definedness)**.** *If $M \Downarrow^{\Delta} N$, then N is a result, namely $\Delta \vDash N$.*

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

Lemma 4.3.5 (Determinacy)**.** *If $M \Downarrow^{\Delta} R_1$ and $M \Downarrow^{\Delta} R_2$, then $R_1 = R_2$.*

Proof. The proof is straightforward by the induction on the given derivation of $M \Downarrow^{\Delta} R_1$. □

Lemma 4.3.6 (Normal form)**.**

1. $\bar{n} \Downarrow^{\Delta} \bar{n}$.
2. $V \Downarrow^{\Delta} V$.
3. *If $\Delta \vDash R$, then $R \Downarrow^{\Delta} R$.*

Proof. (1) and (2) can be easily shown by the structural induction on $\bar{n}$ and V, respectively; the proof of (2) uses (1). (3) is shown by the induction on the given derivation. □

Lemma 4.3.7.

1. *If $M \xrightarrow{\Delta}^{*} M'$, then $\mathcal{E}[M] \xrightarrow{\Delta}^{*} \mathcal{E}[M']$.*
2. *If $M \xrightarrow{\Delta\cup\{\Phi_+\}} M'$, then $(M \|^{\Phi} N) \xrightarrow{\Delta}^{*} (M' \|^{\Phi} N)$.*
3. *If $N \xrightarrow{\Delta\cup\{\Phi_+\}} N'$, then $(M \|^{\Phi} N) \xrightarrow{\Delta}^{*} (M \|^{\Phi} N')$.*

Proof. Each can be shown by mathematical induction on the length of the given reduction sequence. □

Lemma 4.3.8. *If $M \xrightarrow{\Delta} N$ and $M \Downarrow^{\Delta} R$, then $N \Downarrow^{\Delta} R$.*

Lemma 4.3.9. *Suppose Δ is normal. If $M \xrightarrow{\Delta} N$ and $N \Downarrow^{\Delta} R$, then $M \Downarrow^{\Delta} R$.*

Lemma 4.3.10. *If $M \Downarrow^{\Delta} R$, then $M \xrightarrow{\Delta}^{*} R$.*

Proof. The proof is done by the induction on the given derivation. The axiom cases follows by the reflexivity of the mult-step reduction. Other cases can be shown by using the transitivity of the mult-step reduction, Lemma 4.3.7, and proper reduction rules. For instance, in the case (N-Pred), we have $M \xrightarrow{\Delta}^{*} \mathtt{succ}(\bar{n})$ by the induction hypothesis. Using transitivity and (R-Pred), we can have the goal as $M \xrightarrow{\Delta}^{*} \mathtt{succ}(\bar{n}) \xrightarrow{\Delta} \bar{n}$. □

Lemma 4.3.11. *If Δ is normal, $M \xrightarrow{\Delta}{}^{*} R$, and $\Delta \models R$, then $M \Downarrow^{\Delta} R$.*

Proof. The proof is done by the length of the mult-step reduction. If the length is zero, we can assume $M = R$. Therfore, the goal $R \Downarrow^{\Delta} R$ follows by Lemma 4.3.6. The length is more than zero, otherwise. We can assume $M \xrightarrow{\Delta} N$ and $N \xrightarrow{\Delta}{}^{*} R$ for some N. Applying the induction hypothesis, we have $N \Downarrow^{\Delta} R$. Now the goal follows by Lemma 4.3.9. □

Now, Lemma 4.3.3 follows from the following properties.

Lemma 4.3.12 (Co-termination)**.** *Suppose $N \xrightarrow{\Delta} N'$ or $N' \xrightarrow{\Delta} N$, and Δ is normal. If $M[x := N] \Downarrow^{\Delta} R$, then there exists R' such that*

- *$\Delta \models R'$;*
- *$R = R'[x := N]$; and*
- *$M[x := N'] \Downarrow^{\Delta} R'[x := N']$.*

Proof. The proof is done by the induction on the given derivation of $M[x := N] \Downarrow^{\Delta} R$. In each case, if $M = x$, the goal can be shown by choosing $R' = R$ and using Lemma 4.3.8 and Lemma 4.3.9. In the following, we assume $M \neq x$. The axiom cases can be shown by choosing $R' = M$ since $M[x := N] = R$ in the cases. Inductive cases follows by the induction hypothesis. We just show the proof for the case (N-Choice) because other cases can be shown in a similar manner. In the case, the last step of the derivation is as follows.

$$\frac{M_1 \Downarrow^{\Delta \uplus \{\Phi_+\}} R_1 \qquad M_2 \Downarrow^{\Delta \uplus \{\Phi_-\}} R_2}{(M_1 \parallel^{\Phi} M_2) \Downarrow^{\Delta} (R_1 \parallel^{\Phi} R_2)}$$

Observing $M[x := N] = (M_1 \parallel^{\Phi} M_2)$, M must be $(M'_1 \parallel^{\Phi} M'_2)$ for some M'_1, M'_2 such that $M'_1[x := N], M'_2[x := N]$. $\Delta \uplus \{\Phi_+\}, \Delta \uplus \{\Phi_-\}$ are normal by the disjunctivity. Now we can have R'_1, R'_2 satisfying the following by the induction hypothesis.

$$\begin{gathered}
\Delta \uplus \{\Phi_+\} \models R'_1 \\
R_1 = R'_1[x := N] \\
M'_1[x := N'] \Downarrow^{\Delta \uplus \{\Phi_+\}} R'_1[x := N'] \\
\Delta \uplus \{\Phi_-\} \models R'_2 \\
R_2 = R'_2[x := N] \\
M'_2[x := N'] \Downarrow^{\Delta \uplus \{\Phi_-\}} R'_2[x := N']
\end{gathered}$$

Now we choose $R' = (R'_1 \parallel^{\Phi} R'_2)$, and then the goal becomes

$$\begin{gathered}
\Delta \models (R'_1 \parallel^{\Phi} R'_2) \\
(R_1 \parallel^{\Phi} R_2) = (R'_1 \parallel^{\Phi} R'_2)[x := N] \\
(M'_1 \parallel^{\Phi} M'_2)[x := N'] \Downarrow^{\Delta} (R'_1 \parallel^{\Phi} R'_2)[x := N].
\end{gathered}$$

Those easily follows from the definitions. □

Corollary 4.3.13 (Co-termination for true)**.** *Suppose Δ is normal and $N \xrightarrow{\Delta} N'$. Then $M[x := N] \xrightarrow{\Delta}{}^{*} \mathtt{True} \iff M[x := N'] \xrightarrow{\Delta}{}^{*} \mathtt{True}$.*

Proof. We only show one direction—if $M[x \coloneqq N] \xrightarrow{\Delta}{}^{*}$ $\texttt{True}$, then $M[x \coloneqq N'] \xrightarrow{\Delta}{}^{*}$ $\texttt{True}$ because the opposite is similar. $M[x \coloneqq N] \Downarrow^{\Delta} \texttt{True}$ by Lemma 4.3.11. We can have R' such that $\texttt{True} = R'[x \coloneqq N], M[x \coloneqq N'] \Downarrow^{\Delta} R'[x \coloneqq N']$ by Lemma 4.3.12. By the former condition, R' must be $\texttt{True}$. Now the goal $M[x \coloneqq N'] \xrightarrow{\Delta}{}^{*} \texttt{True}$ follows by Lemma 4.3.10. □

4.3.2 Type Soundness

We start from miscellaneous lemmas to show type soundness.

Lemma 4.3.14 (Common sub-expression reduction)**.** *If* $N \xrightarrow{\Delta} N'$*, then* $\tau[x \coloneqq N] \overset{\Delta}{\equiv} \tau[x \coloneqq N']$*.*

Proof. The proof is routine by structural induction on τ. □

Lemma 4.3.15 (Typing is closed)**.**

1. *If* $\Gamma \Vdash^{\Delta} \tau$*, then* $\mathit{fv}(\tau) \subseteq \mathit{dom}(\Gamma)$ *and* $\mathit{fsv}(\tau) \subseteq \mathit{sdom}(\Gamma)$*.*
2. *If* $\Gamma \vdash^{\Delta} M : \tau$*, then* $\mathit{fv}(M) \cup \mathit{fv}(\tau) \subseteq \mathit{dom}(\Gamma)$ *and* $\mathit{fsv}(M) \cup \mathit{fsv}(\tau) \subseteq \mathit{sdom}(\Gamma)$*.*

Lemma 4.3.16 (Generalized weakening)**.** *Suppose* $\Gamma \Vdash^{\Delta} \sigma$ *and* $x \notin \mathit{dom}(\Gamma) \cup \mathit{dom}(\Gamma')$*.*

1. *If* $\Gamma, \Gamma' \Vdash^{\Delta} \tau$*, then* $\Gamma, x{:}\sigma, \Gamma' \Vdash^{\Delta} \tau$*.*
2. *If* $\Gamma, \Gamma' \vdash^{\Delta} M : \tau$*, then* $\Gamma, x{:}\sigma, \Gamma' \vdash^{\Delta} M : \tau$*.*

Lemma 4.3.17 (Generalized string weakening)**.** *Suppose* $\alpha \notin \mathit{sdom}(\Gamma) \cup \mathit{sdom}(\Gamma')$*.*

1. *If* $\Gamma, \Gamma' \Vdash^{\Delta} \tau$*, then* $\Gamma, \alpha, \Gamma' \Vdash^{\Delta} \tau$*.*
2. *If* $\Gamma, \Gamma' \vdash^{\Delta} M : \tau$*, then* $\Gamma, \alpha, \Gamma' \vdash^{\Delta} M : \tau$*.*

Lemma 4.3.18 (Orthant substitution)**.**

1. *If* $M \xrightarrow{\Delta} N$*, then* $M \xrightarrow{\Delta[\alpha \coloneqq \Phi]} N$*.*
2. *If* $M \xrightarrow{\Delta}{}^{*} N$*, then* $M \xrightarrow{\Delta[\alpha \coloneqq \Phi]} N$*.*
3. *If* $\sigma \overset{\Delta}{\equiv} \tau$*, then* $\sigma \overset{\Delta[\alpha \coloneqq \Phi]}{\equiv} \tau$*.*
4. *If* $\Gamma \Vdash^{\Delta} \tau$ *and* $\alpha \notin \mathit{sdom}(\Gamma)$*, then* $\Gamma \Vdash^{\Delta[\alpha \coloneqq \Phi]} \tau$*.*
5. *If* $\Gamma \vdash^{\Delta} M : \tau$ *and* $\alpha \notin \mathit{sdom}(\Gamma)$*, then* $\Gamma \vdash^{\Delta[\alpha \coloneqq \Phi]} M : \tau$*.*

Proof of (1). The proof is by induction on the given derivation. □

Proof of (2) and (3). These are corollary of (1). □

Proof of (4) and (5). □

Lemma 4.3.19 (Substitution)**.**

1. *If* $\sigma \overset{\Delta}{\equiv} \tau$*, then* $\sigma[x := N] \overset{\Delta}{\equiv} \tau[x := N]$.
2. *If* $\sigma \sim \tau$*, then* $\sigma[x := N] \sim \tau[x := N]$.
3. *If* $\Gamma, x{:}\sigma, \Gamma' \Vdash^{\Delta} \tau$ *and* $\Gamma \vdash^{\Delta} N : \sigma$*, then* $\Gamma, \Gamma'[x := N] \Vdash^{\Delta} \tau[x := N]$.
4. *If* $\Gamma, x{:}\sigma, \Gamma' \vdash^{\Delta} M : \tau$ *and* $\Gamma \vdash^{\Delta} N : \sigma$*, then* $\Gamma, \Gamma'[x := N] \vdash^{\Delta} M[x := N] : \tau[x := N]$.

