Static Types for Dynamic Documents

Mark Brian Shields
B.Sc. (Hons), Melbourne University, 1996
B.Sc., Monash University, 1991

A dissertation submitted to the faculty of the
Oregon Graduate Institute of Science and Technology
in partial fulfilment of the
requirements for the degree
Doctor of Philosophy
in
Computer Science and Engineering

February 2001

Copyright 2001, Mark Brian Shields

The dissertation "Static Types for Dynamic Documents" by Mark Brian Shields has been examined and approved by the following Examination Committee:

Dr. John Launchbury
Professor
Thesis Advisor

Dr. Mark P. Jones
Associate Professor

Dr. David Maier
Professor

Dr. Philip Wadler
Avaya Labs

Acknowledgements

My sincere and heartfelt thanks to:

- John Launchbury, for his supervision, guidance, and, most of all, friendship. John's mature approach to research has been a great inspiration for me, and our conversations over the years have been consistently fun and stimulating.
- Simon Peyton Jones, also for his supervision, his uncanny knack of asking the right questions, and for teaching me the value of a good example.
- Erik Meijer, for encouraging me to look in this direction for research ideas when I was despondent over my other work.
- Tim Sheard, for explaining to me the subtleties of staged computation.
- Andrew Tolmach, for always keeping his head when all about him we were losing ours
- Jeff Lewis, for succeeding in pushing through the write-up boundary on implicit parameters when I failed.
- Mark Jones, for his insights on all things typed, for writing such a fine dissertation upon which this work is built, and for taking the time to examine this thesis.
- Philip Wadler, for showing that good computing science can make a difference in the bewildering world of real systems, and for his insightful examination of the thesis.
- David Maier, for thoroughly proof-reading a dissertation somewhat different from what was promised to him, and for teaching me some basic English grammar. All the remaining errors are because I didn't have time to follow fully his advice.
- Zino Benaissa, for helpful discussions on staging.
- Alex Aiken, for helpful discussions on type-indexed rows.
- Daan Leijen, for his boundless enthusiasm for language implementation.
- Harald Søndergaard, for encouraging me first to complete my honours degree, and then to further my studies overseas.
- Andy Moran, for his seemingly endless optimism and sense of humour, for always asking me whether I'd finished the thesis yet, and for his terrific friendship. I don't think I would have finished without him.
- Ruth Rowland, for her companionship, love, and quiet support during both good and miserable times.
- Magnus Carlsson, Koen Claessen, David Clarke, Byron Cook, Sigbjorn Finne, Andy Gill, Bill Harrison, Jim Hook, Dick Kieburtz, John Mathews, Laszlo Nemeth, Thomas Nordin, Johan Nordlander, Walid Taha and Keith Wansbrough, for their conversations and friendship over the years.
- The Oregon Graduate Institute, the University of Glasgow, the University of Utrecht, the British Board of Graduate Studies (Overseas Research Student award #96017029), the United States Air Force Air Materiel Command (contract #F19628-96-C-0161), and the National Science Foundation (grant CCR-9970980), for their financial support.
- Finally, my father, Brian Shields, for his life-long example of moral and intellectual integrity. I dedicate this thesis to him.

Contents

Acknowledgements iv
Abstract x

1 Introduction **1**
1.1 Outline of Thesis 9
1.2 How to read this dissertation 10

I Type-Indexed Rows **11**

2 Introduction **12**
2.1 Review: Label-Indexed Rows 13
2.2 From Label- to Type-Indexed Rows 16
2.3 Equality Constraints 17
2.4 Simplifying Constraints 19
2.5 Newtypes 20
2.6 Implementing Records 22
2.7 Implementing TIPs and TICs 24
2.8 Ambiguity 25
2.9 Satisfiability 26

3 Examples **30**
3.1 Tuples 30
3.2 Records Revisited 31
3.3 Recursive Datatypes 31
3.4 XML 32
3.5 Overloading 37

4 Type Checking **41**
4.1 Syntax 41

4.2 Well-typed Terms 45
4.3 Type Order 49
4.4 Constraint Entailment 53
4.4.1 Unification and Saturation 53
4.4.2 Entailment Judgement 56
4.4.3 Soundness of Entailment 56
4.4.4 (In)Completeness of Entailment 59
4.4.5 Complexity of Entailment 60
4.5 Type Soundness 60

5 Type Inference **65**
5.1 Inference Rules 65
5.2 Constraint Simplification 68
5.3 Correctness 72
5.4 Row Extension 78

6 Conclusions to Part I **81**
6.1 Related Work 81
6.2 Conclusions and Future Work 84

II Dynamically-Typed Staged Computation **85**

7 Introduction **86**
7.1 Staged Computation 88
7.2 Monomorphically Typed Staged Computation 89
7.3 Polymorphically Typed Staged Computation 92
7.4 Constrained Polymorphism and Staging 93
7.5 Dynamically Typed Staged Computation 95
7.6 Constrained Polymorphism and Dynamic Typing 97
7.7 The `rttype` and `liftable` Constraints 97

8 Examples **99**
8.1 Dynamic Typing 99
8.2 Partial Evaluation 102
8.3 Distributed Computing 109

9 Formal Development **114**
9.1 Syntax 114

9.2 Well-kinded Types 115
9.3 Constraint Entailment 117
9.3.1 Soundness of Entailment 119
9.4 Well-typed Terms 121
9.5 Denotational Semantics 125
9.5.1 Monads 126
9.5.2 Semantic Sets and Predomains 132
9.5.3 Denotation of Types 133
9.5.4 Denotation of Run-Time Terms 135
9.5.5 Type Soundness 139

10 Conclusions to Part II **142**
10.1 Related Work 142
10.2 Conclusions and Future Work 143

A Recognising XML Elements **145**

B Proofs for Chapter 4 **154**
B.1 Type Order 154
B.2 Unification 157
B.3 Entailment 160
B.4 Type Soundness 176

C Proofs for Chapter 5 **197**
C.1 Simplifier Correctness 197
C.2 Soundness of Type Inference 208

D Proofs for Chapter 9 **216**
D.1 Entailment 216
D.2 Type Soundness 217

Biographical Sketch 261

List of Figures

3.1 Some (type specialised) standard library functions 35

4.1 Syntax of λ^{TIR} kinds, types and terms 42
4.2 λ^{TIR} type and term constructors 42
4.3 Initial λ^{TIR} type var context Δ_{init} 42
4.4 Well-kinded λ^{TIR} types, constraints, type schemes and type contexts 43
4.5 Definitions of functions *named*, *names*, *anon*, *inheritable*, *norm*, *eqs*, *inss* and *inhs* 44
4.6 Syntax of λ^{TIR} run-time terms 45
4.7 Initial λ^{TIR} type context Γ_{init} 45
4.8 Well-typed λ^{TIR} terms 46
4.9 Well-typed λ^{TIR} pattern abstractions 47
4.10 Total order on λ^{TIR} monotypes 49
4.11 Partial ordering on normalized λ^{TIR} types of kind `Type` and `Row` 51
4.12 Definition of *fv*, *mgus*, and *saturate* 54
4.13 λ^{TIR} constraint entailment 55
4.14 Definition of the set $\mathcal{I}$, the denotation of λ^{TIR} witnesses in $\mathcal{I}$, the denotation of λ^{TIR} primitive constrants as subsets of $\mathcal{I}$, and *env* 57
4.15 Evaluation monad $\mathbf{E}$ 60
4.16 Denotation of λ^{TIR} normalized monotypes and type schemes as ideals of $\mathbf{E}\,\mathcal{V}$ 61
4.17 Denotation of λ^{TIR} run-time terms as members of $\mathbf{E}\,\mathcal{V}$ (part 1 of 2) 63
4.18 Denotation of λ^{TIR} run-time terms as members of $\mathbf{E}\,\mathcal{V}$ (part 2 of 2) 64

5.1 Type inference and translation for λ^{TIR} terms 66
5.2 Type inference and translation for λ^{TIR} patterns 67
5.3 Definition of *notIn* 68
5.4 Simplification of λ^{TIR} constraints (part 1 of 2) 69
5.5 Simplification of λ^{TIR} constraints (part 2 of 2) 70

5.6 The logical relation $=$ on $\mathbf{E}\,\mathcal{V} \times \mathbf{E}\,\mathcal{V}$ indexed by λ^{TIR} monotypes of kind Type 77

8.1 Signatures for operations on sets and relations 104

9.1 Syntax of λ^{SC} source types and terms 115
9.2 Syntax of λ^{SC} run-time terms 116
9.3 Well-kinded λ^{SC} types, type schemes and constraints 117
9.4 Entailment of λ^{SC} constraints 118
9.5 Denotation of λ^{SC} witnesses into $\mathcal{T}$, and the definition of *env* 119
9.6 Denotation of λ^{SC} ground primitive constraints as subsets of $\mathcal{T}$ 119
9.7 Types for λ^{SC} constants in Γ_{init} 120
9.8 Well-typed λ^{SC} stage 0 terms 122
9.9 Well-typed λ^{SC} stage $n+1$ terms (part 1 of 2) 123
9.10 Well-typed λ^{SC} stage $n+1$ terms (part 2 of 2) 124
9.11 Evaluation monad $\mathbf{E}$ 127
9.12 Reader monad $\mathbf{D}\,E$ 128
9.13 Renaming monad $\mathbf{R}$ 128
9.14 Name supply monad $\mathbf{M}$ 129
9.15 Name supply and renaming monad $\mathbf{N}$ 129
9.16 I/O monad $\mathbf{IO}$ 130
9.17 Name supply and I/O monad $\mathbf{MIO}$ 131
9.18 The semantic sets $\mathcal{Z}$ and $\mathcal{D}$, and the predomain $\mathcal{V}$ 132
9.19 Denotation of λ^{SC} types as ideals of $\mathbf{E}\,\mathcal{V}$ 134
9.20 Denotation of λ^{SC} stage $n+1$ terms 136
9.21 Denotation of λ^{SC} stage 0 pure terms 137
9.22 Denotation of λ^{SC} stage 0 monadic terms 138
9.23 Denotation of λ^{SC} stage 0 constants 139

A.1 Extensions to λ^{TIR} types, terms and patterns for handling XML elements, and syntax for recogniser components 146
A.2 Executing a sequence of recogniser actions upon a stack of λ^{TIR} run-time terms147
A.3 Building a recogniser from a λ^{TIR} type (part 1 of 3) 148
A.4 Building a recogniser from a λ^{TIR} type (part 2 of 3) 149
A.5 Building a recogniser from a λ^{TIR} type (part 3 of 3) 150
A.6 Converting a recogniser to use **start** 151
A.7 Extensions to λ^{TIR} type inference rules to recognise and convert XML elements to λ^{TIR} run-time terms 152

Abstract

Static Types for Dynamic Documents
Mark Brian Shields

Ph.D., Oregon Graduate Institute of Science and Technology
February 2001

Thesis Advisor: Dr. John Launchbury

Dynamic, active documents are particularly troublesome to program within conventional languages. Documents are typically represented in XML or HTML, which use regular-expression like types instead of the familiar sums-of-products datatypes supported by conventional languages. Furthermore, documents tend to include embedded programs in a variety of scripting languages, for which conventional languages offer no support at all. It is thus very difficult to verify that these programs generate even syntactically well-formed documents, let alone documents which are valid for their document type definition, and contain only well-typed scripts.

This thesis develops the core type system for a Haskell-like functional programming language that directly supports dynamic, active documents. The first part presents a system of *type-indexed rows*, that supports many aspects of XML's regular-expression types without abandoning the type features which make functional programming attractive. In particular, type-indexed rows coexist cleanly with higher-order types and parametric polymorphism. The second part presents a system of *staged computation*, that allows server-side and client-side code to be cleanly separated.

In both cases, the type system can guarantee that only well-formed and valid documents are generated. Hence, not only are document-generating programs easier to write using these systems, in addition they are much more likely to be correct.

Any system that allows no criteria other than those arbitrarily chosen as the basis of the system itself can be called a terrorist system.

Georges Perec

Chapter 1

Introduction

The adoption of a standard document description language, HTML [91], was essential to the early success of the world-wide-web. HTML provides a small, fixed, and reasonably simple set of primitive datatypes for describing both the structure and typographic layout of a document. Motivated by the popularity of on-line services, interest has since grown in using the web's mechanism to distribute data of any type, independently of its typographic representation. To this end, XML [12], an evolution of SGML [45], has been adopted as a standard language for documents representing first-order data. Unlike HTML, XML documents may define their own datatypes within the document itself. Hence XML is an "extensible" markup language.

XML

Though syntactically baroque, XML is built upon a simple model of tree-structured data. Documents may contain both a regular-tree grammar (termed a *document type definition*, or DTD) and a labelled-tree (termed an *element*), such that the tree is recognised by the grammar. For example, the following document, in slightly idealised syntax, describes a grammar of e-mail messages and a single message:

```
element Msg = (((To|Bcc)* & From), Body)
element To = String
element Bcc = String
element From = String
element Body = P*
element P = String

<Msg>
  <From>mbs@cse.ogi.edu</From>
  <To>jl@cse.ogi.edu</To>
  <Bcc>mbs@cse.ogi.edu</Bcc>
  <Body>
    <P>The thesis is almost finished.</P>
    <P>All that's needed is an example for the introduction.</P>
  </Body>
</Msg>
```

Each grammar production (termed an *element type declaration*) has a distinct left-hand side non-terminal (termed a *tag name*), and implicitly generates a single tree labelled by the non-terminal. Production right-hand sides are regular expressions built from the following

eight operators:

`String`	"parsed character data", or string
`A`	sub-tree
r `*`	list of r's
r `+`	non-empty list of r's
r `?`	optional r
(r_1 , ..., r_n)	tuple: all of r_1, r_2, etc, in that order
(r_1 | ... | r_n)	"choice," or union: one of r_1, r_2, etc
(r_1 & ... & r_n)	"unordered tuple": all of r_1, r_2, etc, in any order

(The & operator does not appear in XML, but is in SGML [45] and, abstractly, in XML Schema [24].)

A tree is a sequence of sub-trees and primitive strings delimited by matching tag names. Deciding whether a tree is recognised by the regular tree grammar is called *document validation*. Its easy to check the above example tree is recognised by its grammar. By comparison, the following tree is not valid:

```
<Msg>
  <Body/>
  <From>mbs@cse.ogi.edu</From>
</Msg>
```

(Here `<Body/>` is sugar for `<Body></Body>`). A `Body` sub-tree cannot appear before a `From` sub-tree within the children of a `Msg` tree.

Note that there are very few constraints on the form of regular expressions. In particular, choices and unordered tuples are anonymous, may appear deeply nested within other expressions, and may reuse the same tag name.

From static to dynamic, active documents

Though XML captures the notion of a *static document*, most documents are in fact *dynamic*. On-line services typically generate documents on-the-fly in response to an ongoing user dialogue, using a mixture of databases, live information feeds and user-supplied data. Furthermore, because XML documents have no inherent typographic representation, they must be further transformed, often by the client, before being rendered.

To further complicate matters, documents, particularly HTML documents, tend to contain embedded scripts which are to be executed by the client rather than the server. We call these *active* documents. Scripts are written in a variety of languages, and are represented as uninterpreted strings.

How should a server program be implemented to generate dynamic, active documents?

XML and the next 700 programming languages

Of course almost any language can be used to manipulate XML. This manipulation can be done at a concrete level by generating and concatenating strings containing XML and scripting language fragments, for which Perl [110] is a popular choice. Less error-prone is

to use a library to manipulate XML in abstract form. For example, JavaServer Pages [82] is a sophisticated library for Java [34] programs which implement on-line services. However, these approaches tend to be syntactically awkward, and cannot guarantee that only valid XML is generated.

Hence many custom *domain-specific languages* have been developed to generate, filter and transform XML documents, including:

- CSS [58] and XSL [3, 18] for applying typographic styling and other transformations.
- XML Query [25, 26] for filtering and generating XML using tree-structured query operators.
- `<BigWig>` [96] and Compaq's Web Language [60] for specifying all aspects of an on-line service within a single typed program.
- A plethora of untyped, ad-hoc scripting languages which extend XML with "active" tags denoting common control structures. For example: XML Script [22], XEXPR [74], XFA [116] and XPL [15].

This situation is unfortunate. Other than their common use of XML, these languages share little common syntax and have no unified semantics. There is much overlap in functionality, and little or no support for abstraction and extensibility, suggesting that even more languages will arise as XML finds new applications.

Similarly, a number of domain-specific scripting languages have been developed for use within active documents, including Java [34] and JavaScript [29]. Again there is no agreement on syntax, type system (if any) or semantics.

Functional programming and the next 700 programming languages

An old [53] and well-tested idea in functional programming is to embed domain-specific languages (DSLs) as *combinator libraries* within a single functional programming language. We refer the reader to the work of Hudak [41] and Swierstra *et al.* [103] for an overview of this methodology. Examples from the literature include:

- Reactive animation [23]
- Graphical user interfaces [27]
- Computer music [42]
- Pretty printing [44]
- Typesetting [52]
- Database querying [55]
- Hardware description [63]
- CGI scripting [64]
- Robot control [83]
- Financial modelling [84]
- Computer vision [92]
- Parsing [102]

This approach has many advantages over developing a DSL from scratch:

- DSLs may be readily combined because they are simply libraries in a common language.

- Because the underlying functional programming language has a relatively simple equational theory, it is often quite feasible to verify formally static properties of DSL programs.
- Furthermore, with a little cunning, the functional programming language's type system may often be exploited to verify statically the well-typing of DSL programs.
- The DSL designer may reuse the already extensive intellectual investment which has gone into functional programming languages, and is thus less likely to make fundamentally poor design decisions. Indeed, the simplicity of the functional programming language's semantics favours DSLs with a similarly clear, equational semantics.

The functional programming approach works because of its unique combination of *higher-order types*, *laziness*, *parametric polymorphism* and *monads*. Together they allow type-compatible DSL program fragments to be "glued" together regardless of their internal structure [43], and may allow side-effects to be controlled by representing DSL computations as functional programming language values [108].

Note that not all functional languages support all these features. For example, languages in the ML family [67] are eager with implicit effects, and hence laziness and monads must be simulated when required. However, we think it is telling that *all* of the above combinator libraries have been implemented in Haskell [85], a language which directly supports all four features.

XML in Haskell?

Thus, the obvious question is whether the custom languages developed for XML may be embedded as combinator libraries within a Haskell-like language. The most appealing approach is to map XML concepts to functional-programming concepts as follows:

document type definition	⟶	type definitions
regular expression	⟶	type
element	⟶	term
document	⟶	program
document validation	⟶	type checking

Wallace and Runciman [111] have already tackled this question, and have developed two approaches. Their first approach ignores DTDs, and represents all elements in the universal datatype:

```
data Element = Atom String
             | Node String (List Element)
```

Under this scheme, our example would be represented as:

```
Node "Msg" [
  Node "From" [Atom "mbs@cse.ogi.edu"],
  Node "To" [Atom "jl@cse.ogi.edu"],
  Node "Bcc" [Atom "mbs@cse.ogi.edu"],
  Node "Body" [
    Node "P" [Atom "The..."],
    Node "P" [Atom "All..."]
  ]
]
```

Since every element now has type `Element`, it's easy to implement generic tree-manipulation combinators. However, Haskell's type system cannot ensure that all generated elements are valid with respect to any particular DTD.

To address this limitation, Wallace *et al.* also present a second approach which translates a DTD into a set of Haskell newtype declarations. Under this second scheme, our example would be represented as:

```
newtype Msg = Msg (List (Either To Bcc), From, Body)
newtype To = To String
newtype Bcc = Bcc String
newtype From = From String
newtype Body = List P
newtype P = P String

Msg (
  [Left (To "jl@cse.ogi.edu"),
   Right (Bcc "mbs@cse.ogi.edu")],
  From "mbs@cse.ogi.edu",
  Body [
    P "The...",
    P "All..."
  ]
)
```

Here `Either` is the datatype of "anonymous" sums:

```
data Either a = Left a
              | Right a
```

Notice how XML lists become Haskell `Lists`, XML tuples become Haskell tuples, choices become sums, and an arbitrary ordering is imposed on XML unordered tuples to become Haskell tuples.

This translation approach has the advantage of exploiting Haskell's type system to ensure only valid elements are generated. However, it does not respect XML's notion of type equality. In particular, the XML choice types `(To | Bcc)` and `(Bcc | To)` are equal in XML, but are translated into the distinct Haskell anonymous sum types `Either To Bcc` and `Either Bcc To`. Similarly, XML unordered tuple types are equal up to permutation, but are translated into Haskell tuples which, in general, are not equal up to permutation.

As a result, a programmer using the intended interpretation of elements as trees would be surprised if a Haskell complier rejected their program because of a "spurious" type error

involving these sum or tuple types. More concisely: this model of XML in Haskell is sound but not *complete*.

The underlying problem is that XML choice types are *unions* rather than *sums*, and any attempt to convert a union into a sum is forced to introduce an arbitrary label for each summand. The same problem arises if we attempt to convert an unordered tuple to an ordered tuple: again we are forced to impose an arbitrary ordering amongst member types. Thus there appears to be a fundamental mismatch between XML's regular expression types, and Haskell's sums-of-products datatypes.

XDuce

A third approach is thus to abandon sums-of-products types—and Haskell—and instead take regular expression types as fundamental. The language XDuce [38, 40, 39] has been developed specifically to test this idea. It is built upon subtype polymorphism using regular-expression language containment to induce the subtype relation. This form of subtype polymorphism allows an element to be viewed as belonging to more than one DTD simultaneously, and hence supports both code reuse and "DTD migration." Subtyping also meshes cleanly with a notion of regular-expression patterns.

Since XDuce models elements as trees and DTDs as a form of regular-tree grammar, it is both sound and complete. Thus a programmer would never be surprised by a "spurious" XDuce type error.

Our example would appear in XDuce as:

```
type msg = Msg[(to|bcc)* & from, body]
type to = To[String]
type bcc = Bcc[String]
type from = From[String]
type body = p*
type p = P[String]

Msg[
  To["jl@cse.ogi.edu"],
  Bcc["mbs@cse.ogi.edu"],
  From["mbs@cse.ogi.edu"],
  Body[
    P["The..."],
    P["All..."]
  ]
]
```

Notice type names and tag names are distinct within XDuce type declarations. Indeed, the e-mail DTD may be more concisely represented in XDuce by the single declaration:

```
type msg = Msg[(To[String]|Bcc[String])* & From[String],
               Body[P[String]*]]
```

Unfortunately, it is not at all clear whether this approach is compatible with higher-order functions and parametric polymorphism, which we have already seen to be essential to the combinator library approach to language embedding.

Type-Indexed Sums and Products

Thankfully, a compromise between Haskell's sums-of-products datatypes and XDuce's regular-expression types exists. Hidden within Appendix E of the XML standard [12] is the statement:

> "[I]t is required that content models in element type declarations be deterministic."

Here "deterministic" means that an element type declaration's regular expression must be *1-unambiguous*.

Informally, a regular expression is 1-unambiguous if, given a position within the regular expression and the tag of the next input element, there is a unique follow position. Formally, this condition holds if and only if the regular expression is recognisable by a deterministic Glushkov automaton [13, Lemma 2.5].

For example, the choice type `((P, Q) | (Q, P))` is unambiguous, while `((P, Q) | P)` is ambiguous. Similarly, the unordered tuple type `((P, Q) & (Q, P))` is unambiguous, but `((P, Q) & P)` is ambiguous.

There are two consequences of this restriction. Firstly, each member of an XML choice type or unordered tuple type must be distinct. In other words, XML choice types and unordered tuple types are formed from *sets* of types. Thus we can think of a choice type as a *variant* (sum) in which each member type serves as its own variant label. Dually, an unordered tuple type is like a *record* (product) in which each member type serves as its own record label. We call these *type-indexed sums* and *type-indexed products.*

The second consequence is that it is possible to transform any XML element into a term which represents lists, tuples, type-indexed sums, and type-indexed products explicitly. This transformation involves first (recursively) converting each sub-element to an appropriate sub-term, and then running an augmented Glushkov automaton corresponding to the element's type definition on the sub-term sequence. The automaton makes a transition based on the type of each sub-term, and incrementally constructs the result term using a stack of intermediate sub-terms.

In this thesis, we develop the idea of *type-indexed sums* and *type-indexed products* within a small calculus called λ^{TIR}. We show that the constructs are compatible with parametric polymorphism, higher-order functions and type inference. Furthermore, we show that conventional sum-of-products datatypes and records may be easily encoded within λ^{TIR}. Thus it is possible to retain all of the type features required for implementing combinator libraries, while simultaneously supporting XML document type definitions, and XML element syntax.

Under this approach, the XML types `(P | Q)` and `(Q | P)` are translated to the λ^{TIR} types `One (P # Q # Empty)` and `One (Q # P # Empty)`, which are equal. Note, however, that the equal XML types `(P | (Q | R))` and `((P | Q) | R)` are translated to the *unequal* λ^{TIR} types `One (P # (One (Q # R # Empty)) # Empty)` and `One ((One (P # Q # Empty)) # R # Empty)`. This inequality is a consequence of the compromise we must make between full regular-expression types and sums-of-products datatypes.

Note that 1-unambiguity is a stronger restriction on choice and unordered tuple types than distinctness of their member types. For example, the choice type `((P, Q) | P)` is

ambiguous, even though `(P, Q)` and `P` are distinct types. Thus, λ^{TIR} also allows sum and product types which are not deterministic XML types. This mismatch may be easily repaired.

Staging

Though the calculus λ^{TIR} goes much of the way towards supporting dynamic documents, it does not address the problem of active documents. Here the problem is to allow XML elements to contain scripts which are constructed on-the-fly just as any other data. Of course we could follow current practice and simply embed such scripts as strings, but this makes their syntactic and type correctness difficult to verify.

A better approach is to allow functions to appear within XML elements just as any other value. However, this approach would require all such functions to be converted from an intensional representation (*e.g.*, compiled code) to an extensional representation (*e.g.*, source or intermediate language code) whenever a document is moved between machines.

In this thesis, we tackle this problem by developing a system of *dynamically typed staged-computation* within a small calculus called λ^{SC}. Staging allows a single program to have its execution distributed over distinct run-time stages [88]. Furthermore, it is possible for distinct stages to be performed on distinct machines, since code values are easily transmitted over a network.

Under this approach an active document would be generated by a two stage program. In the first stage (run on the server), a piece of XML is generated which contains embedded code. These pieces of code may then be run as required by the client in the second stage.

This approach to active documents ensures that all generated program fragments are syntactically well-formed. Furthermore, it also guarantees such code is well-typed: either by checking at compile-time (for *statically typed code* [106]), or at run-time (for *dynamically typed code* [99]). This choice of static *vs.* dynamic is up to the programmer: static code gives a stronger guarantees of correctness, but can be overly restrictive.

From Calculi to a Language

Of course, λ^{TIR} and λ^{SC} are small and distinct *calculi*, whereas what's really required is a single *language*. Furthermore, we can hardly claim that λ^{TIR} and λ^{SC} alone subsume all the custom XML-centric languages mentioned above.

For example, XML elements may include *attributes*, and CSS [58] has special support for attribute inheritance. Much of this behaviour can be modelled using the *implicit parameters* of Lewis, Shields *et al.* [57] coupled with the *first-class polymorphism* of Jones [49].

Furthermore, query-like operations on documents, such as "collect all elements with tag `P`," are directly supported by XML Query [26, 25], but must be redefined afresh for each document type definition within λ^{TIR}. We think *generic programming* [37] is a viable solution to this problem.

This thesis does not address the difficult problem of combining all these distinct calculi, either theoretically or within an implementation. The problem is a topic for future research and implementation. Some early steps towards an integrated language have been taken in the design of XMλ [65], an experimental Haskell-like functional programming language

with direct support for XML. XMλ uses λ^{TIR} and λ^{SC} as its core, and also includes implicit parameters, first-class polymorphism, and support for definitions given by induction over (first-order) types.

1.1 Outline of Thesis

The thesis naturally divides into two parts.

Part I presents λ^{TIR}. Chapter 2 motivates the key ideas from the perspective of a polymorphic record calculus, which it most closely resembles. Chapter 3 presents some larger examples, including our motivating example of encoding XML types. (The machinery necessary to also support XML element syntax is outlined in Appendix A.) This chapter also demonstrates how λ^{TIR} supports a simple form of type-based overloading, which was the original motivation for its development.

Chapter 4 begins our formal development of λ^{TIR} by presenting its syntax, kind system, type system and a notion of constraint entailment. The calculus λ^{TIR} builds upon a system of qualified [47, 109], or constrained [79], polymorphism, and much of its machinery is devoted to the entailment and simplification of these constraints. This chapter also presents a denotational semantics for λ^{TIR}, and demonstrates type soundness. All the proofs for this chapter may be found in Appendix B.

Chapter 5 continues the formal development of λ^{TIR} by presenting a type inference and constraint simplification system. We demonstrate inference is sound and, with one caveat, complete. Some of the proofs for this chapter may be found in Appendix C. The proof of completeness is somewhat involved, because we cannot assume that constraints are in any particular normal form, and because we make no assumptions as to how often constraints are simplified. We have decided to omit this proof from Appendix C.

Chapter 6 concludes Part I by reviewing related work and outlining future work. In particular, this dissertation does not study the complexity of constraint entailment, satisfaction or simplification.

Part II presents λ^{SC}. Chapter 7 introduces the three constructs to defer, splice and run code, and motivates their typing rules, which turn out to be quite subtle. Chapter 8 presents larger examples of staging, including partial evaluation, dynamic typing and a small example of a server and client exchanging HTML-generating code. Some of these examples mix statically and dynamically typed code, demonstrating the utility of including both within a single calculus.

Chapter 9 presents a formal development of λ^{SC}, which includes type checking and a denotational semantics. We also demonstrate that the semantics is sound. The key problem for any semantics of staged computation is correctly accounting for the dynamic generation of variable names required whenever code is spliced under a binding operator. Our semantics is very pragmatic, and indeed is suitable for direct implementation. However the cost of this choice of semantics is a rather complicated soundness proof, which appears in Appendix D.

Chapter 10 concludes Part II, and the thesis, with an overview of related work and an outline of future work, which includes the problems of type inference, and *correctness* of our semantics with respect to a semantics which collapses all stages.

1.2 How to read this dissertation

Readers coming to this thesis from the XML community will, unfortunately, have a rather hard time. Of necessity, our work is at a very primitive level, and so the reader may find it difficult to see any connection with documents at all! We recommend starting with the introductory material of Chapters 2 and 7, then tackling the examples in Chapters 3 and 8.

To the reader coming from a functional programming background, we assume familiarity with Haskell [85] and with the system of qualified types [47] from which its type-class system is constructed. A passing familiarity with monadic semantics [11, 108] will aid the understanding of our denotational semantics. Implicit parameters [57] are used as an example constraint domain in Section 7.4. Otherwise, Parts I and II are mostly self-contained, and may be read independently.

The proofs in Appendix B, C and D have been included for completeness. For the most part they proceed by obvious induction on the relevant derivation. This is not to say that the theorems themselves are always straightforward! As is typical in type-theoretic proofs, the hard part is getting the induction hypothesis *just right*.

Part I

Type-Indexed Rows

Abstract

Record calculi use labels to distinguish between the elements of products and sums. This part presents a novel variation, *type-indexed rows*, in which labels are discarded and elements are indexed by their type alone. The calculus, λ^{TIR}, can express tuples, recursive datatypes, monomorphic records, polymorphic extensible records, and closed-world style type-based overloading. Our motivating application of λ^{TIR}, however, is to encode the "choice" types of XML, and the "unordered tuple" types of SGML. Indeed, λ^{TIR} is the kernel of the language XMλ, a lazy functional language with direct support for XML types ("DTDs") and terms ("documents").

The system is built from rows, equality constraints, insertion constraints and constrained, or qualified, parametric polymorphism. The test for constraint satisfaction is complete, and for constraint entailment is only mildly incomplete. We present a type checking algorithm and show how λ^{TIR} may be implemented by a type-directed translation which replaces type-indexing by conventional natural-number indexing. We also present a constraint simplification algorithm and type inference system.

Chapter 2

Introduction

Record calculi (and less often, variant calculi) appear in many contexts. Some functional languages incorporate them in conjuction with more conventional tuples and recursive sums-of-products datatypes [46]. They have been used as foundations for object-oriented languages [112]: Objects can be modelled by records, and subclassing can be built upon record subtyping. Database query languages often model relations as sets of records, and, because database schema are dynamic, require a particularly flexible type system [14].

In this part we present a system very much like an extensible, polymorphic record calculus, but with an essential twist: *We discard labels.* Instead of labels, elements of products and sums are distinguished by their type alone. That is, a *type-indexed row* (TIR) is a list of types (possibly with a type variable as tail), from which we form *type-indexed products* (TIPs) and *type indexed co-products* (TICs). The rôle of labels is played by *newtypes*, which introduce fresh type names.

Of course, in a monomorphic setting such a system is straightforward. In the presence of polymorphism, however, we must somehow resolve the paradox of rows indexed by types which are partially or fully unknown (*i.e.*, contain free type variables).

We developed λ^{TIR} to treat the *regular expression types* of XML [12] and SGML [45] as types in a functional language we are developing called XMλ [65]. XML includes "choice" types of the form $(\tau_1 \mid \ldots \mid \tau_n)$ and SGML includes "unordered tuple" types of the form $(\tau_1 \& \ldots \& \tau_n)$. Neither of these types include any syntactic information, such as labels, to guide a type checker in deciding which summand of a sum, or which permutation of a product, a given term belongs to. Instead, a *1-unambiguity* condition is imposed, which implies membership of a term in a regular-expression type may be decided by a deterministic Glushkov automaton [13]. In λ^{TIR}, we abstract from this formulation by requiring only that each type in a sum or product be distinct. Such types may then be encoded within λ^{TIR}, which allows XML elements to be manipulated within a polymorphic functional programming language.

Serendipitously, we also found λ^{TIR} could naturally encode:

- conventional tuples and recursive sums-of-products datatypes;
- many existing record calculi, both monomorphic and polymorphic, extensible and non-extensible;
- types resembling Algol 68's union types; and,
- the closed-world style of type-based overloading (modulo subtyping) popular in object-oriented languages [34].

XMλ has many of the types mentioned above. The XMλ complier simply translates each into λ^{TIR}, resulting in a compact and uniform compiler. Hence λ^{TIR}'s expressiveness is not merely of theoretical interest, but can also be exploited in practice.

Many of the ingredients of λ^{TIR} are well known:

- We use a *kind system* to distinguish rows from types.
- As in record calculi, we require *insertion constraints* to ensure the well-formedness of rows, only now they state that a *type* may be inserted into a row.
- Unlike in record calculi, we also require *equality constraints*, as sometimes the unification of two rows must be delayed if there is any ambiguity as to the matching of their element types.
- Constrained polymorphism [47, 79] is used to propagate constraint information throughout the program, thus ensuring soundness.
- We eagerly test for the unsatisfiability of constraints so as to reject programs as early as possible.
- As in Gaster and Jones' record calculus [31], λ^{TIR} is implemented by a type-directed translation which replaces type-indexing by natural-number indexing. These indices propagate via implicit parameters at run-time to parallel the propagation of insertion constraints at compile-time.

We first review record calculi (Section 2.1), then motivate the introduction of each of the components above by small examples (Sections 2.2–2.9). More extensive worked examples are also presented in Chapter 3. We then develop a type-checking system for λ^{TIR} which simultaneously performs a type-directed translation into an untyped run-time language (Chapter 4). This system requires the notion of *constraint entailment* (Section 4.4). We also demonstrate our system is sound (Section 4.5).

In Chapter 5 we consider type inference for λ^{TIR} programs. This is built upon a constraint simplification system (Section 5.2), which we show correct with respect to constraint entailment. We then show soundness and completeness of inference with respect to type-checking (Section 5.3).

A very much shorter version of this part appeared in POPL'01 [98].

2.1 Review: Label-Indexed Rows

To aid the transition to λ^{TIR}, we first quickly review existing calculi of labelled records and variants. We use a somewhat unorthodox syntax, though none is particularly standard anyway. We assume an ambient type system and a set of *label names*.

Rows

We first introduce *rows* [112], which are lists of labelled types. For example:

```
(xCoord: Int) # (yCoord: Int) # Empty
```

is a row with label names `xCoord` and `yCoord`, both labelling type `Int`. Here we use the `#` operator to denote *row extension*, and `Empty` to denote the empty row. (Note that in this dissertation we shall assume labels are formed from label names by appending a '`:`'.)

Sometimes *row concatenation* replaces or augments row extension [35], though we do not consider this here.

Rows are equal up to a permutation of their labelled types. That is, the elements of a row are distinguished by their label name rather than by their position.

A record calculus is *extensible* if a row may end with a type variable instead of just `Empty`. For example:

```
(xCoord: Int) # (yCoord: Int) # a
```

is an *open row*, with *tail* variable `a`. Binding `a` to `col: Colour # Empty` yields the extended *closed* row:

```
(xCoord: Int) # (yCoord: Int) # (col: Colour) # Empty
```

In this manner, when coupled with parametric polymorphism, extensible rows may simulate record subtyping [16].

A record calculus is *label polymorphic* if the same label name may label different types in different rows. For example, the rows:

```
(xCoord: Int) # Empty
(xCoord: Real) # (depth: Real) # Empty
(xCoord: a) # b
```

may all coexist within one program. As we shall see, the type system must work a little harder to ensure type correctness in the presence of polymorphic labels.

Rows are distinct from types, but may be used to form both record and variant types.

Records

A *record type* interprets a row as a product of label-indexed types. For example:

```
All ((xCoord: Int) # (yCoord: Int) # Empty)
```

is a record type with two labels. We write `All` to denote the record type constructor because records contain *all* elements of a row.

At the term level, we have the empty record `Triv` (of type `All Empty`), and a record extension operator `(l: _ && _)` for each label name `l`. (Throughout this dissertation we assume a distfix syntax for operators in which argument positions are written as `_`.) For example:

```
((xCoord: 1) && (yCoord: 2) && Triv)
```

is a record with the record type given above.

Calculi typically also include a label selection operator `(_.l)` for each label name `l`. For our purposes we prefer to use pattern matching. For example:

```
let getYCoord = \((yCoord: z) && _) . z
in getYCoord ((xCoord: 1) && (yCoord: 2) && Triv)
```

evaluates to 2.

Variants

Dually to records, a *variant type* interprets a row as a sum of label-indexed types. For example:

```
One ((isInt: Int) # (isBool: Bool) # Empty)
```

is a variant type with two labels. We write `One` to denote the variant type constructor because sums contain *one* element of a row.

At the term level, we have an injector `(Inj l: _)` for each label name `l`. For example:

```
(Inj isBool: True)
```

injects `True` with the label `isBool` into the above variant type.

We also need a way to test a variant against a label. Again, we prefer to allow an injector to be used as a pattern, and shall allow a set of λ-abstractions to be grouped together to mimic case-analysis. For example, consider:

```
{ \(Inj isInt: x) . 1 - x;
  \(Inj isBool: y) . if y then 0 else 1 }
  (Inj isBool: True)
```

The two λ-abstraction patterns will be tried in left-to-right sequence. In this case, the second pattern will match, and the term reduces to `0`.

Notice that the type `One Empty` contains only the undefined term.

Soundness

Though liberal, record and variant calculi are not anarchic: Somehow they must prevent a row from ever containing duplicate label names. For extensible record calculi this constraint requires some form of global analysis. For example, to reject (as surely we must) the program:

```
let f = \x y . ((xCoord: x) && y)
in (f 2 ((xCoord: 1) && Triv))
```

involves looking both at the definition and call sites for `f`.

