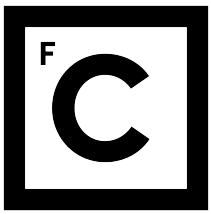
UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS



Ciências
ULisboa

Adding Dependent Types to Class-based Mutable Objects

Doutoramento em Informática
Ciência da Computação

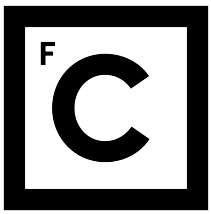
Joana Correia Campos

Tese orientada por:
Prof. Doutor Vasco Manuel Thudichum de Serpa Vasconcelos

Documento especialmente elaborado para a obtenção do grau de doutor

2018

UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS



Ciências
ULisboa

Adding Dependent Types to Class-based Mutable Objects

Doutoramento em Informática

Ciência da Computação

Joana Correia Campos

Tese orientada por:
Prof. Doutor Vasco Manuel Thudichum de Serpa Vasconcelos

Júri

Presidente:

- Doutor Nuno Fuentecilla Maia Ferreira Neves, Professor Catedrático
  Faculdade de Ciências da Universidade de Lisboa

Vogais:

- Doutor Edwin Brady, *Lecturer*
  Faculty of Science da *University of St Andrews*, Reino Unido
- Doutor Luís Manuel Marques da Costa Caires, Professor Catedrático
  Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa
- Doutor Jorge Miguel Matos Sousa Pinto, Professor Associado com Agregação
  Escola de Engenharia da Universidade do Minho
- Doutor Vasco Manuel Thudichum de Serpa Vasconcelos, Professor Catedrático
  Faculdade de Ciências da Universidade de Lisboa (orientador)
- Doutor Alysson Neves Bessani, Professor Associado
  Faculdade de Ciências da Universidade de Lisboa

Documento especialmente elaborado para a obtenção do grau de doutor

2018

Abstract

In this thesis, we present an imperative object-oriented language featuring a dependent type system designed to support class-based programming and inheritance. The system brings classes and dependent types into play so as to enable types (classes) to be refined by value parameters (indices) drawn from some constraint domain. This combination allows statically checking interesting properties of imperative programs that are impossible to check in conventional static type systems for objects.

From a pragmatic point of view, this work opens the possibility to combine the scalability and modularity of object orientation with the safety provided by dependent types in the form of index refinements. These may be used to provide additional guarantees about the fields of objects, and to prevent, for example, a method call that could leave an object in a state that would violate the class invariant. One key feature is that the programmer is not required to prove equations between indices issued by types, but instead the typechecker depends on external constraint solving. From a theoretic perspective, our fundamental contribution is to formulate a system that unifies the three very different features: dependent types, mutable objects and class-based inheritance with subtyping. Our approach includes universal and existential types, as well as union types. Subtyping is induced by inheritance and quantifier instantiation. Moreover, dependent types require the system to track type varying objects, a feature missing from standard type systems in which the type is constant throughout the object's lifetime. To ensure that an object is used correctly, aliasing is handled via a linear type discipline that enforces unique references to type varying objects. The system is decidable, provided indices are drawn from some decidable theory, and proved sound via subject reduction and progress. We also formulate a typechecking algorithm that gives a precise account of quantifier instantiation in a bidirectional style, combining type synthesis with checking. We prove that our algorithm is sound and complete.

By way of example, we implement insertion and deletion for binary search trees in an imperative style, and come up with types that ensure the binary search tree invariant. To attest the relevance of the language proposed, we provide a fully functional prototype where this and other examples can be typechecked, compiled and run. The prototype can be found at `http://rss.di.fc.ul.pt/tools/dol/`.

Keywords: Dependent types, object-oriented programming, type systems, mutable objects

Resumo

Os sistemas de tipos generalizados em linguagens de programação do tipo Java são eficazes na verificação de invariantes simples, impedindo, por exemplo, que um inteiro seja copiado para uma referência do tipo string. No entanto, muitos erros comuns, que podem ser detectados em tempo de compilação, não são tratados adequadamente por sistema de tipos estáticos de linguagens de programação convencionais. Nesta categoria de erros incluem-se referências null tantas vezes (incorrectamente) usadas para invocar métodos ou aceder a atributos, e chamadas de métodos que deixam os objectos num estado que viola os invariantes de classe. Linguagens do tipo Java são capazes de assinalar situações em que referências null estão a ser usadas no lugar de objectos, muito embora apenas o façam em tempo de execução. Do mesmo modo, pressupostos errados sobre o estado dos objectos podem causar erros em tempo de execução, ou não ser de todo detectados, devolvendo resultados errados ou inesperados.

A teoria de tipos aplicada a linguagens de programação tem-se dedicado ao longo das últimas décadas a estudar mecanismos de abstracção que descrevem comportamentos de programas. Neste contexto, os tipos dependentes são um mecanismo poderoso, já que a abstracção subjacente (dependências de termos que definem propriedades observáveis em tempo de execução) oferece uma grande precisão. Os tipos dependentes foram aplicados com sucesso em linguagens lógicas e funcionais [Augustsson, 1998, Xi and Pfenning, 1999, McBride, 2004, Flanagan, 2006, Bove et al., 2009], existindo poucas contribuições no âmbito do paradigma imperativo [Xi, 2000, Condit et al., 2007]. Desconhecem-se trabalhos no contexto de programação orientada a objectos com estado mutável.

Esta tese visa preencher esta lacuna, estudando uma solução para integrar tipos dependentes numa linguagem de objectos imperativa no sentido de a tornar mais precisa e segura. A linguagem proposta explora a capacidade expressiva da teoria dos tipos dependentes para especificação e correcção no contexto do paradigma orientado a objectos – especificação no sentido em que os tipos dependentes dão alguma indicação do estado dos objectos e dos métodos que podem ser invocados (ao contrário dos tipos convencionais que não têm essa capacidade expressiva); correcção na medida em que os programas implementados na linguagem proposta não têm erros em tempo de execução.

O trabalho contribui com a definição de uma pequena linguagem orientada a objectos com estado mutável e tipos dependentes, um sistema de tipos, uma semântica operacional

e a prova de *soundness*, e um sistema de tipos algorítmico, implementado num compilador.

DOL. A integração de tipos dependentes numa linguagem de programação tem assumido duas formas distintas: dependência total de termos arbitrários da linguagem, o que requer o desenvolvimento de estratégias que assegurem a terminação de programas (uma vez que a equivalência arbitrária de termos é em geral indecidível), ou separação de fases, restringindo o domínio dos termos dos quais os tipos dependem. A linguagem DOL (Dependent Object-oriented Language) inspira-se na abordagem adoptada pela linguagem DML [Xi, 2007], optando pela segunda forma de integração de tipos dependentes, através da qual os tipos dependem de termos especiais, chamados índices, que pertencem a um domínio restrito. De realçar que restringir a linguagem de índices a um domínio decidível, nomeadamente desigualdades sobre inteiros, é um requisito para obter um sistema de tipos decidível.

Em termos concretos, pode estabelecer-se um paralelo com os genéricos de linguagens do tipo Java, que permitem especificar tipos que são parâmetros de classes e métodos; os tipos dependentes na linguagem DOL permitem que índices sejam parâmetros de classes e métodos, usados na especificação formal de invariantes de forma não intrusiva. Por exemplo, **class** Account⟨b:**natural**⟩ {...}, onde **natural** é um tipo refinado que abrevia {x: **integer** | $x \geq 0$}, é a declaração de uma classe indexada que representa uma conta bancária, sendo o invariante introduzido pela variável de índice b. Neste contexto, Account representa uma família de classes, o que significa que os tipos Account⟨0⟩, Account⟨1⟩, ... obtidos por substituição de b por valores concretos, são tipos possíveis para instâncias desta família. Com classes indexadas, os programadores podem implementar uma conta bancária que, por definição, não suporta saldo negativo; código cliente é impedido em tempo de compilação do uso indevido dos métodos oferecidos pela classe.

Uma consequência da combinação de objectos mutáveis e tipos dependentes é a necessidade de garantir a correcção do sistema de tipos em caso de *aliasing*: se o objecto original muda, aceder ao *alias* pode produzir um resultado inesperado. O controlo deste tipo de situações é efectuado na linguagem DOL através de uma disciplina de tipos lineares, que impõe referências únicas a objectos com tipos variáveis. Em paralelo, coexiste na linguagem a categoria distinta de objectos partilhados, com tipos invariantes, mas que não pode subverter o sistema linear.

A linguagem oferece ainda suporte para herança, desde que o subtipo satisfaça as restrições de índice definidas pela superclasse, e as especificações herdadas tenham significado no contexto da subclasse. A título de exemplo, a classe Account pode ser estendida por uma subclasse que adiciona novos atributos e os relaciona com o saldo da superclasse. Para demonstrar a expressividade da linguagem através exemplos mais complexos, apresenta-se a implementação de uma árvore de pesquisa binária mutável, cujos

tipos são capazes de garantir o invariante associado a esta estrutura de dados.

Sistema Formal. Do ponto de vista teórico, a linguagem formal é semelhante à linguagem dos exemplos sem o "açúcar sintáctico". Para facilitar as provas, utiliza-se como ponto de partida a linguagem base de Gay et al. [2015], na qual os tipos de sessão são substituídos por tipos dependentes e à qual é adicionada herança. A linguagem definida nesta tese estende com índices a noção de tipos de classe do Java da forma $C\bar{i}$. Da teoria de tipos dependentes, é generalizado o tipo de função no tipo dependente $\Pi a : I.T$ no qual o tipo T pode depender do valor do argumento a do tipo de índice I, pertencente a um domínio de restrições. Do mesmo modo, a linguagem formal define o tipo soma dependente, que assume a forma $\Sigma a : I.T$. A linguagem inclui ainda tipos união $(T + U)$, que eliminam a necessidade de existir um valor null em DOL e de lidar com referências não definidas, além de permitirem agrupar classes independentes.

O sistema de tipos revela um conjunto de propriedades desejáveis, nomeadamente *subject reduction* e progresso, que, em conjunto, provam que os tipos dos objectos descrevem os seus valores em tempo de execução, que nunca existe mais do que uma referência para um objecto linear e que nenhum *alias* produz um valor de tipo inesperado.

Para além do sistema de tipos declarativo, é apresentado um sistema algorítmico que, com base nas ideias desenvolvidas por Dunfield and Krishnaswami [2016], fornece regras precisas para a instanciação de quantificadores. O sistema algorítmico utiliza ainda a técnica de tipificação bidireccional que combina a *síntese* de tipos com a *verificação* de termos de encontro a tipos conhecidos. O resultado é um algoritmo apto a ser implementado. É ainda apresentada a prova de correcção do sistema algorítmico em relação ao sistema declarativo.

Protótipo. Com base no sistema algorítmico, foi implementado um compilador, onde todos os exemplos apresentados na tese podem ser verificados, compilados e executados. A verificação de restrições é parte integrante da verificação de tipos, e completamente transparente para o programador. O *typechecker* recorre ao verificador de teoremas Z3 [de Moura and Bjørner, 2008] através de uma interface directa.

A implementação disponibiliza ainda um plugin para o Eclipse desenvolvido em Xtext [2017]. Embora generalizado no contexto de linguagens orientadas a objectos, o uso de ferramentas de desenvolvimento em linguagens com tipos dependentes é uma novidade. O ambiente de desenvolvimento inclui um assistente de código para programas DOL, a verificação de erros *on-the-fly*, e a geração de código-fonte na forma de classes Java que podem ser compiladas e executadas. O protótipo pode ser encontrado em `http://rss.di.fc.ul.pt/tools/dol/`.

Palavras-chave: Tipos dependentes, programação orientada a objectos, sistemas de tipos, objectos mutáveis

Acknowledgements

I am grateful to Vasco Vasconcelos for supervising my graduate studies, for always being available to discuss ideas, and for his wisdom, attention to detail and rigour. I thank Edwin Brady, Luís Caires, Jorge Sousa Pinto and Alysson Benassi for agreeing to be my thesis jury and for helpful and detailed comments.

Many thanks to Antónia Lopes, Alcides Fonseca and to members of LaSIGE (Universidade de Lisboa) for offering interesting perspectives on the work that led to this thesis. I thank Francisco Martins, César Santos and Fábio Ferreira for their help with making DOL available on the Internet.

Contents

Abstract i

Resumo iii

Acknowledgements vii

Contents ix

List of Figures xiii

List of Judgements xv

1 Introduction 1

1.1 Overview 4

1.2 Reader's Guide 6

2 DOL by Example 9

2.1 Indexed Classes 9

2.2 Bank Account 10

2.2.1 State Modifying Methods 11

2.2.2 Base Types and Literals 13

2.2.3 Controlled Aliasing 13

2.2.4 Inheritance and Subtyping 14

2.3 Binary Search Tree 17

2.3.1 BST Insertion 19

2.3.2 BST Deletion 21

2.4 DOL Code Example: Binary Search Tree 24

3 The DOL Language **27**
3.1 Syntax . . . 28
3.1.1 Types . . . 30
3.1.2 Terms . . . 30
3.1.3 Index Refinements . . . 31
3.1.4 Additional Syntax Not Available to Programmers . . . 31
3.2 Static Semantics . . . 34
3.2.1 Index Typing . . . 34
3.2.2 Index Substitution . . . 36
3.2.3 Kinding . . . 37
3.2.4 Subtyping . . . 38
3.2.5 Auxiliary Functions and Predicates . . . 40
3.2.6 Term Typing . . . 42
3.2.7 Program Typing . . . 47
3.2.8 Runtime Term Typing . . . 49
3.3 Operational Semantics . . . 50

4 Type Soundness **53**
4.1 State and Heap Typing . . . 54
4.2 Properties of Typing . . . 56
4.2.1 Inversion . . . 56
4.2.2 Exchange and Weakening . . . 57
4.2.3 Substitution . . . 58
4.2.4 Agreement . . . 60
4.2.5 Soundness of Function mtype . . . 61
4.3 Hiding Field Typings . . . 63
4.4 Properties of Evaluation Contexts . . . 64
4.5 Subject Reduction . . . 67
4.6 Progress . . . 81

5 Algorithmic Typechecking **87**
5.1 Algorithmic Type System . . . 87
5.1.1 Algorithmic Type Formation . . . 89
5.1.2 Quantifier Instantiation . . . 89
5.1.3 Algorithmic Subtyping . . . 92

5.1.4 Bidirectional Typechecking 94
5.2 Correctness of the Algorithmic System 98
5.2.1 Soundness 100
5.2.2 Completeness 103
5.3 Implementation 104
5.3.1 DOL IDE 106
5.3.2 Local Variables 106
5.3.3 Constraint Solving 107
5.3.4 Error Reporting 107

6 Related Work **109**
6.1 Dependent Types 109
6.1.1 Full-spectrum Dependent Types 110
6.1.2 Domain-specific Logics 110
6.1.3 Languages with Phase Separation 111
6.1.4 Other Forms of Dependent Types 112
6.2 Other Approaches to Program Verification 112
6.3 Affine Types 113
6.4 Ownership of Objects 114

7 Conclusion **115**

Bibliography **117**

List of Figures

2.1 A dependently-typed bank account 10
2.2 “Interfaces” of native Integer and Boolean classes 13
2.3 A class derived from Account 15
2.4 Classes that implement a dependently-typed binary search tree 17
2.5 The diagrammatic representation of type Node⟨2,5,8⟩ 18

3.1 Top-level syntax 28
3.2 Extended syntax 32
3.3 Formation rules for index types, propositions and contexts 34
3.4 Index subtyping rules 35
3.5 Typing rules for index terms 35
3.6 Typing rules for substitution formation 36
3.7 Kind and type formation rules 37
3.8 Subtyping rules 39
3.9 Auxiliary functions and predicates 41
3.10 Formation rules for object contexts 42
3.11 Typing rules for paths 42
3.12 Typing rules for terms in the top-level language 43
3.13 Typing rules for program formation 48
3.14 Typing rules for runtime terms 49
3.15 Reduction rules for sequenced object creation 50
3.16 Reduction rules for states 51
3.17 Two examples of reduction and typing 52

4.1 Dependency structure of the lemmas that support subject reduction 54
4.2 Typing rules for heaps and states 55

5.1 Algorithmic type formation rules 90

5.2 Rewrite rules to isolate an existential index variable 90
5.3 Index instantiation and equality rules 91
5.4 Algorithmic subtyping rules 93
5.5 Algorithmic formation rules for object contexts 94
5.6 Type synthesis rules for paths 94
5.7 Type synthesis and type checking rules for terms 96
5.8 Algorithmic typing rules for program formation 97
5.9 Two examples of error reporting in DOL 105

List of Judgements

$\Delta \vdash I$	Index type formation	Figure 3.3
$\Delta \vdash p$	Proposition formation	Figure 3.3
$\vdash \Delta$	Index context formation	Figure 3.3
$\Delta \vdash I <: J$	Index subtyping	Figure 3.4
$\Delta \vdash i : I$	Index typing	Figure 3.5
$\Delta_1 \vdash \Delta_2 : \theta$	Substitution formation	Figure 3.6
$\Delta \vdash K$	Kind formation	Figure 3.7
$\Delta \vdash T : K$	Type formation	Figure 3.7
$\Delta \vdash T <: U$	Subtyping	Figure 3.8
$\Delta \vdash \Gamma$	Context formation	Figure 3.10
$\Delta_1; \Gamma \vdash r : T \dashv \Delta_2$	Path typing	Figure 3.11
$\Delta_1; \Gamma_1 * r_1 \vdash t : T \dashv \Delta_2; \Gamma_2 * r_2$	Term typing	Figure 3.12
$\vdash_C M$	Method formation	Figure 3.13
$\Delta \vdash_T l : U$	Member formation	Figure 3.13
$\vdash L$	Class formation	Figure 3.13
$\vdash P$	Program formation	Figure 3.13
$\Delta_1; \Gamma_1 * r_1 \vdash \bar{t} : \bar{T} \dashv \Delta_2; \Gamma_2 * r_2$	Seq. term typing	Figure 3.14
$S_1 \longrightarrow S_2$	State reduction	Figure 3.16
$(h_1 * r, \mathsf{new}\ \bar{C}()) \longrightarrow (h_2 * r, \bar{o})$	State reduction for object creation	Figure 3.15
$\Delta; \Gamma \vdash h$	Heap formation	Figure 4.2
$\Delta_1; \Gamma_1 \vdash S : T \dashv \Delta_2; \Gamma_2 * r$	State typing	Figure 4.2
$\Delta_1; \Gamma_1 \vdash (h * r, \bar{t}) : \bar{T} \dashv \Delta_2; \Gamma_2 * r$	State typing for seq. terms	Figure 4.2
$\Delta \triangleright K$	Alg. kinding	Figure 5.1
$\Delta \triangleright T : K$	Alg. type formation	Figure 5.1
$\Delta_1 \vdash \hat{a} := i \dashv \Delta_2$	Quantifier instantiation	Figure 5.3
$\Delta_1 \vdash i \equiv j \dashv \Delta_2$	Index term equivalence	Figure 5.3
$\Delta_1 \vdash p_1 \equiv p_2 \dashv \Delta_2$	Proposition equivalence	Figure 5.3
$\Delta_1 \vdash \bar{i} \equiv \bar{j} \dashv \Delta_2$	Seq. index term equivalence	Figure 5.3
$\Delta_1 \vdash T <: U \dashv \Delta_2$	Alg. subtyping	Figure 5.4
$\Delta \triangleright \Gamma$	Alg. context formation	Figure 5.5

$\Delta_1; \Gamma \vdash r \uparrow T \dashv \Delta_2$	Type synthesis for paths	Figure 5.6
$\Delta_1; \Gamma_1 \vdash t \uparrow T \dashv \Delta_2; \Gamma_2$	Type synthesis for terms	Figure 5.7
$\Delta_1; \Gamma_1 \vdash t \downarrow T \dashv \Delta_2; \Gamma_2$	Type checking for terms	Figure 5.7
$\triangleright_C M$	Alg. method formation	Figure 5.8
$\Delta \triangleright_T l : U$	Alg. member formation	Figure 5.8
$\triangleright L$	Alg. class formation	Figure 5.8
$\triangleright P$	Alg. program formation	Figure 5.8

Chapter 1

Introduction

This thesis develops Dependent Object-oriented Language (DOL), a programming language featuring dependent types, mutable objects and class-based inheritance with subtyping. It provides a solid type system and an operational semantics for DOL, and describes a typechecking algorithm implemented in a *proof-of-concept prototype*.

Traditional type systems have been shown to be effective for verifying basic invariants, but are somehow limited in the kind of properties they can express. In object-oriented languages, in particular, programmers typically work in the context of informally and imprecisely specified behavioural aspects, most often only described in documentation. Many common errors, some of which may be observable at compile time, cannot be dealt with adequately by the static type system of a conventional programming language. Examples of such errors include dereferencing null, say, by taking an element of an empty data structure, or a method call that leaves an object in a state that violates the class invariant. Java-like languages will immediately halt at unsafe null values, yet this comes at the expense of runtime checks. Similarly, wrong assumptions about the state of an object may cause a runtime error, or not may not be detected at all, and simply return a wrong or unexpected result.

Type theory for programming has been concerned over the past decades with studying abstraction mechanisms of various sorts that describe program behaviours. Studied extensively in this context is the topic of dependent types, whose growing interest suggests that the underlying abstraction mechanism (dependencies on terms that assert additional properties about programs) may be useful when added to conventional programming languages. In particular, dependent types in the setting of object-oriented languages have the potential of filling the wide gap between simple invariants enforced by conventional type systems and more expressive (and complex) ones supported by verification techniques included in static checking tools, such as ESC [Leino, 2001] and Spec♯ [Barnett et al., 2005], and explored in the context of modular reasoning about multi-object invariants

and ownership relations [Müller, 2002, Barnett et al., 2004, Müller et al., 2006, Summers and Drossopoulou, 2010, Balzer and Gross, 2011].

The aim of this work is therefore to study a solution, embodied in DOL, that is a first step in the evolution of combining the scalability and modularity of object orientation with the safety provided by dependent types. The main challenge is to integrate in a unique system three very different features: dependent types, mutable objects, and class-based inheritance with subtyping.

Dependent types constrain types with values that specify intrinsic properties of programs. The practical utility of dependent types has been explored with great success over the past decades in the context of logic and functional languages [Augustsson, 1998, Xi and Pfenning, 1999, McBride, 2004, Flanagan, 2006, Bove et al., 2009], with implementations in functional programming languages such as Agda [Norell, 2007], DML [Xi, 2007], and Idris [Brady, 2013]. Less work has been done in an imperative setting [Xi, 2000, Condit et al., 2007], and none (that we know of) in class-based object-oriented programming with mutable objects. The reasons why dependent types benefit functional programming – increasing expressiveness and safety – are exactly the same why they may be useful in object-oriented programming. However, the added complexity and the challenge of combining dependent types and subtyping, a feature missing from the mentioned dependently-typed functional languages, explain why the combination of dependent types and object orientation is still largely unexplored ground in the context of type theory for programming.

Inspired by generics in Java-like languages that were introduced to enable types to be parameters to classes and methods, we introduce a restricted form of dependent types to enable special terms, called indices, to be parameters to classes and methods. The type List⟨n⟩ of lists of length n should look natural, even for an object-oriented programmer, being just another kind of polymorphism. Another example is the type Account⟨b⟩ of bank accounts with balance b. The programmer abstracts the class declaration on the Account's balance which in DOL becomes

```
class Account⟨b:natural⟩ {
  balance: Integer⟨b⟩
  ...
}
```

where **natural** is a subset type that abbreviates {x:**integer** | x ≥ 0}. The programmer then uses the index variable b to sharpen the type of fields and methods defined in Account, so that the typechecker will be able to enforce through types a behaviour that forbids overdrafts. We say that Account defines a family of classes representing bank accounts whose instances can have many types, including the concrete type Account⟨100⟩ obtained

by instantiating b with 100. Like generics, types in DOL support a variety of arguments. The difference is that the arguments to types in DOL are index terms that satisfy the specified constraints.

Mutable objects are closely linked to what objects are intended to be, that is, entities with private state and an interface that specifies the messages (or method calls) they are willing to accept. In certain states, some methods must not be available at the risk of violating object invariants. Dependent types allow the programmer to specify method availability using fine-grained method signatures. For an withdraw method in the Account class, the signature should be roughly "withdraw takes an Integer⟨m⟩ where $0 \leq m \leq b$ on any Account⟨b⟩ that becomes Account⟨$b - m$⟩". This enables the typechecker to statically track objects and any state change, guaranteeing that calling withdraw with an invalid argument leads to a type error caught by the compiler. For example, the following trivial client code in DOL will simply fail and show a type error:

```
acc := new Account(); // acc: Account⟨0⟩
acc.deposit(100);     // acc: Account⟨100⟩
acc.withdraw(105)     // Type error!
```

Class-based inheritance with subtyping is present in most object-oriented programming languages. This feature allows code that was originally written for a given class to be extended, and methods from the original class to be reused by subclasses, as well as offering the convenience of type substitutivity. The interplay of dependent types and inheritance with subtyping turns out to be substantially more challenging than the study of each feature taken separately. Still, our system can capture in types the discipline of the "safe substitutability principle" [Liskov and Wing, 1994]. For example, a subclass PlusAccount inherits the superclass fields and methods, adds its own, and redefines methods, as long as it satisfies the constraints of the superclass. We write this as follows:

```
class PlusAccount⟨s,c,b:natural⟩ extends Account⟨b⟩ {
  savings: Integer⟨s⟩  // two extra fields
  checking: Integer⟨c⟩
  ...
}
```

where the declaration **extends** Account⟨b⟩ allows the subtype to inherit the Account's only field balance, and all its methods (constructor excluded). The two new fields that describe two portions of the balance can be related to the inherited field in method signatures in the subclass, as we will see in the examples.

In contrast to other dependently-typed object languages such as DOT [Amin, 2016, Rompf and Amin, 2016], a calculus that provides a type-theoretic foundation for Scala, our language formalises mutable objects and inheritance. As we will elaborate, DOL's

type system is able to track mutation through the types of fields that may change throughout the life of an object. On the other hand, mutable state is not modelled in DOT. Instead, it is proposed as an extension to the operational semantics by providing a subtype of the type in the store typing, which remains invariant under reads and updates.

1.1 Overview

In this thesis, we show how to make the three features – dependent types, mutable objects and class-based inheritance with subtyping – coexist in object-oriented programming. We give an overview and rationale of the novelties and strategies devised for DOL, before getting into the details of the examples, typing rules and algorithm.

DOL. There are two basic approaches to the topic of introducing dependent types, and any type theory for that matter, in programming: starting with a type theory and turning it into a programming language, or starting with a language and adding type theoretic features that make it safer. This thesis takes the latter approach. DOL is an object-oriented language designed along the lines of Gay et al. [2015], with session types removed. The language, meant to support mutable state, allows a natural integration of dependent types in the form of value parameters (indices) that are used for specification and correctness – specification in the sense that dependent types in DOL give some indication of the valid state of objects and the usage intended by the programmer; correctness in that programs in DOL do not encounter runtime errors.

A difficult question is what kind of properties should indices be allowed to capture. Constraints have to be enforced by the type system, and so the requirements of decidability and efficiency of typechecking must be taken into account. It turns out that restricting the domain of the constraint language is a suitable choice to render a type system decidable. Drawing inspiration from DML [Xi and Pfenning, 1999], we use index refinements to achieve a separation between index terms and computation terms, refining class types with indices drawn from a decidable constraint domain. We give examples using the integer and boolean domains, which are by far the most explored constraint theories. A key feature is that our language does not require the programmer to prove equations between indices issued by types, but instead relies on an external constraint solver to compute them efficiently during typechecking.

Given that objects are mutable, we must allow their types to vary throughout the program so that the compiler can record changes and detect errors by typecheking. The type system is sound, relying on indices but also on method signatures in which the programmer describes the input and output types of the receiver. Moreover, aliasing in a language

that allows types to change can result in a program "getting stuck": if an aliased object changes, reading from the alias will produce an unexpected result. In DOL, some control of aliasing is handled by the type system via a linear discipline that enforces unique references to type varying objects. A distinct category of type invariant, shared objects, is allowed to coexist, but cannot subvert the linear system.

DOL also provides support for single class inheritance as long as the subtype satisfies the index constraints defined by its supertype, and the inherited specifications remain meaningful in the context of the subclass.

At this point, it may be pointed out that the formulation of DOL depends heavily on formalisms that compromise the language usability, namely by means of an index language with limited expressivity and of linear control of objects. To this, we argue that DOL is flexible for a dependently-typed language: programmers may start with standard types, writing code in an imperative style, and add more type information so as to gain additional guarantees. Towards a full programming language, we could easily scale to more expressive index languages at the cost of decidable typechecking, and we could use existing mechanisms for more flexible control of aliasing. Alternatives to our approach are discussed in related work.

To attest the relevance of our language, we have implemented a prototype compiler for DOL that includes a plugin for the Eclipse IDE, a development tool widely used in the context of object-oriented languages but still new for dependently-typed languages. This additional material is available at `http://rss.di.fc.ul.pt/tools/dol/`. All the examples in this thesis have been run, compiled and executed in the latest version of the prototype at the time of writing.

Formal System. DOL's core language is the same as the language from the examples without the syntactic sugar. It extends with indices the Java notion of class types that take the form of $C\bar{i}$. From the dependent type theory, it generalises the simple function space to a dependent function space $\Pi a : I.T$ where the result type T can depend on the value of the argument a, restricted to special terms of index type I. Similarly, dependent sum types, written $\Sigma a : I.T$, generalise ordinary product types restricted to some constraint domain. The language also includes union types of the form $T + U$ that eliminate the need of an unsafe null value, and can be used to group independently developed classes.

The type system studied in this thesis involves a large number of judgments from where emerges a stratification: terms in the object language have types that contain indices; indices also have types (often called sorts in the literature); indices and terms in the object language are distinct. We include subtyping in our typing rules induced by inheritance and quantifier instantiation, but this combination significantly complicates the

development of the meta-theory. We prove that the system possesses desirable properties, such as type soundness, expressed via subject reduction and progress. As a result, static typing guarantees runtime safety properties, namely that the types of objects describe their runtime values, that there never exists more than one reference to a linear object, and that all aliasing never produce a value of unexpected type.

While the declarative system is natural to explain, it is difficult to implement, and some features are hard to analyse directly, because of the use of existential and universal types. We formulate an algorithmic type system for DOL that gives a specification for quantifier instantiation based on the techniques of Dunfield and Krishnaswami [2013, 2016], and applies bidirectional typechecking [Pierce and Turner, 2000] which distinguishes the two distinct modes of type *synthesis* and *checking* against a known type. The algorithm is both simple to understand and to implement. We prove that typechecking is sound and complete with respect to the declarative system.

Publications. The following are the source of improved and adapted material (to the context of dependent types in the case of the second reference) presented in this thesis:

- Joana Campos and Vasco T. Vasconcelos. Imperative objects with dependent types. In *Formal Techniques for Java-like Programs*, pages 2:1–2:6, 2015

- Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Denilou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. Behavioral types in programming languages. *Foundations and Trends in Programming Languages*, 3(2-3):95–230, 2016

- Joana Campos and Vasco T. Vasconcelos. Programming with mutable objects and dependent types. In *INForum*. Atas do Oitavo Simpósio de Informática, 2016

1.2 Reader's Guide

Chapter 2 motivates DOL using programming examples that describe the language from the user's perspective. A large example of a binary search tree implementation, modified in place, is also included.

Chapter 3 presents DOL from the perspective of its type theoretic features, giving the static and operational semantics.

Chapter 4 describes several properties of DOL, and provides the proof of soundness via subject reduction and progress theorems

Chapter 5 defines a set of algorithmic typing rules, and proves that typechecking is sound and complete with respect to the declarative system.

Chapter 6 discusses related work, focusing on dependent types and pointing extensions to the type system of DOL.

The reader should have a solid grasp of type theory as applied to programming languages [Pierce, 2002].

Chapter 2

DOL by Example

This chapter gives a description of the language from the programmer's perspective, while subsequent chapters provide the technical details.

Chapter Outline. This chapter consists of three sections:

- Section 2.1 explains the overall concept of object-oriented programming with dependent types in the form of indexed classes.
- Section 2.2 motivates the interplay between classes and indices using a bank account class that has mutable state and a subclass which demonstrates DOL's support for inheritance.
- Section 2.3 presents an implementation of insertion and deletion for binary search trees in an imperative style that relies on types to ensure the binary search tree invariant.

2.1 Indexed Classes

A class in DOL is declared just like any other class in a Java-like language, except that index variables may be introduced in the header and be used within the class to constrain member types. The class body contains fields and methods, including a constructor method named `init`. Like Java, DOL supports single class inheritance using the optional **`extends`** declaration. If omitted, the class is derived from the default superclass `Top`, a concrete class which has no fields or methods, except for the constructor. Methods may also be indexed, introducing typed variables that may be used to constrain types in method signatures.

```
1  class Account⟨b:natural⟩ {
2    balance: Integer⟨b⟩ // the only field
3
4    init(): Account⟨0⟩ =
5      balance := 0
6
7    ⟨m:natural⟩
8    [Account⟨b⟩ ⇝ ⟨b+m⟩]
9    deposit(amount: Integer⟨m⟩) =
10     balance := balance + amount
11
12   ⟨m:natural{m≤b}⟩
13   [Account⟨b⟩ ⇝ ⟨b−m⟩]
14   withdraw(amount: Integer⟨m⟩) =
15     balance := balance − amount
16
17   getBalance(): Integer⟨b⟩ =
18     balance // return balance
19 }
```

Figure 2.1: A dependently-typed bank account

As noted in the previous chapter, we follow a style of type dependencies whereby index terms are used only to constrain types, being syntactically separate from the language of objects. Thus, types in DOL cannot depend directly on the value of a field, for example, neither can an index variable be used as a parameter to a method. Instead, index terms are the only values that may inhabit types, having no place in the separate world of computations.

2.2 Bank Account

In Figure 2.1 we define the indexed class Account which includes the usual three methods – deposit, withdraw and getBalance. The index variable b of type **natural** is declared at the beginning of the class as a parameter in angle brackets. Programmers gain additional safety guarantees with the possibility of indexing classes and using DOL's richer type system. Notice that if we omit the extra type annotations in the example, we get plain Java-like code, with Account being simply a class type.

However, when indexed, we regard the class name Account as denoting a family of classes. Instantiations, or concrete classes, represent bank accounts that cannot be overdrawn, and may have many types, namely Account ⟨0⟩, Account⟨1⟩, ... where the occurrence of the index variable introduced in the class header is replaced by the corresponding value (an index term). The special init method behaves as a typical constructor that

initialises fields, creating a fresh object assigned the proper (or concrete) type Account⟨0⟩ (line 4). For example, whenever init is invoked as in

```
acc := new Account()   // acc: Account⟨0⟩
```

no actual index parameter is passed to the constructor, since the compiler will be able to read the type Account⟨0⟩ from the method signature. As we will see below in detail, one consequence of instantiations at different types is that the compiler must track state changes throughout the program. The preceding example may continue as follows:

```
acc := new Account(); // acc: Account⟨0⟩
acc.deposit(100);     // acc: Account⟨100⟩
acc.withdraw(30)      // acc: Account⟨70⟩
```

The special index variable b, introduced in the class header, is used in the declared type of field balance – Integer⟨b⟩ – representing the field's runtime value, enforced at compile time to be a natural number. The Account class is mutable, so internally the typechecker replaces occurrences of the index variable b by the corresponding value, with the field type becoming Integer⟨0⟩ at object creation, changing to Integer⟨100⟩ after the call to the deposit method in the snippet above, and then to Integer⟨70⟩ after the call to method withdraw. Note that state is indirectly exposed in types through indices, but fields are always *private* to a class, even if we do not use the corresponding keyword.

2.2.1 State Modifying Methods

We can give indexed signatures to methods that are defined in indexed classes. For example, the withdraw method is as follows:

```
⟨m: natural{m≤b}⟩
[Account⟨b⟩ ⇝ ⟨b−m⟩]
withdraw(amount: Integer⟨m⟩) =
  balance := balance − amount
```

The method must be invoked on a receiver of type Account⟨b⟩, accepts an amount of type Integer⟨m⟩, modifies the type of the receiver from Account⟨b⟩ to Account⟨b – m⟩, and does so for any amount that is a natural number smaller or equal to the balance. This is indicated by the declared index variable in angle brackets at the beginning of the method signature. The method type, written $\Pi m : \{x\colon \text{integer} \mid 0 \leq x \leq b\}.T$ in the formal language, is a universal type that binds the index variable m to a type T (where T represents the types of the implicit and explicit parameters and the return type from the example), so that we can mention m in T. The scope of the index variable m is therefore local; it may appear in the method signature, but not outside. The type [Account⟨b⟩ ⇝ ⟨b – m⟩], read "Account⟨b⟩ becomes Account⟨b – m⟩", is an abbreviation for a pair of types. The first type is seen as the

input type of the (implicit) receiver and the second one is viewed as its output type. This pair of types induces a kind of usage protocol for withdrawing. In other words, an instance of Account may call withdraw only if passing an amount that does not exceed its balance. If this condition is satisfied, the assignment is evaluated and the balance field is updated, internally becoming Integer⟨b − m⟩. As mentioned earlier, the compiler tracks changes to an object through its "private" fields, with state being somewhat exposed through indices – notice that the operation is reflected in the output type Account⟨b − m⟩. Finally, when a method does not explicitly declare a return type, the typechecker assumes the supertype Top.

To illustrate the precision of the types in DOL, here is a variant of the preceding example, changed by adding a second call to method withdraw that violates the object invariant:

```
acc := new Account();   // acc: Account⟨0⟩
acc.deposit(100);       // acc: Account⟨100⟩
acc.withdraw(70);       // acc: Account⟨30⟩
acc.withdraw(50)        // Type error: 50 > 30
```

No doubt, the second and third lines are fine (because $0 \leq 70 \leq 100$), but the last line is in error (because $50 > 30$). So, the typechecker keeps track of the exact balance at each point in the program, and any attempt to break the object invariant is promptly detected at compile time. It should now be clear the need for DOL's typechecker to record and track state changes in the typing context. Without this feature, dependent types would be useless in a language with mutable state.

Both deposit and withdraw are examples of methods that change state, which we sometimes call *type varying* methods. In DOL, these methods must explicitly declare the input and output types of their implicit receivers. On the contrary, in *type invariant* methods, that is, methods whose input and output types coincide, receiver types may be omitted. Consider the getBalance method:

```
getBalance(): Integer⟨b⟩ =
  balance // return balance
```

The method does not accept parameters and returns an integer of type Integer⟨b⟩. Because the receiver type is omitted, it can be inferred by instantiating the class family with index variables declared in the header. In this case, type inference yields the following type:

```
[Account⟨b⟩ ⇝ ⟨b⟩]
getBalance(): Integer⟨b⟩ =
  balance // return balance
```

```
1  class Integer⟨i:integer⟩ {
2    init(): Integer⟨0⟩
3    ⟨j:integer⟩+(value: Integer⟨j⟩): Integer⟨i+j⟩
4    ⟨j:integer⟩−(value: Integer⟨j⟩): Integer⟨i−j⟩
5    ⟨j:integer⟩≤(value: Integer⟨j⟩): Boolean⟨i≤j⟩
6    ⟨j:integer⟩≥(value: Integer⟨j⟩): Boolean⟨i≥j⟩
7    ...
8  }
9  class Boolean⟨b:boolean⟩ {
10   init(): Boolean⟨false⟩
11   ⟨a:boolean⟩&&(value: Boolean⟨a⟩): Boolean⟨b∧a⟩
12   ⟨a:boolean⟩||(value: Boolean⟨a⟩): Boolean⟨b∨a⟩
13   ...
14 }
```

Figure 2.2: "Interfaces" of native Integer and Boolean classes (an excerpt)

2.2.2 Base Types and Literals

Constants and operators are used in the programmer's language only to make arithmetic and logic operations look more familiar, since they are not part of DOL's core language. Constant 100 can be used as an argument as follows:

```
acc.deposit(100)
```

The subtraction operator appears in the body of the withdraw method, namely

```
balance := balance − amount
```

In fact, constants and operators are desugared into object references and method calls in the core language. Formally, Integer and Boolean, implemented natively, are families of classes. We give in Figure 2.2 some of the signatures defined in the "interfaces" provided by DOL. Each desugared object of a primitive class is assigned a singleton type, with the constants used in the examples representing the values on which the types depend. Technically, the argument 100 used in an earlier example is an object reference of type Integer⟨100⟩, obtained by creating a new location, and subtraction is translated into the call balance.minus(amount) before typecheking.

2.2.3 Controlled Aliasing

Aliasing is part of what makes mutable objects useful in programming. However, shared state can be tricky to handle in a type system such as that of DOL, where the type of a variable may no longer be a fixed class type; instead, it may be a (dependent) type that changes throughout the program. In DOL, the potential sources of aliasing problems are assignment and parameter passing. Consider assignment and what would happen if an object reference to a bank account object was allowed to be freely shared:

```
alias := acc;          // acc: Account⟨70⟩, alias: Account⟨70⟩
acc.withdraw(50);      // acc: Account⟨20⟩, alias: Account⟨70⟩
alias.withdraw(30)     // acc: Account⟨??⟩, alias: Account⟨??⟩
```

After the first call to withdraw, variable acc acquires type Account⟨20⟩. However, since the compiler does not keep track of the number of references to a given object, variable alias is still at type Account⟨70⟩, even though the first line makes both variables reference the same object. So, the second call to withdraw (the one made on variable alias) will break the object invariant. To deal with aliasing, we adopt a solution that uses linear control of those objects defined by type varying classes. We say that a class is type varying when at least one of its methods is type varying, giving different input and output types to its receiver. Because the Account class is type varying as per methods deposit and withdraw, the type system forbids creating aliases of instances of Account. Consider now how DOL's typechecker handles the preceding snippet:

```
alias := acc;          // alias: Account⟨70⟩
acc.withdraw(50);      // Type error: acc has been consumed!
alias.withdraw(30)     // alias: Account⟨40⟩
```

Instead of creating an alias, the assignment "consumes" variable acc, removing it from the typing context; hence, the call to withdraw in the second line is not allowed, with alias being the only variable available in the typing context.

On the other hand, we say that a class is type invariant when its methods are type invariant, i.e. when the input and output types coincide (or are omitted) in all methods. Linearity is not imposed in this case, allowing objects to be freely shared. The native Integer and Boolean provide two examples of such classes. Since each new assignment creates a new location, instances of these classes carry their types unchanged irrespective of being accessed or aliased.

Besides an index language with limited expressivity (restricted to the integer and boolean domains), a linear type system is the second concession that can be pointed out to the type system of DOL. Still, as we will see later in this chapter, we can implement (advanced) examples in which the use of linearity is not a problem. We believe linearity to be an orthogonal question to this thesis' contribution of designing a dependently typed object-oriented language. As mentioned in the previous chapter, we discuss existing alternatives in Chapter 6, guiding future work towards more flexible solutions.