Lemma 4.3.20 (String substitution)**.**

1. *If* $\sigma \overset{\Delta}{\equiv} \tau$*, then* $\sigma[\alpha := \Phi] \overset{\Delta}{\equiv} \tau[\alpha := \Phi]$.
2. *If* $\sigma \sim \tau$*, then* $\sigma[\alpha := \Phi] \sim \tau[\alpha := \Phi]$.
3. *If* $\Gamma, \alpha, \Gamma' \Vdash^{\Delta} \tau$ *and* $\mathit{fsv}(\Phi) \subseteq \mathit{sdom}(\Gamma)$*, then* $\Gamma, \Gamma'[\alpha := \Phi] \Vdash^{\Delta[\alpha:=\Phi]} \tau[\alpha := \Phi]$.
4. *If* $\Gamma, \alpha, \Gamma' \vdash^{\Delta} M : \tau$ *and* $\mathit{fsv}(\Phi) \subseteq \mathit{sdom}(\Gamma)$*, then* $\Gamma, \Gamma'[\alpha := \Phi] \vdash^{\Delta} M[\alpha := \Phi] : \tau[\alpha := \Phi]$.

Definition 4.3.21 (Well-formed environments)**.** The following rules define well-formedness of environments in explicit style like $\mathrm{PCF_H}$.

$$\frac{}{\Gamma\,\mathrm{ok}^{\Delta}} \qquad \frac{\Gamma\,\mathrm{ok}^{\Delta} \qquad \Gamma \Vdash^{\Delta} \tau \qquad (x \notin \mathtt{dom}(\Gamma))}{\Gamma, x{:}\tau\,\mathrm{ok}^{\Delta}} \qquad \frac{\Gamma\,\mathrm{ok}^{\Delta} \qquad (\alpha \notin \mathtt{sdom}(\Gamma))}{\Gamma, \alpha\,\mathrm{ok}^{\Delta}}$$

Lemma 4.3.22 (Presupposition)**.**

1. *If* $\Gamma \Vdash^{\Delta} \tau$*, then* Γ ok^{Δ}.
2. *If* $\Gamma \vdash^{\Delta} M : \tau$*, then* Γ ok^{Δ}.
3. *If* $\Gamma \vdash^{\Delta} M : \tau$*, then* $\Gamma \Vdash^{\Delta} \tau$.

Now we can show subject reduction property.

Lemma 4.3.23 (Subject reduction)**.** *If* $\vdash^{\Delta} M : \tau$ *and* $M \overset{\Delta}{\longrightarrow} N$*, then* $\vdash^{\Delta} N : \tau$.

Next, we show a value inversion lemma and then the contract satisfaction lemma by lifting the former result pointwise.

Definition 4.3.24 (Refinements)**.** The set of expressions refining τ, written $\mathtt{refines}(\tau)$, is defined as follows.

$$\mathtt{refines}(\mathtt{bool}) = \mathtt{refines}(\mathtt{nat}) = \mathtt{refines}((x{:}\sigma) \to \tau) = \{\}$$
$$\mathtt{refines}(\{x{:}\tau \mid M\}) = \{(x)M\} \cup \mathtt{refines}(\tau)$$

Lemma 4.3.25 (Value inversion)**.** *If* $\vdash^{\Delta} V : \tau$ *and* Δ *is normal, then* $M[x := V] \overset{\Delta}{\longrightarrow}{}^{*}$ True *for all* $(x)M \in \mathit{refines}(\tau)$*.*

Definition 4.3.26. *Result* R *satisfies refinement type* $\{x{:}\tau \mid M\}$ *in orthant* Δ, written $R \vDash^{\Delta} \{x{:}\tau \mid M\}$, if the judgment is derived by the following rules.

$$\frac{M[x := V] \overset{\Delta}{\longrightarrow}{}^{*} \mathtt{True}}{V \vDash^{\Delta} \{x{:}\tau \mid M\}} \qquad \mathtt{blame} \vDash^{\Delta} \{x{:}\tau \mid M\}$$

$$\frac{R_1 \vDash^{\Delta \cup \{\Phi_+\}} \{x{:}\tau \mid M\} \qquad R_2 \vDash^{\Delta \cup \{\Phi_-\}} \{x{:}\tau \mid M\}}{(R_1 \parallel^{\Phi} R_2) \vDash^{\Delta} \{x{:}\tau \mid M\}}$$

Lemma 4.3.27 (Contract satisfaction)**.** *If* Δ *is normal and* $\vdash^{\Delta} R : \{x{:}\tau \mid M\}$*, then* $R \models^{\Delta} \{x{:}\tau \mid M\}$*.*

Lastly, we show the usual progress property. The proof depends on Lemma 4.3.27 since our $\texttt{pred}(M)$ is a partial operation.

Lemma 4.3.28 (Progress)**.** *If* Δ *is normal and* $\vdash^{\Delta} M : \tau$*, then* M *is a result in* Δ*; or there exists* N *such that* $M \xrightarrow{\Delta} N$*.*

We close the section by combining the results above into the following theorem.

Theorem 4.3.29 (Soundness)**.** *If* $\vdash^{\{\}} M : \tau$*, then (1)* $M \xrightarrow{\{\}}{}^{*} R$ *and* $\{\} \models R$*; or (2)* M *diverges. Moreover, if* τ *is a refinement type* $\{x{:}\sigma \mid N\}$ *and (1) holds,* $R \models^{\{\}} \{x{:}\sigma \mid N\}$*.*

4.4 Summary

We develop $\lambda^{H\|^{\Phi}}$, a nondeterministic manifest contract calculus. By introducing a new kind of choice called coordinated choice, the calculus is free from restriction in the kind of programs used in predicates in refinements types. Coordinated choice can share nondeterministic decisions by using names and the semantics is given in such a way that a program reduces to the set (expressed by $\|$) of all possible results. The semantics makes a choice copyable by substitution and solves problems that occur when we introduce dependent function types. The semantics and type system of $\lambda^{H\|^{\Phi}}$ are given by using the characteristic concept called orthant. Orthant gives the view for a choice. Since the view does not change the meaning of expressions except choices, we expect that coordinated choice is quite easily integrated into other manifest contract systems. By the observation, we hope this approach is easily applied to any other languages.

5 Compilation of Coordinated Choice

In the previous chapters, we have seen two manifest contract systems: PCFvΔ_{H} with intersection types but no dependent function types because of nondeterministic semantics; and $\lambda^{H\|^{\Phi}}$ with nondeterminism, but careful programming is required to use coordinated choice.

In this chapter, we consider how to simulate usual choices with coordinated choices. For the purpose, we give a compilation algorithm from a simply typed lambda calculus with (usual) nondeterministic choice into one with coordinated choice; and show an evaluation in the former calculus is simulated by the latter calculus, and vice versa. The idea of compilation has been seen in Section 4.1.3—give different names to each choice; abstract names in lambda abstractions; and instantiate the abstracted names by fresh names at applications.

5.1 Compilation

In this section, we introduce the compilation rules and discuss the correctness of the given rules. First of all, we recall how a coordinated choice is synchronized. The following is an example of an evaluation in which synchronization happens (reprint of Example 4.1.2).

$$
\begin{aligned}
(1 \|^{\Phi} 2) + (3 \|^{\Phi} 4) &\overset{\emptyset}{\longrightarrow}{}^{*} (1 + (3 \|^{\Phi} 4) \|^{\Phi} 2 + (3 \|^{\Phi} 4)) \\
&\overset{\emptyset}{\longrightarrow}{}^{*} (1 + 3 \|^{\Phi} 2 + 4) \\
&\overset{\emptyset}{\longrightarrow}{}^{*} (4 \|^{\Phi} 6)
\end{aligned}
$$

The devices that enable synchronization are the following rules, which are used in the evaluation from the first line to the second line.

$$
\frac{M_1 \overset{\Delta \cup \{\Phi_+\}}{\longrightarrow} M_1'}{(M_1 \|^{\Phi} M_2) \overset{\Delta}{\longrightarrow} (M_1' \|^{\Phi} M_2)}
\qquad
\frac{}{(M_1 \|^{\Phi} M_2) \overset{\Delta \cup \{\Phi_+\}}{\longrightarrow} M_1}
$$

Concentrating on one step of an evaluation, we can easily find that a sufficient condition to prevent synchronization is that every name of choices in an expression before the evaluation is distinct. However, this condition is easily broken by evaluation since some reduction rules, e.g., $(\lambda x.M)\, V \overset{\Delta}{\longrightarrow} M[x \coloneqq V]$, duplicate an expression. In other words, it is easy to prevent synchronization in the first step of an evaluation, but it is hard to prevent synchronization in every step after the first step of an evaluation.

Fortunately, thanks to call-by-value semantics, names that will be duplicated and become a cause of synchronization only occur in the body of lambda abstractions. So we abstract names (giving a name containing variables) occurring in the body of lambda abstractions and instantiate the names as distinct ones where the lambda abstractions are called. The

$$
\begin{aligned}
[\![x]\!]^{\alpha}_{\Phi} &= x \\
[\![e_1\, e_2]\!]^{\alpha}_{\Phi} &= [\![e_1]\!]^{\alpha}_{\Phi\circ\circ}\, [\![e_2]\!]^{\alpha}_{\Phi\circ\bullet}\, \alpha\bar{\Phi}\bullet \\
[\![\lambda x.e]\!]^{\alpha}_{\Phi} &= \lambda x.\Lambda\beta.[\![e]\!]^{\beta}_{\Phi} \\
[\![\mu f.e]\!]^{\alpha}_{\Phi} &= \mu f.[\![e]\!]^{\alpha}_{\Phi} \\
[\![(e_1 \parallel e_2)]\!]^{\alpha}_{\Phi} &= ([\![e_1]\!]^{\alpha}_{\Phi\circ} \parallel^{\alpha\bar{\Phi}\bullet} [\![e_2]\!]^{\alpha}_{\Phi\circ})
\end{aligned}
$$

Figure 5.1: Compilation of expressions

following two evaluation sequences show the effect of this idea.

$$
\begin{aligned}
(\mu f.\lambda x.(f\, x \parallel^{\Phi} f\, x))\, 0 &\xrightarrow{\varnothing}{}^{*} (f'\, 0 \parallel^{\Phi} f'\, 0) \\
&\xrightarrow{\varnothing}{}^{*} ((f'\, 0 \parallel^{\Phi} f'\, 0) \parallel^{\Phi} (f'\, 0 \parallel^{\Phi} f'\, 0)),
\end{aligned}
$$

where $f' = \mu f.\lambda x.(f\, x \parallel^{\Phi} f\, x)$.

$$
\begin{aligned}
&(\mu f.\lambda x.\Lambda\alpha.(f\, x\, \alpha\circ \parallel^{\alpha} f\, x\, \alpha\bullet))\, 0\, \circ \\
\xrightarrow{\varnothing}{}^{*}&(f'\, 0\, \circ\circ \parallel^{\circ} f'\, \circ\bullet) \\
\xrightarrow{\varnothing}{}^{*}&((f'\, 0\, \circ\circ\circ \parallel^{\circ\circ} f'\, 0\, \circ\circ\bullet) \parallel^{\circ} (f'\, 0\, \circ\bullet\circ \parallel^{\circ\bullet} f'\, 0\, \circ\bullet\bullet)),
\end{aligned}
$$

where $f' = \mu f.\lambda x.\Lambda\alpha.(f\, x\, \alpha\circ \parallel^{\alpha} f\, x\, \alpha\bullet)$.