A particularly elegant solution is to introduce qualified (constrained) polymorphism [47] and *insertion constraints* (called "lacks" constraints in the system of Harper *et al.* [35].) We refer the reader to the work of Gaster and Jones [31] for a cogent exposition of this approach. Briefly, let-bound terms are assigned a type scheme which includes any constraints on the possible instantiations of quantified type variables. In the example above, `f` would be assigned the scheme:

```
forall a b . xCoord ins b =>
  a -> All b -> All ((xCoord: a) # b)
```

which can be read as:

> "for all types a and rows b such that the label name xCoord may be inserted into b, the function from a and All b to All ((xCoord: a) # b)."

Now each use of f is free to instantiate a and b, but *subject to the constraint* xCoord *ins* b. Since our example program attempts to instantiate b to ((xCoord: Int) # Empty), which already contains the label name xCoord, it is rejected.

2.2 From Label- to Type-Indexed Rows

As a first step towards λ^{TIR}, consider naïvely erasing labels from the record and variant operators above.

We let the kind system keep rows, of kind Row, separate from types, of kind Type. Our presentation will be greatly simplified if we also allow *higher-kinds*, so that we may present our type operators as constants. We use : to denote "has kind" (and later, "has type").

A *type indexed row* (TIR) is either the empty row or an extension of another row. Row extension is now free of label names:

```
Empty : Row
(_ # _) : Type -> Row -> Row
```

For example:

```
(Int # Bool # Empty)
```

is a closed row containing the element types Int and Bool. Rows are considered equal up to a permutation of their element types.

We also have two dual interpretations for a row: as a *type-indexed product* (TIP) or *type-indexed coproduct* (TIC) type:

```
(All _) : Row -> Type
(One _) : Row -> Type
```

A TIR is useful if its element types are all distinct. Because we allow open rows, this cannot be verified locally, and so will be propagated using constraints. The insertion constraints of λ^{TIR} resemble those of record calculi, but with a type instead of a label. For example:

a *ins* (Int # Bool # Empty)

constrains a to be any type other than Int or Bool. Hence:

(List b) *ins* (Int # Bool # Empty)

is true: for every type b, List b cannot be equal to Int or Bool, and hence may be inserted into the row.

With the types and constraints in place, we now consider terms. A TIP is either the trivial

product, or an extension of another:

```
Triv : All Empty
(_ && _) : forall (a : Type) (b : Row) .
           a ins b => a -> All b -> All (a # b)
```

A TIC is an injection of a term:

```
(Inj _) : forall (a : Type) (b : Row) .
          a ins b => a -> One (a # b)
```

Notice the use of insertion constraints to ensure the type a to insert does not already appear within the row b of the TIP or TIC.

For example:

```
(1 && True && Triv) : All (Int # Bool # Empty)
(Inj True) : forall (a : Row) . Bool ins a => One (Bool # a)
```

We also allow any of the above three constants to appear within patterns. For example:

```
let flip = \(x && y && Triv) . ((1 - x) && (not y) && Triv)
in flip (True && 1 && Triv)
```

evaluates to `(0 && False && Triv)`. Notice the pattern `(x && y && Triv)` contains no explicit type information, and certainly no labels! It was the *type* of `x` within the body of `flip` which determined it was bound to `1` rather than `True`.

Case analysis of TIPs and TICs is also possible. For example, consider:

```
let flop = { \(Inj x) . 1 - x;
             \(Inj y) . if y then 0 else 1 }
in flop (Inj True)
```

Since `x` is of type `Int`, and `y` of type `Bool`, the second pattern will match, and the term reduces to `0`. Since all functions grouped by `{...}` must have the same type, we find:

```
flop : forall (a : Row) .
         Int ins a, Bool ins a =>
           One (Int # Bool # a) -> Int
```

2.3 Equality Constraints

Consider a more challenging variation of the `flip` example:

```
let tuple = \(x && y && Triv) . (x, y)
in tuple (True && 1 && Triv)
```

(Here we assume λ^{TIR} to be enriched by conventional tuples, though they are easily encoded: See Section 3.1.) Unlike `flip`, the body of `tuple` is fully polymorphic in the types of `x`

and y. Hence:

```
tuple : forall (a : Type) (b : Type) .
          a ins (b # Empty) =>
            All (a # b # Empty) -> (a, b)
```

Now consider how to type-check the application of `tuple`. Assume its scheme has been specialised to fresh type variables `c` and `d`. Then we must unify rows `All (c # d # Empty)` and `All (Int # Bool # Empty)` subject to the constraint `c` *ins* `(d # Empty)`. Depending on which of `Int` or `Bool` we bind to `c`, the overall term has type `(Int, Bool)` or `(Bool, Int)`. Choosing one solution above another would destroy completeness of type inference. Rejecting such terms would prevent many useful examples (in particular, overloading: See Section 3.5).

Our solution is to introduce *equality constraints* to record which types and rows must be equal for a term to be well-typed. For example:

```
(c # d # Empty) eq (Int # Bool # Empty)
```

represents the constraint that `tuple` and its argument `(True && 1 && Triv)` agree in type. As with insertion constraints, equality constraints propagate until sufficient type information is available to simplify them.

For convenience, we allow equality constraints on both rows and types. (Type equality constraints may always be simplified down to row equality constraints as soon as they are introduced, hence they add no expressiveness to the system.)

Now consider:

```
let oneTrue =
  let tuple = \(x && y && Triv) . (x, y)
  in tuple (True && 1 && Triv)
in (1 - fst oneTrue, not (fst oneTrue))
```

Using equality constraints, we may assign `oneTrue` a *principal type scheme*:

```
forall (c : Type) (d : Type) .
  (c # d # Empty) eq (Int # Bool # Empty),
  c ins (d # Empty) =>
    (c, d)
```

Notice that the first element of `oneTrue` has been used in both an `Int` context and `Bool` context, and the term reduces to `(0, False)`. To see how this works, consider each use of `oneTrue`. For the left use, `oneTrue` is specialised to a tuple with an `Int` first component. Hence its constraint is specialised to:

```
(Int # e # Empty) eq (Int # Bool # Empty),
Int ins (e # Empty)
```

where `e` is a fresh variable. This constraint may be simplified by binding `e` to `Bool`, and is thus `true`.

Similarly, for the right use the specialised constraint is:

```
(Bool # f # Empty) eq (Int # Bool # Empty),
Bool ins (f # Empty)
```

Again, the constraint is simplified to `true` with `f` bound to `Int`.

Membership and equality constraints interact in interesting ways. Indeed, much of the machinery of λ^{TIR} is devoted to the entailment and simplification of such mixed constraints. For example, the constraint:

```
Int ins (a # Empty),
(Int # Bool # Empty) eq (a # b # Empty)
```

may be simplified to `true` by binding `a` to `Bool` and `b` to `Int`, because the membership constraint prevents the binding of `a` to `Int`.

2.4 Simplifying Constraints

We say a substitution is a *satisfying substitution* for constraint C if it makes C ground and `true`. For example, the substitution [a $\mapsto$ Int] satisfies the constraint

```
a ins (Bool # Char # Empty)
```

We say a constraint C *entails* a constraint D if every satisfying substitution for C also satisfies D. Two constraints are *logically equivalent* if each entails the other.

Constraint simplification attempts to reduce a constraint to a smaller but logically equivalent constraint, and a *residual substitution.* The substitution can be thought of simply as a particularly efficient representation for equality constraints between type variables and types. We have already seen some examples of constraint simplification. In this section we outline the *simplification rules* which guide this process.

Firstly, we require rules for *simple unification* of types. For example

```
(a -> Int) eq (Bool -> b)
```

is simplified to

```
a eq Bool, Int eq b
```

using a rule which "unwraps" the common type constructor `(_ -> _)`.

We also require rules for the unification of rows. Because rows are only equal up to permutation, row unification is a little more subtle than simple unification. The row *matching* rule allows a type from each row to be removed and unified when this choice is unambiguous. For example

```
(Int # a # Empty) eq (Bool # b # Empty)
```

is simplified to

```
(Int eq b), (a # Empty) eq (Bool # Empty)
```

by matching `Int` with `b`.

The row *extension* rule allows a type from one row to extend the tail of another row, again provided the choice of type is unambiguous. For example

(Int # a) *eq* (Bool # b)

is simplified to

a *eq* (Bool # b')

with residual substitution [b $\mapsto$ Int # b']. Here `b'` is a fresh type variable of kind `Row`.

Another set of rules allow insertion constraints to be simplified when types are guaranteed to be distinct. For example

(a, b) *ins* (Bool # c # Empty)

is simplified to

(a, b) *ins* (c # Empty)

since `(a, b)` can never be unified with `Bool`.

The simplifier also has rules for constraint *projection*, however a discussion of these rules is best deferred to Chapter 5.

2.5 Newtypes

So far λ^{TIR} can only distinguish types *structurally*. In order to distinguish types by *name* we allow the programmer to introduce fresh type names, called *newtypes* (as in Haskell [85]).

A newtype declaration takes the form:

newtype A = \Δ . τ

where A is the newtype name, Δ a sequence of kinded type variables, and τ a type (of kind `Type`).

At the type level, newtype names behave as uninterpreted types (or, in general, *type constructors*). For example, assuming the declarations:

```
newtype A = \(a : Type) . a
newtype B = Int
newtype C = Int
```

then `A Int`, `A Bool`, `B`, `C` and `Int` are all distinct types.

At the term level, newtype names behave as single-argument data constructors. These names may be used both to construct terms:

```
((A 1) && (A True) && (B 2) && (C 3) && 4) :
  All ((A Int) # (A Bool) # B # C # Int # Empty)
```

and to pattern match against terms in λ-abstractions:

```
\A x . x + 1 : A Int -> Int
\A x . not x : A Bool -> Bool
\B x . x + 1 : B -> Int
```

In effect, every newtype declaration introduces a polymorphic constant:

$$\boxed{A : \texttt{forall}\ \Delta\ .\ \tau \texttt{ -> } A\ \Delta}$$

Using newtypes, we can encode conventional monomorphic records by declaring a newtype for each label. For example, with declarations:

```
newtype xCoord = Int
newtype yCoord = Int
```

we have:

```
((xCoord 1) && (yCoord 2) && Triv) :
  All (xCoord # yCoord # Empty)
```

What about polymorphic record calculi? A obvious approach would be to declare each label to be the type-identity function:

```
newtype xCoord = \(a : Type) . a
newtype yCoord = \(a : Type) . a
```

With these declarations, xCoord and yCoord may "label" terms of any type in any "record:"

```
((xCoord 1) && (yCoord 2) && Triv) :
  All ((xCoord Int) # (yCoord Int) # Empty)
((xCoord '1') && (yCoord "two") && Triv) :
  All ((xCoord Char) # (yCoord String) # Empty)
```

Unfortunately, it also allows the same newtype to appear within the same record, provided it labels terms of different types:

```
((xCoord 1) && (xCoord '1') && Triv) :
  All ((xCoord Int) # (xCoord Char) # Empty)
```

Though at first glance this may seem a useful generalisation of labels, we quickly run into problems when unifying rows containing them. For example, if xCoord really was a polymorphic label, then the following constraint should be simplified by binding a to Int:

```
((xCoord a) # b) eq ((xCoord Int) # c),
(xCoord a) ins b,
(xCoord Int) ins c
```

However, as things stand, the simplifier would be incorrect if it were to do so.

To see why, consider the possible substitution which binds b to (xCoord Int) # Empty,

and c to (xCoord Bool) # Empty. The constraint becomes:

```
((xCoord a) # (xCoord Int) # Empty) eq
  ((xCoord Int) # (xCoord Bool) # Empty),
(xCoord a) ins ((xCoord Int) # Empty),
(xCoord Int) ins ((xCoord Bool) # Empty)
```

which implies a must be Bool, not Int. Hence, our simplifier is stymied by an excess of polymorphism.

Our solution is to introduce *opaque newtypes*, a variation of newtypes in which the type arguments are ignored when considering the simplification of insertion constraints.

Returning to our example, consider redeclaring the labels as:

```
newtype opaque xCoord = \(a : Type) . a
newtype opaque yCoord = \(a : Type) . a
```

Now the simplifier is free to bind a to Int in our constraint:

```
((xCoord a) # b) eq ((xCoord Int) # c),
(xCoord a) ins b,
(xCoord Int) ins c
```

This is because the membership constraint (xCoord a) *ins* b implies that b cannot contain any type of the form xCoord τ, hence b cannot be extended to include xCoord Int, and hence xCoord Int must match xCoord a.

Furthermore, with xCoord declared as an opaque newtype, the term:

```
((xCoord 1) && (xCoord '1') && Triv)
```

is ill-typed, because the constraint

```
(xCoord Int) ins ((xCoord Char) # Empty)
```

is unsatisfiable.

Though at first glance they appear somewhat ad-hoc, opaque newtypes require very little special support within the machinery of λ^{TIR}.

Why not make *all* newtypes opaque? Though this would simplify the presentation and machinery of λ^{TIR}, it would prevent type-based overloading on the arguments to type constructors. This will be covered in Section 3.5.

2.6 Implementing Records

For the moment we put type-indexed rows aside and consider how to implement conventional label-indexed records. A naïve approach is as a map from labels to values, but then each access requires a dynamic lookup. A better approach, first suggested by Ohori [80], and independently, by Jones [47], is to use the type information we already have to replace label names with natural number indices, and records with vectors. When a closed record is manipulated, these indices can be easily generated by finding a canonical ordering of

label names. When an open record is manipulated within a polymorphic function, these indices must be passed as implicit arguments because their actual values will depend on how the function has been instantiated.

This situation seems rather complicated until it is noticed that indices propagate at run-time in parallel with insertion constraints at compile-time, except in the opposite direction.

Consider:

```
let f = \x . ((yCoord: 20) && x)
in f ((xCoord: 10) && Triv)
```

To ensure its body is well-formed, `f` is assigned the type scheme:

```
forall (b : Row) . yCoord ins b =>
  All b -> All ((yCoord: Int) # b)
```

At the application of `f`, `b` is specialised to `(xCoord: Int) # Empty`, and thus `f`'s constraint is specialised to `yCoord` ***ins*** `((xCoord: Int) # Empty)`. This constraint is then introduced into the application's constraint context, where it may be simplified to `true`. Notice how `f`'s constraint propagated (at compile-time) from the site of its definition to the site of its use.

Now associate a run-time *index variable*, w, with `f`'s constraint `yCoord` ***ins*** `b`, with the understanding that w will be bound at run-time to the *insertion index* of `yCoord` within whatever row `b` is specialised to. Or, to use OML's terminology [47], w will be bound to a *witness* of the satisfaction of the constraint that `yCoord` may be inserted into row `b`.

The function `f` is now compiled to a function accepting w as an additional implicit parameter:

$$\begin{array}{l}\mathsf{let\ f} = \lambda \mathsf{w}\ .\ \lambda \mathsf{x}\ .\ \mathsf{insert\ 20\ at\ w\ into\ x} \\ \mathsf{in}\ \ldots\end{array}$$

Here we use sans-serif font to denote run-time terms, and insert U at W into T inserts the term U at index W into the vector T.

In the application of `f`, again associate an index variable w′ with the specialised constraint `yCoord`***ins***`((xCoord: Int) # Empty)`. This variable is passed to f, along with its argument:

$$\mathsf{f\ w'}\ \langle 10 \rangle$$

Here $\langle \ldots \rangle$ denotes a base-1 vector of run-time terms. (We shall use a special syntax for indices to prevent their semantic confusion with ordinary integers: One is the base index, and Inc W, Dec W the obvious offsets.)

Now when the simplifier rewrites `yCoord` ***ins*** `((xCoord: Int) # Empty)` to `true`, it is also obliged to supply a binding for w′. Assuming a lexicographic ordering on label names, `yCoord` should be inserted at index Inc One into the row `(xCoord: Int) # Empty`, hence w′ is bound to the *absolute index* Inc One.

Thus the overall term is compiled as:

$$\begin{array}{l}\mathsf{let\ f} = \lambda \mathsf{w}\ .\ \lambda \mathsf{x}\ .\ \mathsf{insert\ 20\ at\ w\ into\ x} \\ \mathsf{in\ let\ w'} = \mathsf{Inc\ One} \\ \mathsf{in\ f\ w'}\ \langle 10 \rangle\end{array}$$

which reduces to the vector $\langle 10, 20\rangle$.

Notice how the insertion index for yCoord within b was passed at run-time from the use site to the definition site, exactly in reverse of the propagation of the constraint yCoord ins b at compile-time.

This type-directed translation is an instance of the *dictionary translation* [109]. We call a set of constraints with associated index variables a *constraint context*, by analogy with type contexts.

An index may sometimes depend on another. For example, the constraint context:

$$(\mathsf{w} : \mathtt{yCoord}\ \boldsymbol{ins}\ \mathtt{((xCoord:\ Int)\ \#\ b)}), (\mathsf{w}' : \mathtt{yCoord}\ \boldsymbol{ins}\ \mathtt{b})$$

can be simplified to $\mathsf{w} : \mathtt{yCoord}\ \boldsymbol{ins}\ \mathtt{((xCoord:\ Int)\ \#\ b)}$ by binding w' to the *relative index* Dec w. This simplification is possible because yCoord will always be after xCoord in any row.

The same technique works for variants, which are represented as a pair of a natural number and value.

2.7 Implementing TIPs and TICs

Can we implement λ^{TIR} also using only natural number indices, vectors and pairs? The trick only works if we have an ordering on types. Clearly a total order on all types won't do, as then the relative ordering of non-ground types may change under substitution—disaster!

An obvious approach is to choose some ordering on monotypes, and only consider simplifying an insertion constraint $\upsilon\ \boldsymbol{ins}\ (\tau_1\ \#\ \ldots\ \#\ \tau_n\ \#\ \mathtt{Empty})$ when υ and each τ_i are ground. Then finding the index for υ is simply a matter of sorting these types. Unfortunately, because programs are often polymorphic all the way up to their top level, this approach would result in many insertion constraints propagating to the top level, leading to very large constraint contexts.

Thankfully, a less conservative ordering is possible. Assume we have a total order, $\leq^F$, on all built-in type constants (such as Int, (All _) and (_ -> _)), and all newtype names. Let $\leq^{F\mathsf{a}}$ be $\leq^F$ extended to type variables, on which it is always false. So, for example:

$$\mathtt{Int} \leq^{F\mathsf{a}} \mathtt{Bool} \leq^{F\mathsf{a}} \mathtt{String} \leq^{F\mathsf{a}} \mathtt{(_ \text{->} _)} \leq^{F\mathsf{a}} \ldots$$

but $\mathsf{a} \not\leq^{F\mathsf{a}} \mathtt{Int}$ and $\mathtt{Int} \not\leq^{F\mathsf{a}} \mathsf{a}$.

Every type τ has a *pre-order flattening*, denoted by $\mathit{preorder}(\tau)$. For example, $\mathit{preorder}(\mathtt{A\ Int\ \text{->}\ B\ Bool\ a}) = [\mathtt{(_ \text{->} _)}, \mathtt{A}, \mathtt{Int}, \mathtt{B}, \mathtt{Bool}, \mathsf{a}]$. We then (roughly) define the partial order, $\leq$, on all types as follows:

$$\tau \leq \upsilon \iff \mathit{preorder}(\tau) \leq^{lex} \mathit{preorder}(\upsilon)$$

where $\leq^{lex}$ is the lexicographic ordering induced by $\leq^{F\mathsf{a}}$. Notice that $\leq$ enjoys invariance under substitution, *viz*:

$$\tau \leq \upsilon \Longrightarrow \forall\theta\ .\ \theta\,\tau \leq \theta\,\upsilon$$

This property allows many insertion constraints to be discharged even when they contain

type variables.

For example, consider the constraint:

```
w : (Bool -> a) ins ((Int -> b) # Int # Empty)
```

All of these types may be totally ordered:

```
Int < (Int -> b) < (Bool -> a)
```

Thus we eliminate the constraint and bind w to Inc Inc One.

However, since the types in:

```
w : (Bool -> a) ins ((b -> c) # Int # Empty)
```

cannot be totally ordered, this constraint cannot be further simplified.

The alert reader will notice we ignored the possible permutation of row elements in the description above. To account for this, we must first find the *canonical* order of every row within types before flattening them. We defer the full definition of type order to Section 4.3.

2.8 Ambiguity

λ^{TIR} type schemes sometimes quantify over type variables which appear only in the scheme's constraint. For example, in

```
forall (a : Type) (b : Row) . (a # b) eq (Int # Bool # Empty) => a -> a
```

the variable b is not free in a -> a. However, since a binding for a uniquely determines a binding for b, this scheme is still sensible.

However, the scheme

```
forall (a : Type) (b : Type) . b ins (Int # Bool # Empty) => a -> a
```

is inherently *ambiguous*. Since the insertion constraint may never be eliminated, it will float to the top-level of the program and cause an error. Furthermore, a binding for b cannot be chosen arbitrarily, since different bindings may lead to different indices, and hence change the behaviour of the program.

Somewhat more subtle is the scheme:

```
forall (a : Type) (b : Type) . a ins (b # Empty) => One (a # b # Empty)
```

Even though all quantified type variables appear within its type, this scheme is still ambiguous. For example, though both of the instantiations

$$[\mathsf{a} \mapsto \texttt{Int}, \mathsf{b} \mapsto \texttt{Char}]$$
$$[\mathsf{a} \mapsto \texttt{Char}, \mathsf{b} \mapsto \texttt{Int}]$$

yield the same result type One (Int # Char # Empty), the index determined for the insertion

constraint differs.

These examples demonstrate that a simple syntactic test for ambiguity of λ^{TIR} type schemes is probably impossible. In particular, checking that each quantified variable appears within a scheme's type is neither a sound nor complete test for ambiguity. As a result, a compiler for λ^{TIR} should treat ambiguity as a warning rather than an error.

2.9 Satisfiability

When a let-bound term is generalised, any residual constraints accumulated while inferring its type which mention quantified type variables are shifted into its type scheme. However, we would also like to be sure such constraints are *satisfiable*, for two reasons. Practically, it helps improve the locality of type error messages if unsatisfiable constraints are caught at the point of definition rather than at some remote point of use. Theoretically, it simplifies our proof of type soundness if every type scheme is known to have at least one satisfying instance.

Often, the simplifier will detect unsatisfiability in the course of examining each primitive constraint. For example, in:

```
newtype opaque xCoord = \(a : Type) . a
let f = \x . ((xCoord 2) && (xCoord 1) && x)
in 1
```

assuming `x : All a`, then `f` has the constraint:

```
(xCoord Int) ins a,
(xCoord Int) ins ((xCoord Int) # a)
```

This constraint will be simplified to `false`, which is easily detected when generalising.

However, sometimes the simplifier will fail to detect unsatisfiability, because it never speculatively unifies rows. For example, in:

```
let g : All (Int # Bool # Empty) -> Int = ...
    h : All (Char # String # Empty) -> Int = ...
    f = \x y z . g (x && y && Triv) +
                 h (x && z && Triv)
in 1
```

assuming `x : a`, `y : b`, `z : c`, then `f` has the unsatisfiable constraint:

```
a ins (b # Empty), a ins (c # Empty),
(a # b # Empty) eq (Int # Bool # Empty),
(a # c # Empty) eq (Char # String # Empty)
```

Since this constraint will not be further simplified to `false`, the system must explicitly test for satisfiability when generalising.

Unfortunately, relying on the simplifier to show unsatisfiability is not quite enough. Consider the example:

```
newtype opaque xCoord = \(a : Type) . a
let f = \x . let g = \y . ((xCoord y) && x) in 1
in f ((xCoord 1) && Triv)
```

Assume `x : All a` and `y : b`. Then `g` has the satisfiable constraint:

`(xCoord b)` *ins* `a`

Thus `f` is assigned the type:

`forall (a : Row) . All a -> Int`

and the entire program has type `Int`.

However, under a naïve operational semantics for λ^{TIR}, β-reducing the application of `f` yields the program:

```
let g = \y . ((xCoord y) && (xCoord 1) && Triv) in 1
```

Now g's constraint becomes

`(xCoord b)` *ins* `((xCoord Int) # Empty)`

which is unsatisfiable. Hence, subject-reduction fails for this semantics. (Our semantics will actually be denotational rather than operational, but the problem remains the same.)

This problem occurs only when a let-bound term is both unused *and* has a constraint mentioning type variables bound at an outer scope. In the above example, g was unused in the body of `f`, and g's constraint contained the type variable a bound by `f`'s type scheme. This observation suggests four approaches to a solution.

The first approach attempts to constrain outer-scope variables in order to ensure the satisfiability of inner-scope constraints. One way of doing this is to use a new primitive constraint of the form:

`exists` Δ `.` C

with intended interpretation "C is satisfiable for some binding of the type variables of Δ." Existential constraints may be simplified "lazily," just as for equality and insertion constraints. This approach is advocated by HM(X) [79].

Using an existential constraint, `f` may be assigned the more precise type scheme:

```
forall (a : Row) . (exists (b : Type) . (xCoord b) ins a) =>
  All a -> Int
```

Now the application of `f` is ill-typed:

```
error: constraint
  exists (b : Type) . (xCoord b) ins ((xCoord Int) # Empty)
arising from application of 'f' is unsatisfiable.
```

Though elegant, existential constraints have a very subtle entailment theory. Indeed, an early version of λ^{TIR} included them, but the implementation was complicated and difficult to prove correct.

A variation on this first approach is to carry over generalised constraints into the current

constraint context unchanged. This method is termed *duplication* by Odersky *et al.* [79]. Now `f` would be assigned the type scheme:

```
forall (a : Row) (b : Type) . (xCoord b) ins a => All a -> Int
```

However, since `b` does not appear within the right hand side of `f`'s type, such a scheme is inherently ambiguous. Furthermore, this approach may result in many redundant insertion constraints. For example, the constraint:

```
a ins (b # Empty),
a ins (c # Empty),
a ins (b # c # Empty)
```

cannot be simplified, even though it is satisfiable exactly when the constraint:

```
a ins (b # c # Empty)
```

is satisfiable. Both these problems arise because insertion constraints imply the need for indices, whereas no such indices are required if our only interest is satisfiability.

A solution is, again, to introduce a new primitive constraint, but this time of the form:

$$\tau \ \textit{\textbf{nin}}\ \rho$$

τ ***nin*** ρ ("τ is not in row ρ") resembles τ ***ins*** ρ, but does not require the simplifier to calculate any index witnessing its satisfaction. During duplication, ***ins*** constraints are replaced by ***nin*** constraints.

Now `f` is assigned the type scheme:

```
forall (a : Row) (b : Type) . (xCoord b) nin a =>
  All a -> Int
```

This is no longer ambiguous since `b` may be chosen arbitrarily so as to satisfy the constraint. Again, the application of `f` is ill-typed.

Though quite workable, we feel this variation is ugly. In particular, the difference between "***ins***" and "***nin***" is a likely source of confusion.

The third approach is very simple: simply reject programs containing redundant let-bindings. Of course, an actual implementation would remove such bindings rather than reject the program. (Indeed, compilers tend to do this anyway as an optimisation.) This approach is adopted in OML [47, 48], and we adopt it for λ^{TIR}.

This approach works because if `x` is a let-bound variable with constraint C, and `x` is free in t, then the satisfiability of t's constraint implies the satisfiability of C.

Now a constraint may be tested for satisfiability *regardless of the scope of its free type variables.* If the test fails, the constraint is unsatisfiable for any instantiation of outer-scope variables, and an error may be reported. If the test succeeds, no further processing is required, because the satisfiability test for any let-bound terms in an outer scope shall entail the satisfiability of the current constraint.

In a sense, however, we have put the horse before the cart in all of this. Rather than change

the system to simplify the model, the fourth approach is to refine the model to correctly explain redundant, unsatisfiable let-bindings. Since such bindings cannot be observed, the problem is caused by *incompleteness* of semantic equality with respect to observational equality. However, such issues are notoriously subtle, hence our preference for the second (simple!) approach.

Chapter 3

Examples

In this section we show that λ^{TIR} may encode many conventional types, such as tuples and recursive sums-of-products datatypes. We also demonstrate an encoding of XML document-type definitions and a simple form of type-based overloading.

We write $\mathcal{T}[\![\ldots]\!]$ to denote the encoding function at the type level, and $\mathcal{S}[\![\ldots]\!]$ at the term level. Later examples assume the encoding provided by earlier examples.

Our XMλ compiler supports all of the types covered in this section by expanding each into λ^{TIR}. In order that error messages may use whatever syntax was used by the programmer rather than its translation, the compiler is careful to annotate translated types and terms with additional "hints" describing how they arose. Though not foolproof, this method seems preferable to extending the λ^{TIR} type system to deal with all of these types as primitives.

3.1 Tuples

We can simulate the positional notation of tuples by introducing an opaque newtype for each position:

```
newtype opaque fst = \(a : Type) . a
newtype opaque snd = \(a : Type) . a
...
```

Now `fst` τ is distinct from `snd` τ for any type τ.

A little sugar provides the familiar notation:

$$\begin{aligned}\mathcal{T}[\![\texttt{()}]\!] &= \texttt{All Empty}\\ \mathcal{T}[\![\texttt{(}\tau\texttt{, }\upsilon\texttt{)}]\!] &= \texttt{All ((fst } \mathcal{T}[\![\tau]\!]\texttt{) \# (snd } \mathcal{T}[\![\upsilon]\!]\texttt{) \# Empty)}\\ &\qquad\text{(...etc...)}\end{aligned}$$

$$\begin{aligned}\mathcal{S}[\![\texttt{()}]\!] &= \texttt{Triv}\\ \mathcal{S}[\![\texttt{(}t\texttt{, }u\texttt{)}]\!] &= \texttt{((fst } \mathcal{S}[\![t]\!]\texttt{) \&\& (snd } \mathcal{S}[\![u]\!]\texttt{) \&\& Triv)}\\ &\qquad\text{(...etc...)}\end{aligned}$$

Tuple projection is polymorphic on both the element type and tuple length:

```
fst : forall (a : Type) (b : Row) .
        (fst a) ins b => All (fst a # b) -> a
  = \(fst x && _) . x

fst (1, "two") : Int
fst ("one", 2, '3') : String
```

3.2 Records Revisited

Section 2.5 has already sketched how newtypes may simulate labels. A little syntactic sugar can make this encoding more convenient. Firstly, we allow any type or term to be "labelled":

$$\mathcal{T}[\![(\mathtt{l}\colon \tau)]\!] = \mathtt{l}\ \mathcal{T}[\![\tau]\!]$$
$$\mathcal{S}[\![(\mathtt{l}\colon t)]\!] = \mathtt{l}\ \mathcal{S}[\![t]\!]$$

(In a practical implementation, one could imagine the first occurrence of such a labelled type or term automatically adding the declaration:

```
newtype opaque l = \(a : Type) . a
```

to the compiler's internal tables.)

Secondly, some more sugar makes closed products and sums more convenient (where $n > 1$):

$$\mathcal{T}[\![(\tau_1\ \mathtt{\&}\ \ldots\ \mathtt{\&}\ \tau_n)]\!] = \mathtt{All}\ (\mathcal{T}[\![\tau_1]\!]\ \mathtt{\#}\ \ldots\ \mathtt{\#}\ \mathcal{T}[\![\tau_n]\!]\ \mathtt{\#}\ \mathtt{Empty})$$
$$\mathcal{T}[\![(\tau_1\ \mathtt{|}\ \ldots\ \mathtt{|}\ \tau_n)]\!] = \mathtt{One}\ (\mathcal{T}[\![\tau_1]\!]\ \mathtt{\#}\ \ldots\ \mathtt{\#}\ \mathcal{T}[\![\tau_n]\!]\ \mathtt{\#}\ \mathtt{Empty})$$
$$\mathcal{S}[\![(t_1\ \mathtt{\&}\ \ldots\ \mathtt{\&}\ t_n)]\!] = (\mathcal{S}[\![t_1]\!]\ \mathtt{\&\&}\ \ldots\ \mathtt{\&\&}\ \mathcal{S}[\![t_n]\!]\ \mathtt{\&\&}\ \mathtt{Triv})$$

With these, non-extensible records and variants are straightforward:

```
type Point = ((xCoord: Int) & (yCoord: Int))
let movex : Point -> Point
  = \((xCoord: x) && rest) . ((xCoord: x + 1) && rest)
in movex ((xCoord: 1) & (yCoord: 2))

type Num = ((isInt: Int) | (isReal: Real))
let asInt : Num -> Int
  = { \(Inj isInt: i) . i;
      \(Inj isReal: r) . floor r }
in asInt (Inj isReal: 3.1415)
```

(Here type introduces a type synonym.)

Extensible records and variants are similar.

3.3 Recursive Datatypes

Recursive datatypes may be simulated by recursive newtypes. Consider the datatype of binary trees (in an idealized ML notation):

```
data Tree = \(a : Type) . Node (Tree a, a, Tree a)
                        | Leaf
```

We may take this to be shorthand for the declarations:

```
newtype Tree = \(a : Type) . One ((Node a) # (Leaf a) # Empty)
newtype Node = \(a : Type) . (Tree a, a, Tree a)
newtype Leaf = \(a : Type) . ()
```

Each data constuctor wraps a newtype around its argument, and injects the result into the overall datatype. A little sugar can simulate the familar data constructor notation of ML:

$$\mathcal{S}[\![\texttt{Node}\ t]\!] = \texttt{Tree (Inj (Node}\ \mathcal{S}[\![t]\!]\texttt{))}$$
$$\mathcal{S}[\![\texttt{Leaf}]\!] = \texttt{Tree (Inj (Leaf ()))}$$

For example:

```
let flatten : forall (a : Type) . Tree a -> List a
  = { \Leaf . [];
      \Node (l, x, r) . (flatten l) ++ [x] ++ (flatten r) }
in flatten (Node (Leaf, 1, Node (Leaf, 2, Leaf)))
```

Note that if λ^{TIR} is given a lazy semantics, as is the case in this dissertation, this encoding suffers the "double lifting" problem for multi-argument data constructors. That is, λ^{TIR} programs may now distinguish an undefined datatype and a data constructor applied to an undefined tuple. For example, with the declarations:

```
undefined = undefined
test = \Node _ . True
```

we have:

```
test undefined ⇑
test (Node undefined) ⇓ True
```

3.4 XML

Chapter 1 introduced XML, and discussed the problem with naïvely encoding XML "choice" and "unordered tuple" regular expressions as ordinary Haskell-style sum and product types. In particular, equal XML regular expressions may become unequal Haskell types under the naïve encoding.

In this section we shall encode choice regular expressions as type-indexed sums, and unordered tuple regular expressions as type-indexed products. This encoding is total since XML's determinism constraint implies the components of a choice or unordered tuple must be distinct types. Furthermore, this encoding respects the commutativity of these XML operators. However, it does not respect any of the other regular expression equalities. Though the encoding is not perfect, it does allow XML elements to co-exist with all the other datatypes familiar to functional programmers: in particular higher-order functions and parametric polymorphism. We think this is a good compromise.

By design, our sugared syntax for tuples introduced in Section 3.1 coincides with XML's syntax for tuples. Similarly, our syntax for (closed) sums and products introduced in Section 3.2 also coincides with XML's syntax for choice and unordered tuple regular ex-

pressions. For the remaining regular expressions, we first introduce the datatypes of lists and optional terms (using the syntax of Section 3.3):

```
data List = \(a : Type) . Cons (a, List a) | Nil
data Option = \(a : Type) . Some a | None
```

We then introduce the following sugar:

$$\mathcal{T}[\![r\ *]\!] = \texttt{List}\ \mathcal{T}[\![r]\!]$$
$$\mathcal{T}[\![r\ ?]\!] = \texttt{Option}\ \mathcal{T}[\![r]\!]$$
$$\mathcal{T}[\![r\ +]\!] = \mathcal{T}[\![(r, r\ *)]\!]$$

There are two possible encodings of a document-type definition within λ^{TIR}. The first, which we shall term *DTD-style*, maps each XML element definition to a λ^{TIR} newtype definition. For example, the XML e-mail document-type definition of Chapter 1 may be trivially encoded as:

```
newtype Msg = (((To|Bcc)* & From), Body)
newtype To = String
newtype Bcc = String
newtype From = String
newtype Body = P*
newtype P = String
```

Just like XML DTDs, each newtype is given a fixed body type.

The second encoding, which we term *Scheme-style*, declares each tag name as a label-like newtype:

```
newtype Msg = \(a : Type) . a
newtype To = \(a : Type) . a
...
```

Then the specific structure of the e-mail DTD may be given by a single type declaration:

```
type MsgType = Msg (((To String | Bcc String)* & From String),
                    Body ((P String)*))
```

This second encoding is very similar to that used for XDuce, as shown in Chapter 1. It has the advantage of allowing the same tag name to be reused with differing body types. For example, `From` and `To` could be used elsewhere to tag dates instead of strings. This second encoding would thus be appropriate for the more general form of document type definitions allowed under XML Schema [24]. The disadvantage of this second encoding is that more type annotations must be supplied by the programmer when using XML element syntax. This shall be explained shortly.

XML documents are easy to manipulate in λ^{TIR}. For example, here is a program to implement a spam filter:

```
killSpam : Msg* -> Msg*
  = filter (not . isSpam)

isSpam : Msg -> Bool
  = \msg .
      getReceiver msg == "mbs@cse.ogi.edu" &&
      ( contains suspiciousWords (getWords msg) ||
        mem (getSender msg) suspiciousSenders )

getReceiver : Msg -> String
  = \(Msg ((rcvrs && _), _)) .
      (\[To to] . to)
        (filter' { \(Inj (To _)) . True; \_ . False } rcvrs)

suspiciousWords : String*
  = [ "money", "rich", "won", ... ]

getWords : Msg -> String*
  = ( words
    o toLowerCase
    o concat
    o map (\(P s) -> s)
    o (\(Msg (_, Body body)) . body)
    )

getSender : Msg -> String
  = \(Msg ((From from && _) , _)) . from

suspiciousSenders : String*
  = [ "quickcash@aol.com", "jl@cse.ogi.edu", ... ]
```

We assume a libary of standard functions whose types are given in Figure 3.1. (Some of these types have been specialised so that we may ignore the overloading of the equality operator within type schemes.) The filter discards all messages sent to `mbs@cse.ogi.edu` which are either from one of the `suspiciousSenders`, or contains one of the `suspiciousWords`.

Though λ^{TIR} newtype declarations resemble XML element type definitions, the same cannot be said for λ^{TIR} terms and XML elements. The example e-mail message of Chapter 1 (of type `Msg`) appears in native λ^{TIR} syntax as:

```
Msg (
  ( From "mbs@cse.ogi.edu"
  & [ Inj (To "jl@cse.ogi.edu"),
      Inj (Bcc "mbs@cse.ogi.edu") ] ),
  Body [
    P "The...",
    P "All..."
  ]
)
```

```
       filter : (Msg -> Bool) -> Msg* -> Msg*
      filter' : ((To|Bcc) -> Bool) -> (To|Bcc)* -> (To|Bcc)*
          not : Bool -> Bool
         (||) : Bool -> Bool -> Bool
     contains : String* -> String* -> Bool
          mem : String -> String* -> Bool
        words : String -> String*
  toLowerCase : String -> String
       concat : forall a . a* -> a
          map : forall a b . (a -> b) -> a* -> b*
            o : forall a b c . (b -> c) -> (a -> b) -> (a -> c)
```

Figure 3.1: Some (type specialised) standard library functions

Notice the explicit use of Inj to inject the To and Bcc terms into the correct sum, and the explicit type-indexed product, tuple, and list syntax.