2.2.4 Inheritance and Subtyping

The example we now present, adapted from JML [Dhara and Leavens, 1996, Leavens et al., 2006], illustrates how DOL can achieve the "safe substitutability principle" [Liskov

```
1  class PlusAccount⟨s,c,b:natural⟩ extends Account⟨b⟩ {
2    savings: Integer⟨s⟩   // two extra fields
3    checking: Integer⟨c⟩
4
5    init(): PlusAccount⟨0,0,0⟩ =
6      balance, savings, checking := 0
7
8    ⟨m:natural⟩
9    [PlusAccount⟨s,c,b⟩ ⇝ ⟨s+m,c,b+m⟩]
10   deposit(amount: Integer⟨m⟩) =
11     super.deposit(amount);
12     savings := savings + amount
13
14   ⟨m:natural⟩
15   [PlusAccount⟨s,c,b⟩ ⇝ ⟨s,c+m,b+m⟩]
16   deposit2Checking(amount: Integer⟨m⟩) =
17     super.deposit(amount);
18     checking := checking + amount
19
20   ⟨m:natural{m≤b ∧ b=s+c}⟩
21   [PlusAccount⟨s,c,b⟩ ⇝ ⟨max(s−m,0),min(c,c−m+s),b−m⟩]
22   withdraw(amount: Integer⟨m⟩) =
23     super.withdraw(amount);
24     if amount ≤ savings {
25       savings := savings − amount
26     } else {
27       checking := checking − amount + savings;
28       savings := 0
29     }
30 }
```

Figure 2.3: A class derived from Account

and Wing, 1994] via indexed types.

The PlusAccount class given in Figure 2.3 extends Account. As usual in Java-like languages, one type is a subtype of another if they are related by the **extends** declaration. In DOL, this also means that the invariant of the superclass is required to hold in this relation. By declaring

```
class PlusAccount⟨s,c,b:natural⟩ extends Account⟨b⟩ { ... }
```

we make the subtype inherit the Account's only field, as well as all of its methods (except the constructor). In PlusAccount, we declare two additional index variables, s and c, and use them to constrain two new fields, savings and checking, that describe two portions of the balance.

We then use the trick proposed by Dhara and Leavens [1996], Leavens et al. [2006] of relating the two new fields in the subclass with the superclass's field. We adapt this device to our requirements by enforcing that b=s+c via method signatures. We override the deposit method as follows:

```
⟨m: natural⟩
[PlusAccount⟨s,c,b⟩ ⇝ ⟨s+m,c,b+m⟩]
deposit(amount: Integer⟨m⟩) =
  super.deposit(amount);
  savings := savings + amount
```

The deposit method adds the amount to the account's balance by calling the superclass method, and then adds it to the savings field. A new method deposit2Checking (lines 14–18) also adds the given amount to the checking field. The withdraw method must be redefined in order to take out the amount from each balance portion as follows:

```
⟨m: natural{m≤b ∧ b=s+c}⟩
[PlusAccount⟨s,c,b⟩ ⇝ ⟨max(s−m,0),min(c,c−m+s),b−m⟩]
withdraw(amount: Integer⟨m⟩) =
  super.withdraw(amount);
  if amount ≤ savings {
    savings := savings − amount
  } else {
    checking := checking − amount + savings;
    savings := 0
  }
```

In order to determine if the output type is valid, DOL's typechecker gives the index equations issued by types to the external constraint solver, and asks if they hold.

Common mistakes that violate the (inherited) invariant are readily detected. For example, the code

```
⟨m: natural{m≤s}⟩
[PlusAccount⟨s,c,b⟩ ⇝ ⟨max(s−m,0),min(c,c−m+s),b−m⟩]
withdraw(amount: Integer⟨m⟩) = ...
```

yields a type error, since a subtype cannot accept a *stronger* requirement, that is, it cannot accept less arguments as valid [Liskov and Wing, 1994] (it should be clear that the constraint $m \leq s$ does not imply $m \leq b$ under the assumption that $b=s+c$).

A subtler type error is found in the following variant:

```
⟨m: natural{m≤b}⟩
[PlusAccount⟨s,c,b⟩ ⇝ ⟨max(s−m,0),min(c,c−m+s),b−m⟩]
withdraw(amount: Integer⟨m⟩) = ...
```

The problem here is that the constraint $m \leq b$ does not relate the value of amount with the two portions of the balance, unlike the indices in the output type. Specifically, the index refinement does not provide enough evidence that allows the typechecker to conclude, after interaction with the solver, that the index term **min**(c,c − m + s) is a natural number that can safely replace the index variable c introduced in the class header.

```
class BST⟨l,u:integer⟩ {
  root: Nil + Node⟨l,k,u⟩ where k:integer{l≤k≤u} // the only field

  init(): BST⟨2,1⟩ =
    root := new Nil()

  ⟨v:integer⟩
  [BST⟨l,u⟩ ⇝ ⟨min(l,v),max(u,v)⟩]
  insert(value: Integer⟨v⟩) = ...

  ⟨v:integer⟩
  remove(value: Integer⟨v⟩) = ...
}
class Nil {
  init(): Nil = skip
}
class Node⟨l,k,u:integer{l≤k≤u}⟩ {
  key: Integer⟨k⟩ // fields
  left: Nil + Node⟨l,k1,u1⟩ where k1,u1:integer{l≤k1≤u1≤k}
  right: Nil + Node⟨l1,k1,u⟩ where l1,k1:integer{k≤l1≤k1≤u}

  ⟨v:integer⟩
  init(value: Integer⟨v⟩): Node⟨v,v,v⟩ =
    key, left, right := value, new Nil(), new Nil()

  ⟨v:integer⟩
  [Node⟨l,k,u⟩ ⇝ ⟨min(l,v),k,max(u,v)⟩]
  add(value: Integer⟨v⟩) = ...

  ⟨v:integer⟩
  [Node⟨l,k,u⟩ ⇝ ⟨l,k1,u⟩ where k1:integer{l≤k1≤u}]
  deleteChild(value: Integer⟨v⟩) = ...
}
```

Figure 2.4: Classes that implement a dependently-typed binary search tree

2.3 Binary Search Tree

Binary search trees can naturally be described by the discipline of dependent types. In fact, there has been much research on implementing ordered data structures in the context of dependently-typed functional programming languages [Xi, 1998, Dunfield, 2007, Knowles, 2014, McBride, 2014].

We use types to enforce statically the standard binary search tree property: a binary search tree is either empty or nonempty in which case it has two subtrees that are binary search trees, and the key in the root node of the binary search tree is greater than all the keys appearing in its left subtree and smaller than all the keys appearing in its right subtree. This example shows that our type system can be precise and expressive while implementations remain as usual. The effort of programming in DOL is essentially to

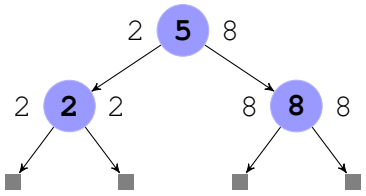


Figure 2.5: The diagrammatic representation of an object of type Node⟨2,5,8⟩ where labels at each tree node denote the smallest and greatest keys appearing in the tree

come up with the right type.

We implement the binary search tree in an imperative style, allowing subtrees to be modified *in place*. The complete code can be found at the end of this chapter. In Figure 2.4, we show the types, defining BST as the "public" family of classes that creates and manages both empty and nonempty trees using Nil (a proper class) and Node (a family of classes). Our binary search tree contains integer numbers included in a *loose* pair of bounds ⟨l,u:**integer**⟩ declared in the header of BST that can be used to define an *interval* [l,u]. Any element in a tree can find a place within the minimum (l) and maximum (u) keys.

Nullable references are commonly used in Java-like languages to build data structures, but have been a recurring source of error (Hoare's billion dollar mistake [Hoare, 2009]). They are the cause of a variety of bugs, mainly because null can mean a missing value, failure, success, it can mean almost anything. While many approaches have been proposed [Fähndrich and Leino, 2003, Cielecki et al., 2006, Chalin and James, 2007], DOL provides an elegant solution using union types (cf. Igarashi and Nagira [2006]) that enable programmers to build imperative linked data structures in a null-free style – object references are, after all, the only values in DOL. Union types (denoted by T + U) represent objects that can be of any of the specified types. Note that the lack of null in DOL means that every variable must be initialised, and that every variable of a union type must be analysed by way of a **case** construct before being used.

So, class BST has a single field root that is either Nil or Node. We declare it as follows:

```
root: Nil + Node⟨l,k,u⟩ where k:integer{l ≤ k ≤ u}
```

A dependent existential quantified constructor (denoted by **where** ...) is used to keep track of the key, hidden in the field type, so that the binary search tree invariant can be maintained. We write it as a dependent sum type in the formal language of the form $\Sigma k : \{x : \mathsf{integer} \mid l \leq x \leq u\}.\mathsf{Node}\ l\ k\ u$. Notice that this type conforms to the constraint (l≤k≤u) issued by the signature of class Node that ensures the binary search ordered invariant.

The special init method (lines 4–5) creates an empty binary search tree to which

we give the type BST⟨2,1⟩, making root an instance of Nil. As we will see below, when inserting a value in an empty tree, we replace an instance of Nil with an instance of Node with no children (a leaf). Then, when inserting in a nonempty tree, we recursively push the requirements of data inward, requiring that the value at each node falls within the interval [l,u]. For example, a type BST⟨2,8⟩ may represent the binary search tree whose root of type Node⟨2,5,8⟩ (depicted in Figure 2.5) issues the minimum and maximum keys outward. The value 5 stored in the root is not exposed, but is internally constrained by the tree bounds.

The Node class defines a field key, which holds the node value, and fields left and right that may represent the two subtrees. We use union and existential types, again pushing the data requirements inward to the types of the left and right fields. For example, the type of the left field defined as

```
left: Nil + Node⟨l,k1,u1⟩ where k1,u1:integer{l ≤ k1 ≤ u1 ≤ k}
```

enforces the fact that all the values appearing in the left subtree must be in the interval defined by [l,k], so that the binary search tree invariant can be maintained. Similarly, the type of the right field (line 20) enforces the fact that all the values appearing in this subtree must be included in [k,u]. The init method (lines 22–24) creates a leaf by accepting an integer value to be stored in the key field, making both left and right instances of Nil. By definition, leaf nodes are such that l=k=u. So, for example, Node⟨2,2,2⟩ may be the type of the left subtree (a leaf) in Figure 2.5.

2.3.1 BST Insertion

We now define the insert method in the BST class, which takes as argument an integer value and provides a useful demonstration of a case discrimination construct that looks at the type structure of the root field:

```
⟨v:integer⟩
[BST⟨l,u⟩ ⇝ ⟨min(l,v),max(u,v)⟩]
insert(value: Integer⟨v⟩) =
  case root {
    Nil ⇒ root := new Node(value)
    Node ⇒ root.add(value)
  }
```

The Node class implements the main insertion algorithm. Its method add takes an argument similar to the one above, and also uses a case discrimination construct to look at the type structure of the two subtree fields:

```
⟨v:integer⟩
[Node⟨l,k,u⟩ ⇝ ⟨min(l,v),k,max(u,v)⟩]
```

```
add(value: Integer⟨v⟩) =
  if value < key {
    case left {
      Nil ⇒ left := new Node(value)
      Node ⇒ left.add(value)
    }
  } else if value > key {
    case right {
      Nil ⇒ right := new Node(value)
      Node ⇒ right.add(value)
    }
  } else { // value == key
    skip
  }
```

We add an element to the tree by comparing the value to the key stored at each node, and recursively descending into the appropriate subtree until a leaf is reached that allows adding the new node.

The precise types given to the BST and Node classes allow the typechecker to detect a number of common programming errors. For example, the compiler will report a type error if we try to call the add method as follows:

```
⟨v:integer⟩
[BST⟨l,u⟩ ⇝ ⟨min(l,v),max(u,v)⟩]
insert(value: Integer⟨v⟩) =
  root.add(value)
```

Because root declares a union type, we cannot call a method directly on it; first, we must use a **case** construct to analyse the type of the field and discover whether the object is an instance of Nil or Node, noting that, at each branch, the typechecker requires that root be bound to only one of the types which are subtypes of the union of types. This guarantees that either branch is taken and its execution succeeds.

Similarly, the compiler will object to the wrong conditional test below:

```
⟨v:integer⟩
[Node⟨l,k,u⟩ ⇝ ⟨min(l,v),k,max(u,v)⟩]
add(value: Integer⟨v⟩) =
  if value > key {
    case left {
      Nil ⇒ left := new Node(value)
      ...
```

Here, the compiler will report inconsistent constraints. The **case** construct is correctly used to find out that left is a Nil. Then, the assignment changes the type of the left field to Node⟨v,v,v⟩ (which is the type given to it by init). However, the compiler assumes v > k

from the conditional test, after which will not be able to assert $v \leq k$ issued from the new type of the left field. Recall the constraints on the declared type of left, requiring its value be left-bounded by the minimum key l (which has become v) and right-bounded by value k (known to be also v) such that $l \leq k$. Again, DOL relies on the external constraint solver to statically verify that the specified constraints hold.

2.3.2 BST Deletion

Deletion from the binary search tree may involve removing a key not only from the tree's leaf nodes but also from an interior node, which requires some sort of rearrangement of the tree structure. Moreover, unlike insertion, in which **min** and **max** could be used to issue the new value's standing vis-à-vis the minimum and maximum keys existing in the tree, deletion delivers the same binary search tree where the new minimum or maximum may be hidden in the subtrees.

However, we can still establish that the implementation is type correct, ensuring that the tree after deletion is within the bounds, no matter where the key removal occurs (from a fringe or the middle of the tree). The remove method in the BST class requires the value v about to be removed, if it exists, to be within the tree bounds ($l \leq v \leq u$). The method is implemented as usual:

```
⟨v:integer⟩
remove(value: Integer⟨v⟩) =
  case root {
    Nil ⇒ skip
    Node ⇒
      if root.isLeaf(value) {
        root := new Nil()
      } else {
        root.deleteChild(value)
      }
  }
```

We check if root is a Nil, in which case we are done as the tree is empty. Otherwise, root is a Node, and we determine if it is a leaf holding the value to be removed. Deletion is accomplished either by setting root to a fresh Nil object, or by calling the recursive method deleteChild on it in order to run the algorithm. As we will see, the deleteChild method may change locally the type of the root field. However, the ordering constraint is hidden in the field type, which means that the type of the receiver as viewed from outside does not change, so we simply omit it in the remove method signature.

The task of deleting a child can basically be divided in three stages: finding the key to remove, looking for a minimum key (guaranteed via types to be within bounds) to replace

the one being deleted, and applying deleteChild to remove the node whose key we took. We encode this as follows:

```
⟨v:integer⟩
[Node⟨l,k,u⟩ ⇝ ⟨l,k1,u⟩ where k1:integer{l ≤ k1 ≤ u}]
deleteChild(value: Integer⟨v⟩) =
  if value < key {
    case left {
      Nil ⇒ skip
      Node ⇒
        if left.isLeaf(value) {
          left := new Nil()
        } else {
          left.deleteChild(value)
        }
    }
  } else if value > key {
    case right {
      Nil ⇒ skip
      Node ⇒
        if right.isLeaf(value) {
          right := new Nil()
        } else {
          right.deleteChild(value)
        }
    }
  } else { // value == key
    var newKey := value;
    case right {
      Nil ⇒
        case left {
          Nil ⇒ skip
          Node ⇒ newKey := left.getMaxKey()
        }
      Node ⇒ newKey := right.getMinKey()
    };
    deleteChild(newKey);
    key := newKey
  }
```

In a manner similar to add, we recursively search for the key to remove. The easy case, when the child node to be removed is a leaf, is handled by setting the child node to Nil. Otherwise, when the node that holds the value is found in the middle of the tree, we hoist a minimum key using methods getMinKey and getMaxKey defined at the end of the chapter, which work inside the subtrees of the node whose key is being deleted. Then, we remove

the node whose key we found by calling deleteChild, and replace the value being deleted with the found key. Notice that the type assigned at each recursive call represents ordering evidence around the deleted element. As with insertion, indexing with bounds allows the typechecker to ensure the binary search tree invariant.

2.4 DOL Code Example: Binary Search Tree

```
class BST⟨l,u:integer⟩ {
  root: Nil + Node⟨l,k,u⟩ where k:integer{l≤k≤u} // the only field

  init(): BST⟨2,1⟩ =
    root := new Nil()

  ⟨v:integer⟩
  [BST⟨l,u⟩ ⇝ ⟨min(l,v),max(u,v)⟩]
  insert(value: Integer⟨v⟩) =
    case root {
      Nil ⇒ root := new Node(value)
      Node ⇒ root.add(value)
    }

  ⟨v:integer⟩
  remove(value: Integer⟨v⟩) =
    case root {
      Nil ⇒ skip
      Node ⇒
        if root.isLeaf(value) {
          root := new Nil()
        } else {
          root.deleteChild(value)
        }
    }

  ⟨v:integer⟩
  search(value: Integer⟨v⟩): Boolean⟨b⟩ where b:boolean =
    case root {
      Nil ⇒ false
      Node ⇒ root.contains(value)
    }
}
class Nil {
  init(): Nil = skip
}
class Node⟨l,k,u:integer{l≤k≤u}⟩ {
  key: Integer⟨k⟩ // fields
  left: Nil + Node⟨l,k1,u1⟩ where k1,u1:integer{l≤k1≤u1≤k}
  right: Nil + Node⟨l1,k1,u⟩ where l1,k1:integer{k≤l1≤k1≤u}
```

```
41
42  ⟨v:integer⟩
43  init(value: Integer⟨v⟩): Node⟨v,v,v⟩ =
44    key := value;
45    left, right := new Nil(), new Nil()
46
47  ⟨v:integer⟩
48  [Node⟨l,k,u⟩ ⇝ ⟨min(l,v),k,max(u,v)⟩]
49  add(value: Integer⟨v⟩) =
50    if value < key {
51      case left {
52        Nil ⇒ left := new Node(value)
53        Node ⇒ left.add(value)
54      }
55    } else if value > key {
56      case right {
57        Nil ⇒ right := new Node(value)
58        Node ⇒ right.add(value)
59      }
60    } else { // value == key
61      skip
62    }
63
64  ⟨v:integer⟩
65  [Node⟨l,k,u⟩ ⇝ ⟨l,k1,u⟩ where k1:integer{l ≤ k1 ≤ u}]
66  deleteChild(value: Integer⟨v⟩) =
67    if value < key {
68      case left {
69        Nil ⇒ skip
70        Node ⇒
71          if left.isLeaf(value) {
72            left := new Nil()
73          } else {
74            left.deleteChild(value)
75          }
76      }
77    } else if value > key {
78      case right {
79        Nil ⇒ skip
80        Node ⇒
81          if right.isLeaf(value) {
82            right := new Nil()
83          } else {
84            right.deleteChild(value)
85          }
86      }
87    } else { // value == key
88      var newKey := value;
89      case right {
90        Nil ⇒
91          case left {
92            Nil ⇒ skip
93            Node ⇒ newKey := left.getMaxKey()
94          }
95        Node ⇒ newKey := right.getMinKey()
96      };
97      deleteChild(newKey);
98      key := newKey
99    }
```

```
100
101   ⟨v:integer⟩
102   contains(value: Integer⟨v⟩): Boolean⟨b⟩ where b:boolean =
103     if value < key {
104       case left {
105         Nil ⇒ false
106         Node ⇒ left.contains(value)
107       }
108     } else if value > key {
109       case right {
110         Nil ⇒ false
111         Node ⇒ right.contains(value)
112       }
113     } else { // value == key
114         true
115     }
116
117   ⟨v:integer⟩
118   isLeaf(value: Integer⟨v⟩): Boolean⟨b⟩ where b:boolean =
119     if value == key {
120       case left {
121         Nil ⇒
122           case right {
123             Nil ⇒ true
124             Node ⇒ false
125           }
126         Node ⇒ false
127       }
128     } else {
129       false
130     }
131
132   getMinKey(): Integer⟨l1⟩ where l1:integer{l ≤ l1 ≤ k} =
133     case left {
134       Nil  ⇒ key
135       Node ⇒ left.getMinKey()
136     }
137
138   getMaxKey(): Integer⟨u1⟩ where u1:integer{k ≤ u1 ≤ u} =
139     case right {
140       Nil ⇒ key
141       Node ⇒ right.getMaxKey()
142     }
143 }
```

Chapter 3

The DOL Language

In this chapter, we formalise the core language, a desugared version of the language used in the previous chapter, that comprises all the properties informally described in the examples. We build on the core sequential language from Gay et al. [2015] with session types removed, which offers a versatile basis for class-based object-oriented languages with mutable state, providing some of the techniques that allow us to simplify proofs while keeping them manageable. The proofs are provided in the subsequent chapter.

We adapt and extend Gay et al. [2015]'s language in three ways. (1) We replace session types with dependent types and study the consequences of this idea. (2) We incorporate inheritance and nominal subtyping, a feature absent from the base language. (3) We combine linear and unrestricted objects in the formalisation, building a less restrictive type system than the original one.

While many theoretical works exist in the domain of dependent types, no formalism that we know of has successfully combined dependent types and objects in a class-based language with mutable state.

Chapter Outline. This chapter is divided into the following sections:

- Section 3.1 defines the formal syntax of DOL.
- Section 3.2 presents the static type system, which is described in several stages: typing index refinements, kinding, subtyping, typing terms and typing programs.
- Section 3.3 defines an operational semantics on states.

$$
\begin{array}{rcll}
P & ::= & \bar{L} & \text{(programs)} \\
L & ::= & \mathsf{class}\ C : \Delta\ \mathsf{extends}\ T\{\bar{l} : \bar{T}\}\ \mathsf{is}\ \{\bar{M}\} & \text{(classes)} \\
M & ::= & m(x) = t & \text{(methods)} \\
T & ::= & C\bar{i} \mid \Pi a : I.T \mid \Sigma a : I.T \mid T + T & \text{(types)} \\
 & & \mid T \times T \mid T \rightsquigarrow T \mid T \to T & \\
t & ::= & x \mid f \mid \mathsf{new}\ C() \mid f \coloneqq t \mid t; t \mid m(t) & \text{(terms)} \\
 & & \mid f.m(t) \mid \mathsf{case}\ f\ \mathsf{of}\ (C_k \Rightarrow t_k)_{k \in 1,2} & \\
 & & \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t \mid \mathsf{while}\ t\ \mathsf{do}\ t & \\
\Delta & ::= & \epsilon \mid \Delta, a : I & \text{(index contexts)} \\
I & ::= & \mathsf{integer} \mid \mathsf{boolean} \mid \{a \colon I \mid p\} & \text{(index types)} \\
i & ::= & a \mid n \mid i \oplus i \mid p & \text{(index terms)} \\
p & ::= & \mathsf{false} \mid \mathsf{true} \mid \neg p \mid i \circledcirc i \mid p \mathbin{\text{⍟}} p & \text{(propositions)} \\
\oplus & ::= & + \mid - & \text{(arithmetic operators)} \\
\circledcirc & ::= & < \mid \leq \mid \doteq \mid \geq \mid > & \text{(relational operators)} \\
\text{⍟} & ::= & \wedge \mid \vee & \text{(logical operators)}
\end{array}
$$

Figure 3.1: Top-level syntax

3.1 Syntax

The formal language omits some features of the practical syntax used in the examples, even though our prototype includes them. We summarize the main differences, some of which have already been discussed in Chapter 2.

- Primitive values as used in the examples are translated into object references, which are the only values in our language, and all computations are performed by calling methods. This lightens the type system and the proofs without affecting expressivity, even if the resulting formal language reveals a somewhat cumbersome style of programming.

- All methods have exactly one parameter. A method written $m() = t$ abbreviates $m(\mathsf{top}) = t$ where top of type Top is used as a dummy parameter. The special init constructor method, which initialises all fields to new objects, is assumed to always have a parameter of type Top, even if we do not include it in the syntax. Defining methods that take an arbitrary number of parameters does not introduce any major technical challenge.

- Local variables are omitted, since they can be simulated by introducing extra pa-

rameters or fields.

We define the top-level syntax in Figure 3.1. Identifiers are drawn from the following disjoint countable sets: that of class names (denoted by B, C, D), that of fields (denoted by f, g), that of methods (denoted by m), that of object variables (denoted by x, y, o), and that of index variables (denoted by a, b). Labels l identify class members, that can either be fields or methods. The metavariables T, U, V, W range over object types; I, J range over index types; and i, j range over index terms.

Notation 3.1 (Sequences)**.** *The overbar notation, written $\bar{A}$, denotes a possibly empty sequence of A items (e.g. $A_1, \ldots, A_n$). The symbol ϵ represents the empty sequence. We sometimes use a subscript k to indicate a position in the sequence. Moreover, if the same subscript is used in different sequences, then these sequences are assumed to be of the same length.*

Programs P consist of collections of class declarations L. A class family, written $\mathsf{class}\ C : \Delta\ \mathsf{extends}\ T\{\bar{l} : \bar{T}\}\ \mathsf{is}\ \{\bar{M}\}$, associates a class named C to an index context Δ, a supertype T, a sequence of member declarations $\bar{l} : \bar{T}$ (field and method signatures), and a sequence of method implementations $\bar{M}$. An index context maps index variables to index types, fixing the class family arity, with each entry having the form $a : I$. A concrete or proper type, written $C\bar{i}$, is obtained by instantiating a class family with indices in application position. Index variables in Δ can be used to constrain types inside the class, including that of the explicit superclass T, where T is of the form $D\bar{i}$. (As we will see later, this restriction is enforced by the typing rules.) Finally, a method is implemented separately from its signature as $m(x) = t$, where t is the method body and x its single parameter.

The predefined Top denotes a concrete class (having an empty Δ). We assume its declaration to be

$$\mathsf{class\ Top} : \{\}\{\}$$

It does not declare neither a supertype nor members (fields or methods). We also assume the two predefined class families Integer and Boolean declared as

$$\mathsf{class\ Integer} : (\mathsf{i} : \mathsf{integer})\ \mathsf{extends\ Top}\{\mathsf{plus} : \ldots\}\ \mathsf{is}\ \{\ldots\}$$
$$\mathsf{class\ Boolean} : (\mathsf{b} : \mathsf{boolean})\ \mathsf{extends\ Top}\{\mathsf{and} : \ldots\}\ \mathsf{is}\ \{\ldots\}$$

which do not contain fields but provide several useful type invariant methods for arithmetic and logic operations as described in the examples (Section 2.2.2). None of these classes define a constructor, which means that they cannot be instantiated by new.

Definition 3.2 (Free Index Variables)**.** *The set of index variables appearing free in an item A, written FV(A), is inductively defined on the structure of A, which can take the form of a type T, an index type I, an index term i and a proposition p. For example, FV($Ci_1 \ldots Ci_n$) $\triangleq$ FV(i_1) $\cup \ldots \cup$ FV(i_n), while FV($\Pi a : I.T$) $\triangleq$ FV(I) $\setminus \{a\} \cup$ FV(T), and similarly FV($\{a\colon I \mid p\}$) $\triangleq$ FV(I) $\setminus \{a\} \cup$ FV(p).*

3.1.1 Types

Types T either classify objects or build method signatures. They can be of the following seven forms:

- A type $C\bar{i}$ extends with indices the Java notion of class types.
- A universal dependent type constructor, written $\Pi a : I.T$, where a may occur free in T, is a type that maps elements of the index type I to elements in the type T. It is used to build method signatures.
- An existential type constructor, written $\Sigma a : I.T$, where a may occur free in T, also maps elements of the index type I to elements in the type T, with the index variable a representing some unknown value in T. It is used to represent undetermined properties of a concrete object type.
- A union type $T + T$ classifies the set of objects belonging either to the left or the right type. It is used to define a supertype grouping independently developed classes.
- A product type, written $T \times T$, is used in method signatures, with the first type classifying the current object this, implicitly passed to the method, and the second one classifying the only explicit parameter.
- A parameter type of the form $T \rightsquigarrow T$ relates the two components that classify the current object this, the input type and a possibly different output type.
- A method type, written $T \to T$, maps the type of the parameters to a return type.

3.1.2 Terms

Terms t are fairly standard, except for some restricted forms that allow the type system to record more precisely how the types of objects vary. The variable x denotes a parameter. There is no qualified $x.f$. Instead, field access, written f, is only defined for a shared field (a restriction enforced by the typing rules), or in combination with assignment, method

calls and case constructs. This is part of the linear control of objects. All fields are private in the sense that every f always refers to a field of the current object (cf. Gay et al. [2015]). Object creation (new $C()$) does not take any parameters. Assignment ($f \coloneqq t$) is defined in terms of a non-standard swap operation in the style of Gay et al. [2015]. The operation assigns the value of t to the field f and returns the old value of f as its result. This prevents aliasing linear fields in terms such as $f_2 \coloneqq (f_1 \coloneqq t)$. Here, instead of evaluating to the value computed by t, that becomes the new content of f_1, the innermost assignment evaluates to the former content of f_1 which is put into f_2. The sequential term composition $(t; t)$ is standard. Method call is available both on the current object itself (a *self call*), written $m(t)$, and on a field of the current object this, written $f.m(t)$, but not on a parameter or an arbitrary term for that matter. This is because calling a method may change the type of the object on which the method is called. Note that the type system only records changes on the type of the current object this, the only one that can access its fields. To simplify, the case construct may only depend on a field, taking the form of case f of $(C_k \Rightarrow t_k)_{k\in 1,2}$ where f plays the role of the binding occurrence in the branches. Conditionals and while loops are standard.

3.1.3 Index Refinements

Index types I comprise the integer and boolean types, as well as the subset type of the form $\{a\colon I \mid p\}$. Index terms i include some of the possible index constructs, namely variables, integer literals, arithmetic operations, and also propositions, since a term of the boolean index type can instantiate a class family (for example, in method signatures of the predefined Boolean class). Propositions p take the form of the truth values, the negation and linear inequalities. We omit functions max and min from the examples as they do not introduce any additional technical challenge.

3.1.4 Additional Syntax Not Available to Programmers

Figure 3.2 defines syntactic extensions required for the formal system only. The internal type $C[F]$ (cf. [Gay et al., 2015]) is an alternative form of an object type that contains the class name C and a record field typing F that provides types for all the fields of C, including the inherited ones. For example, $C[\{f_1 : T_1, f_2 : T_2\}]$ is the internal type of an object of C having two fields of types T_1 and T_2, which may be defined either in C or in any of its superclasses. The internal type, used to classify the current object (this) in the context, cannot be the type of an arbitrary term, which never evaluates to this (as enforced by the typing rules). Instead, the purpose of the internal type is to allow the

$$
\begin{array}{rcll}
T & ::= & \ldots \mid C[F] & \text{(types)} \\
F & ::= & \{\bar{f} : \bar{T}\} & \text{(field types)} \\
r & ::= & o \mid r.f & \text{(paths)} \\
t & ::= & \ldots \mid \mathsf{return}\ t & \text{(terms)} \\
\theta & ::= & \epsilon \mid \theta, i/a & \text{(index substitutions)} \\
\Delta & ::= & \ldots \mid \Delta, p & \text{(index contexts)} \\
\Gamma & ::= & \epsilon \mid \Gamma, x : T & \text{(object contexts)} \\
K & ::= & \star \mid \Pi a : I.K & \text{(kinds)} \\
h & ::= & \epsilon \mid h, o = R & \text{(heaps)} \\
R & ::= & C\{\bar{f} = \bar{o}\} & \text{(object records)} \\
S & ::= & (h * r, t) & \text{(states)} \\
\mathcal{E} & ::= & [\_] \mid f := \mathcal{E} \mid \mathcal{E}; t \mid m(\mathcal{E}) \mid f.m(\mathcal{E}) & \text{(evaluation contexts)} \\
 & & \mid \mathsf{return}\ \mathcal{E} \mid \mathsf{if}\ \mathcal{E}\ \mathsf{then}\ t\ \mathsf{else}\ t \mid \mathsf{while}\ \mathcal{E}\ \mathsf{do}\ t &
\end{array}
$$

Figure 3.2: Extended syntax, used only in the type system and operational semantics

current object (this) to access its own fields, for typechecking assignment, method calls and case constructs, operations that may change its type through the field typing. Note that a type of the form $C\bar{i}$, if used for the current object instead of a type $C[F]$, might somehow be enough to track fields whose types depend on indices introduced in the class header and the methods' Π, yet it would not allow us to track fields whose types depend on existential variables bound to Σ whose scope is local to the type.

Terms evaluate to object references o, the only values in our language. To simplify, we do not define a separate syntactic category for object references. Instead, object references o are a subset of the variable names. Paths r in the style of Gay et al. [2015] represent locations in the heap, formed by the top-level object followed by a sequence of an arbitrary number of fields. For example, if r indicates the path of the currently active object, when a method call on a field f relative to r is entered, $r.f$ becomes the path that indicates the new current object that becomes active. The return term represents an ongoing method call during which the path changes as described above (Gay et al. [2015]). Paths and the return term are constructs belonging to the operational semantics.

Substitutions θ map index variables to index terms. Index contexts Δ are extended to accept propositions p. Object contexts Γ map object variables to types.

Types are classified into kinds K, much the same way as terms are classified into types. Kind $\star$ characterizes proper types, while kind $\Pi a : I.K$ classifies families of classes, i.e. types that have to be applied to index terms to form proper types. As we will see in the typing rules, we only ever need to check for proper types (with kind $\star$). In

fact, the only way to construct a type of a kind other than $\star$ is by declaring an indexed class. So, in the bank account example, the class family Account has kind $\Pi \mathsf{b} : \mathsf{natural}.\star$, and an instantiation, say Account 0, of kind $\star$, denotes the proper type of an object reference.

We sometimes write $\Pi\Delta.T$ as an abbreviation for $\Pi a_1 : I_1 \ldots \Pi a_n : I_n.T$, and similarly for a type Σ and a kind Π. When Δ is empty, we abbreviate $\Pi\Delta.T$ to T, $\Sigma\Delta.U$ to U and $\Pi\Delta.\star$ to $\star$.

Definition 3.3 (Domain)**.**

1. *The* domain *of an index context Δ, notation* $\mathsf{dom}(\Delta)$*, can be inductively defined as follows:*

$$\begin{array}{lcl} \mathsf{dom}(\epsilon) & = & \{\} \\ \mathsf{dom}(\Delta, a : I) & = & \mathsf{dom}(\Delta) \cup \{a\} \\ \mathsf{dom}(\Delta, p) & = & \mathsf{dom}(\Delta) \end{array}$$

2. *Let* $\theta = \{i_1/a_1, \ldots, i_n/a_n\}$*. Then* $\mathsf{dom}(\theta) = \{a_1, \ldots, a_n\}$*.*

3. *Let* $\Gamma = \{x_1 : T_1, \ldots, x_n : T_n\}$*. Then* $\mathsf{dom}(\Gamma) = \{x_1, \ldots, x_n\}$*.*

4. *Let* $h = \{o_1 = R_1, \ldots, o_n = R_n\}$*. Then* $\mathsf{dom}(h) = \{o_1, \ldots, o_n\}$*.*

A heap h is a mapping from object references o to object records R. We assume three special objects ($\mathsf{top} : \mathsf{Top}, \mathsf{false} : \mathsf{Boolean}\ \mathsf{false}, \mathsf{true} : \mathsf{Boolean}\ \mathsf{true}$) that are initially placed in the heap. The heap produced by the operation $h, (o = R)$ contains a new mapping from object reference o to record R. The operation of adding this binding to the heap h is only defined if $o \notin \mathsf{dom}(h)$. Note that the order in h is irrelevant. Records R are instances of classes, represented by $C\{\bar{f} = \bar{o}\}$, comprising the class of the object followed by a mutable record mapping field names to object references.

The operational semantics is defined as a reduction relation on states S of the form $(h * r, t)$, consisting of a heap h, a path r that represents the current object, and a term t. Evaluation contexts $\mathcal{E}$ are defined in the style of Wright and Felleisen [1994]. Intuitively, an evaluation context is a term with a hole $[\_]$ at the point where the next reduction step must take place in a call-by-value evaluation order; $\mathcal{E}[t]$ is the term obtained by replacing the hole in $\mathcal{E}$ by term t.

$\boxed{\Delta \vdash I}$ Under context Δ, index type I is well-formed

$$\frac{\vdash \Delta}{\Delta \vdash \mathsf{integer}}\ (\textsc{Wf-Integer}) \qquad \frac{\vdash \Delta}{\Delta \vdash \mathsf{boolean}}\ (\textsc{Wf-Boolean})$$

$$\frac{\Delta \vdash I \qquad \Delta, a : I \vdash p}{\Delta \vdash \{a\colon I \mid p\}}\ (\textsc{Wf-Subset})$$

$\boxed{\Delta \vdash p}$ Under context Δ, proposition p is well-formed

$$\frac{\vdash \Delta}{\Delta \vdash \mathsf{true}}\ (\textsc{Wf-True}) \qquad \frac{\vdash \Delta}{\Delta \vdash \mathsf{false}}\ (\textsc{Wf-False}) \qquad \frac{\Delta \vdash p}{\Delta \vdash \neg p}\ (\textsc{Wf-Neg})$$

$$\frac{\Delta \vdash i : \mathsf{integer} \qquad \Delta \vdash j : \mathsf{integer}}{\Delta \vdash i \circledcirc j}\ (\textsc{Wf-}\circledcirc) \qquad \frac{\Delta \vdash p_1 \qquad \Delta \vdash p_2}{\Delta \vdash p_1 \oslash p_2}\ (\textsc{Wf-}\oslash)$$

$\boxed{\vdash \Delta}$ Context Δ is well-formed

$$\frac{}{\vdash \epsilon}\ (\textsc{Wf-Empty}\Delta) \qquad \frac{\vdash \Delta \qquad \Delta \vdash p}{\vdash \Delta, p}\ (\textsc{Wf-Prop}\Delta)$$

$$\frac{\vdash \Delta \qquad a \notin \mathsf{dom}(\Delta) \qquad \Delta \vdash I}{\vdash \Delta, a : I}\ (\textsc{Wf-Var}\Delta)$$

Figure 3.3: Formation rules for index types, propositions and contexts

3.2 Static Semantics

We typecheck our language with respect to one index context Δ, and one object context Γ. The ordering is important in the index context, because of (index) variable-to-type dependencies, and irrelevant in the object context. For example, an index context $(\Delta_1, a : I, \Delta_2)$ is said to be well-formed if $a \notin \mathsf{dom}(\Delta_1) \cup \mathsf{dom}(\Delta_2)$ and $a \notin \mathrm{FV}(\Delta_1)$; an index context such as $(c : \{b\colon \mathsf{integer} \mid b \geq a\}, a : I)$ is ill-formed. These requirements are made explicit in the rules for the formation of index contexts, index types, and propositions that appear in subsequent subsections.

3.2.1 Index Typing

Our formulation of index refinements closely follows that of Xi [1998], Xi and Pfenning [1998, 1999], requiring a way to somehow decide the semantically defined relation

$$\Delta \models p$$

which has the following reading: "proposition p follows from the assumptions Δ". The relation is decidable provided index terms are drawn from some decidable theory (e.g. that of linear inequalities). Then, all proof obligations generated during typechecking fall within a decidable theory, and so typechecking can be proved to be decidable. In the

$\boxed{\Delta \vdash I <: J}$ Under context Δ, index type I is a subtype of J

$$\frac{\vdash \Delta}{\Delta \vdash \mathsf{integer} <: \mathsf{integer}}\ (\text{S-INTEGER}) \qquad \frac{\vdash \Delta}{\Delta \vdash \mathsf{boolean} <: \mathsf{boolean}}\ (\text{S-BOOLEAN})$$

$$\frac{\Delta \vdash I <: J \qquad \Delta \vdash \{a\colon I \mid p\}}{\Delta \vdash \{a\colon I \mid p\} <: J}\ (\text{S-SUBSETL})$$

$$\frac{\Delta \vdash I <: J \qquad \Delta, a : I \models p}{\Delta \vdash I <: \{a\colon J \mid p\}}\ (\text{S-SUBSETR})$$

Figure 3.4: Index subtyping rules

$\boxed{\Delta \vdash i : I}$ Under context Δ, index term i has type I

$$\frac{\vdash \Delta \qquad a : I \in \Delta}{\Delta \vdash a : I}\ (\text{I-VAR}) \qquad \frac{\vdash \Delta}{\Delta \vdash n : \mathsf{integer}}\ (\text{I-INTEGER})$$

$$\frac{\vdash \Delta}{\Delta \vdash \mathsf{false} : \mathsf{boolean}}\ (\text{I-TRUE}) \qquad \frac{\vdash \Delta}{\Delta \vdash \mathsf{true} : \mathsf{boolean}}\ (\text{I-FALSE})$$

$$\frac{\Delta \vdash i : \mathsf{integer} \qquad \Delta \vdash j : \mathsf{integer}}{\Delta \vdash i \oplus j : \mathsf{integer}}\ (\text{I-}\oplus)$$

$$\frac{\Delta \vdash i : I \qquad \Delta, a : I \vdash p \qquad \Delta \models p[i/a]}{\Delta \vdash i : \{a\colon I \mid p\}}\ (\text{I-INTRO})$$

$$\frac{\Delta \vdash i : J \qquad \Delta \vdash J <: I}{\Delta \vdash i : I}\ (\text{I-SUB})$$

Figure 3.5: Typing rules for index terms

case of an extended index language that would include multiplication or quantifiers, the relation might become undecidable. In the implementation of DOL, the proof obligations are passed to an auxiliary theorem prover, which is a powerful technique used also in systems such as Stardust [Dunfield, 2007] and Liquid Types [Rondon et al., 2008].

Figure 3.3 uses three typing judgements to check the formation of index types, propositions and contexts:

$$\Delta \vdash I \quad \text{and} \quad \Delta \vdash p \quad \text{and} \quad \vdash \Delta$$

Rule WF-SUBSET checks if the base type of the index variable a is well-formed and if the proposition is well-formed in the augmented (well-formed) context of index variables. The remaining rules are straightforward.

Figure 3.4 introduces subtyping on index types defined by the judgement

$$\Delta \vdash I <: J$$

which tell us that index type I is a subtype of index type J under the assumptions in context Δ. The rules for subset types draw inspiration from Gordon and Fournet [2010].

$\boxed{\Delta_1 \vdash \Delta_2 : \theta}$ Under context Δ_1, context Δ_2 derives substitution θ

$$\frac{\vdash \Delta}{\Delta \vdash \epsilon : \epsilon}\ (\theta\text{-EMPTY}) \qquad \frac{\Delta_1 \vdash \Delta_2 : \theta \qquad \Delta_1 \vdash i : I \qquad a \notin \mathsf{dom}(\Delta_1 \cup \theta)}{\Delta_1 \vdash (\Delta_2, a : I) : \theta, i/a}\ (\theta\text{-VAR})$$

$$\frac{\Delta_1 \vdash \Delta_2 : \theta \qquad \Delta_1 \models p[\theta]}{\Delta_1 \vdash (\Delta_2, p) : \theta}\ (\theta\text{-PROP})$$

Figure 3.6: Typing rules for substitution formation

Rule S-SUBSETL states that an index subset type is more specific than its base index type. On the other hand, rule S-SUBSETR states that an index subset type may form a supertype provided propostion p holds under the assumptions of the augmented index context.

Index typing defined by the judgement

$$\Delta \vdash i : I$$

checks that i has type I under the assumptions in Δ. The rules in Figure 3.5 closely follow Xi [1998]. The most important ones are rule I-VAR that ensures that an index variable is declared in the index context, and rule I-INTRO that introduces a subset type only when its proposition p logically follows from the assumptions in the index context. Since the goal of this thesis is the study of a language of mutable objects in the presence of dependent types, rather than a language of index refinements, we refer the interested reader to related work for a more detailed description of index typing.

3.2.2 Index Substitution

The binding occurrences of index variables appear in subset types, and also in types and kinds in the object language. We say that a occurs bound in p within $\{a\colon I \mid p\}$, in T within $\Pi a : I.T$ and $\Sigma a : I.T$, and in K within $\Pi a : I.K$.