In the first sequence, the duplicated name Φ causes synchronization. In the second sequence, on the other hand, the name of a coordinated choice becomes different one at every time that a lambda abstraction is applied (so synchronization does not happen).

Summing up the discussion, concrete compilation rules become the ones in Figure 5.1. The function $[\![\cdot]\!]^{\alpha}_{\omega}$ gives the compilation, where α and ω are seeds generating distinct names during a compilation. Note that the seeds given for the sub-expressions of a choice are used the same ones. It does not cause a problem because the left and right sides of a choice are completely separated, i.e., the sub-expressions never collaborate.

5.1.1 Correctness of compilation

The main contribution of this chapter is that we have formally shown that the compilation is *correct*. Informally, correctness of a compilation is stated as—a compiled expression behaves the same as the original expression does. This behavioral correspondence between compiled expressions and original expressions is formally defined as a binary relation, called *bisimulation* [40], between them. For instance, if we use the compilation rules as the relation, what we need to show is stated as the following two propositions.

Proposition 5.1.1. *If* $[\![e]\!]^{\alpha}_{\Phi} = M$ *and* $e \longrightarrow e'$, *then* $M \xrightarrow{\Delta}{}^{*} M'$ *and* $[\![e']\!]^{\alpha}_{\Phi} = M'$.

Proposition 5.1.2. *If* $[\![e]\!]^{\alpha}_{\Phi} = M$ *and* $M \xrightarrow{\Delta} M'$, *then* $e \longrightarrow^{*} e'$ *and* $[\![e']\!]^{\alpha}_{\Phi} = M'$.

$$
\begin{array}{rcl}
T & ::= & \text{nat} \mid T_1 \to T_2 \\
e & ::= & x \mid e_1\, e_2 \mid \lambda x.e \mid \mu f.e \mid (e_1 \parallel e_2) \\
\Gamma & ::= & \varnothing \mid \Gamma, x{:}T \\
v & ::= & \lambda x.e
\end{array}
$$

Figure 5.2: Syntax of $\lambda^{\parallel}$

However, we can easily find a counter-example for the reduction $(\lambda x.x)\,\lambda x.x \longrightarrow \lambda x.x$ as follows.

$$
\begin{aligned}
[\![(\lambda x.x)\,\lambda x.x]\!]^{\alpha}_{\bar{\Phi}} &= [\![\lambda x.x]\!]^{\alpha}_{\bar{\Phi}\circ\circ}\,[\![\lambda x.x]\!]^{\alpha}_{\bar{\Phi}\circ\bullet}\,\alpha\bar{\Phi}\bullet \\
&= (\lambda x.\Lambda\beta.x)\,[\![\lambda x.x]\!]^{\alpha}_{\bar{\Phi}\circ\bullet}\,\alpha\bar{\Phi}\bullet \xrightarrow{\Delta} [\![\lambda x.x]\!]^{\alpha}_{\bar{\Phi}\circ\bullet}
\end{aligned}
$$

A seed is alternated by the reduction. Unfortunately, weakening the proposition as follows does not help us, because, if a sub-expression (for example, the argument part of an application) is evaluated, only the seed for the sub-expression part is alternated and the whole expression after the evaluation does not follow the compilation rule.

Proposition 5.1.3. *If* $[\![e]\!]^{\alpha}_{\bar{\Phi}} = M$ *and* $e \longrightarrow e'$*, then* $M \xrightarrow{\Delta} M'$ *and* $[\![e']\!]^{\alpha}_{\bar{\Phi}'} = M'$ *for some* $\bar{\Phi}'$*.*

Ultimately, we believe that the relation cannot be obtained directly by a compilation algorithm. That means it is not because of how the compilation rules are defined. To prevent coordination of coordinated choice, we need to give distinct names for each coordinated choice, which is a sub-expression, but as we have seen in the counter-example, a sub-expression could pop-up to the top level, which leads us to an inconsistency.

Summarizing the discussion, we could see that coordination does not happen in the first step of the evaluation for a complied expression; but we cannot guarantee the property after the first step because an expression can have no relation to the compilation rules. So, we take another indirect strategy. Firstly, we define a relation characterizing expressions that do not cause coordination and is preserved by an evaluation, and then we show the image of the compilation is included in the relation. After accomplishing this, we can easily obtain a bisimulation relation since the compilation just inserts name abstractions and applications—an evaluation after compilation just involves (R-Sigma) reduction in some points.

5.2 Formal System

We formalize the idea as $\lambda^{\parallel}$, a simply typed lambda calculus with a fix-point operator and *non-collapse* choices, and $\lambda^{\parallel\omega}$, a simply typed lambda calculus with a fix-point operator and coordinated choice; and give a compilation rule from the former to the latter.

5.2.1 Source Language: $\lambda^{\parallel}$

The syntax, semantics, and type system of $\lambda^{\parallel\omega}$ are defined as Figure 5.2, Figure 5.3, and Figure 5.4, respectively. Those are standard ones for a simply typed lambda calculus and with straightforward extensions for a fix-point operator $\mu f.e$ and non-collapse choices $(e_1 \parallel e_2)$.

$$\frac{}{(\lambda x.e)\, v \longrightarrow e[x \coloneqq v]} \quad \text{(SR-Beta)}$$

$$\frac{}{\mu f.\lambda x.e \longrightarrow (\lambda x.e)[f \coloneqq \mu f.\lambda x.e]} \quad \text{(SR-Fix)}$$

$$\frac{}{(e_{11} \parallel e_{12})\, e_2 \longrightarrow (e_{11}\, e_2 \parallel e_{12}\, e_2)} \quad \text{(SR-DistAppL)}$$

$$\frac{}{v\,(e_{21} \parallel e_{22}) \longrightarrow (v\, e_{21} \parallel v\, e_{22})} \quad \text{(SR-DistAppR)}$$

$$\frac{e_1 \longrightarrow e_1'}{e_1\, e_2 \longrightarrow e_1'\, e_2} \quad \text{(SR-AppL)}$$

$$\frac{e_2 \longrightarrow e_2'}{v\, e_2 \longrightarrow v\, e_2'} \quad \text{(SR-AppR)}$$

$$\frac{e_1 \longrightarrow e_1'}{(e_1 \parallel e_2) \longrightarrow (e_1' \parallel e_2)} \quad \text{(SR-ChoiceL)}$$

$$\frac{e_2 \longrightarrow e_2'}{(e_1 \parallel e_2) \longrightarrow (e_1 \parallel e_2')} \quad \text{(SR-ChoiceR)}$$

Figure 5.3: Operational semantics of $\lambda^{\parallel}$

$$\frac{}{\varnothing \text{ ok}} \quad \text{(SW-Empty)}$$

$$\frac{\Gamma \text{ ok} \qquad (x \notin \text{dom}(\Gamma))}{\Gamma, x\!:\!T \text{ ok}} \quad \text{(SW-Push)}$$

$$\frac{\Gamma \text{ ok} \qquad (x\!:\!T \in \Gamma)}{\Gamma \vdash x : T} \quad \text{(ST-Var)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \to T_2 \qquad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1\, e_2 : T_2} \quad \text{(ST-App)}$$

$$\frac{\Gamma, x\!:\!T_1 \vdash e : T_2}{\Gamma \vdash \lambda x.e : T_1 \to T_2} \quad \text{(ST-Abs)}$$

$$\frac{\Gamma, f\!:\!T_1 \to T_2 \vdash \lambda x.e : T_1 \to T_2}{\Gamma \vdash \mu f.\lambda x.e : T_1 \to T_2} \quad \text{(ST-Fix)}$$

$$\frac{\Gamma \vdash e_1 : T \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash (e_1 \parallel e_2) : T} \quad \text{(ST-Choice)}$$

Figure 5.4: Typing rules for $\lambda^{\parallel}$

$$\begin{array}{rcl}
\Phi & ::= & \alpha \mid \epsilon \mid \bigcirc \mid \bullet \mid \Phi_1\Phi_2 \\
\varphi & ::= & \varnothing \mid \Phi \mid \varphi_1\varphi_2 \mid \varphi_1 + \varphi_2 \mid \varphi^* \\
\tau & ::= & \mathrm{nat} \mid \tau_1 \xrightarrow{\varphi} \tau_2 \mid \forall\alpha.^{\varphi}\tau \\
M & ::= & x \mid M_1\, M_2 \mid \lambda x.M \mid M\,\Phi \mid \Lambda\alpha.M \mid \mu f.M \mid (M_1 \parallel^{\Phi} M_2) \\
\Xi & ::= & \varnothing \mid \Xi, x : \tau \mid \Xi, \alpha \\
V & ::= & \lambda x.M \mid \Lambda\alpha.M
\end{array}$$

Figure 5.5: Syntax of $\lambda^{\parallel\omega}$

“Non-collapse” means that a choice dose not really choose an alternative (like $(e_1 \parallel e_2) \longrightarrow e_1$) as coordinated choice does not, but also it never coordinate alternatives.

5.2.2 Target Language: $\lambda^{\parallel\omega}$

The syntax and semantics of $\lambda^{\parallel\omega}$ are shown in Figure 5.5 and Figure 5.6, respectively. As far as the semantics, $\lambda^{\parallel\omega}$ is just a simply typed lambda calculus with coordinated choice, c.f. $\lambda^{H\parallel^{\Phi}}$. An interesting part of $\lambda^{\parallel\omega}$ is an *effect* system [42], which is a kind of type system. The effect system of $\lambda^{\parallel\omega}$ estimates names happening during execution and rejects an expression which will causes synchronization, and so this is the relation that we have mentioned at the last of Section 5.1.

We range over effects with a meta-variable φ. Effects of $\lambda^{\parallel\omega}$ are denoted by regular expressions: the empty set $\varnothing$; literal strings Φ including the empty string ϵ; concatenations $\varphi_1\varphi_2$; alternations $\varphi_1 + \varphi_2$; and Kleene star φ^*.

Function types and forall types are annotated by effects, which express the effect happening when a function is used. Forall types $\forall\alpha.^{\varphi}\tau$ bind α in φ and τ.

Definition 5.2.1. A *closed* term is one in which both free variables and free name variables are empty. We denote closed terms with over-barred meta-variables, e.g., $\bar{\Phi}, \bar{\varphi}$.

Definition 5.2.2. We denote the language, a set of names, represented by an effect φ as $\mathbb{L}(\varphi)$.