We would prefer to be able to write this term in familiar XML syntax:

```
<Msg>
  <From>mbs@cse.ogi.edu</From>
  <To>jl@cse.ogi.edu</To>
  <Bcc>mbs@cse.ogi.edu</Bcc>
  <Body>
    <P>The...</P>
    <P>All...</P>
  </Body>
</Msg>
```

Notice that, as usual for XML, there is no need to explicitly inject the To and Bcc elements. Furthermore, the list of paragraphs is implicit, as is the tupling of the sender, reciever and Body elements. This additional syntax is unnecessary because, as far as XML is concerned, this term is simply a tree.

Thankfully, it is possible to further exploit the determinism of XML regular expressions and convert the XML element above to the coresponding λ^{TIR} term. In order to avoid cluttering this chapter, the precise technical development is deferred to Appendix A, and we present only an outline here.

We shall assume the e-mail DTD has been encoded in DTD-style. Roughly, the type checker first constructs an augmented Glushkov automaton for the body type of Msg, *viz*:

```
(((To | Bcc)* & From), Body)
```

This automaton is then run on the sequence of *types* From, To, Bcc, Body. Since this sequence is in the language of the type above when viewed as a regular expression, the automaton reaches an accepting state.

Furthermore, the automaton is augmented so as to maintain an internal stack of λ^{TIR} terms. As each element is seen, this stack will be updated to contain its λ^{TIR} representation. For

example, after seeing the From type, the automaton will have on it's stack the λ^{TIR} term:

```
From "mbs@cse.ogi.edu"
```

After seeing the Bcc type, the stack will be (from bottom to top):

```
From "mbs@cse.ogi.edu",
Inj (To "jl@cse.ogi.edu"),
Inj (Bcc "mbs@cse.ogi.edu")
```

Notice how the Inj constructors have been automatically inserted. When the Body type is seen, the two Inj terms are popped from the stack and replaced with a single list:

```
From "mbs@cse.ogi.edu",
[Inj (To "jl@cse.ogi.edu"), Inj (Bcc "mbs@cse.ogi.edu")]
```

These two terms are then replaced with a single type-indexed product:

```
( From "mbs@cse.ogi.edu" &
  [Inj (To "jl@cse.ogi.edu"), Inj (Bcc "mbs@cse.ogi.edu")])
```

This process continues until the stack contains the single λ^{TIR} message term given above. (For clarity the above explanation used λ^{TIR} source terms, whereas the automaton actually manipulates λ^{TIR} run-time terms.)

XMλ includes this support for XML element syntax. Furthermore, XMλ allows XML and λ^{TIR} syntax to be intermixed. For example, another way of writing the example e-mail message is:

```
let name = { \"Mark" . "mbs@cse.ogi.edu";
             \"John" . "jl@cse.ogi.edu" };
    body = [<P>The...</P>, <P>All...</P>]
in <Msg>
     <From><<name "Mark">></From>
     <To><<name "John">></To>
     <Bcc><<name "Mark">></Bcc>
     <Body><<body>></Body>
   </Msg>
```

The <<...>> brackets escape from XML syntax back into λ^{TIR} syntax.

XML syntax is also supported within XMλ patterns. For example:

```
getWords : Msg -> String*
  = ( words
    o toLowerCase
    o concat
    o map (\<P><<s>></P> -> s)
    o (\<Msg><<(_ & _)>><Body><<body>></Body></Msg> . body)
    )
```

Notice the use of the pattern (_ & _) within the body of Msg. This pattern is required so that the type checker can unambiguously determine that the address component of the

`Msg` should be ignored.

What happens if our e-mail DTD were encoded in Scheme-style? Implicit in the discussion above is the assumption that every newtype has a monotype body. Without this assumption, the technique of using a Glushkov automaton to convert from XML to λ^{TIR} syntax breaks down. To see why, consider the XML fragment:

```
<Body><P>The...</P><P>All...</P></Body>
```

Clearly we intend this to denote the λ^{TIR} term:

```
Body [P "The...", P "All... "]
```

However, all the type checker knows about `Body` and `P` is that:

```
newtype Body = \(a : Type) . a
newtype P = \(a : Type) . a
```

Thus, the above XML term could also denote the λ^{TIR} term

```
Body (P "The...", P "All...")
```

or

```
Body (P ["The..."], P ["All..."])
```

or indeed any one of a countably infinite set of λ^{TIR} terms.

To avoid this ambiguity as simply as possible, XMλ requires the above XML term to be written as:

```
<Body (P*)><P String>The...</P><P String>All...</P></Body>
```

Notice how the newtypes `Body` and `P` were explicitly instantiated with *type* arguments. These arguments tell the type checker exactly which monotype each element should belong to.

Of course this is far from convenient. Hence in practice the programmer should use the DTD-style of encoding as much as is feasible, and introduce type abbreviations where required:

```
newtype Body = \(a : Type) . a
newtype P = \(a : Type) . a

type BodyT = Body (P*)
type PT = P String

<BodyT><PT>The...</PT><PT>All...</PT></BodyT>
```

3.5 Overloading

As our final example, we show how equality constraints may be exploited to allow identifiers to be overloaded with multiple definitions.

There are two approaches to overloading an identifier `x`. The *open-world* view, as adopted in Haskell's class system [109], assumes the multiple definitions for `x` are all instances

of a common type scheme σ, but otherwise makes no assumptions about any particular definition. Hence, a new definition for x may be added without the need to recompile programs using x. This approach is most conveniently implemented by passing definitions as implicit parameters at *run-time* [47].

In contrast, the *closed-world* view, as adopted for method-overloading in Java [34] and many other object-oriented languages, assumes all definitions for x are known at each point of use, but otherwise only requires each definition to be *at a distinct type*. (Of course Java has a notion of subtyping which has no counterpart in λ^{TIR}, hence our examples are simpler.) Closed-world overloading is typically implemented by selecting the appropriate definition at *compile-time*. Hence, adding a new definition for x requires recompiling all programs using x, but there is no associated run-time cost.

We now show that λ^{TIR} is able to express closed-world-style overloading. In conjunction with *implicit parameters* [57], an open-world style of overloading is also possible, though unfortunately outside the scope of this thesis.

For a classic example, assume we have two addition functions:

```
intPlus : Int -> Int -> Int
realPlus : Real -> Real -> Real
```

To overload + on both these definitions, we first build a TIP containing them:

```
let allPlus
  : All ((Int -> Int -> Int) #
         (Real -> Real -> Real) # Empty)
  = (intPlus && realPlus && Triv)
```

We then define + to project one element from allPlus:

```
let (+)
  : forall (a : Type) (b : Row) .
      a ins b,
      (a # b) eq
        ((Int -> Int -> Int) #
         (Real -> Real -> Real) # Empty) => a
  = (\(x && _) . x) allPlus
```

(This type scheme is actually inferred and need not be supplied by the programmer.)

Because x is used polymorphically in the λ-abstraction `\(x && _) . x`, the type inferencer cannot determine which of `Int -> Int -> Int` and `Real -> Real -> Real` should unify with its type a. Hence this equality constraint, and the membership constraint arising from the pattern `(x && _)`, must be deferred.

When typing the term

```
\y . (1 + 1, 1.0 + y)
```

we find it has type

```
e -> (c, f)
```

subject to the constraints introduced by each use of +:

```
(Int -> Int -> c) ins d,
((Int -> Int -> c) # d) eq
  ((Int -> Int -> Int) # (Real -> Real -> Real) # Empty),
(Real -> e -> f) ins g,
((Real -> e -> f) # g) eq
  ((Int -> Int -> Int) # (Real -> Real -> Real) # Empty)
```

The simplifier reduces this constraint to `true`, with the bindings:

$$\left[\begin{array}{l}\texttt{c} \mapsto \texttt{Int}, \texttt{d} \mapsto \texttt{Real -> Real -> Real \# Empty}, \\ \texttt{e} \mapsto \texttt{Real}, \texttt{f} \mapsto \texttt{Real}, \texttt{g} \mapsto \texttt{Int -> Int -> Int \# Empty}\end{array}\right]$$

Hence, the final inferred type is

```
Real -> (Int, Real)
```

However, for the term:

```
1.0 + 1
```

we find:

```
error: the constraint
  (Real -> Int -> a # b) eq
    ((Int -> Int -> Int) #
     (Real -> Real -> Real) # Empty)
is unsatisfiable
```

In conventional closed-world overloading, each use of an overloaded identifier must be at a type sufficiently monomorphic to resolve the overloading statically. λ^{TIR} lifts this restriction. For example, consider defining `nList` to form a list of between 1 and 3 arguments:

```
let allNList
  : forall (a : Type) .
      All ((a -> List a) #
           (a -> a -> List a) #
           (a -> a -> a -> List a) # Empty)
  = ((\x . [x]) &&
     (\x y . [x, y]) &&
     (\x y z . [x, y, z]) && Triv)

let nList
  : forall (a : Type) (b : Type) (c : Row) .
      b ins c,
      (b # c) eq
        ((a -> List a) #
         (a -> a -> List a) #
         (a -> a -> a -> List a) # Empty) => b
  = (\(x && _) . x) allNList
```

We may now specialze `nList` to `oneList`, which will append at most one more integer to

[1, 2]:

```
let oneList
  : forall (a : Type) (d : Type) (e : Row) .
      (Int -> Int -> d) ins ((a -> List a) # e),
      ((Int -> Int -> d) # e) eq
        ((a -> a -> List a) # (a -> a -> a -> List a) # Empty) => d
  = nList 1 2
```

Notice how `oneList` is still overloaded, but "less so" than `nList`.

The overloading of `oneList` is finally fully resolved in the program:

```
oneList ++ oneList 3 : List Int
```

which reduces to `[1, 2, 1, 2, 3]`.

This last example highlights the limitations of the simplifier. One may expect `oneList` to have the simpler type:

```
forall (d : Type) (f : Row) .
  d ins f,
  (d # f) eq ((List Int) # (Int -> List Int) # Empty) => d
```

Unfortunately, the simplifier is not powerful enough to determine that a must be `Int`, and cannot "project" away the common type `Int -> Int -> _` in order to reduce the first constraint to the second. Perhaps worse, if the programmer were to supply the above scheme as an annotation, the system would be unable to show that the second constraint entails the first, because the row variables `e` and `f` do not appear within the result type `d` of the two schemes and so cannot be related. Hence, this more sophisticated style of type-based overloading may surprise the novice programmer.

An aggressive λ^{TIR} compiler could inline `allPlus` and `allNList`, and perform β-reduction of the projection functions where indices are constant. Hence, λ^{TIR} couples some of the flexibility of open-world overloading with the efficiency of closed-world overloading.

Chapter 4

Type Checking

This section begins our formal development of λ^{TIR}. We'll introduce its syntax and kind system, and present the well-typing judgement. Well-typing requires the notions of constraint entailment, which in turn is built from a notion of type order. We conclude by demonstrating the soundness of our type system w.r.t. a simple denotational semantics.

4.1 Syntax

Figure 4.1 presents the kinds, types and terms of the source language, most of which should be familiar from examples. Our presentation is made more uniform if we allow higher-kinds, type abstraction and type application, though care will be taken to avoid the need for higher-order unification. For simplicity the only base type is `Int`.

The empty constraint will be written as `true`, and a generic unsatisfiable constraint as `false`, though neither may appear explicitly within programs. We write $C +\!\!+ D$ to denote concatenation of the primitive constraints of C and D. Equality constraints are only allowed at kind `Type` or `Row`; we'll usually elide their annotation. As is customary, we identify the type scheme `forall` $\cdot$ `. true =>` τ with the type τ.

We allow λ-abstractions to contain patterns, which may be nested arbitrarily. We assume all pattern variables to be distinct, and will also assume no type or term variable binding ever shadows another. We identify the unitary discriminator `{` *abs* `}` with *abs*.

In much of what follows we assume types and terms are represented in applicative form. For example, τ `->` υ is represented by the application `(_ -> _)` τ υ. Furthermore, we assume the binary operator `(_ # _)` to be generalised to a family of $(n+1)$-ary row-consing operators $(\#)_n$ for $n \geq 0$, so that τ_1 `#` $\ldots$ `#` τ_n `#` l may be represented by the single application $(\#)_n\ \tau_1 \ldots \tau_n\ l$. We also identify $(\#)_0\ l$ with l. Figure 4.2 defines F and G to range over all type constructors, and f and g to range over all term constructors.

We shall write $\overline{\tau}$ to denote $\tau_1 \ldots \tau_n$, and $\overline{\tau}_{\setminus i}$ to denote $\tau_1 \ldots \tau_{i-1}\ \tau_{i+1} \ldots \tau_n$; n will typically be clear from context. Many other constructs shall be similarly overlined. For example, we write $\Delta \vdash \overline{\tau : \kappa}$ as shorthand for:

$$\Delta \vdash \tau_1 : \kappa_1 \wedge \ldots \wedge \Delta \vdash \tau_n : \kappa_n$$

The λ^{TIR} type language forms a strongly normalising simply-typed λ-calculus with constants. We let Δ range over *kind-contexts* (mapping type variables to kinds), and let Δ_{init} denote the initial kind context given in Figure 4.3. Figure 4.4 defines the *well-kinding*

Kinds	$\kappa ::= \texttt{Type} \mid \texttt{Row} \mid \kappa_1 \texttt{ -> } \kappa_2$	
Type variables	$a, b ::= \texttt{a}, \texttt{b}, \ldots$	
Newtype names	$A, B ::= \texttt{A}, \texttt{B}, \ldots$	
Types	$\tau, \upsilon, \rho ::= \texttt{Int} \mid \upsilon \texttt{ -> } \tau$	
	$\mid \texttt{Empty} \mid \tau \texttt{ \# } \rho \mid \texttt{One } \rho \mid \texttt{All } \rho$	
	$\mid A \mid a \mid \texttt{\textbackslash(}a \texttt{ : } \kappa\texttt{) . } \tau \mid \tau\ \upsilon$	
Row tails	$l ::= \texttt{Empty} \mid a$	
Type var context	$\Delta ::= a_1 : \kappa_1, \ldots, a_n : \kappa_n$	$n \geq 0$
Primitive constraints	$c, d ::= \tau\ \textbf{\textit{ins}}\ \rho \mid \tau\ \textbf{\textit{eq}}_\kappa\ \upsilon$	$\kappa \in \{\texttt{Type}, \texttt{Row}\}$
Constraints	$C, D, E ::= c_1, \ldots, c_n$	$n \geq 0$
Type schemes	$\sigma ::= \texttt{forall } \Delta \texttt{ . } C \texttt{ => } \tau$	
Integers	i	
Variables	$x, y, z ::= \texttt{x}, \texttt{y}, \texttt{z}, \ldots$	
Abstractions	$abs ::= \texttt{\textbackslash}p \texttt{ . } t$	
Terms	$t, u ::= i \mid A \mid \texttt{Inj} \mid t \texttt{ \&\& } u \mid \texttt{Triv}$	
	$\mid t\ u \mid x \mid \texttt{\{}abs_1 \texttt{ ; } \ldots \texttt{ ; } abs_n\texttt{\}}$	$n > 0$
	$\mid \texttt{let } x \texttt{ = } u \texttt{ in } t$	
Patterns	$p, q ::= i \mid A\ p \mid \texttt{Inj } p \mid p \texttt{ \&\& } q \mid \texttt{Triv} \mid x$	
Newtype decls	$tdecl ::= \texttt{newtype } \{\text{opaque}\}^{opt}\ A \texttt{ = } \tau$	
Programs	$prog ::= tdecl_1 \ldots tdecl_n\ t$	$n \geq 0$

Figure 4.1: Syntax of λ^{TIR} kinds, types and terms

$$F, G ::= \texttt{Int} \mid \texttt{(_ -> _)} \mid \texttt{Empty} \mid \texttt{(_ \# _)} \mid \texttt{(One _)} \mid \texttt{(All _)} \mid A$$
$$f, g ::= \texttt{(Inj _)} \mid \texttt{Triv} \mid \texttt{(_ \&\& _)} \mid A$$

Figure 4.2: λ^{TIR} type and term constructors

$$\begin{aligned}\Delta_{const} = \quad & \texttt{Int} : \texttt{Type}, \\ & \texttt{Empty} : \texttt{Row}, \\ & \texttt{(_ \# _)} : \texttt{Type -> Row -> Row}, \\ & \texttt{(One _)} : \texttt{Row -> Type}, \\ & \texttt{(All _)} : \texttt{Row -> Type}, \\ & \texttt{(_ -> _)} : \texttt{Type -> Type -> Type}\end{aligned}$$

$\Delta_{init} = \Delta_{const} +\!\!+ \ \{A_i : \kappa_i \mid (\texttt{newtype } \{\text{opaque}\}^{opt}\ A_i \texttt{ = } \tau_i) \in tdecls\}$
such that $\forall i \ .\ \Delta_{init} \vdash \tau_i : \kappa_i$
$\wedge\ \kappa_i = \kappa'_1 \texttt{ -> } \ldots \texttt{ -> } \kappa'_m \texttt{ -> Type}$
$\wedge\ \forall j \ .\ \kappa'_j \in \{\texttt{Type}, \texttt{Row}\}$
$\wedge$ every cycle involving A passes through at least one `All`/`One` constructor

Figure 4.3: Initial λ^{TIR} type var context Δ_{init}

$$\boxed{\Delta \vdash \tau : \kappa}$$

$$\frac{(a/F : \kappa) \in \Delta}{\Delta \vdash a/F : \kappa}$$

$$\frac{\Delta, a : \kappa' \vdash \tau : \kappa}{\Delta \vdash \texttt{\textbackslash(}a \texttt{ : } \kappa'\texttt{) . } \tau : \kappa' \texttt{ -> } \kappa} \qquad \frac{\Delta \vdash \tau : \kappa' \texttt{ -> } \kappa \quad \Delta \vdash \upsilon : \kappa'}{\Delta \vdash \tau\, \upsilon : \kappa}$$

$$\boxed{\Delta \vdash \tau \text{ constraint}}$$

$$\frac{\Delta \vdash \tau : \kappa \quad \Delta \vdash \upsilon : \kappa \quad \kappa \in \{\mathsf{Type}, \mathsf{Row}\}}{\Delta \vdash \tau\ \textit{eq}_\kappa\ \upsilon \text{ constraint}} \qquad \frac{\Delta \vdash \tau : \mathsf{Type} \quad \Delta \vdash \rho : \mathsf{Row}}{\Delta \vdash \tau\ \textit{ins}\ \rho \text{ constraint}}$$

$$\frac{\Delta \vdash \overline{c \text{ constraint}}}{\Delta \vdash \overline{c} \text{ constraint}}$$

$$\boxed{\Delta \vdash \sigma \text{ scheme}}$$

$$\frac{\overline{\kappa \in \{\mathsf{Type}, \mathsf{Row}\}} \quad \Delta \mathbin{+\!\!+} \overline{a : \kappa} \vdash C \text{ constraint} \quad \Delta \mathbin{+\!\!+} \overline{a : \kappa} \vdash \tau : \mathsf{Type}}{\Delta \vdash \texttt{forall}\ \overline{a : \kappa}\ \texttt{.}\ C \texttt{ => } \tau \text{ scheme}}$$

$$\boxed{\Delta \vdash \Gamma \text{ context}}$$

$$\frac{\Delta \vdash \overline{\sigma \text{ scheme}}}{\Delta \vdash \overline{x : \sigma} \text{ context}}$$

Figure 4.4: Well-kinded λ^{TIR} types, constraints, type schemes and type contexts

judgement $\Delta \vdash \tau : \kappa$, and its extension to constraints, schemes and type contexts. Both sides of an equality constraint must have the same kind; insertion constraints must be with a Type and a Row. Type schemes must have a body type of kind Type, and each universally quantified type variable must have kind Row or Type.

We let θ ranges over substitutions, which are idempotent maps from type variables to types or rows, and which are the identity on all but a finite set of type variables. We also write **Id** to denote the identity substitution.

Define the judgement $\Delta \vdash \theta$ subst to be true iff $dom(\theta) \subseteq dom(\Delta)$ and $\forall (a : \kappa) \in \Delta \,.\, \Delta \vdash \theta\ a : \kappa$.

Similarly, define $\vdash \theta : \Delta \rightarrow \Delta'$ to be true iff $dom(\theta) = dom(\Delta)$ and $\forall (a : \kappa) \in \Delta \,.\, \Delta' \vdash \theta\ a : \kappa$. Notice the strict equality on domains. Clearly, because substitutions are idempotent, Δ and Δ' must be disjoint.

We shall write $\theta_{\restriction S}$ to denote the restriction of θ to the domain S. Similarly, $\theta_{\backslash a}$ denotes θ restricted to all type variables except a. We shall use the same notation for restricting the domains of other maps, such as environments.

Every recursive newtype must be *well-founded*; *viz* every cycle passing through a new-

$$
\begin{aligned}
named(\overline{c}) &= \overline{w : c} && \text{where } \overline{w} \text{ fresh}\\
names(\overline{w : c}) &= (\overline{w})\\
anon(\overline{w : c}) &= \overline{c}\\[1em]
inheritable(C) &= \mathbf{tt}\\[1em]
norm(F) &= F\\
norm(a) &= a\\
norm(\backslash(a : \kappa) \ . \ \tau) &= \backslash(a : \kappa) \ . \ \tau\\
norm(\tau \ v) &= \begin{cases} norm(\tau'[a \mapsto v]), & \textbf{if } norm(\tau) = \backslash(a : \kappa) \ . \ \tau' \\ F \ v'_1 \ldots v'_n \ norm(v), & \textbf{if } norm(\tau) = F \ v'_1 \ldots v'_n \end{cases}\\[1em]
eqs(C) &= \{\tau \ \mathbf{eq} \, v \mid (\tau \ \mathbf{eq} \, v) \in C\}\\
inss(C) &= \{w : \tau \ \mathbf{ins} \, v \mid (w : \tau \ \mathbf{ins} \, v) \in C\}\\
inhs(C) &= \{(w : c) \in C \mid inheritable(c)\}
\end{aligned}
$$

Figure 4.5: Definitions of functions *named*, *names*, *anon*, *inheritable*, *norm*, *eqs*, *inss* and *inhs*

type must also pass through at least one `All` or `One` type constructor. This restriction is necessary because newtype declarations such as:

```
newtype A = B
newtype B = A
```

cannot be given a semantics in the model to be presented in Section 4.5.

Figure 4.5 collects some ancillary definitions. Some judgements require *constraint contexts* in which every primitive constraint is associated with a unique index variable. The function $names(C)$ associates fresh witness names with each primitive constraint in C. The function $named(C)$ is the tuple of witness names of C, and shall be used when constructing run-time terms; $anon(C)$ is C with all witness names removed.

We write $norm(\tau)$ to denote the *β-normal form* for a type τ of kind `Type`. Newtype names are considered as free variables for the purpose of normalisation.

We let $eqs(C)$ be the primitive equality constraints of C, and $inss(C)$ be the primitive insertion constraints. We let $inhs(C)$ be only the *inheritable* primitive constraints of C. In this dissertation, $inheritable(C)$ is defined to be the constant **tt** (true) function. If λ^{TIR} were extended with implicit parameters [57], $inheritable(C)$ would be redefined to be **ff** (false) if C contains implicit-parameter constraints. However, much of the remainder of the system, and its proofs of correctness, would remain unchanged.

We let τ^m and v^m range over all normalised monotypes of kind `Type` or `Row`.

Figure 4.6 presents the syntax of the untyped run-time language—the target of our type-directed translation. Parts of this syntax have already been introduced in Section 2.6.

TIP's are represented as ordered tuples $\langle T_1, \ldots, T_n \rangle$. TIC's are a pair $\mathsf{Inj} \ W \ T$ of an index and a run-time term. Each declared newtype `A` is represented by an injector A, and corresponding extractor A^{-1}. Though both these terms would be the identity in any operational semantics, they shall be important when we consider a model for λ^{TIR} in Section 4.5.

Index vars	w	$::= \mathsf{w}, \ldots$	
Indices	W	$::= w \mid \mathsf{One} \mid \mathsf{Inc}\ W \mid \mathsf{Dec}\ W \mid \mathsf{True}$	
Bindings	B	$::= w_1 = W_1, \ldots, w_n = W_n$	$n \geq 0$
Variables	x, y, z	$::= \mathsf{x}, \mathsf{y}, \mathsf{z}, \ldots$	
Terms	T, U	$::= i \mid \langle T_1, \ldots, T_n \rangle \mid \mathsf{Inj}\ W\ T$	$n \geq 0$
		$\mid \lambda x\ .\ T \mid \lambda(w_1, \ldots, w_n)\ .\ T$	$n \geq 0$
		$\mid T\ U \mid T\ (W_1, \ldots, W_n) \mid x \mid A \mid A^{-1}$	$n \geq 0$
		$\mid \mathsf{insert}\ U\ \mathsf{at}\ W\ \mathsf{into}\ T \mid \mathsf{let}\ \langle\rangle = U\ \mathsf{in}\ T$	
		$\mid \mathsf{let}\ x\ y = \mathsf{remove}\ W\ \mathsf{from}\ U\ \mathsf{in}\ T$	
		$\mid \mathsf{case}\ U\ \mathsf{of}\ \{\ \mathsf{Inj}\ W\ x \rightarrow T_1;\ \mathsf{otherwise} \rightarrow T_2\}$	
		$\mid \mathsf{case}\ U\ \mathsf{of}\ \{i \rightarrow T_1\ ; \mathsf{otherwise} \rightarrow T_2\}$	
		$\mid \mathsf{let}\ x = U\ \mathsf{in}\ T \mid \mathsf{letw}\ B\ \mathsf{in}\ T$	

Figure 4.6: Syntax of λ^{TIR} run-time terms

$$\begin{aligned}
\Gamma_{const} = \ & \texttt{(Inj _)} : \texttt{forall (a : Type), (b : Row) . a}\ \textit{ins}\ \texttt{b => a -> One (a \# b)},\\
& \texttt{(_ \&\& _)} : \texttt{forall (a : Type), (b : Row) . a}\ \textit{ins}\ \texttt{b => a -> All b -> All (a \# b)},\\
& \texttt{Triv} : \texttt{All Empty}
\end{aligned}$$

$$\Gamma_{init} = \Gamma_{const} +\!\!+ \left\{ \begin{array}{l} A : \texttt{forall}\ a_1 : \kappa_1, \ldots, a_n : \kappa_n\ .\ norm(\tau\ a_1 \ldots a_n)\ \texttt{->}\ A\ a_1 \ldots a_n \\ \quad \mid A : \kappa_1\ \texttt{->}\ \ldots\ \texttt{->}\ \kappa_n\ \texttt{->}\ \texttt{Type} \in \Delta_{init}, \\ \quad \mid (\texttt{newtype}\ \{\texttt{opaque}\}^{opt}\ A = \tau) \in tdecls \end{array} \right\}$$

Figure 4.7: Initial λ^{TIR} type context Γ_{init}

We keep indices separate from run-time terms to simplify our soundness proof. One is the first index, and Inc W and Dec W offset index W by one position to the right or left. Indices are abstracted and passed in tuples, and may be let-bound by letw B in T. The "index" True witnesses the satisfaction of an equality constraint. It plays no part in an implementation, but makes the proofs of correctness more uniform.

The term $\mathsf{let}\ \langle\rangle = U\ \mathsf{in}\ T$ forces evaluation of U. In the term $\mathsf{let}\ x\ y = \mathsf{remove}\ W\ \mathsf{from}\ U\ \mathsf{in}\ T$, x is bound to the term at index W in U, and y to the remaining product. The first case-form checks if U evaluates to a TIC with index W. The second simply checks for matching integers.

4.2 Well-typed Terms

We let Γ range over *type-contexts* (mapping variables to type schemes) and let Γ_{init} denote the initial type context defined in Figure 4.7.

Figure 4.8 presents the rules for deciding the *well-typing judgement* $\Delta \mid C \mid \Gamma \vdash t : \tau \hookrightarrow T$, with intended interpretation:

> "Term t has type τ, and translates to the run-time term T, assuming the free term variables typed in Γ, the free type variables kinded in Δ, the satisfiability

$$\boxed{\Delta \mid C \mid \Gamma \vdash t : \tau \hookrightarrow T}$$

$$\frac{}{\Delta \mid C \mid \Gamma \vdash i : \mathsf{Int} \hookrightarrow i}\ \text{INT}$$

$$\frac{\begin{array}{c}\Delta \mid C \mid \Gamma \vdash t : v \hookrightarrow T \\ \Delta \mid C \mid \Gamma \vdash u : v' \hookrightarrow U \quad C \vdash^{e} v\ eq_{\mathsf{Type}}\ (v' \texttt{ -> } \tau) \hookrightarrow \mathsf{True}\end{array}}{\Delta \mid C \mid \Gamma \vdash t\ u : \tau \hookrightarrow T\ U}\ \text{APP}$$

$$\frac{\begin{array}{c}(x/f : \texttt{forall}\ \overline{a : \kappa}\ .\ D \texttt{ => } \tau) \in \Gamma \\ D' = named(D) \quad \Delta \vdash \overline{v : \kappa} \\ C \vdash^{e} D'[\overline{a \mapsto v}] \hookrightarrow B\end{array}}{\begin{array}{c}\Delta \mid C \mid \Gamma \vdash x/f : \tau[\overline{a \mapsto v}] \\ \hookrightarrow \mathsf{letw}\ B\ \mathsf{in}\ x/f\ names(D')\end{array}}\ \text{VAR}$$

$$\frac{\Delta \mid C \mid \Gamma \vdash_1 abs : \tau \hookrightarrow T[\bullet]}{\Delta \mid C \mid \Gamma \vdash \texttt{\{}abs\texttt{\}} : \tau \hookrightarrow T[\mathsf{undefined}]}\ \text{ABS}$$

$$\frac{\begin{array}{c}\Delta \mid C \mid \Gamma \vdash_1 abs_1 : \tau \hookrightarrow T[\bullet] \\ \Delta \mid C \mid \Gamma \vdash \texttt{\{}abs_2, \ldots, abs_{n+1}\texttt{\}} : \tau' \hookrightarrow U \\ C \vdash^{e} \tau\ eq_{\mathsf{Type}}\ \tau' \hookrightarrow \mathsf{True} \quad z\ \mathsf{fresh}\end{array}}{\begin{array}{c}\Delta \mid C \mid \Gamma \vdash \texttt{\{}abs_1, \ldots, abs_{n+1}\texttt{\}} : \tau \\ \hookrightarrow \mathsf{let}\ z = U\ \mathsf{in}\ T[z]\end{array}}\ \text{DISC}$$

$$\frac{\begin{array}{c}x \in fv(t) \quad \Delta \vdash D_1\ \mathsf{constraint} \quad \Delta +\!\!+ \Delta' \vdash D_2\ \mathsf{constraint} \\ D_1 = inhs(C) \quad saturate(D_1 +\!\!+ D_2) \neq \emptyset \quad \sigma = \texttt{forall}\ \Delta'\ .\ anon(D_2) \texttt{ => } v \\ \Delta +\!\!+ \Delta' \mid D_1 +\!\!+ D_2 \mid \Gamma \vdash u : v \hookrightarrow U \\ \Delta \mid C \mid \Gamma, x : \sigma \vdash t : \tau \hookrightarrow T\end{array}}{\begin{array}{c}\Delta \mid C \mid \Gamma \vdash \texttt{let}\ x \texttt{ = } u\ \texttt{in}\ t : \tau \\ \hookrightarrow \mathsf{let}\ x = \lambda names(D_2)\ .\ U\ \mathsf{in}\ T\end{array}}\ \text{LET}$$

Figure 4.8: Well-typed λ^{TIR} terms

of the constraint context C, and the free index variables of C."

We intend the VAR rule to apply to variables (ranged over by x), and constants and newtypes (ranged over by f).

Note that, as discussed in Section 2.9, the LET rule must check not only that the constraint for a let-bound term is well-kinded, but also that it is *satisfiable*, and that the let-bound variable appears free in the let body. The test for satisfiability uses the *saturate* function, which will be defined in Section 4.4.

The LET rule contains an additional subtlety. Typically, all the constraints of C would be available when type checking u. However, in a system with implicit parameter constraints [57], any implicit parameters within C must be removed when checking u. This restriction is necessary to force any implicit parameters within u to appear within D_2, and thus ensure

$$\boxed{\Delta \mid C \mid \Gamma \vdash_n t : \tau \hookrightarrow T[\bullet]}$$

$$\frac{\Delta \mid C \mid \Gamma \vdash t : \tau \hookrightarrow T}{\Delta \mid C \mid \Gamma \vdash_0 t : \tau \hookrightarrow T}\ \text{P1}$$

$$\frac{\Delta \mid C \mid \Gamma \vdash_n t : \tau \hookrightarrow T[\bullet] \quad x \text{ fresh}}{\begin{array}{c}\Delta \mid C \mid \Gamma \vdash_{n+1} \texttt{\\}i\ \texttt{.}\ t : \texttt{Int -> } \tau \\ \hookrightarrow \lambda x\ .\ \mathsf{case}\ x\ \mathsf{of}\ \{i \to T[\bullet\ x]\,;\mathsf{otherwise} \to \bullet\ x\}\end{array}}\ \text{P2}$$

$$\frac{\begin{array}{c}(\texttt{newtype}\ \{\text{opaque}\}^{opt}\ A = v') \in tdecls \\ (A : \kappa_1 \texttt{ -> } \ldots \texttt{ -> } \kappa_n \texttt{ -> Type}) \in \Delta_{init} \\ \Delta \vdash \overline{v : \kappa} \quad C \vdash^e norm(v'\ v_1 \ldots v_n)\ eq_{\text{Type}}\ \tau' \hookrightarrow \text{True} \\ \Delta \mid C \mid \Gamma \vdash_{n+1} \texttt{\\}p\ \texttt{.}\ t : (\tau' \texttt{ -> } \tau) \hookrightarrow T[\bullet] \quad x, y \text{ fresh}\end{array}}{\begin{array}{c}\Delta \mid C \mid \Gamma \vdash_{n+1} \texttt{\\(}A\ p\texttt{)}\ \texttt{.}\ t : A\ v_1 \ldots v_n \texttt{ -> } \tau \\ \hookrightarrow \lambda x\ .\ \mathsf{let}\ y = A^{-1}\ x\ \mathsf{in}\ T[\lambda y\ .\ \bullet\ (A\ y)]\ y\end{array}}\ \text{P3}$$

$$\frac{\begin{array}{c}\Delta \mid C \mid \Gamma \vdash_{n+1} \texttt{\\}p\ \texttt{.}\ t : (v \texttt{ -> } \tau) \hookrightarrow T[\bullet] \\ \Delta \vdash \rho : \texttt{Row} \quad C \vdash^e v\ \textit{\textbf{ins}}\ \rho \hookrightarrow W \quad x, y \text{ fresh}\end{array}}{\begin{array}{c}\Delta \mid C \mid \Gamma \vdash_{n+1} \texttt{\\(Inj}\ p\texttt{)}\ \texttt{.}\ t : \texttt{One}\ (v\ \texttt{\#}\ \rho) \texttt{ -> } \tau \\ \hookrightarrow \lambda x\ .\ \mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{Inj}\ W\ y\ \to T[\lambda y\ .\ \bullet\ (\mathsf{Inj}\ W\ y)]\ y; \\ \mathsf{otherwise} \to \bullet\ x\}\end{array}}\ \text{P4}$$

$$\frac{\begin{array}{c}\Delta \mid C \mid \Gamma \vdash_{n+2} \texttt{\\}p\ \texttt{.}\ \texttt{\\}q\ \texttt{.}\ t : (v_1 \texttt{ -> } v_2 \texttt{ -> } \tau) \hookrightarrow T[\bullet] \\ C \vdash^e \texttt{(All}\ \rho\texttt{)}\ eq_{\text{Type}}\ v_2 \hookrightarrow \text{True} \quad \Delta \vdash \rho : \texttt{Row} \quad C \vdash^e v_1\ \textit{\textbf{ins}}\ \rho \hookrightarrow W \quad x, y, z \text{ fresh}\end{array}}{\begin{array}{c}\Delta \mid C \mid \Gamma \vdash_{n+1} \texttt{\\(}p\ \texttt{\&\&}\ q\texttt{)}\ \texttt{.}\ t : \texttt{All}\ (v_1\ \texttt{\#}\ \rho) \texttt{ -> } \tau \\ \hookrightarrow \lambda x\ .\ \mathsf{let}\ y\ z = \mathsf{remove}\ W\ \mathsf{from}\ x \\ \mathsf{in}\ T[\lambda y\ .\ \lambda z\ .\ \bullet\ (\mathsf{insert}\ y\ \mathsf{at}\ W\ \mathsf{into}\ z)]\ y\ z\end{array}}\ \text{P5}$$

$$\frac{\Delta \mid C \mid \Gamma \vdash_n t : \tau \hookrightarrow T[\bullet] \quad x \text{ fresh}}{\begin{array}{c}\Delta \mid C \mid \Gamma \vdash_{n+1} \texttt{\\Triv}\ \texttt{.}\ t : \texttt{All Empty -> } \tau \\ \hookrightarrow \lambda x\ .\ \mathsf{let}\ \langle\rangle = x\ \mathsf{in}\ T[\bullet\ x]\end{array}}\ \text{P6}$$

$$\frac{\Delta \vdash v : \text{Type} \quad \Delta \mid C \mid \Gamma, x : v \vdash_n t : \tau \hookrightarrow T[\bullet]}{\Delta \mid C \mid \Gamma \vdash_{n+1} \texttt{\\}x\ \texttt{.}\ t : (v \texttt{ -> } \tau) \hookrightarrow \lambda x\ .\ T[\bullet\ x]}\ \text{P7}$$

Figure 4.9: Well-typed λ^{TIR} pattern abstractions

they are dynamically rather than lexically scoped. For λ^{TIR}, we abstract from this by using the predicate *inheritable* (defined in Figure 4.5). We intend *inheritable*(c) to be ff if c should be removed from C when checking u. Thus if λ^{TIR} were extended with implicit parameters, we would define *inhertable*$(?x : \tau +\!\!+ C) = \mathbf{ff}$.

Notice the symmetry of index abstraction in the LET rule and index application in the VAR rule.

The ABS and DISC rules both make use of the mutually recursively defined pattern compiler of Figure 4.9. The subscript n is the number of λ-abstractions of t to be compiled as patterns, and $T[\bullet]$ is the compiled run-time term with a "hole," $\bullet$, which should be filled by a term (of the same type) to evaluate should the pattern fail. The ABS rule fills the hole with undefined, since there is no other alternative to try. The DISC rule chains together each discriminant such that failure of abs_i will cause abs_{i+1} to be tried. Notice the use of a let binding within the run-time code generated by the DISC rule to prevent code size explosion.

Note than a "vanilla" λ-abstraction `\`x `.` t is typed by treating it as a singleton discriminator `{\`x `.` t`}` in the ABS rule. This discriminator in turn invokes the pattern rule P7 to remove the argument x, and then the rule P1 for the body t, which then continues in the well-typing judgement.

As a term is deconstructed, the pattern compiler must insert re-construction code so that failure will be handled correctly. A real compiler will attempt to β-reduce pattern code once the hole has been filled.

At the heart of all these rules is the entailment judgement, $\vdash^e$, to be presented in Section 4.4. It is used in three ways:

(i) When two types must be equivalent (*e.g.*, in the APP and DISC rules) the type checker asks if the current constraint context entails their equality.

(ii) Whenever a row is constructed or pattern-matched (*e.g.*, in the P4 and P5 rules), the row must be well-formed (the insertion constraint satisfied), and an index must be known at run-time. The type checker thus asks if the current constraint context entails the membership constraint. If so, the entailment judgement yields the index W.

(iii) Each variable occurrence propagates any constraints from the variable's definition-site to the use-site. In the VAR rule, the type checker thus asks if the current constraint context entails the variable's constraints, suitably specialised.