To simplify the proofs, in particular to avoid having to rename bound variables in substitution, we follow Barendregt [1984]'s variable convention whereby the names of bound variables must all be distinct from each other and from any other variables occurring free in terms and types. Note that Barendregt's variable convention is employed both to index variables a and object variables x in order to obtain simple proofs.

We denote by $i_1[i_2/a]$ the capture-avoiding substitution of i_2 for the free occurrences of a in i_1. Index substitutions are defined inductively on the structure of index terms. For example, $(i_1 + i_2)[i_3/a]$ is defined as $i_1[i_3/a] + i_2[i_3/a]$. A single index substitution is extended pointwise to multiple index substitution θ, which maps index variables to index terms, by defining $i\epsilon \triangleq i$ and $i_1([i_2/a], \theta) \triangleq (i_1[i_2/a])[\theta]$.

$\boxed{\Delta \vdash K}$ Under context Δ, kind K is well-formed

$$\frac{\vdash \Delta}{\Delta \vdash \star}\ (\text{WF-}\star) \qquad \frac{\Delta, a : I \vdash K}{\Delta \vdash \Pi a : I.K}\ (\text{WF-}\Pi)$$

$\boxed{\Delta \vdash T : K}$ Under context Δ, type T has kind K

$$\frac{\mathsf{class}\ C : (\bar{a} : \bar{I})\ \mathsf{extends}\ \_\{\_\}\ \mathsf{is}\ \{\_\} \qquad \Delta \vdash \Pi \bar{a} : \bar{I}.\star}{\Delta \vdash C : \Pi \bar{a} : \bar{I}.\star}\ (\text{K-CLASS})$$

$$\frac{\Delta \vdash \star}{\Delta \vdash \mathsf{Top} : \star}\ (\text{K-TOP}) \qquad \frac{\Delta \vdash C\bar{i} : \Pi a : I.K \qquad \Delta \vdash i : I}{\Delta \vdash C\bar{i}i : K[i/a]}\ (\text{K-APP})$$

$$\frac{\Delta, a : I \vdash T : \star}{\Delta \vdash \Pi a : I.T : \star}\ (\text{K-}\Pi) \quad \frac{\Delta, a : I \vdash T : \star}{\Delta \vdash \Sigma a : I.T : \star}\ (\text{K-}\Sigma) \quad \frac{\Delta \vdash T : \star \qquad \Delta \vdash U : \star}{\Delta \vdash T + U : \star}\ (\text{K-}+)$$

$$\frac{\Delta \vdash T : \star \qquad \Delta \vdash U : \star}{\Delta \vdash T \times U : \star}\ (\text{K-}\times) \qquad \frac{\Delta \vdash C : K \qquad \Delta \vdash \bar{T} : \star}{\Delta \vdash C[\{\bar{f} : \bar{T}\}] : \star}\ (\text{K-RECORD})$$

Figure 3.7: Kind and type formation rules

The judgement for deriving a substitution θ from an index context Δ_2 has the form

$$\Delta_1 \vdash \Delta_2 : \theta$$

where, under the assumptions in context Δ_1, we think of Δ_2 as the input and θ as the output. The rules are defined in Figure 3.6 – here we are only interested in substitutions of index terms for index variables that satisfy the assumptions of the constraint domain. The rules require that Δ_2 and θ have the same arity and that each substituent is well-formed in the context. Specifically, rule θ-VAR ensures that for each substitution i/a, there is an entry $a : I$ such that $\Delta_1 \vdash i : I$. Rule θ-PROP checks that the substitution applied to a proposition p is satisfiable.

As for index terms, application of a capture-avoiding substitution θ to a type T, denoted by $T[\theta]$, is standard, defined inductively on the structure of T.

3.2.3 Kinding

Types may be syntactically valid, yet defining the wrong arity or containing ill-typed (index) terms. The rules for kind and type formation are defined in Figure 3.7. Kind formation uses a judgement

$$\Delta \vdash K$$

with rule WF-$\star$ checking Δ formation. Rule WF-Π checks the kind of class families, mapping an index term of type I to a type of kind K. The second judgement

$$\Delta \vdash T : K$$

verifies that T has kind K under the assumptions in Δ. We do not define rules for types $\rightsquigarrow$ and $\rightarrow$, which are auxiliary forms to build method signatures. (As we will see, the rules for program formation in Figure 3.13 control how each type component is checked.)

Rule K-CLASS assigns a class name C the kind given to it by its class declaration. By definition of rule K-TOP, the native type has a proper kind $\star$. An instance of a family of classes can be instantiated in rule K-APP provided its indices have the expected index type. Rules K-Π and K-Σ add bindings to the index context and check that, under the added assumptions, the type T is well-formed. The rules K-$+$, K-$\times$ and K-RECORD simply check each type component.

Example. The system uses the kinding rules to check the supertype declared by a class. Consider the PlusAccount class from the examples. Assume a context

$$\Delta \triangleq \mathsf{s} : \mathsf{natural}, \mathsf{c} : \mathsf{natural}, \mathsf{b} : \mathsf{natural}$$

where natural abbreviates an index subset type of the form $\{\mathsf{a} : \mathsf{integer} \mid \mathsf{a} \geq 0\}$. Also, suppose that $\vdash \Delta$. We omit these derivations in the example below. We can show that the type Account b is well-kinded as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\vdash \Delta} \quad \mathsf{a} \notin \mathsf{dom}(\Delta) \quad \dfrac{\vdots}{\Delta \vdash \mathsf{natural}}\,(\text{WF-SUBSET})}{\vdash \Delta, \mathsf{a} : \mathsf{natural}}\,(\text{WF-VAR}\Delta)}{\Delta, \mathsf{a} : \mathsf{natural} \vdash \star}\,(\text{WF-}\star)}{\Delta \vdash \Pi \mathsf{a} : \mathsf{natural}.\star}\,(\text{WF-}\Pi) \qquad \ldots}{\Delta \vdash \mathsf{Account} : \Pi \mathsf{a} : \mathsf{natural}.\star}\,(\text{K-CLASS}) \qquad \text{I-VAR}}{\Delta \vdash \mathsf{Account\ b} : \star}\,(\text{K-APP})$$

3.2.4 Subtyping

Term typing and method overriding rely on the subtyping relation defined as the reflexive and transitive closure of the inheritance relation as in Java, guided by the "safe substitutability principle" [Liskov and Wing, 1994]. The judgement

$$\Delta \vdash T <: U$$

asserts that T is a subtype of U under the assumptions in context Δ. As we will see later, the typing relation includes an explicit rule for subsumption T-SUB, which may

$\boxed{\Delta \vdash T <: U}$ Under context Δ, type T is a subtype of U

$$\frac{\mathsf{class}\ C : (\bar{a} : \bar{I})\ \mathsf{extends}\ T\{\_\}\ \mathsf{is}\ \{\_\} \quad \Delta \vdash \bar{i} : \bar{I} \quad \Delta \vdash T[\bar{i}/\bar{a}] : \star}{\Delta \vdash C\bar{i} <: T[\bar{i}/\bar{a}]}\ (\text{S-SUPER})$$

$$\frac{\Delta \models \bar{i} \doteq \bar{j} \qquad \Delta \vdash C\bar{j} : \star}{\Delta \vdash C\bar{i} <: C\bar{j}}\ (\text{S-APP}) \qquad \frac{\Delta \vdash T[i/a] <: U \qquad \Delta \vdash i : I}{\Delta \vdash \Pi a : I.T <: U}\ (\text{S-}\Pi\text{L})$$

$$\frac{\Delta, a : I \vdash T <: U}{\Delta \vdash T <: \Pi a : I.U}\ (\text{S-}\Pi\text{R}) \qquad \frac{\Delta, a : I \vdash T <: U}{\Delta \vdash \Sigma a : I.T <: U}\ (\text{S-}\Sigma\text{L})$$

$$\frac{\Delta \vdash T <: U[i/a] \qquad \Delta \vdash i : I}{\Delta \vdash T <: \Sigma a : I.U}\ (\text{S-}\Sigma\text{R}) \qquad \frac{\Delta \vdash T_1 <: U \qquad \Delta \vdash T_2 <: U}{\Delta \vdash (T_1 + T_2) <: U}\ (\text{S-}+\text{L})$$

$$\frac{\Delta \vdash T <: U_k}{\Delta \vdash T <: (U_1 + U_2)}\ (\text{S-}+\text{R}_k) \qquad \frac{\Delta \vdash T_1 <: U_1 \quad \Delta \vdash T_2 <: U_2}{\Delta \vdash (T_1 \times T_2) <: (U_1 \times U_2)}\ (\text{S-}\times)$$

$$\frac{\Delta \vdash \bar{T} <: \bar{U}}{\Delta \vdash C[\{\bar{f} : \bar{T}\}] <: C[\{\bar{f} : \bar{U}\}]}\ (\text{S-RECORD})$$

$$\frac{\Delta \vdash T_1 <: T_2 \qquad \Delta \vdash T_2 <: T_3}{\Delta \vdash T_1 <: T_3}\ (\text{S-TRANS})$$

Figure 3.8: Subtyping rules

be invoked at any time within a typing derivation. For this reason, this is a declarative specification of the subtyping relation (for the the algorithmic system, see Chapter 5).

The rules for subtyping are defined in Figure 3.8. The work by Aspinall and Compagnoni [1996] was important to understand the subtyping relation in the presence of dependent types. The subtyping rules for Π and Σ types have been adapted from Dunfield and Pfenning [2003], Dunfield [2007].

All types in the top-level language are subject to subtyping, except for $\rightsquigarrow$ and $\rightarrow$, since these types cannot arise from terms, and are not used to check method overriding (cf. Figures 3.12 and 3.13). The internal field typing is also subject to subtyping in order to check compatibility between fields of the same class. This relation is always derived with respect to the internal type of the current object (this), the only one that has access to its own fields (as enforced by the typing rules).

Rule S-SUPER is completely standard for object-oriented languages, adjusted to dependent types. Because class Top does not declare a supertype, it follows that Top is a supertype of every other type. By rule S-APP, subtyping is reflexive on class types, extended pointwise to all possible applications of the class type that satisfy the $\models$ relation, and by rule S-TRANS, subtyping is transitive. This makes it a pre-order.

In rules S-ΠL and S-ΠR, the left rule instantiates the index variable a to i in the subtype, while the right rule relates two types T and U provided the variable a does not

appear free in T. The reasoning for rules S-ΣL and S-ΣR is similar, yet inverted. As mentioned, we follow Barendregt [1984]'s variable convention, i.e. we implicitly assume that the variable a in the extended context of both S-ΠR and S-ΣL is distinct from all the variables already in Δ.

The two rules S-+L and S-+R$_k$ together imply that a type $T + U$ is a least upper bound of T and U. Rule S-$\times$ expresses that the subtyping relation is a congruence. Rule S-RECORD checks compatibility between field typings.

3.2.5 Auxiliary Functions and Predicates

As in Featherweight Java (FJ) [Igarashi et al., 2001], our typing rules rely on a few auxiliary functions and predicates. These are defined in Figure 3.9 and described below. We write $m \notin \bar{l}$ and $m \notin \bar{M}$ to indicate that the method name m is not included respectively in the sequence of member names $\bar{l}$ and method definitions $\bar{M}$. We denote by $T[C\bar{i}/D]$ the substitution of $C\bar{i}$ for the free occurrences of D bound to a type $\rightsquigarrow$ in T by defining

$$
\begin{array}{rcl}
B\bar{j}[C\bar{i}/D] & = & \begin{cases} B\bar{j} & \text{if } B \neq D \\ C\overline{ij} & \text{otherwise} \end{cases} \\
(\Sigma a : I.T)[C\bar{i}/D] & = & \Sigma a : I.T[C\bar{i}/D] \\
(T + U)[C\bar{i}/D] & = & T[C\bar{i}/D] + U[C\bar{i}/D] \\
(T \rightsquigarrow U)[C\bar{i}/D] & = & T[C\bar{i}/D] \rightsquigarrow U[C\bar{i}/D] \\
(\Pi\Delta.T \times U \to W)[C\bar{i}/D] & = & \Pi\Delta.T[C\bar{i}/D] \times U \to W
\end{array}
$$

Definition 3.4 (Union of Field Types)**.** *We denote by $\sqcup$ the disjoint union of field types, i.e. the operation of $F_1 \sqcup F_2$ is defined by merging F_1 and F_2 if their domains are disjoint, being undefined otherwise.*

- Function $\mathsf{classof}(T)$ looks up the class of a type T of the form $C\bar{i}$ and $\Sigma a : I.U$, begin undefined for other forms.
- Function $\mathsf{fields}(T)$ expects T to be of the appropriate form – Top, $\Sigma a : I.U$ and $C\bar{i}$ –, and builds an internal type consisting of the class name and a record of the field names with their types declared by the superclasses of T and by T itself. Notice that a subclass may extend an instantiated superclass, which means that, because of substitutions, the types of fields in the subclass may not be identical to those in the superclass.
- Function $\mathsf{mtype}(m, C\bar{i})$ looks up the signature of a method m for a receiver type of the form $C\bar{i}$. The function uses two rules: MT-CLASS looks up the method

$\boxed{\mathsf{classof}(T) = C}$

$$\mathsf{classof}(C\bar{i}) = C \qquad \mathsf{classof}(\Sigma a : I.T) = \mathsf{classof}(T)$$

$\boxed{\mathsf{fields}(T) = U}$

$$\mathsf{fields}(\mathsf{Top}) = \mathsf{Top}[\{\}] \qquad \mathsf{fields}(\Sigma a : I.T) = \Sigma a : I.\mathsf{fields}(T)$$

$$\frac{\mathsf{class}\ C : (\bar{a} : \bar{I})\ \mathsf{extends}\ D\bar{j}\{\bar{f} : \bar{U}, \bar{m} : \_\}\ \mathsf{is}\ \{\_\} \qquad \mathsf{fields}(D\bar{j}[\bar{i}/\bar{a}]) = D[\{\bar{g} : \bar{V}\}]}{\mathsf{fields}(C\bar{i}) = C[\{\bar{g} : \bar{V}\} \sqcup \{\bar{f} : \bar{U}[\bar{i}/\bar{a}]\}]}$$

$\boxed{\mathsf{mtype}(m, C\bar{i}) = T}$

$$\frac{\mathsf{class}\ C : (\bar{a} : \_)\ \mathsf{extends}\ \_\{\ldots, m : U, \ldots\}\ \mathsf{is}\ \{\_\}}{\mathsf{mtype}(m, C\bar{i}) = U[\bar{i}/\bar{a}]}\ (\text{MT-CLASS})$$

$$\frac{\mathsf{class}\ C : (\bar{a} : \_)\ \mathsf{extends}\ D\bar{j}\{\bar{l} : \_\}\ \mathsf{is}\ \{\_\} \qquad m \notin \bar{l}}{\mathsf{mtype}(m, C\bar{i}\bar{i}') = \mathsf{mtype}(m, D\bar{j}[\bar{i}\bar{i}'/\bar{a}])[C\bar{i}/D]}\ (\text{MT-SUPER})$$

$\boxed{\mathsf{mbody}(m, C) = \lambda x.t}$

$$\frac{\mathsf{class}\ C : \_\ \mathsf{extends}\ \_\{\_\}\ \mathsf{is}\ \{\ldots, m(x) = t, \ldots\}}{\mathsf{mbody}(m, C) = \lambda x.t}\ (\text{MB-CLASS})$$

$$\frac{\mathsf{class}\ C : \_\ \mathsf{extends}\ D\bar{j}\{\_\}\ \mathsf{is}\ \{\bar{M}\} \qquad m \notin \bar{M}}{\mathsf{mbody}(m, C) = \mathsf{mbody}(m, D)}\ (\text{MB-SUPER})$$

$\boxed{\mathsf{q}(T)\ \text{where}\ \mathsf{q} ::= \mathsf{un} \mid \mathsf{lin}}$

$$\mathsf{un}(\mathsf{Top}) \qquad \frac{\text{not}\ \mathsf{un}(T)}{\mathsf{lin}(T)}$$

$$\frac{\mathsf{classof}(T) = C \qquad \mathsf{class}\ C : \_\ \mathsf{extends}\ \_\{\bar{f} : \_, \bar{m} : \Pi\_.(\bar{T} \rightsquigarrow \bar{T} \times \_ \to \_)\}\ \mathsf{is}\ \{\_\}}{\mathsf{un}(T)}$$

Figure 3.9: Auxiliary functions and predicates

$\boxed{\Delta \vdash \Gamma}$ Under context Δ, context Γ is well-formed

$$\frac{\vdash \Delta}{\Delta \vdash \epsilon}\ (\text{WF-EMPTY}\Gamma) \qquad \frac{\Delta \vdash \Gamma \qquad x \notin \mathsf{dom}(\Gamma) \qquad \Delta \vdash T : K}{\Delta \vdash \Gamma, x : T}\ (\text{WF-}\Gamma)$$

Figure 3.10: Formation rules for object contexts

$\boxed{\Delta_1; \Gamma \vdash r : T \dashv \Delta_2}$ Under initial contexts $\Delta_1; \Gamma$, path r has type T, with final context Δ_2

$$\frac{\Delta \vdash \Gamma}{\Delta; \Gamma, r : T \vdash r : T \dashv \Delta}\ (\text{T-REF}) \qquad \frac{\Delta; \Gamma \vdash r : C[F] \dashv \Delta}{\Delta; \Gamma \vdash r.f : F(f) \dashv \Delta}\ (\text{T-FIELD})$$

$$\frac{\Delta_1; \Gamma \vdash r : \Sigma a : I.T \dashv \Delta_2}{\Delta_1; \Gamma_1 \vdash r : T \dashv \Delta_2, a : I}\ (\text{T-UNPACK})$$

$$\frac{\Delta; \Gamma \vdash r : C[F] \dashv \Delta \qquad \Delta \vdash C[F] <: \mathsf{fields}(C\bar{i})}{\Delta; \Gamma \vdash r : C\bar{i} \dashv \Delta}\ (\text{T-HIDE})$$

Figure 3.11: Typing rules for paths

signature in class C, while MT-SUPER looks for it by ascending the inheritance hierarchy of classes. The above remark about substitutions in types is also valid for the result of $\mathsf{mtype}(m, C\bar{i})$.

- Function $\mathsf{mbody}(m, C)$, defined for a class name, is similar to $\mathsf{mtype}(m, C\bar{i})$ yet simpler: rules MB-CLASS and MB-SUPER look up the body of a method m in class C or in one of its superclasses, returning a pair, written $\lambda x.t$, composed of a parameter x and a term t. The function is used only in the operational semantics

- Predicate $\mathsf{q}(T)$, where q is either un or lin, may use the class obtained by function $\mathsf{classof}(T)$ to assign a qualifier to a type (rather than rely on user-defined annotations): a type is said to be unrestricted (un) if denotes an instance of a type invariant class, that is, a class whose methods do not change the state of the current object (the input and output types coincide); it is linear (lin) if its class defines at least one type varying method, which indicates that the state of the current object is modified.

3.2.6 Term Typing

For checking the formation of object contexts, we use a judgement of the form

$$\Delta \vdash \Gamma$$

The judgement for typing paths, defined in Figure 3.11, is of the form

$\boxed{\Delta_1;\Gamma_1 * r_1 \vdash t : T \dashv \Delta_2;\Gamma_2 * r_2}$ Under initial contexts $\Delta_1;\Gamma_1$ with path r_1, term t has type T, with final contexts $\Delta_2;\Gamma_2$ and path r_2

$$\frac{\Delta \vdash \Gamma \qquad \mathsf{un}(T)}{\Delta;\Gamma, x : T * r \vdash x : T \dashv \Delta;\Gamma, x : T * r} \text{ (T-UnVar)}$$

$$\frac{\Delta \vdash \Gamma \qquad \mathsf{lin}(T)}{\Delta;\Gamma, x : T * r \vdash x : T \dashv \Delta;\Gamma * r} \text{ (T-LinVar)}$$

$$\frac{\Delta;\Gamma \vdash r.f : T \dashv \Delta \qquad \mathsf{un}(T)}{\Delta;\Gamma * r \vdash f : T \dashv \Delta;\Gamma * r} \text{ (T-UnField)}$$

$$\frac{\Delta \vdash \Gamma}{\Delta;\Gamma * r \vdash \mathsf{new}\ C() : C.\mathsf{init} \dashv \Delta;\Gamma * r} \text{ (T-New)}$$

$$\frac{\begin{array}{c}\Delta_1;\Gamma_1 * r_1 \vdash t : T \dashv \Delta_2;\Gamma_2 * r_2 \\ \Delta_2;\Gamma_2 \vdash r_2 : C[F] \dashv \Delta_2 \qquad \Delta_2;\Gamma_2\{r_2.f \leftarrow\!\!\shortmid T\} \vdash r_2 : C\bar{i} \dashv \Delta_2\end{array}}{\Delta_1;\Gamma_1 * r_1 \vdash f \coloneqq t : F(f) \dashv \Delta_2;\Gamma_2\{r_2.f \leftarrow\!\!\shortmid T\} * r_2} \text{ (T-Assign)}$$

$$\frac{\begin{array}{c}\Delta_1;\Gamma_1 * r_1 \vdash t_1 : U \dashv \Delta_2;\Gamma_2 * r_2 \\ \Delta_2;\Gamma_2 * r_2 \vdash t_2 : T \dashv \Delta_3;\Gamma_3 * r_2 \qquad \mathsf{un}(U)\end{array}}{\Delta_1;\Gamma_1 * r \vdash t_1; t_2 : T \dashv \Delta_3;\Gamma_3 * r_2} \text{ (T-Seq)}$$

$$\frac{\begin{array}{c}\Delta_1;\Gamma_1 * r_1 \vdash t : U[\theta] \dashv \Delta_2;\Gamma_2 * r_2 \qquad \Delta_2;\Gamma_2 \vdash r_2 : C\bar{i} \dashv \Delta_2 \\ \mathsf{mtype}(m, C\bar{i}) = \Pi\Delta.(C\bar{i} \rightsquigarrow T \times U \to W) \qquad \Delta_2 \vdash \Delta : \theta \\ \Delta_2;\Gamma_2\{r_2 \leftarrow\!\!\shortmid T[\theta]\} \vdash r_2 : C\bar{j} \dashv \Delta_3\end{array}}{\Delta_1;\Gamma_1 * r_1 \vdash m(t) : W[\theta] \dashv \Delta_3;\Gamma_2\{r_2 \leftarrow\!\!\shortmid \mathsf{fields}(C\bar{j})\} * r_2} \text{ (T-SelfCall)}$$

$$\frac{\begin{array}{c}\Delta_1;\Gamma_1, r_1 : C[F] * r_1 \vdash t : U[\theta] \dashv \Delta_2;\Gamma_2 * r_2 \qquad \Delta_2;\Gamma_2 \vdash r_2.f : T_1 \dashv \Delta_3 \\ \mathsf{mtype}(m, T_1) = \Pi\Delta.(T_1 \rightsquigarrow T_2 \times U \to W) \qquad \Delta_3 \vdash \Delta : \theta \\ \Delta_3;\Gamma_2\{r_2.f \leftarrow\!\!\shortmid T_2[\theta]\} \vdash r_2 : C\bar{i} \dashv \Delta_3\end{array}}{\Delta_1;\Gamma_1, r_1 : C[F] * r_1 \vdash f.m(t) : W[\theta] \dashv \Delta_3;\Gamma_2\{r_2.f \leftarrow\!\!\shortmid T_2[\theta]\} * r_2} \text{ (T-Call)}$$

$$\frac{\begin{array}{c}\Delta_1;\Gamma_1 \vdash r.f : (U_1 + U_2) \dashv \Delta_2 \qquad \mathsf{classof}(U_k) = C_k \\ \Delta_2;\Gamma_1\{r.f \leftarrow\!\!\shortmid U_k\} * r \vdash t_k : T \dashv \Delta_3;\Gamma_2 * r \qquad C_1 \neq C_2\end{array}}{\Delta_1;\Gamma_1 * r \vdash \mathsf{case}\ f\ \mathsf{of}\ (C_k \Rightarrow t_k)_{k\in 1,2} : T \dashv \Delta_3;\Gamma_2 * r} \text{ (T-Case)}$$

$$\frac{\begin{array}{c}\Delta_1;\Gamma_1 * r_1 \vdash t : \mathsf{Boolean}\ p \dashv \Delta_2;\Gamma_2 * r_2 \\ \Delta_2, p;\Gamma_2 * r_2 \vdash t_1 : T \dashv \Delta_3;\Gamma_3 * r_2 \qquad \Delta_2, \neg p;\Gamma_2 * r_2 \vdash t_2 : T \dashv \Delta_3;\Gamma_3 * r_2\end{array}}{\Delta_1;\Gamma_1 * r_1 \vdash \mathsf{if}\ t\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 : T \dashv \Delta_3;\Gamma_3 * r_2} \text{ (T-If)}$$

$$\frac{\begin{array}{c}\Delta_1;\Gamma_1 * r_1 \vdash t_1 : \mathsf{Boolean}\ p \dashv \Delta_2;\Gamma_2 * r_2 \\ \Delta_2, p;\Gamma_2 * r_2 \vdash t_2 : \mathsf{Top} \dashv \Delta_2;\Gamma_2 * r_2\end{array}}{\Delta_1;\Gamma_1 * r_1 \vdash \mathsf{while}\ t_1\ \mathsf{do}\ t_2 : \mathsf{Top} \dashv \Delta_2, \neg p;\Gamma_2 * r_2} \text{ (T-While)}$$

$$\frac{\Delta_1;\Gamma_1 * r_1 \vdash t : U \dashv \Delta_2;\Gamma_2 * r_2 \qquad \Delta_2 \vdash U <: T}{\Delta_1;\Gamma_1 * r_1 \vdash t : T \dashv \Delta_2;\Gamma_2 * r_2} \text{ (T-Sub)}$$

Figure 3.12: Typing rules for terms in the top-level language

$$\Delta; \Gamma \vdash r : T \dashv \Delta$$

Rule T-REF obtains the type of the current object r without generating any new index type information, so the final index context in the conclusion is the same as the initial one. Rule T-FIELD returns the type of a field $r.f$. The existential elimination rule T-UNPACK unpacks an existential type by extending the final index context from its premise in the conclusion. Rule T-HIDE restores a type $C\bar{i}$, which is the type viewed from outside, from the internal type $C[F]$ of the current object r. Note that the internal type contains a field typing F that describes the state of the object from the perspective of the class itself, but not from the perspective of client code in other classes. In order to type a class, we need to consider that the two types, the internal and the one viewed from outside, are correct and consistent with respect to the subtyping relation $C[F] <: \mathsf{fields}(C\bar{i})$ under an index context Δ.

The two judgements above are used by the typing judgement for terms that take the form

$$\Delta_1; \Gamma_1 * r_1 \vdash t : T \dashv \Delta_2; \Gamma_2 * r_2$$

The meaning of this judgement is that the evaluation of term t may both extend the context Δ_1 (for example, with existential variables that arise from the types of fields, or with propositions) and change the types contained in Γ_1 (for example, by assigning values to objects, or by calling methods on them), giving rise to the final contexts $\Delta_2; \Gamma_2$. Linearity is yet another reason for a different final object context: if x is linear and is used in t, then x is consumed and does not appear in Γ_2. The judgement includes r_1 and r_2 in the style of Gay et al. [2015], which are paths needed for typing runtime terms and tracing objects in the heap. When typechecking a program, both r_1 and r_2 are always this, and are used exclusively to access the fields of the current class. Hence, the judgement for typing top-level terms will always have the form

$$\Delta_1; \Gamma_1, \mathsf{this} : C[F_1] * \mathsf{this} \vdash t : T \dashv \Delta_2; \Gamma_2, \mathsf{this} : C[F_2] * \mathsf{this}$$

where Δ_1 may be extended by Δ_2, and Γ_2 may differ from Γ_1 on the method parameter x – if Γ_1 is $x : U$ and U is linear, then Γ_2 must be ϵ since x has been consumed by t.

Before we describe the rules, we need some additional definitions.

Definition 3.5 (Member Access)**.** *We write $C.l_k$ to mean T_k for a class declaration of the form* $\mathsf{class}\ C : \Delta\ \mathsf{extends}\ T\{l_1 : T_1, \ldots, l_n : T_n\}\ \mathsf{is}\ \{\bar{M}\}$ *with* $1 \leq k \leq n$.

Definition 3.6 (Operations on Field Types and Object Contexts)**.**

- *If $F = \{f_1 : T_1, \ldots, f_n : T_n\}$, then $F(f_k) \triangleq T_k$, and we define $F\{f_j \hookleftarrow U\} \triangleq \{f_1 : T'_1, \ldots, f_n : T'_n\}$ where $T'_k = T_k$ and $T'_j = U$ for $k \neq j$ and $n \geq 1$ and $1 \leq k \leq n$ and $1 \leq j \leq n$.*

- $(\Gamma, x : T)\{x \leftarrow\!\!\!\rightarrow U\} \triangleq \Gamma, x : U$.
- $\Gamma\{r.f \leftarrow\!\!\!\rightarrow T\} \triangleq \Gamma\{r \leftarrow\!\!\!\rightarrow C[F\{f \leftarrow\!\!\!\rightarrow T\}]\}$ *if* $\Delta; \Gamma \vdash r : C[F] \dashv \Delta$ *for some* Δ.

The typing rules for the top-level terms (Figure 3.1) are given in Figure 3.12.

- Rules T-UNVAR and T-LINVAR are used to access a parameter. The former is the standard rule for reading a variable, while the latter implements destructive reads, taking x from the final context in the conclusion.
- Rule T-UNFIELD is used for field access, being defined for unrestricted types only (since the effect of reading f linear would remove it from the current object type).
- Rule T-NEW is the rule for object creation, giving the new object the type read from the init method signature. The term does not depend on any information in the object context, so the final context in the conclusion is the same as the initial one.
- Rule T-ASSIGN modifies a field of the current object, acting on its type $C[F]$. Unlike the rule for assignment in Java, when a field is changed, we need to check all the other fields in F to ensure that any dependencies are satisfied. We do this with judgement $\Delta_2; \Gamma_2\{r_2.f \leftarrow\!\!\!\rightarrow T\} \vdash r_2 : C\bar{i} \dashv \Delta_2$ derived by rule T-HIDE that recovers a top-level type $C\bar{i}$ using with the updated context as its initial context. Again, unlike the standard rule for assignment, our rule returns as its result the type of the old object contained in the field as part of the linear control of objects.
- Rule T-SEQ is the standard rule for the sequence operation, except that it checks the first subterm and considers its possible effects in the typing context that checks the second one.
- Rules T-SELFCALL and T-CALL are used for calling methods. Rule T-SELFCALL checks the type of the parameter as usual, but uses rule T-HIDE to obtain a top-level type for the current object r of the form $C\bar{i}$ that allows method m to be called (its signature yielding a substitution θ applied to the parameter and output types). The final object context is updated in the conclusion with a type obtained by $\mathsf{fields}(C\bar{j})$ for r, where $C\bar{j}$ is derived by possibly unpacking the receiver output type $T[\theta]$ in the premise $\Delta_2; \Gamma_2\{r_2 \leftarrow\!\!\!\rightarrow T[\theta]\} \vdash r_2 : C\bar{j} \dashv \Delta_3$. Rule T-CALL checks a method call on a field, combining the strategies used in rules T-SELFCALL and T-ASSIGN. As T-SELFCALL, it expects the receiver ($r_2.f$ in this case) to have a type that allows method m to be called (its signature yielding a substitution θ applied to the parameter and output types). As T-ASSIGN, it retrieves the top-level type in order to check that this mutation does not break dependencies elsewhere in F before updating the final object context in the conclusion.

- Rule T-CASE makes the case distinction on a field f with a union type. Each branch is then typed with a initial context where f is bound to either the left or the right type. The rule requires that two branches have the same type and final contexts, because f can only be bound to one type.

- Rule T-IF expects t in the condition to be of type Boolean p. Each branch is then typed with initial contexts asserting or negating the proposition p. As in T-CASE, both branches must return the same type and final contexts, since only one of the branches will be executed.

- Rule T-WHILE is analogous to T-IF yet simpler, with the negation of the loop invariant being added to the index context in the conclusion.

- Rule T-SUB is the usual subsumption rule, adjusted to our requirements.

Example. Rules T-SELFCALL and T-CALL use a substitution θ to replace index variables with index terms in types. To show how this works, we typecheck a method call from the examples in Chapter 2, namely

$$\epsilon; \Gamma_1 * \mathsf{this} \vdash \mathsf{acc.withdraw(v)} : \mathsf{Top} \dashv \epsilon; \Gamma_2 * \mathsf{this}$$

assuming

$$\Gamma_1 \triangleq \mathsf{v} : \mathsf{Integer}\ 70, \mathsf{this} : \mathsf{C}[\{\mathsf{acc} : \mathsf{A}\ 100\}]$$
$$\Gamma_2 \triangleq \mathsf{v} : \mathsf{Integer}\ 70, \mathsf{this} : \mathsf{C}[\{\mathsf{acc} : \mathsf{A}\ 30\}]$$

That is, variable acc, an instance of A (which for space reasons abbreviates the class name Account), is the only field of some class C. In the derivation of the method call above, we obtain the following premises:

1. $\epsilon; \Gamma_1 * \mathsf{this} \vdash \mathsf{v} : \mathsf{Integer}\ \mathsf{m}[70/\mathsf{m}] \dashv \epsilon; \Gamma_1 * \mathsf{this}$
2. $\epsilon; \Gamma_1 \vdash \mathsf{this.acc} : \mathsf{A}\ 100 \dashv \epsilon$
3. $\epsilon; \Gamma_1\{\mathsf{this.acc} \leftarrow\!\!\shortmid \mathsf{A}\ (100 - \mathsf{m})[70/\mathsf{m}]\} \vdash \mathsf{this} : \mathsf{C}\ 30 \dashv \epsilon$
4. $\mathsf{mtype}(\mathsf{withdraw}, \mathsf{A}\ 100) =$
   $\Pi \mathsf{m} : \{\mathsf{x}\colon \mathsf{integer} \mid \mathsf{x} \geq 0 \wedge \mathsf{x} \leq 100\}.(\mathsf{A}\ 100 \rightsquigarrow \mathsf{A}(100 - \mathsf{m}) \times \mathsf{Integer\ m} \rightarrow \mathsf{Top})$
5. $\epsilon \vdash \mathsf{m} : \{\mathsf{x}\colon \mathsf{integer} \mid \mathsf{x} \geq 0 \wedge \mathsf{x} \leq 100\} : (70/\mathsf{m})$

After applying the substitution, these premises become:

1. $\epsilon; \Gamma_1 * \mathsf{this} \vdash \mathsf{v} : \mathsf{Integer}\ 70 \dashv \epsilon; \Gamma_1 * \mathsf{this}$
2. $\epsilon; \Gamma_1 \vdash \mathsf{this.acc} : \mathsf{A}\ 100 \dashv \epsilon$
3. $\epsilon; \Gamma_1\{\mathsf{this.acc} \hookleftarrow \mathsf{A}\ 30\} \vdash \mathsf{this} : \mathsf{C}\ 30 \dashv \epsilon$
4. $\mathsf{mtype}(\mathsf{withdraw}, \mathsf{A}\ 100) =$
   $\Pi \mathsf{m} : \{\mathsf{x}\colon \mathsf{integer} \mid \mathsf{x} \geq 0 \wedge \mathsf{x} \leq 100\}.(\mathsf{A}\ 100 \rightsquigarrow \mathsf{A}\ (100 - \mathsf{m}) \times \mathsf{Integer}\ \mathsf{m} \rightarrow \mathsf{Top})$
5. $\epsilon \vdash \mathsf{m} : \{\mathsf{x}\colon \mathsf{integer} \mid \mathsf{x} \geq 0 \wedge \mathsf{x} \leq 100\} : (70/\mathsf{m})$

Premise (1) is derived using T-UNVAR, while premise (2) obtains the field type from rule T-FIELD. Premise (3) updates the field type in the context with the output type from mtype, and obtains from rule T-HIDE the corresponding top-level type of the current object this. Function mtype in (4) instantiates the method signature by performing a substitution $100/\mathsf{b}$ (omitted). Premise (5) is derived by rule θ-VAR that checks that the substitution is satisfiable.

3.2.7 Program Typing

A well-formed program relies on well-typed fields, methods and classes, which we formally define in Figure 3.13. The judgement of the form

$$\vdash_C M$$

states that a method in a class C is well-typed, while the judgement

$$\Delta \vdash_T l : U$$

checks that a member type is well-formed under a supertype T. The judgements for checking a class and a program take the following shape:

$$\vdash L \quad \text{and} \quad \vdash P$$

- Rule T-METHOD constructs the judgement for checking the body of a regular method by assuming two contexts: (1) an index context from the class declaration extended with the index context from the method signature, and (2) an object context containing the type of the parameter x from the method signature and the initial internal type of the current object this obtained by $\mathsf{fields}(T_1)$. Index variables that appear in the types from the signature must be bound by the extended context (this is ensured by T-CLASS, explained shortly). The rule uses the typing judgement for

$\boxed{\vdash_C M}$ Method M is well-formed in class C

$$\frac{\begin{array}{l}\mathsf{class}\ C : \Delta_1\ \mathsf{extends}\ \_\{\ldots, m : \Pi\Delta_2.(T_1 \rightsquigarrow T_2 \times U \to W), \ldots\}\ \mathsf{is}\ \{\_\} \\ \Delta_1, \Delta_2; x : U, \mathsf{this} : \mathsf{fields}(T_1) * \mathsf{this} \vdash t : W \dashv \Delta_3; \Gamma, \mathsf{this} : C[F] * \mathsf{this} \\ x : U \in \Gamma \Rightarrow \mathsf{un}(U) \qquad \Delta_3 \vdash C[F] <: \mathsf{fields}(T_2) \qquad m \neq \mathsf{init}\end{array}}{\vdash_C m(x) = t}\ \text{(T-METHOD)}$$

$$\frac{\mathsf{fields}(C.\mathsf{init}) = C[F] \quad \epsilon; \epsilon * r \vdash \mathsf{new}\ \bar{C}() : F(\bar{f}) \dashv \epsilon; \epsilon * r \quad \text{no cycles in } C}{\vdash_C \mathsf{init}() = \bar{f} := \mathsf{new}\ \bar{C}()}\ \text{(T-INIT)}$$

$\boxed{\Delta \vdash_T l : U}$ Member l has type U with supertype T

$$\frac{\Delta \vdash U : \star}{\Delta \vdash_T f : U}\ \text{(T-FTYPE)}$$

$$\frac{\begin{array}{c}\mathsf{mtype}(m, T)\ \text{undefined} \\ \Delta_1 \vdash \Pi\Delta_2.(C\bar{i} \times U \times W) : \star \quad \Delta_1, \Delta_2 \vdash C\bar{j} <: T_2\end{array}}{\Delta_1 \vdash_T m : \Pi\Delta_2.(C\bar{i} \rightsquigarrow T_2 \times U \to W)}\ \text{(T-MTYPE)}$$

$$\frac{\begin{array}{c}\mathsf{mtype}(m, T) = \Pi\Delta_2'.(T_1' \rightsquigarrow T_2' \times U' \to W') \\ \Delta_1 \vdash \Pi\Delta_2.(T_1 \times T_2 \times U' \times W) <: \Pi\Delta_2'.(T_1' \times T_2' \times U \times W')\end{array}}{\Delta_1 \vdash_T m : \Pi\Delta_2.(T_1 \rightsquigarrow T_2 \times U \to W)}\ \text{(T-OVERRIDE)}$$

$\boxed{\vdash L}$ Class declaration L is well-formed

$$\frac{\Delta \vdash T : \star \qquad \Delta \vdash_T \bar{l} : \bar{T} \qquad \vdash_C \bar{M}}{\vdash \mathsf{class}\ C : \Delta\ \mathsf{extends}\ T\{\bar{l} : \bar{T}\}\ \mathsf{is}\ \{\bar{M}\}}\ \text{(T-CLASS)}$$

$\boxed{\vdash P}$ Program P is well-formed

$$\frac{\vdash L_1 \quad \ldots \quad \vdash L_n}{\vdash L_1 \ldots L_n}\ \text{(T-PROGRAM)}$$

Figure 3.13: Typing rules for program formation

terms. On typing completion, the type of the parameter is checked for linearity and the final type of this is checked for consistency against the declared top-level output type using subtyping. The other rule, T-INIT, initialises all fields, including the inherited ones, under the condition that there are no cycles in the hierarchy induced by subclassing.

- Rule T-FTYPE states that a field must be "typed" by kind $\star$ of proper types – recall that only class families have a kind $\Pi\Delta.\star$ with Δ nonempty. When checking a method type, we must establish for both the input and output types of the current object this that the underlying class type is C and one of following holds: the method is altogether new (T-MTYPE), or the method type is a correct override of a

$\boxed{\Delta_1; \Gamma_1 * r_1 \vdash t : T \dashv \Delta_2; \Gamma_2 * r_2}$ Under initial contexts $\Delta_1; \Gamma_1$ with path r_1, term t has type T, with final contexts $\Delta_2; \Gamma_2$ and path r_2

$$\frac{\Delta_1; \Gamma_1 * r_1 \vdash t : T \dashv \Delta_2; \Gamma_2 * r_2.f \quad \Delta_2; \Gamma_2 \vdash r_2.f : C[F] \dashv \Delta_2 \quad \Delta_2 \vdash C[F] <: \mathsf{fields}(U)}{\Delta_1; \Gamma_1 * r_1 \vdash \mathsf{return}\ t : T \dashv \Delta_2; \Gamma_2\{r_2.f \hookleftarrow U\} * r_2} \text{ (T-RETURN)}$$

$\boxed{\Delta_1; \Gamma_1 * r \vdash \bar{t} : \bar{T} \dashv \Delta_2; \Gamma_2 * r}$ Under initial contexts $\Delta_1; \Gamma_1$ with path r, term sequence $\bar{t}$ has type $\bar{T}$, with final contexts $\Delta_2; \Gamma_2$ and the same path r

$$\frac{\Delta \vdash \Gamma}{\Delta; \Gamma * r \vdash \epsilon : \epsilon \dashv \Delta; \Gamma * r} \text{ (T-EMPTY)}$$

$$\frac{\Delta_1; \Gamma_1 * r \vdash t : T \dashv \Delta_2; \Gamma_2 * r \quad \Delta_2; \Gamma_2 * r \vdash \bar{t} : \bar{T} \dashv \Delta_3; \Gamma_3 * r}{\Delta_1; \Gamma_1 * r \vdash t\bar{t} : T\bar{T} \dashv \Delta_3; \Gamma_3 * r} \text{ (T-MULTI)}$$

Figure 3.14: Typing rules for runtime terms

superclass method (T-OVERRIDE). Method overriding requires the initial and final types of this in the current class to be subtypes of the corresponding types in the superclass, and the parameter and return types to be in a valid subtype relation, contravariant in the argument type and covariant in the return type.

- The two judgements above are used by rule T-CLASS that essentially makes the following three checks: that the direct supertype has the kind $\star$ of proper types, that member types are well-formed, and that each method body is well-typed.

- Finally, by rule T-PROGRAM a program is well formed if each class defined in it is well-typed.