5.2.3 Effect System

The effect system for $\lambda^{\parallel\omega}$ consists of the subtyping relation $\tau_1 <: \tau_2$; the well-formedness relations Ξ ok, $\Xi \Vdash \Phi$, $\Xi \Vdash \varphi$, and $\Xi \Vdash \tau$; and typing relation $\Xi \vdash M : \tau \,\&\, \varphi$.

Well-formedness rules just check a closedness of types by (TW-NVar) and (TW-Forall).

Typing rules are the heart of the effect system. The judgment $\Xi \vdash M : \tau \,\&\, \varphi$ denotes an usual typing relation by the left side of & and what names will occur during an evaluation of M by the right side of &. So, ignoring the right side of &, rules are already familiar. For the effect part, the rules becomes complicated because of showing meta-properties.

(TT-Var), (TT-Abs), (TT-SAbs), and (TT-Fix) are rather easy. Corresponding expressions for those are never evaluated (fix-point operator is evaluated but soon reaches a value); and therefore, those produce no names examined. This is why the effect part of those rules becomes empty. Note that functions' body is evaluated when an argument is given; so the effect of the body is recorded on types and added at an application point, c.f. (TT-Abs) and (TT-App).

$$\frac{}{(\lambda x.M)\,V \xrightarrow{\Delta} M[x \coloneqq V]} \quad \text{(TR-Beta)}$$

$$\frac{}{(\Lambda\alpha.M)\,\Phi \xrightarrow{\Delta} M[\alpha \coloneqq \Phi]} \quad \text{(TR-Sigma)}$$

$$\frac{}{\mu f.\lambda x.M \xrightarrow{\Delta} (\lambda x.M)[f \coloneqq \mu f.\lambda x.M]} \quad \text{(TR-Fix)}$$

$$\frac{}{(M_1 \parallel^{\Phi} M_2)\,M_3 \xrightarrow{\Delta} (M_1\,M_3 \parallel^{\Phi} M_2\,M_3)} \quad \text{(TR-DistAppL)}$$

$$\frac{}{V(M_1 \parallel^{\Phi} M_2) \xrightarrow{\Delta} (VM_1 \parallel^{\Phi} VM_2)} \quad \text{(TR-DistAppR)}$$

$$\frac{}{(M_1 \parallel^{\Phi_1} M_2)\,\Phi_2 \xrightarrow{\Delta} (M_1\,\Phi_2 \parallel^{\Phi_1} M_2\,\Phi_2)} \quad \text{(TR-DistSApp)}$$

$$\frac{M_1 \xrightarrow{\Delta} M_1'}{M_1\,M_2 \xrightarrow{\Delta} M_1'\,M_2} \quad \text{(TR-AppL)}$$

$$\frac{M \xrightarrow{\Delta} M'}{VM \xrightarrow{\Delta} VM'} \quad \text{(TR-AppR)}$$

$$\frac{M \xrightarrow{\Delta} M'}{M\,\Phi \xrightarrow{\Delta} M'\,\Phi} \quad \text{(TR-SApp)}$$

$$\frac{M_1 \xrightarrow{\Delta \cup \{\Phi_+\}} M_1'}{(M_1 \parallel^{\Phi} M_2) \xrightarrow{\Delta} (M_1' \parallel^{\Phi} M_2)} \quad \text{(TR-ChoiceL)}$$

$$\frac{M_2 \xrightarrow{\Delta \cup \{\Phi_-\}} M_2'}{(M_1 \parallel^{\Phi} M_2) \xrightarrow{\Delta} (M_1 \parallel^{\Phi} M_2')} \quad \text{(TR-ChoiceR)}$$

$$\frac{}{(M_1 \parallel^{\Phi} M_2) \xrightarrow{\Delta \cup \{\Phi_+\}} M_1} \quad \text{(TR-WorldL)}$$

$$\frac{}{(M_1 \parallel^{\Phi} M_2) \xrightarrow{\Delta \cup \{\Phi_-\}} M_2} \quad \text{(TR-WorldR)}$$

Figure 5.6: Operational semantics of $\lambda^{\parallel\omega}$

$$\frac{}{\mathrm{nat} <: \mathrm{nat}} \quad \text{(TS-Nat)}$$

$$\frac{\tau_{21} <: \tau_{11} \qquad \tau_{12} <: \tau_{22} \qquad (\mathbb{L}(\varphi_1) \subseteq \mathbb{L}(\varphi_2))}{\tau_{11} \xrightarrow{\varphi_1} \tau_{12} <: \tau_{21} \xrightarrow{\varphi_2} \tau_{22}} \quad \text{(TS-Arrow)}$$

$$\frac{\tau_1 <: \tau_2 \qquad (\mathbb{L}(\varphi_1) \subseteq \mathbb{L}(\varphi_2))}{\forall\alpha.^{\varphi_1}\tau_1 <: \forall\alpha.^{\varphi_2}\tau_2} \quad \text{(TS-Forall)}$$

Figure 5.7: Effect system of $\lambda^{\parallel\omega}$ (1): subtyping rules

$$\frac{}{\varnothing\ \mathrm{ok}} \quad \text{(TW-Empty)}$$

$$\frac{\Xi\ \mathrm{ok} \qquad \Xi \Vdash \tau \qquad (x \notin \mathrm{dom}(\Xi))}{\Xi, x : \tau\ \mathrm{ok}} \quad \text{(TW-Push)}$$

$$\frac{\Xi\ \mathrm{ok} \qquad (\alpha \notin \mathrm{ndom}(\Xi))}{\Xi, \alpha\ \mathrm{ok}} \quad \text{(TW-PushS)}$$

$$\frac{\Xi\ \mathrm{ok} \qquad (\alpha \in \Xi)}{\Xi \Vdash \alpha} \quad \text{(TW-NVar)}$$

$$\frac{\Xi\ \mathrm{ok}}{\Xi \Vdash \epsilon} \quad \text{(TW-Eps)}$$

$$\frac{\Xi\ \mathrm{ok}}{\Xi \Vdash \bigcirc} \quad \text{(TW-On)}$$

$$\frac{\Xi\ \mathrm{ok}}{\Xi \Vdash \bullet} \quad \text{(TW-Off)}$$

$$\frac{\Xi \Vdash \Phi_1 \qquad \Xi \Vdash \Phi_2}{\Xi \Vdash \Phi_1\Phi_2} \quad \text{(TW-Append)}$$

$$\frac{\Xi\ \mathrm{ok}}{\Xi \Vdash \varnothing} \quad \text{(TW-Empty)}$$

$$\frac{\Xi \Vdash \varphi_1 \qquad \Xi \Vdash \varphi_2}{\Xi \Vdash \varphi_1\varphi_2} \quad \text{(TW-Dot)}$$

$$\frac{\Xi \Vdash \varphi_1 \qquad \Xi \Vdash \varphi_2}{\Xi \Vdash \varphi_1 + \varphi_2} \quad \text{(TW-Plus)}$$

$$\frac{\Xi \Vdash \varphi}{\Xi \Vdash \varphi^*} \quad \text{(TW-Star)}$$

$$\frac{\Xi\ \mathrm{ok}}{\Xi \Vdash \mathrm{nat}} \quad \text{(TW-Nat)}$$

$$\frac{\Xi \Vdash \tau_1 \qquad \Xi \Vdash \tau_2 \qquad \Xi \Vdash \varphi}{\Xi \Vdash \tau_1 \xrightarrow{\varphi} \tau_2} \quad \text{(TW-Arrow)}$$

$$\frac{\Xi, \alpha \Vdash \tau \qquad \Xi, \alpha \Vdash \varphi}{\Xi \Vdash \forall \alpha.^{\varphi} \tau} \quad \text{(TW-Forall)}$$

Figure 5.8: Effect system of $\lambda^{\|\omega}$ (2): well-formedness rules

$$\frac{\Xi\ \mathrm{ok} \qquad (x : \tau \in \Xi)}{\Xi \vdash x : \tau \mathbin{\&} \emptyset} \quad \text{(TT-Var)}$$

$$\frac{\Xi \vdash M_1 : \tau_1 \xrightarrow{\Phi\bar{\varphi}_1} \tau_2 \mathbin{\&} \Phi\bar{\varphi}_2 \qquad \Xi \vdash M_2 : \tau_1 \mathbin{\&} \Phi\bar{\varphi}_3 \qquad (\biguplus_{k \in \{1,2,3\}} \mathbb{L}(\bar{\varphi}_k))}{\Xi \vdash M_1\, M_2 : \tau \mathbin{\&} \Phi(\bar{\varphi}_1 + \bar{\varphi}_2 + \bar{\varphi}_3)} \quad \text{(TT-App)}$$

$$\frac{\Xi, x : \tau_1 \vdash M : \tau_2 \mathbin{\&} \varphi}{\Xi \vdash \lambda x.M : \tau_1 \xrightarrow{\varphi} \tau_2 \mathbin{\&} \emptyset} \quad \text{(TT-Abs)}$$

$$\frac{\Xi \vdash M : \forall\alpha.^{\varphi_1}\tau \mathbin{\&} \Phi'\bar{\varphi}_2 \qquad \Xi \Vdash \Phi \qquad (\mathbb{L}(\varphi_1[\alpha \coloneqq \Phi]) \subseteq \mathbb{L}(\Phi'\bar{\varphi}_3)) \qquad (\mathbb{L}(\bar{\varphi}_2) \uplus \mathbb{L}(\bar{\varphi}_3))}{\Xi \vdash M\,\Phi : \tau[\alpha \coloneqq \Phi] \mathbin{\&} \Phi'(\bar{\varphi}_2 + \bar{\varphi}_3)} \quad \text{(TT-SApp)}$$

$$\frac{\Xi, \alpha \vdash M : \tau \mathbin{\&} \varphi}{\Xi \vdash \Lambda\alpha.M : \forall\alpha.^{\varphi}\tau \mathbin{\&} \emptyset} \quad \text{(TT-SAbs)}$$

$$\frac{\Xi, f : \tau_1 \xrightarrow{\varphi} \tau_2 \vdash \lambda x.M : \tau_1 \xrightarrow{\varphi} \tau_2 \mathbin{\&} \emptyset}{\Xi \vdash \mu f.\lambda x.M : \tau_1 \xrightarrow{\varphi} \tau_2 \mathbin{\&} \emptyset} \quad \text{(TT-Fix)}$$

$$\frac{\Xi \vdash M_1 : \tau \mathbin{\&} \Phi'\bar{\varphi}_1 \qquad \Xi \vdash M_2 : \tau \mathbin{\&} \Phi'\bar{\varphi}_1 \qquad \Xi \Vdash \Phi \qquad (\mathbb{L}(\Phi) \subseteq \mathbb{L}(\Phi'\bar{\varphi}_2)) \qquad (\mathbb{L}(\bar{\varphi}_1) \uplus \mathbb{L}(\bar{\varphi}_2))}{\Xi \vdash (M_1 \parallel^{\Phi} M_2) : \tau \mathbin{\&} \Phi'(\bar{\varphi}_1 + \bar{\varphi}_2)} \quad \text{(TT-Choice)}$$

$$\frac{\Xi \vdash M : \tau_1 \mathbin{\&} \varphi_1 \qquad \Xi \Vdash \tau_2 \qquad \Xi \Vdash \varphi_2 \qquad \tau_1 <: \tau_2 \qquad (\mathbb{L}(\varphi_1) \subseteq \mathbb{L}(\varphi_2))}{\Xi \vdash M : \tau_2 \mathbin{\&} \varphi_2} \quad \text{(TT-Sub)}$$

Figure 5.9: Effect system of $\lambda^{\parallel\omega}$ (3): typing rules

(TT-App), (TT-SApp), and (TT-Choice) have complicated side-conditions. To explain the rules, we will starts from the following rather ideal and simple rule for coordinate choices.

$$\frac{\Xi \vdash M_1 : \tau \mathbin{\&} \varphi_1 \qquad \Xi \vdash M_2 : \tau \mathbin{\&} \varphi_2 \qquad \Xi \Vdash \Phi}{\Xi \vdash (M_1 \parallel^{\Phi} M_2) : \tau \mathbin{\&} \varphi_1 + \varphi_2 + \Phi}$$

This rule just estimate the names which occur along an evaluation and impose no restriction. M_1 will cause names represented by the regular expression φ_1, M_2 will cause names represented by the regular expression φ_2, and the choice itself already causes the name Φ. So the whole effect is the alternation of those.