We assume the following definitions for the source-language constants in the run-time language:

$$\begin{aligned}
(\texttt{Inj _}) &= \lambda(\mathsf{w})\ .\ \lambda\mathsf{x}\ .\ \mathsf{Inj\ w\ x} \\
(\texttt{Triv}) &= \langle\rangle \\
(\texttt{_ \&\& _}) &= \lambda(\mathsf{w})\ .\ \lambda\mathsf{x}\ .\ \lambda\mathsf{y}\ .\ \mathsf{insert\ x\ at\ w\ into\ y} \\
\texttt{A} &= \lambda\mathsf{x}\ .\ \mathsf{A\ x}
\end{aligned}$$

Notice these definitions match the types for these constructors given in Figure 4.7.

$$lexleq^m([], []) = \mathbf{tt}$$

$$lexleq^m(F :: r, G :: r') = \begin{cases} \mathbf{tt}, & \textbf{if } F <^F G \\ \mathbf{ff}, & \textbf{if } G <^F F \\ lexleq^m(r, r'), & \textbf{otherwise} \end{cases}$$

$$preorder^m_O(F\ \overline{\tau^m}) = \begin{cases} [F], & \textbf{if } F \in O \\ F :: preorder^m_O(\tau^m_1) +\!\!+ \ldots +\!\!+ preorder^m_O(\tau^m_n), & \textbf{otherwise} \end{cases}$$

$$preorder^m_O((\texttt{\#})_n\ \overline{\tau^m}\ \texttt{Empty}) = (\texttt{\#})_n :: preorder^m_O(\tau^m_{\pi 1}) +\!\!+ \ldots +\!\!+ preorder^m_O(\tau^m_{\pi n})$$
$$\text{where } \pi \text{ is a permutation on } n \text{ s.t.}$$
$$\forall i, j \ .\ i \leq j \implies leq^m_O(\tau^m_{\pi i}, \tau^m_{\pi j})$$

$$leq^m_O(\tau^m, v^m) = lexleq^m(preorder^m_O(\tau^m), preorder^m_O(v^m))$$

$$eq^m_O(\tau^m, v^m) = leq^m_O(\tau^m, v^m) \wedge leq^m_O(v^m, \tau^m)$$

Figure 4.10: Total order on λ^{TIR} monotypes

4.3 Type Order

This section formalises the notions of type order and equality introduced in Section 2.7. We shall first construct a total order on monotypes, and then show how this order may be extended to a partial order on all types that is stable under substitution.

Let $<^F$ be an arbitrary total order on all type constructors and newtype names. For concreteness, our examples will assume the ordering (where the A_i are the newtypes of the program):

$$\texttt{Int} <^F \texttt{Bool} <^F \texttt{String} <^F (_\texttt{ -> }_) <^F \texttt{Empty} <^F$$
$$(\texttt{One }_) <^F (\texttt{All }_) <^F A_0 <^F \ldots <^F A_n <^F$$
$$(\texttt{\#})_0 <^F (\texttt{\#})_1 <^F \ldots$$

Notice that we have included the type constants `Bool` and `String`, even though these are not included in the formal syntax of Figure 4.1.

Figure 4.10 defines the binary monotype relation, leq^m_O, which is parameterised over a set of type constructors O. This relation is well-defined for any pair of normalised monotypes of kind `Type` or `Row`. Note that because only similarly kinded types need be compared, we could replace leq^m_O with a pair of relations. However, this precision comes at the cost of additional notational complexity.

This relation defines the monotype τ^m to be less-than or equal to v^m, written $leq^m_O(\tau^m, v^m)$, when their pre-order flattenings are lexicographically ordered under $lexleq^m$. The latter uses $<^F$ to order each type constructor. For convenience the definition uses a list-like syntax, where [] is nil and :: cons. Notice that, because each type constructor is both of a fixed arity and saturated, there is no need for $lexleq^m$ to consider the case of unequal length argument lists.

Since the ordering of types should be stable under permutation of row elements, $preorder^m_O$

first sorts a row's elements using leq_O^m before flattening them. In this way we have:

$$leq_O^m(\texttt{Bool \# Int \# Empty}, \texttt{Int \# String \# Empty}) = \texttt{tt}$$

This recursion is well-defined because the row elements are strictly smaller than the row containing them.

In the sequel, we shall instantiate O with either $\emptyset$ or *opaque*, the set of all newtype names declared as opaque. In this way leq_O^m may be used to decide both *transparent* and *opaque* (in)equality. For example, assuming

```
newtype opaque A = \a . a
```

we have

$$leq_{opaque}^m(\texttt{A String \# Bool \# Empty}, \texttt{Bool \# A Int \# Empty})$$

but

$$\neg leq_\emptyset^m(\texttt{A String \# Bool \# Empty}, \texttt{Bool \# A Int \# Empty})$$

The relation eq_O^m, for type equality, is defined in the obvious way.

Fact 4.1 Let $\kappa \in \{\texttt{Type}, \texttt{Row}\}$ and $\Delta_{init} \vdash \tau^m/\upsilon'^m/\upsilon^m : \kappa$. Then

(i) $leq_O^m(\tau^m, \upsilon^m)$ is well-defined.

(ii) leq_O^m is a partial order, *viz* $leq_O^m(\tau^m, \tau^m)$; $leq_O^m(\tau^m, \upsilon'^m)$ and $leq_O^m(\upsilon'^m, \upsilon^m)$ imply $leq_O^m(\tau^m, \upsilon^m)$; $leq_O^m(\tau^m, \upsilon^m)$ and $leq_O^m(\upsilon^m, \tau^m)$ iff τ^m and υ^m are equal up to permutation of row elements and ignoring the arguments of type constructors in O.

(iii) leq_O^m is a total order, *viz* $leq_O^m(\tau^m, \upsilon^m)$ or $leq_O^m(\upsilon^m, \tau^m)$.

We now consider how to lift leq_O^m to all types. The lifted relation is most conveniently expressed as a binary function, cmp_O, into the four-valued set of **lt** (less-than), **gt** (greater-than), **eq** (equal) and **unk** (unknown).

Before plunging into the definition, it is worthwhile to consider what is required. Clearly, cmp_O should agree with leq_O^m on monotypes:

$$cmp_O(\tau^m, \upsilon^m) \in \{\mathbf{lt}, \mathbf{eq}\} \iff leq_O^m(\tau^m, \upsilon^m)$$

However, to ensure soundness of entailment, cmp_O must also be stable under substitution:

$$cmp_O(\tau, \upsilon) = x \wedge x \neq \mathbf{unk} \Longrightarrow cmp_O(\theta\,\tau, \theta\,\upsilon) = x$$

An obvious definition is for cmp_O to yield **unk** whenever its arguments are not monotypes, but definition this is needlessly conservative. Figure 4.11 presents the actual definition, which will yield **unk** only when the possible instantiation of a type variable is significant in deciding the (in)equality of two types. For example, again assuming

```
newtype opaque A = \a . a
```

$$lexcmp^t([],[]) = \mathbf{eq}$$

$$lexcmp^t(a :: r, b :: r') = \begin{cases} \mathbf{lt}, & \mathbf{if}\ a <^{\mathbf{a}} b \\ \mathbf{gt}, & \mathbf{if}\ b <^{\mathbf{a}} a \\ \mathbf{eq}, & \mathbf{otherwise} \end{cases}$$

$$lexcmp^t(a :: r, G :: r') = \mathbf{lt}$$

$$lexcmp^t(F :: r, a :: r') = \mathbf{gt}$$

$$lexcmp^t(F :: r, G :: r') = \begin{cases} \mathbf{lt}, & \mathbf{if}\ F <^F G \\ \mathbf{gt}, & \mathbf{if}\ G <^F F \\ lexcmp^t(r, r'), & \mathbf{otherwise} \end{cases}$$

$$lexcmp^p([],[]) = \mathbf{eq}$$

$$lexcmp^p(a :: r, b :: r') = \begin{cases} \mathbf{eq}, & \mathbf{if}\ a = b \\ \mathbf{unk}, & \mathbf{otherwise} \end{cases}$$

$$lexcmp^p(a :: r, G :: r') = \mathbf{unk}$$

$$lexcmp^p(F :: r, a :: r') = \mathbf{unk}$$

$$lexcmp^p(F :: r, G :: r') = \begin{cases} \mathbf{lt}, & \mathbf{if}\ F <^F G \\ \mathbf{gt}, & \mathbf{if}\ G <^F F \\ lexcmp^p(r, r'), & \mathbf{otherwise} \end{cases}$$

$$preorder^p_O(a) = [a]$$

$$preorder^p_O(F\ \overline{\tau}) = \begin{cases} [F], & \mathbf{if}\ F \in O \\ F :: preorder^p_O(\tau_1) +\!\!+ \ldots +\!\!+ preorder^p_O(\tau_n), & \mathbf{otherwise} \end{cases}$$

$$preorder^p_O((\#)_n\ \overline{\tau}\ l) = l :: (\#)_n :: preorder^p_O(\tau_{\pi\,1}) +\!\!+ \ldots +\!\!+ preorder^p_O(\tau_{\pi\,n})$$

where π is a permutation on n s.t.

$$\forall i, j\ .\ i \leq j \Longrightarrow cmp^t_O(\tau_{\pi\,i}, \tau_{\pi\,j}) = \mathbf{lt}/\mathbf{eq}$$

$$cmp^t_O(\tau, v) = lexcmp^t(preorder^p_O(\tau), preorder^p_O(v))$$

$$cmp_O(\tau, v) = lexcmp^p(preorder^p_O(\tau), preorder^p_O(v))$$

Figure 4.11: Partial ordering on normalized λ^{TIR} types of kind `Type` and `Row`

we have

$$cmp_{opaque}(\texttt{Int}, \texttt{a -> b}) = \mathbf{lt}$$

$$cmp_{opaque}(\texttt{A a}, \texttt{A Int}) = \mathbf{eq}$$

$$cmp_{opaque}(\texttt{Bool -> a}, \texttt{Int -> b}) = \mathbf{gt}$$

$$cmp_{opaque}(\texttt{a -> b}, \texttt{a -> b}) = \mathbf{eq}$$

$$cmp_{opaque}(\texttt{a -> b}, \texttt{Int -> b}) = \mathbf{unk}$$

The relation cmp_O is defined analogously to leq^m_O using a lexicographic ordering, $lexcmp^p$, on a pre-order flattening, $preorder^p_O$. The function $lexcmp^p$ will yield **unk** whenever it encounters a type variable (though the comparison of a type variable against itself may safely yield **eq**).

The treatment of row comparisons involves some subtlety. Firstly, consider how to order the rows `a # d` and `b # c # d`. Since these rows share the same tail `d`, the first will always be smaller than the second, suggesting:

$$cmp_{opaque}(\texttt{a \# d}, \texttt{b \# c \# d}) = \textbf{lt}$$

Furthermore, consider how to compare `Int # Bool # Empty` and `(a -> b) # (c -> d) # Empty`. Even though `(a -> b)` and `(c -> d)` cannot be ordered with respect to each other, each of `Int` and `Bool` may be ordered with respect to `(a -> b)` and `(c -> d)`, suggesting:

$$\begin{aligned} cmp_{opaque}(&\texttt{Int \# Bool \# Empty},\\ &\texttt{(a -> b) \# (c -> d) \# Empty}) = \textbf{lt} \end{aligned}$$

Hence, one row may be less than another even though the elements of one or both rows cannot be ordered amongst themselves. However, any rows with differing tails cannot be ordered, since one or both tails may be instantiated to a row of arbitrary length.

To implement this requires two tricks within $preorder^p_O$. Firstly, a row's tail is placed before both its $(\texttt{\#})_n$ type constructor and its flattened element types. In this way, unequal row tails cause $lexcmp^p$ to yield **unk**. Secondly, the elements of a row are sorted not by cmp_O, but by a *total order*, cmp^t_O, which places type variables before all other type constructors.

We assume $<^a$ is an arbitrary total order on all type variables, which for concreteness we shall take to be lexicographic on the variable's name. The relation cmp^t_O is defined as for cmp_O, but using $lexcmp^t$ to lexicographically order the flattened types instead of $lexcmp^p$. Of course, cmp^t_O is *not* stable under substitution, or even α-conversion! The stability of cmp_O is thus a little subtle.

The following lemma summarises the properties of cmp_O.

Lemma 4.2 Given $\kappa \in \{\texttt{Type}, \texttt{Row}\}$ and $\Delta \vdash \tau/v'/v : \kappa$, then:

(i) $cmp_O(\tau, v)$ is well-defined.

(ii) If $\vdash \theta : \Delta \to \Delta_{init}$ then $cmp_O(\theta\, \tau, \theta\, v) \in \{\textbf{lt}, \textbf{eq}\}$ iff $leq^m_O(\theta\, \tau, \theta\, v)$.

(iii) If $\Delta \vdash \theta$ subst and $cmp_O(\tau, v) = x$ for $x \neq \textbf{unk}$, then $cmp_O(\theta\, \tau, \theta\, v) = x$.

(iv) $cmp_O(\tau, v) = \textbf{eq}$ iff τ is equal to v up to permutation of row elements and ignoring the arguments of type constructors in O.

(v) $cmp_O(\tau, v) = \textbf{eq}$ iff $cmp_O(v, \tau) = \textbf{eq}$.

(vi) $cmp_O(\tau, v') = \textbf{eq}$ and $cmp_O(v', v) = \textbf{eq}$ implies $cmp_O(\tau, v) = \textbf{eq}$.

(vii) $cmp_O(\tau, v) = \textbf{lt}$ iff $cmp_O(v, \tau) = \textbf{gt}$.

(viii) $cmp_O(\tau, v') = \textbf{lt}$ and $cmp_O(v', v) = \textbf{lt}$ implies $cmp_O(\tau, v) = \textbf{lt}$.

(ix) $cmp_O(\tau, \tau') = \textbf{eq}$ and $cmp_O(\tau', v') = \textbf{lt}$ and $cmp_O(v', v) = \textbf{eq}$ implies $cmp_O(\tau, v) = \textbf{lt}$.

(x) $cmp_O(\tau, \tau') = \textbf{eq}$ and $cmp_O(\tau', v') = \textbf{unk}$ and $cmp_O(v', v) = \textbf{eq}$ implies $cmp_O(\tau, v) = \textbf{unk}$.

(xi) $cmp_O(\tau, \upsilon) = \textbf{unk}$ iff $cmp_O(\upsilon, \tau) = \textbf{unk}$.

(xii) $cmp_O(\tau, \upsilon) = \textbf{lt}$ then $cmp_\emptyset(\tau, \upsilon) = \textbf{lt}$.

(xiii) $cmp_\emptyset(\tau, \upsilon) = \textbf{eq}$ then $cmp_O(\tau, \upsilon) = \textbf{eq}$.

Proof Most are by definition of *cmp*. Property (iii), however, is a little subtle: see Lemma B.3. □

4.4 Constraint Entailment

Roughly speaking, a constraint C *entails* a constraint D, written $C \vdash^e D$, if every *satisfying substitution* for C satisfies every primitive constraint in D. However, we also ask that the satisfaction of each primitive constraint be *witnessed.* Hence the full judgement form is $C \vdash^e D \hookrightarrow B$, where B is a set of bindings of witness names of D to witnesses, which may contain witness names from C. Thus B resembles a coercion from C to D, and our $\vdash^e$ judgement decides implication in an intuitionistic logic of constraints.

4.4.1 Unification and Saturation

Our strategy for deciding entailment is to first *saturate* the equality constraints of C by reducing them to a set of unifying substitutions. We then discard those unifiers which violate any insertion constraints in C, and then check each primitive constraint in D is satisfied for each remaining unifier.

Figure 4.12 presents the definition for *saturate*. Much of the work is performed by $mgus_O$, which, given a set of equality constraints, collects the set of their most-general unifiers (if any). Here, "most-general" refers to the unifier for a fixed permutation of all rows, and does not imply *the set itself* is "most-general" in any sense. An empty unifier set implies a pair of types are non-unifiable. A non-singleton, non-empty set implies at least one pair of rows are unifiable under more than one permutation of row elements.

As in Section 4.3, O is a set of type constructors, and will be instantiated to either $\emptyset$ or (in Chapter 5) *opaque*. For the latter, the resulting "unifiers" need not unify the arguments of opaque newtypes.

Notice that the case for row unification collects the unifiers for each possible matching of the first left-hand side element type to each right-hand side element type or the right-hand side tail. Unifying a type with a row tail requires the introduction of a fresh type variable of kind `Row`, hence some care shall be required when stating properties involving $mgus_O$. Furthermore, no attempt is made to eliminate unifiers which lead to obviously ill-formed rows. For example

$$mgus_\emptyset(\texttt{Id} \vdash \texttt{(Int \# Bool \# a)}\ eq\ \texttt{(String \# Int \# b)}) = \left\{ \begin{array}{l} [\texttt{a} \mapsto \texttt{String \# d}, \texttt{b} \mapsto \texttt{Bool \# d}], \\ [\texttt{a} \mapsto \texttt{String \# Int \# e}, \texttt{b} \mapsto \texttt{Int \# Bool \# e}] \end{array} \right\}$$

Here the second unifier (an instance of the first) duplicates the `Int` element types in both rows. This definition is in keeping with the definition of cmp_O. In the sequel we shall see how such unifiers are rejected when it comes to deciding entailment.

$$
\begin{aligned}
fv_O(a) &= \{a\} \\
fv_O(F\ \overline{\tau}) &= \textbf{if } F \in O \textbf{ then } \emptyset \\
&\quad\ \textbf{else } \textstyle\bigcup_i fv_O(\tau_i) \\
fv_O((\#)_n\ \overline{\tau}\ l) &= \textstyle\bigcup_{1 \le i \le n} fv_O(\tau_i) \cup fv_O(l)
\end{aligned}
$$

$$
\begin{aligned}
mgus_O(\theta \vdash \textbf{true}) &= \{\theta\} \\
mgus_O(\theta \vdash b\ \textit{eq}\ b, C) &= mgus_O(\theta \vdash C) \\
mgus_O(\theta \vdash b\ \textit{eq}\ \tau, C) &= \textbf{if } b \in fv_O(\tau) \textbf{ then } \emptyset \\
&\quad\ \textbf{else } mgus_O([b \mapsto \tau] \circ \theta \vdash C\ [b \mapsto \tau]) \\
mgus_O(\theta \vdash \tau\ \textit{eq}\ b, C) &= mgus_O(\theta \vdash b\ \textit{eq}\ \tau, C) \\
mgus_O(\theta \vdash F\ \overline{\tau}\ \textit{eq}\ F\ \overline{v}, C) &= \textbf{if } F \in O \textbf{ then } \{\theta\} \\
&\quad\ \textbf{else } mgus_O(\theta \vdash \overline{\tau\ \textit{eq}\ v}, C) \\
mgus_O(\theta \vdash F\ \overline{\tau}\ \textit{eq}\ G\ \overline{v}, C) &= \emptyset \qquad\qquad \text{when } F \ne G \\
mgus_O(\theta \vdash (\#)_m\ \overline{\tau}\ l\ \textit{eq}\ (\#)_n\ \overline{v}\ l', C) &= \textstyle\bigcup_{1 \le j \le n} S_j \cup S'
\end{aligned}
$$

where $S_j = \{mgus_O(\theta \vdash \tau_1\ \textit{eq}\ v_j, (\#)_{m-1}\ \overline{\tau}_{\setminus 1}\ l\ \textit{eq}\ (\#)_{n-1}\ \overline{v}_{\setminus j}\ l', C)\}$
and $S' = \textbf{if } l' = a \textbf{ and } a \notin fv_O(\tau_1) \textbf{ then}$
$\quad mgus_O([a \mapsto \tau_1\ \#\ b] \circ \theta \vdash ((\#)_{m-1}\ \overline{\tau}_{\setminus 1}\ l\ \textit{eq}\ (\#)_n\ \overline{v}\ b, C)[a \mapsto \tau_1\ \#\ b])$
$\quad \textbf{else } \emptyset$
and b : Row fresh

$$
\begin{aligned}
isIn(\tau, (\#)_n\ \overline{v}\ l) &= \exists i\ .\ cmp_{opaque}(\tau, v_i) = \textbf{eq} \\
satisfied(C) &= \nexists(\tau\ \textit{ins}\ \rho) \in C\ .\ isIn(\tau, \rho) \\
saturate(C) &= \left\{ \theta \;\middle|\; \begin{array}{l} \theta \in mgus_\emptyset(\textbf{Id} \vdash eqs(C)), \\ satisfied(\theta\ inss(C)) \end{array} \right\}
\end{aligned}
$$

Figure 4.12: Definition of *fv*, *mgus*, and *saturate*

Furthermore, $mgus_O$ may also include "junk" unifiers which, though sound, are not most general. For example:

$$mugs_\emptyset(\textbf{Id} \vdash (\texttt{a \# b \# Empty})\ \textit{eq}\ (\texttt{a \# b \# Empty})) = \{\textbf{Id}, \texttt{a} \mapsto \texttt{b}\}$$

Here the second unifier is redundant, but to prevent its inclusion, or to detect and discard it, seems to be much more trouble than simply accounting for such unifiers in a few points within the correctness proofs.

Though we shall speak of *sets* of unifiers, *multi-sets* are also appropriate. Hence $mgus_O$ need not attempt to collapse duplicate unifiers.

Of course an actual implementation of $mgus_O$ needn't use such a brute-force collection of all unifiers. By using cmp^t to first sort each row, many obviously failing combinations may be rejected.

Much of the rest of the technical development will depend on substitutions being equal only up to the equality on types induced by cmp_O. To this end, let $\theta \equiv_O \theta'$ iff $\forall a\ .\ cmp_O(\theta\ a, \theta'\ a) = \textbf{eq}$.

Lemma 4.3 (Correctness of Unification)

(i) If $\forall i\ .\ \theta\ \tau_i = \tau_i \wedge \theta\ v_i = v_i$ then $\theta' \in mgus_O(\theta \vdash \overline{\tau\ \textit{eq}\ v})$ implies $\exists \theta''\ .\ \theta' = \theta'' \circ \theta$ and

$$\boxed{C \vdash^m \tau\ \boldsymbol{ins}\ \rho \hookrightarrow W}$$

$$\frac{}{C \vdash^m \tau\ \boldsymbol{ins}\ \mathsf{Empty} \hookrightarrow \mathsf{One}}\ \text{MEMPTY} \qquad \frac{(w : \tau'\ \boldsymbol{ins}\ \rho') \in C \quad cmp_{opaque}(\tau, \tau') = \mathbf{eq} \quad cmp_{opaque}(\rho, \rho') = \mathbf{eq}}{C \vdash^m \tau\ \boldsymbol{ins}\ \rho \hookrightarrow w}\ \text{MREF}$$

$$\frac{cmp_{opaque}(\tau, v_i) = \mathbf{lt} \quad C \vdash^m \tau\ \boldsymbol{ins}\ (\#)_n\ \overline{v}\ l \hookrightarrow W}{C \vdash^m \tau\ \boldsymbol{ins}\ (\#)_{n-1}\ \overline{v}_{\backslash i}\ l \hookrightarrow W}\ \text{MCONT} \qquad \frac{cmp_{opaque}(\tau, v_i) = \mathbf{gt} \quad C \vdash^m \tau\ \boldsymbol{ins}\ (\#)_n\ \overline{v}\ l \hookrightarrow W}{C \vdash^m \tau\ \boldsymbol{ins}\ (\#)_{n-1}\ \overline{v}_{\backslash i}\ l \hookrightarrow \mathsf{Dec}\ W}\ \text{MDEC}$$

$$\frac{cmp_{opaque}(\tau, v_i) = \mathbf{lt} \quad C \vdash^m \tau\ \boldsymbol{ins}\ (\#)_{n-1}\ \overline{v}_{\backslash i}\ l \hookrightarrow W}{C \vdash^m \tau\ \boldsymbol{ins}\ (\#)_n\ \overline{v}\ l \hookrightarrow W}\ \text{MEXP} \qquad \frac{cmp_{opaque}(\tau, v_i) = \mathbf{gt} \quad C \vdash^m \tau\ \boldsymbol{ins}\ (\#)_{n-1}\ \overline{v}_{\backslash i}\ l \hookrightarrow W}{C \vdash^m \tau\ \boldsymbol{ins}\ (\#)_n\ \overline{v}\ l \hookrightarrow \mathsf{Inc}\ W}\ \text{MINC}$$

$$\boxed{C \vdash^e d \hookrightarrow W}$$

$$\frac{\forall \theta \in saturate(C)\ .\ cmp_{\emptyset}(\theta\ \tau, \theta\ v) = \mathbf{eq}}{C \vdash^e \tau\ \boldsymbol{eq}\ v \hookrightarrow \mathsf{True}}\ \text{EQUALS} \qquad \frac{\forall \theta \in saturate(C)\ .\ \theta\ inss(C) \vdash^m \theta\ \tau\ \boldsymbol{ins}\ \theta\ \rho \hookrightarrow W}{C \vdash^e \tau\ \boldsymbol{ins}\ \rho \hookrightarrow W}\ \text{INSERT}$$

$$\boxed{C \vdash^e D \hookrightarrow B}$$

$$\frac{C \vdash^e \overline{d \hookrightarrow W}}{C \vdash^e \overline{w : d} \hookrightarrow \overline{w = W}}\ \text{CONJ}$$

Figure 4.13: λ^{TIR} constraint entailment

$\forall i\ .\ cmp_O(\theta''\ \tau_i, \theta''\ v_i) = \mathbf{eq}$.

(ii) If $\forall i\ .\ cmp_O(\theta\ \tau_i, \theta\ v_i) = \mathbf{eq}$ then $\exists \theta' \in mgus_O(\mathbf{Id} \vdash \overline{\tau\ \boldsymbol{eq}\ v})$ and θ'' s.t. $\theta'' \circ \theta'_{\restriction dom(\theta)} \equiv_O \theta$.

Proof See Lemma B.6 and Lemma B.7. □

Notice the use of domain restriction in the statement of equivalence of substitutions in (ii) above. This restriction is necessary because both θ' and θ'' may contain spurious bindings for row variables introduced by $mgus_O$. It is exceedingly tedious to include these restrictions in the (very many) places we must show the equivalence of substitutions. Hence, in the sequel we shall assume, unless noted otherwise, that $\equiv_O$ is equivalence *up to restriction* to the relevant variables. Here, "relevant" will be clear from context. (Jones' $\approx$ relation [47] is defined similarly, though its motivation is very different.)

4.4.2 Entailment Judgement

Figure 4.13 presents the constraint entailment judgements.

The rules of the ancillary judgement $C \vdash^m \tau \; \boldsymbol{ins} \, \rho \hookrightarrow W$ attempt to find a suitable index, W, for type τ within row ρ. Notice these rules are non-deterministic: There may be many possible derivations, and hence many possible witnesses. Furthermore, infinite derivations are possible. Both these properties are an artifact of our presentation, which is pleasantly concise compared to a fully deterministic and finite system.

Rule MEMPTY is the obvious base case (recall indices are base 1). Rule MREF allows an index to be drawn from the environment, provided all types agree opaquely up to permutation. Notice that all comparisons in these rules use cmp_{opaque} rather than $cmp_{\emptyset}$, since the type arguments of opaque newtypes should not be significant in determining the insertion position of a type in a row.

The remaining rules all attempt to build a relative index by adding or removing a type from a row for which the index is known. These rules are only applicable when the type being added or removed can be strictly ordered with respect to the type being inserted.

Sometimes W will be an absolute index. For example:

$$\texttt{true} \vdash^m \texttt{Bool} \; \boldsymbol{ins} \, (\texttt{Int \# String \# Empty}) \hookrightarrow \mathsf{Inc\ One}$$

Otherwise, W will be relative to an index in C. For example:

$$\mathsf{w} : \texttt{Bool} \; \boldsymbol{ins} \, (\mathsf{a} \; \texttt{\# Empty}) \vdash^m \texttt{Bool} \; \boldsymbol{ins} \, (\mathsf{a} \; \texttt{\# Int \# Empty}) \hookrightarrow \mathsf{Inc\ w}$$

The rules for the $C \vdash^e d \hookrightarrow W$ judgement first saturate C, then check d is satisfied under each unifier. Notice that rule INSERT requires the index W witnessing $\tau \; \boldsymbol{ins} \, \rho$ to be (syntactically) the same under each unifier. Doing so prevents a membership constraint from being incorrectly discharged. For example, the following judgement is *not* true:

$$(\mathsf{a} \; \texttt{\#} \; \mathsf{b} \; \texttt{\# Empty}) \; \boldsymbol{eq} \, (\texttt{Int \# String \# Empty}) \vdash^e \mathsf{a} \; \boldsymbol{ins} \, (\texttt{Bool \# Empty}) \hookrightarrow W$$

Depending on whether a is bound to Int or String, W can be One or Inc One. When there are multiple ways to bind an index, we assume the entailment fails if there is no single derivation which yields the same index under all unifiers. An actual implementation can avoid having to try many possible derivations of the $\vdash^m$ judgement by preferring relative to absolute indices.

Finally, the $C \vdash^e D \hookrightarrow B$ judgement extends the $C \vdash^e d \hookrightarrow W$ judgement from primitive constraints to full constraints. Notice this definition implies $saturate(C)$ is performed for each $d \in D$: Of course an implementation need not do so!

4.4.3 Soundness of Entailment

Figure 4.14 presents a simple denotational semantics for λ^{TIR} witnesses and primitive constraints over ground types. The semantics uses the set $\mathcal{I}$ of witness values. We write $\mathbf{1}$ to denote the singleton set $\{*\}$, and we let η range over all mappings from witness names to witnesses. (In the sequel these maps shall be extended to include ordinary variables.)

$$\mathcal{I} = (\mathsf{iwrong} : \mathbf{1} + \mathsf{iind} : \mathcal{N}^+ + \mathsf{itrue} : \mathbf{1})$$

$$\begin{aligned}
[\![w]\!]_\eta &= \eta\ w \\
[\![\mathsf{One}]\!]_\eta &= \mathsf{iind} : 1 \\
[\![\mathsf{Inc}\ W]\!]_\eta &= \mathbf{case}\ [\![W]\!]_\eta\ \mathbf{of}\ \{ \\
&\qquad \mathsf{iind} : i \rightarrow \mathsf{iind} : i + 1; \\
&\qquad \mathbf{otherwise} \rightarrow \mathsf{iwrong} : *\ \} \\
[\![\mathsf{Dec}\ W]\!]_\eta &= \mathbf{case}\ [\![W]\!]_\eta\ \mathbf{of}\ \{ \\
&\qquad \mathsf{iind} : i \rightarrow \mathbf{if}\ i > 1\ \mathbf{then}\ \mathsf{iind} : i - 1\ \mathbf{else}\ \mathsf{iwrong} : *; \\
&\qquad \mathbf{otherwise} \rightarrow \mathsf{iwrong} : *\ \} \\
[\![\mathsf{True}]\!]_\eta &= \mathsf{itrue} : *
\end{aligned}$$

$$sortingPerms(\tau_1^m, \ldots, \tau_n^m) = \left\{ \pi \;\middle|\; \begin{array}{l} \pi \text{ is a permutation on } n, \\ \forall i, j\ .\ i \leq j \Longrightarrow leq^m_{opaque}(\tau^m_{\pi\ i}, \tau^m_{\pi\ j}) \end{array} \right\}$$

$$\begin{aligned}
[\![\tau^m\ \boldsymbol{ins}\ (\texttt{\#})_n\ \overline{v^m}\ \texttt{Empty}]\!] &= \mathbf{if}\ \forall \pi \in S\ .\ \pi^{-1}\ 1 = i\ \mathbf{then}\ \{\mathsf{iind} : i\}\ \mathbf{else}\ \emptyset \\
&\quad \text{where } S = sortingPerms(\tau^m, v_1^m, \ldots, v_n^m) \\
[\![\tau^m\ \boldsymbol{eq}\ v^m]\!] &= \mathbf{if}\ eq^m_\emptyset(\tau^m, v^m)\ \mathbf{then}\ \{\mathsf{itrue} : *\}\ \mathbf{else}\ \emptyset
\end{aligned}$$

$$\begin{aligned}
env(B) &= env(B, \cdot) \\
env(\cdot, \eta) &= \eta \\
env((w = W, B), \eta) &= env(B, (\eta, w \mapsto [\![W]\!]_\eta))
\end{aligned}$$

Figure 4.14: Definition of the set $\mathcal{I}$, the denotation of λ^{TIR} witnesses in $\mathcal{I}$, the denotation of λ^{TIR} primitive constrants as subsets of $\mathcal{I}$, and *env*

Notice that the denotation of a primitive constraint will be either the empty set (if unsatisfied) or a singleton (if satisfied). The only subtlety is the denotation for insertion constraints. We allow *sortingPerms* to yield more that one sorting permutation, provided they all agree on the index for τ. For example

$$[\![\texttt{Bool}\ \boldsymbol{ins}\ \texttt{Int \# Int \# Empty}]\!] = \{\mathsf{iind} : 3\}$$

but

$$[\![\texttt{Bool}\ \boldsymbol{ins}\ \texttt{Int \# Bool \# Empty}]\!] = \emptyset$$

Using this model we may show our entailment judgement is sound. Notice that, for clarity, we have suppressed the trivial True witnesses for equality constraints in the proof-theoretic development, even though the following model-theoretic development requires them. They may always be reinserted where required.

We say η *satisfies* C, written $\eta \models C$, if $(w : c) \in C \Longrightarrow \eta\ w \in [\![c]\!]$. If $\Delta \vdash C$ constraint, we define *satisfiable*(C) to be true if there exists a $\Delta \vdash \theta$ subst and η s.t. $\eta \models \theta\ C$.

Let $\Delta \vdash C/D$ constraint. Then we say C *model-theoretically entails* D *with coercion* B, written $C \Vdash^e D \hookrightarrow B$, if for every $\vdash \theta : \Delta \rightarrow \Delta_{init}$ and η s.t. $\eta \models \theta\ C$, we have $env(B, \eta) \models \theta\ D$.

We say $(\tau\ \textit{eq}\ v)$ is *equivalent* to $(\tau'\ \textit{eq}\ v')$, written $(\tau\ \textit{eq}\ v) \equiv (\tau'\ \textit{eq}\ v')$, if $\mathit{cmp}_\emptyset(\tau, \tau') = \mathbf{eq}$ and $\mathit{cmp}_\emptyset(v, v') = \mathbf{eq}$ or $\mathit{cmp}_\emptyset(\tau, v') = \mathbf{eq}$ and $\mathit{cmp}_\emptyset(v, \tau') = \mathbf{eq}$. Similarly, define $(\tau\ \textit{ins}\ \rho) \equiv (\tau'\ \textit{eq}\ \rho')$ to be true if $\mathit{cmp}_\emptyset(\tau, \tau') = \mathbf{eq}$ and $\mathit{cmp}_\emptyset(\rho, \rho') = \mathbf{eq}$. We extend $\equiv$ pointwise to all constraints.

Lemma 4.4 (Soundness of Entailment) If $C \vdash^e D \hookrightarrow B$ then $C \Vdash^e D \hookrightarrow B$.

Proof See Lemma B.13 for the full theorem statement and proof. □

As an immediate consequence of soundness we have:

Lemma 4.5

(i) Types are tautologically equal if they are equivalent: $\mathtt{true} \vdash^e \tau\ \textit{eq}\ v$ implies $\mathit{cmp}_\emptyset(\tau, v) = \mathbf{eq}$.

(ii) A type may be tautologically inserted into a row if it has a unique insertion index: $\mathtt{true} \vdash^e \mathsf{w} : v\ \textit{ins}\ (\tau_1\ \texttt{\#}\ \ldots\ \texttt{\#}\ \tau_n\ \texttt{\#}\ \mathtt{Empty}) \hookrightarrow B$ implies $S \neq \emptyset$ and there exists an i s.t. $\forall \pi \in S\ .\ \pi^{-1}\ 1 = i$, where $S = \mathit{sortingPerms}(v, \tau_1, \ldots, \tau_n)$.

Proof See Lemma B.14. □

We can also show that entailment is well-behaved:

Lemma 4.6

(i) $\vdash^e$ is reflexive: $C \vdash^e C \hookrightarrow \cdot$

(ii) $\vdash^e$ is transitive: $C \vdash^e D' \hookrightarrow B$ and $D' \vdash^e D \hookrightarrow B'$ and $\eta \models \theta\ C$ implies $C \vdash^e D \hookrightarrow B''$ and $\mathit{env}(B +\!\!+ B', \eta)_{\restriction \mathit{names}(D)} = \mathit{env}(B'', \eta)_{\restriction \mathit{names}(D)}$.

(iii) $\vdash^e$ is closed under substitution: $C \vdash^e D \hookrightarrow B$ implies $\theta\ C \vdash^e \theta\ D \hookrightarrow B$.

Proof

(i) See Lemma B.28.

(ii) See Lemma B.31 for the full statement and proof.

(iii) See Lemma B.27 for the full statement and proof.

□

Finally, we can show $\mathit{saturate}(C)$ is non-empty if and only if C is satisfiable:

Lemma 4.7 $\mathit{saturate}(C) \neq \emptyset$ iff $\mathit{satisfiable}(C)$.

Proof See Lemma B.16. □

4.4.4 (In)Completeness of Entailment

The rules of entailment in Figure 4.13 are *not* complete with respect to the model of constraints given above. That is to say, $C \Vdash^e D \hookrightarrow B$ does not imply $C \vdash^e D \hookrightarrow B$. This incompleteness arises because the $\vdash^m$ judgement does not exploit the way in which types are ordered.

For example, notice that for any η and θ such that

$$\eta \models \mathsf{w} : \theta\, \mathsf{b}\ \mathit{ins}\ (\mathtt{Bool}\ \#\ \mathtt{Empty})$$

we have

$$[\![\mathsf{w}]\!]_\eta \in [\![\theta\ ((\mathtt{b},\ \mathtt{c})\ \mathit{ins}\ (\mathtt{Bool},\ \mathtt{Int})\ \#\ \mathtt{Empty})]\!]$$

However

$$\mathsf{w} : \mathsf{b}\ \mathit{ins}\ (\mathtt{Bool}\ \#\ \mathtt{Empty}) \not\vdash^m ((\mathtt{b},\ \mathtt{c})\ \mathit{ins}\ (\mathtt{Bool},\ \mathtt{Int})\ \#\ \mathtt{Empty}) \hookrightarrow \mathsf{w}$$

Some progress can be made by including the *projection* rules:

$$\frac{(w : (\tau\ v)\ \mathit{ins}\ (\tau\ v_1'\ \#\ \ldots\ \#\ \tau\ v_n'\ \#\ \mathtt{Empty})) \in C}{C \vdash^m w' : v\ \mathit{ins}\ (\#)_n\ \overline{v'}\ \mathtt{Empty} \hookrightarrow w' = w}\ \text{MPROJL}$$

$$\frac{(w : v\ \mathit{ins}\ (\#)_n\ \overline{v'}\ \mathtt{Empty}) \in C}{C \vdash^m w' : (\tau\ v)\ \mathit{ins}\ (\tau\ v_1'\ \#\ \ldots\ \#\ \tau\ v_n'\ \#\ \mathtt{Empty}) \hookrightarrow w' = w}\ \text{MPROJR}$$

Here τ is any type functor of kind $\mathtt{Type} \to \mathtt{Type}$ which does not discard its type argument (though it may be duplicated). Since such rules seem potentially very expensive to implement, we would like to first gain some experience with an implementation before deciding if such an expense is justified.

A variation of these rules for functors of kind $\mathtt{Row} \to \mathtt{Type}$ is also possible, but potentially even more expensive, since it must work with rows in canonical order.