3.2.8 Runtime Term Typing

To conclude the static semantics of DOL, we give in Figure 3.14 the typing rules for the return term and for sequenced terms. While the path r is always this when typing programs, it may vary when typing a term t as part of a runtime state, indicating the current object active in that state (cf. Gay et al. [2015]). As we have seen, any difference between r_1 and r_2 means that the runtime term contains a return, in which case r_1 and r_2 represent the object introduced by rule R-CALL and suppressed by rule R-RETURN. Note in particular that in the first premise of T-RETURN, we have $r_1 = r_2.f$ if the term t does not contain a return; otherwise, the paths are different.

Sequenced terms are a convenient technical device that we use in the proofs in the subsequent chapter. Rule T-MULTI is as expected: it checks the first term t and considers

$$\boxed{(h_1 * r, \mathsf{new}\ \bar{C}()) \longrightarrow (h_2 * r, \bar{o})} \quad \text{State } (h_1 * r, \mathsf{new}\ \bar{C}()) \text{ reduces to } (h_2 * r, \bar{o})$$

$$(h * r, \epsilon) \longrightarrow (h * r, \epsilon)\ \text{(R-EMPTY)}$$

$$\frac{(h_1 * r, \mathsf{new}\ C()) \longrightarrow (h_2 * r, o) \quad (h_2 * r, \mathsf{new}\ \bar{C}()) \longrightarrow (h_3 * r, \bar{o})}{(h_1 * r, (\mathsf{new}\ C() \mathsf{new}\ \bar{C}())) \longrightarrow (h_3 * r, o\bar{o})}\ \text{(R-MULTINEW)}$$

Figure 3.15: Reduction rules for sequenced object creation

the extensions in the index context and the possible effects in the object context that check the remaining terms $\bar{t}$.

3.3 Operational Semantics

Figure 3.15 gives the reduction rules for states of the form $(h * r, \mathsf{new}\ \bar{C}())$, consisting of a heap h, a path r and sequenced object creation $\mathsf{new}\ \bar{C}()$. These are used by the rules in Figure 3.16, which define an operational semantics on states S the form $(h * r, t)$ where the object path r is used to resolve field references appearing in the term t.

As usual, we denote by $t[o/x]$ the substitution of o for the free occurrences of x in t defined in the standard way. We also use the following definition:

Definition 3.7 (Operations on Heaps)**.** *Let h be the heap of the form $h = (h_0, o = R)$ where R is the object record $C\{f_1 = o_1, \ldots, f_n = o_n\}$. Then, $h(o) \triangleq R$ and $h(o).\mathsf{class} = C$, and, for all k such that $1 \leq k \leq n$,*

- $R.f_k \triangleq o_k$.
- $R\{f_j \leftarrow\!\!\shortmid\ o\} \triangleq C\{\bar{f} = \bar{o}'\}$ *where $o'_k = o_k$ and $o'_j = o$ for $k \neq j$ and $1 \leq j \leq n$.*
- $h\{o.f_k \leftarrow\!\!\shortmid\ o'\} \triangleq (h_0, o = R\{f_k \leftarrow\!\!\shortmid\ o'\})$.
- $h(r) \triangleq o_k$ *if $r = o.f_k$, and $h\{r.f \leftarrow\!\!\shortmid\ o'\} \triangleq h\{o_k.f \leftarrow\!\!\shortmid\ o'\}$.*

Rule R-NEW creates a fresh object and adds it to the heap, after having initialised all fields. For this, it relies on the reduction of states for sequenced object creation. Rule R-ASSIGN replaces the value of a field f of the current object located at r with a new reference, and returns the former object pointed by f. Rule R-SEQ reduces to the second part of the sequence of terms, discarding the first part only after it has become an object.

Rule R-SELFCALL is relative to a method call on the current object at r. The rule prepares the method body t with a substitution (the actual parameter for the formal one) before evaluating the term. R-CALL is the rule for a call on the object at f (relative to the

$\boxed{S_1 \longrightarrow S_2}$ State S_1 reduces to S_2

$$\frac{\mathsf{mbody}(\mathsf{init}, C) = \bar{f} := \mathsf{new}\ \bar{C}() \qquad (h_1 * r, \mathsf{new}\ \bar{C}()) \longrightarrow (h_2 * r, \bar{o}) \quad o \notin \mathsf{dom}(h_2)}{(h_1 * r, \mathsf{new}\ C()) \longrightarrow ((h_2, o = C\{\bar{f} = \bar{o}\} * r), o)}\ \text{(R-NEW)}$$

$$\frac{h(r).f = o_1}{(h * r, f := o_2) \longrightarrow (h\{r.f \leftarrow\!\!\shortmid\ o_2\} * r, o_1)}\ \text{(R-ASSIGN)}$$

$$(h * r, o; t) \longrightarrow (h * r, t)\ \text{(R-SEQ)}$$

$$\frac{h(r).\mathsf{class} = C \quad \mathsf{mbody}(m, C) = \lambda x.t}{(h * r, m(o)) \longrightarrow (h * r, t[o/x])}\ \text{(R-SELFCALL)}$$

$$\frac{h(r.f).\mathsf{class} = C \quad \mathsf{mbody}(m, C) = \lambda x.t}{(h * r, f.m(o)) \longrightarrow (h * r.f, \mathsf{return}\ t[o/x])}\ \text{(R-CALL)}$$

$$(h * r.f, \mathsf{return}\ o) \longrightarrow (h * r, o)\ \text{(R-RETURN)}$$

$$\frac{h(r.f).\mathsf{class} = C_k}{(h * r, \mathsf{case}\ f\ \mathsf{of}\ (C_k \Rightarrow t_k)_{k\in 1,2}) \longrightarrow (h * r, t_k)}\ \text{(R-CASE}_k\text{)}$$

$$(h * r, \mathsf{if}\ \mathsf{true}\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2) \longrightarrow (h * r, t_1)\ \text{(R-IFTRUE)}$$

$$(h * r, \mathsf{if}\ \mathsf{false}\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2) \longrightarrow (h * r, t_2)\ \text{(R-IFFALSE)}$$

$$\frac{t_2 = \mathsf{if}\ t\ \mathsf{then}\ (t_1; \mathsf{while}\ t\ \mathsf{do}\ t_1)\ \mathsf{else}\ \mathsf{top}}{(h * r, \mathsf{while}\ t\ \mathsf{do}\ t_1) \longrightarrow (h * r, t_2)}\ \text{(R-WHILE)}$$

$$\frac{(h_1 * r_1, t_1) \longrightarrow (h_2 * r_2, t_2)}{(h_1 * r_1, \mathcal{E}[t_1]) \longrightarrow (h_2 * r_2, \mathcal{E}[t_2])}\ \text{(R-CONTEXT)}$$

Figure 3.16: Reduction rules for states

current object at r), being defined in a slightly different way. The rule makes $r.f$ become the current object and wraps the method body t, prepared with the parameter substitution, in a return term that replaces the method call. Then, the body is reduced to an object in rule R-RETURN which also recovers the previous current object at r.

Rule R-CASE$_k$ means that either branch is taken, with the first having precedence over the second, i.e. the second branch is only tried if the condition $(h(r.f).\mathsf{class} = C_1)$ fails. The two rules R-IFTRUE and R-IFFALSE use the special true and false objects for the references that control the condition. In rule R-WHILE, the term is rewritten to a nested conditional, using top for the body of the else branch. Rule R-CONTEXT is standard for reduction in contexts, defining which term should be evaluated next.

$$
\begin{array}{rcl}
o : \mathsf{BST}[\{f : \mathsf{Nil} + \mathsf{Node}\}] * o & \vdash & \mathsf{case}\ f\ \mathsf{of}\ \mathsf{Nil} \Rightarrow f \coloneqq \mathsf{new}\ \mathsf{Node}()\ \mid\ \mathsf{Node} \Rightarrow f.m() \\
& \downarrow & (\text{R-CASE}_1) \\
o : \mathsf{BST}[\{f : \mathsf{Nil}\}] * o & \vdash & f \coloneqq \mathsf{new}\ \mathsf{Node}()\ \textbf{where}\ h(o.f).\mathsf{class} = \mathsf{Nil} \\
& \downarrow & (\text{R-EMPTY}) \\
o : \mathsf{BST}[\{f : \mathsf{Nil}\}] * o & \vdash & \epsilon\ \textbf{where}\ \mathsf{mbody}(\mathsf{init}, \mathsf{Node}) = \lambda().() \\
& \downarrow & (\text{R-NEW}) \\
o : \mathsf{BST}[\{f : \mathsf{Nil}\}] * o & \vdash & f \coloneqq o_2\ \textbf{where}\ h(o).f = o_1\ \textbf{and}\ o_2\ \mathsf{fresh} \\
& \downarrow & (\text{R-ASSIGN}) \\
o : \mathsf{BST}[\{f : \mathsf{Node}\}] * o & \vdash & o_1
\end{array}
$$

$$
\begin{array}{rcl}
o : \mathsf{BST}[\{f : \mathsf{Nil} + \mathsf{Node}\}] * o & \vdash & \mathsf{case}\ f\ \mathsf{of}\ \mathsf{Nil} \Rightarrow f \coloneqq \mathsf{new}\ \mathsf{Node}()\ \mid\ \mathsf{Node} \Rightarrow f.m() \\
& \downarrow & (\text{R-CASE}_2) \\
o : \mathsf{BST}[\{f : \mathsf{Node}\}] * o & \vdash & f.m()\ \textbf{where}\ h(o.f).\mathsf{class} = \mathsf{Node} \\
& \downarrow & (\text{R-CALL}) \\
o : \mathsf{BST}[\{f : \mathsf{Node}[\{\bar{f} : \bar{T}\}]\}] * o.f & \vdash & \mathsf{return}\ t\ \textbf{where}\ \mathsf{mbody}(m, \mathsf{Node}) = \lambda().t \\
& \downarrow & \vdots\ \ldots\ \textbf{where}\ (\_ * o.f, t) \longrightarrow (\_ * o.f, o_3) \\
o : \mathsf{BST}[\{f : \mathsf{Node}[\{\bar{f} : \bar{U}\}]\}] * o.f & \vdash & \mathsf{return}\ o_3 \\
& \downarrow & (\text{R-RETURN}) \\
o : \mathsf{BST}[\{f : \mathsf{Node}\}] * o & \vdash & o_3
\end{array}
$$

Figure 3.17: Two examples of reduction and typing (heaps, indices and rightmost typing context omitted), assuming a simplified initial heap of the form $h = (o = \mathsf{BST}\{f = o_1\})$

Example. In order to present an example of reduction and typing, we simplify method insert from class BST in Chapter 2, and then extract its body that becomes

$$\mathsf{case}\ f\ \mathsf{of}\ \mathsf{Nil} \Rightarrow f \coloneqq \mathsf{new}\ \mathsf{Node}()\ \mid\ \mathsf{Node} \Rightarrow f.m()$$

where f denotes the only field of class BST (which we named root in the example), m is a method of f (corresponding to method add in the example), and where we have ignored parameters as well as fields in class Node, with init being a constructor with an empty body. We also sacrifice types which cannot be fully detailed in the example for space reasons, omitting Σ and indices. The rather simplified initial typing context becomes

$$\mathsf{BST}[\{f : \mathsf{Nil} + \mathsf{Node}\}]$$

We then follow the execution of the above method body in Figure 3.17, showing how the typing context changes as the term reduces.

Chapter 4

Type Soundness

Having introduced the syntax and the static and dynamic semantics of DOL, in this chapter we prove by standard techniques [Wright and Felleisen, 1994] the expected result, that is, type soundness via subject reduction and progress.

We need to define a set of additional rules and prove a number of basic lemmas which support the soundness result, namely inversion of the term typing relation, exchange for object contexts, weakening for index contexts, substitution for objects in term typing, substitution for indices, substitution for class types, and agreement of judgements. We also prove soundness and instantiation of function mtype as well as two lemmas (opening and closing) for the replacement of an internal object type by an equivalent top-level one when typing the heap. We then show that in well-formed DOL programs the types of objects describe their runtime values, that there never exists more than one reference to a linear object, and that all aliasing never produce a value of unexpected type. Figure 4.1 shows a dependency graph of the main lemmas defined in this chapter.

Chapter Outline. This chapter is structured as follows:

- Section 4.1 gives a new set of rules for heaps and states that we need for the proofs.
- Section 4.2 defines basic properties of typing.
- Section 4.3 deals with the relation between an internal field typing and its top-level type in the style of Gay et al. [2015], adjusted to our requirements.
- Section 4.4 provides lemmas for evaluation contexts in the style of Wright and Felleisen [1994], adapted to our requirements.
- Section 4.5 proves the first key result, subject reduction.

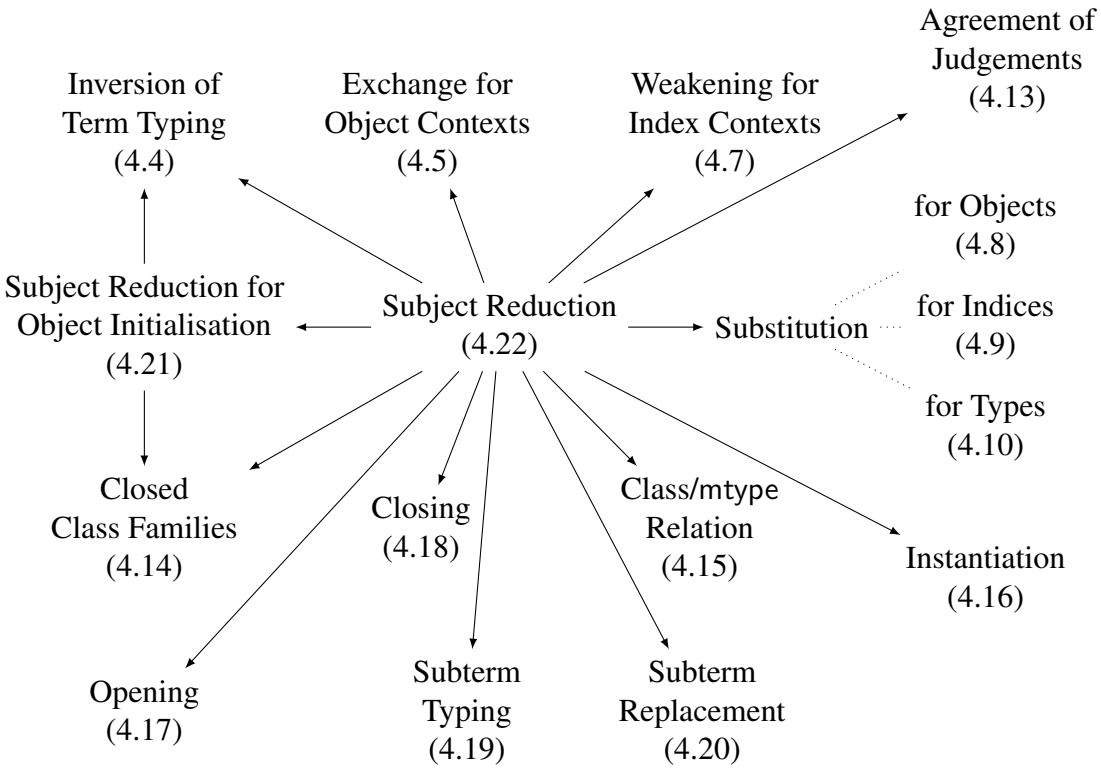


Figure 4.1: Dependency structure of the lemmas that support subject reduction

- Section 4.6 proves the second key result, progress.

4.1 State and Heap Typing

In order to establish type soundness, we first give additional definitions and relations that describe heaps and runtime states.

Definition 4.1 (Initial Heap and Object Context)**.** *In any well-formed program,* h_0 *and* Γ_0 *represent the initial heap and object context such that* $h_0 = (\mathsf{top} = \mathsf{Top}\{\}, \mathsf{false} = \mathsf{Boolean}\{\}, \mathsf{true} = \mathsf{Boolean}\{\})$ *and* $\Gamma_0 = (\mathsf{top} : \mathsf{Top}, \mathsf{false} : \mathsf{Boolean\ false}, \mathsf{true} : \mathsf{Boolean\ true})$.

Definition 4.2 (Children of Objects in the Heap)**.** *Let* h *be a heap. Then for any object* $o = C\{f_1 = o_1, \ldots, f_n = o_n\}$ *such that* $o \in h$ *with* $n \geq 0$*, the children of* o*, notation* $\mathsf{children}_h(o)$*, is defined as the set* $\{o_1, \ldots, o_n\}$.

Definition 4.3. *A heap* h *is complete if for any* $o \in \mathsf{dom}(h)$, $\mathsf{children}_h(o) \subseteq \mathsf{dom}(h)$.

In Figure 4.2 we give the additional set of typing rules. For typing the heap, we use a judgement of the form

$\boxed{\Delta;\Gamma \vdash h}$ Under contexts $\Delta;\Gamma$, heap h is well-formed

$$\frac{\Delta \vdash \Gamma}{\Delta;\Gamma \vdash \epsilon}\ \text{(T-EMPTYHEAP)}$$

$$\frac{\Delta;\Gamma_1 \vdash h \qquad \Delta;\Gamma_1 * o \vdash \bar{o} : F(\bar{f}) \dashv \Delta;\Gamma_2, o : C[F] * o}{\Delta;\Gamma_2, o : C[F] \vdash h, (o = C\{\bar{f} = \bar{o}\})}\ \text{(T-HEAP)}$$

$$\frac{\Delta;\Gamma, o : C[F] \vdash h \qquad \Delta;\Gamma, o : C[F] \vdash o : C\bar{i} \dashv \Delta}{\Delta;\Gamma, o : C\bar{i} \vdash h}\ \text{(T-HEAPHIDE)}$$

$\boxed{\Delta_1;\Gamma_1 \vdash S : T \dashv \Delta_2;\Gamma_2 * r}$ Under initial contexts $\Delta_1;\Gamma_1$, state S has type T, with final contexts $\Delta_2;\Gamma_2$ and path r

$$\frac{\begin{array}{c}h \text{ complete} \\ \mathsf{dom}(\Gamma_1) \subseteq \mathsf{dom}(h) \qquad \Delta_1;\Gamma_1 \vdash h \qquad \Delta_1;\Gamma_1 * r_1 \vdash t : T \dashv \Delta_2;\Gamma_2 * r_2\end{array}}{\Delta_1;\Gamma_1 \vdash (h * r_1, t) : T \dashv \Delta_2;\Gamma_2 * r_2}\ \text{(T-STATE)}$$

$\boxed{\Delta_1;\Gamma_1 \vdash (h * r, \bar{t}) : \bar{T} \dashv \Delta_2;\Gamma_2 * r}$ Under initial contexts $\Delta_1;\Gamma_1$, state $(h * r, \bar{t})$ has type $\bar{T}$, with final contexts $\Delta_2;\Gamma_2$ and the same path r

$$\frac{\begin{array}{c}h \text{ complete} \\ \mathsf{dom}(\Gamma_1) \subseteq \mathsf{dom}(h) \qquad \Delta_1;\Gamma_1 \vdash h \quad \Delta_1;\Gamma_1 * r \vdash \bar{t} : \bar{T} \dashv \Delta_2;\Gamma_2 * r\end{array}}{\Delta;\Gamma_1 \vdash (h * r, \bar{t}) : \bar{T} \dashv \Delta_2;\Gamma_2 * r}\ \text{(T-MULTISTATE)}$$

Figure 4.2: Typing rules for heaps and states

$$\Delta;\Gamma \vdash h$$

that states that under contexts $\Delta;\Gamma$ the heap h is well-formed. By rule T-EMPTYHEAP, a heap is constructed from typing contexts containing assumptions and types for all the objects relative to locations added to the heap by rule T-HEAP. The latter ensures that each heap entry has the prescribed field typing. The most important feature of this rule is that all aliases of linear references are explicitly forbidden by the rightmost premise. When typing sequenced objects $\bar{o}$ as part of a runtime state, we rely on rule T-MULTI (Figure 3.14) that uses T-UNVAR and T-LINVAR as appropriate to type each object. In particular, for each linear o_k in $o_1, \ldots, o_n$, with $1 \leq k \leq n$, the initial typing context contains o_k and the final one of the extended heap does not, meaning that a heap that contains multiple references to the same linear object is not be typable. The similar inverse argument justifies the existence of cyclic structures in the heap. Rule T-HEAPHIDE is used as needed in order to replace an internal object type by an equivalent top-level one (cf. Gay et al. [2015]).

Finally, we use the two following judgements

$$\Delta_1;\Gamma_1 \vdash S : T \dashv \Delta_2;\Gamma_2 * r \quad \text{and} \quad \Delta_1;\Gamma_1 \vdash (h * r, \bar{t}) : \bar{T} \dashv \Delta_2;\Gamma_2 * r$$

to type states and formalize the main invariant of subject reduction. By rule T-STATE, given a state S of the form $(h * r_1, t)$, the heap h must be compatible with a context Γ_1

under the assumptions in Δ_1, both of which are the initial contexts that type the runtime term t, knowing from the leftmost premises that h is complete and that $\mathsf{dom}(\Gamma_1) \subseteq \mathsf{dom}(h)$, i.e. that every object that has a type in Γ_1 appears in h along with all of its children. Rule T-MULTISTATE describes how the two parts of a runtime state for sequenced terms (composed of a heap h with path r and sequenced terms $\bar{t}$) are related by typing.

4.2 Properties of Typing

We now prove a number of basic lemmas.

4.2.1 Inversion

Inversion of the term typing relation is a simple result used throughout our proofs.

Lemma 4.4 (Inversion of the Term Typing Relation)**.** *Let* $\Delta_1; \Gamma_1 * r_1 \vdash t : T \dashv \Delta_2; \Gamma_2 * r_2$. *Then, depending on* t*, we have the following:*

1. *If* $t = o$*, then* $\Delta_1 \vdash \Gamma_1$ *with* $\Delta_2 = \Delta_1$ *and* $r_2 = r_1$ *and* $\Gamma_1 = (\Gamma_3, o : U)$ *for some* Γ_3 *and* U *such that* $\Delta_1 \vdash U <: T$*, and either*
   - $\mathsf{un}(U)$ *and* $\Gamma_2 = \Gamma_1$*, or*
   - $\mathsf{lin}(U)$ *and* $\Gamma_2 = \Gamma_3$.
2. *If* $t = f$*, then* $\Delta_1; \Gamma_1 \vdash r.f : U \dashv \Delta_2$ *and* $\mathsf{un}(U)$ *with* $\Delta_2 = \Delta_1$ *and* $\Gamma_2 = \Gamma_1$ *and* $r_2 = r_1$ *and* $\Delta_2 \vdash U <: T$.
3. *If* $t = \mathsf{new}\ C()$*, then* $\Delta_1 \vdash \Gamma_1$ *and* $\Delta_1 \vdash C.\mathsf{init} <: T$ *with* $\Delta_2 = \Delta_1$ *and* $\Gamma_2 = \Gamma_1$ *and* $r_2 = r_1$.
4. *If* $t = (f := t')$*, then* $\Delta_1; \Gamma_1 * r_1 \vdash t' : U \dashv \Delta_2; \Gamma_3 * r_2$ *and* $\Delta_2; \Gamma_3 \vdash r_2 : C[F] \dashv \Delta_2$ *and* $\Delta_2; \Gamma_2 \vdash r_2 : C\bar{i} \dashv \Delta_2$ *with* $\Gamma_2 = \Gamma_3\{r_2.f \hookleftarrow U\}$ *and* $\Delta_2 \vdash F(f) <: T$.
5. *If* $t = (t_1; t_2)$*, then* $\Delta_1; \Gamma_1 * r_1 \vdash t_1 : U_1 \dashv \Delta_3; \Gamma_3 * r_2$ *and* $\Delta_3; \Gamma_3 * r_2 \vdash t_2 : U_2 \dashv \Delta_2; \Gamma_2 * r_2$ *with* $\mathsf{un}(U_1)$ *and* $\Delta_2 \vdash U_2 <: T$.
6. *If* $t = m(t')$*, then* $\Delta_1; \Gamma_1 * r_1 \vdash t' : U[\theta] \dashv \Delta_3; \Gamma_3 * r_2$ *and* $\Delta_3; \Gamma_3 \vdash r_2 : C\bar{i} \dashv \Delta_3$ *and* $\Delta_3; \Gamma_3\{r_2 \hookleftarrow V[\theta]\} \vdash r_2 : C\bar{j} \dashv \Delta_2$ *with* $\Gamma_2 = \Gamma_3\{r_2 \hookleftarrow \mathsf{fields}(C\bar{j})\}$ *and* $\mathsf{mtype}(m, C\bar{i}) = \Pi\Delta.(C\bar{i} \rightsquigarrow V \times U \to W)$ *and* $\Delta_3 \vdash \Delta : \theta$ *and* $\Delta_2 \vdash W[\theta] <: T$.
7. *If* $t = f.m(t')$*, then* $\Gamma_1 = (\Gamma, r_1 : C[F])$ *and* $\Delta_1; \Gamma_1 * r_1 \vdash t' : U[\theta] \dashv \Delta_3; \Gamma_3 * r_2$ *and* $\Delta_3; \Gamma_3 \vdash r_2.f : T_1 \dashv \Delta_2$ *and* $\Delta_2; \Gamma_2 \vdash r_2 : C\bar{i} \dashv \Delta_2$ *with* $\Gamma_2 = \Gamma_3\{r_2.f \hookleftarrow$

$T_2[\theta]\}$ *and* $\mathsf{mtype}(m, T_1) = \Pi\Delta.(T_1 \rightsquigarrow T_2 \times U \to W)$ *and* $\Delta_2 \vdash \Delta : \theta$ *and* $\Delta_2 \vdash W[\theta] <: T$.

8. *If* $t = (\mathsf{return}\ t')$, *then* $\Delta_1; \Gamma_1 * r_1 \vdash t' : U \dashv \Delta_2; \Gamma_3 * r_2.f$ *and* $\Delta_2; \Gamma_3 \vdash r_2.f : C[F] \dashv \Delta_2$ *with* $\Delta_2 \vdash C[F] <: \mathsf{fields}(V)$ *and* $\Gamma_2 = \Gamma_3\{r_2.f \hookleftarrow V\}$ *and* $\Delta_2 \vdash U <: T$.

9. *If* $t = (\mathsf{case}\ f\ \mathsf{of}\ (C_k \Rightarrow t_k)_{k\in 1,2})$, *then* $\Delta_1; \Gamma_1 \vdash r_1.f : (U_1 + U_2) \dashv \Delta_3$ *and* $\Delta_3; \Gamma_1\{r_1.f \hookleftarrow U_k\} * r_1 \vdash t_k : U \dashv \Delta_2; \Gamma_2 * r_1$ *with* $\mathsf{classof}(U_k) = C_k$ *and* $C_1 \neq C_2$ *and* $\Delta_2 \vdash U <: T$ *and* $r_2 = r_1$.

10. *If* $t = (\mathsf{if}\ t'\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2)$, *then* $\Delta_1; \Gamma_1 * r_1 \vdash t' : \mathsf{Boolean}\ p \dashv \Delta_3; \Gamma_3 * r_2$ *and* $\Delta_3, p; \Gamma_3 * r_2 \vdash t_1 : U \dashv \Delta_2; \Gamma_2 * r_2$ *and* $\Delta_3, \neg p; \Gamma_3 * r_2 \vdash t_2 : U \dashv \Delta_2; \Gamma_2 * r_2$ *with* $\Delta_2 \vdash U <: T$.

11. *If* $t = (\mathsf{while}\ t_1\ \mathsf{do}\ t_2)$, *then* $\Delta_1; \Gamma_1 * r_1 \vdash t_1 : \mathsf{Boolean}\ p \dashv \Delta_3; \Gamma_2 * r_2$ *and* $\Delta_3, p; \Gamma_2 * r_2 \vdash t_2 : T \dashv \Delta_3; \Gamma_2 * r_2$ *with* $T = \mathsf{Top}$ *and* $\Delta_2 = (\Delta_3, \neg p)$.

Proof. By simple inspection of the typing rules. Any derivation concludes with $\Delta_1; \Gamma_1 * r_1 \vdash t : T \dashv \Delta_2; \Gamma_2 * r_2$, using the "natural" typing rule for t, followed by any number of applications of the rule T-SUB to obtain that result whenever $\Delta_1; \Gamma_1 * r_1 \vdash t : U \dashv \Delta_2; \Gamma_2 * r_2$. □

4.2.2 Exchange and Weakening

Towards type soundness, we establish basic structural properties satisfied by the core language. As noted in Chapter 3, the ordering of variables is irrelevant in the context Γ, and makes no difference in the heap h (but crucial in the context Δ). The first property, *exchange*, defined for Γ and h only, states exactly that. A corollary of exchange is that if we can typecheck a term with Γ, then we can typecheck that term with any permutation of the variables in Γ, and the same applies to h. We also define an *object swapping* lemma for rearranging objects in a sequence. Finally, *weakening*, defined for a context Δ, allows introducing new items in the index typing context without affecting typing relations.

Lemma 4.5 (Exchange for Object Contexts)**.**

1. *If* $\Delta_1; \Gamma_1 \vdash r : T \dashv \Delta_2$ *and* Γ_2 *is a permutation of* Γ_1, *then* $\Delta_1; \Gamma_2 \vdash r : T \dashv \Delta_2$.

2. *If* $\Delta_1; \Gamma_1 * r_1 \vdash t : T \dashv \Delta_2; \Gamma_2 * r_2$ *and* Γ_3 *is a permutation of* Γ_1, *then* $\Delta_1; \Gamma_3 * r_1 \vdash t : T \dashv \Delta_2; \Gamma_4 * r_2$ *where* Γ_4 *is a permutation of* Γ_2. *(And similarly for* $\Delta_1; \Gamma_1 * r \vdash \bar{t} : \bar{T} \dashv \Delta_2; \Gamma_2 * r$.*)*

3. *If* $\Delta; \Gamma \vdash h_1$ *and* h_2 *is a permutation of* h_1*, then* $\Delta; \Gamma \vdash h_2$.

Proof. By rule induction on typing derivations. Parts 1 and 2 are proved together. Part 3 then follows by another induction using part 2. □

Lemma 4.6 (Object Swapping)**.** *If* $\Delta; \Gamma_1 * r \vdash \bar{o}o\bar{o}'o' : \bar{T}T\bar{T}'U \dashv \Delta; \Gamma_2 * r$*, then* $\Delta; \Gamma_1 * r \vdash \bar{o}o'\bar{o}'o : \bar{T}U\bar{T}'T \dashv \Delta; \Gamma_2 * r$.

Proof. By a straightforward rearrangement of the derivation tree that concludes with rule T-MULTI. □

Lemma 4.7 (Weakening for Index Contexts)**.** *Let* $\vdash \Delta_1, \Delta_2$. *If* $\Delta_1 \vdash T : K$*, then* $\Delta_1, \Delta_2 \vdash T : K$. *(And similarly for* $\Delta_1 \vdash T <: U$*, for* $\Delta_1; \Gamma \vdash r : T \dashv \Delta_3$*, for* $\Delta_1; \Gamma_1 * r_1 \vdash t : T \dashv \Delta_3; \Gamma_2 * r_2$ *and for* $\Delta_1; \Gamma \vdash h$*.)*

Proof. By rule induction on derivations of the corresponding judgement provided as the second hypothesis. □

4.2.3 Substitution

Substitution is defined in the object and index systems. It allows substituting a parameter x by an object reference o in a term t, an index variable a by an index term i in a type T, and a type $D\bar{j}$ by a type $C\bar{i}$ in a type T, provided $C\bar{i}$ is a subtype of $D\bar{j}$. Regarding index substitution, recall that, from Section 3.2.2, a single index substitution is extended pointwise to multiple index substitution θ, built using the rules in Figure 3.6. We also define a composition of index substitutions by merging the two sets.

Lemma 4.8 (Substitution for Objects)**.** *If* $\Delta_1; \Gamma_1 * r \vdash o : U \dashv \Delta_1; \Gamma_2 * r$ *and* $\Delta_1; \Gamma_2 \vdash r : C[F_1] \dashv \Delta_1$ *and* $\Delta_1; x : U, \mathsf{this} : C[F_1] * \mathsf{this} \vdash t : W \dashv \Delta_2; \Gamma, \mathsf{this} : C[F_2] * \mathsf{this}$*, then* $\Delta_1; \Gamma_1 * r \vdash t[o/x] : W \dashv \Delta_2; \Gamma_2\{r \leftarrowtail C[F_2]\} * r$.

Proof. By rule induction on the structure of t. We show the base case and one inductive case. The rest can be proved similarly.

Case $t = x$**.** By Lemma 4.4 (Inversion of the Term Typing Relation) on the first hypothesis, there are two cases to consider: if $\mathsf{un}(U)$, then $\Gamma_2 = \Gamma_1$, otherwise $\Gamma_2 = \Gamma_1 \setminus o$. Use inversion again but on the third hypothesis, and make a similar unrestricted/linear distinction to conclude that, depending on the nature of U, either Γ is $x : U$ or Γ is empty. Finally, notice that in the case of $\mathsf{lin}(U)$, the path r cannot not start with o because of the second hypothesis, hence the current object r cannot be t, and all the premises are satisfied.

Case $t = t_1; t_2$**.** By Lemma 4.4 (Inversion of the Term Typing Relation) on the third hypothesis,

(1) $\Delta_1; x : U, \mathsf{this} : C[F_1] * \mathsf{this} \vdash t_1 : U_1 \dashv \Delta_3; \Gamma_3 * \mathsf{this}$

(2) $\Delta_3; \Gamma_3 * \mathsf{this} \vdash t_2 : U_2 \dashv \Delta_2; \Gamma, \mathsf{this} : C[F_2] * \mathsf{this}$

(3) $\mathsf{un}(U_1)$

(4) $\Delta_2 \vdash U_2 <: W$

Notice that because of the final context in (2), the initial context Γ_3 must be $(\Gamma', \mathsf{this} : C[F_3])$. Use the induction hypothesis with the first two hypothesis and (1) to obtain

(5) $\Delta_1; \Gamma_1 * r \vdash t_1[o/x] : U_1 \dashv \Delta_3; \Gamma_2\{r \hookleftarrow C[F_3]\} * r$

Without loss of generality, let Γ' be $x : U$. Then, replace the type of r in the initial context of the first hypothesis, which propagates to the second hypothesis as follows: $\Delta_3; \Gamma_1\{r \hookleftarrow C[F_3]\} * r \vdash o : U \dashv \Delta_3; \Gamma_2\{r \hookleftarrow C[F_3]\} * r$ and $\Delta_3; \Gamma_2\{r \hookleftarrow C[F_3]\} \vdash r : C[F_3] \dashv \Delta_3$. Use these judgements with (2) and the induction hypothesis to obtain

(6) $\Delta_3; \Gamma_2\{r \hookleftarrow C[F_3]\} * r \vdash t_2[o/x] : U_2 \dashv \Delta_2; \Gamma_2\{r \hookleftarrow C[F_2]\} * r$

Conclude by using (3),(4), (5) and (6) with rules T-SEQ and T-SUB. □

Lemma 4.9 (Substitution for Indices)**.** *Suppose* $\Delta_1 \vdash \Delta_2 : \theta$*.*

1. *If* $\Delta_2 \vdash T : K$*, then* $\Delta_1 \vdash T[\theta] : K[\theta]$*. (And similarly for* $\Delta_2 \vdash T <: U$*.)*
2. *If* $\Delta_2; \Gamma_1 * r_1 \vdash t : T \dashv \Delta_3; \Gamma_2 * r_2$*, then* $\Delta_1; \Gamma_1[\theta] * r_1 \vdash t : T[\theta] \dashv \Delta_3[\theta]; \Gamma_2[\theta] * r_2$*. (And similarly for* $\Delta_2; \Gamma \vdash r : T \dashv \Delta_3$*.)*

Proof. By rule induction on the derivation of the corresponding judgement provided as the hypothesis for each part. □

Lemma 4.10 (Substitution for Types)**.** *If* $\Delta \vdash T : K$ *and* $\Delta \vdash C\bar{i} <: D\bar{j}$*, then* $\Delta \vdash T[C\bar{i}/D\bar{j}] : K$*. (And similarly for* $\Delta \vdash T <: U$*.)*

Proof. By rule induction on the derivation of the judgement given as the first hypothesis. □

Lemma 4.11 (Substitution Composition)**.** *Let* $\vdash \Delta_1\Delta_2$*. If* $\Delta \vdash \Delta_1 : \theta_1$ *and* $\Delta \vdash \Delta_2 : \theta_2$*, then* $\Delta \vdash \Delta_1\Delta_2 : \theta_1\theta_2$*.*

Proof. By rule induction on the derivation of the second hypothesis, noticing that we follow Barendregt [1984]'s variable convention whereby a variable should appear no more than once, so this composition of substitutions is just their union. □

4.2.4 Agreement

In this section, we define properties that describe the behaviour of well-formed contexts, and show agreement between judgements, e.g. that every term in the object system has a type of kind $\star$. The next proposition is a direct consequence of the rules for well-formed programs and heaps (Figures 3.13 and 4.2)

Proposition 4.12 (Typing Context Properties)**.**

1. *If* $\Delta_1; \Gamma_1, o : T * r_1 \vdash t : U \dashv \Delta_2; \Gamma_2 * r_2$*, then* $\Delta_1 \vdash T : \star$*. (And similarly for* $\Delta_1; \Gamma, o : T \vdash r : U \dashv \Delta_2$*.)*

2. *If* $\Delta; \Gamma \vdash r : C[\{f_1 : T_1, \ldots, f_n : T_n\}] \dashv \Delta$*, then* $\Delta \vdash T_k : \star$ *for* $1 \leq k \leq n$

3. *If* $\Delta; \Gamma \vdash h$*, then* $\Delta \vdash \Gamma$*.*

Lemma 4.13 (Agreement of Judgements)**.**

1. *If* $\Delta \vdash T : K$*, then* $\Delta \vdash K$*.*

2. *If* $\Delta_1; \Gamma_1 * r_1 \vdash t : T \dashv \Delta_2; \Gamma_2 * r_2$*, then* $\Delta_2 \vdash T : \star$*. (And similarly for* $\Delta_1; \Gamma \vdash r : T \dashv \Delta_2$*.)*

3. *If* $\Delta \vdash T <: U$*, then* $\Delta \vdash T : K$ *and* $\Delta \vdash U : K$ *for some* K*.*

Proof. By rule induction on the derivations of the hypothesis; parts 2 and 3 are proved together. Use Proposition 4.12 (Typing Context Properties) for rules T-REF, T-UNVAR, T-LINVAR and T-UNFIELD, Lemma 4.9 (Substitution for Indices) for rules T-SELFCALL, T-CALL, K-APP, S-APP, S-ΠL and S-ΣR. The remaining cases follow by simple inductions. □

The next lemma concerns a property of object creation, stating that any initialisation does not have side effects. This is a consequence of rule T-INIT (Figure 3.13) by which the constructor may only initialise local objects using new. In practice, this means that new $C()$ does depend on the initial and final typing contexts.

Lemma 4.14 (Closed Class Families)**.** *Let* $\Delta \vdash \Gamma$*. If* $\Delta_1; \Gamma_1 * r \vdash \mathsf{new}\ \bar{C}() : \bar{T} \dashv \Delta_2; \Gamma_2 * r$*, then* $\Delta; \Gamma * r \vdash \mathsf{new}\ \bar{C}() : \bar{T} \dashv \Delta; \Gamma * r$*.*

Proof. By induction on the length of new $\bar{C}()$. □

4.2.5 Soundness of Function mtype

We use the properties of substitution to prove soundness of function mtype, first by relating it with the hierarchy of classes and then by showing its instantiation with function mbody.

Lemma 4.15 (Class/mtype Relation)**.** *If* $\Delta_1 \vdash C\bar{i} : \star$ *and* $\mathsf{mtype}(m, C\bar{i}) = U$*, then* $\Delta_1 \vdash U : \star$.

Proof. By rule induction on the derivation of $\mathsf{mtype}(m, C\bar{i})$.

Case MT-CLASS**.** Then,

$$\begin{aligned}&\mathsf{class}\ C : (\bar{a} : \bar{I})\ \mathsf{extends}\ T\{\ldots, m : V, \ldots\}\ \mathsf{is}\ \{\_\}\\ &U = V[\bar{i}/\bar{a}]\end{aligned}$$

Let $\Delta_2 = \bar{a} : \bar{I}$ and $\theta = \bar{i}/\bar{a}$. Notice that the hypothesis $\Delta_1 \vdash C\bar{i} : \star$ is derived from one application of rule K-CLASS possibly followed by rule K-APP applied as many times as needed. Use this derivation to build $\Delta_1 \vdash \Delta_2 : \theta$. On the other hand, by rules T-CLASS and T-MTYPE, we have $\Delta_2 \vdash_T m : V$ whose premises imply that $\Delta_2 \vdash V : \star$. Then, apply Lemma 4.9 (Substitution for Indices) to conclude with $\Delta_1 \vdash V[\theta] : \star$.

Case MT-SUPER**.** Then,

$$\begin{aligned}&\mathsf{class}\ C : (\bar{a} : \bar{I})\ \mathsf{extends}\ D\bar{j}\{\bar{l} : \_\}\ \mathsf{is}\ \{\_\}\\ &m \notin \bar{l}\\ &C\bar{i} = C\bar{i}'\bar{i}''\\ &\mathsf{mtype}(m, D\bar{j}[\bar{i}/\bar{a}]) = W\\ &U = W[C\bar{i}'/D]\end{aligned}$$

Let $\Delta_2 = \bar{a} : \bar{I}$ and $\theta = \bar{i}/\bar{a}$. As in the previous case, use the fact that the hypothesis $\Delta_1 \vdash C\bar{i} : \star$ is derived by one application of rule K-CLASS possibly followed by rule K-APP applied as many times as needed in order to build $\Delta_1 \vdash \Delta_2 : \theta$. By rule T-CLASS, we have $\Delta_2 \vdash D\bar{j} : \star$. Apply Lemma 4.9 (Substitution for Indices) to obtain $\Delta_1 \vdash D\bar{j}[\theta] : \star$. From this and $\mathsf{mtype}(m, D\bar{j}[\theta]) = W$, use the induction hypothesis to get $\Delta_1 \vdash W : \star$. Conclude with Lemma 4.10 (Substitution for Types) by using $\Delta_1 \vdash C\bar{i} <: D\bar{j}[\theta]$ (implied by extends). □

Lemma 4.16 (Instantiation)**.** *If* $\mathsf{mbody}(m, C) = \lambda x.t$ *and* $\Delta_1 \vdash T_1 : \star$ *and* $\mathsf{mtype}(m, T_1) = \Pi\Delta_2.(T_1 \rightsquigarrow T_2 \times U \to W)$ *and* $\Delta_1 \vdash \Delta_2 : \theta$*, then there exist contexts* Δ_3 *and* Γ *and a field typing* F *such that*

$$\Delta_1; x : U[\theta], \mathsf{this} : \mathsf{fields}(T_1) * \mathsf{this} \vdash t : W[\theta] \dashv \Delta_3; \Gamma, \mathsf{this} : C[F] * \mathsf{this}$$

with $\Delta_3 \vdash C[F] <: \mathsf{fields}(T_2[\theta])$.

Proof. By rule induction on the derivation of $\mathsf{mbody}(m, C)$.