The next step is to introduce a restriction to the rule. The idea of restriction is so simple—coordination never happens if names of sub-expressions do not overlap. So the rule will become the following.

$$\frac{\Xi \vdash M_1 : \tau \mathbin{\&} \varphi_1 \qquad \Xi \vdash M_2 : \tau \mathbin{\&} \varphi_2 \qquad \Xi \Vdash \Phi \qquad (\mathbb{L}(\varphi_1) \uplus \mathbb{L}(\varphi_2) \uplus \mathbb{L}(\Phi))}{\Xi \vdash (M_1 \parallel^{\Phi} M_2) : \tau \mathbin{\&} \varphi_1 + \varphi_2 + \Phi}$$

The last step comes from a technical reason. The naive side-condition breaks subject reduction lemma. We need the following property for the subject reduction lemma.

Proposition 5.2.3. *If* $\Xi, \alpha \vdash M : \tau \mathbin{\&} \varphi$ *and* $\mathrm{fnv}(\Phi) \subseteq \mathrm{ndom}(\Xi)$*, then* $\Xi \vdash M[\alpha := \Phi] : \tau[\alpha := \Phi] \mathbin{\&} \varphi[\alpha := \Phi]$.

However, the name substitution for the effect breaks the disjunctivity. (For instance, $\mathbb{L}(\alpha)$ and $\mathbb{L}(\circ)$ are disjunctive, but $\mathbb{L}(\alpha[\alpha := \circ])$ and $\mathbb{L}(\circ[\alpha := \circ])$ are not.)

To amend the problem we adopt more specific side-condition as follows and make the effect in the conclusion as $\Phi'(\bar{\varphi}_1 + \bar{\varphi}_2 + \bar{\varphi}_3)$.

$$\begin{gathered}\mathbb{L}(\varphi_1) \subseteq \mathbb{L}(\Phi'\bar{\varphi}_1)\\ \mathbb{L}(\varphi_2) \subseteq \mathbb{L}(\Phi'\bar{\varphi}_2)\\ \mathbb{L}(\Phi) \subseteq \mathbb{L}(\Phi'\bar{\varphi}_3)\\ \mathbb{L}(\bar{\varphi}_1) \uplus \mathbb{L}(\bar{\varphi}_2) \uplus \mathbb{L}(\bar{\varphi}_3)\end{gathered}$$

The point is that string variables are gathered into the prefix and name disjunctivity is guaranteed by the closed effects, which are not altered by a substitution. Indeed, this side-condition quite respects the compilation rules, which create a fresh name by appending the constants $\circ, \bullet$ to the tail of seed and a string variable is pushed to the head of the seed.

The (TT-Choice) is almost obtained. We can drive away the first two subset relation into (TT-Sub). Additionally, it can be seen that the effects from left and right side of a choice, namely φ_1 and φ_2, need not be distinct since both sides never collaborate. So, we can take one large effect $\bar{\varphi}$ which subsume φ_1 and φ_2 instead of $\bar{\varphi}_1$ and $\bar{\varphi}_2$, i.e., $\mathbb{L}(\varphi_1) \subseteq \mathbb{L}(\Phi'\bar{\varphi})$ and $\mathbb{L}(\varphi_2) \subseteq \mathbb{L}(\Phi'\bar{\varphi})$.

(TT-App) and (TT-SApp) is constructed in a similar manner.

5.3 Property

In the proof of this section, we implicitly use well-known properties about regular expressions, e.g., $\mathbb{L}(\epsilon\varphi) = \mathbb{L}(\varphi)$, $\mathbb{L}(\varphi_1(\varphi_2 + \varphi_3)) = \mathbb{L}(\varphi_1\varphi_2 + \varphi_1\varphi_3)$, etc. This is one reason we have not fully mechanized the proofs yet.

5.3.1 Type Soundness of $\lambda^{\|\omega}$

We start from investigation for the effect system: the effect system prevents coordination by Corollary 5.3.6 and the property is preserved by the reduction by Lemma 5.3.3.

Lemma 5.3.1 (Substitution)**.** *If* $\Xi, x : \tau' \vdash M : \tau \,\&\, \varphi$ *and* $\Xi \vdash M' : \tau' \,\&\, \varnothing$*, then* $\Xi \vdash M[x := M'] : \tau \,\&\, \varphi$.

Lemma 5.3.2 (Name substitution)**.** *If* $\Xi, \alpha \vdash M : \tau \,\&\, \varphi$ *and* $\mathrm{fnv}(\Phi) \subseteq \mathrm{ndom}(\Xi)$*, then* $\Xi \vdash M[\alpha := \Phi] : \tau[\alpha := \Phi] \,\&\, \varphi[\alpha := \Phi]$.

Lemma 5.3.3 (Subject reduction)**.** *If* $\varnothing \vdash M : \tau \,\&\, \varphi$ *and* $M \xrightarrow{\Delta} M'$*, then* $\varnothing \vdash M' : \tau \,\&\, \varphi$.

Definition 5.3.4. Non-coordinated reduction relation, denoted as $M \Longrightarrow M'$, is derived from the rules in Figure 5.6 by replacing $\xrightarrow{\Delta}$ with $\Longrightarrow$ and removing (TR-WorldL) and (TR-WorldR).

Lemma 5.3.5. *If* $\varnothing \vdash M : \tau \,\&\, \varphi$*,* $M \xrightarrow{\Delta} M'$*, and* $\mathbb{L}(\varphi) \uplus \{\omega \mid \omega_+ \in \Delta \vee \omega_- \in \Delta\}$*; then* $M \Longrightarrow M'$.

Corollary 5.3.6 (Non-coordination)**.** *If* $\varnothing \vdash M : \tau \,\&\, \varphi$ *and* $M \xrightarrow{\varnothing} M'$*, then* $M \Longrightarrow M'$.

5.3.2 Soundness of Compilation

Towards the goal we show a well-typed expression in $\lambda^{\|}$ is mapped into a well-typed expression in $\lambda^{\|\omega}$ by the compilation.

Definition 5.3.7 (Compilation for types)**.** Compilation for types is given as follows.

$$\begin{aligned} [\![\mathrm{nat}]\!] &= \mathrm{nat} \\ [\![T_1 \to T_2]\!] &= [\![T_1]\!] \overset{\varnothing}{\to} \forall \alpha.^{\alpha(\circ+\bullet)^*} [\![T_2]\!] \end{aligned}$$

Definition 5.3.8 (Compilation for typing environment)**.** Compilation for typing environment is given by the following obvious way.

$$\begin{aligned} [\![\varnothing]\!] &= \varnothing \\ [\![\Gamma, x : T]\!] &= [\![\Gamma]\!], x : [\![T]\!] \end{aligned}$$

Lemma 5.3.9. $\varnothing \Vdash [\![T]\!]$.

Lemma 5.3.10. $[\![\Gamma]\!]$ ok.

Lemma 5.3.11. *If* $\Gamma \vdash e : T$*, then* $[\![\Gamma]\!], \alpha \vdash [\![e]\!]^{\alpha}_{\bar{\Phi}} : [\![T]\!] \,\&\, \alpha\bar{\Phi}(\circ + \bullet)^*$.

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

$$\begin{aligned}
\lfloor x \rfloor &= x \\
\lfloor M_1\, M_2 \rfloor &= \lfloor M_1 \rfloor \lfloor M_2 \rfloor \\
\lfloor \lambda x.M \rfloor &= \lambda x.\lfloor M \rfloor \\
\lfloor M\, \Phi \rfloor &= \lfloor M \rfloor\, \lambda x.x \\
\lfloor \Lambda \alpha.M \rfloor &= \lambda x.\lfloor M \rfloor && \text{where } x \text{ is fresh} \\
\lfloor \mu f.M \rfloor &= \mu f.\lfloor M \rfloor \\
\lfloor (M_1 \parallel^{\Phi} M_2) \rfloor &= (\lfloor M_1 \rfloor \parallel \lfloor M_2 \rfloor)
\end{aligned}$$

Figure 5.10: Name erasure function

5.3.3 Bisimulation

Next we show a well-typed expression in $\lambda^{\parallel\omega}$ behaves as same as the expression of $\lambda^{\parallel}$ which is obtained by erasing the name related parts.

Definition 5.3.12 (strong bisimulation between $\lambda^{\parallel}$ and $\lambda^{\parallel\omega}$)**.** A binary relation R between $\lambda^{\parallel}$ and $\lambda^{\parallel\omega}$ expressions is called *strong bisimulation* iff the following conditions hold.

- If $e\ R\ M$ and $e \longrightarrow e'$, then $M \xrightarrow{\emptyset} M'$ and $e'\ R\ M'$.
- If $e\ R\ M$ and $M \xrightarrow{\emptyset} M'$, then $e \longrightarrow e'$ and $e'\ R\ M'$.

Definition 5.3.13 (weak bisimulation between $\lambda^{\parallel}$ and $\lambda^{\parallel\omega}$)**.** A binary relation R between $\lambda^{\parallel}$ and $\lambda^{\parallel\omega}$ expressions is called *weak bisimulation* iff the following conditions hold.

- If $e\ R\ M$ and $e \longrightarrow e'$, then $M \xrightarrow{\emptyset}{}^{*} M'$ and $e'\ R\ M'$.
- If $e\ R\ M$ and $M \xrightarrow{\emptyset} M'$, then $e \longrightarrow^{*} e'$ and $e'\ R\ M'$.

Definition 5.3.14 (weak bisimulation for $\lambda^{\parallel}$)**.** We also call a binary relation R between $\lambda^{\parallel}$ expressions weak bisimulation iff the following conditions hold.

- If $e_1\ R\ e_2$ and $e_1 \longrightarrow e_1'$, then $e_2 \longrightarrow^{*} e_2'$ and $e_1'\ R\ e_2'$.
- If $e_1\ R\ e_2$ and $e_2 \longrightarrow e_2'$, then $e_1 \longrightarrow^{*} e_1'$ and $e_1'\ R\ e_2'$.