However, even with the rules above the example entailment above still fails, and hence $\vdash^e$ remains incomplete. The problem is that these rules do not exploit the lexicographic ordering of types. Though variations of the rules above to exploit this information seem plausible, we feel this problem is one of the model being too rich rather than the entailment relation being too poor. A better approach would be to parameterise the definitions of Figure 4.14 by the definition of leq^m. We would then write $C \Vdash^e D \hookrightarrow B$ iff $\eta \models \theta\ C$ implies $\mathit{env}(B, \eta) \models \theta\ D$ for all definitions of leq^m. which satisfy the properties of Fact 4.1.

It is unknown whether $\vdash^e$ remains incomplete even with all of the refinements mentioned above.

Incompleteness of entailment has two consequences. Firstly, may properties, such as closure under substitution and transitivity, are trivial to show for $\Vdash^e$. Without completeness, we are forced to prove these properties for $\vdash^e$ also, which is substantially more complicated. Secondly, when we come to showing λ^{TIR} enjoys completeness of type inference in Chapter 5, we must base the theorem upon $\Vdash^e$ rather than $\vdash^e$.

$$
\begin{aligned}
\mathbf{E}\ A &= \{\bot\} \cup \{[a] \mid a \in A\} \\
\mathbf{unit_E} &: A \to \mathbf{E}\ A \\
&= \lambda a\ .\ [a] \\
\mathbf{bind_E} &: \mathbf{E}\ A \to (A \to \mathbf{E}\ B) \to \mathbf{E}\ B \\
&= \lambda ea\ f\ .\ \mathbf{case}\ ea\ \mathbf{of}\ \{\bot \to \bot\ ;\ [a] \to f\ a\} \\
\mathbf{strength_E} &: A \times \mathbf{E}\ B \to \mathbf{E}\ (A \times B) \\
&= \lambda a\ eb\ .\ \mathbf{case}\ eb\ \mathbf{of}\ \{\bot \to \bot\ ;\ [b] \to [(a, b)]\}
\end{aligned}
$$

Figure 4.15: Evaluation monad **E**

4.4.5 Complexity of Entailment

We do not have any complexity results for entailment, or even satisfaction. Of course, entailment is mostly a *theoretical* stepping stone towards simplification, for which care has been taken to avoid explosive time complexity. An *implementation* of entailment is used by the compiler in only two situations:

(i) The simplifier uses (a variation of) entailment to eliminate constraints containing only type variables known not to appear outside the constraint. These constraints tend to be small.

(ii) The compiler must check the constraint in a programmer-supplied type annotation entails the inferred constraint. However, programmers tend not to write very large constraints when annotating a term, usually because they are only interested in an instance of (one of) the term's principal type(s) in which most of the constraints become tautological. Furthermore, if experience with Haskell is any guide, they would prefer to be able to supply a type annotation *without* also supplying a constraint. (A "..." notation, denoting "any constraint," has been proposed for Haskell, and would likewise be suitable for λ^{TIR}.) In such cases the type checker only needs to check that the inferred constraint is satisfiable when instantiated by the annotated type.

In both cases, the left-hand side constraint is small when deciding entailment. Furthermore, rows tend not to be highly polymorphic and not deeply nested, in which case *saturate* yields only a modest number of substitutions.

4.5 Type Soundness

This section presents a denotational call-by-name semantics for λ^{TIR}. The model is inspired by that for HM(X) [79], which in turn is a mild generalisation of Milner's original model for let-bound polymorphism [66]. Types are denoted by *ideals* [59] of the domain $\mathbf{E}\ \mathcal{V}$, and terms by members of $\mathbf{E}\ \mathcal{V}$.

$\mathcal{V}$ is the pre-domain of values, defined by:

$$
\begin{aligned}
\mathcal{V} = (\ &(\mathsf{wrong} : \mathbf{1}) + (\mathsf{int} : \mathcal{Z}) + (\mathsf{func} : \mathbf{E}\ \mathcal{V} \to \mathbf{E}\ \mathcal{V}) \\
&+ (\textstyle\sum_{n \geq 0} \mathsf{prod}_n : \prod_{1 \leq i \leq n} \mathbf{E}\ \mathcal{V}) + (\mathsf{inj} : \mathcal{N}^+ \times \mathbf{E}\ \mathcal{V}) \\
&+ (\textstyle\sum_{n \geq 0} \mathsf{ifunc}_n : (\prod_{1 \leq i \leq n} \mathcal{I}) \to \mathbf{E}\ \mathcal{V})\ )
\end{aligned}
$$

$$
\begin{aligned}
[\![\texttt{Int}]\!] &= \mathbf{E}\,\{\mathsf{int} : i \mid i \in \mathcal{Z}\} \\
[\![v^m \texttt{ -> } \tau^m]\!] &= \mathbf{E}\,\{\mathsf{func} : f \mid f \in \mathbf{E}\,\mathcal{V} \to \mathbf{E}\,\mathcal{V}, v \in [\![v^m]\!] \Longrightarrow f\ v \in [\![\tau^m]\!]\} \\
[\![\texttt{All ((\#)}_n\ \overline{\tau^m}\ \texttt{Empty)}]\!] &= \mathbf{E}\,\{\mathsf{prod}_n : \langle v_1, \ldots, v_n\rangle \mid v_1 \in [\![\tau^m_{\pi\,1}]\!], \ldots, v_n \in [\![\tau^m_{\pi\,n}]\!]\} \\
[\![\texttt{One ((\#)}_n\ \overline{\tau^m}\ \texttt{Empty)}]\!] &= \mathbf{E}\,\{\mathsf{inj} : \langle i, v\rangle \mid 1 \leq i \leq n, v \in [\![\tau^m_{\pi\,i}]\!]\} \\
& \quad \text{where } \pi \in \mathit{sortingPerms}(\tau^m_1, \ldots, \tau^m_n) \\
[\![A\ v^m_1 \ldots v^m_n]\!] &= \mathbf{E} \downarrow (\mathbf{lfp}\ \lambda d\ .\ [\![(\mathit{norm}(\tau\ v^m_1 \ldots v^m_n))[A\ v^m_1 \ldots v^m_n \mapsto d]]\!]) \\
& \quad \text{where } (\texttt{newtype}\ \{\texttt{opaque}\}^{opt}\ A = \tau) \in \mathit{tdecls} \\
[\![d]\!] &= d
\end{aligned}
$$

$$
\begin{aligned}
&[\![\texttt{forall}\ \Delta\ .\ C \texttt{ => } \tau]\!] = \bigcap \left\{ S_{(\theta,B)} \;\middle|\; \begin{array}{l} \vdash \theta : \Delta \to \Delta_{init}, \\ \mathit{env}(B) \models \theta\ D \end{array} \right\} \\
&\quad \text{where } D = \mathit{named}(C) \\
&\quad \text{and } \mathit{names}(D) = (w_1, \ldots, w_n) \\
&\quad \text{and } S_{(\theta,B)} = \mathbf{E}\ \left\{ \mathsf{ifunc}_n : f \;\middle|\; \begin{array}{l} f \in (\prod_{1 \leq i \leq n} \mathcal{I}) \to \mathbf{E}\,\mathcal{V}, \\ f\ ([\![w_1]\!]_{\mathit{env}(B)}, \ldots, [\![w_n]\!]_{\mathit{env}(B)}) \in [\![\theta\ \tau]\!] \end{array} \right\}
\end{aligned}
$$

Figure 4.16: Denotation of λ^{TIR} normalized monotypes and type schemes as ideals of $\mathbf{E}\,\mathcal{V}$

Here $+$ is categorical sum, $\to$ continuous (not necessarily strict) function space, $\mathcal{Z}$ the set of integers, $\mathcal{N}^+$ the set of non-zero naturals, $\mathcal{I}$ the set of indices defined in Figure 4.14, and $\mathbf{E}$ is the evaluation (lifting) monad defined in Figure 4.15. Each summand is tagged by a mnemonic for its injector. We use the summand $\mathsf{wrong} : *$ to denote all ill-typed programs.

(This somewhat unorthodox presentation of $\mathcal{V}$ as a pre-domain rather than a domain has been chosen so as to make the monad $\mathbf{E}$ explicit, which in turn simplifies the proof of type soundness.)

Figure 4.16 presents the denotation of λ^{TIR} monotypes and (closed) type schemes. The denotation for an `All` type is a product of types ordered by a sorting permutation. Similarly, a `One` type is a pair of an index and type, where the index must match the type under a sorting permutation. (Recall *sortingPerms* was defined in Figure 4.14.) Notice that we say "a" rather than "the" sorting permutation here so that we may assign a meaning to all well-kinded types, including TIP's and TIC's containing duplicate types. Notice that the choice of permutation does not change the denotation of these types, because we shall show equal types have equal denotations. Furthermore, since all types are ground, there will always be at least one permutation.

Newtypes are possibly recursive: We assume they are never mutually recursive and all the recursion is *regular.* (A model for all λ^{TIR} recursive types is possible but would take us too far afield.) We write **lfp** to denote the usual least-fixed-point solution (up to isomorphism) of mixed-variance recursive types using e-p pairs and strict function spaces. This solution is always well defined (and thus the result pointed) since the denotation of all other types are pointed, and every recursive cycle for a newtype passes through a `One` or `All` constructor. We write $\mathbf{unfold}_A$ and $\mathbf{fold}_A$ for the usual mediating morphisms. That is, if $(\texttt{newtype}\ \{\texttt{opaque}\}^{opt}\ \texttt{A} = v) \in \mathit{tdecls}$ and $(\texttt{A} : \kappa_1 \texttt{ -> } \ldots \texttt{ -> } \kappa_n \texttt{ -> Type}) \in \Delta_{init}$, then

for any $\Delta_{init} \vdash \overline{\tau : \kappa}$ we have

$$\begin{aligned} \mathbf{fold}_A &: [\![norm(\upsilon\ \tau_1 \dots \tau_n)]\!] \to [\![A\ \tau_1 \dots \tau_n]\!] \\ \mathbf{unfold}_A &: [\![A\ \tau_1 \dots \tau_n]\!] \to [\![norm(\upsilon\ \tau_1 \dots \tau_n)]\!] \end{aligned}$$

(For clarity we suppress the parameterisation on A, which is always clear from context.) The operation $\downarrow$ removes the bottom element from a domain. We use it so that the denotation of every type has $\perp$ as its least element.

The most important aspect of our model is the denotation of type schemes. If a scheme contains insertion constraints, its denotation is the ideal of all index abstractions which are well-behaved for all possible solutions to the constraints. This is defined by taking the intersection over all grounding substitutions θ for which $env(B) \models \theta\ D$. Then each index abstraction must yield a well-typed result given the (meaning of the) bindings in B. (Again, recall env was defined in Figure 4.14.)

It is easy to see $\mathsf{wrong} : *$ never appears within the denotation of a monotype:

Fact 4.8 If $\Delta_{init} \vdash \tau : \mathsf{Type}$ then $[\mathsf{wrong} : *] \notin [\![\tau]\!]$.

Furthermore, the denotation of monotypes respects equality:

Fact 4.9 $eq^m_\emptyset(\tau^m, \upsilon^m)$ implies $[\![\tau^m]\!] = [\![\upsilon^m]\!]$.

The situation is not so simple for type schemes. If C is unsatisfiable, $[\![\texttt{forall}\ \Delta'\ .\ C\ \texttt{=>}\ \tau]\!] = \mathbf{E}\ \mathcal{V}$, which clearly does contain $[\mathsf{wrong} : *]$. However, provided the top-level constraint of a term is satisfiable, all of the constraints arising within it are also satisfiable. This reasoning is built into the soundness proof, to follow shortly.

Figure 4.18 presents the denotation of λ^{TIR} terms. For convenience, we allow η to bind both term values (members of $\mathbf{E}\ \mathcal{V}$) and index values (members of $\mathcal{I}$). We write $\mathbf{let_E}\ x \leftarrow u\ \mathbf{in}\ t$ as shorthand for $\mathbf{bind_E}\ u\ (\lambda x\ .\ u)$.

We now show the translation of every well-typed λ^{TIR} term has a denotation within the denotation of its type. Since no λ^{TIR} type contains $[\mathsf{wrong} : *]$, this property implies a well-typed program, when translated, will not encounter a run-time type error.

We say η *models* Γ, written $\eta \models \Gamma$, if $dom(\eta) = dom(\Gamma)$ and for every $(x : \sigma) \in \Gamma$, $\eta\ x \in [\![\sigma]\!]$.

Theorem 4.10 (Type Soundness) If $\Delta \mid C \mid \Gamma \vdash t : \tau \hookrightarrow T$, and θ is grounding and well-kinded under Δ, and $env(B) \models \theta\ C$, and $\eta \models \theta\ \Gamma$ then $[\![T]\!]_{\eta + env(B)} \in [\![\theta\ \tau]\!]$.

Proof See Theorem B.39 for the full theorem statement and proof. □

$$
\begin{aligned}
[\![i]\!]_\eta &= \mathbf{unit_E}\ (\mathsf{int} : i) \\
[\![\langle T_1, \ldots, T_n \rangle]\!]_\eta &= \mathbf{unit_E}\ (\mathsf{prod}_n : \langle [\![T_1]\!]_\eta, \ldots, [\![T_n]\!]_\eta \rangle) \\
[\![\mathsf{Inj}\ W\ \ T]\!]_\eta &= \mathbf{case}\ [\![W]\!]_\eta\ \mathbf{of}\ \{ \\
&\qquad \mathsf{iind} : i \rightarrow \mathbf{unit_E}\ (\mathsf{inj} : \langle i, [\![T]\!]_\eta \rangle); \\
&\qquad \mathbf{otherwise} \rightarrow \mathbf{unit_E}\ (\mathsf{wrong} : *)\ \} \\
[\![\lambda x\ .\ T]\!]_\eta &= \mathbf{unit_E}\ (\mathsf{func} : \lambda y\ .\ [\![T]\!]_{\eta, x \mapsto y}) \\
[\![\lambda(w_1, \ldots, w_n)\ .\ T]\!]_\eta &= \mathbf{unit_E}\ (\mathsf{ifunc}_n : \lambda(y_1, \ldots, y_n)\ . \\
&\qquad [\![T]\!]_{\eta, w_1 \mapsto y_1, \ldots, w_n \mapsto y_n}) \\
[\![T\ \ U]\!]_\eta &= \mathbf{let_E}\ v \leftarrow [\![T]\!]_\eta \\
&\quad \mathbf{in\ case}\ v\ \mathbf{of}\ \{ \\
&\qquad \mathsf{func} : f \rightarrow f\ [\![U]\!]_\eta; \\
&\qquad \mathbf{otherwise} \rightarrow \mathbf{unit_E}\ (\mathsf{wrong} : *)\ \} \\
[\![T\ (W_1, \ldots, W_n)]\!]_\eta &= \mathbf{let_E}\ v \leftarrow [\![T]\!]_\eta \\
&\quad \mathbf{in\ case}\ v\ \mathbf{of}\ \{ \\
&\qquad \mathsf{ifunc}_n : f \rightarrow f\ ([\![W_1]\!]_\eta, \ldots, [\![W_n]\!]_\eta); \\
&\qquad \mathbf{otherwise} \rightarrow \mathbf{unit_E}\ (\mathsf{wrong} : *)\ \} \\
[\![x]\!]_\eta &= \eta\ x \\
[\![A]\!]_\eta &= \mathbf{fold}_A \\
[\![A^{-1}]\!]_\eta &= \mathbf{unfold}_A
\end{aligned}
$$

Figure 4.17: Denotation of λ^{TIR} run-time terms as members of $\mathbf{E}\ \mathcal{V}$ (part 1 of 2)

$$
\begin{array}{rl}
[\![\mathsf{insert}\ U\ \mathsf{at}\ W\ \mathsf{into}\ T]\!]_\eta = & \mathbf{let_E}\ v \leftarrow [\![T]\!]_\eta \\
& \mathbf{in\ case}\ (v, [\![W]\!]_\eta)\ \mathbf{of}\ \{ \\
& \quad (\mathsf{prod}_n : \langle v'_1, \ldots, v'_n \rangle, \mathsf{iind} : i) \rightarrow \\
& \quad\quad \mathbf{unit_E}\ (\mathbf{if}\ 1 \le i \le n+1\ \mathbf{then}\ v''\ \mathbf{else}\ \mathsf{wrong} : *); \\
& \quad \mathbf{otherwise} \rightarrow \mathbf{unit_E}\ (\mathsf{wrong} : *)\ \} \\
& \mathrm{where}\ v'' = \mathsf{prod}_{n+1} : \langle v'_1, \ldots, v'_{i-1}, [\![U]\!]_\eta, v'_i, \ldots, v'_n \rangle \\
[\![\mathsf{let}\ \langle\rangle = U\ \mathsf{in}\ T]\!]_\eta = & \mathbf{let_E}\ v \leftarrow [\![U]\!]_\eta \\
& \mathbf{in\ case}\ v\ \mathbf{of}\ \{ \\
& \quad \mathsf{prod}_0 : \langle\rangle \rightarrow [\![T]\!]_\eta; \\
& \quad \mathbf{otherwise} \rightarrow \mathbf{unit_E}\ (\mathsf{wrong} : *)\ \} \\
[\![\mathsf{let}\ x\ y = \mathsf{remove}\ W\ \mathsf{from}\ U\ \mathsf{in}\ T]\!]_\eta = & \\
& \mathbf{let_E}\ v \leftarrow [\![U]\!]_\eta \\
& \mathbf{in\ case}\ (v, [\![W]\!]_\eta)\ \mathbf{of}\ \{ \\
& \quad (\mathsf{prod}_n : \langle v'_1, \ldots, v'_n \rangle, \mathsf{iind} : i) \rightarrow \\
& \quad\quad \mathbf{if}\ 1 \le i \le n\ \mathbf{then}\ [\![T]\!]_{\eta, x \mapsto v'_i, y \mapsto v''}\ \mathbf{else\ unit_E}\ (\mathsf{wrong} : *); \\
& \quad \mathbf{otherwise} \rightarrow \mathbf{unit_E}\ (\mathsf{wrong} : *)\ \} \\
& \mathrm{where}\ v'' = \mathbf{unit_E}\ (\mathsf{prod}_{n-1} : \langle v'_1, \ldots, v'_{i-1}, v'_{i+1}, \ldots, v'_n \rangle) \\
[\![\mathsf{case}\ U\ \mathsf{of}\ \{\ \mathsf{Inj}\ W\ x \rightarrow T_1; \mathsf{otherwise} \rightarrow T_2\ \}]\!]_\eta = & \\
& \mathbf{let_E}\ v \leftarrow [\![U]\!]_\eta \\
& \mathbf{in\ case}\ (v, [\![W]\!]_\eta)\ \mathbf{of}\ \{ \\
& \quad (\mathsf{inj} : \langle j, v' \rangle, \mathsf{iind} : i) \rightarrow \mathbf{if}\ i = j\ \mathbf{then}\ [\![T_1]\!]_{\eta, x \mapsto v'}\ \mathbf{else}\ [\![T_2]\!]_\eta; \\
& \quad \mathbf{otherwise} \rightarrow \mathbf{unit_E}\ (\mathsf{wrong} : *)\ \} \\
[\![\mathsf{case}\ U\ \mathsf{of}\ \{\ i \rightarrow T_1; \mathsf{otherwise} \rightarrow T_2\ \}]\!]_\eta = & \\
& \mathbf{let_E}\ v \leftarrow [\![U]\!]_\eta \\
& \mathbf{in\ case}\ v\ \mathbf{of}\ \{ \\
& \quad \mathsf{int} : j \rightarrow \mathbf{if}\ i = j\ \mathbf{then}\ [\![T_1]\!]_\eta\ \mathbf{else}\ [\![T_2]\!]_\eta; \\
& \quad \mathbf{otherwise} \rightarrow \mathbf{unit_E}\ (\mathsf{wrong} : *)\ \} \\
[\![\mathsf{let}\ x = U\ \mathsf{in}\ T]\!]_\eta = & [\![T]\!]_{\eta, x \mapsto [\![U]\!]_\eta} \\
[\![\mathsf{letw}\ B\ \mathsf{in}\ T]\!]_\eta = & [\![T]\!]_{env(B,\eta)}
\end{array}
$$

Figure 4.18: Denotation of λ^{TIR} run-time terms as members of $\mathbf{E}\ \mathcal{V}$ (part 2 of 2)

Chapter 5

Type Inference

This chapter develops a type inference system for λ^{TIR}, which we show sound and (with one caveat) complete with respect to the type checking system given in Chapter 4.

5.1 Inference Rules

The *type inference* judgement $\theta \mid C \mid \Gamma \vdash t : \tau \hookrightarrow T$ is defined by the rules of Figure 5.1. This relation may be read as a type inference algorithm with t and Γ as inputs, and θ, C and T as outputs. Its intended interpretation is:

> "Given term t in type context Γ, t has the most general type τ and constraint C, assuming the free-variables of Γ are bound by θ. Furthermore, t may be implemented by the run-time term T."

An ancillary judgement for inferring the types of patterns is defined in Figure 5.2.

These rules are, for the most part, mechanically derived from those for type checking given in Figures 4.8 and 4.9:

- Types arbitrarily introduced by a type-checking rule must be replaced by a fresh type variable (of the same kind) in the corresponding type-inference rule. For example, the well-kinded types $\overline{v}$ in rule VAR become the fresh type variables $\overline{b}$ in rule IVAR.
- Similarly, types which appear only in the conclusion of a type-checking rule must be replaced by a fresh type variable in the type-inference rule. For example, τ in rule APP becomes variable b in rule IAPP.
- Each primitive constraint tested for entailment by a type-checking rule must instead be accumulated by the corresponding type-inference rule. For example, the constraint $v\, \boldsymbol{eq}(v' \texttt{ -> } \tau)$ in rule APP becomes the constraint $(\theta_2\ \tau)\, \boldsymbol{eq}(v \texttt{ -> } b)$ in rule IAPP, which is included in the result constraint.
- The substitution θ must be threaded linearly throughout the derivation, and applied to Γ in any intermediate derivations. (The proof of completeness will turn out to be a little easier if the domain of θ is restricted to $fv_{\emptyset}(\Gamma)$, hence the explicit restrictions in rules ISIMP and IP7.)

There are two exceptions to this transliteration. Firstly, and as usual [19, 47], the ILET rule must generalise the type and constraint for u when inferring the type of `let` x = u `in` t.

$$\boxed{\theta \mid C \mid \Gamma \vdash t : \tau \hookrightarrow T}$$

$$\frac{}{\mathbf{Id} \mid \mathbf{true} \mid \Gamma \vdash i : \mathbf{Int} \hookrightarrow i}\ \text{IINT}$$

$$\frac{\begin{array}{c}\theta_1 \mid D \mid \Gamma \vdash t : \tau \hookrightarrow T \quad \theta_2 \mid D' \mid \theta_1\,\Gamma \vdash u : \upsilon \hookrightarrow U \\ b : \mathsf{Type}\ \text{fresh} \quad C = (\theta_2\ D) +\!\!+\ D' +\!\!+\ (\theta_2\ \tau)\ eq_{\mathsf{Type}}\ (\upsilon\ \texttt{->}\ b)\end{array}}{\theta_2 \circ \theta_1 \mid C \mid \Gamma \vdash t\ u : b \hookrightarrow T\ U}\ \text{IAPP}$$

$$\frac{\begin{array}{c}(x/f : \texttt{forall}\ \overline{a : \kappa}\ .\ D \texttt{ => } \tau) \in \Gamma \\ \overline{b : \kappa}\ \text{fresh} \quad C = named(D)[\overline{a \mapsto b}]\end{array}}{\mathbf{Id} \mid C \mid \Gamma \vdash x/f : \tau[\overline{a \mapsto b}] \hookrightarrow x/f\ names(C)}\ \text{IVAR}$$

$$\frac{\theta \mid C \mid \Gamma \vdash_1 abs : \tau \hookrightarrow T[\bullet]}{\theta \mid C \mid \Gamma \vdash \texttt{\{}abs\texttt{\}} : \tau \hookrightarrow T[\mathsf{undefined}]}\ \text{IABS}$$

$$\frac{\begin{array}{c}\theta_1 \mid D \mid \Gamma \vdash_1 abs_1 : \tau \hookrightarrow T[\bullet] \\ \theta_2 \mid D' \mid \theta_1\,\Gamma \vdash \texttt{\{}abs_2, \ldots, abs_{n+1}\texttt{\}} : \tau' \hookrightarrow U \\ C = (\theta_2\ D) +\!\!+\ D' +\!\!+\ (\theta_2\ \tau)\ eq_{\mathsf{Type}}\ \tau'\end{array}}{\theta_2 \circ \theta_1 \mid C \mid \Gamma \vdash \texttt{\{}abs_1, \ldots, abs_{n+1}\texttt{\}} : \tau' \hookrightarrow \mathsf{let}\ z = U\ \mathsf{in}\ T[z]}\ \text{IDISC}$$

$$\frac{\begin{array}{c}x \in fv(t) \quad \theta_1 \mid D_1 \mid \Gamma \vdash u : \upsilon \hookrightarrow U \\ gen(D_1 \mid \theta_1\,\Gamma \mid \upsilon) = (D_2 \mid \Delta \mid D_3) \\ \sigma = \texttt{forall}\ \Delta\ .\ anon(D_3) \texttt{ => } \upsilon \\ \theta_2 \mid D_4 \mid (\theta_1\,\Gamma), x : \sigma \vdash t : \tau \hookrightarrow T \\ saturate((\theta_2\ D_1) +\!\!+\ D_4) \neq \emptyset \quad C = (\theta_2\ D_2) +\!\!+\ D_4\end{array}}{\begin{array}{c}\theta_2 \circ \theta_1 \mid C \mid \Gamma \vdash \texttt{let}\ x \texttt{ = } u\ \texttt{in}\ t : \tau \\ \hookrightarrow \mathsf{let}\ x = \lambda names(D_3)\ .\ U\ \mathsf{in}\ T\end{array}}\ \text{ILET}$$

$$\frac{\begin{array}{c}\theta_1 \mid C' \mid \Gamma \vdash t : \tau \hookrightarrow T \\ \langle fv_{\emptyset}(\theta_1\,\Gamma) \cup fv_{\emptyset}(\tau) \mid C' \rangle \rhd^{*} \langle \theta_2 \mid C \mid B \rangle\end{array}}{(\theta_2 \circ \theta_1)_{\restriction fv_{\emptyset}(\Gamma)} \mid C \mid \Gamma \vdash t : \theta_2\ \tau \hookrightarrow \mathsf{letw}\ B\ \mathsf{in}\ T}\ \text{ISIMP}$$

Figure 5.1: Type inference and translation for λ^{TIR} terms

$$\boxed{\theta \mid C \mid \Gamma \vdash_n t : \tau \hookrightarrow T[\bullet]}$$

$$\frac{\theta \mid C \mid \Gamma \vdash t : \tau \hookrightarrow T}{\theta \mid C \mid \Gamma \vdash_0 t : \tau \hookrightarrow T} \text{ IP1}$$

$$\frac{\theta \mid C \mid \Gamma \vdash_n t : \tau \hookrightarrow T[\bullet]}{\begin{array}{c}\theta \mid C \mid \Gamma \vdash_{n+1} \texttt{\textbackslash}i\ .\ t : \texttt{Int -> } \tau \\ \hookrightarrow \lambda x\ .\ \mathsf{case}\ x\ \mathsf{of}\ \{i \rightarrow T[\bullet\ x]\ ;\mathsf{otherwise} \rightarrow \bullet\ x\}\end{array}} \text{ IP2}$$

$$\frac{\begin{array}{c}(\texttt{newtype}\ \{\mathsf{opaque}\}^{opt}\ A = v') \in tdecls \\ (A : \kappa_1 \texttt{ -> } \ldots \texttt{ -> } \kappa_n \texttt{ -> Type}) \in \Delta_{init} \\ \overline{b : \kappa}\ \mathsf{fresh} \quad \theta \mid D \mid \Gamma \vdash_{n+1} \texttt{\textbackslash}p\ .\ t : (\tau' \texttt{ -> } \tau) \hookrightarrow T[\bullet] \\ C = D +\!\!+\ norm(v'\ b_1 \ldots b_n)\ eq_{\mathsf{Type}}\ \tau'\end{array}}{\begin{array}{c}\theta \mid C \mid \Gamma \vdash_{n+1} \texttt{\textbackslash(}A\ p\texttt{)}\ .\ \mathrm{t}{:}\ A\ b_1 \ldots b_n \texttt{ -> } \tau \\ \hookrightarrow \lambda x\ .\ \mathsf{let}\ y = A^{-1}\ x\ \mathsf{in}\ T[\lambda y\ .\ \bullet\ (A\ y)]\ y\end{array}} \text{ IP3}$$

$$\frac{\begin{array}{c}\theta \mid D \mid \Gamma \vdash_{n+1} \texttt{\textbackslash}p\ .\ t : (v \texttt{ -> } \tau) \hookrightarrow T[\bullet] \\ b : \texttt{Row}\ \mathsf{fresh} \quad w\ \mathsf{fresh} \\ C = D +\!\!+\ w : v\ \boldsymbol{ins}\ b\end{array}}{\begin{array}{c}\theta \mid C \mid \Gamma \vdash_{n+1} \texttt{\textbackslash(Inj}\ p\texttt{)}\ .\ t : \texttt{One (}v\ \texttt{\#}\ b\texttt{) -> } \tau \\ \hookrightarrow \lambda x\ .\ \mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{Inj}\ W\ y\ \rightarrow T[\lambda y\ .\ \bullet\ (\mathsf{Inj}\ W\ y)]\ y; \\ \mathsf{otherwise} \rightarrow \bullet\ x\}\end{array}} \text{ IP4}$$

$$\frac{\begin{array}{c}\theta \mid D \mid \Gamma \vdash_{n+2} \texttt{\textbackslash}p\ .\ \texttt{\textbackslash}q\ .\ t : (v_1 \texttt{ -> } v_2 \texttt{ -> } \tau) \hookrightarrow T[\bullet] \\ b : \texttt{Row}\ \mathsf{fresh} \quad w\ \mathsf{fresh} \\ C = D +\!\!+\ \texttt{All}\ b\ eq_{\mathsf{Type}}\ v_2 +\!\!+\ w : v_1\ \boldsymbol{ins}\ b\end{array}}{\begin{array}{c}\theta \mid C \mid \Gamma \vdash_{n+1} \texttt{\textbackslash(}p\ \texttt{\&\&}\ q\texttt{)}\ .\ t : \texttt{All (}v_1\ \texttt{\#}\ b\texttt{) -> } \tau \\ \hookrightarrow \lambda x\ .\ \mathsf{let}\ y\ z = \mathsf{remove}\ W\ \mathsf{from}\ x \\ \mathsf{in}\ T[\lambda y\ .\ \lambda z\ .\ \bullet\ (\mathsf{insert}\ y\ \mathsf{at}\ W\ \mathsf{into}\ z)]\ y\ z\end{array}} \text{ IP5}$$

$$\frac{\theta \mid C \mid \Gamma \vdash_n t : \tau \hookrightarrow T[\bullet]}{\begin{array}{c}\theta \mid C \mid \Gamma \vdash_{n+1} \texttt{\textbackslash Triv}\ .\ t : \texttt{All Empty -> } \tau \\ \hookrightarrow \lambda x\ .\ \mathsf{let}\ \langle\rangle = x\ \mathsf{in}\ T[\bullet\ x]\end{array}} \text{ IP6}$$

$$\frac{b : \texttt{Type}\ \mathsf{fresh} \quad \theta \mid C \mid \Gamma, x : b \vdash_n t : \tau \hookrightarrow T[\bullet]}{\theta_{\setminus b} \mid C \mid \Gamma \vdash_{n+1} \texttt{\textbackslash}x\ .\ t : ((\theta\ b) \texttt{ -> } \tau) \hookrightarrow \lambda x\ .\ T[\bullet\ x]} \text{ IP7}$$

Figure 5.2: Type inference and translation for λ^{TIR} patterns

$$notEqual(C \vdash \tau, v) = \forall \theta \in mgus_{opaque}(\mathbf{Id} \vdash \tau \; \mathbf{eq} \; v) \; . \; \neg satisfied(\theta \; inss(C))$$

$$notIn(C \vdash \tau, \texttt{(\#)}_n \; \overline{v} \; l) = \forall i \; . \; notEqual(C \vdash \tau, v_i) \wedge \left(\begin{array}{l} l = \texttt{Empty} \\ \vee \; \exists (\tau' \; \mathbf{ins} \; \texttt{(\#)}_m \; \overline{v'} \; l') \in inss(C) \; . \\ \quad cmp_{opaque}(\tau, \tau') = \mathbf{eq} \wedge l = l' \end{array} \right)$$

Figure 5.3: Definition of *notIn*

To this end we define the *generalisation function*, *gen*, as:

$$\begin{array}{l} gen(C \mid \Gamma \mid \tau) = (D_1 \mid \Delta \mid D_2) \\ \quad \text{where } \Delta = (fv_\emptyset(C) \cup fv_\emptyset(\tau)) \setminus fv_\emptyset(\Gamma) \\ \quad \text{and } D_1 = \{(w : c) \in C \mid fv_\emptyset(c) \cap dom(\Delta) = \emptyset \wedge inheritable(c)\} \\ \quad \text{and } D_2 = \{(w : c) \in C \mid fv_\emptyset(c) \cap dom(\Delta) \neq \emptyset \vee \neg inheritable(c)\} \end{array}$$

Here we intend the resulting generalised type scheme to be `forall` Δ . D_2 `=>` τ, and the constraint D_1 to be "held over" into the current constraint context. Notice that only non-inheritable constraints with free variables contained in $fv_\emptyset(\Gamma)$ may be lifted outside the scope of the universal quantification over Δ. If λ^{TIR} were to be extended with implicit parameters [57], this restriction would ensure any implicit parameters in u are captured by u's generalised type scheme.

The second exception is the inclusion of the simplification rule ISIMP. This rule may be used to simplify the current constraint context at arbitrary points of the derivation. Hence the type inference rules are not fully syntax directed. In a practical implementation type inference should be syntax directed, and so the ISIMP rule should either be applied after each derivation step, or just before generalisation. However, unlike in the HM(X) framework [79], our development shall not assume simplification occurs at any particular point in the derivation—not even before generalisation! Our approach to simplification is instead based on Jones' refinement of OML to handle context improvement and simplification [48].

5.2 Constraint Simplification

The constraint simplifier is presented in Figures 5.4 and 5.5. Rules S1–S18, of the form $\langle \overline{a} \mid C \rangle \triangleright \langle \theta \mid C' \mid B \rangle$, allow a constraint C to be simplified by a single step into constraint C' and a *residual substitution* θ. One may think of θ as a particularly efficient representation for a set of equality constraints of the form $a \; \mathbf{eq} \; \tau$. The bindings B describe how the witnesses of C may be constructed from those of C'. (We shall explain the purpose of $\overline{a}$, a set of type variables, shortly.) If C is unsatisfiable it may be rewritten to the canonical unsatisfiable constraint `false`, thus signalling a type error.

Rules S1–S4 implement conventional unification over finite Herbrand terms.

Rules S5–S8 extend unification to rows. Rules S5 and S6 reject rows of obviously incompatible arities. The remaining rules are guided by an ancillary function, *notIn*, defined in Figure 5.3.

$$\boxed{\langle \overline{a} \mid C \rangle \triangleright \langle \theta \mid C' \mid B \rangle}$$

Simple Unification

$$\langle \overline{a} \mid C, \tau \ eq_\kappa \ v \rangle \triangleright \langle \textbf{Id} \mid C, v \ eq_\kappa \ \tau \mid \cdot \rangle \qquad \text{S1}$$

$$\langle \overline{a} \mid C, b \ eq_\kappa \ \tau \rangle \triangleright \langle [b \mapsto \tau] \mid C[b \mapsto \tau] \mid \cdot \rangle \quad \text{when } b \notin fv_\emptyset(\tau) \qquad \text{S2}$$

$$\langle \overline{a} \mid C, F \ \overline{\tau} \ eq_{\texttt{Type}} \ F \ \overline{v} \rangle \triangleright \langle \textbf{Id} \mid C, \overline{\tau \ eq_{\kappa'} \ v} \mid \cdot \rangle \quad \text{when } F : \kappa'_1 \ \texttt{->} \ \ldots \ \texttt{->} \ \kappa'_n \ \texttt{->} \ \texttt{Type} \qquad \text{S3}$$

$$\langle \overline{a} \mid C, F \ \overline{\tau} \ eq_{\texttt{Type}} \ G \ \overline{v} \rangle \triangleright \langle \textbf{Id} \mid \texttt{false} \mid \cdot \rangle \quad \text{when } F \neq G \qquad \text{S4}$$

Row Unification

$$\langle \overline{a} \mid C, (\texttt{\#})_m \ \overline{\tau} \ b \ eq_{\texttt{Row}} \ (\texttt{\#})_n \ \overline{v} \ \texttt{Empty} \rangle \triangleright \langle \textbf{Id} \mid \texttt{false} \mid \cdot \rangle \quad \text{when } m > n \qquad \text{S5}$$

$$\langle \overline{a} \mid C, (\texttt{\#})_m \ \overline{\tau} \ \texttt{Empty} \ eq_{\texttt{Row}} \ (\texttt{\#})_n \ \overline{v} \ \texttt{Empty} \rangle \triangleright \langle \textbf{Id} \mid \texttt{false} \mid \cdot \rangle \quad \text{when } m \neq n \qquad \text{S6}$$

$$\langle \overline{a} \mid C, (\texttt{\#})_m \ \overline{\tau} \ l \ eq_{\texttt{Row}} \ (\texttt{\#})_n \ \overline{v} \ l' \rangle \triangleright \langle \textbf{Id} \mid \texttt{false} \mid \cdot \rangle \qquad \text{S7}$$
$$\text{when } notIn(C \vdash \tau_i, (\texttt{\#})_n \ \overline{v} \ l')$$

$$\langle \overline{a} \mid C, (\texttt{\#})_m \ \overline{\tau} \ l \ eq_{\texttt{Row}} \ (\texttt{\#})_n \ \overline{v} \ l' \rangle$$
$$\triangleright \langle \textbf{Id} \mid C, \tau_i \ eq_{\texttt{Type}} \ v_j, (\texttt{\#})_{m-1} \ \overline{\tau}_{\backslash i} \ l \ eq_{\texttt{Row}} \ (\texttt{\#})_{n-1} \ \overline{v}_{\backslash j} \ l' \mid \cdot \rangle \qquad \text{S8}$$
$$\text{when } notIn(C \vdash \tau_i, (\texttt{\#})_{n-1} \ \overline{v}_{\backslash j} \ l') \text{ and } cmp_{opaque}(\tau_i, v_j) = \textbf{eq/unk}$$

Figure 5.4: Simplification of λ^{TIR} constraints (part 1 of 2)

We intend $notIn(C \vdash \tau, \rho)$ to be true if C entails that type τ cannot appear within row ρ. For example, if τ is not unifiable with any member of ρ, and ρ is closed, *notIn* yields true:

$$notIn(\texttt{true} \vdash \texttt{Int}, \texttt{Bool \# Char \# Empty}) = \texttt{tt}$$

$$notIn(\texttt{true} \vdash \texttt{(a, b)}, \texttt{Bool \# Char \# Empty}) = \texttt{tt}$$

If τ is unifiable with members of ρ, and ρ is closed, *notIn* yields true if each unification would contradict a constraint in C:

$$notIn(\texttt{true} \vdash \texttt{Int}, \texttt{Bool \# Int \# Empty}) = \texttt{ff}$$

$$notIn(\texttt{a} \ \textit{ins} \ \texttt{Int \# Empty} \vdash \texttt{(a, b)}, \texttt{(Int, Bool) \# Int \# Empty}) = \texttt{tt}$$

Finally, when ρ is open, *notIn* is true only when the conditions above hold and C contains a constraint preventing τ from appearing in ρ's tail:

$$notIn(\texttt{true} \vdash \texttt{Int}, \texttt{Bool \# Char \# a}) = \texttt{ff}$$

$$notIn(\texttt{Int} \ \textit{ins} \ \texttt{a} \vdash \texttt{Int}, \texttt{Bool \# Char \# a}) = \texttt{tt}$$

The *notIn* function is exploited by rules S7 and S8. Rule S7 signals failure if a type within ρ cannot appear anywhere within ρ'. Rule S8 allows a type within ρ to be matched against a type within ρ', provided there are no other possible matchings involving one of this pair of types.