Case MB-CLASS**.** Then,

$$\mathsf{class}\ C : \Delta\ \mathsf{extends}\ \_\{\_\}\ \mathsf{is}\ \{\ldots, m(x) = t, \ldots\}$$

By rules T-CLASS and T-METHOD, we have $\vdash_C m(x) = t$ whose premises are:

$$\begin{array}{l}
\mathsf{class}\ C : \Delta\ \mathsf{extends}\ _\{\ldots, m : \Pi\Delta'.(T \rightsquigarrow T' \times U' \to W'), \ldots\}\ \mathsf{is}\ \{_\} \\
\Delta, \Delta'; x : U', \mathsf{this} : \mathsf{fields}(T) * \mathsf{this} \vdash t : W' \dashv \Delta''; \Gamma', \mathsf{this} : C[F'] * \mathsf{this} \\
x : U' \in \Gamma' \Rightarrow \mathsf{un}(U') \\
\Delta'' \vdash C[F'] <: \mathsf{fields}(T') \\
m \neq \mathsf{init}
\end{array}$$

Let $\Delta = \bar{a} : \bar{I}$ and $V = \Pi\Delta'.(T \rightsquigarrow T' \times U' \to W')$. By definition of $\mathsf{mtype}(m, T_1)$, we have that type T_1 is of the form $C\bar{i}$, and hence $\Delta_1 \vdash C\bar{i} : \star$ and $\mathsf{mtype}(m, C\bar{i}) = \Pi\Delta_2.(C\bar{i} \rightsquigarrow T_2 \times U \to W)$ (by inspection of rules MT-CLASS and T-MTYPE). Let $\theta' = \bar{i}/\bar{a}$. Then, notice that $\Delta_1 \vdash C\bar{i} : \star$ is derived from one application of rule K-CLASS possibly followed by rule K-APP applied as many times as needed. Use this derivation to build $\Delta_1 \vdash \Delta : \theta'$. From rule MT-CLASS, we have $V[\theta'] = \Pi\Delta_2.(C\bar{i} \rightsquigarrow T_2 \times U \to W)$, which implies that $\Delta_2 = \Delta'[\theta']$. By Lemma 4.11 (Substitution Composition), obtain $\Delta_1 \vdash \Delta\Delta_2 : \theta'\theta$. Then, use $\Delta_1 \vdash C\bar{i} : \star$ and $\mathsf{mtype}(m, C\bar{i}) = V[\theta']$ with Lemma 4.15 (Class/mtype Relation) to get $\Delta_1 \vdash V[\theta'] : \star$. Conclude by applying Lemma 4.9 (Substitution for Indices) to the appropriate typing judgements in the premises of T-METHOD given above.

Case MB-SUPER**.** Then,

$$\begin{array}{l}
\mathsf{class}\ C : \Delta\ \mathsf{extends}\ D\bar{j}\{_\}\ \mathsf{is}\ \{\bar{M}\} \\
m \notin \bar{M}
\end{array}$$

Let $\Delta = \bar{a} : \bar{I}$ and $\theta = \bar{i}/\bar{a}$. As in the previous case, use the fact that the hypothesis $\Delta_1 \vdash C\bar{i} : \star$ is derived by one application of rule K-CLASS possibly followed by rule K-APP applied as many times as needed in order to build $\Delta_1 \vdash \Delta : \theta$. By rule T-CLASS, we have $\Delta \vdash D\bar{j} : \star$. Apply Lemma 4.9 (Substitution for Indices) to obtain $\Delta_1 \vdash D\bar{j}[\theta] : \star$. From this, conclude immediately by using the induction hypothesis and

the fact that $\Delta_1 \vdash C\bar{i} <: D\bar{j}[\theta]$ (implied by extends). □

4.3 Hiding Field Typings

The next step towards type soundness is to prove that the internal object type $C[F]$ can be replaced by a top-level equivalent form that hides field typings (and vice-versa) in the judgement that types the heap. To break down the proof into smaller pieces, we define the notion of *opening* a top-level type and *closing* the internal object type using two separate lemmas, adapted from Gay et al. [2015] to our requirements. They enable us to show that hiding and showing type information about the fields does not violate type preservation.

Lemma 4.17 (Opening)**.** *If* $\Delta_1; \Gamma \vdash h$ *and* $\Delta_1; \Gamma \vdash r : C\bar{i} \dashv \Delta_2$ *and* $h(r)$ *is defined, then* $\Delta_2; \Gamma\{r \hookleftarrow \mathsf{fields}(C\bar{i})\} \vdash h$*.*

Proof. By induction on the depth of r. The base case is when $r = o$. Notice that the second hypothesis is derived by rule T-REF with as many applications of T-UNPACK as needed, and hence $r \in \mathsf{dom}(\Gamma)$ and $\Delta_1 \subseteq \Delta_2$. By Lemma 4.13 (Agreement of Judgements), we have $\Delta_2 \vdash C\bar{i} : \star$, and from this derivation $\vdash \Delta_2$. Use Lemma 4.7 (Weakening for Index Contexts) on the first hypothesis to derive $\Delta_2; \Gamma', o : C\bar{i} \vdash h$, which must be the conclusion of rule T-HEAPHIDE. Then, $\Delta_2; \Gamma \vdash o : C\bar{i} \dashv \Delta_2$ must be its premise at the right. Conclude with the rule's leftmost premise.

The inductive case is when $r = o'.f.\bar{f}$. The typing derivation ends with rule T-HEAP, and field $o'.f$ references some object o'' typed in the rule premises by T-MULTI. Let $r' = o''.\bar{f}$. Knowing from the hypothesis that $\Delta_1; \Gamma \vdash r : C\bar{i} \dashv \Delta_2$, the only way that the type of r' in Γ can differ from $C\bar{i}$ is if it is a Σ, hence by rule T-UNPACK $\Delta_1 \subseteq \Delta_2$. Conclude with the induction hypothesis by using Lemma 4.7 (Weakening for Index Contexts) and replacing Γ with $\Gamma\{r \hookleftarrow \mathsf{fields}(C\bar{i})\}$ on the first hypothesis. Then just use it as the left premise of T-HEAP, propagating the new type of r to the remaining premises and the conclusion. □

Lemma 4.18 (Closing)**.** *If* $\Delta; \Gamma \vdash h$ *and* $\Delta; \Gamma \vdash r : C[F] \dashv \Delta$ *and* $\Delta \vdash C[F] <: \mathsf{fields}(T)$*, then* $\Delta; \Gamma\{r \hookleftarrow T\} \vdash h$*.*

Proof. Again by induction on the depth of r. The base case is when $r = o$. Notice that $\Delta; \Gamma \vdash r : C[F] \dashv \Delta$ is necessarily the conclusion of T-REF, which implies that $r \in \mathsf{dom}(\Gamma)$. Conclude by applying rules T-HIDE and T-HEAPHIDE using the three hypothesis.

The inductive case is similar to the one from the preceding lemma: in the last application of rule T-HEAP, look at the type of r' (defined above) and notice that its use in rule

T-MULTI implies that it can only differ from the type of r by subsumption, and hence its type must be $C[F']$ with $\Delta \vdash C[F'] <: C[F]$. By transitivity, $\Delta \vdash C[F'] <: \mathsf{fields}(T)$. Use the induction hypothesis to change the type of r' to T in the leftmost premise of rule T-HEAP, which propagates to the type of r in the conclusion. □

4.4 Properties of Evaluation Contexts

Next, we provide lemmas for evaluation contexts in the style of Wright and Felleisen [1994], conveniently adapted to our requirements. First, we show that terms in holes are typable. Then, we prove that a subterm t_1 may be replaced by a different subterm t_2 with the same type. Unlike the usual replacement lemma, our lemma allows changing typing contexts for the term being replaced.

Lemma 4.19 (Subterm Typing)**.** *If D_1 is a derivation of the judgement $\Delta_1; \Gamma_1 * r_1 \vdash \mathcal{E}[t] : T \dashv \Delta_2; \Gamma_2 * r_2$, then there exists a subderivation D_2 concluding $\Delta_1; \Gamma_1 * r_1 \vdash t : U \dashv \Delta_3; \Gamma_3 * r_3$ for some U, Δ_3, Γ_3 and r_3 such that the position of D_2 in D_1 corresponds to the position of the hole in $\mathcal{E}$.*

Proof. By induction on the structure of $\mathcal{E}$. In any subderivation of D_1, the possible positions of the hole determine that the initial contexts are always $\Delta_1; \Gamma_1$. Suppose a term t' such that $\Delta_1; \Gamma_1 * r_1 \vdash t' : U \dashv \Delta_3; \Gamma_3 * r_3$. It suffices to show that $\Delta_1; \Gamma_1 * r_1 \vdash \mathcal{E}[t'] : T \dashv \Delta_2; \Gamma_2 * r_2$. All cases are similar. We provide a few examples.

Case $[\_]$**.** Then $\mathcal{E}[t] = t$. Let $U = T$ and $\Delta_3 = \Delta_2$ and $\Gamma_3 = \Gamma_2$ and $r_3 = r_2$. Note that $\mathcal{E}[t'] = t'$. Then, $\Delta_1; \Gamma_1 * r_1 \vdash t' : U \dashv \Delta_3; \Gamma_3 * r_3$.

Case $f \coloneqq \mathcal{E}'$**.** That is,

(1) $\Delta; \Gamma_1 * r_1 \vdash f \coloneqq \mathcal{E}'[t'] : T \dashv \Delta_2; \Gamma_2 * r_2$

By Lemma 4.4 (Inversion of the Term Typing Relation),

(2) $\Delta_1; \Gamma_1 * r_1 \vdash \mathcal{E}'[t'] : U \dashv \Delta_2; \Gamma_4 * r_2$

(3) $\Delta_2; \Gamma_4 \vdash r_2 : C[F] \dashv \Delta_2$

(4) $\Delta_2; \Gamma_2 \vdash r_2 : C\bar{i} \dashv \Delta_2$

(5) $\Gamma_2 = \Gamma_4\{r_1.f \hookleftarrow U\}$

(6) $\Delta_2 \vdash F(f) <: T$

By (2) and the induction hypothesis, there is a subderivation concluding $\Delta; \Gamma_1 * r_1 \vdash t' : U' \dashv \Delta_3; \Gamma_3 * r_3$ for some U', Δ_3, Γ_3 and r_3. This is the desired subderivation of D_1.

Case $\mathsf{return}\ t'$**.** That is,

(1) $\Delta_1; \Gamma_1 * r_1 \vdash \mathsf{return}\ \mathcal{E}'[t'] : T \dashv \Delta_2; \Gamma_2 * r_2$

By Lemma 4.4 (Inversion of the Term Typing Relation),

(2) $\Delta_1; \Gamma_1 * r_1 \vdash \mathcal{E}'[t'] : V \dashv \Delta_2; \Gamma_4 * r_2.f$

(3) $\Delta_2; \Gamma_4 \vdash r_2.f : C[F] \dashv \Delta_2$

(4) $\Delta_2 \vdash C[F] <: \mathsf{fields}(U)$

(5) $\Gamma_2 = \Gamma_4\{r_2.f \leftarrowtail U\}$

(6) $\Delta_2 \vdash V <: T$

By (2) and the induction hypothesis, there is a subderivation concluding $\Delta; \Gamma_1 * r_1 \vdash t' : U \dashv \Delta_3; \Gamma_3 * r_3$ for some U', Δ_3, Γ_3 and r_3. This is the desired subderivation of D_1.

Case $\mathcal{E}'; t_2$**.** That is,

(1) $\Delta; \Gamma_1 * r_1 \vdash \mathcal{E}'[t']; t_2 : T \dashv \Delta_2; \Gamma_2 * r_2$

By Lemma 4.4 (Inversion of the Term Typing Relation),

(2) $\Delta; \Gamma_1 * r_1 \vdash \mathcal{E}'[t'] : U \dashv \Delta_4; \Gamma_4 * r_2$

(3) $\Delta_4; \Gamma_4 * r_2 \vdash t_2 : V \dashv \Delta_2; \Gamma_2 * r_2$

(4) $\mathsf{un}(U)$

(5) $\Delta_2 \vdash V <: T$

By (2) and the induction hypothesis, there is a subderivation concluding $\Delta_1; \Gamma_1 * r_1 \vdash t' : U' \dashv \Delta_3; \Gamma_3 * r_3$ for some U', Δ_3, Γ_3 and r_3. This is the desired subderivation of D_1. □

Lemma 4.20 (Subterm Replacement)**.** *If*

1. *D_1 is a derivation concluding $\Delta_1; \Gamma_1 * r_1 \vdash \mathcal{E}[t_1] : T \dashv \Delta_2; \Gamma_2 * r_2$, and*
2. *D_2 is a subderivation of D_1 concluding $\Delta_1; \Gamma_1 * r_1 \vdash t_1 : U \dashv \Delta_3; \Gamma_3 * r_3$, and*
3. *the position of D_2 in D_1 corresponds to the position of the hole in $\mathcal{E}$, and*
4. *$\Delta_1'; \Gamma_1' * r_1' \vdash t_2 : U \dashv \Delta_3; \Gamma_3' * r_3$ for some Γ_3' such that $\Gamma_3 \subseteq \Gamma_3'$,*

then $\Delta_1'; \Gamma_1' * r_1' \vdash \mathcal{E}[t_2] : T \dashv \Delta_2; \Gamma_2' * r_2$ for some Γ_2' such that $\Gamma_2 \subseteq \Gamma_2'$.

Proof. By induction on the structure of $\mathcal{E}$ using Lemma 4.19 (Subterm Typing). All cases are similar: replace D_2 in D_1 by a derivation of $\Delta_1'; \Gamma_1' * r_1' \vdash t_2 : U \dashv \Delta_3; \Gamma_3' * r_3$. From the structure of $\mathcal{E}$, the typings of t_2 and $\mathcal{E}[t_2]$ have the same leftmost contexts. We show a few examples.

Case $[\_]$**.** Let $\Delta_1' = \Delta_1$ and $\Gamma_1' = \Gamma_1$ and $r_1' = r_1$ and $\Gamma_2' = \Gamma_2$.

Case $f \coloneqq \mathcal{E}'$**.** That is,

(1) $\Delta_1; \Gamma_1 * r_1 \vdash f \coloneqq \mathcal{E}'[t_1] : T \dashv \Delta_2; \Gamma_2 * r_2$

By Lemma 4.4 (Inversion of the Term Typing Relation),

(2) $\Delta_1; \Gamma_1 * r_1 \vdash \mathcal{E}'[t_1] : U \dashv \Delta_2; \Gamma_3 * r_2$

(3) $\Delta_2; \Gamma_3 \vdash r_2 : C[F] \dashv \Delta_2$

(4) $\Delta_2; \Gamma_2 \vdash r_2 : C\bar{i} \dashv \Delta_2$

(5) $\Gamma_2 = \Gamma_3\{r_2.f \hookleftarrow U\}$

(6) $\Delta_2 \vdash F(f) <: T$

Let D_1 be the derivation concluding (2). Then, by Lemma 4.19 (Subterm Typing), D_1 has a subderivation D_2 concluding

(7) $\Delta_1; \Gamma_1 * r_1 \vdash t_1 : U' \dashv \Delta_3'; \Gamma_3' * r_3$

such that the position of D_2 in D_1 corresponds to the position of the hole in $\mathcal{E}'$. Assume there exist $\Delta_1', \Gamma_1', r_1', t_2$ and $\Gamma_3' \subseteq \Gamma_4'$ such that

(8) $\Delta_1'; \Gamma_1' * r_1' \vdash t_2 : U' \dashv \Delta_3'; \Gamma_4' * r_3$

By the induction hypothesis with (2), (7) and (8), there is a Γ_4 such that $\Delta_1'; \Gamma_1' * r_1' \vdash \mathcal{E}'[t_2] : U \dashv \Delta_2; \Gamma_4 * r_2$ with $\Gamma_3 \subseteq \Gamma_4$. From (3) and rule T-REF, $r_2 \in \mathsf{dom}(\Gamma_3)$. Then use the fact that $\Gamma_3 \subseteq \Gamma_4$ to obtain $\Delta_2; \Gamma_4 \vdash r_2 : C[F] \dashv \Delta_2$ and $\Delta_2; \Gamma_2' \vdash r_2 : C\bar{i} \dashv \Delta_2$, with $\Gamma_2' = \Gamma_4\{r_2.f \hookleftarrow U\}$. Combine these two judgements with the one obtained by induction and (6), and conclude by applying T-ASSIGN and T-SUB.

Case $\mathcal{E}'; t_2$**.** That is,

(1) $\Delta_1; \Gamma_1 * r_1 \vdash \mathcal{E}'[t_1]; t_2 : T \dashv \Delta_2; \Gamma_2 * r_2$

By Lemma 4.4 (Inversion of the Term Typing Relation),

(2) $\Delta_1; \Gamma_1 * r_1 \vdash \mathcal{E}'[t_1] : U \dashv \Delta_3'; \Gamma_3 * r_2$

(3) $\Delta_3'; \Gamma_3 * r_2 \vdash t_2 : V \dashv \Delta_2; \Gamma_2 * r_2$

(4) $\mathsf{un}(U)$

(5) $\Delta_2 \vdash V <: T$

Let D_1 be the derivation concluding (2). Then, by Lemma 4.19 (Subterm Typing), D_1 has a subderivation D_2 concluding

(6) $\Delta_1; \Gamma_1 * r_1 \vdash t_1 : U' \dashv \Delta_4; \Gamma_3' * r_3$

such that the position of D_2 in D_1 corresponds to the position of the hole in $\mathcal{E}'$. Assume there exist $\Delta_1', \Gamma_1', r_1', t_3$ and $\Gamma_3' \subseteq \Gamma_4'$ such that

(7) $\Delta_1'; \Gamma_1' * r_1' \vdash t_3 : U' \dashv \Delta_4; \Gamma_4' * r_3$

By the the induction hypothesis with (2), (6) and (7), there is a Γ_4 such that $\Delta_1'; \Gamma_1' * r_1' \vdash \mathcal{E}'[t_3] : U \dashv \Delta_3'; \Gamma_4 * r_2$ with $\Gamma_3 \subseteq \Gamma_4$. From this, replace Γ_3 with Γ_4 in (3) in order to obtain $\Delta_3'; \Gamma_4 * r_2 \vdash t_2 : V \dashv \Delta_2; \Gamma_2' * r_2$. Conclude from these two typing judgements by applying T-SEQ and T-SUB using (4) and (5). □

4.5 Subject Reduction

The most important property of DOL's system is subject reduction, which ensures that reduction preserves typings. For object initialisation, subject reduction is a matter of showing that newly created objects are not already used in the heap, and that all the children objects are initialised and have the type declared by the location (field) they represent. We prove this result in a separate lemma and then use it in the proof of the theorem.

Lemma 4.21 (Subject Reduction for Object Initialisation)**.** *Suppose that P is a well-formed program ($\vdash P$). In this context, let $\Gamma_0 \subseteq \Gamma_1$ and $h_0 \subseteq h_1$.*

1. *If $\Delta; \Gamma_1 \vdash (h_1 * r, \mathsf{new}\ C()) : T \dashv \Delta; \Gamma_1 * r$ and $(h_1 * r, \mathsf{new}\ C()) \longrightarrow (h_2 * r, o)$, then $\Delta; \Gamma_1' \vdash (h_2 * r, o) : T \dashv \Delta; \Gamma_2 * r$ for some Γ_1' and Γ_2 such that $\Gamma_1 \subseteq \Gamma_2$.*

2. *If $\Delta; \Gamma_1 \vdash (h_1 * r, \mathsf{new}\ \bar{C}()) : \bar{T} \dashv \Delta; \Gamma_1 * r$ and $(h_1 * r, \mathsf{new}\ \bar{C}()) \longrightarrow (h_2 * r, \bar{o})$, then $\Delta; \Gamma_1' \vdash (h_2 * r, \bar{o}) : \bar{T} \dashv \Delta; \Gamma_2 * r$ for some Γ_1' and Γ_2 such that $\Gamma_1 \subseteq \Gamma_2$.*

Proof. By mutual induction on the premises of reduction.

Case R-NEW**.** Then,

$$\begin{aligned}
&\mathsf{mbody}(\mathsf{init}, C) = \bar{f} := \mathsf{new}\ \bar{C}() \\
&(h_1 * r, \mathsf{new}\ \bar{C}()) \longrightarrow (h_3 * r, \bar{o}) \\
&o \notin \mathsf{dom}(h_3) \\
&h_2 = (h_3, o = C\{\bar{f} = \bar{o}\})
\end{aligned}$$

Notice that a well-formed program ($\vdash P$) and $\mathsf{mbody}(\mathsf{init}, C)$ imply that the method init appears in the declaration of class C. From rule T-CLASS, $\vdash_C \mathsf{init}() = \bar{f} := \mathsf{new}\ \bar{C}()$ which is necessarily the conclusion of rule T-INIT whose premises are:

(i) $\mathsf{fields}(C.\mathsf{init}) = C[F]$

(ii) $\epsilon; \epsilon * r \vdash \mathsf{new}\ \bar{C}() : F(\bar{f}) \dashv \epsilon; \epsilon * r$

(iii) no cycles in C

We claim that the final state typing $\Delta; \Gamma_1' \vdash (h_2 * r, o) : T \dashv \Delta; \Gamma_2 * r$ has an initial context $\Gamma_1' = (\Gamma_3', o : C.\mathsf{init})$ with $\Gamma_3' = (\Gamma_1, \Gamma)$ where Γ depends on the unrestricted/-linear nature of each object o_k in $\bar{o} = o_1, \ldots, o_n$ and $1 \leq k \leq n$ that may have been "consumed" along the derivation of the judgement, or may be carried unchanged from Γ_1' to Γ_2. Moreover, depending on the unrestricted/linear nature of type $C.\mathsf{init}$, context Γ_2 is either Γ_1' or $\Gamma_1' \setminus o$. Either way, notice that $\Gamma_1 \subseteq \Gamma_2$.

Proving the claim proceeds through a number of steps, each contributing to build the final state typing. Start by reading the premises of rule T-STATE from the hypothesis, which are:

(1) h_1 complete

(2) $\mathsf{dom}(\Gamma_1) \in \mathsf{dom}(h_1)$

(3) $\Delta; \Gamma_1 \vdash h_1$

(4) $\Delta; \Gamma_1 * r \vdash \mathsf{new}\ C() : T \dashv \Delta; \Gamma_1 * r$

By Lemma 4.4 (Inversion of the Term Typing Relation) on (4), obtain $\Delta_1 \vdash \Gamma_1$. Use it with (ii) and Lemma 4.14 (Closed Class Families) to get $\Delta; \Gamma_1 * r \vdash \mathsf{new}\ \bar{C}() : F(\bar{f}) \dashv \Delta; \Gamma_1 * r$. Combine this result with (1), (2) and (3) and apply rule T-MULTISTATE, $\Delta; \Gamma_1 \vdash (h_1 * r, \mathsf{new}\ \bar{C}()) : F(\bar{f}) \dashv \Delta; \Gamma_1 * r$. From here, do an induction using the second premise of R-NEW to get Γ_3, Γ_3' such that $\Delta; \Gamma_3 \vdash (h_3 * r, \bar{o}) : F(\bar{f}) \dashv \Delta; \Gamma_3' * r$ with $\Gamma_1 \subseteq \Gamma_3'$. This judgement is necessarily the conclusion of T-MULTISTATE whose premises are:

(5) h_3 complete

(6) $\mathsf{dom}(\Gamma_3) \in \mathsf{dom}(h_3)$

(7) $\Delta; \Gamma_3 \vdash h_3$

(8) $\Delta; \Gamma_3 * r \vdash \bar{o} : F(\bar{f}) \dashv \Delta; \Gamma'_3 * r$

Since o is fresh, build a derivation that adds o to the heap, and by which (7) and (8) become

(9) $\Delta; \Gamma_3, o : C[F] \vdash h_3$

(10) $\Delta; \Gamma_3, o : C[F] * r \vdash \bar{o} : F(\bar{f}) \dashv \Delta; \Gamma'_3, o : C[F] * r$

Use these judgements in the premises of rule T-HEAP; follow by rule T-HEAPHIDE with $\Delta; \Gamma'_3, o : C[F] \vdash o : C.\mathsf{init} \dashv \Delta$ to obtain $\Delta; \Gamma'_3, o : C.\mathsf{init} \vdash h_2$. By Proposition 4.12 (Typing Context Properties), $\Delta \vdash (\Gamma'_3, o : C.\mathsf{init})$. By Lemma 4.4 (Inversion of the Term Typing Relation) on (4), $\Delta \vdash C.\mathsf{init} <: T$. Use these two judgements with T-UNVAR or T-LINVAR as appropriate, followed by T-SUB, to deduce $\Delta; \Gamma'_3, o : C.\mathsf{init} * r \vdash o : T \dashv \Delta; \Gamma_2 * r$. Notice that r cannot start with o since o is fresh. Also, (5) and (6) imply h_2 complete and $\mathsf{dom}(\Gamma'_3, o : C.\mathsf{init}) \in \mathsf{dom}(h_2)$. From this, conclude with rule T-STATE by using the result from T-HEAPHIDE and the one from T-SUB.

Case R-MULTINEW**.** Let $\mathsf{new}\ \bar{C}() = \mathsf{new}\ C()\mathsf{new}\ \bar{C}'()$ and $\bar{T} = T\bar{T}'$ with $\mathsf{new}\ \bar{C}'() \neq \epsilon$. Then,

$$
\begin{aligned}
&(h_1 * r, \mathsf{new}\ C()) \longrightarrow (h_3 * r, o) \\
&(h_3 * r, \mathsf{new}\ \bar{C}'()) \longrightarrow (h_2 * r, \bar{o}') \\
&\mathsf{new}\ \bar{C}'() = \mathsf{new}\ C()\mathsf{new}\ \bar{C}'() \\
&\bar{o} = o\bar{o}'
\end{aligned}
$$

Notice that the hypothesis is necessarily the conclusion of T-MULTISTATE whose premises are:

(1) h_1 complete

(2) $\mathsf{dom}(\Gamma_1) \subseteq \mathsf{dom}(h_1)$

(3) $\Delta; \Gamma_1 \vdash h_1$

(4) $\Delta; \Gamma_1 * r \vdash \mathsf{new}\ C()\mathsf{new}\ \bar{C}'() : T\bar{T}' \dashv \Delta; \Gamma_1 * r$

By (4) and the premises of T-MULTI,

(5) $\Delta; \Gamma_1 * r \vdash \mathsf{new}\ C() : T \dashv \Delta; \Gamma_1 * r$

(6) $\Delta; \Gamma_1 * r \vdash \mathsf{new}\ \bar{C}'() : \bar{T}' \dashv \Delta; \Gamma_1 * r$

By Lemma 4.4 (Inversion of the Term Typing Relation) on (5), $\Delta \vdash C.\mathsf{init} <: T$. Use inversion also on (6) to obtain $\Delta \vdash \bar{C}'.\mathsf{init} <: \bar{T}'$.

We claim that the final state typing $\Delta; \Gamma_1' \vdash (h_2 * r, \bar{o}) : \bar{T} \dashv \Delta; \Gamma_2 * r$ has an initial context $\Gamma_1' = (\Gamma_1, \bar{o} : \bar{C}.\mathsf{init})$ and that the final context Γ_2 depends on the unrestricted/linear nature of each object o_k in $\bar{o} = o_1, \ldots, o_n$ for $1 \leq k \leq n$ that may be "consumed" in the premises of the judgement, or be carried unchanged from Γ_1' to Γ_2.

Again, proving the claim proceeds through a number of steps. Combine (1), (2), (3) and (5) and apply rule T-STATE to obtain $\Delta; \Gamma_1 \vdash (h * r, \mathsf{new}\ C()) : T \dashv \Delta; \Gamma_1 * r$. Then, do an induction using the first premise of R-MULTINEW to get $\Gamma_3 = (\Gamma_1, o : C.\mathsf{init})$ and Γ_3' such that $\Delta; \Gamma_3 \vdash (h_3 * r, o) : T \dashv \Delta; \Gamma_3' * r$ with $\Gamma_1 \subseteq \Gamma_3'$. From the premises of rule T-STATE,

(7) h_3 complete

(8) $\mathsf{dom}(\Gamma_3) \subseteq \mathsf{dom}(h_3)$

(9) $\Delta; \Gamma_3 \vdash h_3$

(10) $\Delta; \Gamma_3 * r \vdash o : T \dashv \Delta; \Gamma_3' * r$

By (9) and Proposition 4.12 (Typing Context Properties), $\Delta \vdash \Gamma_3$. From this, use Lemma 4.14 (Closed Class Families) with (6) to obtain $\Delta; \Gamma_3 * r \vdash \mathsf{new}\ \bar{C}'() : \bar{T}' \dashv \Delta; \Gamma_3 * r$. Do a second induction using the second premise of R-MULTINEW to get $\Gamma_1' = (\Gamma_3, \bar{o}' : \bar{C}'.\mathsf{init})$ and Γ_2' such that $\Delta; \Gamma_1' \vdash (h_2 * r, \bar{o}') : \bar{T}' \dashv \Delta; \Gamma_2' * r$ with $\Gamma_3' \subseteq \Gamma_2'$. Towards the final context Γ_2, we need an additional step. From the premises of T-MULTISTATE,

(11) h_2 complete

(12) $\mathsf{dom}(\Gamma_1') \subseteq \mathsf{dom}(h_2)$

(13) $\Delta; \Gamma_1' \vdash h_2$

(14) $\Delta; \Gamma_1' * r \vdash \bar{o}' : \bar{T}' \dashv \Delta; \Gamma_2' * r$

Since $\Gamma_1' = (\Gamma_3, \bar{o}' : \bar{C}'.\mathsf{init})$, rearrange (10) and (14) in the premises of rule T-MULTI using Lemma 4.6 (Object Swapping) to obtain $\Delta; \Gamma_1' * r \vdash \bar{o} : \bar{T} \dashv \Delta; \Gamma_2 * r$. Conclude from this result with (11), (12) and (13) by applying rule T-MULTISTATE. □

Theorem 4.22 (Subject Reduction)**.** *Suppose that P is a well-formed program ($\vdash P$). In this context, let $\Gamma_0 \subseteq \Gamma_1$ and $h_0 \subseteq h_1$, and $S_1 = (h_1 * r_1, t)$. If $\Delta_1; \Gamma_1 \vdash S_1 : T \dashv \Delta_2; \Gamma_2 * r_2$ and $S_1 \longrightarrow S_2$, then $\Delta_1'; \Gamma_1' \vdash S_2 : T \dashv \Delta_2; \Gamma_2' * r_2$ for some Δ_1', Γ_1' and Γ_2' such that $\Delta_1 \subseteq \Delta_1'$ and $\Gamma_2 \subseteq \Gamma_2'$.*

Proof. By rule induction on the premise $S_1 \longrightarrow S_2$. For this proof, we use the hypothesis $\Delta_1; \Gamma_1 \vdash S_1 : T \dashv \Delta_2; \Gamma_2 * r_1$, which is necessarily the conclusion of rule T-STATE, whose premises are:

(a) h_1 complete

(b) $\mathsf{dom}(\Gamma_1) \subseteq \mathsf{dom}(h_1)$

(c) $\Delta_1; \Gamma_1 \vdash h_1$

(d) $\Delta_1; \Gamma_1 * r_1 \vdash t : T \dashv \Delta_2; \Gamma_2 * r_2$

We start with the inductive case. Then, to prove the base cases, we will repeatedly apply the appropriate clause of Lemma 4.4 (Inversion of the Term Typing Relation) on (d).

Case R-CONTEXT**.** Then,

$$
\begin{aligned}
&t = \mathcal{E}[t_1] \\
&(h_1 * r_1, t_1) \longrightarrow (h_2 * r_1', t_2) \\
&S_2 = (h_2 * r_1', \mathcal{E}[t_2])
\end{aligned}
$$

Let D_1 be the typing derivation concluding (d), that is, $\Delta_1; \Gamma_1 * r_1 \vdash \mathcal{E}[t_1] : T \dashv \Delta_2; \Gamma_2 * r_2$. From this and Lemma 4.19 (Subterm Typing), D_1 has a subderivation D_2 concluding $\Delta_1; \Gamma_1 * r_1 \vdash t_1 : U \dashv \Delta_3; \Gamma_3 * r_3$ for some U, Δ_3, Γ_3 and r_3 such that the position of D_2 in D_1 corresponds to the position of the hole in $\mathcal{E}$. Use this judgement with (a), (b), (c) and rule T-STATE to get $\Delta_1; \Gamma_1 \vdash (h_1 * r_1, t_1) : U \dashv \Delta_3; \Gamma_3 * r_3$. Now, use the induction hypothesis with the premise of R-CONTEXT and the assumption ($\vdash P$) to obtain Δ_1', Γ_1' and Γ_3' such that $\Delta_1'; \Gamma_1' \vdash (h_2 * r_1', t_2) : U \dashv \Delta_3; \Gamma_3' * r_3$ with $\Delta_1 \subseteq \Delta_1'$ and $\Gamma_3 \subseteq \Gamma_3'$. From this, by reading the premises of rule T-STATE:

(1) h_2 complete

(2) $\mathsf{dom}(\Gamma_1') \subseteq \mathsf{dom}(h_2)$

(3) $\Delta_1'; \Gamma_1' \vdash h_2$

(4) $\Delta_1'; \Gamma_1' * r_1' \vdash t_2 : U \dashv \Delta_3; \Gamma_3' * r_3$

Use the judgements concluding D_1 and D_2 together with (4) and Lemma 4.20 (Subterm Replacement) to obtain $\Delta_1'; \Gamma_1' * r_1' \vdash \mathcal{E}[t_2] : T \dashv \Delta_2; \Gamma_2' * r_2$ with $\Gamma_2 \subseteq \Gamma_2'$. Conclude from this with (1), (2) and (3) by applying rule T-STATE.

Case R-NEW**.** Let $\Delta_1' = \Delta_1$. Immediate by applying Lemma 4.21 (Subject Reduction for Object Initialisation).

Case R-ASSIGN**.** Then,

$$t = (f := o')$$
$$h_1(r_1).f = o$$
$$S_2 = (h_1\{f \hookleftarrow o'\} * r_1, o)$$

Without loss of generality, let $h_1 = h, (r_1 = C\{f_1 = o_1, \ldots, f_n = o_n\})$, with $n \geq 1$, and let $f = f_n$, hence $h_1(r_1).f_n = o$. By (c) and the premises of rule T-HEAP,

(1) $\Delta_1; \Gamma \vdash h$

(2) $\Delta_1; \Gamma * r \vdash o_1 \ldots o_{n-1} : F(f_1) \ldots F(f_{n-1}) \dashv \Delta_1; \Gamma' * r_1$

(3) $\Delta_1; \Gamma' * r \vdash o : F(f) \dashv \Delta_1; \Gamma_1 * r_1$

(4) $\Gamma_1 = \Gamma'', r_1 : C[F]$

By inversion of (d),

(5) $\Delta_1; \Gamma_1 * r_1 \vdash o' : U \dashv \Delta_1; \Gamma_3 * r_1$

(6) $\Delta_1; \Gamma_3 \vdash r_1 : C[F] \dashv \Delta_1$

(7) $\Delta_1; \Gamma_2 \vdash r_1 : C\bar{i} \dashv \Delta_1$

(8) $\Gamma_2 = \Gamma_3\{r_1.f \hookleftarrow U\}$

(9) $\Delta_1 \vdash F(f) <: T$

(10) $\Delta_2 = \Delta_1$

Consider judgements (2) and (3) in the premises of T-HEAP together with judgements (5) and (6) from inverting (d). Notice that (6) implies that either $r_1 \notin o_1, \ldots, o_n, o'$, or r_1 is unrestricted. Indeed, if r_1 is linear the effect of typing r_1 in (2), (3) or (5) can only be to remove it from the context, which contradicts (6) and the assumption on well-formed programs ($\vdash P$). Moreover, from inversion of (3), there are two cases: if $\mathsf{un}(V)$ for some V such that $\Delta_1 \vdash V <: F(f)$, then $\Gamma_1 = \Gamma'$, otherwise $\Gamma_1 = \Gamma' \setminus o$. A similar

remark applies to (5) and how the final context Γ_3 relates to the initial Γ_1, i.e. depending on the unrestricted/linear nature of some type W such that $\Delta_1 \vdash W <: U$, either $\Gamma_3 = \Gamma_1$ or $\Gamma_3 = \Gamma_1 \setminus o'$.

Let $\Delta_1' = \Delta_1, \Gamma_2' = \Gamma_2$ and $r_2 = r_1$. We claim that contexts Γ_1' and Γ_2 in the final state typing $\Delta_1; \Gamma_1' * r_1 \vdash (h_1\{f \hookleftarrow o'\} * r_1, o) : T \dashv \Delta_2; \Gamma_2 * r_2$ differ in the same way as Γ' and Γ_1 in (3). Indeed, in the case where the inversion of (3) gives $\mathsf{un}(V)$, then $\Gamma_1' = \Gamma_2$, otherwise $\mathsf{lin}(V)$ and $\Gamma_1' = (\Gamma_2, o : V)$, with the final context being Γ_2.

It remains to prove that the final state can be typed in the initial context Γ_1'. By (6), (7) and the premises of rule T-HIDE, $\Delta_1; \Gamma_2 \vdash r_1 : C[F\{f \hookleftarrow U\}] \dashv \Delta_1$. By Lemma 4.13 (Agreement of Judgements), $\Delta_1 \vdash C[F\{f \hookleftarrow U\}] : \star$. Since $r_1 \notin \mathsf{dom}(h)$ and $o \in \mathsf{dom}(\Gamma)$ by (2) and (3), there is a heap typing that can be built with another type for f that concludes with

(11) $\Delta_1; \Gamma\{r_1.f \hookleftarrow U\} \vdash h$

In possession of the modified context, use (2), (3) and (5) in Lemma 4.6 (Object Swapping) to exchange o and o' and obtain a typing judgement that is equivalent to the following:

(12) $\Delta_1; \Gamma\{r_1.f \hookleftarrow U\} * r_1 \vdash o_1 \dots o_{n-1} o' : F(f_1) \dots F(f_{n-1}) U \dashv \Delta_1; \Gamma_1' * r_1$

(13) $\Delta_1; \Gamma_1' * r_1 \vdash o : F(f) \dashv \Delta_1; \Gamma_2 * r_1$

With (11), (12) and rule T-HEAP,

(14) $\Delta_1; \Gamma_1' \vdash h, (r_1 = C\{f_1 = o_1, \dots, f_{n-1} = o_{n-1}, f_n = o'\})$

On the other hand, notice that $\mathsf{dom}(h_1\{f \hookleftarrow o'\}) = \mathsf{dom}(h_1)$, which using (a) implies that $h_1\{f \hookleftarrow o'\}$ complete. Also, by the form of Γ_1' and (b), $\mathsf{dom}(\Gamma_1') \subseteq \mathsf{dom}(h_1\{f \hookleftarrow o'\})$. From this with (9), (13), (14) and T-SUB, conclude by applying rule T-STATE.

Case R-SEQ**.** Then,

$$t = o; t_2$$
$$S_2 = (h_1 * r_1, t_2)$$

By inversion of (d),

(1) $\Delta_1; \Gamma_1 * r_1 \vdash o : U_1 \dashv \Delta_1; \Gamma_3 * r_1$

(2) $\Delta_1; \Gamma_3 * r_1 \vdash t_2 : U_2 \dashv \Delta_2; \Gamma_2 * r_1$

(3) $\mathsf{un}(U_1)$

(4) $\Delta_2 \vdash U_2 <: T$

Let $\Delta_1' = \Delta_1, \Gamma_1' = \Gamma_1, \Gamma_2' = \Gamma_2$, and $r_2 = r_1$. Since the heap does not change, we show that $\Delta_1; \Gamma_1 \vdash (h_1 * r_1, t_2) : T \dashv \Delta_2; \Gamma_2 * r_2$. Noticing (3), apply inversion on (1) to obtain $\Gamma_3 = \Gamma_1$. Conclude directly with T-STATE using (a), (b), (c) and the result of applying T-SUB to (2) and (4).

Case R-SELFCALL**.** Then,

$$
\begin{aligned}
&t = m(o) \\
&h_1(r_1).\mathsf{class} = C \\
&\mathsf{mbody}(m, C) = \lambda x.t' \\
&S_2 = (h_1 * r_1, t'[o/x])
\end{aligned}
$$

To help us build the final state typing, notice a number of facts. The hypothesis on well-formed programs ($\vdash P$) implies that method m appears in the declaration of either C or some superclass of C. Let D be the class where the method is defined. Hence, $\vdash_D m(x) = t'$, which is necessarily the conclusion of T-METHOD whose premises are:

(i) $\mathsf{class}\ D : \Delta\ \mathsf{extends}\ \_\{\ldots, m : \Pi\Delta'.(T_1 \rightsquigarrow T_2 \times U \to W), \ldots\}\ \mathsf{is}\ \{\_\}$

(ii) $\Delta, \Delta'; x : U, \mathsf{this} : \mathsf{fields}(T_1) * \mathsf{this} \vdash t' : W \dashv \Delta''; \Gamma, \mathsf{this} : D[F'] * \mathsf{this}$

(iii) $\Delta'' \vdash C[F'] <: \mathsf{fields}(T_2)$

(iv) $x : U \in \Gamma \Rightarrow \mathsf{un}(U)$

(v) $m \neq \mathsf{init}$

By inversion of (d),

(1) $\Delta_1; \Gamma_1 * r_1 \vdash o : U'[\theta] \dashv \Delta_1; \Gamma_3 * r_1$

(2) $\Delta_1; \Gamma_3 \vdash r_1 : C\bar{i} \dashv \Delta_1$

(3) $\Delta_1; \Gamma_3\{r_1 \hookleftarrow T'[\theta]\} \vdash r_1 : C\bar{j} \dashv \Delta_2$

(4) $\mathsf{mtype}(m, C\bar{i}) = \Pi\Delta_3.(C\bar{i} \rightsquigarrow T' \times U' \to W')$

(5) $\Delta_1 \vdash \Delta_3 : \theta$

(6) $\Gamma_2 = \Gamma_3\{r_1 \hookleftarrow \mathsf{fields}(C\bar{j})\}$

(7) $\Delta_2 \vdash W'[\theta] <: T$

Let $\bar{i} = \bar{i}'\bar{i}''$ and $V = \Pi\Delta'.(T_1 \rightsquigarrow T_2 \times U \to W)$. We claim that (i) and (4) imply that $\mathsf{mtype}(m, C\bar{i}) = V[\theta'][C\bar{i}'/D]$ for some θ' such that $\Delta_1 \vdash \Delta : \theta'$ with $\Delta = \bar{a} : \bar{i}$ and $\theta' = \bar{i}/\bar{a}$.

Assuming that the claim holds, then we have the following as regards the relation implied above:

- $\Delta_3 = \Delta'[\theta']$
- $C\bar{i} = (T_1[\theta'])[C\bar{i}'/D]$
- $T' = (T_2[\theta'])[C\bar{i}'/D]$ and $T'[\theta] = (T_2[\theta'\theta])[C\bar{i}'/D]$
- $U' = U[\theta']$ and $U'[\theta] = U[\theta'\theta]$
- $W' = W[\theta']$ and $W'[\theta] = W[\theta'\theta]$

On the other hand, from inversion of (1), there are two cases to consider: if $\mathsf{un}(U_0)$ for some U_0 such that $\Delta_1 \vdash U_0 <: U'[\theta]$, then $\Gamma_3 = \Gamma_1$, otherwise $\Gamma_3 = \Gamma_1 \setminus o$. In both cases, notice that the difference between Γ_1 and Γ_3 depends on the linear/unrestricted nature of type U' in the same way as Γ depends on type U (ii). Moreover, from this and (2), notice that the type of r_1 goes unchanged from Γ_1 to Γ_3.