Definition 5.3.15. We define the *name erasure* function $\lfloor\cdot\rfloor$ from $\lambda^{\parallel\omega}$ expressions into $\lambda^{\parallel}$ expressions as in Figure 5.10. The important point of the definition is that name abstractions and applications are replaced by dummy lambda abstractions and applications. If we do not do that, i.e., just erase the name abstractions and applications, it will happens that a value of $\lambda^{\parallel\omega}$, which cannot be evaluated, is evaluated in $\lambda^{\parallel}$ after applying the name erasure function. (Consider $\Lambda\alpha.(\lambda x.x)\,\lambda x.x$ and name erased expression $(\lambda x.x)\,\lambda x.x$.)

Lemma 5.3.16. $\lfloor M[x \coloneqq M'] \rfloor = \lfloor M \rfloor[x \coloneqq \lfloor M' \rfloor]$.

Proof. The proof is routine by structural induction on M. □

$$
\begin{aligned}
\lceil x \rceil &= x \\
\lceil e_1\, e_2 \rceil &= \lceil e_1 \rceil\, \lceil e_2 \rceil\, \lambda x.x \\
\lceil \lambda x.e \rceil &= \lambda x.\lambda y.\lceil e \rceil && \text{where } y \text{ is fresh} \\
\lceil \mu f.e \rceil &= \mu f.\lceil e \rceil \\
\lceil (e_1 \parallel e_2) \rceil &= (\lceil e_1 \rceil \parallel \lceil e_2 \rceil)
\end{aligned}
$$

Figure 5.11: Pseudo compilation

Lemma 5.3.17. $\lfloor M[\alpha \coloneqq \Phi] \rfloor = \lfloor M \rfloor$.

Proof. The proof is routine by structural induction on M. □

Lemma 5.3.18. *If* $\varnothing \vdash M : \tau \,\&\, \varphi$ *and* $\lfloor M \rfloor \longrightarrow e'$, *then* $M \xrightarrow{\varnothing} M'$ *and* $\lfloor M' \rfloor = e'$.

Proof. The proof is by induction on the given derivation of $\varnothing \vdash M : \tau \,\&\, \varphi$. Note that well-typedness of M is necessary because function applications and name applications are collapsed by name erasing. □

Lemma 5.3.19. *If* $M \Longrightarrow M'$, *then* $\lfloor M \rfloor \longrightarrow \lfloor M' \rfloor$.

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

Corollary 5.3.20. *If* $\varnothing \vdash M : \tau \,\&\, \varphi$ *and* $M \xrightarrow{\varnothing} M'$, *then* $\lfloor M \rfloor \longrightarrow \lfloor M' \rfloor$.

Proof. This is a corollary of Corollary 5.3.6 and Lemma 5.3.19. □

Definition 5.3.21. We give the binary relation between source expressions and target expressions, written $e \sim M$ as follows.

$$e \sim M \iff \varnothing \vdash M : \tau \,\&\, \varphi \wedge \lfloor M \rfloor = e$$

Corollary 5.3.22. $\sim$ *is a strong bisimulation.*

Proof. This is a corollary of Lemma 5.3.3, Lemma 5.3.18, and Corollary 5.3.20. □

Corollary 5.3.23. *If* $\varnothing \vdash e : T$, *then* $\lfloor [\![e]\!]^{\alpha}_{\Phi}[\alpha \coloneqq \epsilon] \rfloor \sim [\![e]\!]^{\alpha}_{\Phi}[\alpha \coloneqq \epsilon]$.

Proof. This is a corollary of Lemma 5.3.2 and Lemma 5.3.11. □

Definition 5.3.24. We define the *pseudo compilation* from/to $\lambda^{\parallel}$ expressions as shown in Figure 5.11. This function is not essential for our discussion, but we use the function for convenience writing in the following.

Lemma 5.3.25. $\lfloor [\![e]\!]^{\alpha}_{\Phi} \rfloor = \lceil e \rceil$.

Lemma 5.3.26. $\lceil e[x \coloneqq e'] \rceil = \lceil e \rceil [x \coloneqq \lceil e' \rceil]$.

Lemma 5.3.27. *If* $\lceil e \rceil \longrightarrow e'$, *then* $e \longrightarrow^{*} e''$ *and* $\lceil e'' \rceil = e'$.

Lemma 5.3.28. *If $e \longrightarrow e'$, then $\lceil e \rceil \longrightarrow^{*} e''$ and $\lceil e' \rceil = e''$.*

Definition 5.3.29. We give the binary relation between $\lambda^{\parallel}$ expressions, denoted by $\approx$, as $e \approx \lceil e \rceil$.

Corollary 5.3.30. *The binary relation $e \approx \lceil e \rceil$ is a weak bisimulation.*

Theorem 5.3.31. *If $\varnothing \vdash e : T$, then $e \approx \circ \sim [\![e]\!]^{\alpha}_{\epsilon}[\alpha \coloneqq \epsilon]$.*

The result is a bit blurred since expressions before and after compilation are related by two relations. More directly, the correspondence can be shown as follows.

$$\begin{array}{rcl} e & \longrightarrow^{*} & e' \\ & \varnothing & \\ [\![e]\!]^{\alpha}_{\epsilon}[\alpha \coloneqq \epsilon] & \longrightarrow^{*} & M' \end{array} \quad \text{and} \quad \lceil e' \rceil = \lfloor M' \rfloor$$

5.4 Summary

We give a compilation algorithm from $\lambda^{\parallel}$, a simply typed lambda calculus with nondeterministic choice, into $\lambda^{\parallel\omega}$, a simply typed lambda calculus with coordinated choice; and show the compilation is sound and correct. The discussion is based not on a manifest contract system, but we believe that the result can be applied to PCFvΔ_{H} and $\lambda^{H\parallel^{\Phi}}$, and we could implement dependent function types in PCFvΔ_{H} since the facility of coordinate choices and manifest contract systems are orthogonal.

6 Related Work

Manifest contract calculi. Since the beginning of manifest contracts for hybrid type checking [19], several extended calculi have been proposed: one with algebraic data types [54]; one with stateful computation [52]; ones with parametric polymorphism [1, 53]; and Sage [22], a manifest contract calculus with recursive types, dynamic types, and the Type:Type discipline [4]. We think that nondeterminism is an orthogonal issue and our approach is easy to integrate to those calculi.

One remark is that some studies [29, 54] use nondeterminism, but One [54] force to make their operational semantics deterministic by assuming an oracle machine which chooses correct alternatives deterministically. Another [29] use actual nondeterminism. However, they use nondeterministic expressions only for a type reconstruction, which includes predicate synthesis; and thus it is not intended that they appear in source code.

Static verification by using a dependent refinement type system. Although we give only dynamic checking in this paper, there are broad studies on static verification of higher-order functional programs [51, 58, 31, 57, 66, 61, 59, 11]. The main difference from a manifest contract calculus is that predicates of their refinement types are written in logical formulae. Thus, the decidability of their verification methods rely on that of the logic they adopt. It carries the trade-off: if logic is decidable, predicates are less expressive than the manifest way; or otherwise, predicates are indeed more expressive, but a verifier (compiler) makes false positives. A manifest contract system stays in the middle with paying run-time cost in exchange for no false positives. Note that hybrid type checking [19]—the origin of manifest contract systems—combines static checking via subsumption rules; and some studies [1, 53] conjectured that *up-casts*, e.g., $(0 : \{x{:}\mathtt{int} \mid x = 0\} \Rightarrow \{x{:}\mathtt{int} \mid x \geq 0\})$, can be eliminated as optimization by defining a subtyping relation as an afterthought. So we would pay back some run-time cost even in a manifest contract system.

We should relate [59] in more detail. They proposed a verification method for a nondeterministic functional programming language. They avoid the difficulties discussed in this paper by restricting predicates in refinement types to be pure—actually, those are formulae of second-order arithmetic. The novelty in their type system is that it can expresses safety, non-safety, termination, and non-termination properties by using extended refinement types. A safety property means that *the property holds in every possibility*; and a non-safety property means that *there exists a possibility in which the property holds.* What we study in this paper can be viewed as a technique of *dynamic* verification of a non-safety property. It requires further work to see how static verification of casts (to see if each cast is successful) is related to Unno, Satake, and Terauchi's static verification technique. Due to the nature of dynamic verification, termination verification is out of our scope.

Nondeterministic lambda calculi. Semantics of nondeterministic choice is studied from a long time ago [36]. Well known semantics of nondeterministic choice is summarized by Søndergaard and Sestoft [55]. Dezani-Ciancaglini et al. studied a calculus equipped

with two of those semantics together, as *parallelism* and *internal choice*, and gave a fully abstract semantics to justify their operational semantics [13]. Set-based reduction for choice was proposed by Kutzner and Schmidt-Schauß in order to guarantee confluence with nondeterministic choice [32]. They also demonstrated that their language has the *unfolding property* [55]—a function application can be unfolded into a function body where actual arguments are substituted for formal (so the argument might be duplicated) without changing the meaning of the program; but what they actually proved is that an application to a “pure” expression can be unfolded, and they gave the method to discriminate pure and impure expressions. Therefore, their semantics does not suffice for our setting—we need semantics in which arbitrary expressions could duplicate.

There are few studies on dependent type systems for nondeterministic languages. In his manuscript [63], Warrell proposed a probabilistic dependent type system. His motivation is an investigation of Curry-Howard isomorphism for Markov logic network. As far as we understand, the type system has a (probabilistic) branching operator for types, as well as terms, and it appears that typechecking requires computation of normal forms. So, it is not clear how his method can be applied to Turing-complete languages.

Variational programming. Apart from nondeterminism, software product line (SPL) community develops a type system [25] which deal with variation of code. They try to type check all variants of a single code—like C code switching features by `#ifdef` macro—without generating each variant code (because, in SPL engineering, the number of variants tends to be a quite large and it is hard to type check all variants separately). Their system has some similarity to our type system in the sense that a type of code varies according to the context it inhabits.

For a more theoretical aspect, Erwig study the *choice calculus* [16], a primitive model handling software variation. Since their work targets code variation, it has no operational semantics; but one extension [7] gave an operational semantics in order to manipulate variants at runtime. The semantics is quite similar in the point of synchronization of *choice* which is a one for code variants in their work (and not for nondeterminism); but other parts are fairly different and specialized for a specific purpose, e.g., their calculus has no mechanism to generate fresh choice points.

Intersection types. Intersection types were introduced in Curry-style type assignment systems by Coppo et al. [10] and Pottinger [48] 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* [48, 60]. Then, intersection types are introduced to programming languages to enrich the descriptive power of types [50, 2, 15].

Intersection Contracts for Untyped Languages. One of the first attempts at implementing intersection-like contracts is found in DrRacket [18]. 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 [26] 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[1] 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. [64] 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 $\mathrm{PCFv}\Delta_{\mathrm{H}}$ extended with blame labels, following Greenberg et al. [21].

Gradual Typing with Intersection Types. Castagna and Lanvin [5] 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 [62]. 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 [6], 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.

[1] A kind of casts in their language.

7 Conclusion

In this thesis, we develop a typed software contracts. Especially we focus on the way to write conjunctive contracts. Technically we give two manifest contract systems and give the idea which could be used to combine two systems.