Membership

$$\langle \overline{a} \mid C, w : \tau \; \textit{\textbf{ins}} \; \rho, w' : \tau' \; \textit{\textbf{ins}} \; \rho' \rangle \rhd \langle \mathbf{Id} \mid C, w : \tau \; \textit{\textbf{ins}} \; \rho \mid w' = w \rangle \quad \text{when } \mathit{cmp}_{\mathit{opaque}}(\tau, \tau') = \mathbf{eq} \text{ and } \mathit{cmp}_{\mathit{opaque}}(\rho, \rho') = \mathbf{eq} \qquad \text{s10}$$

$$\langle \overline{a} \mid C, w : \tau \; \textit{\textbf{ins}} \, \mathtt{Empty} \rangle \rhd \langle \mathbf{Id} \mid C \mid w = \mathsf{One} \rangle \qquad \text{s11}$$

$$\langle \overline{a} \mid C, w : \tau \; \textit{\textbf{ins}} \; (\texttt{\#})_n \, \overline{v} \, l \rangle \rhd \langle \mathbf{Id} \mid C, w' : \tau \; \textit{\textbf{ins}} \; (\texttt{\#})_{n-1} \, \overline{v}_{\backslash i} \, l \mid w = w' \rangle \quad \text{when } w' \text{ fresh and } \mathit{cmp}_{\mathit{opaque}}(\tau, v_i) = \mathbf{lt} \qquad \text{s12}$$

$$\langle \overline{a} \mid C, w : \tau \; \textit{\textbf{ins}} \; (\texttt{\#})_n \, \overline{v} \, l \rangle \rhd \langle \mathbf{Id} \mid C, w' : \tau \; \textit{\textbf{ins}} \; (\texttt{\#})_{n-1} \, \overline{v}_{\backslash i} \, l \mid w = \mathsf{Inc}\ w' \rangle \quad \text{when } w' \text{ fresh and } \mathit{cmp}_{\mathit{opaque}}(\tau, v_i) = \mathbf{gt} \qquad \text{s13}$$

$$\langle \overline{a} \mid C, w : \tau \; \textit{\textbf{ins}} \; (\texttt{\#})_{n-1} \, \overline{v}_{\backslash i} \, l \rangle \rhd \langle \mathbf{Id} \mid C, w' : \tau \; \textit{\textbf{ins}} \; (\texttt{\#})_n \, \overline{v} \, l \mid w = w' \rangle \quad \text{when } w' \text{ fresh and } \mathit{cmp}_{\mathit{opaque}}(\tau, v_i) = \mathbf{lt} \qquad \text{s14}$$

$$\langle \overline{a} \mid C, w : \tau \; \textit{\textbf{ins}} \; (\texttt{\#})_{n-1} \, \overline{v}_{\backslash i} \, l \rangle \rhd \langle \mathbf{Id} \mid C, w' : \tau \; \textit{\textbf{ins}} \; (\texttt{\#})_n \, \overline{v} \, l \mid w = \mathsf{Dec}\ w' \rangle \quad \text{when } w' \text{ fresh and } \mathit{cmp}_{\mathit{opaque}}(\tau, v_i) = \mathbf{gt} \qquad \text{s15}$$

$$\langle \overline{a} \mid C, w : \tau \; \textit{\textbf{ins}} \; \rho \rangle \rhd \langle \mathbf{Id} \mid \mathtt{false} \mid \cdot \rangle \quad \text{when } \mathit{isIn}(\tau, \rho) \qquad \text{s16}$$

Projection

$$\langle \overline{a} \mid C \mathbin{+\!\!+} D \rangle \rhd \langle \theta \mid C \mid B \rangle \quad \text{when } \mathit{fv}_{\emptyset}(D) \cap \mathit{fv}_{\emptyset}(C) = \emptyset,\ \mathit{fv}_{\emptyset}(D) \cap \overline{a} = \emptyset,\ \theta \in \mathit{saturate}(D) \text{ and } \forall \theta' \in \mathit{saturate}(D)\ .\ \mathtt{true} \vdash^{e} \theta'\, D \hookrightarrow B \qquad \text{s17}$$

$$\langle \overline{a} \mid C \mathbin{+\!\!+} D \rangle \rhd \langle \mathbf{Id} \mid \mathtt{false} \mid \cdot \rangle \quad \text{when } \mathit{fv}_{\emptyset}(C) \cap \mathit{fv}_{\emptyset}(D) = \emptyset,\ \mathit{fv}_{\emptyset}(D) \cap \overline{a} = \emptyset, \text{ and } \mathit{saturate}(D) = \emptyset \qquad \text{s18}$$

$$\boxed{\langle \overline{a} \mid C \rangle \rhd^{*} \langle \theta \mid C' \mid B' \rangle}$$

$$\frac{}{\langle \overline{a} \mid C \rangle \rhd^{*} \langle \mathbf{Id} \mid C \mid \cdot \rangle}\ \textsc{sdone}$$

$$\frac{\langle \overline{a} \mid C \rangle \rhd \langle \theta \mid C'' \mid B \rangle \quad \langle \overline{a} \cup \bigcup_{a \in \overline{a}} \mathit{fv}_{\emptyset}(\theta\ a) \mid C'' \rangle \rhd^{*} \langle \theta' \mid C' \mid B' \rangle}{\langle \overline{a} \mid C \rangle \rhd^{*} \langle \theta' \circ \theta \mid C' \mid B' \mathbin{+\!\!+} B \rangle}\ \textsc{sstep}$$

Figure 5.5: Simplification of λ^{TIR} constraints (part 2 of 2)

The reader will notice rule S9 is missing from Figures 5.4 and 5.5. We shall have more to say on this in Section 5.4.

Rules S10–S15 simplify insertion constraints, which may involve binding a witness variable of C. They are an immediate consequence of the entailment rules MREF, MEMPTY, MEXP, MINC, MCONT and MDEC respectively. Rule S16 signals failure when a type obviously cannot be inserted into a row.

Finally, rules S17 and S18 implement a weak form of *constraint projection* [79]. Projection is a more aggressive form of simplification for constraints which are known to be self contained. These rules are the only ones to make use of $\overline{a}$, a set of type variables, given as input to the simplifier. We intend $\overline{a}$ to contain all those free variables of C which are

"visible" outside of C; that is, which may be further constrained as type inference proceeds. Indeed, the ISIMP rule takes $\overline{a}$ to be $fv_{\emptyset}(\theta_1\ \Gamma) \cup fv_{\emptyset}(\tau)$.

These two rules apply only when the current constraint may be partitioned into two constraints, C and D, such that no type variable is shared between them, and D contains no "visible" type variables. In this case, the simplifier is free to choose an *arbitrary* substitution, θ, s.t. D is satisfied, provided that any witnesses for D do not depend on θ. In other words, the simplifier may do what it wishes with D provided any choices it makes cannot be observed. In practice, we cannot enumerate *all* possible substitutions, so instead try only those in *saturate*(D).

Rule S18 signals failure if D is unsatisfiable. Notice that this rule could be applied for arbitrary D, regardless of its free variables, but attempting to do so would be prohibitively expensive. Instead, this rule catches the case that *saturate*(D) in rule S17 yields the empty set.

For example, if `c` and `d` are not visible, then the constraint

$$(w : \texttt{a}\ \boldsymbol{ins}\, \texttt{b}), (\texttt{c \# d \# Empty})\ \boldsymbol{eq}\ (\texttt{Int \# Bool \# Empty})$$

may be simplified by eliminating the equality constraint. Without rule S17, this equality constraint would propagate all the way to the top level of the program and cause an error. By contrast, the constraint

$$(w : \texttt{a}\ \boldsymbol{ins}\, \texttt{b}), (w' : \texttt{c}\ \boldsymbol{ins}\ (\texttt{d \# Int \# Empty}))$$

cannot be further simplified, since there is no single binding for w' which is consistent with all bindings for `c` and `d`. In this case, the program is inherently ambiguous, and an error may be reported.

Roughly speaking, the judgement $\langle \overline{a} \mid C \rangle \triangleright^* \langle \theta \mid C' \mid B' \rangle$ takes the transitive closure of $\langle \overline{a} \mid C \rangle \triangleright \langle \theta \mid C' \mid B \rangle$, modulo the need to recalculate $\overline{a}$ as type variables become bound by unification steps.

There is a considerable gap between the rules as presented here and a simplification *algorithm*:

(i) This formulation of the simplifier is non-deterministic. More than one rule may be appropriate for a given constraint, and there is no guarantee of confluence since different choices may yield different final constraints.

However, this non-determinism affords the implementor the greatest flexibility in adopting heuristics to guide the simplification process, and avoids much extraneous detail inessential to the correctness of type inference.

(ii) There is no metric m on constraints such that $\langle \ldots \mid C \rangle \triangleright^* \langle \ldots \mid C' \mid \ldots \rangle$ implies $m(C') < m(C)$. To see why, notice rules S12 and S14 (or S13 and S15) allow a member of a row to be removed and then reinserted, thus making no progress in simplifying C.

However, this possible non-termination is easily avoided by merging rules S10–S15 into a single composite rule which considers all members of an insertion constraint simultaneously. But again, this composite approach is more difficult to reason with.

(iii) The simplifier does not necessarily yield constraints in a *simplified form*. As in HM(X) [79], we say C is in simplified form if $C \vdash^e \tau \; eq \; \upsilon$ implies $\mathtt{true} \vdash^e \tau \; eq \; \upsilon$ for all τ and υ. Unfortunately, requiring the simplifier to yield only constraints in simplified form would be prohibitively expensive, since it would require a brute-force enumeration of all most-general unifiers.

For example, the constraint

$$(\mathtt{a} \; \# \; \mathtt{b}) \; eq \; (\mathtt{Int} \; \# \; \mathtt{Bool} \; \# \; \mathtt{Empty}), (\mathtt{a} \; \# \; \mathtt{c}) \; eq \; (\mathtt{Int} \; \# \; \mathtt{Char} \; \# \; \mathtt{Empty})$$

has simplified form `true` with residual substitution

$$[\mathtt{a} \mapsto \mathtt{Int}, \mathtt{b} \mapsto \mathtt{Bool} \; \# \; \mathtt{Empty}, \mathtt{c} \mapsto \mathtt{Char} \; \# \; \mathtt{Empty}]$$

but this can be determined only by looking at both equality constraints simultaneously. However, for simplicity and practicality, none of the simplifier rules look at more than one equality constraint at a time.

Not being able to assume all constraints are in simplified form shall complicate the proof of completeness in the sequel, but not intractably so.

The following lemma shows that the simplifier preserves the satisfiability of constraints, binds witnesses consistently with entailment, and never over-commits to a solution by binding a type variable which should remain free.

Lemma 5.1 If $\langle \overline{a} \mid C_1 \rangle \triangleright^* \langle \theta_1 \mid C_2 \mid B_1 \rangle$ then

(i) $C_2 \vdash^e \theta_1 \; C_1 \hookrightarrow B_2$ where if $\eta_1 \models \theta_2 \; C_2$ then $env(B_2, \eta_1) = env(B_1, \eta_1)_{\restriction names(C_1)}$;

(ii) $\theta_1 \; C_1 \vdash^e C_2 \hookrightarrow B_3$; and

(iii) if $\eta_2 \models \theta_3 \; C_1$ then there exists a θ_4 s.t.

(iii.1) $\theta_{3 \restriction \overline{a} \cup fv_\emptyset(C_2)} \equiv_\emptyset (\theta_4 \circ \theta_1)_{\restriction \overline{a} \cup fv_\emptyset(C_2)}$

(iii.2) $\eta_2 \models \theta_4 \circ \theta_1 \; C_1$

(iii.3) $env(B_3, \eta_2) \models \theta_4 \; C_2$ (where B_3 is from (ii) above)

Proof See Lemma C.5 for the precise theorem statement and its proof. Notice the restriction of the domain of θ' in (iii.1) is essential lest rule S17 break the theorem. □

5.3 Correctness

It is straightforward to show soundness of type inference with respect to type checking.

Theorem 5.2 (Soundness of Inference) If $\theta \mid C \mid \Gamma \vdash t : \tau$ and $saturate(C) \neq \emptyset$ then there exists a Δ s.t. $\Delta \mid C \mid \theta \, \Gamma \vdash t : \tau$.

Proof See Theorem C.10 for the full theorem statement and proof. □

We now consider completeness of inference with respect to type checking. In the previous section we saw the difficulty of implementing a simplifier guaranteed to yield constraints in simplified form. Furthermore, in Section 4.4.4, we saw that the proof-theoretic entailment relation $\vdash^e$ is incomplete with respect to the model-theoretic relation $\Vdash^e$. Both of these aspects shall complicate both the notion of completeness, and its proof.

The first step is to define an *instantiation ordering*, $\preceq$, on *type schemes in context* of the form $(D \mid \sigma)$. Here D is a *global* constraint which does not contain any of the quantified variables of σ. We call the constraint within σ a *local* constraint. The pair $(D \mid \sigma)$ is typically the result of generalisation; indeed we define

$$\begin{array}{l} \mathit{genscheme}(C \mid \Gamma \mid \tau) = (D_1 \mid \texttt{forall}\ \Delta\ .\ \mathit{anon}(D_2)\ \texttt{=>}\ \tau) \\ \quad \text{where } (D_1 \mid \Delta \mid D_2) = \mathit{gen}(C \mid \Gamma \mid \tau) \end{array}$$

Roughly, we intend $(D_1 \mid \sigma_1) \preceq (D_2 \mid \sigma_2)$ when σ_1 is an instance of σ_2, subject to the global constraints D_1 and D_2. (Note that our orientation of $\preceq$ follows that of OML [48], but is the transpose of the ordering in HM(X) [79].)

Jones' approach [48] is to relate schemes by their *ground instances*:

$$\begin{array}{l} (D_1 \mid \texttt{forall}\ \Delta_1\ .\ C_1\ \texttt{=>}\ \tau_1) \preceq^J (D_2 \mid \texttt{forall}\ \Delta_2\ .\ C_2\ \texttt{=>}\ \tau_2) \iff \\ \quad \forall \vdash \theta_1 : \Delta_1 \to \Delta_{init}\ . \\ \quad\quad \texttt{true} \vdash^e D_1 +\!\!+ (\theta_1\ C_1) \Longrightarrow \\ \quad\quad\quad \exists \vdash \theta_2 : \Delta_2 \to \Delta_{init}\ . \\ \quad\quad\quad\quad (\texttt{true} \vdash^e D_2 +\!\!+ (\theta_2\ C_2) \\ \quad\quad\quad\quad \wedge\ \mathit{cmp}_\emptyset(\theta_1\ \tau_1, \theta_2\ \tau_2) = \mathbf{eq}) \end{array}$$

(Actually, Jones generalises this definition slightly by replacing `true` with an arbitrary but fixed ground constraint.)

Though conceptually simple, and pleasingly easy to reason with, this instantiation ordering is too *coarse* for λ^{TIR} constraints. For example

$$(\texttt{a}\ \mathit{eq}\ \texttt{Int} \mid \texttt{forall b . b}\ \mathit{eq}\ \texttt{a => b}) \preceq^J (\texttt{a}\ \mathit{eq}\ \texttt{Bool} \mid \texttt{forall b . b}\ \mathit{eq}\ \texttt{a => b})$$

holds vacuously. Hence a proof of completeness built upon $\preceq^J$ would be too weak.

The approach in HM(X) [79] is more promising, as it takes account of type variables shared between global and local constraints. It is defined as:

$$\begin{array}{l} (D_1 \mid \texttt{forall}\ \Delta_1\ .\ C_1\ \texttt{=>}\ \tau_1) \preceq^H (D_2 \mid \texttt{forall}\ \Delta_2\ .\ C_2\ \texttt{=>}\ \tau_2) \iff \\ \quad (D_1 \vdash^e D_2 \\ \quad \wedge\ \mathit{satisfiable}(D_1 +\!\!+ C_1) \\ \quad \wedge\ \exists \vdash \theta_2 : \Delta_2 \to \Delta' +\!\!+ \Delta_1\ . \\ \quad\quad D_1 +\!\!+ C_1 \vdash^e (\theta_2\ C_2) +\!\!+ \tau_1\ \mathit{eq}\ (\theta_2\ \tau_2)) \end{array}$$

(This definition assumes, without loss of generality, that Δ' contains the free variable of D_1 and D_2, and that Δ', Δ_1, and Δ_2 are distinct.)

Now we find

$$(\texttt{a}\ \mathit{eq}\ \texttt{Int} \mid \texttt{forall b . b}\ \mathit{eq}\ \texttt{a => b}) \not\preceq^H (\texttt{a}\ \mathit{eq}\ \texttt{Bool} \mid \texttt{forall b . b}\ \mathit{eq}\ \texttt{a => b})$$

Unfortunately, even though

$$(\texttt{a}\ \textit{eq}\ \texttt{Int} \mid \texttt{forall}\ \cdot\ .\ \texttt{true => a}) \preceq^H (\texttt{a}\ \textit{eq}\ \texttt{Int} \mid \texttt{forall c . Int}\ \textit{eq}\ \texttt{c => c})$$

we find

$$(\texttt{true} \mid \texttt{forall b . (a, b)}\ \textit{eq}\ \texttt{(Int, Char) => a}) \not\preceq^H (\texttt{a}\ \textit{eq}\ \texttt{Int} \mid \texttt{forall c . Int}\ \textit{eq}\ \texttt{c => c})$$

Thus, $\preceq^H$ is sensitive to whether constraints, in this case `(a, b)` *eq* `(Int, Char)`, are simplified before generalisation. Since we have already stated we cannot make any such assumptions, we conclude $\preceq^H$ is too *fine* a relation for λ^{TIR}.

Thankfully, there is a simple way out of this dilemma. Roughly speaking (the precise definition must also take account of constraint witnesses and inheritable constraints), our ordering is

$$\begin{array}{l}(D_1 \mid \texttt{forall}\ \Delta_1\ .\ C_1 \texttt{ => } \tau_1) \preceq (D_2 \mid \texttt{forall}\ \Delta_2\ .\ C_2 \texttt{ => } \tau_2) \iff \\ \quad \textit{satisfiable}(D_1 +\!\!+ C_1) \\ \quad \wedge \exists \vdash \theta : \Delta_2 \to \Delta' +\!\!+ \Delta_1\ . \\ \qquad D_1 +\!\!+ C_1 \vdash^e D_2 +\!\!+ (\theta\ C_2) +\!\!+ \tau_1\ \textit{eq}\ \theta\ \tau_2\end{array}$$

Now we find

$$(\texttt{true} \mid \texttt{forall b . (a, b)}\ \textit{eq}\ \texttt{(Int, Char) => a}) \preceq (\texttt{a}\ \textit{eq}\ \texttt{Int} \mid \texttt{forall c . Int}\ \textit{eq}\ \texttt{c => c})$$

By inspection, $\preceq$ is not sensitive to how a constraint is split into a global and local component by *gen*. Thus, in λ^{TIR}, *constraint splitting is merely an optimisation*, and is not required for completeness.

With the notion of instantiation ordering fixed, we now turn to formalising the statement of completeness. Roughly speaking, we require every valid typing for a term t to be an instance of every valid inferred type of t. More formally, and as a first approximation, we require that if

$$\Delta \mid C_1 \mid \theta_1\ \Gamma \vdash t : \tau_1$$

is derivable in the type-checking system, then there exists (at least one) derivation

$$\theta_2 \mid C_2 \mid \Gamma \vdash t : \tau_2$$

in the type-inference system, and there exists a θ_3, such that

$$\textit{genscheme}(C_1 \mid \theta_1\ \Gamma \mid \tau_1) \preceq \theta_3\ \textit{genscheme}(C_2 \mid \theta_2\ \Gamma \mid \tau_2)$$

and

$$\theta_1 \equiv_\emptyset \theta_3 \circ \theta_2$$

(Furthermore, these properties must hold for every such type-inference derivation.) However this statement is too strong for λ^{TIR}.

To see the problem, consider the type checking derivation:

$$\frac{\ldots \vdash \texttt{f : Int -> Int} \quad \ldots \vdash \texttt{x : a} \quad \texttt{a}\ \textit{eq}\ \texttt{Int} \vdash^e \texttt{Int -> Int}\ \textit{eq}\ \texttt{a -> a}}{\texttt{a : Type} \mid \texttt{a}\ \textit{eq}\ \texttt{Int} \mid \texttt{f : Int -> Int, x : a} \vdash \texttt{f x : a}}\ \text{APP}$$

One matching type inference derivation is:

$$\frac{\ldots \vdash \texttt{f} : \texttt{Int -> Int} \quad \ldots \vdash \texttt{x} : \texttt{a} \quad \texttt{b} : \mathsf{Type\ fresh}}{\mathbf{Id} \mid \texttt{Int -> Int}\ \textit{eq}\ \texttt{a -> b} \mid \texttt{f} : \texttt{Int -> Int}, \texttt{x} : \texttt{a} \vdash \texttt{f x} : \texttt{b}} \text{ IAPP}$$

To connect these derivations, we need only show that

$$\begin{array}{l} \textit{genscheme}(\texttt{a}\ \textit{eq}\ \texttt{Int} \mid \texttt{f} : \texttt{Int -> Int}, \texttt{x} : \texttt{a} \mid \texttt{a}) = \\ \quad (\texttt{a}\ \textit{eq}\ \texttt{Int} \mid \texttt{forall} \cdot \texttt{. true => a}) \end{array}$$

and

$$\begin{array}{l} \textit{genscheme}(\texttt{Int -> Int}\ \textit{eq}\ \texttt{a -> b} \mid \texttt{f} : \texttt{Int -> Int}, \texttt{x} : \texttt{a} \mid \texttt{b}) = \\ \quad (\texttt{true} \mid \texttt{forall b . Int -> Int}\ \textit{eq}\ \texttt{a -> b => b}) \end{array}$$

are related under $\preceq$. So far all is well.

However, another possible type inference derivation applies rule ISIMP to the above conclusion to yield:

$$\frac{\dfrac{\ldots}{\ldots \vdash \texttt{f x} : \texttt{b}} \quad \langle \{\texttt{a}, \texttt{b}\} \mid \texttt{Int -> Int}\ \textit{eq}\ \texttt{a -> b} \rangle \rhd^* \langle [\texttt{a} \mapsto \texttt{Int}, \texttt{b} \mapsto \texttt{Int}] \mid \texttt{true} \mid \cdot \rangle}{[\texttt{a} \mapsto \texttt{Int}] \mid \texttt{true} \mid \texttt{f} : \texttt{Int -> Int}, \texttt{x} : \texttt{a} \vdash \texttt{f x} : \texttt{Int}} \text{ ISIMP}$$

Again, we must show that

$$(\texttt{a}\ \textit{eq}\ \texttt{Int} \mid \texttt{forall} \cdot \texttt{. true => a})$$

and

$$\begin{array}{l} \textit{genscheme}(\texttt{true} \mid \texttt{f} : \texttt{Int -> Int}, \texttt{x} : \texttt{a} \mid \texttt{Int}) = \\ \quad (\texttt{true} \mid \texttt{forall} \cdot \texttt{. true => Int}) \end{array}$$

are related under $\preceq$. But we must also show (taking $\theta_1 = \mathbf{Id}$) that there exists a θ_3 such that

$$\mathbf{Id} \equiv_\emptyset \theta_3 \circ [\texttt{a} \mapsto \texttt{Int}]$$

which is clearly impossible.

The problem is that the simplifier may bind free type variables within Γ. Thankfully, we may show this happens only when such type variables are similarly constrained within the type-checking derivation. In the example above, even though `Int` was substituted for `a`, this substitution was entailed by the constraint `a` $\textit{eq}$ `Int`.

Thus, the refined (but still only approximate—see below) statement of completeness weakens the requirement

$$\theta_1 \equiv_\emptyset \theta_3 \circ \theta_2$$

to

$$\forall a \in \mathit{fv}_\emptyset(\Gamma)\ .\ C_1 \vdash^e (\theta_1\ a)\ \textit{eq}\ (\theta_3 \circ \theta_2\ a)$$

One final subtlety is that because $\vdash^e$ is incomplete, we must show completeness using its model-theoretic counterpart $\Vdash^e$.

The remainder of this section develops these ideas formally. Unlike the other theorems in this dissertation, we shall elide the actual proof of completeness. This is partly because of

time constraints, and partly because we plan to redo the proofs using a variation on the definitions they are built upon (see Section 5.4).

We first define

$$Env(C) = \{\eta \mid \forall (w : c) \in C \,.\, \eta\; w \in \mathcal{I}\}$$

and similarly

$$Env(\Gamma) = \{\eta \mid \forall (x : \sigma) \in \Gamma \,.\, \eta\; x \in \mathbf{E}\; \mathcal{V}\}$$

Let $\Delta' \vdash D_1/D_2$ constraint, $\Delta' +\!\!+ \Delta_1 \vdash C_1$ constraint, $\Delta' +\!\!+ \Delta_2 \vdash C_2$ constraint, $\Delta' +\!\!+ \Delta_1 \vdash \tau_1 : \mathsf{Type}$, and $\Delta' +\!\!+ \Delta_2 \vdash \tau_2 : \mathsf{Type}$. Furthermore, let $dom(\Delta_1) \cap dom(\Delta_2) = \emptyset$. Then we define the *expanded instantiation ordering* as

$$\begin{array}{l}
\vdash (D_1 \mid \Delta_1 \mid C_1 \mid \tau_1) \preceq (D_2 \mid \Delta_2 \mid C_2 \mid \tau_2) \hookrightarrow B \Longleftrightarrow \\
\quad (inhs(D_1) = D_1 \\
\quad \wedge\; inhs(D_2) = D_2 \\
\quad \wedge\; satisfiable(D_1 +\!\!+ C_1) \\
\quad \wedge\; \exists \vdash \theta : \Delta_2 \to \Delta' +\!\!+ \Delta_1 \,. \\
\qquad D_1 +\!\!+ C_1 \Vdash^e D_2 +\!\!+ \theta\; C_2 +\!\!+ \tau_1\; \mathbf{eq}\, \theta\; \tau_2 \hookrightarrow B)
\end{array}$$

We may extend the relation above to type contexts as follows. We let ϕ range over finite maps from variables to triples $(B \mid \overline{w} \mid \overline{w'})$. Let $\Delta' \vdash \Gamma_1/\Gamma_2$ context and $\Delta' \vdash D_1/D_2$ constraint. Then we define the *type context ordering* as

$$\begin{array}{l}
\vdash (D_1 \mid \Gamma_1) \preceq (D_2 \mid \Gamma_2) \hookrightarrow \phi \Longleftrightarrow \\
\quad dom(\Gamma_1) = dom(\Gamma_2) \\
\quad \wedge\; ((x : \mathtt{forall}\; \Delta_1 \;.\; C_1 \Rightarrow \tau_1) \in \Gamma_1 \wedge (x : \mathtt{forall}\; \Delta_2 \;.\; C_2 \Rightarrow \tau_2) \in \Gamma_2) \Longrightarrow \\
\qquad (\phi\; x = (B \mid \overline{w} \mid \overline{w'}) \\
\qquad \wedge \vdash (D_1 \mid \Delta_1 \mid C_1' \mid \tau_1) \preceq (D_2 \mid \Delta_2 \mid C_2' \mid \tau_2) \hookrightarrow B \\
\qquad \wedge\; C_1' = named(C_1) \text{ s.t. } names(C_1') = \overline{w} \\
\qquad \wedge\; C_2' = named(C_2) \text{ s.t. } names(C_2') = \overline{w'})
\end{array}$$

Let $\Delta' \vdash \sigma$ scheme, where $\sigma = \mathtt{forall}\; \Delta \;.\; C \Rightarrow \tau$. Let $\Delta' \vdash D$ constraint. Then we say $(D \mid \sigma)$ is *unambiguous* if

$$\begin{array}{l}
\forall \Delta' \vdash D' \text{ constraint}, \vdash \theta_1 : \Delta \to \Delta', \vdash \theta_2 : \Delta \to \Delta', \vdash \theta' : \Delta' \to \Delta_{init} \,. \\
\quad (D' \Vdash^e D +\!\!+ (\theta_1\; C) \hookrightarrow B_1 \\
\quad \wedge\; D' \Vdash^e D +\!\!+ (\theta_2\; C) \hookrightarrow B_2 \\
\quad \wedge\; D' \Vdash^e \theta_1\; \tau\; \mathbf{eq}\, \theta_2\; \tau \\
\quad \wedge\; \eta \models \theta'\; D') \Longrightarrow \\
\qquad env(B_1, \eta) = env(B_2, \eta)
\end{array}$$

$$
\begin{aligned}
\bot &=^{\tau} \bot \\
[\mathsf{int} : i] &=^{\mathtt{Int}} [\mathsf{int} : j] \iff i = j \\
[\mathsf{func} : f] &=^{\tau\ \texttt{->}\ \upsilon} [\mathsf{func} : g] \iff v =^{\tau} v' \Longrightarrow f\ v =^{\upsilon} g\ v' \\
[\mathsf{prod}_n : \langle v_1, \ldots, v_n \rangle] &=^{\mathtt{All}\ ((\texttt{\#})_n\ \overline{\tau}\ \mathtt{Empty})} [\mathsf{prod}_{n'} : \langle v'_1, \ldots, v'_{n'} \rangle] \iff n = n' \wedge \forall i\ .\ v_i =^{\tau_{\pi\ i}} v'_i \\
[\mathsf{inj} : \langle i, v \rangle] &=^{\mathtt{One}\ ((\texttt{\#})_n\ \overline{\tau}\ \mathtt{Empty})} [\mathsf{inj} : \langle j, v' \rangle] \iff i = j \wedge v =^{\tau_{\pi\ i}} v' \\
&\text{where } \pi \in \mathit{sortingPerms}(\tau_1, \ldots, \tau_n) \\
[\mathsf{ifunc}_n : f] &=^{\mathtt{forall}\ \Delta\ .\ C\ \texttt{=>}\ \tau} [\mathsf{ifunc}_{n'} : g] \iff \\
n = n' \wedge (\vdash \theta : \Delta \to \Delta_{init} &\wedge \eta \models \theta\ C \Longrightarrow f\ (\eta\ w_1, \ldots, \eta\ w_n) =^{\theta\ \tau} g\ (\eta\ w_1, \ldots, \eta\ w_n))
\end{aligned}
$$

Figure 5.6: The logical relation $=$ on $\mathbf{E}\ \mathcal{V} \times \mathbf{E}\ \mathcal{V}$ indexed by λ^{TIR} monotypes of kind Type

Let $\vdash (D_1 \mid \Delta_1 \mid C_1 \mid \tau_1) \preceq (\mathtt{true} \mid \Delta_2 \mid C_2 \mid \tau_2) \hookrightarrow B$, and let $\eta \in \mathit{Env}(D_1)$, $\mathit{names}(C_1) = \overline{w}$, and $\mathit{names}(C_2) = \overline{w'}$. Then we define

$$
\begin{aligned}
\mathit{coerce}(B \mid \overline{w} \mid \overline{w'})\ \eta = \lambda v\ .\ & \mathbf{let_E}\ v' \leftarrow v \\
& \mathbf{in\ case}\ v'\ \mathbf{of}\ \{ \\
& \quad \mathsf{ifunc}_{n'} : f \to \mathbf{unit_E}\ (\mathsf{ifunc} : g); \\
& \quad\quad \text{where} \\
& \quad\quad\quad g = \lambda(y_1, \ldots, y_n)\ .\ f\ ([\![w'_1]\!]_{\eta'}, \ldots, [\![w'_{n'}]\!]_{\eta'}) \\
& \quad\quad\quad \eta' = \mathit{env}(B, \eta_{\restriction \mathit{names}(D_1)} \mathbin{+\!\!+} [w_1 \mapsto y_1, \ldots, w_n \mapsto y_n]) \\
& \quad \mathbf{otherwise} \to \mathbf{unit_E}\ (\mathsf{wrong} : *)\ \}
\end{aligned}
$$

Notice that if $\eta \in \mathit{Env}(D)$ then $\mathit{coerce}(B)\ \eta \in \mathbf{E}\ \mathcal{V} \to \mathbf{E}\ \mathcal{V}$.

Let $\vdash (D_1 \mid \Gamma_1) \preceq (\mathtt{true} \mid \Gamma_2) \hookrightarrow \phi$. Then we extend *coerce* to ϕ as follows:

$$\mathit{coerce}(\phi)\ \eta = \lambda \eta'\ .\ x \mapsto (\mathit{coerce}(\phi\ x)\ \eta)\ (\eta'\ x)$$

Notice that if $\eta \in \mathit{Env}(D)$ then $\mathit{coerce}(\phi)\ \eta \in \mathit{Env}(\Gamma_2) \to \mathit{Env}(\Gamma_1)$.

Finally, Figure 5.6 defines a logical relation on $\mathbf{E}\ \mathcal{V} \times \mathbf{E}\ \mathcal{V}$ indexed by types τ such that $\Delta_{init} \vdash \tau : \mathsf{Type}$.

Theorem 5.3 (Completeness of Inference) Let Δ_1, Δ_2, Γ_1, Γ_2, C_1, t, τ_1, T_1 and ϕ be s.t.

(a) $\Delta_1 \vdash C_1$ constraint and $\Delta_1 \vdash \Gamma_1$ context

(b) $\mathit{satisfiable}(C_1)$

(c) $\Delta_1 \mid C_1 \mid \Gamma_1 \vdash t : \tau_1 \hookrightarrow T_1$

(d) $\Delta_2 \vdash \Gamma_2$ context

(e) $\Delta_1 \cup \Delta_2 \vdash \theta_1$ subst, $\mathit{dom}(\theta_1) \subseteq \mathit{fv}_{\emptyset}(\Gamma_2)$, $\mathit{rng}(\theta_1) \subseteq \mathit{dom}(\Delta_1)$

(f) $\vdash (\mathit{inhs}(C_1) \mid \Gamma_1) \preceq (\mathtt{true} \mid \theta_1\ \Gamma_2) \hookrightarrow \phi$

Then there exists θ_2, C_2, τ_2 and T_2 s.t.

(i) $\theta_2 \mid C_2 \mid \Gamma_2 \vdash t : \tau_2 \hookrightarrow T_2$

and for every θ_2, C_2, τ_2 and T_2, s.t. (i) holds, there exists a Δ_3, θ_3, and B_1 s.t.

(ii) $(\Delta_1 \cup \Delta_2) +\!\!+ \Delta_3 \vdash \theta_3 \text{ subst}$, $dom(\theta_3) \subseteq fv_\emptyset(\theta_2\, \Gamma_2)$, $rng(\theta_3) \subseteq dom(\Delta_1)$

(iii) If $gen(C_1 \mid \Gamma_1 \mid \tau_1) = (D_1 \mid \Delta_4 \mid D_2)$ and $gen(C_2 \mid \theta_2\, \Gamma_2 \mid \tau_2) = (D_3 \mid \Delta_5 \mid D_4)$ then

$$\vdash (D_1 \mid \Delta_4 \mid D_2 \mid \tau_1) \preceq (\theta_3\, D_3 \mid \Delta_5 \mid \theta_3\, D_4 \mid \theta_3\, \tau_2) \hookrightarrow B_1$$

(iv) $\forall a \in fv_\emptyset(\Gamma_2)\, .\, inhs(C_1) \Vdash^e \theta_3 \circ \theta_2\; a\; eq\, \theta_1\; a$

(v) Furthermore, let θ_4, η, and B_2 be s.t.

(g) $\Delta_1 \vdash \theta_4 \text{ subst}$
(h) $\Delta_{init} \vdash \theta_4 \circ \theta_3 \circ \theta_2\, \Gamma_2 \text{ context}$
(i) $\Delta_{init} \vdash \theta_4\, C_1 \text{ constraint}$
(j) $\eta \models \theta_4 \circ \theta_3 \circ \theta_2\, \Gamma_2$
(k) $env(B_2) \models \theta_4\, C_1$

Then

$$[\![T_1]\!]_{(coerce(\phi)\; env(B_2)\; \eta) +\!\!+ env(B_2)} =^{(\theta_4\, \tau_1)} [\![T_2]\!]_{\eta +\!\!+ env(B_1, env(B_2))}$$

Proof By laborious induction on (c), and by showing rule ISIMP preserves properties (ii)–(v) of its hypothesis inference judgement. The theorem could be slightly simplified by separating completeness (properties (i)–(iv)) and coherence (property (v)). However, this separation would duplicate the exceedingly tedious setup of (a)–(f). Hence it seems simpler to merge completeness and coherence into a single *übersatz*. □

As a corollary to Theorem 5.3 we may show that if t has an unambiguous principal type, then all possible type-checking derivations of t yield run-time terms which are related by the logical relation of Figure 5.6.

Furthermore, by Theorem 5.2 and Theorem 5.3 we may show all the principal types of a term are equivalent under the instantiation ordering.

5.4 Row Extension

Recall from Section 2.4 that another way of simplifying a row equality constraint $\rho\; eq\, \rho'$ is to allow a type in ρ' to extend the (open) tail of ρ. This simplification is valid only when the chosen type within ρ' cannot be matched with any type within ρ. Formally, we may define the rule:

$$\begin{array}{ll} \langle \overline{a} \mid C, (\#)_m\, \overline{\tau}\; b\; eq_{\mathsf{Row}}\; (\#)_n\, \overline{v}\; l \rangle & \\ \quad \rhd \langle [b \mapsto v_j \,\#\, b'] \mid (C, (\#)_m\, \overline{\tau}\; b'\; eq_{\mathsf{Row}}\; (\#)_{n-1}\, \overline{v}_{\setminus j}\; l)[b \mapsto v_j \,\#\, b'] \mid \cdot \rangle & \text{S9} \\ \quad \text{when } b \notin fv_\emptyset(v_j),\; b' : \mathsf{Row} \text{ fresh and } notIn(C \vdash v_j, (\#)_m\, \overline{\tau}\; \mathtt{Empty}) & \end{array}$$

Notice that the result constraint contains a fresh type variable, b'.

For example, this rule would rewrite (in two steps)

$$\texttt{(Int \# a)}\ \mathit{eq}\ \texttt{(Bool \# b)}$$

to `true`, with the residual substitution

$$[\texttt{a} \mapsto \texttt{Bool \# c}, \texttt{b} \mapsto \texttt{Int \# c}]$$

where `c` is fresh.

Unfortunately, though rule S9 seems both desirable (it reduces the size of constraints) and reasonable (it preserves the ground instances of constraints), it is *not* compatible with our instantiation ordering.