Let $\Delta_1' = \Delta_1, \Gamma_1' = \Gamma_1, \Gamma_2' = \Gamma_2$ and $r_2 = r_1$. To make the claim concrete, we show that $\Delta_1; \Gamma_1 \vdash (h_1 * r_1, t'[o/x]) : T \dashv \Gamma_2 * r_1$, which follows by a number of steps.

First, by (2) and Lemma 4.13 (Agreement of Judgements), $\Delta_1 \vdash C\bar{i} : \star$. From this, (4) and Lemma 4.15 (Class/mtype Relation), obtain $\Delta_1 \vdash V[\theta'][C\bar{i}'/D] : \star$ which implies $\Delta_1 \vdash \Delta : \theta'$. Use this together with (5) and Lemma 4.11 (Substitution Composition) to get $\Delta_1 \vdash \Delta\Delta_3 : \theta'\theta$. Then, by Lemma 4.16 (Instantiation) using the second premise of R-SELFCALL, the judgement $\Delta_1 \vdash C\bar{i} : \star$, (4) and (5), obtain

(8) $\Delta_1; x : U'[\theta], \mathsf{this} : \mathsf{fields}(C\bar{i}) * \mathsf{this} \vdash t' : W'[\theta] \dashv \Delta_2; \Gamma[\theta'\theta], \mathsf{this} : \mathsf{fields}(C\bar{j}) * \mathsf{this}$

with $\Delta_2 \vdash \mathsf{fields}(C\bar{j}) <: T'[\theta]$. Now, apply Lemma 4.8 (Substitution for Objects) using (1) and (8) to get

(9) $\Delta_1; \Gamma_1 * r_1 \vdash t'[o/x] : W'[\theta] \dashv \Delta_2; \Gamma_3\{r_1 \leftarrow\!\!\shortmid \mathsf{fields}(C\bar{j})\} * r_1$

Conclude by applying rule T-STATE to (a), (b), (c) and the result of T-SUB with (7) and (9).

Case R-CALL**.** Then,

$$
\begin{aligned}
& t = f.m(o) \\
& h(r_1).f.\mathsf{class} = C \\
& \mathsf{mbody}(m, C) = \lambda x.t' \\
& S_2 = (h_1 * r_1.f, \mathsf{return}\ t'[o/x])
\end{aligned}
$$

As in the previous case, notice a number of facts that help us build the final state typing. Let B be the first superclass of C where method m is defined. The assumption on well-formed programs ($\vdash P$) implies that method m appears in the declaration of B. Hence, $\vdash_B m(x) = t'$, which is necessarily the conclusion of T-METHOD whose premises are:

(i) $\mathsf{class}\ B : \Delta\ \mathsf{extends}\ \_\{\ldots, m : \Pi\Delta'.(T_1 \rightsquigarrow T_2 \times U \to W), \ldots\}\ \mathsf{is}\ \{\_\}$

(ii) $\Delta, \Delta'; x : U, \mathsf{this} : \mathsf{fields}(T_1) * \mathsf{this} \vdash t' : W \dashv \Delta''; \Gamma, \mathsf{this} : B[F'] * \mathsf{this}$

(iii) $\Delta'' \vdash B[F'] <: \mathsf{fields}(T_2)$

(iv) $x : U \in \Gamma \Rightarrow \mathsf{un}(U)$

(v) $m \neq \mathsf{init}$

Let D be the class of the current object. Without loss of generality, let $h_1 = (h, r_1 = D\{f_1 = o_1, \ldots, f_n = o_n\})$, with $n \geq 1$, and let $f = f_n$, hence $h_1(r_1).f = o_n$. Then, by (c) and the premises of rule T-HEAP,

(1) $\Delta_1; \Gamma \vdash h$

(2) $\Delta_1; \Gamma * r_1 \vdash o_1 \ldots o_{n-1} : F(f_1) \ldots F(f_{n-1}) \dashv \Delta_1; \Gamma' * r_1$

(3) $\Delta_1; \Gamma' * r_1 \vdash o_n : F(f) \dashv \Delta_1; \Gamma_1 * r_1$

(4) $\Gamma_1 = \Gamma'', r_1 : D[F]$

By inversion of (d),

(5) $\Delta_1; \Gamma_1 * r_1 \vdash o : U'[\theta] \dashv \Delta_1; \Gamma_3 * r_1$

(6) $\Delta_1; \Gamma_3 \vdash r_1.f : T' \dashv \Delta_2$

(7) $\mathsf{mtype}(m, T') = \Pi\Delta_3.(T' \rightsquigarrow T'' \times U' \to W')$

(8) $\Delta_2 \vdash \Delta_3 : \theta$

(9) $\Delta_2; \Gamma_2 \vdash r_1 : C\bar{j} \dashv \Delta_2$

(10) $\Gamma_2 = \Gamma_3\{r_1.f \leftarrowtail T''[\theta]\}$

(11) $\Delta_2 \vdash W'[\theta] <: T$

We have that T' is $C\bar{i}$ by (7). Let $\bar{i} = \bar{i}'\bar{i}''$ and $V = \Pi\Delta'.(T_1 \rightsquigarrow T_2 \times U \to W)$. We claim that (i) and (7) imply that $\mathsf{mtype}(m, T') = V[\theta'][C\bar{i}'/B]$ for some θ' such that $\Delta_2 \vdash \Delta : \theta'$ with $\Delta = \bar{a} : \bar{i}$ and $\theta' = \bar{i}/\bar{a}$.

Assuming that the claim holds, then we have the following as regards the relation implied above:

- $\Delta_3 = \Delta'[\theta']$
- $T' = T_1[\theta'][C\bar{i}'/B]$ and $T' = T_1[\theta'][C\bar{i}'/B]$
- $T'' = T_2[\theta'][C\bar{i}'/B]$ and $T'[\theta] = T_2[\theta'\theta][C\bar{i}'/B]$
- $U' = U[\theta']$ and $U'[\theta] = U[\theta'\theta]$
- $W' = W[\theta']$ and $W'[\theta] = W[\theta'\theta]$

On the other hand, from inversion of (5), there are two cases to consider: if $\mathsf{un}(U_0)$ for some U_0 such that $\Delta_1 \vdash U_0 <: U'[\theta]$ then $\Gamma_3 = \Gamma_1$, otherwise $\Gamma_3 = \Gamma_1 \setminus o$. In both cases, notice that the difference between Γ_1 and Γ_3 depends on the linear/unrestricted nature of type U' in the same way as Γ depends on type U (ii). From this and (6), conclude that the type of r_1 (and as a result of $r_1.f$) goes unchanged from Γ_1 to Γ_3.

Let $\Delta_1' = \Delta_2$ and $\Gamma_2' = \Gamma_2$. To make the claim concrete, we show that $\Delta_2; \Gamma_1' \vdash (h_1 * r_1.f, \mathsf{return}\ t'[o/x]) : T \dashv \Delta_2; \Gamma_2 * r_1$ for some Γ_1', which follows by a number of steps.

By (6) and Lemma 4.13 (Agreement of Judgements), $\Delta_2 \vdash T' : \star$. From this, (7) and Lemma 4.15 (Class/mtype Relation) noticing (i), $\Delta_2 \vdash V[\theta'][C\bar{i}'/B] : \star$ which implies $\Delta_2 \vdash \Delta : \theta'$. From this together with (8) and Lemma 4.11 (Substitution Composition), obtain $\Delta_2 \vdash \Delta\Delta_3 : \theta'\theta$. Now, use the second premise of rule R-CALL, (7), (8) with Lemma 4.16 (Instantiation) in order to get

(12) $\Delta_2; x : U'[\theta], \mathsf{this} : \mathsf{fields}(T') * \mathsf{this} \vdash t' : W'[\theta] \dashv \Delta_2; \Gamma', \mathsf{this} : C[F''] * \mathsf{this}$

(13) $\Delta_2 \vdash C[F''] <: \mathsf{fields}(T''[\theta])$

Then, by Lemma 4.17 (Opening) with (c), (6) and $h_1(r_1).f = o_n$,

(14) $\Delta_2; \Gamma_1\{r_1.f \leftarrowtail \mathsf{fields}(T')\} \vdash h_1$

Using the fact that the type of $r_1.f$ goes unchanged from Γ_1 to Γ_3, build a derivation that concludes as follows:

(15) $\Delta_2; \Gamma_1\{r_1.f \hookleftarrow \mathsf{fields}(T')\} * r_1.f \vdash o : U'[\theta] \dashv \Delta_2; \Gamma_3\{r_1.f \hookleftarrow \mathsf{fields}(T')\} * r_1.f$

By (6), (12), (15) and Lemma 4.8 (Substitution for Objects),

(16) $\Delta_2; \Gamma_1\{r_1.f \hookleftarrow \mathsf{fields}(T')\} * r_1.f \vdash t'[o/x] : W'[\theta] \dashv \Delta_2; \Gamma_3\{r_1.f \hookleftarrow C[F'']\} * r_1.f$

Use (13) and (16) with rule T-RETURN in order to obtain

(17) $\Delta_2; \Gamma_1\{r_1.f \hookleftarrow \mathsf{fields}(T')\} * r_1.f \vdash \mathsf{return}\ t'[o/x] : W'[\theta] \dashv \Delta_2; \Gamma_2 * r_1$

Let $\Gamma_1' = \Gamma_1\{r_1.f \hookleftarrow \mathsf{fields}(T')\}$. From $\mathsf{dom}(\Gamma_1) \subseteq \mathsf{dom}(h_1)$ (b) and (14), it follows that $\mathsf{dom}(\Gamma_1') \subseteq \mathsf{dom}(h_1)$. Combine this with (a), (11), (14), (17) and T-SUB, and conclude by applying rule T-STATE.

Case R-RETURN**.** Then,

$$t = \mathsf{return}\ o$$
$$r_1 = r_2.f$$
$$S_2 = (h_1 * r_2, o)$$

By inversion of (d),

(1) $\Delta_1; \Gamma_1 * r_2.f \vdash o : V \dashv \Delta_1; \Gamma_3 * r_2.f$

(2) $\Delta_1; \Gamma_3 \vdash r_2.f : C[F] \dashv \Delta_1$

(3) $\Delta_1 \vdash C[F] <: \mathsf{fields}(U)$

(4) $\Gamma_2 = \Gamma_3\{r_2.f \hookleftarrow U\}$

(5) $\Delta_1 \vdash V <: T$

(6) $\Delta_2 = \Delta_1$

Let $\Delta_1' = \Delta_1$ and $\Gamma_2' = \Gamma_2$. Since the heap does not change, we build a derivation that leads to $\Delta_1; \Gamma_1' \vdash (h * r_2, o) : T \dashv \Delta_1; \Gamma_2 * r_2$ for some Γ_1'.

By inversion of (1), $\Delta_1 \vdash \Gamma_1$, and there are two cases to consider: if $\mathsf{un}(V')$ for some V' such that $\Delta_1 \vdash V' <: V$, then $\Gamma_3 = \Gamma_1$, otherwise $\Gamma_3 = \Gamma_1 \setminus o$ for $\mathsf{lin}(V')$. Moreover, from (2) notice that the type of $r_2.f$ goes unchanged from Γ_1 to Γ_3. Using $\Delta_1; \Gamma_1 \vdash h_1$ (c), (2), (3) and Lemma 4.18 (Closing), obtain $\Delta_1; \Gamma_1\{r_2.f \hookleftarrow U\} \vdash h_1$,

and hence $\Gamma_1' = \Gamma_1\{r_2.f \leftarrowtail U\}$. Then apply T-UNVAR or T-LINVAR as appropriate, followed by T-SUB with (5), to get $\Delta_1; \Gamma_1' * r_2 \vdash o : T \dashv \Delta_1; \Gamma_2 * r_2$. Conclude from this result with h_1 complete (a), $\mathsf{dom}(\Gamma_1') \subseteq \mathsf{dom}(h_1)$ (b), the heap typing from the closing lemma and rule T-STATE.

Case R-CASE$_k$**.** Then,

$$t = \mathsf{case}\ f\ \mathsf{of}\ (C_k \Rightarrow t_k)_{k\in 1,2}$$
$$h_1(r_1.f).\mathsf{class} = C_k$$
$$S_2 = (h_1 * r_1, t_k)_{k\in\{1,2\}}$$

By inversion of (d),

(1) $\Delta_1; \Gamma_1\{r_1.f \leftarrowtail (U_1 + U_2)\} \vdash r_1.f : (U_1 + U_2) \dashv \Delta_1$

(2) $\Delta_1; \Gamma_1 * r_1 \vdash t_k : U \dashv \Delta_2; \Gamma_2 * r_1$

(3) $\mathsf{classof}(U_k) = C_k$

(4) $C_1 \neq C_2$

(5) $\Delta_1 \vdash U <: T$

(6) $r_2 = r_1$

The reason why we have Γ_1 as the initial object context in (2) is because we modify that context in (1) by updating $r_1.f$ with the union type and replacing type U_k to which $r_1.f$ is bound.

Let $\Delta_1' = \Delta_1$ and $\Gamma_1' = \Gamma_1$. Since the heap does not change, conclude $\Delta_1; \Gamma_1 \vdash (h_1 * r_1, t_k) : T \dashv \Delta_2; \Gamma_2 * r_1$ directly by applying rules T-SUB and T-STATE using (a), (b), (c), (2) and (5).

Case R-IFTRUE**.** Then,

$$t = \mathsf{if\ true\ then}\ t_1\ \mathsf{else}\ t_2$$
$$S_2 = (h_1 * r_1, t_1)$$

By inversion of (d),

(1) $\Delta_1; \Gamma_1 * r_1 \vdash \mathsf{true} : \mathsf{Boolean}\ p \dashv \Delta_1; \Gamma_1 * r_1$

(2) $\Delta_1, p; \Gamma_1 * r_1 \vdash t_1 : U \dashv \Delta_2; \Gamma_2 * r_2$

(3) $\Delta_1, \neg p; \Gamma_1 * r_1 \vdash t_2 : U \dashv \Delta_2; \Gamma_2 * r_2$

(4) $\Delta_2 \vdash U <: T$

Let $\Delta_1' = (\Delta_1, p)$, and hence $\Delta_1 \subseteq \Delta_1'$, and let $\Gamma_1' = \Gamma_1$ and $\Gamma_2' = \Gamma_2$. Since the heap does not change, we build a derivation that leads to $\Delta_1'; \Gamma_1 \vdash (h * r_1, t_1) : T \dashv \Delta_2; \Gamma_2 * r_1$.

First, notice that the class declaration of Boolean implies $\mathsf{un}(\mathsf{Boolean}\ p)$. From this and $\Gamma_0 \subseteq \Gamma_1$, we have $(\mathsf{true} : \mathsf{Boolean}\ \mathsf{true}) \in \Gamma_1$. Also, from the assumption on well-formed programs ($\vdash P$), we have $\vdash \Delta_1'$. Use this and $\Delta_1; \Gamma_1 \vdash h_1$ (c) with Lemma 4.7 (Weakening for Index Contexts) to obtain $\Delta_1'; \Gamma_1 \vdash h_1$. Combine (2), (4) and rule T-SUB to derive $\Delta_1'; \Gamma_1 * r_1 \vdash t_1 : T \dashv \Delta_2; \Gamma_2 * r_2$. Conclude from this, together with (a), (b) and the heap typing from weakening, by applying rule T-STATE.

Case R-IFFALSE**.** Then,

$$t = \mathsf{if}\ \mathsf{false}\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2$$
$$S_2 = (h_1 * r_1, t_2)$$

The proof is omitted, since it is analogous to the one of case R-IFTRUE, except that it uses $\Delta_1' = (\Delta_1, \neg p)$ as the initial index context, and (3) in the premises of rule T-SUB.

Case R-WHILE**.** Then,

$$t = \mathsf{while}\ \mathsf{true}\ \mathsf{do}\ t_1$$
$$t_2 = \mathsf{if}\ \mathsf{true}\ \mathsf{then}\ (t_1; \mathsf{while}\ \mathsf{true}\ \mathsf{do}\ t_1)\ \mathsf{else}\ \mathsf{top}$$
$$S_2 = (h_1 * r_1, t_2)$$

By inversion of (d),

(1) $\Delta_1; \Gamma_1 * r_1 \vdash \mathsf{true} : \mathsf{Boolean}\ p \dashv \Delta_1; \Gamma_1 * r_1$

(2) $\Delta_1, p; \Gamma_1 * r_1 \vdash t_1 : T \dashv \Delta_1; \Gamma_1 * r_1$

(3) $T = \mathsf{Top}$

(4) $\Gamma_2 = \Gamma_1$

(5) $\Delta_2 = (\Delta_1, \neg p)$

(6) $r_2 = r_1$

Let $\Delta_1' = \Delta_1, \Gamma_1' = \Gamma_1$ and $\Gamma_2' = \Gamma_1'$. Since the heap does not change, we build a derivation that leads to $\Delta_1; \Gamma_1 \vdash (h_1 * r_1, t_2) : T \dashv \Delta_2; \Gamma_1 * r_1$.

Notice that the class declaration of Top implies $\mathsf{un}(\mathsf{Top})$. From this and $\Gamma_0 \subseteq \Gamma_1$, we have $(\mathsf{top} : \mathsf{Top}) \in \Gamma_1$. From the assumption on well-formed programs ($\vdash P$), we

have $\vdash \Delta_2$. By inversion of (1), $\Delta_1 \vdash \Gamma_1$ and then by Lemma 4.7 (Weakening for Index Contexts) obtain $\Delta_2 \vdash \Gamma_1$. Use this with rule T-UNVAR to get $\Delta_2; \Gamma_1 * r_1 \vdash \mathsf{top} : \mathsf{Top} \dashv \Delta_2; \Gamma_1 * r_1$. Combine with (1), (2), (d) and rules T-SEQ and T-IF, and conclude from this result with (a), (b) and (c) by applying rule T-STATE. □

4.6 Progress

Towards type soundness, we show progress by proving that reduction never gets stuck on well-formed DOL programs. Although simpler than subject reduction, the progress result states crucial properties about locations in the heap.

Theorem 4.23 (Progress)**.** *Suppose that P is a well-formed program ($\vdash P$). In this context, let $\Gamma_0 \subseteq \Gamma_1$ and $h_0 \subseteq h_1$.*

1. *If $\Delta_1; \Gamma_1 \vdash (h_1 * r_1, t_1) : T \dashv \Delta_2; \Gamma_2 * r_2$, then t_1 is an object reference or $(h_1 * r_1, t_1) \longrightarrow (h_2 * r_2, t_2)$.*
2. *If $\Delta_1; \Gamma_1 \vdash (h_1 * r, \bar{t}) : \bar{T} \dashv \Delta_2; \Gamma_2 * r$, then $(h_1 * r, \bar{t}) \longrightarrow (h_2 * r, \bar{t'})$.*

Proof. By mutual induction on t and $\bar{t}$. For the cases that fall in part 1, notice that the hypothesis $\Delta_1; \Gamma_1 \vdash (h_1 * r_1, t_1) : T \dashv \Delta_2; \Gamma_2 * r_2$ is necessarily the conclusion of rule T-STATE, which implies the following premises:

(a) h_1 complete

(b) $\mathsf{dom}(\Gamma_1) \subseteq \mathsf{dom}(h_1)$

(c) $\Delta_1; \Gamma_1 \vdash h_1$

(d) $\Delta_1; \Gamma_1 * r_1 \vdash t_1 : T \dashv \Delta_2; \Gamma_2 * r_2$

To prove some of the cases, we will refer to the above hypothesis and apply the appropriate clause of Lemma 4.4 (Inversion of the Term Typing Relation) on (d).

For the inductive case, if $t_1 = \mathcal{E}[t_3]$ with $t_3 \neq o$ and $\mathcal{E} \neq [\_]$, use (d) and Lemma 4.19 (Subterm Typing) to obtain $\Delta_1; \Gamma_1 * r_1 \vdash t_3 : U \dashv \Delta_3; \Gamma_3 * r_3$. Combine with (a), (b), (c) and rule T-STATE to get $\Delta_1; \Gamma_1 \vdash (h_1 * r_1, t_3) : U \dashv \Delta_3; \Gamma_3 * r_3$. Then by the induction hypothesis, $(h_1 * r_1, t_3) \longrightarrow (h_2 * r_2, t_2)$. Conclude by applying rule R-CONTEXT.

Case $t_1 = o$**.** t_1 is an object reference.

Case $t_1 = \mathsf{new}\ C()$**.** Then the hypothesis ($\vdash P$) implies that $\vdash_C \mathsf{init}() = \bar{f} := \mathsf{new}\ \bar{C}()$, which is necessarily the conclusion of rule T-INIT. From this, we have

(1) $\mathsf{mbody}(\mathsf{init}, C) = \bar{f} := \mathsf{new}\ \bar{C}()$

By reading the premises of T-INIT, we also have $\mathsf{fields}(C.\mathsf{init}) = C[F]$ and $\epsilon; \epsilon * r_1 \vdash \mathsf{new}\ \bar{C}() : F(\bar{f}) \dashv \epsilon; \epsilon * r_1$. By inversion of (d), $\Delta_1 \vdash \Gamma_1$ and $r_2 = r_1$. Use Lemma 4.14 (Closed Class Families) to obtain

(2) $\Delta_1; \Gamma_1 * r_1 \vdash \mathsf{new}\ \bar{C}() : F(\bar{f}) \dashv \Delta_1; \Gamma_1 * r_1$

Apply rule T-MULTISTATE with (a), (b), (c) and (2) to get $\Delta_1; \Gamma_1 \vdash (h_1 * r_1, \mathsf{new}\ \bar{C}()) : F(\bar{f}) \dashv \Delta_1; \Gamma_1 * r_1$. Then by the induction hypothesis,

(3) $(h_1 * r_1, \mathsf{new}\ \bar{C}()) \longrightarrow (h_3 * r_1, \bar{o})$

Since reduction rules generate fresh objects, let $h_2 = (h_3, o = C\{\bar{f} = \bar{o}\})$, and hence $o \notin \mathsf{dom}(h_3)$. From this together with (1) and (3), conclude by applying rule R-NEW.

Case $t_1 = (f := o)$**.** By inversion of (d),

(1) $\Delta_1; \Gamma_1 * r_1 \vdash o : U \dashv \Delta_3; \Gamma_3 * r_2$

(2) $\Delta_2; \Gamma_3 \vdash r_2 : C[F] \dashv \Delta_2$

(3) $\Delta_2; \Gamma_2 \vdash r_2 : C\bar{i} \dashv \Delta_2$

(4) $\Gamma_2 = \Gamma_3\{r_2.f \hookleftarrow U\}$

Apply inversion again to (1) to obtain $r_2 = r_1$ and, depending on the linear/unrestricted nature of U, either $\Gamma_3 = \Gamma_1$ or $\Gamma_3 = \Gamma_1 \setminus o$, and hence $\Gamma_3 \subseteq \Gamma_1$. By (2) and rule T-REF, $r_2 \in \mathsf{dom}(\Gamma_3)$. It follows that $r_2 \in \mathsf{dom}(\Gamma_1)$. From $\mathsf{dom}(\Gamma_1) \subseteq \mathsf{dom}(h_1)$ (b), we have $r_2 \in \mathsf{dom}(h_1)$ (by transitivity). Notice that (3) and (4) imply that f is a field of the object at r_2. Then from $r_2 \in \mathsf{dom}(h_1)$ and h_1 complete (a), $h_1(r_2).f$ is defined. Conclude by applying rule R-ASSIGN.

Case $t_1 = o; t_2$**.** Apply rule R-SEQ.

Case $t_1 = m(o)$**.** By inversion of (d),

(1) $\Gamma_1; \Delta_1 * r_1 \vdash o : U[\theta] \dashv \Gamma_3; \Delta_3 * r_2$

(2) $\Delta_3; \Gamma_3 \vdash r_2 : C\bar{i} \dashv \Delta_3$

(3) $\Delta_3; \Gamma_2 \vdash r_2 : C\bar{j} \dashv \Delta_2$

(4) $\mathsf{mtype}(m, C\bar{i}) = \Pi\Delta.(C\bar{i} \rightsquigarrow T_2 \times U \to W)$

(5) $\Delta_3 \vdash \Delta : \theta$

(6) $\Gamma_2 = \Gamma_2\{r_2 \hookleftarrow T_2[\theta]\}$

Apply inversion again to (1) to obtain $r_2 = r_1$ and, depending on the linear/unrestricted nature of U, either $\Gamma_3 = \Gamma_1$ or $\Gamma_3 = \Gamma_1 \setminus o$, and hence $\Gamma_3 \subseteq \Gamma_1$. By (2) and rule T-HIDE, $r_2 \in \mathsf{dom}(\Gamma_3)$. It follows that $r_2 \in \mathsf{dom}(\Gamma_1)$. From $\mathsf{dom}(\Gamma_1) \subseteq \mathsf{dom}(h_1)$ (b), $r_2 \in \mathsf{dom}(h_1)$ (by transitivity). On the other hand, notice that (4) and a well-formed program ($\vdash P$) imply that method m appears in the declaration of either C or some superclass of C. Let D be the class where the method is defined such that $\vdash_D m(x) = t$ for a parameter x and a method body t, and hence $\mathsf{mbody}(m, C) = \lambda x.t$. From this and $h_1(r_2).\mathsf{class} = C$, conclude by applying rule R-SELFCALL.

Case $t_1 = f.m(o)$**.** By inversion of (d),

(1) $\Gamma_1; \Delta_1 * r_1 \vdash o : U[\theta] \dashv \Gamma_3; \Delta_2 * r_3$

(2) $\Gamma_3; \Delta_3 \vdash r_3.f : T_1 \dashv \Delta_2$

(3) $\Delta_2; \Gamma_2 \vdash r_3 : C\bar{j} \dashv \Delta_2$

(4) $\mathsf{mtype}(m, T_1) = \Pi\Delta.(T_1 \rightsquigarrow T_2 \times U \to W)$

(5) $\Delta_3 \vdash \Delta : \theta$

(6) $\Gamma_2 = \Gamma_2\{r_3.f \hookleftarrow T_2[\theta]\}$

For the first part, proceed as in case $(t_1 = (f \coloneqq o))$: apply inversion to (1) to obtain $r_3 = r_1$ and $\Gamma_3 \subseteq \Gamma_1$. Notice that (2) is derived by rule T-FIELD followed by as many applications of rule T-UNPACK as needed, which implies that $r_3 \in \mathsf{dom}(\Gamma_1)$. Then, from h_1 complete (a) and $\mathsf{dom}(\Gamma_1) \subseteq \mathsf{dom}(h_1)$ (b), we have that $h(r_3.f)$ is defined. For the second part, proceed as in case $(t_1 = m(o))$: notice that type T_1 must be of the form $C\bar{i}$ by (4), and this together with a well-formed program ($\vdash P$) imply that method m appears in the declaration of either C or some superclass of C. Let D be the class where the method is defined such that such that $\vdash_D m(x) = t$ for parameter x and a method body t, and hence $\mathsf{mbody}(m, C) = \lambda x.t$. Conclude from this and $h_1(r_3.f).\mathsf{class} = C$ by applying rule R-CALL with $r_2 = r_3.f$ and $r_3 = r_1$.

Case $t_1 = \mathsf{return}\ o$**.** By inversion of (d),

(1) $\Delta_1; \Gamma_1 * r_1 \vdash o : T \dashv \Delta_2; \Gamma_3 * r_2.f$

(2) $\Delta_2; \Gamma_3 \vdash r_2.f : C[F] \dashv \Delta_2$

(3) $\Delta_2 \vdash C[F] <: \mathsf{fields}(U)$

(4) $\Gamma_2 = \Gamma_3\{r_2.f \hookleftarrow U\}$

Apply inversion to (1) to obtain $r_2.f = r_1$ and, depending on the linear/unrestricted nature of U, either $\Gamma_3 = \Gamma_1$ or $\Gamma_3 = \Gamma_1 \setminus o$, and hence $\Gamma_3 \subseteq \Gamma_1$. By (2) and the derivation of rule T-FIELD, $r_2 \in \mathsf{dom}(\Gamma_3)$, and hence $r_2 \in \mathsf{dom}(\Gamma_1)$. From h_1 complete (a) and $\mathsf{dom}(\Gamma_1) \subseteq \mathsf{dom}(h_1)$ (b), we have that $r_2.f \in \mathsf{dom}(h_1)$. Conclude by applying rule R-RETURN.

Case $t_1 = \mathsf{case}\ f\ \mathsf{of}\ (C_k \Rightarrow t_k)_{k\in 1,2}$**.** By inversion of (d),

(1) $\Delta_1; \Gamma_1 \vdash r_1.f : (U_1 + U_2) \dashv \Delta_3$

(2) $\mathsf{classof}(U_k) = C_k$

(3) $\Delta_3; \Gamma_1\{r_1.f \leftarrowtail U_k\} * r_1 \vdash t_k : T \dashv \Delta_2; \Gamma_2 * r_1$

Notice that (1) is derived by rule T-FIELD, followed by as many applications as needed of rule T-UNPACK. From this derivation, $r_1 \in \mathsf{dom}(\Gamma_1)$. Then by $\mathsf{dom}(\Gamma_1) \subseteq \mathsf{dom}(h_1)$ (a), $h(r_1)$ is defined. From h_1 complete (a) and the fact that f is a field of the object at r_1, $h(r_1.f)$ is also defined. On the other hand, from (2) we have that $h_1(r_1.f).\mathsf{class} = C_k$. Conclude by applying rule R-CASE$_k$.

Case $t_1 = \mathsf{if}\ o\ \mathsf{then}\ t_3\ \mathsf{else}\ t_4$**.** By inversion of (d), $\Delta_1; \Gamma_1 * r_1 \vdash o : \mathsf{Boolean}\ p \dashv \Delta_3; \Gamma_3 * r_2$. Apply inversion again to obtain $r_2 = r_1$ and, depending on the unrestricted/linear nature of Boolean p, either $\Gamma_3 = \Gamma_1$ or $\Gamma_3 = \Gamma_1 \setminus o$. However, from the Boolean class declaration, we have that $\mathsf{un}(\mathsf{Boolean}\ p)$, and therefore $\Gamma_3 = \Gamma_1$. Also, use $\Gamma_0 \subseteq \Gamma_1$ and the fact that the predefined Boolean does not declare a constructor to conclude that $(\mathsf{false} : \mathsf{Boolean}\ \mathsf{false}, \mathsf{true} : \mathsf{Boolean}\ \mathsf{true}) \in \mathsf{dom}(\Gamma_1)$ are the only two instances of Boolean, and hence o must be either false or true. Similarly, from $(\mathsf{false}, \mathsf{true}) \in \mathsf{dom}(h_0)$ and $h_0 \subseteq h_1$, we also have that $h_1(o)$ is defined. Conclude by using $\Delta_1 \models p$ to decide whether to apply rule R-IFTRUE or R-IFFALSE.

Case $t_1 = \mathsf{while}\ o\ \mathsf{do}\ t_2$**.** Apply rule R-WHILE.

Case $\bar{t} = \epsilon$**.** Apply rule R-EMPTY.

Case $\bar{t} = \mathsf{new}\ C()\mathsf{new}\ \bar{C}()\ \mathsf{with}\ \mathsf{new}\ \bar{C}() \neq \epsilon$**.** Let $\bar{T} = T'\bar{T}'$. Notice that the hypothesis is necessarily the conclusion of rule T-MULTISTATE, which implies the following premises:

(1) h_1 complete

(2) $\mathsf{dom}(\Gamma_1) \subseteq \mathsf{dom}(h_1)$

(3) $\Delta_1; \Gamma_1 \vdash h_1$

(4) $\Delta_1; \Gamma_1 * r \vdash \mathsf{new}\ C()\mathsf{new}\ \bar{C}() : T'\bar{T}' \dashv \Delta_2; \Gamma_2 * r$

Then notice that (4) is necessarily the conclusion of rule T-MULTI whose premises are:

(5) $\Delta_1; \Gamma_1 * r \vdash \mathsf{new}\ C() : T' \dashv \Delta_3; \Gamma_3 * r$

(6) $\Delta_3; \Gamma_3 * r \vdash \mathsf{new}\ \bar{C}() : \bar{T}' \dashv \Delta_2; \Gamma_2 * r$

Apply inversion to (5) to obtain $\Delta_3 = \Delta_1$ and $\Gamma_3 = \Gamma_1$. Then, combine (1), (2), (3) and (5) and apply rule T-STATE to get

(7) $\Delta_1; \Gamma_1 \vdash (h_1 * r, \mathsf{new}\ C()) : T' \dashv \Delta_1; \Gamma_1 * r$

By the induction hypothesis,

(8) $(h_1 * r, \mathsf{new}\ C()) \longrightarrow (h_3 * r, o)$

From the hypothesis ($\vdash P$), (7), (8) and Lemma 4.21 (Subject Reduction for Object Initialisation), $\Delta_1; \Gamma_1' \vdash (h_3 * r, o) : T' \dashv \Delta_1; \Gamma_2' * r$ whose premises are

(9) h_1 complete

(10) $\mathsf{dom}(\Gamma_1') \in \mathsf{dom}(h_3)$

(11) $\Delta_1; \Gamma_1' \vdash h_3$

(12) $\Delta_1; \Gamma_1' * r \vdash o : T' \dashv \Delta_1; \Gamma_2' * r$

By inversion of (12), we have $\Delta_1 \vdash \Gamma_1'$. From this and (6), use Lemma 4.14 (Closed Class Families) to obtain $\Delta_1; \Gamma_1' * r \vdash \mathsf{new}\ \bar{C}() : \bar{T}' \dashv \Delta_1; \Gamma_1' * r$. Combine this judgement with (9), (10), (11) and rule T-MULTISTATE to get $\Delta_1; \Gamma_1' \vdash (h_3 * r, \mathsf{new}\ \bar{C}()) : \bar{T} \dashv \Delta_1; \Gamma_1' * r$. Then by the induction hypothesis,

(13) $(h_3 * r, \mathsf{new}\ \bar{C}()) \longrightarrow (h_2 * r, \bar{o})$

From (8) and (13), conclude by applying rule R-MULTINEW. □

Chapter 5

Algorithmic Typechecking

Chapter 3 formalised a declarative type system for DOL's core language. The definitions presented there are designed to show how typing should behave and to simplify proofs, but are not immediately suitable for implementation. To define the algorithmic system, we modify the rules that require guessing quantifier instantiation. For this, we build on the techniques developed by Dunfield [2007], Dunfield and Krishnaswami [2013, 2016], applying them to a much simpler setting, that of our restricted language of indices. The basic idea is to defer quantifier instantiation by relying on new judgements, such as index variable instantiation and index equivalence, which will find the appropriate solutions and propagate the increased knowledge using final index contexts. We also apply bidirectional typechecking [Pierce and Turner, 2000] in order to distinguish rules that synthesize types from those that check terms against types already known. The result is a simple yet precise algorithm from which it is straightforward to read off an implementation.

Chapter Outline. This chapter is structured as follows:

- Section 5.1 presents the algorithmic type system.
- Section 5.2 proves that the algorithmic system is sound and complete with respect to the declarative system.
- Section 5.3 briefly describes the prototype, which is a direct implementation of the algorithmic typing rules.

5.1 Algorithmic Type System

We develop the algorithmic system in two steps. The first step is to introduce an existential index variable (written $\hat{a}$ with the hat in the style of Dunfield [2007], Dunfield and

Krishnaswami [2013, 2016]) into the initial index context whenever there is a need to make a guess at the appropriate index term i. The oracular rules in the declarative system are S-ΠL and S-ΣR, T-HIDE, T-SELFCALL, T-CALL and T-MTYPE. The corresponding algorithmic rules are defined by judgements that take an initial index context and yield a final index context, possibly augmented with knowledge about what index terms have to be. Instead of guessing, the algorithmic system adds judgements to instantiate existential index variables and equate index terms. A final index context in the algorithmic system also serves to propagate increased information, namely that $\hat{a}$ is equal to a specific i.

The second step is to apply bidirectional typing, a technique that easily supports subtyping and index refinements. In fact, bidirectional typechecking has become popular for specifying typecheckers in object-oriented languages such as C# [Bierman et al., 2007] and Scala [Odersky et al., 2001], languages with indexed and refinement types [Xi, 1998, Dunfield, 2007, Knowles et al., 2007, Lovas and Pfenning, 2008, Bierman et al., 2010], and languages with higher-rank polymorphism and indexed types [Jones et al., 2007, Dunfield and Krishnaswami, 2013, 2016]. We modify the typing rules for terms by distinguishing the rules which synthesize types from those that check terms against types already known.

In the process, we also eliminate the nondeterminism associated with the subtyping rules and the typing rules for paths. We arrive at an algorithmic type system for DOL that consists of type formation rules, index instantiation and equivalence rules, subtyping rules, and typing rules for paths and terms. The rest of this section is devoted to describing the technical details of this system.

Syntax. The algorithmic system uses the syntax and meta-variables of the declarative system (Figures 3.1 and 3.2).

Existential Index Variables. Index contexts in the declarative system may contain index variables a, regarded as universal quantifiers, and propositions p. However, index contexts in the algorithmic system need to include existential index variables, denoted by the meta-variable $\hat{a}$ introduced earlier, to be refined as typechecking proceeds. Rather than defining a distinct syntactic category, we again opt to simplify by taking existential index variables $\hat{a}$ to be a subset of the index variable names in Δ.

Index Contexts. Index contexts in the algorithmic system extend index contexts in the declarative system as follows:

$$\Delta \ ::= \ \ldots \ \mid \ \hat{a} : I \doteq i$$

In addition to *unsolved* existential index variable declarations $\hat{a} : I$, index contexts in the algorithmic system may also contain *solved* existential index variable declarations given above. As index contexts in the declarative system, index contexts in the algorithmic system are ordered sequences. For example, the index context in the algorithmic system $(\Delta, \hat{a} : I \doteq i)$ is said to be well-formed if $\hat{a} \notin \text{FV}(\Delta)$ and $\text{FV}(I) \cup \text{FV}(i) \in \Delta$.

Constraint-based Typechecking. In the declarative system, we rely on a set of rules that check index type formation, subtyping, typing and substitution (Figures 3.3–3.6), which use an external constraint solver only to check if propositions hold. In the algorithmic system, most of the work in the index language is handled by the external solver. We add another semantically defined relation of the form

$$\Delta \models i : I$$

that reads "$i : I$ holds under the assumptions in Δ". The solver will reject, for example, unknown identifiers and terms such as $(a > b)+1$ under a context $(b : \mathsf{integer})$. Of course, the typechecker itself still has to handle existential index variables (as we will see later). For example, we may want the constraint $\hat{a} : \mathsf{positive} \models \hat{a} \doteq 1$ to hold, meaning that we may want to instantiate the existential index variable $\hat{a}$ with the value 1. Although some constraint solvers provide (limited) support for quantifiers, undecidability may follow. As a result, we currently only give quantifier-free formulas to the external solver, so that typechecking remains decidable.

5.1.1 Algorithmic Type Formation

The rules for well-formed types in the declarative system (Figure 3.7) make use of the index formation relation in rule WF-$\star$ and of the index typing relation in rule K-APP. We thus need to define algorithmic versions that instead only make use of $\models$ judgements. This is done in Figure 5.1.

5.1.2 Quantifier Instantiation

We provide a rewrite system for integer equations in order to isolate existential index variables. The rules are given in Figure 5.2. We also define a set of auxiliary judgements that instantiate existential index variables and equate index terms. To instantiate unsolved existential index variables, we define a judgement of the form

$$\Delta_1 \vdash \hat{a} := i \dashv \Delta_2$$

$\boxed{\Delta \triangleright K}$ Under context Δ, kind K is alg. well-formed

$$\frac{\models \Delta}{\Delta \triangleright \star}\ (\text{AWF-}\star) \qquad \frac{\Delta, a : I \vdash K}{\Delta \triangleright \Pi a : I.K}\ (\text{AWF-}\Pi)$$

$\boxed{\Delta \triangleright T : K}$ Under context Δ, type T has alg. kind K

$$\frac{\mathsf{class}\ C : (\bar{a} : \bar{I})\ \mathsf{extends}\ \_\{\_\}\ \mathsf{is}\ \{\_\} \qquad \Delta \triangleright \Pi\bar{a} : \bar{I}.\star}{\Delta \triangleright C : \Pi\bar{a} : \bar{I}.\star}\ (\text{AK-CLASS})$$

$$\frac{\Delta \triangleright \star}{\Delta \triangleright \mathsf{Top} : \star}\ (\text{AK-TOP}) \qquad \frac{\Delta \triangleright C\bar{i} : \Pi a : I.K \qquad \Delta \models i : I}{\Delta \triangleright C\bar{i}i : K[i/a]}\ (\text{AK-APP})$$

$$\frac{\Delta, a : I \triangleright T : \star}{\Delta \triangleright \Pi a : I.T : \star}\ (\text{AK-}\Pi) \qquad \frac{\Delta, a : I \triangleright T : \star}{\Delta \triangleright \Sigma a : I.T : \star}\ (\text{AK-}\Sigma)$$

$$\frac{\Delta \triangleright T : \star \qquad \Delta \triangleright U : \star}{\Delta \triangleright T + U : \star}\ (\text{AK-}+) \qquad \frac{\Delta \triangleright T : \star \qquad \Delta \triangleright U : \star}{\Delta \triangleright T \times U : \star}\ (\text{AK-}\times)$$

$$\frac{\Delta \triangleright \bar{T} : \star}{\Delta \triangleright C[\{\bar{f} : \bar{T}\}] : \star}\ (\text{AK-RECORD})$$

Figure 5.1: Algorithmic type formation rules

$$\begin{array}{ll}
(i_1 + i_2 \doteq j) \xrightarrow{rw} (i_1 \doteq j - i_2) & \text{if } \hat{a} \in \text{FV}(i_1) \\
(i_1 + i_2 \doteq j) \xrightarrow{rw} (i_2 \doteq j - i_1) & \text{if } \hat{a} \in \text{FV}(i_2) \\
(i_1 - i_2 \doteq j) \xrightarrow{rw} (i_1 \doteq j + i_2) & \text{if } \hat{a} \in \text{FV}(i_1) \\
(i_1 - i_2 \doteq j) \xrightarrow{rw} (i_2 \doteq i_1 - j) & \text{if } \hat{a} \in \text{FV}(i_2)
\end{array}$$

Figure 5.2: Rewrite rules to isolate an existential index variable $\hat{a}$ appearing on the left-hand side of an equation (rules for $\hat{a}$ appearing on the right-hand side are omitted)

To equate index terms and propositions, we define the following three judgements:

$$\Delta_1 \vdash i \equiv j \dashv \Delta_2 \quad \text{and} \quad \Delta_1 \vdash p_1 \equiv p_2 \dashv \Delta_2 \quad \text{and} \quad \Delta_1 \vdash \bar{i} \equiv \bar{j} \dashv \Delta_2$$

The rules for these judgements are given in Figure 5.3, and below we provide additional definitions used by them.

Definition 5.1 (Index Context Update)**.** $(\Delta_1, \hat{a} : I, \Delta_2)\{\hat{a} \hookleftarrow i\} \triangleq \Delta_1, \hat{a} : I \doteq i, \Delta_2$.