In Chapter 3, we have designed and formalized a manifest contract system $\mathrm{PCFv}\Delta_{\mathrm{H}}$ with refinement intersection types. As a result of our formal development, $\mathrm{PCFv}\Delta_{\mathrm{H}}$ 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.

In Chapter 4, we have developed $\lambda^{H\|^{\Phi}}$, a nondeterministic manifest contract calculus. By introducing a new kind of choice called coordinated choice, the calculus is free from restriction in the kind of programs used in predicates in refinements types. Coordinated choice can share nondeterministic decisions by using names and the semantics is given in such a way that a program reduces to the set (expressed by $\|$) of all possible results. The semantics makes a choice copyable by substitution and solves problems that occur when we introduce dependent function types. The semantics and type system of $\lambda^{H\|^{\Phi}}$ are given by using the characteristic concept called orthant. Orthant gives the view for a choice. Since the view does not change the meaning of expressions except choices, we expect that coordinated choice is quite easily integrated into other manifest contract systems. By the observation, we hope this approach is easily applied to any other languages.

In Chapter 5, we give a compilation algorithm from $\lambda^{\|}$, a simply typed lambda calculus with nondeterministic choice, into $\lambda^{\|\omega}$, a simply typed lambda calculus with coordinated choice; and show the compilation is sound and complete. The discuss is based not on a manifest contract system, but we believe that the result can be applied to $\mathrm{PCFv}\Delta_{\mathrm{H}}$ and $\lambda^{H\|^{\Phi}}$, and we could implement dependent function types in $\mathrm{PCFv}\Delta_{\mathrm{H}}$ since the facility of coordinate choices and manifest contract systems are orthogonal.

Future Work. Obvious future work is to complete the integration of $\mathrm{PCFv}\Delta_{\mathrm{H}}$ and $\lambda^{H\|^{\Phi}}$. We show the compilation method on a simply typed system. So, it is still unknown how we re-organize the discussion on a dependently typed system. Actually, we predict some coordination is unavoidable in a dependently typed system. It arises another interesting question—what nondeterministic choices in a dependently typed system actually are?

Another future work is a specific one for manifest contract systems. A property called *up-cast elimination* [1]—*a cast from subtype into supertype can be safely removed at compile-time.* This property is quite important not only because of efficiency but also to show a cast raises

unintentional `blame`. Some one noticed that a type soundness holds even if a cast always fails because soundness only tell us an information about succeeded computation. In spite of the importance, the property is shown in few manifest contract systems because the proof tends to too complicated.

Towards a practical language, cast semantics we adopt in this thesis is too naive and inefficient, which duplicate code frequently. For instance, 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.

Bibliography

[1] João Filipe Belo et al. "Polymorphic Contracts". In: *Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*. Ed. by Gilles Barthe. Vol. 6602. Lecture Notes in Computer Science. Springer, 2011, pp. 18–37. ISBN: 978-3-642-19717-8. DOI: 10.1007/978-3-642-19718-5_2. URL: https://doi.org/10.1007/978-3-642-19718-5_2.

[2] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. "CDuce: an XML-centric general-purpose language". In: *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, Uppsala, Sweden, August 25-29, 2003*. Ed. by Colin Runciman and Olin Shivers. ACM, 2003, pp. 51–63. ISBN: 1-58113-756-7. DOI: 10.1145/944705.944711. URL: https://doi.org/10.1145/944705.944711.

[3] Matthias Blume and David A. McAllester. "Sound and complete models of contracts". In: *J. Funct. Program.* 16.4-5 (2006), pp. 375–414. DOI: 10.1017/S0956796806005971. URL: https://doi.org/10.1017/S0956796806005971.

[4] Luca Cardelli. *A Polymorphic λ-calculus with Type:Type*. Technical Report 10. DEC Systems Research Center, 1986.

[5] Giuseppe Castagna and Victor Lanvin. "Gradual typing with union and intersection types". In: *PACMPL* 1.ICFP (2017), 41:1–41:28. DOI: 10.1145/3110285. URL: https://doi.org/10.1145/3110285.

[6] Giuseppe Castagna et al. "Gradual typing: a new perspective". In: *PACMPL* 3.POPL (2019), 16:1–16:32. DOI: 10.1145/3290329. URL: https://doi.org/10.1145/3290329.

[7] Sheng Chen, Martin Erwig, and Eric Walkingshaw. "A Calculus for Variational Programming". In: *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*. 2016, 6:1–6:28. DOI: 10.4230/LIPIcs.ECOOP.2016.6. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2016.6.

[8] Koen Claessen and John Hughes. "QuickCheck: a lightweight tool for random testing of Haskell programs". In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*. Ed. by Martin Odersky and Philip Wadler. ACM, 2000, pp. 268–279. ISBN: 1-58113-202-6. DOI: 10.1145/351240.351266. URL: https://doi.org/10.1145/351240.351266.

[9] Youyou Cong and Kenichi Asai. "Handling delimited continuations with dependent types". In: *PACMPL* 2.ICFP (2018), 69:1–69:31. DOI: 10.1145/3236764. URL: https://doi.org/10.1145/3236764.

[10] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. “Functional Characters of Solvable Terms”. In: *Math. Log. Q.* 27.2-6 (1981), pp. 45–58. DOI: 10.1002/malq.19810270205. URL: https://doi.org/10.1002/malq.19810270205.

[11] Benjamin Cosman and Ranjit Jhala. “Local refinement typing”. In: *PACMPL* 1.ICFP (2017), 26:1–26:27. DOI: 10.1145/3110270. URL: https://doi.org/10.1145/3110270.

[12] Patrick Cousot and Radhia Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. In: *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*. Ed. by Robert M. Graham, Michael A. Harrison, and Ravi Sethi. ACM, 1977, pp. 238–252. DOI: 10.1145/512950.512973. URL: https://doi.org/10.1145/512950.512973.

[13] Mariangiola Dezani-Ciancaglini, Ugo de’Liguoro, and Adolfo Piperno. “A Filter Model for Concurrent lambda-Calculus”. In: *SIAM J. Comput.* 27.5 (1998), pp. 1376–1419. DOI: 10.1137/S0097539794275860. URL: https://doi.org/10.1137/S0097539794275860.

[14] Christos Dimoulas et al. “Correct blame for contracts: no more scapegoating”. In: *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. Ed. by Thomas Ball and Mooly Sagiv. ACM, 2011, pp. 215–226. ISBN: 978-1-4503-0490-0. DOI: 10.1145/1926385.1926410. URL: https://doi.org/10.1145/1926385.1926410.

[15] Joshua Dunfield. “Refined typechecking with Stardust”. In: *Proceedings of the ACM Workshop Programming Languages meets Program Verification, PLPV 2007, Freiburg, Germany, October 5, 2007*. Ed. by Aaron Stump and Hongwei Xi. ACM, 2007, pp. 21–32. ISBN: 978-1-59593-677-6. DOI: 10.1145/1292597.1292602. URL: https://doi.org/10.1145/1292597.1292602.

[16] Martin Erwig and Eric Walkingshaw. “The Choice Calculus: A Representation for Software Variation”. In: *ACM Trans. Softw. Eng. Methodol.* 21.1 (Dec. 2011), 6:1–6:27. ISSN: 1049-331X. DOI: 10.1145/2063239.2063245. URL: http://doi.acm.org/10.1145/2063239.2063245.

[17] Robert Bruce Findler and Matthias Felleisen. “Contracts for higher-order functions”. In: *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP ’02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002*. Ed. by Mitchell Wand and Simon L. Peyton Jones. ACM, 2002, pp. 48–59. ISBN: 1-58113-487-8. DOI: 10.1145/581478.581484. URL: https://doi.org/10.1145/581478.581484.

[18] Robert Bruce Findler and PLT. *DrRacket: Programming Environment*. Tech. rep. PLT-TR-2010-2. https://racket-lang.org/tr2/. PLT Design Inc., 2010.

[19] Cormac Flanagan. "Hybrid type checking". In: *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*. Ed. by J. Gregory Morrisett and Simon L. Peyton Jones. ACM, 2006, pp. 245–256. ISBN: 1-59593-027-2. DOI: 10.1145/1111037.1111059. URL: https://doi.org/10.1145/1111037.1111059.

[20] Michael Greenberg. "Space-Efficient Manifest Contracts". In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. Ed. by Sriram K. Rajamani and David Walker. ACM, 2015, pp. 181–194. ISBN: 978-1-4503-3300-9. DOI: 10.1145/2676726.2676967. URL: https://doi.org/10.1145/2676726.2676967.

[21] Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. "Contracts made manifest". In: *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*. Ed. by Manuel V. Hermenegildo and Jens Palsberg. ACM, 2010, pp. 353–364. ISBN: 978-1-60558-479-9. DOI: 10.1145/1706299.1706341. URL: https://doi.org/10.1145/1706299.1706341.

[22] Jessica Gronski et al. "Sage: Hybrid checking for flexible specifications". In: *Scheme and Functional Programming Workshop*. 2006, pp. 93–104.

[23] Matthew Hennessy and Edward A. Ashcroft. "Parameter-Passing Mechanisms and Nondeterminism". In: *Proceedings of the 9th Annual ACM Symposium on Theory of Computing, May 4-6, 1977, Boulder, Colorado, USA*. Ed. by John E. Hopcroft, Emily P. Friedman, and Michael A. Harrison. ACM, 1977, pp. 306–311. DOI: 10.1145/800105.803420. URL: https://doi.org/10.1145/800105.803420.

[24] Manuel V. Hermenegildo and Jens Palsberg, eds. *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*. ACM, 2010. ISBN: 978-1-60558-479-9. URL: http://dl.acm.org/citation.cfm?id=1706299.

[25] Christian Kästner et al. "Type Checking Annotation-based Product Lines". In: *ACM Trans. Softw. Eng. Methodol.* 21.3 (July 2012), 14:1–14:39. ISSN: 1049-331X. DOI: 10.1145/2211616.2211617. URL: http://doi.acm.org/10.1145/2211616.2211617.

[26] Matthias Keil and Peter Thiemann. "Blame assignment for higher-order contracts with intersection and union". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. Ed. by Kathleen Fisher and John H. Reppy. ACM, 2015, pp. 375–386. ISBN: 978-1-4503-3669-7. DOI: 10.1145/2784731.2784737. URL: https://doi.org/10.1145/2784731.2784737.

[27] Brian W. Kernighan and Dennis Ritchie. *The C Programming Language*. Prentice-Hall, 1978. ISBN: 0-13-110163-3.

[28] Kenneth Knowles and Cormac Flanagan. "Hybrid Type Checking". In: *ACM Transactions on Programming Languages and Systems* 32.2:6 (2010). DOI: 10.1145/1667048.1667051.

[29] Kenneth Knowles and Cormac Flanagan. "Type Reconstruction for General Refinement Types". In: *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings.* Ed. by Rocco De Nicola. Vol. 4421. Lecture Notes in Computer Science. Springer, 2007, pp. 505–519. ISBN: 978-3-540-71314-2. DOI: 10.1007/978-3-540-71316-6_34. URL: https://doi.org/10.1007/978-3-540-71316-6_34.