For example, consider the term:

```
{ \(Inj x) . 1 - x;
  \(Inj y) . if y then 0 else 1 }
```

This term may be assigned the type scheme:

$$\begin{aligned}\sigma_1 = {} & \texttt{forall (a : Row) (b : Row) .} \\ & \quad \texttt{Int}\ \mathit{ins}\ \texttt{a}, \texttt{Bool}\ \mathit{ins}\ \texttt{b}, \texttt{(Int \# a)}\ \mathit{eq}\ \texttt{(Bool \# b) =>} \\ & \quad\quad \texttt{One (Int \# a) -> Int}\end{aligned}$$

Were the simplifier to be augmented by rule S9, this term could also be assigned the more intuitive scheme:

$$\begin{aligned}\sigma_2 = {} & \texttt{forall (c : Row) .} \\ & \quad \texttt{Int}\ \mathit{ins}\ \texttt{c}, \texttt{Bool}\ \mathit{ins}\ \texttt{c =>} \\ & \quad\quad \texttt{One (Int \# Bool \# c) -> Int}\end{aligned}$$

However, though we have $\sigma_2 \preceq \sigma_1$, we find that $\sigma_1 \not\preceq \sigma_2$. In particular, there is no τ such that:

$$\begin{aligned}& \texttt{Int}\ \mathit{ins}\ \texttt{a}, \texttt{Bool}\ \mathit{ins}\ \texttt{b}, \texttt{(Int \# a)}\ \mathit{eq}\ \texttt{(Bool \# b)} \vdash^e \\ & \quad [\texttt{c} \mapsto \tau]\ (\texttt{Int}\ \mathit{ins}\ \texttt{c}, \texttt{Bool}\ \mathit{ins}\ \texttt{c}) +\!\!+ \\ & \quad \texttt{(One (Int \# a) -> Int)}\ \mathit{eq} [\texttt{c} \mapsto \tau] \texttt{(One (Int \# Bool \# c) -> Int)}\end{aligned}$$

Hence, rule S9 does not preserve the invariant necessary for the proof of completeness in Theorem 5.3. For this reason we have removed rule S9 from Figure 5.4. However, the real problem is that our invariant is too strong.

The solution appears to be to generalise the instantiation ordering of Section 5.1 by replacing the existentially quantified substitution, θ, on the left-hand side of $\Vdash^e$, with an existentially quantified constraint, C_3, on the right-hand side of $\Vdash^e$. Of course, C_3 cannot be *any* constraint: We require that C_3 does not "disturb" (change the satisfying substitutions of) the constraint $D_1 +\!\!+ C_1$.

Returning to the example we find that $\sigma_1 \preceq \sigma_2$ under this generalised instantiation ordering, because:

$$\begin{aligned}& \texttt{Int}\ \mathit{ins}\ \texttt{a}, \texttt{Bool}\ \mathit{ins}\ \texttt{b}, \texttt{(Int \# a)}\ \mathit{eq}\ \texttt{(Bool \# b)}, \\ & \texttt{a}\ \mathit{eq}\ \texttt{(Bool \# c)} \vdash^e \\ & \quad \texttt{Int}\ \mathit{ins}\ \texttt{c}, \texttt{Bool}\ \mathit{ins}\ \texttt{c}, \\ & \quad \texttt{(One (Int \# a) -> Int)}\ \mathit{eq}\ \texttt{(One (Int \# Bool \# c) -> Int)}\end{aligned}$$

Notice how the introduced constraint, `a` *eq* `(Bool # c)`, allows the type variables `a` and `c` to be related without disturbing the constraint:

`Int` *ins* a, `Bool` *ins* b, `(Int # a)` *eq* `(Bool # b)`

Rule S9 is just one of a number of desirable simplification rules not included in Figures 5.4 and 5.5. For example, the entailment rules MPROJL and MPROJR, sketched in Section 4.4.4, induce two corresponding simplification rules. It is open whether the revised instantiation ordering is also compatible with rules.

At the time of writing we are re-running the proofs of this Chapter under the revised instantiation ordering, and we expect to include these revisions in a journal version of this part of the dissertation. The programme to replace a substitution by a constraint may be applied profitably in a number of other places within the development of λ^{TIR}, including the properties of entailment, and the correctness of the simplifier. A similar programme has been carried out by Sulzmann [101] in the context of HM(X) [79] (though, curiously, the instantiation ordering remains unchanged in his revision).

Chapter 6

Conclusions to Part I

6.1 Related Work

Record Calculi

Wand [112] first introduced *rows* to encode record subtyping (and, in turn, inheritance) using parametric polymorphism, though the system did not enjoy completeness of type inference. Rémy [94] introduced label *presence* and *absence* flags in types, and demonstrated completeness of inference. Variations allowing *record concatenation* [35, 113] rather than just *record extension* were also proposed. Rémy [93] has demonstrated that concatenation may often be encoded using just extension.

Ohori [80] and, independently, Jones [47] developed polymorphic record and variant calculi, and a compilation method which represented records as natural-number indexed vectors. Ohori's system dealt only with closed rows; Jones' system allowed extensible rows. Our system is a strict generalisation of Gaster and Jones' system of polymorphic extensible records [31]. The latter exploits *qualified types* and the *dictionary translation* [47] as a compilation method.

Parallel to the parametric polymorphism approach followed in this work are record calculi based on *subtyping* [16].

Constrained Polymorphism

Odersky *et al.* have developed HM(X) [79] as a framework for constraint-based type inference. It adds to Jones' qualified types the notion of *constraint projection*, and guarantees any constraint domain X enjoying a *principal constraint* property can be lifted to a type-inference system enjoying completeness of type inference. Principle constraints are defined relative to a set $\mathcal{S}$ of constraints in *solved form.*

Since both Ohori's and Gaster and Jones' record calculi are instances of HM(X), we initially hoped λ^{TIR} would be likewise. Unfortunately, the definition of $\mathcal{S}$ for λ^{TIR} constraints appears to be as complicated as the definition of the simplifier itself, and hence not particularly theoretically pleasing. Furthermore, the statement of completeness for HM(X) when $\mathcal{S}$ is smaller than all satisfiable constraints (as it would have to be for λ^{TIR}) further requires that $\mathcal{S}$ contain only those constraints in *simplified* form. As mentioned in Section 5.2, our simplifier is designed *not* to always yield constraints in this form as to do so would require a brute-force enumeration of all most-general unifiers, with concomitant exponential growth in both time and space.

Sulzmann [101] has since generalised the HM(X) framework to address some of these limitations. (The work of this thesis has been done independently of his work on the revised system.) However, there are four aspects of Sulzmann's revised HM(X) which prevent its use for λ^{TIR}. Firstly, his development still depends critically on existential constraints, which, as mentioned in Section 2.9, we find quite technically challenging for λ^{TIR} constraints. Secondly, though his system does not require constraints to be normalised at each step of type inference, his constraint simplification rule still builds upon the notion of solved form, which for λ^{TIR} is as problematic as in the original HM(X). Thirdly, his presentation is in "term-free" form, meaning the inferred type of a term is represented implicitly within the current constraint context rather than explicitly as a type. This notion is unnecessarily complicated for λ^{TIR}. Finally, and in common with the original HM(X), no support is provided for constraint witnesses, which we have seen to be essential to the semantics and implementation of λ^{TIR}.

Our technical development is instead based upon Jones' more general framework for simplifying and improving qualified types [48]. In Jones' system, constraints may be simplified arbitrarily, and his proofs do not rely on constraints being in any solved form. Unfortunately, Jones' instantiation ordering is too coarse for λ^{TIR} constraints which contain "global" type variables (type variables bound at an outer scope). Hence, we have been forced to re-prove most of the correctness of our system from scratch.

Set Constraints

Set constraints are popular in program analysis [5, 4] and in constraint logic programming [100]. The constraint domain of λ^{TIR} resembles that of simple set-constraints with primitive subset constraints and set union. However, set-constraints have an *implicit* idempotency law:

$$a \cup \{b, b\} = a \cup \{b\}$$

whereas in λ^{TIR} this property is enforced by an *explicit* insertion constraint:

$$b \;\textit{ins}\; a$$

Using this explicit form leads directly to our implementation method.

Despite this difference, it may still be possible to exploit some of the implementation techniques developed for set constraints if necessary.

Intersection Types

Type-indexed products bear a superficial resemblance to *intersection* types [89, 95]. (And coproducts to *union* types [7].) However, they differ fundamentally in their meaning, as λ^{TIR} products are not subject to any *coherency* condition with respect to a notion of subtyping.

For example, the intersection type

```
Int -> Int -> Int & Real -> Real -> Real
```

contains only those binary functions which behave coherently on integer or real arguments

with respect to the subtyping relation

```
Int < Real
```

Thus it includes the addition function, but excludes the function which adds integer arguments, but subtracts real arguments.

A first approximation to this type is the λ^{TIR} scheme

```
forall a b .
  a # b eq Int # Real # Empty =>
    a -> a -> a
```

Though it indeed has the two required instances, this type is too small since it contains only those functions which do not depend on whether its arguments are integers or reals. That is, the scheme above is simply an instance of

```
forall a . a -> a -> a
```

The closest λ^{TIR} comes to the intersection type above is the scheme

```
forall a b .
  a ins b,
  a # b eq Int # Real # Empty =>
    a -> a -> a
```

However, this scheme is now too large. In addition to the desired addition function

```
(\(x && _) . x) (intPlus && realPlus && Triv)
```

(see Section 3.5), it also includes the mixed addition and subtraction function

```
(\(x && _) . x) (intPlus && realMinus && Triv)
```

This should come as no surprise: The function above is implemented by a *three-argument* function, the first of which effectively serves to distinguish between the integer and real functions. Hence, λ^{TIR} type-indexed-products are just that: type-indexed, and hence not necessarily coherent.

XML

As mentioned in Chapter 1, XDuce [40] is another functional language with similar goals to XMλ, but built upon subtyping polymorphism instead of parametric polymorphism, and using regular expressions as types instead of type-indexed rows. Regular-expression language containment is used to induce the subtyping relation, and regular expressions are *not* required to be 1-unambiguous. At the time of writing XDuce does not support parametric polymorphism or higher-order functions.

Other proposals for XMλ-like languages build on regular-tree transducers [68] or Haskell [111].

6.2 Conclusions and Future Work

Thanks to its notion of *type-indexed rows*, and its expressive constraint domain of *insertion* and *equality* constraints, λ^{TIR} can naturally encode many programming idioms, including record calculi, anonymous sums and products, and closed-world style overloading. It can be straightforwardly compiled into an untyped run-time language in which type-indexing is reduced to conventional natural-number indexing. These indices are generated and passed at run-time as implicit arguments to let-bound expressions, exactly as occurs in some existing record calculi [31, 80].

For the programs we considered, the constraints were compact and reasonably intuitive. We are working on an implementation of λ^{TIR} within the larger language XMλ [65]. At the time of writing, our XMλ compiler can simplify constraints but not yet infer them. We hope to demonstrate the feasibility of λ^{TIR} on larger programs once this compiler is complete.

In common with most constraint-based type systems, λ^{TIR} constraints could conceivably grow to a size beyond the understanding of a programmer, and beyond the capability of the type inference system to solve. In Section 4.4 we discussed why we do not expect this to be a problem, however our hypothesis remains unverified until we can test it within XMλ. One possibility for aiding the programmer in understand large constraint sets is to use "backwards" β-reduction to replace constraints by programmer-declared abbreviations wherever possible.

Since entailment is incomplete, it is possible that a programmer-supplied type scheme may be an instance of an inferred scheme, but the system is unable to prove it. As discussed in Section 4.4.4, this may be partially redressed by adding projection rules to the $\vdash^m$ judgement to exploit the lexicographic ordering of types. However, we would like to gain some experience with the system before deciding if these potentially more expensive rules are justified.

On the theoretical side, we are currently reworking the development of simplifier correctness and completeness of type inference to use the revised instantiation ordering sketched in Section 5.4. We hope this revised development will not only be flexible enough to support the introduction of new constraint simplification rules, but also simplify the statement of these theorems and their proofs. It should come as no surprise to the reader that the ugliness in the statement of Theorem 5.3 also extends to its proof!

We also hope to complete a complexity analysis of constraint satisfiability, entailment and type inference as a whole. The last is likely to be above **EXP**. However, as complexity class seems to be a poor indicator of the typical performance of type inference systems, our priority rests with completing the implementation.

Part II

Dynamically-Typed Staged Computation

Abstract

This part explores a weak form of program reflection called *staged computation*. It is weak in the sense that code may be constructed at run-time, but not deconstructed (*e.g.*, by pattern matching). However, in exchange for this weakness the system is quite simple, requiring only three additional primitives to *defer*, *splice* and *run* code.

Two distinct forms of code are supported. *Statically typed code* is guaranteed at compile-time to be well typed at run-time, and hence is the most reliable method of generating code at run-time. However, since the type of generated code may sometimes depend on run-time values in a way that is difficult to express statically, the system also allows *dynamically typed code* to be generated. In contrast to statically typed code, dynamically typed code is checked for well-typing as late as possible at run-time; that is, just before it is executed.

We introduce the system using some small examples, and then illustrate its great flexibility by some larger worked examples. We present the formal type checking system, which translates well-typed source terms to an untyped run-time language. The system is greatly complicated by the desire to support constrained polymorphism within generated code: We will spend some time explaining the problems which arise and their solutions. Finally, we present a denotational semantics for the run-time language, and demonstrate a *type soundness* result. We leave type inference for this system to future work.

Chapter 7

Introduction

Programs must often manipulate *intensional* representations of other programs. This is called *reflective programming* when the manipulating program (the *meta-program*) and the program being manipulated (the *object-program*) are expressed in the same language. Examples of reflection abound in compilers, interpreters, partial evaluators and programming environments.

This part of the dissertation develops a weak form of reflection in which intensional representations, which we shall call *code*, may be constructed and executed, but never deconstructed (*e.g.*, by pattern-matching). The result of this restriction is called *staged computation*, since programs which merely construct other programs can be seen as having their evaluation *staged* over two or more phases of execution. Staged computation is much simpler than full reflective programming because it does not require any language-level support for manipulating variable names.

Of course it is possible to effect a staging of program evaluation using ordinary higher-order functions. However, separating execution stages by time (generate in one session and execute in another) or space (generate on one machine and execute on another) requires an intensional representation of generated code to be stored or transmitted over a network. Recovering such a representation from run-time closures is very difficult. Staged computation, on the other hand, makes this operation trivial.

The principal benefit of staged computation over more ad-hoc approaches using strings or datatypes of abstract syntax is the ability to statically verify that all code generated at run-time is not only syntactically valid, but also type-correct. However, sometimes code must be generated whose type, in addition to its contents, depends on run-time values. To support this requires a notion of *dynamically typed code* to complement *statically typed code*. Dynamically typed code must have its type checking deferred till run-time in addition to its evaluation.

Examples of staged computation abound, though they are often hidden within the noise of larger systems:

- **Run-time partial evaluation** generates code at run-time to exploit invariants unknown at compile-time. It has found applications in operating systems [62] and advanced compilers [54]. Run-time partial evaluation may be viewed as a form of staged-computation in which only *closed-code* (code which does not contain free variables) may be generated.

- **Dynamic typing** introduces *dynamic values* which contain both a value and a *run-time* representation of the value's type. Because all dynamic values have the same

compile-time type, they may be treated uniformly by programs such as interpreters, persistent stores, generic programs, and distributed programs which pass code between machines. A dynamic value may be viewed as dynamically typed code whose body happens to be evaluated.

- **Document generation** requires a data structure to be created on-demand by one machine (the *server*), and then transmitted to another (the *client*). If documents are simply strings, the server need only concatenate each document fragment and transmit the result. However, we have seen in Chapter 1 that documents are often structured as XML, and often contain embedded scripts. We call these *dynamic, active documents.*

  Part I of the dissertation showed how XML documents may be represented as typed terms. It is a simple matter to transmit such a term from one machine to another if it contains no functions. Thus dynamic documents are already well supported by the material of Part I. However, it seems natural to express embedded scripts simply as functions or monadic commands within the same functional language as the document itself. Unfortunately, terms containing scripts encoded in this way cannot be easily transmitted. Hence dynamic, active documents are problematic.

  Staged computation solves this problem by allowing the server program to distinguish *server-side* code (executed on the server in response to a request) from *client-side* code (executed on the client after a reply is received). That is, a dynamic, active document is simply a *residual program*, and residual programs are easily transmitted from server to client.

- **Online services** interact with a user via a dialogue of successive dynamic documents. A single server may be interacting with many thousands of users simultaneously, any of whom may decide to stop responding or backtrack to an earlier point in their dialogue. Hence, the crux in implementing these systems is managing each user's *dialogue state.* A particularly simple solution is to embed within each dynamic document an intensional representation of its *continuation code.* When the user wishes to continue the dialogue, the client passes this continuation along with any form data back to the server. Staged computation provides some support for this style of programming.

We shall develop a small calculus, λ^{sc}, which adds to higher-order functions and constrained polymorphism the ability to construct and execute code at run-time. Both statically and dynamically typed code may be intermixed within a single program. Though λ^{sc} has all of the type-theoretic framework necessary to support type-indexed rows, implicit parameters, and indeed any other system of constrained types, for simplicity we put these features aside. However, we shall sometimes assume their inclusion in the extended examples of Chapter 8.

λ^{sc} is most closely related to MetaML [97], which also supports both statically and dynamically typed code. The statically typed component of MetaML grew out of the work of Nielson and Nielson on two-level functional languages [77], Davies and Pfenning [20, 21] on multi-level languages, and Taha and Sheard [104, 107]. The dynamically typed component comes from work of Shields, Sheard and Peyton Jones [99]. However, this is the first formal presentation of a system including both kinds of code, and supporting constrained polymorphism. Indeed, we shall see that constrained polymorphism is the key to effi-

ciently implementing dynamically typed code. Furthermore, we shall show type soundness model-theoretically, rather than proof-theoretically as in earlier work.

The remainder of this chapter introduces the three operators to *defer*, *splice* and *run* code, and demonstrates their *statically* and *dynamically* typed variants. We then illustrate the generality of staging by extended examples of partial evaluation, dynamic typing and distributed computing (Chapter 8). The later develops a small document server, and exploits both dynamically and statically typed code within a single program. We then present the formal system and demonstrate type soundness (Chapter 9).

7.1 Staged Computation

Staged computation introduces three operators to construct, evaluate and combine pieces of programs. These can be used to explicitly distribute the evaluation of a program over many *run-time stages*:

- The *defer* operator, `{{ t }}`, defers evaluation of an expression t by one stage. Writing $\Downarrow$ to denote evaluation:

  ```
  1 + 1 ⇓ 2                    evaluated at stage 0
  {{ 1 + 1 }} ⇓ {{ 1 + 1 }}    deferred till stage 1
  ```

  We call t the *body* of `{{ t }}`, and dually, we call `{{ t }}` the *code* of t. (Note that `{{ t }}` is written as `< t >` in many other staged languages—unfortunately this more concise notation clashes with the syntax for XML used in Part I.)

- The *run* operator, **run** t, evaluates t to some code `{{ u }}`, and then evaluates u. Continuing the example:

  ```
  {{ 1 + 1 }} ⇓ {{ 1 + 1 }}    deferred till stage 1
  run {{ 1 + 1 }} ⇓ 2          evaluation brought forward to stage 0
  ```

- The *splice* operator, `~t`, also evaluates t to some code `{{ u }}`, but then splices u into the body of the surrounding code. The term `~t` is thus legal only within lexically enclosing `{{ }}` brackets. For example:

  ```
  let code = {{ 1 + 1 }} in {{ ~code + 2 }} ⇓ {{ (1 + 1) + 2 }}
    ~code replaced with 1 + 1 at stage 0
  ```

  (Note that `~` binds tighter than all other operators.)

  A splice expression may appear deep within the body of a deferred expression, even under a λ-abstraction:

  ```
  let code = {{ 1 + 1 }} in {{ \y . ~code + y }} ⇓ {{ \y . 1 + 1 + y }}
  ```

  Also, t may be any expression yielding some code:

  ```
  let f = \code . {{ 1 + ~code }} in {{ ~(f {{ 2 }}) + 3 }} ⇓ {{ (1 + 2) + 3 }}
    f {{ 2 }} evaluated at stage 0
  ```

Splice can be used to construct and manipulate code with free variables, though these variables must always be bound within a lexically enclosing scope. This feature is most convenient when constructing code representing a function:

```
let f = \code . {{ ~code + ~code }} in {{ \x . ~(f {{ x }}) }}
  ⇓ {{ \x . x + x }}
```
x is free in argument to f, but bound in overall result

We say a subterm u of t is *at stage n* if u is lexically nested within n more `{{ }}` brackets than `~` operators within t. For example:

$t + u$	*u is at stage 0*
`{{` $t + u$ `}}`	*u is at stage 1*
`{{` $t +$ `~`u `}}`	*u is at stage 0*

It is even possible for a sub-term to have a negative stage:

`{{` $t +$ `~~`u `}}` *u is at stage -1*

We say a term is *splice free* if each of its sub-terms is at a non-negative stage.

Very roughly, these operators have two rewrite rules. The first allows a splice to cancel a defer, provided t is splice free and the reduct is at stage 1:

$$\sim \{\{\ t\ \}\} \longrightarrow t$$

The second allows run to cancel a defer, again provided t is splice free, and the reduct is at stage 0:

$$\text{run}\ \{\{\ t\ \}\} \longrightarrow t$$

How should these operators be typed? One approach is to perform all type checking at stage 0, and eliminate any programs which may generate ill-typed code at run-time. We call this *statically typed staging*, and is the method used by existing staged languages such as MetaML [97].

7.2 Monomorphically Typed Staged Computation

For the moment, ignore polymorphism (and in particular, constrained polymorphism), and consider how to ensure that only well-typed code may be constructed.

The first source of errors are *binding-time errors*. For example

```
{{ \x . ~x }}
```

attempts to use x at stage 0 when it is not bound until stage 1. This error is easily detected

by maintaining a separate type context for each stage during type checking:

$$\frac{(x : \tau) \in \overline{\Gamma}^n}{\overline{\Gamma} \vdash^n x : \tau} \text{ VARMONO1}$$

$$\frac{\overline{\Gamma} +\!\!+^n x : v \vdash^n t : \tau}{\overline{\Gamma} \vdash^n \backslash x \; . \; t : v \text{ -> } \tau} \text{ ABSMONO} \qquad \frac{\overline{\Gamma} \vdash^n t : v \text{ -> } \tau \quad \overline{\Gamma} \vdash^n u : v}{\overline{\Gamma} \vdash^n t \; u : \tau} \text{ APPMONO}$$

Here we intend n to be the stage of the sub-term under consideration. We write $\overline{\Gamma}$ to denote an infinite length vector of type contexts, indexed by stage number, only a finite number of which are non-empty. (Of course in practice it is easier to associate a stage number with each variable. This vector notation will prove to be convenient in the sequel.) We write $\overline{\Gamma}^n$ to denote the the nth context of $\overline{\Gamma}$, and $\overline{\Gamma} +\!\!+^n \Gamma'$ for the extension of the n'th context of $\overline{\Gamma}$ by Γ'.

A refinement of the VARMONO1 rule is to allow variables bound at an earlier stage to be used at a later stage:

$$\frac{m \geq 0 \quad (x : \tau) \in \overline{\Gamma}^{n-m} \quad \mathit{liftable}(\tau)}{\overline{\Gamma} \vdash^n x : \tau} \text{ VARMONO}$$

Here $\mathit{liftable}(\tau)$ is true when values of type τ can, *at run-time*, be converted from their representation in the run-time system to their representation as code. Defining *liftable* to be the constant true function may be excessively onerous on an implementation. For example, lifting a function could require it's body to be decompiled back into code. Defining $\mathit{liftable}(\tau \text{ -> } v)$ as false prevents this situation.

Using the revised rule, the term

```
\x . {{ x + 1 }}
```

is well-typed assuming *liftable*(`Int`) is true.

Of course a closure is no easier to lift than a function, regardless of it's type. Hence lifting would typically force evaluation. Consider:

```
let x = primes !! 1024
in {{ x + 1 }}
```

This term evaluates to the code `{{ 8161 + 1 }}` rather than `{{ (primes !! 1024) + 1 }}`.

The second source of error arises when code is spliced into an incompatible context. For example

```
let code = {{ True }} in {{ ~code + 1 }}
```

attempts to splice a `Bool` into an `Int` context, leading to the ill-typed code `{{ True + 1 }}`. This too is easily detected by associating a type `{{` τ `}}` of *code of body type* τ with each

defer expression:

$$\frac{\overline{\Gamma} \vdash^{n+1} t : \tau}{\overline{\Gamma} \vdash^{n} \{\{\ t\ \}\} : \{\{\ \tau\ \}\}}\ \textsc{deferMono} \qquad \frac{\overline{\Gamma} \vdash^{n} t : \{\{\ \tau\ \}\}}{\overline{\Gamma} \vdash^{n+1} \sim t : \tau}\ \textsc{spliceMono}$$

$$\frac{\overline{\Gamma} \vdash^{n} t : \{\{\ \tau\ \}\}}{\overline{\Gamma} \vdash^{n} \mathbf{run}\ t : \tau}\ \textsc{runMono1}$$

Notice how these rules keep track of the current stage, and prevent a splice from appearing at stage 0.

One more source of error remains, which is somewhat more subtle than the others. For example

```
{{ \x . ~(run {{ x }}) }}
```

is type-correct by the rules above (assume x has type $\{\{\ \tau\ \}\}$ for some type τ), but evaluates to

```
{{ \x . ~x }}
```

which is binding-time incorrect.

In the literature this problem is known as the *open code problem*, because {{ x }} is "open" on the variable x. A number of refinements to the type rules above have been considered, such as keeping track of the nesting depth of runs [105], or introducing a separate code constructor and code type for *closed code* [106]. Both these approaches introduce considerable additional complexity to the system (and indeed, to the best of our knowledge neither have been implemented).

A third and somewhat surprising solution to the open code problem is to give **run** a type in the IO monad [87]. Hence the RUNMONO1 rule becomes:

$$\frac{\overline{\Gamma} \vdash^{n} t : \{\{\ \tau\ \}\}}{\overline{\Gamma} \vdash^{n} \mathbf{run}\ t : \mathtt{IO}\ \tau}\ \textsc{runMono}$$

Such computations may also be sequenced and completed:

$$\frac{\overline{\Gamma} \vdash^{n} u : \mathtt{IO}\ v \quad \overline{\Gamma} +\!\!+^{n} x : v \vdash^{n} t : \mathtt{IO}\ \tau}{\overline{\Gamma} \vdash^{n} \mathtt{let}\ x \leftarrow u\ \mathtt{in}\ t : \mathtt{IO}\ \tau}\ \textsc{letmMono} \qquad \frac{\overline{\Gamma} \vdash^{n} t : \tau}{\overline{\Gamma} \vdash^{n} \mathbf{unit}\ t : \mathtt{IO}\ \tau}\ \textsc{unitmMono}$$

Under these rules the example above is ill-typed, since **run** {{ x }} has type IO $\{\{\ \tau\ \}\}$ and so cannot be spliced. To see that all such examples will be rejected, we reason informally as follows. The argument to **run** will only be evaluated if **run** is at stage 0 and is being performed. Since only the external environment may perform IO computations, **run** must therefore be connected by a chain of monadic let-bindings to the top level of the program. Because rule SPLICEMONO prevents the splicing of monadic expressions, it is impossible for this chain of let-bindings to cross under a splice. Hence, **run** cannot be in the context of any bound variables, and its argument must be closed.

Note that the typing rule for **run** *does not* guarantee that each occurrence of **run** in a well-typed program is applied only to closed code. For example, in

```
{{ \x . ~(fst ({{ x }}, run {{ x }})) }}
```

`run` *is* applied to code which is patently open. However, the type system prevents `run {{ x }}` from being performed.

Also notice `run`'s `IO` type has nothing to do with any side-effects of `run`, or of the code it executes, but is rather just a "type trick" to prevent open code. In Section 7.5, however, `run` will be enhanced so that it *does* have a side-effect, and hence its `IO` type is better justified.

Encouraged by the ease of typing monomorphic code, we now consider reintroducing parametric polymorphism.

7.3 Polymorphically Typed Staged Computation

Consider let-binding code which is polymorphic:

```
let id = {{ \x . x }}
in {{ (~id 1, ~id True) }}
```

How should this term be typed?

The most straightforward approach, which we term *let-generalisation style*, is to generalise and specialise types exactly as in the polymorphic λ-calculus. Under this approach, `id` would be assigned the type scheme `forall a . {{ a -> a }}`, and the instances of `id` would be specialised to `Int` and `Bool` respectively. To aid our understanding of the situation, consider rewriting the example using type-passing in the style of System F [32]:

$$\begin{array}{l} \mathsf{let\ id} = \Lambda \mathsf{a}\ .\ \{\{\ \lambda \mathsf{x} : \mathsf{a}\ .\ \mathsf{x}\ \}\} \\ \mathsf{in}\ \{\{\ (\tilde{}(\mathsf{id\ Int})\ 1,\ \tilde{}(\mathsf{id\ Bool})\ \mathsf{True})\ \}\} \end{array}$$

This translation clearly shows that all type abstraction and application is performed at stage 0, even though the code itself is at stage 1. Notice that the type parameter has been lifted implicitly from stage 0 to stage 1.

Another possibility, which we call *defer-generalisation style*, is to generalise defer expressions separately from let-bindings, and specialise at each splice point. (Note that let-bound terms are still generalised as per usual under this scheme.) Under this approach, `id` would be assigned the *rank-2 polymorphic type* `{{ forall a . a -> a }}`. If we again rewrite the term to use explicit type passing, the difference between this approach and the previous is obvious:

$$\begin{array}{l} \mathsf{let\ id} = \{\{\ \Lambda \mathsf{a}\ .\ \lambda \mathsf{x} : \mathsf{a}\ .\ \mathsf{x}\ \}\} \\ \mathsf{in}\ \{\{\ (\tilde{}\mathsf{id\ Int}\ 1,\ \tilde{}\mathsf{id\ Bool\ True})\ \}\} \end{array}$$

Notice all type abstraction and application is now at stage 1. In effect, this approach defers type abstraction and application in parallel with evaluation.

Of course, the first approach is to be preferred to the second, since type inference for rank-2 types is very awkward, and for higher ranks is undecidable [114]. Furthermore, for pure parametric polymorphism, type generalisation and specialisation may always be shifted to the stage of the let-binding. Indeed, the example above in defer-generalised form may have

all type abstraction and application moved to stage 0:

$$\text{let id} = \Lambda a' \,.\, \{\{\ (\Lambda a \,.\, \lambda x : a \,.\, x)\ a'\ \}\}$$
$$\text{in } \{\{\ (\text{\textasciitilde}(\text{id Int})\ 1,\ \text{\textasciitilde}(\text{id Bool})\ \text{True})\ \}\}$$

This translation is valid because types may be freely lifted across stages.

For the reasons above, MetaML [97] uses let-generalisation style. Unfortunately, the situation is not so simple when *constrained* parametric polymorphism is introduced.

7.4 Constrained Polymorphism and Staging

In a system of constrained polymorphism, it is possible for let-generalised and defer-generalised terms to have a different semantics. To see why, consider an example using implicit parameters [57]:

```
( let plus1 = {{ 1 + ?z }}
  in {{ ~plus1 with ?z = 1 }} ) with ?z = 0
```

Notice the implicit parameter `?z` is bound both at stage 0 (to `0`) and stage 1 (to `1`). Thus the constraint `?z : Int` will appear both at stage 0 and stage 1. How shall these two occurrences be handled?

In let-generalisation style, let-bound variables capture all the constraints of the let-bound term, regardless of their stage. Thus `plus1` would be assigned the constrained type scheme `?z : Int => {{ Int }}`, and the term would be implemented (using the translation of [57]) as:

$$(\lambda z \,.\, \text{let plus1} = \lambda z' \,.\, \{\{\ 1 + z'\ \}\}$$
$$\text{in } \{\{\ (\lambda z'' \,.\, \text{\textasciitilde}(\text{plus1 } z))\ 1\ \}\}\ )\ 0$$

Note the implicit lift of the parameter z' from stage 0 to stage 1. Hence the instance of `plus1` would be specialised with the binding of `?z = 0` at stage 0, and the program would reduce to (in source form):

```
{{ (1 + 0) with ?z = 1 }}
```

Alternatively, using defer-generalisation style, `plus1` would be assigned the rank-2 type `{{ ?z : Int => Int }}`. The implementation would then be:

$$(\lambda z \,.\, \text{let plus1} = \{\{\lambda z' \,.\, 1 + z'\ \}\}$$
$$\text{in } \{\{(\lambda z'' \,.\, \text{\textasciitilde plus1 } z'')\ 1\ \}\}\ )\ 0$$

Now the code `{{ 1 + ?z }}` would be specialised with the binding of `?z = 1` at stage 1, and the program would reduce to:

```
{{ 1 + ?z with ?z = 1 }}
```

Since the choice of method effects the semantics, one must be prescribed. Unfortunately, neither is pleasing. Let-generalisation style would only work for implicit parameters of *liftable* type, since implicit parameters which cross stages must be lifted. In most implementations, this would rule out defer expressions with implicit parameters of functional type—a severe restriction. Furthermore, the capturing of a stage 1 implicit variable by a stage 0 binding is unlikely to correspond with the programmer's intended interpretation.

Defer-generalisation style, on the other hand, is incompatible with tractable type inference.

One way out of this impasse is to both give up defer-generalisation, and also ignore any constraints from higher stages when let-generalising. The programmer may then use *first-class polymorphism* [49] to *explicitly* generalise polymorphic deferred expressions where desired.

Under this approach, the example could be written as:

```
newtype WithZ = ?z : Int => Int
unWithZ = \(WithZ x) . x

( let plus1 = {{ WithZ (1 + ?z) }}
  in {{ (unWithZ ~plus1) with ?z = 1 }} ) with ?z = 0
```

Here `(1 + ?z)` is generalised when typing the application `WithZ (1 + ?z)`, and `plus1` is assigned the monomorphic type `{{ WithZ }}`. Dually, the implicit parameter `?z` is reexposed by `(unWithZ ~plus1)`, whence it is bound to `1`.

Unfortunately, this approach is not quite sufficient to avoid problems. Consider a variation of the example, this time with two bindings of `?z` at stage 1:

```
{{ ~( let plus1 = {{ 1 + ?z }}
      in {{ ~plus1 with ?z = 1 }} ) with ?z = 2 }}
```

Since the programmer has not explicitly generalised the code bound by `plus1`, the constraint `?z : Int` (at stage 1) escapes, and is bound to 2 by the outer `with`. Hence, this example reduces to:

```
{{ (1 + ?z with ?z = 2) with ?z = 1 }}
```

Again, this result does not corespond to the programmer's intended interpretation of:

```
{{ 1 + ?z with ?z = 1 }}
```

Furthermore, and more seriously, terms such as these would greatly complicate the semantics.

To avoid these problems, λ^{sc} requires that *every statically typed polymorphic deferred expression must be explicitly fully generalised.* Indeed, the type system will require that in `{{` t `}}`, t must be well-typed assuming only `true`, the trivial constraint, at t's stage.

Thus the example above must be written as:

```
newtype WithZ = ?z : Int => Int
unWithZ = \(WithZ x) . x

{{ ~( let plus1 = {{ WithZ (1 + ?z) }}
      in {{ (unWithZ ~plus1) with ?z = 1 }} ) with ?z = 2 }}
```

To formalise this approach, the well-typing judgement must now include a vector of type variable contexts, $\overline{\Delta}$, tracking which type variables are free at which stages. Similarly, it must also include a vector of constraint contexts, $\overline{C}$, tracking the current constraint context for each stage.

It is important to distinguish $\overline{\Delta}$ and $\overline{\Gamma}$, which may contain variables bound at any stage (and hence resemble temporal logic contexts [20]) from $\overline{C}$, which contains constraint contexts only for the current and previous stages (and hence resembles a modal logic context [21]).

In other words, though $\overline{\Delta}$ and $\overline{\Gamma}$ are persistent across stages, $\overline{C}$ is a stack which must be popped when moving to an earlier stage.

The type rules for defer and splice are now:

$$\frac{\overline{\Delta} \mid \overline{C}\,;\mathbf{true} \mid \overline{\Gamma} \vdash^{n+1} t : \tau}{\overline{\Delta} \mid \overline{C} \mid \overline{\Gamma} \vdash^{n} \texttt{\{\{}\ t\ \texttt{\}\}} : \texttt{\{\{}\ \tau\ \texttt{\}\}}}\ \text{DEFERTSIMP}$$

$$\frac{\overline{\Delta} \mid \overline{C} \mid \overline{\Gamma} \vdash^{n} t : \texttt{\{\{}\ \tau\ \texttt{\}\}}}{\overline{\Delta} \mid \overline{C}\,; D \mid \overline{\Gamma} \vdash^{n+1} \texttt{\char`~}t : \tau}\ \text{SPLICETSIMP}$$

To recap: λ^{SC} may generate well-typed code which uses constrained polymorphism, provided that no constraint crosses outside of any defer expression. Furthermore, this obvious lack of expressibility may be circumvented using first-class polymorphism.

Alas, this approach may quickly become excessively burdensome on the programmer.

7.5 Dynamically Typed Staged Computation

The previous section showed how programming with constrained polymorphic code can become tedious because the programmer must explicitly wrap and unwrap polymorphic code fragments. Furthermore, in many programming situations the type of generated code depends on a run-time value and is difficult to express statically.

Both these problems can be avoided if type inference is staged *in parallel with evaluation.* In this way, type inference may be deferred until sufficient type context is known *at run-time.* This approach neatly extends the staging operators we have already introduced, and also subsumes many proposals for *dynamic typing* [1, 56, 2].

A new type, {?}, is introduced for *dynamically typed code.* Values of this type are code fragments for which type inference has been deferred. Indeed, such code fragments may even be ill-typed.

The three statically typed operators of Section 7.1 also have dynamically typed versions. For simplicity, λ^{SC} overloads the splice and run operators to work on both code types, and only introduces a new form for deferring evaluation:

- {? t ?} is like {{ t }}, but defers both the type inference and evaluation of t by one stage:

1 + 1 : Int	*inferred at compile-time*
1 + 1 $\Downarrow$ 2	*evaluated at stage 0*
{? 1 + 1 ?} : {?}	*inference deferred*
{? 1 + 1 ?} $\Downarrow$ {? 1 + 1 ?}	*evaluation deferred*

- As before, **run** t first evaluates t to a piece of code. If the result is a dynamically typed code fragment of the form {? u ?}, it then infers the type of u. Evaluation continues with u if this type is compatible with run's context. For example (writing

$\Downarrow_{IO}$ for evaluation in a monadic context):

```
let i <- run {? 1 + 1 ?} in unit (i + 2)
  ⟹ 1 + 1 : Int                         inference brought forward to stage 0
  ⟹ Int = Int                           types are compatible
  ⇓IO let i = 1 + 1 in unit (i + 2)     evaluation brought forward to stage 0
  ⇓IO 4                                 evaluation continues
```

Two things can go wrong here: The type of u may be incompatible with that of run's context, or u may be ill-typed to begin with. If either of these occur then run discards u and *raises an exception.* For example:

```
let b <- (try run {? 1 + 1 ?}
          catch unit False)
in unit (not b)
  ⟹ 1 + 1 : Int                            inference brought forward to stage 0
  ⟹ Int ≠ Bool                             types not compatible, exception raised
  ⇓IO let b <- unit False in unit (not b)  exception caught
  ⇓IO True                                 evaluation continues
```

Here the operator `(try _ catch _)`, of type `IO a -> IO a -> IO a`, performs its first argument, passing control to its second argument only upon an exception.

- Also as before, `~`t evaluates t to a piece of code. If it is dynamically typed code of the form `{?` u `?}`, u is spliced into the body of the surrounding code, which clearly must also be dynamically typed. Unlike for statically typed splices, the type of a code fragment with dynamically typed splices may now depend on the code being spliced. For example, in:

```
let code = {? \x . (x, x) ?} in {? ~code 1 ?} ⇓ {? (\x . (x, x)) 1 ?}
```

the resulting body has type `(Int, Int)`. However, in

```
let code = {? \x . True ?} in {? ~code 1 ?} ⇓ {? (\x . True) 1 ?}
```

the resulting body now has type `Bool`.