Definition 5.2. *The set of unsolved existential index variables in a context Δ, notation*

$\boxed{\Delta_1 \vdash \hat{a} := i \dashv \Delta_2}$ Under initial context Δ_1, instantiate $\hat{a}$ such that $\hat{a} \doteq i$, with final context Δ_2

$$\frac{i \neq \hat{b} \qquad \Delta_1 \models i : I}{\Delta_1, \hat{a} : I, \Delta_2 \vdash \hat{a} := i \dashv \Delta_1, \hat{a} : I \doteq i, \Delta_2} \text{ (Q-SOLVE)}$$

$$\frac{\Delta_1, \hat{a} : I, \Delta_2 \models \hat{b} : I \qquad \hat{b} \in \mathsf{unsolved}(\Delta_1 \cup \Delta_2)}{\Delta_1, \hat{a} : I, \Delta_2 \vdash \hat{a} := \hat{b} \dashv (\Delta_1, \hat{a} : I, \Delta_2)\{\hat{b} \hookleftarrow \hat{a}\}} \text{ (Q-SOLVEEX)}$$

$\boxed{\Delta_1 \vdash i \equiv j \dashv \Delta_2}$ Under initial context Δ_1, index term i is equivalent to j, with final context Δ_2

$$\frac{\mathrm{FV}(i, j) \notin \mathsf{unsolved}(\Delta) \qquad \Delta \models i \doteq j}{\Delta \vdash i \equiv j \dashv \Delta} \text{ (EQ-ASSERT)}$$

$$\frac{\hat{a} \in \mathrm{FV}(i_1, i_2) \cap \mathsf{unsolved}(\Delta_1) \qquad (i_1 \oplus i_2 \doteq j) \xrightarrow{rw} (j_1 \doteq j_2) \qquad \Delta_1 \vdash j_1 \equiv j_2 \dashv \Delta_2}{\Delta_1 \vdash i_1 \oplus i_2 \equiv j \dashv \Delta_2} \text{ (EQ-}\oplus\text{L)}$$

$$\frac{\hat{a} \in \mathrm{FV}(i_1, i_2) \cap \mathsf{unsolved}(\Delta_1) \qquad (j \doteq i_1 \oplus i_2) \xrightarrow{rw} (j_1 \doteq j_2) \qquad \Delta_1 \vdash j_1 \equiv j_2 \dashv \Delta_2}{\Delta_1 \vdash j \equiv i_1 \oplus i_2 \dashv \Delta_2} \text{ (EQ-}\oplus\text{R)}$$

$$\frac{\hat{a} \notin \mathrm{FV}(i) \qquad \Delta_1 \vdash \hat{a} := i \dashv \Delta_2}{\Delta_1 \vdash \hat{a} \equiv i \dashv \Delta_2} \text{ (EQ-INSTL)}$$

$$\frac{i \neq \hat{b} \qquad \hat{a} \notin \mathrm{FV}(i) \qquad \Delta_1 \vdash \hat{a} := i \dashv \Delta_2}{\Delta_1 \vdash i \equiv \hat{a} \dashv \Delta_2} \text{ (EQ-INSTR)}$$

$\boxed{\Delta_1 \vdash p_1 \equiv p_2 \dashv \Delta_2}$ Under initial context Δ_1, proposition p_1 is equivalent to p_2, with final context Δ_2

$$\frac{\Delta_1 \vdash p_1 \equiv p_3 \dashv \Delta_3 \qquad \Delta_3 \vdash p_2 \equiv p_4 \dashv \Delta_4}{\Delta_1 \vdash p_1 \circledS p_2 \equiv p_3 \circledS p_4 \dashv \Delta_4} \text{ (EQ-}\circledS\text{)}$$

$$\frac{\Delta_1 \vdash p_1 \equiv p_3 \dashv \Delta_3 \qquad \Delta_3 \vdash p_2 \equiv p_4 \dashv \Delta_4}{\Delta_1 \vdash p_1 \oslash p_2 \equiv p_3 \oslash p_4 \dashv \Delta_4} \text{ (EQ-}\oslash\text{)}$$

$\boxed{\Delta_1 \vdash \bar{i} \equiv \bar{j} \dashv \Delta_2}$ Under initial context Δ_1, index term sequence $\bar{i}$ is equivalent to $\bar{j}$, with final context Δ_2

$$\frac{}{\Delta \vdash \epsilon \equiv \epsilon \dashv \Delta} \text{ (EQ-}\epsilon\text{)} \qquad \frac{\Delta_1 \vdash i \equiv j \dashv \Delta_2 \qquad \Delta_2 \vdash \bar{i} \equiv \bar{j} \dashv \Delta_3}{\Delta_1 \vdash i\bar{i} \equiv j\bar{j} \dashv \Delta_3} \text{ (EQ-MULTI)}$$

Figure 5.3: Index instantiation and equality rules

$\mathsf{unsolved}(\Delta)$, *can be inductively defined as follows:*

$$\begin{array}{lcl}
\mathsf{unsolved}(\epsilon) & = & \{\} \\
\mathsf{unsolved}(\Delta, a : I) & = & \mathsf{unsolved}(\Delta) \\
\mathsf{unsolved}(\Delta, \hat{a} : I) & = & \mathsf{unsolved}(\Delta) \cup \{\hat{a}\} \\
\mathsf{unsolved}(\Delta, \hat{a} : I \doteq i) & = & \mathsf{unsolved}(\Delta) \\
\mathsf{unsolved}(\Delta, p) & = & \mathsf{unsolved}(\Delta)
\end{array}$$

Rule Q-SOLVE sets $\hat{a}$ to i provided that $i : I$ follows from the assumptions in Δ_1, and i is not another existential index variable. On the other hand, rule Q-SOLVEEX is applied when the right-side index term is an existential index variable $\hat{b}$. We cannot set $\hat{a}$ to $\hat{b}$ because $\hat{b}$ may not well-formed under Δ_1. Instead, we set $\hat{b}$ to $\hat{a}$ as long as both existential index variables have the same type.

The other rules compare index terms. Rule EQ-ASSERT checks validity of a constraint that has no unsolved existential index variables. Rule EQ-$\oplus$L uses the rewrite system for equations (Figure 5.2) to isolate an existential index variable appearing on the left-hand side. We omit the corresponding equations for the right rule, since these are similar yet reversed. The instantiation of existential index variables occurs in the rightmost premise of the two symmetric rules EQ-INSTL and EQ-INSTR.

Example. Under an initial index context $\Delta_1 = (\hat{a} : \mathsf{positive})$, where $\mathsf{positive}$ abbreviates $\{a\colon \mathsf{integer} \mid a > 0\}$, take the conclusion $\hat{a}\ (\hat{a} + 1) \equiv 1\ 2$, and a final context $\Delta_2 = (\hat{a} : \mathsf{positive} \doteq 1)$. We omit the free variable premises in the example below. The derivation is as follows:

$$\dfrac{\dfrac{\dfrac{\Delta_1 \models 1 : \mathsf{positive}}{\Delta_1 \vdash \hat{a} := 1 \dashv \Delta_2}\ (\text{Q-SOLVE})}{\Delta_1 \vdash \hat{a} \equiv 1 \dashv \Delta_2}\ (\text{EQ-INSTL}) \qquad \dfrac{\Delta_2 \models (\hat{a} + 1) \doteq 2}{\Delta_2 \vdash (\hat{a} + 1) \equiv 2 \dashv \Delta_2}\ (\text{EQ-ASSERT})}{\Delta_1 \vdash \hat{a}\ (\hat{a} + 1) \equiv 1\ 2 \dashv \Delta_2}\ (\text{EQ-MULTI})$$

5.1.3 Algorithmic Subtyping

The subtyping rules in the declarative system (Figure 3.8) cannot be implemented directly for two main reasons. The first reason is the already mentioned index term i in the premises of rules S-ΠL and S-ΣR which is the result of guessing. The second reason is the nondeterminism introduced by rule S-TRANS and by some rules that are not syntax-directed. For example, when both types are quantifiers or unions, more than one

$\boxed{\Delta_1 \vdash T <: U \dashv \Delta_2}$ Under initial context Δ_1, type T is a subtype of U, with final context Δ_2

$$\frac{\mathsf{class}\ C : (\bar{a} : \bar{I})\ \mathsf{extends}\ T\{\_\}\ \mathsf{is}\ \{\_\} \quad \Delta_1 \vdash T[\bar{i}/\bar{a}] <: D\bar{j} \dashv \Delta_2 \quad C \neq D}{\Delta_1 \vdash C\bar{i} <: D\bar{j} \dashv \Delta_2}\ (\text{AS-SUPER})$$

$$\frac{\Delta \triangleright T^\circ : \star}{\Delta \vdash T^\circ <: \mathsf{Top} \dashv \Delta}\ (\text{AS-TOP}) \qquad \frac{\Delta_1 \vdash \bar{i} \equiv \bar{j} \dashv \Delta_2}{\Delta_1 \vdash C\bar{i} <: C\bar{j} \dashv \Delta_2}\ (\text{AS-APP})$$

$$\frac{\Delta_1, \hat{a} : I \vdash U[\hat{a}/a] <: T^\circ \dashv \Delta_2}{\Delta_1 \vdash \Pi a : I.U <: T^\circ \dashv \Delta_2}\ (\text{AS-}\Pi\text{L}) \qquad \frac{\Delta_1, a : I \vdash T <: U \dashv \Delta_2}{\Delta_1 \vdash T <: \Pi a : I.U \dashv \Delta_2}\ (\text{AS-}\Pi\text{R})$$

$$\frac{\Delta_1, a : I \vdash T <: U \dashv \Delta_2}{\Delta_1 \vdash \Sigma a : I.T <: U \dashv \Delta_2}\ (\text{AS-}\Sigma\text{L}) \qquad \frac{\Delta_1, \hat{a} : I \vdash T^\circ <: U[\hat{a}/a] \dashv \Delta_2}{\Delta_1 \vdash T^\circ <: \Sigma a : I.U \dashv \Delta_2}\ (\text{AS-}\Sigma\text{R})$$

$$\frac{\Delta_1 \vdash T_1 <: U \dashv \Delta_2 \qquad \Delta_1 \vdash T_2 <: U \dashv \Delta_2}{\Delta_1 \vdash (T_1 + T_2) <: U \dashv \Delta_2}\ (\text{AS-+L})$$

$$\frac{\Delta_1 \vdash T^\circ <: U_k \dashv \Delta_2}{\Delta_1 \vdash T^\circ <: (U_1 + U_2) \dashv \Delta_2}\ (\text{AS-+R}_k)$$

$$\frac{\Delta_1 \vdash T_1 <: U_1 \dashv \Delta_2 \qquad \Delta_1 \vdash T_2 <: U_2 \dashv \Delta_2}{\Delta_1 \vdash (T_1 \times T_2) <: (U_1 \times U_2) \dashv \Delta_2}\ (\text{AS-}\times)$$

$$\frac{\Delta_1 \vdash T_1 <: U_1 \dashv \Delta_2 \quad \ldots \quad \Delta_n \vdash T_n <: U_n \dashv \Delta_{n+1}}{\Delta_1 \vdash C[\{f_1 : T_1, \ldots, f_n : T_n\}] <: C[\{f_1 : U_1, \ldots, f_n : U_n\}] \dashv \Delta_{n+1}}\ (\text{AS-RECORD})$$

Figure 5.4: Algorithmic subtyping rules. T° means that T is not a type Π, Σ, or $+$. In rules AS-ΠL and AS-ΣR, index variable $\hat{a}$ is fresh.

rule can be tried. Specifically, in the judgement $\Delta \vdash \Pi a : I.Ca <: \Sigma b : J.Db$ we can choose to instantiate a or b using rules S-ΠL or S-ΣR. If we choose the former, then we have to prove $\Delta \vdash Ci <: \Sigma b : J.Db$ where b may depend on a but not the opposite. If we choose the latter, then we must prove $\Delta \vdash \Pi a : I.Ca <: Di$ where the dependencies are reversed [Dunfield and Krishnaswami, 2016].

The algorithmic subtyping rules in Figure 5.4 use judgements of the form

$$\Delta_1 \vdash T <: U \dashv \Delta_2$$

where the final index context Δ_2 may carry information about solved existential index variables, obtained along the derivation from the rules presented earlier (Figure 5.3).

To make the system deterministic, we drop rule S-TRANS. However, we cannot eliminate transitivity completely, so we allow it along the hierarchy of classes in rule AS-SUPER. We also make sure that at most one rule applies, forming an algorithm when rules are read upwards. We write T° to denote a type that is not a Π, Σ, or $+$. For example, the only rule deriving a judgement with a Σ as a supertype is AS-ΣR.

Rule AS-SUPER has a different final context that comes from its subtyping premise.

$\boxed{\Delta \triangleright \Gamma}$ Under context Δ, context Γ is alg. well-formed

$$\frac{\models \Delta}{\Delta \triangleright \epsilon}\ (\text{AWF-EMPTY}\Gamma) \qquad \frac{\Delta \triangleright \Gamma \qquad x \notin \mathsf{dom}(\Gamma) \qquad \Delta \triangleright T : K}{\Delta \triangleright \Gamma, x : T}\ (\text{AWF-}\Gamma)$$

Figure 5.5: Algorithmic formation rules for object contexts

$\boxed{\begin{array}{l}\Delta_1; \Gamma \vdash r \uparrow T \dashv \Delta_2 \\ \Delta_1; \Gamma \vdash r \uparrow_h T \dashv \Delta_2\end{array}}$ Under contexts $\Delta_1; \Gamma$, path r synthesizes type T, with final context Δ_2

$$\frac{\Delta_1 \triangleright \Gamma \qquad U = \Sigma \Delta_2 . T}{\Delta_1; \Gamma, \mathsf{this} : U \vdash \mathsf{this} \uparrow T \dashv \Delta_1, \Delta_2}\ (\text{AT-REF})$$

$$\frac{\Delta_1; \Gamma \vdash \mathsf{this} \uparrow C[F] \dashv \Delta_1 \qquad F(f) = \Sigma \Delta_2 . T}{\Delta_1; \Gamma \vdash \mathsf{this}.f \uparrow T \dashv \Delta_1, \Delta_2}\ (\text{AT-FIELD})$$

$$\frac{\Delta_1; \Gamma \vdash \mathsf{this} \uparrow C[F] \dashv \Delta_1 \quad \mathsf{class}\ C : (\bar{a} : \bar{I})\ \mathsf{extends}\ \_\{\_\}\ \mathsf{is}\ \{\_\} \qquad \Delta_1, \bar{\hat{a}} : \bar{I} \vdash C[F] <: \mathsf{fields}(C)[\bar{\hat{a}}/\bar{a}] \dashv \Delta_2 \qquad \bar{\hat{a}}\ \mathsf{fresh}}{\Delta_1; \Gamma \vdash \mathsf{this} \uparrow_h C\bar{\hat{a}} \dashv \Delta_2}\ (\text{AT-HIDE})$$

Figure 5.6: Type synthesis rules for paths

Rule AS-TOP does not involve existential index variables, hence uses the same initial and final contexts. The key rule is AS-APP whose premise $\Delta_1 \vdash \bar{i} \equiv \bar{j} \dashv \Delta_2$ solves existential index variables. For example, $Ci <: C\hat{a}$ will lead to $\hat{a} : I \doteq i$ being an entry in the final index context Δ_2 (provided $\hat{a} : I$ is declared in Δ_1 and $\hat{a} \notin \mathrm{FV}(i)$ and $i : I$ follows from the assumptions in Δ_1).

The two rules AS-ΠL and AS-ΣR do not make a guess at the appropriate index term i, unlike the corresponding declarative rules. Instead, they add a fresh existential index variable $\hat{a}$ to the initial context of their premise, replacing a with $\hat{a}$ in the appropriate type, so that a solution will be added to the final context. Rules AS-ΠR and AS-ΣL are similar to the ones of the declarative system, except for the final index contexts, and so are rules AS-$+$L and AS-$\times$. Rule AS-$+\mathrm{R}_k$ is applied when a union $U_1 + U_2$ is a supertype, provided the subtype is not a Π, Σ, or $+$. The rule checks first against U_1 and, if that fails, it checks against U_2. Rule AS-RECORD compares field typings by taking into account final contexts that may contain solutions to existential index variables.

5.1.4 Bidirectional Typechecking

When checking a path, we propagate type information using a judgement of the form

$$\Delta_1; \Gamma \vdash r \uparrow T \dashv \Delta_2$$

where $\uparrow$ is meant to denote type synthesis. In the algorithmic system, paths r take the form of either the current object this or a field $\mathsf{this}.f$. The rules are given in Figure 5.6. Rules AT-REF and AT-FIELD unpack a path if its synthesized type begins with an existential quantifier (cf. Figure 3.11). Rule AT-HIDE, instead of replacing the class quantifiers $\bar{a}$ with guessed $\bar{i}$ of the same type, replaces them with fresh existential index variables $\bar{\hat{a}}$, which are added to the initial context of the subtyping premise. The rule returns the top-level type $C\bar{\hat{a}}$ together with a final context that contains solutions to $\bar{\hat{a}}$.

As mentioned, in the bidirectional typechecking algorithm [Pierce and Turner, 2000], we alternate between synthesizing types and checking terms against types already known. Bidirectional typechecking in DOL is formalised by replacing the typing judgement of the form

$$\Delta_1; \Gamma_1 * r_1 \vdash t : T \dashv \Delta_2; \Gamma_2 * r_2$$

with the following two judgements:

$$\begin{array}{ll} \Delta_1; \Gamma_1 \vdash t \uparrow T \dashv \Delta_2; \Gamma_2 & \text{synthesizing} \\ \Delta_1; \Gamma_1 \vdash t \downarrow T \dashv \Delta_2; \Gamma_2 & \text{checking} \end{array}$$

The intuition is that when a rule is applied the typechecker's "direction" is either towards the root of the syntax tree (synthesis) as it traverses the program, or down towards the leaves (checking), propagating down a known type T to check a term t against it. In practice, however, the difference is between which part of the judgements are *inputs* and *outputs*. During synthesis, the inputs are $\Delta_1; \Gamma_1$ and t, and so the output is T as well as the final contexts $\Delta_2; \Gamma_2$, whereas during checking we already know T and make sure the type of t is a subtype of T. While a path r may represent this or $\mathsf{this}.f$, the currently active object this does not change. Therefore, in the algorithmic judgements we omit r_1 and r_2, which are required for typing runtime terms and proving type soundness, but never for typechecking a program.

Many of the rules in Figure 5.7 follow the corresponding declarative versions (Figure 3.12), using the auxiliary functions and predicates (Figure 3.9), as well as the several definitions given in Section 3.2.6. We define below an additional operation of composition over contexts.

Definition 5.3 (Type Equivalence)**.** *Let $\Delta \triangleright T : K$ and $\Delta \triangleright U : K$. We say that T and U are equivalent types under Δ if $\Delta \vdash T <: U \dashv \_$ and $\Delta \vdash U <: T \dashv \_$.*

Definition 5.4 (Context Composition)**.** *Context composition, denoted by $(\Delta_1; \Gamma_1 \cdot \Delta_2; \Gamma_2)$, with $\Gamma_1 = (\Gamma, \mathsf{this} : C[f_1 : T_1, \ldots, f_n : T_n])$ and $\Gamma_2 = (\Gamma, \mathsf{this} : C[f_1 : U_1, \ldots, f_n : U_n])$,*

$\boxed{\Delta_1;\Gamma_1 \vdash t \uparrow T \dashv \Delta_2;\Gamma_2}$ Under initial contexts $\Delta_1;\Gamma_1$, term t synthesizes type T, with final contexts $\Delta_2;\Gamma_2$

$\boxed{\Delta_1;\Gamma_1 \vdash t \downarrow T \dashv \Delta_2;\Gamma_2}$ Under initial contexts $\Delta_1;\Gamma_1$, term t checks against input type T, with final contexts $\Delta_2;\Gamma_2$

$$\frac{\Delta \triangleright \Gamma \qquad \mathsf{un}(T)}{\Delta;\Gamma, x:T \vdash x \uparrow T \dashv \Delta;\Gamma, x:T}\ (\text{AT-UnVar})$$

$$\frac{\Delta \triangleright \Gamma \qquad \mathsf{lin}(T)}{\Delta;\Gamma, x:T \vdash x \uparrow T \dashv \Delta;\Gamma}\ (\text{AT-LinVar})$$

$$\frac{\Delta;\Gamma \vdash \mathsf{this} \uparrow C[F] \dashv \Delta \qquad \mathsf{un}(F(f))}{\Delta;\Gamma \vdash f \uparrow F(f) \dashv \Delta;\Gamma}\ (\text{AT-UnField})$$

$$\frac{\Delta \triangleright \Gamma}{\Delta;\Gamma \vdash \mathsf{new}\ C() \uparrow C.\mathsf{init} \dashv \Delta;\Gamma}\ (\text{AT-New})$$

$$\frac{\begin{array}{c}\Delta_1;\Gamma_1 \vdash t \uparrow T \dashv \Delta_2;\Gamma_2 \\ \Delta_2;\Gamma_2 \vdash \mathsf{this} \uparrow C[F] \dashv \Delta_2 \qquad \Delta_2;\Gamma_2\{\mathsf{this}.f \hookleftarrow T\} \vdash \mathsf{this} \uparrow_h C\bar{i} \dashv \Delta_3\end{array}}{\Delta_1;\Gamma_1 \vdash f := t \uparrow F(f) \dashv \Delta_2;\Gamma_2\{\mathsf{this}.f \hookleftarrow T\}}\ (\text{AT-Assign})$$

$$\frac{\Delta_1;\Gamma_1 \vdash t_1 \uparrow U \dashv \Delta_2;\Gamma_2 \qquad \Delta_2;\Gamma_2 \vdash t_2 \uparrow T \dashv \Delta_3;\Gamma_3 \qquad \mathsf{un}(U)}{\Delta_1;\Gamma_1 \vdash t_1; t_2 \uparrow T \dashv \Delta_3;\Gamma_3}\ (\text{AT-Seq})$$

$$\frac{\begin{array}{c}\Delta_1;\Gamma_1 \vdash \mathsf{this} \uparrow_h C\bar{i} \dashv \Delta_2 \qquad \mathsf{mtype}(m, C\bar{i}) = \Pi(\bar{a}:\bar{I}).(C\bar{i} \rightsquigarrow T \times U \to W) \\ \Delta_2, \bar{\hat{a}}:\bar{I};\Gamma_1 \vdash t \downarrow U[\bar{\hat{a}}/\bar{a}] \dashv \Delta_3;\Gamma_2 \qquad \bar{\hat{a}}\ \mathsf{fresh} \\ \Delta_3;\Gamma_2\{\mathsf{this} \hookleftarrow T[\bar{\hat{a}}/\bar{a}]\} \vdash \mathsf{this} \uparrow C\bar{j} \dashv \Delta_4\end{array}}{\Delta_1;\Gamma_1 \vdash m(t) \uparrow W[\bar{\hat{a}}/\bar{a}] \dashv \Delta_4;\Gamma_2\{\mathsf{this} \hookleftarrow \mathsf{fields}(C\bar{j})\}}\ (\text{AT-SelfCall})$$

$$\frac{\begin{array}{c}\Delta_1;\Gamma_1 \vdash \mathsf{this}.f \uparrow T_1 \dashv \Delta_2 \qquad \mathsf{mtype}(m, T_1) = \Pi(\bar{a}:\bar{I}).(T_1 \rightsquigarrow T_2 \times U \to W) \\ \Delta_2, \bar{\hat{a}}:\bar{I};\Gamma_1 \vdash t \downarrow U[\bar{\hat{a}}/\bar{a}] \dashv \Delta_3;\Gamma_2 \qquad \bar{\hat{a}}\ \mathsf{fresh} \\ \Delta_3;\Gamma_2\{\mathsf{this}.f \hookleftarrow T_2[\bar{\hat{a}}/\bar{a}]\} \vdash \mathsf{this} \uparrow_h C\bar{i} \dashv \Delta_4\end{array}}{\Delta_1;\Gamma_1 \vdash f.m(t) \uparrow W[\bar{\hat{a}}/\bar{a}] \dashv \Delta_3;\Gamma_2\{\mathsf{this}.f \hookleftarrow T_2[\bar{\hat{a}}/\bar{a}]\}}\ (\text{AT-Call})$$

$$\frac{\begin{array}{c}\Delta_1;\Gamma_1 \vdash \mathsf{this}.f \uparrow (U_1 + U_2) \dashv \Delta_2 \qquad \mathsf{classof}(U_k) = C_k \\ \Delta_2;\Gamma_1\{\mathsf{this}.f \hookleftarrow U_k\} \vdash t_k \uparrow T_k \dashv \Delta_{k+2};\Gamma_{k+2} \qquad C_1 \neq C_2\end{array}}{\Delta_1;\Gamma_1 \vdash \mathsf{case}\ f\ \mathsf{of}\ (C_k \Rightarrow t_k)_{k\in 1,2} \uparrow (T_1 + T_2) \dashv (\Delta_3;\Gamma_3 \cdot \Delta_4;\Gamma_4)}\ (\text{AT-Case})$$

$$\frac{\begin{array}{c}\Delta_1;\Gamma_1 \vdash t \uparrow \mathsf{Boolean}\ p \dashv \Delta_2;\Gamma_2 \\ \Delta_2, p;\Gamma_2 \vdash t_1 \uparrow T_1 \dashv \Delta_3, p, \Delta_3';\Gamma_3 \qquad \Delta_2, \neg p;\Gamma_2 \vdash t_2 \uparrow T_2 \dashv \Delta_4, \neg p, \Delta_4';\Gamma_4\end{array}}{\Delta_1;\Gamma_1 \vdash \mathsf{if}\ t\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 \uparrow (T_1 + T_2) \dashv (\Delta_3, \Delta_3';\Gamma_3 \cdot \Delta_4, \Delta_4';\Gamma_4)}\ (\text{AT-If})$$

$$\frac{\Delta_1;\Gamma_1 \vdash t_1 \uparrow \mathsf{Boolean}\ p \dashv \Delta_2;\Gamma_2 \qquad \Delta_2, p;\Gamma_2 \vdash t_2 \downarrow \mathsf{Top} \dashv \Delta_2;\Gamma_2}{\Delta_1;\Gamma_1 \vdash \mathsf{while}\ t_1\ \mathsf{do}\ t_2 \downarrow \mathsf{Top} \dashv \Delta_2, \neg p;\Gamma_2}\ (\text{AT-While})$$

$$\frac{\Delta_1;\Gamma_1 \vdash t \uparrow U \dashv \Delta_2;\Gamma_2 \qquad \Delta_2 \vdash U <: T \dashv \Delta_3}{\Delta_1;\Gamma_1 \vdash t \downarrow T \dashv \Delta_3;\Gamma_2}\ (\text{AT-Sub})$$

Figure 5.7: Type synthesis and type checking rules for terms

$\boxed{\triangleright_C\ M}$ Method M is alg. well-formed in class C

$$\frac{\begin{array}{c}\mathsf{class}\ C : \Delta_1\ \mathsf{extends}\ \_\{\ldots, m : \Pi\Delta_2.(T_1 \rightsquigarrow T_2 \times U \to W), \ldots\}\ \mathsf{is}\ \{\_\} \\ \Delta_1, \Delta_2; x : U, \mathsf{this} : C[\mathsf{fields}(T_1)] \vdash t \downarrow W \dashv \Delta_3; \Gamma, \mathsf{this} : C[F] \\ x : U \in \Gamma \Rightarrow \mathsf{un}(U) \qquad \Delta_3 \vdash C[F] <: \mathsf{fields}(T_2) \dashv \Delta_4 \qquad m \neq \mathsf{init}\end{array}}{\triangleright_C\ m(x) = t}\ \text{(AT-METHOD)}$$

$$\frac{\mathsf{fields}(C.\mathsf{init}) = C[F] \quad \epsilon; \epsilon \vdash \mathsf{new}\ \bar{C}() \uparrow F(\bar{f}) \dashv \epsilon; \epsilon \quad \text{no cycles in } C}{\triangleright_C\ \mathsf{init}() = \bar{f} := \mathsf{new}\ \bar{C}()}\ \text{(AT-INIT)}$$

$\boxed{\Delta \triangleright_T\ l : U}$ Member l has alg. type U with supertype T

$$\frac{\Delta \triangleright U : \star}{\Delta \triangleright_T\ f : U}\ \text{(AT-FIELD)}$$

$$\frac{\begin{array}{c}\mathsf{mtype}(m, T)\ \text{undefined} \quad \mathsf{class}\ C : (\bar{a} : \bar{I})\ \mathsf{extends}\ \_\{\_\}\ \mathsf{is}\ \{\_\} \\ \Delta_1 \triangleright \Pi\Delta_2.(C\bar{i} \times U \times W) : \star \quad \Delta_1, \Delta_2, \bar{\hat{a}} : \bar{I} \vdash C\bar{\hat{a}} <: T_2 \dashv \Delta_3 \quad \bar{\hat{a}}\ \mathsf{fresh}\end{array}}{\Delta_1 \triangleright_T\ m : \Pi\Delta_2.(C\bar{i} \rightsquigarrow T_2 \times U \to W)}\ \text{(AT-MTYPE)}$$

$$\frac{\begin{array}{c}\mathsf{mtype}(m, T) = \Pi\Delta_2'.(T_1' \rightsquigarrow T_2' \times U' \to W') \\ \Delta_1 \vdash \Pi\Delta_2.(T_1 \times T_2 \times U' \times W) <: \Pi\Delta_2'.(T_1' \times T_2' \times U \times W') \dashv \Delta_3\end{array}}{\Delta_1 \vdash_T\ m : \Pi\Delta_2.(T_1 \rightsquigarrow T_2 \times U \to W)}\ \text{(AT-OVERRIDE)}$$

$\boxed{\triangleright\ L}$ Class L is alg. well-formed

$$\frac{\Delta \triangleright T : \star \qquad \Delta \triangleright_T\ \bar{l} : \bar{T} \qquad \triangleright_C\ \bar{M}}{\triangleright\ \mathsf{class}\ C : \Delta\ \mathsf{extends}\ T\{\bar{l} : \bar{T}\}\ \mathsf{is}\ \{\bar{M}\}}\ \text{(AT-CLASS)}$$

$\boxed{\triangleright\ P}$ Program P is alg. well-formed

$$\frac{\triangleright\ L_1 \quad \ldots \quad \triangleright\ L_n}{\triangleright\ L_1 \ldots L_n}\ \text{(AT-PROGRAM)}$$

Figure 5.8: Algorithmic typing rules for program formation

produces contexts $\Delta_3; \Gamma_3$ *where* $\Delta_3 = \Delta_1 \cup \Delta_2$ *and* $\Gamma_3 = (\Gamma, \mathsf{this} : C[f_1 : V_1, \ldots, f_n : V_n])$ *with* $V_k = T_k$ *if* T_k *and* U_k *are equivalent types under* Δ_3*, or* $V_k = (T_k + U_k)$ *and* $1 \leq k \leq n$*.*

Rules AT-UNVAR and AT-LINVAR propagate the type from the assumption $x : T$, without generating any new index information, so the final index context in the conclusion is the same as the initial index context. Rule AT-UNFIELD is analogous to the corresponding declarative rule, using the same initial and final index context. Rules AT-NEW and AT-SEQ also follow the declarative rules. Rule AT-ASSIGN is the same as in the declarative system, except that it can safely discard the final index context from the premise $\Delta_2; \Gamma_2\{\mathsf{this}.f \hookleftarrow T\} \vdash \mathsf{this} \uparrow_h C\bar{i} \dashv \Delta_3$ since the types used in the conclusion do

not depend on any solutions in Δ_3.

The two rules for method calls are more interesting. Rule AT-SELFCALL generates fresh index variables $\bar{\hat{a}} : \bar{I}$, with $\bar{I}$ issued by the method signature, and adds them to the initial index context of the premise that checks the type of the parameter. Its final context may contain solutions to $\bar{\hat{a}}$, which are then propagated to the remaining premise and the conclusion, so that any dependencies in the output types will be solved in the final context used in the conclusion. Rule AT-CALL is similar, except that, as AT-ASSIGN, discards the final index context from the derivation of its premise that uses rule AT-HIDE.

In rule AT-CASE, the field synthesizes a union type, while the two branches synthesize types T_1 and T_2. The entire term synthesizes their union $(T_1 + T_2)$, and final contexts are produced by composition of the final contexts of the branches. In rule AT-IF, the condition synthesizes a Boolean type, while the final type and contexts are obtained as in rule AT-CASE by composition at the meet-point. It is possible, however, that the context composition gives a union type to a field that needs to have a type of the form $C\bar{i}$, for example in order to call a method. A field that has been given an inappropriate union type may be given a single type by joining the constraints issued by each type.

In rule AT-WHILE, the condition synthesizes a Boolean type and the body of the loop checks against Top, while contexts are invariant. Finally, rule AT-SUB in the algorithmic system states that a term checks against a type T if it synthesizes a subtype of T.

Figure 5.8 defines the algorithmic rules for program formation. Rule AT-MTYPE does not guess index terms, but instead solves the fresh quantifiers in a type $C\bar{\hat{a}}$ along the derivation of its subtyping premise. The other rules do not differ from those in the declarative system, except for the use of the algorithmic index contexts.

5.2 Correctness of the Algorithmic System

We now prove that the algorithmic system is sound and complete with respect to the declarative system. We build our exposition on Dunfield and Krishnaswami [2013, 2016]'s proofs. For this, we require additional definitions of complete index contexts and of index contexts, applied as substitutions, to index terms, types and contexts.

We can view a solved existential index declaration as a substitution for its solved existential index variable. We therefore denote by $i_1[\hat{a} : I \doteq i_2]$ the capture-avoiding substitution of i_2 for the free occurrences of $\hat{a}$ in i_1, defined inductively on the structure of index terms. For example, $\hat{b}[\hat{a} : I \doteq i] \triangleq i$ if $\hat{b} = \hat{a}$, otherwise $\hat{b}[\hat{a} : I \doteq i] \triangleq \hat{b}$; on the other hand $(i_1 + i_2)[\hat{a} : I \doteq i_3] \triangleq i_1[\hat{a} : I \doteq i_3] + i_2[\hat{a} : I \doteq i_3]$.

A single index substitution is extended pointwise to a context Δ applied as a substitu-

tion to an index term by defining

$$\begin{array}{lcl} i[\epsilon] & = & i \\ i[\hat{a} : I \doteq j, \Delta] & = & (i[\hat{a} : I \doteq j])[\Delta] \\ i[\hat{a} : I, \Delta] & = & i[\Delta] \\ i[a : I, \Delta] & = & i[\Delta] \\ i[p, \Delta] & = & i[\Delta] \end{array}$$

Definition 5.5 (Complete Index Contexts)**.** *A complete index context, denoted by ϕ, is an algorithmic index context such that for every existential index variable $\hat{a}$ in $\mathsf{dom}(\phi)$, $\phi(\hat{a}) \triangleq \hat{a} : I \doteq i$.*

Definition 5.6 (Index Context Substitution)**.** *The result of applying a complete context ϕ as a substitution to a context Δ, denoted by $\Delta[\phi]$, is inductively defined as follows:*

$$\begin{array}{lcl} \epsilon[\epsilon] & = & \epsilon \\ \Delta[\phi, \hat{a} : I := i] & = & \begin{cases} \Delta'[\phi] & \textit{if } \Delta = (\Delta', \hat{a} : I := i) \\ \Delta'[\phi] & \textit{if } \Delta = (\Delta', \hat{a} : I) \\ \Delta[\phi] & \textit{otherwise} \end{cases} \\ (\Delta, a : I)[\phi, a : J] & = & \Delta[\phi], a : I[\phi] \textit{ if } I[\phi] = J[\phi] \\ (\Delta, p)[\phi] & = & \Delta[\phi], p[\phi] \end{array}$$

In all other cases, index context substitution is undefined.

Definition 5.7 (Index Context Extension)**.** *We say that an index context Δ_2 extends some index context Δ_1 if $\mathsf{dom}(\Delta_1) \subseteq \mathsf{dom}(\Delta_2)$ and $\Delta_1[\phi] = \Delta_2[\phi]$ for some ϕ.*

We can also apply a complete index context ϕ as a substitution to a well-formed index context Δ in order to obtain a declarative index context. Index context substitution works by dropping all the existential index declarations in Δ and applying ϕ to declarations of the form $a : I$. It is defined if and only if ϕ extends Δ.

The application of an index context as a substitution to a (well-formed) type T, denoted by $T[\Delta]$, is defined inductively on the structure of T, e.g. $(Ci_1 \ldots i_n)[\Delta] \triangleq C(i_1[\Delta]) \ldots (i_n[\Delta])$. Notice that the application of an index context as a substitution to a type T also drops all the existential index variables in T; the result is a type in the declarative system. Similarly, the application of ϕ to an object context Γ, denoted by $\Gamma[\phi]$, is a context where all the free occurrences of index existential variables have been replaced by index terms.

5.2.1 Soundness

To show that the algorithmic system is sound with respect to the original system, we are given an algorithmic judgement, with an initial index context Δ_1 and a final index context Δ_2, and ϕ as a solved extension of context Δ_2, and hence $\mathsf{dom}(\Delta_1) \subseteq \mathsf{dom}(\phi)$ (by transitivity). Applying ϕ as a substitution to the given algorithmic judgement produces a declarative judgement, which is the result we want to obtain.

Lemma 5.8 (Soundness of Type Formation)**.** *If* $\Delta \triangleright T : K$ *and* ϕ *extends* Δ*, then* $\Delta[\phi] \vdash T[\phi] : K[\phi]$.

Proof. By rule induction on the derivation of the first hypothesis. □

Lemma 5.9 (Soundness of Instantiation)**.** *If* $\Delta_1 \vdash \hat{a} := i \dashv \Delta_2$ *and* $\hat{a} \notin FV(i[\Delta_1])$ *and* $i[\Delta_1] = i$ *and* ϕ *extends* Δ_2*, then* $\hat{a}[\phi] = i[\phi]$.

Proof. By rule induction on the derivation of the first hypothesis.

Case Q-SOLVE**.**

$$\frac{i \neq \hat{b} \qquad \Delta_3 \models i : I}{\underbrace{\Delta_3, \hat{a} : I, \Delta_4}_{\Delta_1} \vdash \hat{a} := i \dashv \underbrace{\Delta_3, \hat{a} : I \doteq i, \Delta_4}_{\Delta_2}}$$

From the hypothesis $i[\Delta_1] = i$ and the fact that Δ_2 contains a solution for $\hat{a}$, we have $\hat{a}[\Delta_2] = i[\Delta_2]$. By definition of context extension, we have $\mathsf{dom}(\Delta_2) \subseteq \mathsf{dom}(\phi)$. Apply ϕ to each side to conclude $\hat{a}[\phi] = i[\phi]$.

Case Q-SOLVEEX**.**

$$\frac{\Delta_1 \models \hat{b} : I \qquad \hat{b} \in \mathsf{unsolved}(\Delta_3 \cup \Delta_4)}{\underbrace{\Delta_3, \hat{a} : I, \Delta_4}_{\Delta_1} \vdash \hat{a} := \underbrace{\hat{b}}_{i} \dashv \underbrace{(\Delta_3, \hat{a} : I, \Delta_4)\{\hat{b} \hookleftarrow \hat{a}\}}_{\Delta_2}}$$

By definition, $\hat{a}[\Delta_2] = \hat{b}[\Delta_2]$. Apply ϕ to each side to conclude $\hat{a}[\phi] = \hat{b}[\phi]$. □

Lemma 5.10 (Soundness of Index Equivalence)**.** *If* $\Delta_1 \vdash i \equiv j \dashv \Delta_2$ *and* ϕ *extends* Δ_2*, then* $\Delta_2[\phi] \models i[\phi] \doteq j[\phi]$*. (And similarly for* $\Delta_1 \vdash \overline{i} \equiv \overline{j} \dashv \Delta_2$*.)*

Proof. By rule induction on the derivation of the first hypothesis. □

Lemma 5.11 (Soundness of Context Formation)**.** *If* $\Delta \triangleright \Gamma$ *and* ϕ *extends* Δ*, then* $\Delta[\phi] \vdash \Gamma[\phi]$.

Proof. By rule induction on the derivation of the first hypothesis. □

Theorem 5.12 (Soundness of Algorithmic Subtyping)**.** *If* $\Delta_1 \triangleright T : \star$ *and* $\Delta_1 \triangleright U : \star$ *and* $T[\Delta_1] = T$ *and* $U[\Delta_1] = U$ *and* $\Delta_1 \vdash T <: U \dashv \Delta_2$ *and* ϕ *extends* Δ_2*, then* $\Delta_2[\phi] \vdash T[\phi] <: U[\phi]$.

Proof. By rule induction on the derivation of $\Delta_1 \vdash T <: U \dashv \Delta_2$.

Case AS-SUPER**.**

$$\frac{\mathsf{class}\ C : (\bar{a} : \bar{I})\ \mathsf{extends}\ V\{\_\}\ \mathsf{is}\ \{\_\} \qquad \Delta_1 \vdash V[\bar{i}/\bar{a}] <: D\bar{j} \dashv \Delta_2 \quad C \neq D}{\Delta_1 \vdash C\bar{i} <: D\bar{j} \dashv \Delta_2}$$

Notice that the class declaration and the rules for program formation imply that V is of the form $B\bar{i}'$. Use the fact that the hypothesis $\Delta_1 \triangleright C\bar{i} : \star$ is derived by an application of rule AK-CLASS, possibly followed by rule AK-APP applied as many times as needed, to build $\Delta_1 \models \bar{i} : \bar{I}$, which can be used to derive $\Delta_1 \triangleright V[\bar{i}/\bar{a}] : \star$. From this, the hypothesis $\Delta_1 \triangleright D\bar{j} : \star$ and the second premise of rule AS-SUPER, use the induction hypothesis in order to obtain $\Delta_2[\phi] \vdash (V[\bar{i}/\bar{a}])[\phi] <: D\bar{j}[\phi]$. Then use Lemma 4.13 (Agreement of Judgements) to get $\Delta_2[\phi] \vdash (V[\bar{i}/\bar{a}])[\phi] : \star$, and from that build $\Delta_2 \vdash \bar{i}' : \bar{I}'$ (using rule K-CLASS and possibly K-APP). Apply S-SUPER to the class declaration above together with these two results in order to deduce $\Delta_2[\phi] \vdash C\bar{i}[\phi] <: (V[\bar{i}/\bar{a}])[\phi]$. Conclude from this by using rule S-TRANS with the subtyping judgement obtained above.

Case AS-TOP**.**

$$\frac{\Delta_1 \triangleright U^\circ : \star}{\Delta_1 \vdash U^\circ <: \mathsf{Top} \dashv \Delta_2}$$

Then $\Delta_2 = \Delta_1$, and hence $\mathsf{dom}(\Delta_1) \subseteq \mathsf{dom}(\phi)$. From the hypothesis, we have $\Delta_1 \triangleright \mathsf{Top} : \star$. Use the induction hypothesis in order to obtain $\Delta_1[\phi] \vdash U^\circ[\phi] <: \mathsf{Top}$. Apply rule AS-TOP to get $\Delta_1 \vdash \mathsf{Top} <: \mathsf{Top} \dashv \Delta_1$. Then use the induction hypothesis again to obtain $\Delta_1[\phi] \vdash \mathsf{Top} <: \mathsf{Top}$. Conclude by applying rule S-TRANS to the two subtyping judgements obtained above.