[30] Naoki Kobayashi. "Types and higher-order recursion schemes for verification of higher-order programs". In: *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009.* Ed. by Zhong Shao and Benjamin C. Pierce. ACM, 2009, pp. 416–428. ISBN: 978-1-60558-379-2. DOI: 10.1145/1480881.1480933. URL: https://doi.org/10.1145/1480881.1480933.

[31] Naoki Kobayashi, Ryosuke Sato, and Hiroshi Unno. "Predicate abstraction and CEGAR for higher-order model checking". In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011.* Ed. by Mary W. Hall and David A. Padua. ACM, 2011, pp. 222–233. ISBN: 978-1-4503-0663-8. DOI: 10.1145/1993498.1993525. URL: https://doi.org/10.1145/1993498.1993525.

[32] Arne Kutzner and Manfred Schmidt-Schauß. "A Non-Deterministic Call-by-Need Lambda Calculus". In: *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998.* Ed. by Matthias Felleisen, Paul Hudak, and Christian Queinnec. ACM, 1998, pp. 324–335. ISBN: 1-58113-024-4. DOI: 10.1145/289423.289462. URL: https://doi.org/10.1145/289423.289462.

[33] Xavier Leroy. "Formal verification of a realistic compiler". In: *Commun. ACM* 52.7 (2009), pp. 107–115. DOI: 10.1145/1538788.1538814. URL: https://doi.org/10.1145/1538788.1538814.

[34] Luigi Liquori and Claude Stolze. "The Delta-calculus: Syntax and Types". In: *4th International Conference on Formal Structures for Computation and Deduction, FSCD 2019, June 24-30, 2019, Dortmund, Germany.* Ed. by Herman Geuvers. Vol. 131. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019, 28:1–28:20. ISBN: 978-3-95977-107-8. DOI: 10.4230/LIPIcs.FSCD.2019.28. URL: https://doi.org/10.4230/LIPIcs.FSCD.2019.28.

[35] P. Martin-Löf. "Constructive Mathematics and Computer Programming". In: *Proc. of a Discussion Meeting of the Royal Society of London on Mathematical Logic and Programming Languages.* London, United Kingdom: Prentice-Hall, Inc., 1985, pp. 167–184. ISBN: 0135614651.

[36] John McCarthy. "A Basis for a Mathematical Theory of Computation, Preliminary Report". In: *Papers Presented at the May 9-11, 1961, Western Joint IRE-AIEE-ACM Computer Conference.* IRE-AIEE-ACM '61 (Western). Los Angeles, California: ACM, 1961, pp. 225–238. DOI: 10.1145/1460690.1460715. URL: http://doi.acm.org/10.1145/1460690.1460715.

[37] Bertrand Meyer. *Object-Oriented Software Construction, 1st edition*. Prentice-Hall, 1988. ISBN: 0-13-629031-0.

[38] Bertrand Meyer. *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall, 1997. ISBN: 0-13-629155-4. URL: http://www.eiffel.com/doc/oosc/page.html.

[39] Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[40] Robin Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989. ISBN: 978-0-13-115007-2.

[41] Leonardo Mendonça de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings.* Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340. ISBN: 978-3-540-78799-0. DOI: 10.1007/978-3-540-78800-3_24. URL: https://doi.org/10.1007/978-3-540-78800-3_24.

[42] Flemming Nielson and Hanne Riis Nielson. "Type and Effect Systems". In: *Correct System Design, Recent Insight and Advances, (to Hans Langmaack on the occasion of his retirement from his professorship at the University of Kiel)*. Ed. by Ernst-Rüdiger Olderog and Bernhard Steffen. Vol. 1710. Lecture Notes in Computer Science. Springer, 1999, pp. 114–136. ISBN: 3-540-66624-9. DOI: 10.1007/3-540-48092-7_6. URL: https://doi.org/10.1007/3-540-48092-7%5C_6.

[43] Yuki Nishida and Atsushi Igarashi. "Compilation of Coordinated Choice". In: *CoRR* abs/2004.14084 (2020). arXiv: 2004.14084 [cs.PL]. URL: https://arxiv.org/abs/2004.14084.

[44] Yuki Nishida and Atsushi Igarashi. "Manifest Contracts with Intersection Types". In: *Programming Languages and Systems - 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1-4, 2019, Proceedings*. Ed. by Anthony Widjaja Lin. Vol. 11893. Lecture Notes in Computer Science. Springer, 2019, pp. 33–52. ISBN: 978-3-030-34174-9. DOI: 10.1007/978-3-030-34175-6_3. URL: https://doi.org/10.1007/978-3-030-34175-6_3.

[45] Yuki Nishida and Atsushi Igarashi. "Nondeterministic Manifest Contracts". In: *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming, PPDP 2018, Frankfurt am Main, Germany, September 03-05, 2018*. Ed. by David Sabel and Peter Thiemann. ACM, 2018, 16:1–16:13. DOI: 10.1145/3236950.3236964. URL: https://doi.org/10.1145/3236950.3236964.

[46] Andrew M. Pitts. "Operational Semantics and Program Equivalence". In: *Proceedings of Applied Semantics, International Summer School, APPSEM 2000*. Vol. 2395. Lecture Notes on Computer Science. Springer-Verlag, Sept. 2000, pp. 378–412.

[47] Gordon D. Plotkin. "LCF Considered as a Programming Language". In: *Theor. Comput. Sci.* 5.3 (1977), pp. 223–255. DOI: 10.1016/0304-3975(77)90044-5. URL: https://doi.org/10.1016/0304-3975(77)90044-5.

[48] Garrel Pottinger. “A type assignment for the strongly normalizabile λ-terms”. In: *To H. B. Curry, Essays in Combinatory Logic, Lambda-Calculus and Formalism* (1980), pp. 561–577.

[49] Sriram K. Rajamani and David Walker, eds. *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015.* ACM, 2015. ISBN: 978-1-4503-3300-9. URL: http://dl.acm.org/citation.cfm?id=2676726.

[50] John C. Reynolds. *Preliminary design of the programming language Forsythe.* Tech. rep. CMU-CS-88-159. Carnegie Mellon University, 1988.

[51] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. “Liquid types”. In: *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008.* Ed. by Rajiv Gupta and Saman P. Amarasinghe. ACM, 2008, pp. 159–169. ISBN: 978-1-59593-860-2. DOI: 10.1145/1375581.1375602. URL: https://doi.org/10.1145/1375581.1375602.

[52] Taro Sekiyama and Atsushi Igarashi. “Stateful manifest contracts”. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017.* Ed. by Giuseppe Castagna and Andrew D. Gordon. ACM, 2017, pp. 530–544. ISBN: 978-1-4503-4660-3. DOI: 10.1145/3009837. URL: http://dl.acm.org/citation.cfm?id=3009875.

[53] Taro Sekiyama, Atsushi Igarashi, and Michael Greenberg. “Polymorphic Manifest Contracts, Revised and Resolved”. In: *ACM Trans. Program. Lang. Syst.* 39.1 (2017), 3:1–3:36. DOI: 10.1145/2994594. URL: https://doi.org/10.1145/2994594.

[54] Taro Sekiyama, Yuki Nishida, and Atsushi Igarashi. “Manifest Contracts for Datatypes”. In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015.* Ed. by Sriram K. Rajamani and David Walker. ACM, 2015, pp. 195–207. ISBN: 978-1-4503-3300-9. DOI: 10.1145/2676726.2676996. URL: https://doi.org/10.1145/2676726.2676996.

[55] Harald Søndergaard and Peter Sestoft. “Non-Determinism in Functional Languages”. In: *Comput. J.* 35.5 (1992), pp. 514–523. DOI: 10.1093/comjnl/35.5.514. URL: https://doi.org/10.1093/comjnl/35.5.514.

[56] Nikhil Swamy et al. “Dependent Types and Multi-Monadic Effects in F*”. In: *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL).* ACM, Jan. 2016, pp. 256–270. ISBN: 978-1-4503-3549-2. URL: https://www.fstar-lang.org/papers/mumon/.

[57] Tachio Terauchi. “Dependent types from counterexamples”. In: *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010.* Ed. by Manuel V. Hermenegildo and Jens Palsberg. ACM, 2010, pp. 119–130. ISBN: 978-1-60558-479-9. DOI: 10.1145/1706299.1706315. URL: https://doi.org/10.1145/1706299.1706315.

[58] Hiroshi Unno and Naoki Kobayashi. “Dependent type inference with interpolants”. In: *Proceedings of the 11th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, September 7-9, 2009, Coimbra, Portugal.* Ed. by António Porto and Francisco Javier López-Fraguas. ACM, 2009, pp. 277–288. ISBN: 978-1-60558-568-0. DOI: 10.1145/1599410.1599445. URL: https://doi.org/10.1145/1599410.1599445.

[59] Hiroshi Unno, Yuki Satake, and Tachio Terauchi. “Relatively complete refinement type system for verification of higher-order non-deterministic programs”. In: *PACMPL* 2.POPL (2018), 12:1–12:29. DOI: 10.1145/3158100. URL: https://doi.org/10.1145/3158100.

[60] Silvio Valentini. “An elementary proof of strong normalization for intersection types”. In: *Arch. Math. Log.* 40.7 (2001), pp. 475–488. DOI: 10.1007/s001530000070. URL: https://doi.org/10.1007/s001530000070.

[61] Niki Vazou et al. “Refinement types for Haskell”. In: *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014.* Ed. by Johan Jeuring and Manuel M. T. Chakravarty. ACM, 2014, pp. 269–282. ISBN: 978-1-4503-2873-9. DOI: 10.1145/2628136.2628161. URL: https://doi.org/10.1145/2628136.2628161.

[62] Philip Wadler and Robert Bruce Findler. “Well-Typed Programs Can’t Be Blamed”. In: *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings.* Ed. by Giuseppe Castagna. Vol. 5502. Lecture Notes in Computer Science. Springer, 2009, pp. 1–16. ISBN: 978-3-642-00589-3. DOI: 10.1007/978-3-642-00590-9_1. URL: https://doi.org/10.1007/978-3-642-00590-9_1.

[63] Jonathan H. Warrell. “A Probabilistic Dependent Type System based on Non-Deterministic Beta Reduction”. In: *CoRR* abs/1602.06420 (2016). arXiv: 1602.06420. URL: http://arxiv.org/abs/1602.06420.

[64] Jack Williams, J. Garrett Morris, and Philip Wadler. “The root cause of blame: contracts for intersection and union types”. In: *PACMPL* 2.OOPSLA (2018), 134:1–134:29. DOI: 10.1145/3276504. URL: https://doi.org/10.1145/3276504.

[65] Andrew K. Wright and Matthias Felleisen. “A Syntactic Approach to Type Soundness”. In: *Information and Computation* 115.1 (Nov. 1994), pp. 38–94.

[66] He Zhu and Suresh Jagannathan. “Compositional and Lightweight Dependent Type Inference for ML”. In: *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings.* Ed. by Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni. Vol. 7737. Lecture Notes in Computer Science. Springer, 2013, pp. 295–314. ISBN: 978-3-642-35872-2. DOI: 10.1007/978-3-642-35873-9_19. URL: https://doi.org/10.1007/978-3-642-35873-9_19.