Case AS-APP**.**

$$\frac{\Delta_1 \vdash \bar{i} \equiv \bar{j} \dashv \Delta_2}{\Delta_1 \vdash C\bar{i} <: C\bar{j} \dashv \Delta_2}$$

Use Lemma 5.10 (Soundness of Index Equivalence) in order to obtain $\Delta_2[\phi] \models \bar{i}[\phi] \doteq \bar{j}[\phi]$. From the hypothesis, we have $\Delta_1 \triangleright C\bar{j} : \star$. Use Lemma 5.8 (Soundness of Type Formation) to get $\Delta_2[\phi] \vdash C\bar{j}[\phi] : \star$. Conclude by applying rule S-APP.

Case AS-ΠL**.**

$$\frac{\Delta_1, \hat{a} : I \vdash V[\hat{a}/a] <: U \dashv \Delta_2}{\Delta_1 \vdash \Pi a : I.V <: U \dashv \Delta_2}$$

Given the first hypothesis $\Delta_1 \triangleright \Pi a : I.V : \star$, apply substitution to obtain $\Delta_1, \hat{a} : I \triangleright V[\hat{a}/a] : \star$. By weakening on the second hypothesis, $\Delta_1, \hat{a} : I \triangleright U : \star$. Use the induction hypothesis to get $(\Delta_2, \hat{a} : I)[\phi] \vdash (V[\hat{a}/a])[\phi] <: U[\phi]$. By substitution, obtain $\Delta_2[\phi] \vdash i : I$, and conclude by using weakening and rule S-ΠL.

Case AS-ΠR**.**

$$\frac{\Delta_1, a : I \vdash T <: V \dashv \Delta_2}{\Delta_1 \vdash T <: \Pi a : I.V \dashv \Delta_2}$$

Use the first hypothesis and weakening to get $\Delta_1, a : I \triangleright T : \star$. From the second hypothesis, we also have $\Delta_1 \triangleright \Pi a : I.V : \star$ which is derived by rule AK-Π whose premise is $\Delta_1, a : I \triangleright V : \star$. Apply the induction hypothesis to obtain $(\Delta_2, a : I)[\phi] \vdash T[\phi] <: V[\phi]$. Conclude by using rule S-ΠR.

Case AS-ΣL**.**

$$\frac{\Delta_1, a : I \vdash T_1 <: U_1 \dashv \Delta_2}{\Delta_1 \vdash \Sigma a : I.T_1 <: U_1 \dashv \Delta_2}$$

Similar to case AS-ΠR.

Case AS-ΣR**.**

$$\frac{\Delta_1, \hat{a} : I \vdash T <: U_1[\hat{a}/a] \dashv \Delta_2}{\Delta_1 \vdash T <: \Sigma a : I.U_1 \dashv \Delta_2}$$

Similar to case AS-ΠL.

Case AS-$+\mathrm{R}_k$**.**

$$\frac{\Delta_1 \vdash U^\circ <: U_k \dashv \Delta_2}{\Delta_1 \vdash U^\circ <: (U_1 + U_2) \dashv \Delta_2}$$

From the second hypothesis, we have $\Delta_1 \triangleright (U_1 + U_2) : \star$. Reading the premises of rule AK-$+$, we also have $\Delta_1 \triangleright U_k : \star$. Use this and the first hypothesis $\Delta_1 \triangleright U^\circ : \star$ with the induction hypothesis to obtain $\Delta_1[\phi] \vdash U^\circ[\phi] <: U_k[\phi]$. Conclude by applying rule S-$+\mathrm{R}_k$.

The remaining cases are omitted, since they are similar to case AS-$+\mathrm{R}_k$. □

Theorem 5.13 (Soundness of Algorithmic Typing)**.** *Let ϕ be a complete index context that extends Δ_2.*

1. *If $\Delta_1; \Gamma \vdash r \uparrow T \dashv \Delta_2$, then $\Delta_1[\phi]; \Gamma[\phi] \vdash r : T[\phi] \dashv \Delta_2[\phi]$.*
2. *If $\Delta_1; \Gamma_1 \vdash t \uparrow T \dashv \Delta_2; \Gamma_2$, then $\Delta_1[\phi]; \Gamma_1[\phi] * \mathsf{this} \vdash t : T[\phi] \dashv \Delta_2[\phi]; \Gamma_2[\phi] * \mathsf{this}$.*
3. *If $\Delta_1; \Gamma_1 \vdash t \downarrow T \dashv \Delta_2; \Gamma_2$ and $\Delta_1 \triangleright T : \star$, then $\Delta_1[\phi]; \Gamma_1[\phi] * \mathsf{this} \vdash t : T[\phi] \dashv \Delta_2[\phi]; \Gamma_2[\phi] * \mathsf{this}$.*

Proof. By rule induction on typing derivations. Parts 2 and 3 are proved together using part 1. Much of this proof is just applying the induction hypothesis or the preceding lemmas to each premise to yield declarative judgements, followed by applying the appropriate declarative rule. We show two cases, the remaining are similar.

Case AT-UNVAR**.**

$$\frac{\Delta_1 \triangleright \Gamma \qquad \mathsf{un}(T)}{\Delta_1; \underbrace{\Gamma, x : T}_{\Gamma_1} \vdash x \uparrow T \dashv \Delta_2; \underbrace{\Gamma, x : T}_{\Gamma_2}}$$

Then $\Delta_2 = \Delta_1$, and hence $\mathsf{dom}(\Delta_1) \subseteq \mathsf{dom}(\phi)$ by definition of context extension. Use Lemma 5.11 (Soundness of Context Formation) to get $\Delta_1[\phi] \vdash \Gamma[\phi]$. From this and $\mathsf{un}(T[\phi])$, conclude by applying rule T-UNVAR.

Case AT-SUB**.**

$$\frac{\Delta_1; \Gamma_1 \vdash t \uparrow U \dashv \Delta_3; \Gamma_2 \qquad \Delta_3 \vdash U <: T \dashv \Delta_2}{\Delta_1; \Gamma_1 \vdash t \downarrow T \dashv \Delta_2; \Gamma_2}$$

Use the induction hypothesis on the first premise to obtain $\Delta_1[\phi]; \Gamma_1[\phi] * \mathsf{this} \vdash t : U[\phi] \dashv \Delta_3[\phi]; \Gamma_2[\phi] * \mathsf{this}$. By inspection of the typing rules, $\mathsf{dom}(\Delta_3) \subseteq \mathsf{dom}(\Delta_2)$, and hence by transitivity $\mathsf{dom}(\Delta_3) \subseteq \mathsf{dom}(\phi)$. From this, we have $\Delta_1[\phi]; \Gamma_1[\phi] * \mathsf{this} \vdash t : U[\phi] \dashv \Delta_2[\phi]; \Gamma_2[\phi] * \mathsf{this}$. From the hypothesis (part 3), we also have $\Delta_3 \triangleright T : \star$. Use agreement to get $\Delta_3 \triangleright U : \star$. Now, apply Theorem 5.12 (Soundness of Algorithmic Subtyping) and the definition of index context extension in order to obtain $\Delta_2[\phi] \vdash U[\phi] <: T[\phi]$. Conclude by applying rule T-SUB with the two declarative judgements obtained above. □

5.2.2 Completeness

To prove completeness of the algorithmic system, we somehow do the reverse of soundness: from a declarative derivation, which has no existential index variables, we obtain a complete index context along an algorithmic derivation. In completeness of algorithmic *subtyping*, we are given an initial index context Δ_1 and a complete index context ϕ_1 that extends it. In completeness of algorithmic *typing*, in addition we are given a final context Δ_1' that may extend Δ_1 (with the result of unpacking or with propositions, for example) such that $\mathsf{dom}(\Delta_1) \subseteq \mathsf{dom}(\Delta_1')$ and $\mathsf{dom}(\Delta_1') = \mathsf{dom}(\phi_1)$, and hence $\mathsf{dom}(\Delta_1) \subseteq \mathsf{dom}(\phi_1)$. We show that we can build an algorithmic derivation with a final context Δ_2. However, the algorithmic rules generate fresh index variables that may not be in Δ_1, Δ_1' or ϕ_1. So, completeness will also produce a complete index context ϕ_2 that extends both Δ_2 and ϕ_1 such that $\mathsf{dom}(\Delta_2) = \mathsf{dom}(\phi_2)$. Below, we outline the main

results for instantiation, subtyping and typing; the detailed proofs are routine and therefore omitted.

Lemma 5.14 (Completeness of Instantiation)**.** *Let ϕ_1 be a complete index context that extends Δ_1 such that $\mathsf{dom}(\Delta_1) = \mathsf{dom}(\phi_1)$ and $\Delta_1 \models i : I$ and $\hat{a} \in \mathsf{unsolved}(\Delta_1)$ and $\hat{a} \notin FV(i)$. If $\hat{a}[\phi] = i[\phi]$, then $\Delta_1 \vdash \hat{a} := i \dashv \Delta_2$ and there exists ϕ_2 that extends both Δ_2 and ϕ_1 such that $\mathsf{dom}(\Delta_2) = \mathsf{dom}(\phi_2)$.*

Proof. By induction on the shape of i. □

Lemma 5.15 (Completeness of Index Equivalence)**.** *Let ϕ_1 be a complete index context that extends Δ_1 such that $\mathsf{dom}(\Delta_1) = \mathsf{dom}(\phi_1)$. If $\vdash \Delta_1[\phi_1]$ and $\Delta_1[\phi] \models i[\phi] \doteq j[\phi]$, then $\Delta_1 \vdash i[\Delta_1] \equiv j[\Delta_1] \dashv \Delta_2$ and there exists ϕ_2 that extends both Δ_2 and ϕ_1 such that $\mathsf{dom}(\Delta_2) = \mathsf{dom}(\phi_2)$.*

Proof. By mutual induction on the shapes of $i[\Delta_1]$ and $j[\Delta_1]$. □

Theorem 5.16 (Completeness of Algorithmic Subtyping)**.** *Let ϕ_1 be a complete index context that extends Δ_1 such that $\mathsf{dom}(\Delta_1) = \mathsf{dom}(\phi_1)$. If $\Delta_1[\phi_1] \vdash T[\phi_1] : \star$ and $\Delta_1[\phi_1] \vdash U[\phi_1] : \star$ and $\Delta_1[\phi_1] \vdash T[\phi_1] <: U[\phi_1]$, then $\Delta_1 \vdash T[\Delta_1] <: U[\Delta_1] \dashv \Delta_2$ and there exists ϕ_2 that extends both Δ_2 and ϕ_1 such that $\mathsf{dom}(\Delta_2) = \mathsf{dom}(\phi_2)$.*

Proof. By rule induction on the derivation of $\Delta_1[\phi_1] \vdash T[\phi_1] <: U[\phi_1]$. □

Theorem 5.17 (Completeness of Algorithmic Typing)**.** *Let ϕ_1 and Δ_1' be index contexts that extend Δ_1 such that $\mathsf{dom}(\Delta_1) \subseteq \mathsf{dom}(\Delta_1')$ and $\mathsf{dom}(\Delta_1') = \mathsf{dom}(\phi_1)$.*

1. *If $\Delta_1[\phi_1] \vdash T[\phi_1] : \star$ and $\Delta_1[\phi_1]; \Gamma[\phi_1] \vdash r : T[\phi_1] \dashv \Delta_1'[\phi_1]$, then $\Delta_1; \Gamma[\Delta_1] \vdash r : T[\Delta_1] \dashv \Delta_2$ and there exists ϕ_2 that extends both Δ_2 and ϕ_1 such that $\mathsf{dom}(\Delta_2) = \mathsf{dom}(\phi_2)$.*

2. *If $\Delta_1[\phi_1] \vdash T[\phi_1] : \star$ and $\Delta_1[\phi_1]; \Gamma_1[\phi_1] * r_1 \vdash t : T[\phi_1] \dashv \Delta_1'[\phi_1]; \Gamma_2[\phi_1] * r_2$, then depending on t either $\Delta_1; \Gamma_1[\Delta_1] \vdash t \uparrow T[\Delta_1] \dashv \Delta_2; \Gamma_2[\Delta_1]$ or $\Delta_1; \Gamma_1[\Delta_1] \vdash t \downarrow T[\Delta_1] \dashv \Delta_2; \Gamma_2[\Delta_1]$ and there exists ϕ_2 that extends both Δ_2 and ϕ_1 such that $\mathsf{dom}(\Delta_2) = \mathsf{dom}(\phi_2)$.*

Proof. By rule induction on the derivation of the given typing judgement, with part 2 using part 1. □

5.3 Implementation

We conclude this chapter with a brief description of the prototype implementation.

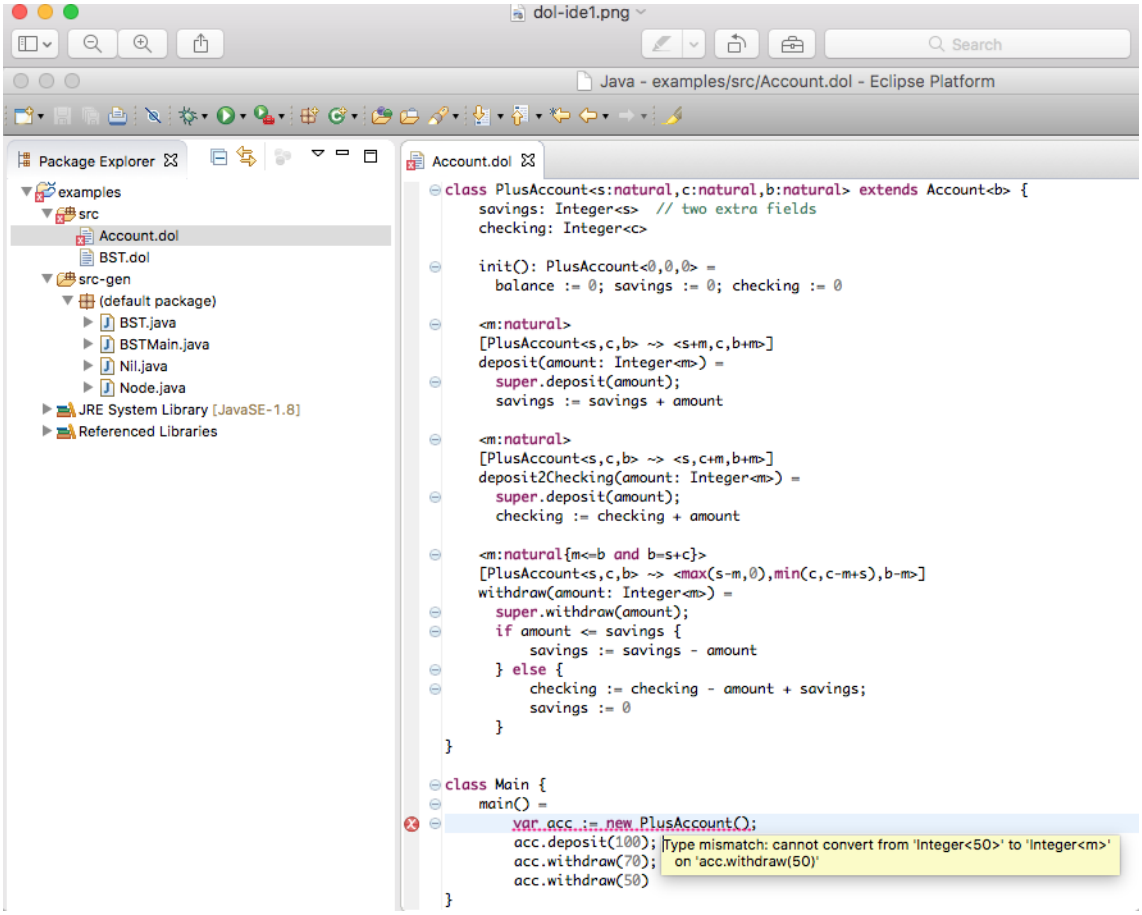


Figure 5.9: Two examples of error reporting in DOL

5.3.1 DOL IDE

Although development tools are popular in the context of object-oriented languages, their use for dependently-typed languages is still new. Our prototype ships with an IDE (Integrated Development Environment) developed as an Eclipse plugin based on the Xtext [2017] framework. The IDE includes: a code editor assistant for DOL programs, on-the-fly error checking, and target code generation in the form of of readable and pretty-printed Java classes.

Our typechecker is a direct implementation of the algorithmic rules given in this chapter, extended with integer and boolean literals, local variables and all the syntactic sugar from the examples. It is developed in Xtend [2017], a flexible and expressive dialect of Java. It uses a direct interface to Z3 [de Moura and Bjørner, 2008] via its API for Java. Constraint checking is performed as part of typechecking (in all the places where $\models$ judgements occur), and is completely transparent to a programmer of DOL. The decidability of our algorithmic system is dependent on the decidability of the solver. The use of the IDE is evidence that DOL programs typecheck quickly: we run the typechecker on the fly, i.e. whenever the program is modified, which leads us to believe that in the decidable domain of integer inequalities our typechecking algorithm is efficient.

Eclipse interactively highlights not only syntax errors while the programmer is typing but also compile time errors. We give two examples in Figure 5.9 that show the IDE underlining the problematic text in the editor. The prototype can be found at `http://rss.di.fc.ul.pt/tools/dol/`.

5.3.2 Local Variables

In the implementation, we include local variables omitted from the formal language (Chapter 3). These are declared by assignment. We therefore eliminate the extra burden of having to explicitly declare these variables as required by most statically typed languages. For example, we create a local variable by writing

```
var acc := new Account()
```

Variable acc is declared by the initialisation from **new** Account(). As we have seen, DOL's type system allows instance variables (fields) to have different types at different points in the program. We implement the same behaviour for local variables, allowing acc to have many types as long as they are instances of the class family Account.

5.3.3 Constraint Solving

Constraint solving lies at the heart of DOL's typechecker, which verifies programs in a fully automatic manner, without generating or requiring additional proofs; all constraints are issued exclusively by types. The typechecker relies on Z3 to determine if each and every constraint is valid under a list of assumptions corresponding to some Δ. It does this by taking the negation of the constraint and passing it to Z3 under the list of satisfiable assumptions. If the solver returns sat, then it found a counter-example, and the program will be rejected. On the other hand, if Z3 returns unsat, then the constraint is valid. For example, take the signature of method withdraw (Figure 2.1):

```
class Account⟨b:natural⟩ {
  ...
  ⟨m:natural{m≤b}⟩
  [Account⟨b⟩ ⇝ ⟨b−m⟩]
  withdraw(amount: Integer⟨m⟩) =
  ...
}
```

Now, consider the following client code:

```
var acc := new Account();
acc.deposit(100);
acc.withdraw(70);
acc.withdraw(30)
```

To check the last line, the typechecker calls the Z3 API for Java by providing the declarations and formulas issued by types, which internally are translated into the appropriate form. For the above example, the list of the typechecker provided formulas and declarations is equivalent to the following script in the SMT-LIB 2.0 standard format:

```
(declare-const b Int)
(declare-const m Int)
(assert (≥ b 0))
(assert (and (≥ m 0) (≤ m b)))
(assert (= b (- (+ 0 100) 70)))
(assert (not (and (≥ 30 0) (≤ 30 b))))
(check-sat)
```

5.3.4 Error Reporting

Any compiler should catch errors as soon as they occur and provide meaningful messages for them. The Xtext framework help us with the former. However, two features make it difficult to deliver the latter: index constraints and union types.

Types may contain fresh existential index variables internally, which complicate the generation of comprehensible explanations during typechecking. However, we maintain a map from existential index variables to the location where they were created. For example, suppose that we call method acc.withdraw(30) as in the snippet above. We generate fresh existential index variables for both b and m and put them in a map with the term acc.withdraw(30). This will enable the compiler to report both the source of error as well as its location. Then, we check if the existential index variables can be replaced by the actual values (30 in the example). If typechecking fails, say, by calling acc.withdraw(40) instead, we can report the line and the invalid constraint ($40 \leq 30$ obtained by replacing the existential index variables with the actual values).

In checking against a union type of the form T + U, we check first against T and if, that fails, we check against U. Therefore, the compiler reports an error when both types fail, indicating the source and location of the failure for T and for U.

Even with some space for improvement, the compiler seems to provide relatively accurate and localised error reporting to which contributes both the quantifier instantiation strategy [Dunfield and Krishnaswami, 2013, 2016] and the bidirectional typechecking technique [Pierce and Turner, 2000].

Chapter 6

Related Work

In this chapter, we present the most important work on dependent types and closely related approaches in the context of different programming languages, providing an overview of the state of the art in this area of research, and comparing with the contributions of DOL. Moreover, we provide references to alternative solutions to some of the problems we have encountered during the design of DOL.

Chapter Outline. This chapter consists of the following sections:

- Section 6.1 discusses different approaches to dependent types.
- Section 6.2 presents other solutions to program verification.
- Section 6.3 analyses work on linear types.
- Section 6.4 points to alternative formulations to a linear type discipline.

6.1 Dependent Types

Index refinements have their origins in the notion of dependent type developed by Martin-Löf [1984], and first applied to proof assistants (logical frameworks), the most important of which are AUTOMATH [van Daalen, 1980], the Calculus of Constructions [Coquand and Huet, 1988], NuPRL [Constable et al., 1986], Lego [Zhaohui and Pollack, 1992] and the Edinburgh Logical Framework [Harper et al., 1993]. According to the original formulation, dependent products and sums of the form $\Pi x : U.T$ and $\Sigma x : U.T$ are types indexed over the collection of terms x of arbitrary types U, unlike index refinements that restrict the domain of the argument. The price, however, is increased complexity of typechecking. When added to (possibly nonterminating) programming languages, the

task of determining type equivalence becomes as difficult as determining term equivalence (which is undecidable in general).

6.1.1 Full-spectrum Dependent Types

Some programming languages have managed to integrate full dependency by providing different strategies to handle nonterminating programs. Cayenne [Augustsson, 1998] is a functional programming language in the style of Haskell with an undecidable dependent type system. A semi-decidable implementation is provided that forces the typechecker to terminate within a number of prescribed steps, eventually providing the user with an answer. In practice, the Cayenne approach requires that the programmer anticipates if an equality will be resolved within a certain number of reductions.

Epigram [McBride, 2004] builds on a tactic-driven proof engine, similar to that of the Coq proof assistant [The Coq development team], requiring correctness proofs to be specified. Unlike Cayenne, Epigram rules out general recursive programs, avoiding nontermination and any form of effects, thus making typechecking decidable. Recursion is supported by the structure of dependent types which are inductive families with inductive indices. Agda [Norell, 2007] is an extension of Martin-Löf's intuitionistic type theory in the spirit of Epigram. It supports the definition of inductive and inductive-recursive data types and families, and it offers a powerful mechanism for dependently typed pattern matching. The language requires all programs to be total, i.e. all possible patterns must be matched, which is ensured by the termination checker. These languages rely on propositions-as-types; proofs, oftentimes complex, must be provided by programmers. In this sense, DOL offers a more high-level programming style. As DOL, these languages support programming with indexed types.

6.1.2 Domain-specific Logics

Dependent types have proved quite useful for enriching host languages with domain-specific logic. The Ynot tool is an extension of the functional dependently-typed language included in Coq providing support for side-effects via Hoare Type Theory (HTT) and Separation Logic [Nanevski et al., 2008b,a]. HTT introduces an indexed monadic type in the style of a Hoare triple to reason about mutation. While DOL's varying types may have similarities with the Hoare type, our approach does not involve the complexity of higher-order abstraction. As HTT, the F* language [Swamy et al., 2016], designed for program verification, employs the monad technique generalising it to multiple monads. This ML-style functional language uses dependent and refinement types to specify effectful programs, and supports automated and interactive proofs. A related approach

is provided by RSP1 [Westbrook et al., 2005] that allows programming with proofs in an imperative setting. The language offers decidable typechecking by banning impure operations from types with the purpose of letting the user prove arbitrary properties of programs. All these languages provide SMT-based automation and handle effectful programming. In that regard, they are close to DOL, yet they differ substantially in their aim to combine programming and theorem proving, which our language does not support.

Targeting the C programming language, Deputy [Condit et al., 2007] also handles mutation using a Hoare-inspired typing rule ensuring that assignment results in a well-typed state. For decidability, Deputy combines compile time and runtime checking, as opposed to our approach in which typechecking is performed statically, relying on the assistance of an external SMT solver.

6.1.3 Languages with Phase Separation

The idea of refinement types proposed by Freeman and Pfenning [1991] as an extension to ML with union and intersection types is a precursor of index refinements. The approach was further developed into a weaker, decidable version of dependent types: index refinements as formulated by Xi and Pfenning [1999] reduce typechecking to a constraint satisfaction problem on terms belonging to index sorts. Their approach (which we build on) offers the additional advantage of relative simplicity of the type system, as well as requiring fewer annotations, when compared to full dependent type systems. Xi later formulated Xanadu [Xi, 2000], a language with a C-like syntax combining imperative programming with index refinements, and ATS [Xi, 2004] which also supports DML-style dependent types. While closely related, DOL extends the ideas of Xanadu to class-based objects that exhibit state and behaviour. Our language also handles object-oriented programming features such as modular development, inheritance with subtyping, which Xanadu does not deal with. A proposal for building an object-oriented system on top of DML was also formulated by Xi [2002]. The language includes inheritance without subtyping – the relation is simulated via existentially quantified dependent types. Xi's object model is simpler than ours, since objects are not regarded as records of fields (they merely respond to messages), and the language does not include imperative features.

Ωmega [Sheard and Linger, 2007] and Liquid Types [Rondon et al., 2008] offer two more examples of functional languages with a strict phase separation; the latter is implemented in DSolve, a tool that automatically infers dependent types from an OCaml program and a set of logical qualifiers. Cyclone [Jim et al., 2002] is a type-safe extension of the C programming language, combining static analysis and runtime checks. It offers domain-specific indexed types, but for the purpose of safe multi-threading and memory

management.

Another reference is Dependent JavaScript (DJS) [Chugh et al., 2012], which introduces refinement types with predicates from an SMT-decidable logic in a dynamic real-world language. In DJS, imperative updates involve the presence of mutation: the types of variables are changed by assignment, for instance, as in DOL. The challenge is handled using *flow-sensitive heap types*, which allow tracking variable types, in combination with refinement types. The result is an increase in the language expressiveness by using type annotations inside JavaScript comments that account for side-effects. DJS employs the *alias types* approach [Smith et al., 2000] for strong updates in combination with thawing/freezing locations. A similar formulation could be used as a powerful alternative to the simple linear treatment that we include in DOL.

6.1.4 Other Forms of Dependent Types

Other forms of dependent types include X10's constrained types [Nystrom et al., 2008], designed around the notion of constraints on the *immutable* state of objects. The core language proposed extends the purely functional FJ [Igarashi et al., 2001]. While appealing, constrained types cannot enforce invariants on mutable state. Another approach in the object-oriented setting is provided by dependent classes [Gasiunas et al., 2007]. A class can be seen as forming a family of collaborating objects, much like a type family in traditional dependent type theory. The model is complex, since it also involves inheritance, and type soundness is hard to prove. Like DOL, dependent classes and its lightweight version [Kamina and Tamai, 2008] support class-based programming and inheritance. A similar model is provided by Scala's path-dependent types, with a type-theoretic foundation in the DOT calculus [Amin, 2016, Rompf and Amin, 2016], that unify nominal and structural type systems by allowing objects to contain type members. Dependent types in this model are expressed not in type signatures but in type placements. An abstract type refers to a type that must be defined by subclasses, becoming dependent on the instance it refers to. Unlike DOL, none of these languages supports an imperative style of programming, whereas DOL is designed to handle mutable objects.

6.2 Other Approaches to Program Verification

Other approaches include advanced techniques for verifying software properties. Assertions are one of the most useful mechanisms, also serving for documenting code. Any boolean predicate can be used to check software properties, which makes statically checking assertions undecidable in general. The pre- and post-types in DOL can be seen as a

(decidable) alternative to a pair of assertions.

The extended static checker for JML (ESC/Java) [Leino, 2001] and related systems are effective in finding bugs in Java programs. ESC provides a simple language of annotations and uses an underlying automatic theorem prover to reason about program behaviour and to verify the absence of certain kinds of errors, such as array-out-of-bounds and null pointer dereference. The tool is based on an approach which is neither sound nor complete. However, by combining a range of techniques, it is more expressive than index refinements.

Spec♯ [Barnett et al., 2005] uses a sound programming methodology to check C♯ programs, which allows reasoning about object invariants even in a multi-threaded setting. Many of the ideas described by Müller [2002], Barnett et al. [2004], Müller et al. [2006], Summers and Drossopoulou [2010], Balzer and Gross [2011] for the verification of invariants for objects have been implemented in Spec♯.

A vast number of tools provide static program verifiers for object-oriented languages relying on various external theorem provers to discharge verification conditions. Boogie [Barnett et al., 2006] is one such tool meant for Spec♯ programs. In the same spirit, Why3 [Filliâtre and Paskevich, 2013] uses WhyML, a first-order specification and programming language, as an intermediate language for the verification of C, Java and Ada programs. This is somewhat different from other verification systems where a general purpose language is usually equipped with a specification language. These systems can verify invariants, track mutable references, aliases and side effects statically. While formal verification is more expressive than dependent types, it is also more complex, and still too costly for mainstream adoption. Language-based verification methods, such as the one we propose in DOL via types, are closer to existing programming methodologies. The main benefit of our approach is to provide lightweight verification without requiring prior training in logic or theorem proving.

A number of techniques also introduces separation logic [Reynolds, 2002], a substructural logic that augments the type system with the ability to control the number and order of uses of data structures, and has proved useful for object-oriented languages in the presence of shared mutable state [Parkinson and Bierman, 2013]. Some of these advanced techniques for the verification of objects could in principle be used to enrich our language.

6.3 Affine Types

In order to demonstrate how dependently typed objects would work in practice in a high-level object-oriented programming language, we have taken a simple approach to alias

control that extends the work of Gay et al. [2015]: we add a linear discipline through which instances of type varying classes cannot be duplicated, while allowing instances of type invariant classes to be handled without restrictions.

A more sophisticated mechanism to regulate the usage of resources seems orthogonal, but it can be found in Alms, a language in the style of OCaml with an affine type system, developed by Tov and Pucella [2011]. Alms is a language with general-purpose substructural type system that can express a variety of stateful type disciplines and, like DOL, also supports interacting with shared types. It has been implemented, and proved sound. Adapting this approach to our language would require encoding affine capabilities in DOL, rather than conveniently assigning a qualifier to a type based on the form of its class.

6.4 Ownership of Objects

Adding a better access control to DOL via a sophisticated alias analysis would also scale the current design up to a broader variety of practical programming examples. There is a vast literature on mechanisms that provide such a flexibility, namely Hogg [1991]'s Islands, Almeida [1997]'s balloon types and [Clarke et al., 1998]'s ownership types, which, differing substantially in technique, allow the use of linked data structures encapsulated within Islands and Balloons, while providing non-aliasing guarantees to the rest of the systems. Other works include Fähndrich and DeLine [2002]'s adoption and focus, Aiken et al. [2003]'s restrict and confine and Morrisett et al. [2005]'s freeze, thaw and refreeze in the $\mathbf{L}^3$ language, influenced by Smith et al. [2000]'s Alias Types. The freeze/thaw/refreeze approach is similar in role to both the adoption/focus and the restrict /confine mechanisms in the sense that they introduce some notion of linearity and a way to express aliasing invariants.

While these solutions are not implemented in DOL, they should be valuable for future work towards making the language more suitable for production use. What we need is a property for performing the controlled duplication of references to mutable objects. The process that allows "thawing" a reference, so that it can be strongly updated, and then refreezing it, once the original type has been restored, may allow reasoning about aliasing patterns and should integrate well with DOL's type system.

Chapter 7

Conclusion

This thesis describes DOL, a programming language with a restricted form of dependent types, designed to support the verification of class-based object-oriented programs featuring mutable objects and inheritance with subtyping.

We have formalised the syntax, type system and operational semantics (Chapter 3). We have proved that the system enjoys a key beneficial property, type soundness via subject reduction and progress (Chapter 4). We have also given a set of algorithmic typing rules, ready for implementation, and proved that the algorithmic system is sound and complete with respect to the declarative system (Chapter 5).

The design of DOL combines the scalability and modularity of object orientation with the safety provided by dependent types, sometimes blurring the lines of programming paradigms:

1. DOL uses the style of generic programming of Java-like languages to introduce dependent types through indexed classes – families of classes that can have many types – so as to simplify object-oriented programming with dependent types;

2. The approach, based on index refinements, renders the type system decidable, provided indices are drawn from some decidable theory, while still being able to express interesting properties of programs;

3. Features of the functional paradigm are present when they need to be, namely by the use of a null-free style of programming, even though programs in DOL are typically structured using mutable objects and class-based inheritance;

4. The type system tracks changes to type varying objects and enforces a linear type discipline without the need for awkward qualifier annotations.

These language features together form a coherent and natural design for a safer object-oriented language, where the effort of programming with dependent types is to come up

with the right types, as evidenced by the technically challenging example of a binary search tree implemented in imperative style (Chapter 2).

In addition, the *proof-of-concept prototype* attests the relevance of the language described in this thesis. It includes a plugin for the Eclipse IDE, a development tool that is widely used in the context of object-oriented languages but still new for dependently-typed languages.

However, much work remains to be done. Towards a more effective object-oriented language with dependent types, suitable for production use, the main topics to study are the integration of richer index languages in domains of interest, possibly at the cost of decidable typechecking, and alternatives to the current strategy for handling aliases.

Bibliography

Alex Aiken, Jeffrey S. Foster, John Kodumal, and Tachio Terauchi. Checking and inferring local non-aliasing. In *PLDI*, pages 129–140. ACM Press, 2003.

Paulo Sérgio Almeida. Balloon types: Controlling sharing of state in data types. In *ECOOP*, volume 1251 of *LNCS*, pages 32–59. Springer, 1997.

Nada Amin. *Dependent Object Types*. PhD thesis, École Polytechnique Fédérale de Lausanne, 2016.

Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. Behavioral types in programming languages. *Foundations and Trends® in Programming Languages*, 3(2-3):95–230, 2016.

David Aspinall and Adriana B. Compagnoni. Subtyping dependent types. In *IEEE Press*, pages 86–97, 1996.

Lennart Augustsson. Cayenne a language with dependent types. In *ICFP*, pages 239–250. ACM Press, 1998.

Stephanie Balzer and Thomas R. Gross. Verifying multi-object invariants with relationships. In *ECOOP*, pages 358–382. Springer, 2011.

Hendrik Pieter Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.

Michael Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure and Interoperable Smart devices*, pages 49–69, 2005.

Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Conference on Formal Methods for Components and Objects*, pages 364–387. Springer, 2006.

Gavin M. Bierman, Erik Meijer, and Mads Torgersen. Lost in translation: Formalizing proposed extensions to C#. In *OOPSLA*, pages 479–498. ACM Press, 2007.

Gavin M. Bierman, Andrew D. Gordon, Cătălin Hriţcu, and David Langworthy. Semantic subtyping with an SMT solver. In *ICFP*, pages 105–116. ACM Press, 2010.

Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda – A functional language with dependent types. In *TPHOLs*, volume 5674 of *LNCS*, pages 73–78. Springer, 2009.

Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, 2013.

Joana Campos and Vasco T. Vasconcelos. Imperative objects with dependent types. In *Formal Techniques for Java-like Programs*, pages 2:1–2:6, 2015.

Joana Campos and Vasco T. Vasconcelos. Programming with mutable objects and dependent types. In *INForum*. Atas do Oitavo Simpósio de Informática, 2016.

Patrice Chalin and Perry R. James. Non-null references by default in Java: Alleviating the nullity annotation burden. In *ECOOP*, pages 227–247. Springer, 2007.

Ravi Chugh, David Herman, and Ranjit Jhala. Dependent types for JavaScript. In *OOPSLA*, pages 587–606. ACM Press, 2012.

Maciej Cielecki, Jędrzej Fulara, Krzysztof Jakubczyk, and Lukasz Jancewicz. Propagation of jml non-null annotations in Java programs. In *Principles and Practice of Programming in Java*, pages 135–140. ACM Press, 2006.

David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA*, pages 48–64. ACM Press, 1998.

Jeremy Condit, Matthew Harren, Zachary R. Anderson, David Gay, and George C. Necula. Dependent types for low-level programming. In *ESOP*, volume 4421 of *LNCS*, pages 520–535. Springer, 2007.

Robert L. Constable, Stuart F. Allen, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, Scott F. Smith, James T. Sasaki, and S. F. Smith. *Implementing mathematics with the Nuprl proof development system*. Prentice Hall, 1986.

Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Inf. Comput.*, 76 (2/3):95–120, 1988.

Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. In *International Conference on Software Engineering*, pages 258–267, 1996.

Joshua Dunfield. *A Unified System of Type Refinements*. PhD thesis, Carnegie Mellon University, 2007. CMU-CS-07-129.

Joshua Dunfield and Neelakantan R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *ICFP*, pages 429–442. ACM Press, 2013.

Joshua Dunfield and Neelakantan R. Krishnaswami. Sound and complete bidirectional typechecking for higher-rank polymorphism with existentials and indexed types. *CoRR*, abs/1601.05106, 2016.

Joshua Dunfield and Frank Pfenning. Type assignment for intersections and unions in call-by-value languages. In *Foundations of Software Science and Computational Structures*, volume 2620 of *LNCS*, pages 250–266. Springer, 2003.

Eclipse. Eclipse IDE, 2017. `http://www.eclipse.org/`.

Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *PLDI*, pages 13–24. ACM Press, 2002.

Manuel Fähndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. In *OOPSLA*, pages 302–312. ACM Press, 2003.

Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - where programs meet provers. In *ESOP*, volume 7792 of *LNCS*, pages 125–128. Springer, 2013.

Cormac Flanagan. Hybrid type checking. In *POPL*, pages 245–256, 2006.

Tim Freeman and Frank Pfenning. Refinement types for ML. In *PLDI*, pages 268–277. ACM Press, 1991.

Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. Dependent classes. In *OOPSLA*, pages 133–152. ACM Press, 2007.

Simon J. Gay, Nils Gesbert, António Ravara, and Vasco Thudichum Vasconcelos. Modular session types for objects. *Logical Methods in Computer Science*, 11(4), 2015.

Andrew D. Gordon and Cédric Fournet. Principles and applications of refinement types. In *Logics and Languages for Reliability and Security*, pages 73–104. IOS Press, 2010.

Robert Harper, Furio Honsell, and Gordon D. Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, 1993.

Tony Hoare. Null references: The billion dollar mistake. QCon, 2009.

John Hogg. Islands: Aliasing protection in object-oriented languages. In *OOPSLA*, pages 271–285. ACM Press, 1991.

Atsushi Igarashi and Hideshi Nagira. Union types for object-oriented programming. In *Symposium on Applied Computing*, pages 1435–1441. ACM Press, 2006.

Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *TOPLAS*, 23(3):396–450, 2001.

Trevor Jim, J. Gregory Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX*, pages 275–288. USENIX, 2002.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Wirich, and Mark Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17(1): 1–82, 2007.

Tetsuo Kamina and Tetsuo Tamai. Lightweight dependent classes. In *GPCE*, pages 113–124. ACM Press, 2008.

Kenneth Knowles, Aaron Tomb, Jessica Gronski, S Freund, and Cormac Flanagan. Sage: Unified hybrid checking for first-class types, general refinement types and dynamic. Technical report, UCSC, Santa Cruz, CA, 2007.

Kenneth L. Knowles. *Executable Refinement Types*. PhD thesis, University of California, 2014.

Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *SIGSOFT Softw. Eng. Notes*, 31(3): 1–38, 2006.

K. Rustan M. Leino. Extended static checking: A ten-year perspective. In *Informatics – 10 Years Back. 10 Years Ahead*, volume 2000 of *LNCS*, pages 157–175. Springer, 2001.

Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *TOPLAS*, 16(6):1811–1841, 1994.

William Lovas and Frank Pfenning. A bidirectional refinement type system for LF. *Electron. Notes Theor. Comput. Sci.*, 196:113–128, 2008.

Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis-Napoli, 1984.

Conor McBride. Epigram: Practical programming with dependent types. In *Advanced Functional Programming*, volume 3622 of *LNCS*, pages 130–170. Springer, 2004.

Conor McBride. How to keep your neighbours in order. In *ICFP*, pages 297–309. ACM Press, 2014.

Greg Morrisett, Amal Ahmed, and Matthew Fluet. L3: A linear language with locations. In *Typed Labmda Calculi and Applications*, pages 293–307. Springer, 2005.

Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer, 2002.

Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular invariants for layered object structures. *Sci. Comput. Program.*, 62(3):253–286, 2006.

Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: dependent types for imperative programs. In *ICFP*, pages 229–240, 2008a.

Aleksandar Nanevski, J. Gregory Morrisett, and Lars Birkedal. Hoare type theory, polymorphism and separation. *Journal of Functional Programming*, 18(5-6):865–911, 2008b.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, 2007.

Nathaniel Nystrom, Vijay Saraswat, Jens Palsberg, and Christian Grothoff. Constrained types for object-oriented languages. In *OOPSLA*, pages 457–474. ACM Press, 2008.

Martin Odersky, Christoph Zenger, and Matthias Zenger. Colored local type inference. In *POPL*, pages 41–53. ACM Press, 2001.

Matthew J. Parkinson and Gavin M. Bierman. Separation logic for object-oriented programming. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of *LNCS*, pages 366–406. Springer, 2013.

Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, 2000.

John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Press, 2002.

Tiark Rompf and Nada Amin. Type soundness for dependent object types (dot). In *OOPSLA*, pages 624–641. ACM Press, 2016.

Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *PLDI*, pages 159–169. ACM Press, 2008.

Tim Sheard and Nathan Linger. Programming in Omega. In *Central European Functional Programming School*, volume 5161 of *LNCS*, pages 158–227. Springer, 2007.

Frederick Smith, David Walker, and J. Gregory Morrisett. Alias types. In *ESOP*, volume 1782 of *LNCS*, pages 366–381. Springer, 2000.

Alexander J. Summers and Sophia Drossopoulou. Considerate reasoning and the composite design pattern. In *Verification, Model Checking, and Abstract Interpretation*, volume 5944 of *LNCS*, pages 328–344. Springer, 2010.

Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *POPL*, pages 256–270. ACM Press, 2016.

The Coq development team. The Coq proof assistant reference manual, version 8.6, 2016. URL `https://coq.inria.fr/refman/`.

Jesse A. Tov and Riccardo Pucella. Practical affine types. In *POPL*, pages 447–458. ACM Press, 2011.

Diederik T. van Daalen. *The Language Theory of Automath*. PhD thesis, Technische Hogeschool Eindhoven,Eindhoven, 1980.

Edwin Westbrook, Aaron Stump, and Ian Wehrman. A language-based approach to functionally correct imperative programming. In *ICFP*, LNCS, pages 268–279. ACM Press, 2005.

Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, Pittsburgh, 1998.

Hongwei Xi. Imperative programming with dependent types. In *LICS*, pages 375–387. IEEE Press, 2000.

Hongwei Xi. Unifying object-oriented programming with typed functional programming. In *PEPM*, pages 117–125. ACM Press, 2002.

Hongwei Xi. Applied type system: Extended abstract. In *TYPES*, pages 394–408. Springer, 2004.

Hongwei Xi. Dependent ML: an approach to practical programming with dependent types. *Journal of Functional Programming*, 17(2):215–286, 2007.

Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *PLDI*, pages 249–257. ACM Press, 1998.

Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *POPL*, pages 214–227. ACM Press, 1999.

Xtend. Xtend programming language, 2017. `https://www.eclipse.org/xtend/`.

Xtext. Xtext framework, 2017. `http://eclipse.org/Xtext/`.

Luo Zhaohui and Robert Pollack. The LEGO proof development system: A user's manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, 1992.