It is quite possible for an expression to be incompatible with the context it is spliced into, yielding *ill-typed code.* For example:

```
let code = {? \x . x + 1 ?} in {? ~code True ?} ⇓ {? (\x . x + 1) True ?}
```

Ill-typed code is detected by run:

```
try run {? (\x . x + 1) True ?}
catch unit False
  ⟹ (\x . x + 1) True : ?     type inference brought forward, ill-typed
  ⇓IO False                   exception caught
```

One choice remains to be made. Should a term `{?` t `?}` be assigned type `{?}` regardless of

t, or should it be rejected if t is ill-typed regardless of code spliced into it? For example:

```
let code = {? 1 ?} in {? ~code + not 1 ?}
```

would be accepted under the former, and rejected under the latter. Since this choice has little effect on the semantics and expressibility of the language, λ^{SC} adopts the later as a small aid to program correctness.

7.6 Constrained Polymorphism and Dynamic Typing

Since dynamically typed code is always assigned the monotype {?}, it may be type checked using the defer-generalisation method sketched in Section 7.3 without any complications. Very roughly, the type rules are:

$$\frac{\overline{\Delta} +\!\!+^{n+1} \Delta' \mid \overline{C}\,; D \mid \overline{\Gamma} \vdash^{n+1} t : \tau}{\overline{\Delta} \mid \overline{C} \mid \overline{\Gamma} \vdash^{n} \texttt{\{?}\ t\ \texttt{?\}} : \texttt{\{?\}}} \text{ DEFERUSIMP}$$

$$\frac{\overline{\Delta} \mid \overline{C} \mid \overline{\Gamma} \vdash^{n} t : \texttt{\{?\}}}{\overline{\Delta} \mid \overline{C}\,; D \mid \overline{\Gamma} \vdash^{n+1} \texttt{\~{}}t : \tau} \text{ SPLICEUSIMP}$$

Notice Δ' and D may be arbitrarily chosen so that t has some type τ. All three properties are then forgotten, and {? t ?} is assigned type {?}. Similarly, in the second rule τ may be chosen arbitrarily so that the context of ~t is well-typed.

Consider the example from Section 7.3, rewritten to use {? ?} brackets:

```
( let plus1 = {? 1 + ?z ?}
  in {? ~plus1 with ?z = 1 ?} ) with ?z = 0
```

Now the type of `1 + ?z` is generalised to give `?z : Int => Int`, and this type is discarded. Hence there is no confusion as to which binding of `?z` applies, and the term reduces to:

```
{? 1 + ?z with ?z = 1 ?}
```

Dynamically typed polymorphic code is thus much easier to program with, but in return cannot be statically verified as type-correct.

Unfortunately, the rules DEFERUSIMP and SPLICEUSIMP fail to differentiate between terms whose type is definitely known, versus those for which the type has been "guessed" by a splice of {?} code. Hence, the actual type system requires two judgement forms at stages 1 and higher.

7.7 The `rttype` and `liftable` Constraints

Recall from Section 7.5 that run must perform a run-time type check of any code of compile-time type {?} to ensure its actual type is compatible with run's context. Furthermore, because run may be used in a polymorphic context, this type may not be known locally.

For example, in:

```
let f = \code . run code
in let b <- f {? True ?};
       i <- f {? True ?}
in unit (not b, i + 1)
```

the first application of `f`, and hence the `run` within `f`, is at type `Bool` (and thus succeeds), while the second is at type `Int` (and thus fails). Somehow a run-time representation of the type of `f`'s context must be conveyed to the occurrence of `run`.

One approach is to use a System F style of type-passing semantics [99]. However, since types are passed into every polymorphic term regardless of whether it actually invokes `run`, this approach is needlessly expensive. Furthermore, it diverges from most existing implementations of functional programming languages which are type-free at run-time.

Instead, λ^{SC} uses the constraint `rttype` τ to indicate that a representation of type τ is required at run-time. This constraint is another example of a "type trick" (analogous to the trick in typing `run` discussed in Section 7.2). Since `rttype` τ is satisfied for *any ground type* τ, it does not really impose a "constraint" on τ at all. Instead, it allows the type system to track which type specialisations require an actual run-time type to be passed as an additional parameter.

Giving `run` (in effect) the constrained polymorphic type

```
run : forall a . rttype a => {?} -> IO a
```

signals that it takes as an additional argument a *witness* of `rttype a`; that is, a representation of whatever monotype `a` is instantiated to. This passing of witnesses parallels the propagation of `rttype` constraints. A type-directed *dictionary translation* rewrites source terms to *run-time terms* in which this witness passing is explicit.

Returning to the example, `f` is assigned the same type scheme as `run`, and the whole term is translated to:

$$
\begin{array}{l}
\mathsf{let\ f} = \lambda\ \mathsf{w}\ .\ \lambda\ \mathsf{code}\ .\ \mathsf{run\ code\ at\ w} \\
\mathsf{in\ let\ b} \leftarrow \mathsf{f\ Bool}\ \langle\mathsf{True}\rangle \\
\qquad\quad \mathsf{i} \leftarrow \mathsf{f\ Int}\ \langle\mathsf{True}\rangle \\
\mathsf{in\ unit\ (not\ b,\ i} + 1)
\end{array}
$$

Notice the witness abstraction in the binding of f, and the witness applications at each occurrence of f.

One more constraint is necessary. Recall from Section 7.2 the side condition *liftable*(τ) in rule VARMONO. Again, in the presence of polymorphism, this condition cannot be checked locally if τ is not ground. Instead the side condition is implemented as a constraint `liftable` τ. Just as for `rttype` τ, this constraint is witnessed by a run-time representation of τ, which may be used at run-time to determine how a value should be lifted. (In λ^{SC}, only `Int` is liftable, so this machinery is somewhat of an overkill.)

Chapter 8

Examples

The system sketched in Chapter 7 is very versatile. This chapter presents examples of dynamic typing, partial evaluation, and distributed computing. The examples are somewhat voluminous, and will assume features beyond those of λ^{SC}—in particular the pattern matching syntax of λ^{TIR}, and the native XML syntax introduced in Section 3.4. However by doing so we demonstrate how staging interacts gracefully with other language features.

These examples have not been formally type checked or tested on a running interpreter. However, key fragments have been tested by transliterating into Haskell.

8.1 Dynamic Typing

Consider replicating C's `printf` procedure in a functional setting. Programmers might like to write:

```
printf "%i = %b" (1, True)
```

where `%i` and `%b` are placeholders for the elements of the argument tuple. Unfortunately, giving `printf` a type such as

```
printf : String -> τ -> IO ()
```

is problematic, as the type τ depends on the value of `printf`'s first argument. This could be expressed using a *dependent type* [9]:

```
printf : Πs : String . (formatType s) -> IO ()
```

where `formatType` converts the format string to a type. However the complexity of dependently-typed programs can quickly become overwhelming.

One solution is to allow `printf` to accept arguments of any type:

```
printf : String -> List Dyn -> IO ()
```

As `printf` parses its format string, it checks each argument is of the appropriate *dynamic type* before outputting its representation.

Examples such as the above are common in:

- Persistent programming, where values of any type may be stored and retrieved from stable storage.
- Distributed programming, where data and code are exchanged between remote programs.

- Interpretive programming, where object language terms of arbitrary type must be represented by meta language constructs of known type.

- Generic programs, such as `printf`, which work non-parametrically over values of arbitrary type.

Existing approaches to dynamic typing [1, 56, 2] introduce a universal datatype of type `Dyn`, and two operations:

- `dynamic` t : τ, which constructs a dynamic value containing both term t and a representation of its type τ;

- `typecase` d `of { x`$_1$: τ_1 `->` t_1 `; ...; x`$_n$: τ_n `->` t_n `}`, which attempts to match the type stored within dynamic value d against one of τ_i, binding the term in d to the appropriate $\mathtt{x}_i$, or failing gracefully if no match is found.

The semantics of these two operators is straightforward when all types involved are monomorphic. However, when `typecase` patterns may contain free type variables, or worse, when `dynamic` values may contain polymorphic terms, the situation becomes much more subtle.

These approaches suffer two main drawbacks:

- Types live in two quite different worlds. *Static types* are generally inferred, and may be implicitly polymorphic with little added complexity for the programmer. *Dynamic types* must be mentioned explicitly within the branches of a `typecase`, and dynamic polymorphism is either forbidden [1], restricted [56], or requires the complex machinery of functors and higher order unification [2].

- Combining dynamic values together to construct a new dynamic value is tedious and verbose to write, since each constituent value requires a separate `typecase`, and the result must be wrapped by `dynamic`.

In λ^{sc}, dynamically typed terms are simply terms for which both evaluation *and type-inference* has been deferred. This approach avoids the problems above:

- The *same* type system as used at compile-time is used at run-time to decide the well-typing of dynamic values. There is no need for explicit type annotations, and dynamic values enjoy type inference just as static values do. As a result, dynamically typed polymorphism is implicit and as convenient to use as statically typed polymorphism.

- The splice operator makes combining dynamically typed values convenient and concise. Even though the type `Dyn` resembles `{?}`, the term `dynamic` t resembles `{?` t `?}`, and `typecase` may be simulated by a chain of `run` commands, dynamic-typing systems have no counterpart to the splice operator.

The implementation of `printf` in λ^{sc} is much the same as in dynamic typing systems: `printf` has type `String -> List {?} -> IO ()`, and the programmer must wrap each argument in `{? ?}` brackets:

```
printf : String -> List {?} -> IO ()
  = letrec format : String -> List {?} -> {?}
      = { \[] [] . {? "" ?};
          \('%' :: 'i' :: cs) (d :: ds) .
            {? itostr ~d ++ ~(format cs ds) ?};
          \('%' :: 'b' :: cs) (d :: ds) .
            {? btostr ~d ++ ~(format cs ds) ?};
          \(c :: cs) ds .
            {? c :: ~(format cs ds) ?};
          \_ _ . {? 0 ?} (* non-string to force error *) }
    in \cs ds . let s <- try run (format cs ds)
                         catch unit "error: bad format"
                in putStr s
```

The helper function, `format`, traverses the format string, splicing together code to construct the result string. The `printf` function attempts to run this code and print the result. An error string is printed if the format string and arguments mismatch in number or type. For example:

```
printf "%i = %b" [{? 1 ?}, {? True ?}]
  ⇓ run({? itostr 1 ++ " = " ++ btostr True ++ "" ?}
  ⇓IO itostr 1 ++ " = " ++ btostr True ++ ""
  ⇓ "1 = True"
```

which is written to output.

Unlike in dynamic typing systems, another implementation is possible which exploits λ^{SC}'s ability to manipulate code containing free variables. This implementation constructs, at run-time, a printing function matching the given format string:

```
makePrintf : forall a . rttype a => String -> IO a
  = letrec makeFun : String -> {?} -> {?}
      = { \[] d . d;
          \('%' :: 'i' :: cs) d .
            {? \x . ~(makeFun cs {? let () <- ~d
                                    in putStr (itostr x) ?}) ?};
          \('%' :: 'b' :: cs) d .
            {? \x . ~(makeFun cs {? let () <- ~d
                                    in putStr (btostr x) ?}) ?};
          \(c :: cs) d . makeFun cs {? let () <- ~d
                                       in putChar c ?} }
    in \cs . run (makeFun cs {? unit () ?})
```

Here, the constraint `rttype` a signals that a run-time representation of type a is required, but does not actually restrict how a may be instantiated.

The helper function, makeFun, traverses the format string, building a λ-abstraction for each argument. Argument d to makeFun accumulates the code to convert and print the arguments seen so far. Notice that x is free in the code passed to the recursive call to makeFun. Without this ability it would be impossible to construct the function at run-time.

Although `makePrintf` may be instantiated to any type, it will raise an exception unless the

type is compatible with the format string. In this respect, `makePrintf` is not parametric polymorphic, but rather ad-hoc polymorphic. Such terms can always be distinguished by their use of the constraint `rttype` τ.

The function `makePrintf` has two advantages over `printf`: It avoids the need to wrap arguments with `{? ?}` brackets, and it allows a printing function to be generated once and reused many times without the overhead of staging.

For example, in:

```
let f <- makePrintf "%i = %b";
    () <- f 1 True
in f 0 False
```

type inference discovers `f` must have type `Int -> Bool -> IO ()`. Hence the application `makePrintf "%i = %b"` returns the function:

```
\x1 . \x2 . let () <- (let () <- (let () <- unit ()
                                  in putStr (itostr x1))
                       in putStr " = ")
            in putStr (btostr x2)
```

8.2 Partial Evaluation

Partial evaluation seeks to specialise code to exploit run-time invariants [50]. For conventional programs, partial evaluation requires a form of binding-time analysis [78]. In λ^{sc}, (and MetaML [97]), partial evaluation is under programmer control through the use of explicit staging anotations. Furthermore, λ^{sc} programs are free to use dynamically typed code whenever it is inconvenient or impossible to express the types of generated programs statically.

Consider implemeting a regular expression compiler which, given a 1-unambiguous regular expression (as introduced in Section 3.4), produces the corresponding Glushkov automaton [17]. Staging can be exploited to encode the automaton directly as a λ^{sc} program, rather than as an interpreter for the automaton's transition function.

The language of regular expressions is represented abstractly:

```
data RegExp = \a .
    Atom a
  | Sum (List (RegExp a))
  | Prod (List (RegExp a))
  | Star RegExp
```

The states of a Glushkov automaton correspond with the positions of atoms in the regular expression it is built from. Hence the first task is to assign a unique position to each atom of the regular expression, and construct a map from positions back to atoms. We shall use natural numbers to represent positions, and assign naturals to atoms from right to left so that the last atom has position `0`. The map is then easily represented as a list indexed by position. For example, the regular expression $a{*}b$ is represented as:

```
Prod [Star (Atom 'a'), Atom 'b']
```

This term is annotated with positions to become:

```
Prod [Star (Atom (1, 'a')), Atom (0, 'b')]
```

The corresponding map is thus:

```
[ 'b', 'a' ]
```

The following function performs this annotation (to avoid complications with overloading the == function, all types in the following program fragements have been specialised to regular expressions over characters, even though most are polymorphic on the atom type):

```
annotate : RegExp Char -> (List Char, RegExp (Int, Char))
  = letrec annList = \cs res . foldr (\re (cs', res') .
                                          let (cs'', re') = ann cs' re
                                          in (cs'', re' :: res'))
                                       (cs, []) res;
           ann = \cs . { \(Atom c)    . (c : cs, Atom (length cs, c));
                         \(Sum res)   . let (cs', res') = annList cs res
                                        in (cs' Sum res');
                         \(Prod res)  . let (cs', res') = annList cs res
                                        in (cs', Prod res');
                         \(Star re)   . let (cs' re') = ann cs re
                                        in (cs', Star re') }
    in \re . let (cs', re') = ann re [] in (reverse cs', re')
```

This and the following functions make use of some standard library functions:

```
length  : forall a . List a -> Int
reverse : forall a . List a -> List a
foldr   : forall a b . (a -> b -> b) -> b -> [a] -> b
map     : forall a b . (a -> b) -> [a] -> [b]
(!!)    : forall a . List a -> Int -> a
and, or : [Bool] -> Bool
```

Some operations on sets of positions, position pairs, and (character, position) pairs are also needed. (In practice these operations would all be instances of more generic operations on sets and relations). Signatures for these operations are given in Figure 8.1. We use 'P" to denote "position", and "C" for "character."

The function hasEmpty is True if its argument regular expression recognises the empty string:

```
hasEmpty : RegExp (Int, Char) -> Bool
  = { \(Atom _) . False;
      \(Sum res) . or (map hasEmpty res);
      \(Prod res) . and (map hasEmpty res);
      \(Star re) . True }
```

The function firstPos is the set of positions of its argument reachable without transition:

```
newtype Set = \a . ...

emptyP      : Set Int
emptyPP     : Set (Int, Int)
singletonP  : Int -> Set Int
unionP      : Set Int -> Set Int -> Set Int
unionPP     : Set (Int, Int) -> Set (Int, Int) -> Set (Int, Int)
unionAllP   : Set (Set Int) -> Set Int
unionAllPP  : Set (Set (Int, Int)) -> Set (Int, Int)
memberP     : Set Int -> Int -> Bool
crossProdP  : Set Int -> Set Int -> Set (Int, Int)
isFunctR    : Set (Int, Int) -> Bool
applyRelPP  : Set (Int, Int) -> Int -> Set Int
mapSetPCP   : (Int -> (Char, Int)) -> Set Int -> Set (Char, Int)
foldSetCP   : forall a . ((Char, Int) -> a -> a) ->
                           a -> Set (Char, Int) -> a
```

Figure 8.1: Signatures for operations on sets and relations

```
firstPos : RegExp (Int, Char) -> Set Int
  = { \(Atom (p, _)) . singletonP p;
      \(Sum res) . unionAllP (map firstPos res);
      \(Prod []) . emptyP;
      \(Prod (re :: res)) .
        unionP (firstPos re)
               (if hasEmpty re then firstPos (Prod res) else emptyP)
      \(Star re) . firstPos re }
```

Similarly, `lastPos` is the set of positions of its argument which are valid stopping states. These are simply the first-positions of the reversed regular-expression:

```
lastPos : RegExp (Int, Char) -> Set Int
  = \re . firstPos (rev re)

rev : forall a . RegExp a -> RegExp a
  = { \(Atom a) . Atom a;
      \(Sum res) . Sum (map rev res);
      \(Prod res) . Prod (reverse (map rev res));
      \(Star re) . Star (rev re) }
```

The function `followPos` yields the set of all pairs of position and successor position. A successor position must be reached by exactly one transition:

```
followPos : RegExp (Int, Char) -> Set (Int, Int)
  = { \(Atom _) . emptyPP;
      \(Sum res) . unionAllPP (map followPos res);
      \(Prod []) . emptyPP;
      \(Prod (re :: res)) .
        unionPP (followPos re)
                (unionPP (followPos (Prod res))
                         (crossProdP (lastPos re)
                                     (firstPos (Prod res))));
      \(Star re) . unionPP (followPos re)
                           (crossProdP (lastPos re)
                                       (firstPos re)) }
```

All the definitons above are now tied together by `makeFollowMaps`, which builds a list of transition relations, one for each position. For simplicity, the starting state is encoded as the "position" one before the leftmost position. Each transition relation maps legal input characters to their following position. The function `makeFollowMaps` also returns the number of positions, and the set of valid final positions for the regular expression:

```
makeFollowMaps : RegExp Char -> (Int, Set Int, List (Set (Char, Int)))
  = \re . let (cs, re') = annoate re;
              nPos = length cs;
              last = unionP (lastPos re')
                            (if hasEmpty re' then
                               singletonP nPos
                             else
                               emptyP);
              follow = unionPP (followPos re')
                               (crossProdP (singletonP nPos) (firstPos re'));
              maps = map (\p -> mapSetPCP (\p' -> (cs !! p', p'))
                                          (applyRelPP follow p))
                         [0..nPos]
              in (nPos, last, maps)
```

This leaves the problem of generating the recogniser itself, which should be code for a function of type `String -> Bool`. Without staging, the only posibility would be to simulate the Glushkov automaton on the given input, requiring two probes per input character: one to map the current position to its transition relation, and another to map the current character to its successor position (or test for final position if the input has been exhausted).

With staging, more efficient solutions are possible. An obvious improvement is to encode the automaton as a single recursive function, and unfold the two probes as a series of `if` expressions. However, this represents the automaton state explicitly as an integer. The following implementation goes one step better by embedding the automaton's state directly in the implicit state of λ^{SC}, thus eliminating all interpretive overhead. This embedding is achieved by generating a set of mutually recursive functions, one for each position (and the starting state), each of which tests the current input and makes a recursive tail-call as required.

The only subtlety is how to generate an arbitrary number of mutually recursive functions. Remember, λ^{SC} does not allow variable names to be generated under programmer control,

and does not allow terms to be built from term-fragments (such as a single `letrec`-binding), only other terms.

The first step is to generate a transition function for each position, which is abstracted over all transition functions (including itself):

```
makeFunN : Int -> Bool -> Set (Char, Int) -> {?}
  = \nPos isLast followMap .
      let makeTests : List {?} -> {?}
            = \fs .
                let testCode = \c, cs . foldSetCP
                  (\(c', p') rest .
                    {? if ~c = c' then ~(fs !! p') ~cs else ~rest ?})
                  {? False ?}
                  followMap
                in {? { \[] . isLast;
                        \(c :: cs) . ~(testCode {? c ?} {? cs ?}) } ?}
      in letrec makeAbs : Int -> List {?} -> {?}
                  = \pos fs .
                      if pos < 0 then
                        makeTests fs
                      else
                        {? \f . ~(makeAbs (pos - 1) ({? f ?} :: fs)) ?}
      in makeAbs nPos []
```

The function `makeAbs` builds a series of function abstractions, one for each position (and the starting state). Notice how though each abstraction argument is statically named "`f`", the run-time system will actually generate fresh argument names for each generated abstraction. These names are accumulated and passed to `makeTests`. This function uses `testCode` to create a nested `if` expression testing the current character against each legal character, and calling the appropriate next-position function. It also tests for valid final positions. Since the type of each transition function depends on the total number of positions, all of this code must be dynamically typed.

The function `makeFuns` generates a nested tuple of transition functions, one for each position (and the starting state). This tuple resembles a list, with `(_ , _)` for `(::)` and `()` for `[]`:

```
makeFuns : Int -> Set Int -> List (Set (Char, Int)) -> {?}
  = \nPos last followMaps .
      letrec genFuns = \p .
        if p < 0 then
          {? () ?}
        else
          {? ( ~(makeFunN nPos (memberP last p) (followMaps !! p))),
               ~(genFuns (p - 1) ) ?}
      in genFuns nPos
```

All that remains is to tie the recursive knot. To do so, we define a family of functions, `fix`n. Given a nested tuple of n n-ary functions, `fix`n returns a nested tuple of n fixed-points. When $n = 1$, the situation is simple:

```
fix1 : forall a . (a -> a, ()) -> (a, ())
  = \(f, ()) . letrec x = f x in (x, ())
```

For $n = 2$, first define two helper functions:

```
app1 : forall a b . (a -> b, ()) -> a -> (b, ())
  = \(f, ()) x . (f x, ())

uncurry1 : forall a b . (a -> b) -> (a, ()) -> b
  = \f (x, ()) . f x
```

Then:

```
fix2 : forall a . ((a -> a -> a), ((a -> a -> a), ())) -> (a, (a, ()))
  = \(f, g) . letrec x = uncurry1 (f x) y;
                     y = fix1 (app1 g x)
              in (x, y)
```

which is equivalent to

```
fix2' = \(f, (g, ())) . letrec x = f x y
                               y = g x y
                        in (x, (y, ()))
```

by Bekič's Lemma.

For $n = 3$, again define two helpers:

```
app2 : forall a b . (a -> b, (a -> b, ())) -> a -> (b, (b, ()))
  = \(f, g) x . (f x, app1 g x)

uncurry2 : forall a b c . (a -> b -> c) -> (a, (b, ())) -> c
  = \f (x, y) . uncurry1 (f x) y
```

And again:

```
fix3 : forall a . ((a -> a -> a -> a),
                   ((a -> a -> a -> a),
                    ((a -> a -> a -> a),
                     ()))) -> (a, (a, (a, ())))
  = \(f, g) . letrec x = uncurry2 (f x) y;
                     y = fix2 (app2 g x)
              in (x, y)
```

The list `fixes` is defined to consist of all such fixed-point combinators (and their helpers), beginning with $n = 1$. Again, the type of each component depends on n, and hence must be dynamically typed:

```
fixes : List ({?}, {?}, {?})
  = letrec next = \(fix, app, uncurry) .
                  let curr =
                    ( {? \(f, g) . letrec x = ~uncurry (f x) y;
                                          y = ~fix (~app g x)
                                   in (x, y) ?},
                      {? \(f, g) x . (f x, ~app g x) ?},
                      {? \f (x, y) . ~uncurry (f x) y ?} )
                  in curr :: next curr
    in let first = ( {? \(f, ()) . letrec x = f x in (x, ()) ?},
                     {? \(f, ()) x . (f x, ()) ?},
                     {? \f (x, ()) . f x ?} )
    in first :: next first
```

The required fixed-point combinator is simply drawn from this list:

```
makeFixN : Int -> {?}
makeFixN = \n . fst (fixes !! (n - 1))
```

Finally, everything is tied together by `makeRecogniser`. This function will return `None` if its argument regular expression is not 1-nonabiguous; that is, at least one transition relation is not functional. Otherwise, it returns `Some` of the recogniser code:

```
makeRecogniser : RegExp Char -> Option {?}
  = \re . let (last, followMaps, nPos) = makeFollowMaps re
          in if and (map isFunctR followMaps) then
            Some {{ fst (~(makeFixN (nPos + 1))
                           ~(makeFuns nPos last followMaps)) }}
          else
            None
```

Notice that even though `makeRecogniser` only builds code of type `String -> Bool` (and hence a `run` of this code could safely elide the type check), this invariant can unfortunately neither be proven by type inference nor indicated by any form of user annotation. Once the programmer steps outside of statically typed code, there is no way to get back in.

The following program constructs a recogniser for the regular expression $a*b$, then repeatedly tests it against input strings:

```
let re = Prod [Star (Atom 'a'), Atom 'b']
in let r <- run (makeRecogniser re)
in { \None . putStrLn "error: r.e. is 1-ambiguous";
     \(Some f) . letrec loop =
                   let s <- getLine;
                       () <- putStrLn (if f s then
                                         "accepted"
                                       else
                                         "rejected")
                   in loop
   } r
```

For this program, `f` would be the term:

```
fst (fix4 (f2, (f1, (f0, ()))))
```

where *fix*4 is as defined by the induction above, and:

```
f2 = \f2 f1 f0 .
        { \[] . False;
          \(c :: cs) . if c == 'a' then f1 cs
                       else if c == 'b' then f0 cs
                       else False }
f1 = \text{same body as f2}
f0 = \f2 f1 f0 .
        { \[] . True;
          \(c :: cs) . False }
```

8.3 Distributed Computing

A *distributed system* involves the co-operation of more than one machine. A contemporary example is the *client-server* model for separating an information provider (*e.g.*, database server or web server) from an information user (*e.g.*, online search program or web browser). Client-server systems are typically implemented as two seperate programs which exchange data in a common format (*e.g.*, SQL or HTML).

This section considers how to implement a distributed system with *programs* as its common exchange format. Staging allows such programs to be generated conveniently, and with static guarantees of well-formedness and (if desired) well-typedness.

These ideas are illustated by implementing a "π-server." Given a request of a natural number n, the server generates a program of type `Html` describing the first n digits of π. Of course, the obvious approach is for the server to calculate π to the required precision itself. However, to demonstrate the flexibility of staging, this calculation will be included within the result program, and hence deferred to the client.

The following will assume some I/O operations to read and write dynamically typed code:

```
readCode : Handle -> IO {?}
writeCode : Handle -> {?} -> IO ()
```

For simplicity, the example also assumes a two-way "pipe," possibly involving a network, has been previously established between the server and client, and the appropriate handles have been supplied to both. Notice this glosses over the problem of ensuring the global envrionment of the sending and receiving programs are compatible. For example, if code contains an application of a newtype `A`, the sender and receiver must agree on `A`'s definition, and similarly for common library functions.

Because `writeCode` has an `IO` type, it's argument is guaranteed to be closed by the same reasoning as used for `run` in Section 7.2. Hence, the code will be ready to be packaged up in form suitable for writing. Furthermore, since `readCode` has the result type of `IO {?}`, the reading system is forced to type-check any code containing imported code before running it. This prevents accidentally or maliciouslsy ill-typed code from entering the system.

Statically typed code may also be coerced to dynamically typed code:

```
forget : forall a . {{ a }} -> {?}
```

The calculation of π exploits an identity established by Bailey, Borwein and Plouffe [6]:

$$\pi = \sum_{i=0}^{\infty} \frac{1}{16^i} \left(\frac{4}{8i+1} - \frac{2}{8i+4} - \frac{1}{8i+5} - \frac{1}{8i+6} \right)$$

This formula may be used to calculate arbitrary base-16 digits of π independently of all preceeding digits. However, it also allows each successive base-16 digit to be calculated on demand by just a few integer operations.

The implementation requires arbitrarily sized `Integer`'s and `Rational`'s. The following assumes the standard binary operators have been overloaded on both types using the techniques of Section 3.5. Furthermore, some additional operators are needed:

```
(%) : Integer -> Integer -> Rational
numerator : Rational -> Integer
denominator : Rational -> Integer
div : Integer -> Integer -> Integer
gcd : Integer -> Integer -> Integer
```

Each term of the sum is given by `term`:

```
term : {{ Integer -> Rational }}
  = {{ \i . let f = i * 8
            in (4 % (f + 1)) - (2 % (f + 4)) -
               (1 % (f + 5)) - (1 % (f + 6)) }}
```

Recall from Section 7.2 that functions are not necessarily *liftable*. Hence this definition, and those following, must be deferred by one stage so that it may be included in the code of the result document.

The base-16 digits are computed as a lazy list. The calculation is careful to expand enough (and only enough) terms ahead of the current digit to guarantee it cannot be changed by a carry-propagation from deeper within the expansion:

```
next_digit : {{ Rational -> Integer }}
  = {{ \r . numerator r 'div' denominator r }}

hex_digits_of_pi : {{ List Integer }}
  = {{ letrec next
           = \i scale remainder .
               let digit = ~next_digit remainder;
                   error = 1 % scale
                   digit' = ~next_digit (remainder + error)
               in if scale > 1 && digit == digit' then
                 digit :: next i (scale 'div' 16)
                                 ((remainder - (digit % 1)) * (16 % 1))
               else
                 next (i + 1) (scale * 16)
                      (remainder + (~term i * (1 % scale)))
        in next 0 1 0 }}
```

The stream of fractional base-16 digits must then be converted to base-10. Again, the calculation looks-ahead just enough base-16 digits to ensure the current base-10 digit cannot

change:

```
dec_digits_of : {{ List Integer -> List Integer }}
  = {{ \hs .
         letrec next = \(h : hs) dec_scale hex_scale remainder .
                         let digit = ~next_digit
                                       (remainder * (dec_scale % 1));
                             error = 16 % hex_scale
                             digit' = ~next_digit ((remainder + error) *
                                                   (dec_scale % 1))
                         in if digit == digit' then
                           digit :: next' (h : hs) (dec_scale * 10)
                                       hex_scale
                                       (remainder - (digit % dec_scale))
                         else
                           next' hs dec_scale (hex_scale * 16)
                                 (remainder + (h % hex_scale));

                next' = \hs dec_scale hex_scale remainder .
                          let factor = gcd dec_scale hex_scale
                          in next hs (dec_scale 'div' factor)
                                     (hex_scale 'div' factor)
                                     (remainder * (factor % 1))
         in next hs 10 16 0 }}
```

Calculating π as a string in base-10 is straightforward:

```
pi : {{ String }}
  = {{ let hex_pi = ~hex_digits_of_pi
       in charOfDigit (head hex_pi) :: '.' ::
            map charOfDigit (~dec_digits_of (tail hex_pi)) }}
```

Here `charOfDigit : Integer -> Char` maps a digit to the character code representing it.

The server can now be presented:

```
server : Handle -> IO ()
  = \h . let errorDoc = <Html><Body>
                         Server Error: ill-typed request
                       </Body></Html>
         in try
           let req <- readCode h;
               n <- run req
           in let title = itostr n ++ " digits of pi";
                  heading = {{ <Head><Title><<title>></Title></Head> }};
                  body = \digits .
                    {{ <Body><H1><<title>></H1><<~digits>></Body> }};
                  html = {{ let pi = ~pi
                            in <Html>
                                 <<~heading>>
                                 <<~(body {{ take (n + 1) pi }})>>
                               </Html> }}
              in writeCode h (forget html)
         catch
           writeCode h (forget errorDoc)
```

Given the appropriate handle, `server` attempts to read a piece of code, and then runs it to check it is an integer. Ill-typed requests are sent an error message as a reply. Otherwise, the code to calculate π is spliced into a `let`-binding in the result program, which is sent as reply.

Notice that all generated code is statically typed throughout this example program, and this type information is forgotten only at the point that code must be written by `writeCode`. Hence, the programmer can be sure only well-typed programs will be constructed at run-time. Also note that the argument to `body` in `server`:

```
{{ take (n + 1) pi }}
```

contains three different ways of using variables within defer expressions:

- `take` is a standard library function, and hence assumed to be available at all stages and in all run-time environments.
- `n` is a stage 0 variable, but since `Int`'s are liftable, may be used at stage 1 without explicit lifting.
- `pi` is a stage 1 `let`-bound variable, which is bound to the code produced by the stage 0 variable of the same name.

To complete the example, consider a client program to request the first 30 digits of π, and displays the result:

```
renderHtml : Html -> IO ()
  = ...

client : Handle -> IO ()
  = \h . let errorDoc = <Html><Body>
                          Client Error: ill-typed reply
                        </Body></Html>
         in let () <- writeCode h (forget {{ 30 }});
                code <- readCode h
            in try
              let html <- run code
              in renderHtml html
            catch
              renderHtml errorDoc
```

Notice how the client fails gracefully with an error message should the server return an ill-typed document.

If all goes to plan, the client will render the HTML page:

```
<Html>
  <Head>
    <Title>30 digits of pi</Title>
  </Head>
  <Body>
    <H1>30 digits of pi</H1>
    3.14159265358979323846264338327
  </Body>
</Html>
```

Chapter 9

Formal Development

The aim of this chapter is to formalise λ^{SC} to the point where we may prove that any program of type τ either diverges or evaluates to a value of type τ. We shall develop a type-checking system, a denotational semantics, and show soundness. We will not, however, show type inference or correctness of the semantics with respect to an unstaged language, both of which are quite subtle problems worthy of future research.

9.1 Syntax

Figure 9.1 presents the syntax of λ^{SC}, most of which should be familiar from examples. The only novelty is the `exists` primitive constraint. The discussion of satisfiability of λ^{TIR} constraints in Section 2.9 is also applicable to λ^{SC}. Partly for historical reasons, and partly for variety, we have chosen within λ^{SC} to ensure the satisfiability of type-scheme constraints by using existential constraints instead of preventing redundant `let`-bindings as was done for λ^{TIR}. Existential constraints play no part at run-time.

We often write `true` for the trivial (empty) constraint $\cdot$, and will assume constraints are equal up to permutation of their primitive constraints. We use κ to range over all kinds, which in λ^{SC} includes only `Type`.

Run-time terms, shown in Figure 9.2, make witness binding (letw B in T), witness abstraction ($\lambda(w_1, \ldots, w_n)$. T) and witness application (T $(W_1, \ldots, W_n)$) explicit. They also associate a witness with `run` (run T at W), and `lift` (lift T using W). In practice, the witnesses themselves are simply representations of monotypes.

Both typed and untyped code is represented in the run-time language using the $\langle t_1 \rangle$ construct, in which t_1 is (almost) a source language term rather than a run-time term. We must use a source term because dynamically typed code cannot be translated to run-time code until run-time, and hence must remain in source language form. However, t_1 is not quite a source language term, as any splice at stage 1 within it must drop back into run-time syntax. This stage dependency is captured by defining the family t_n of terms for each stage $n > 0$. To avoid unnecessary clutter, we shall drop these subscripts wherever possible.

In the following we shall assume all terms are *hygienic* [8]; that is, no bound variable ever shadows another. This restriction applies even across stages, so than `\x . {{\x . 1}}` is *not* hygienic. Of course in a practical application this condition is too restrictive, and type inference and type checking must deal with shadowed variables. The safest approach would be to shadow independently of stage, so that the second `x` shadows the first in the above

Kinds	κ	::=	Type	
Type variables	a, b	::=	`a` | `b` | `c` | ...	
Types	τ, υ	::=	`Int` | τ `->` υ | `{{` τ `}}` | `{?}` | `IO` τ | a	
Prim constraints	c	::=	`rttype` τ | `liftable` τ | `exists` Δ `.` C	
Constraints	C	::=	$c_1, \ldots, c_n$	where $n \geq 0$
Type var contexts	Δ	::=	$a_1 : \kappa_1, \ldots, a_n : \kappa_n$	where $n \geq 0$
Type schemes	σ	::=	`forall` Δ `.` C `=>` τ	
Variables	x, y, z	::=	`x` | `y` | `z` | ...	
Integers	i			
Constants	k	::=	i | `throw` | `(try _ catch _)` | `putint` | `getint`	
Source terms	t, u	::=	x | k | `\`x `.` t | t u	
		|	`let` x `=` u `in` t | `letrec` x `=` u `in` t	
		|	`{{` t `}}` | `{?` t `?}` | `~`t | `lift` t	
		|	`unit` t | `let` x `<-` u `in` t | `run` t	
Type contexts	Γ	::=	$x_1 : \sigma_1, \ldots, x_n : \sigma_n$	where $n \geq 0$

Figure 9.1: Syntax of λ^{sc} source types and terms

term. An implementation would thus have to replace a type context vector with a single map taking a variable name to a pair of a type scheme and a stage number.

In Section 9.5 we shall see that the hygienic invariant can only be maintained by renaming bound variables within code at run-time.

9.2 Well-kinded Types

We write $+\!\!+$ to denote the concatenation of two type variable contexts. This operation is undefined if any variable occurs in both contexts. We write Δ_{init} for the type variable context defining the kinds of any type constants. In λ^{sc}, Δ_{init} may simply be the empty context. We write $\overline{\Delta}$ to denote the ω-vector $\Delta_0 ; \Delta_1 ; \ldots$. All but a finite number of Δ's are Δ_{init}. We write $\overline{\Delta}^n$ to denote Δ_n, and $\overline{\Delta}^{\leq n}$ for $\Delta_0;\ldots;\Delta_n;\Delta_{init};\Delta_{init};\ldots$. We write $\overline{\Delta} +\!\!+^n \Delta'$ for the vector $\Delta_0 ; \ldots ; (\Delta_n +\!\!+ \Delta') ; \Delta_{n+1} ; \ldots$ and $\overline{\Delta} +\!\!+ \overline{\Delta'}$ for $(\Delta_0 +\!\!+ \Delta'_0) ; (\Delta_1 +\!\!+ \Delta'_1) ; \ldots$. By a slight abuse of notation, we write $\overline{\Delta_{init}}$ to denote $\Delta_{init} ; \Delta_{init} ; \ldots$.

Figure 9.3 presents rules for deciding the judgement $\overline{\Delta} \vdash^n \tau : \kappa$, with intended interpretation:

> "Type τ has kind κ at stage n assuming (for every $i \geq 0$) the free stage i type variables are kinded according to $\overline{\Delta}^i$."

Since every type, and every type variable, has kind Type, the real purpose of this judgement is to enforce a form of binding-time correctness on type variables. Assume for the sake of the following examples that λ-bound variables may be type annotated. Then in the term

```
{{ \x : a . ~( (\y : {{ a }} . y) {{ x }} ) }}
```

Witness vars	w	::=	w
Witnesses	W	::=	w | True | Int | $W_1 \to W_2$ | $\{\{\ W\ \}\}$ | $\{?\}$ | IO W
Witness bindings	B	::=	$w_1 = W_1, \ldots, w_n = W_n$ where $n \geq 0$
Constants	K	::=	i | throw | (try _ catch _) | putint | getint
Stage-n terms	t_n, u_n	::=	x | k | `\`x . t_n | $t_n\ u_n$
		|	`let` x = u_n `in` t_n | `letrec` x = u_n `in` t_n
		|	`{{` t_{n+1} `}}` | `{?` t_{n+1} `?}` | `lift` t_n
		|	`~`T if $n = 1$
		|	`~`t_{n-1} if $n > 1$
		|	`unit` t_n | `let` x `<-` u_n `in` t_n | `run` t_n
Runtime Terms	T, U	::=	x | K | $\lambda x\ .\ T$ | $T\ U$
		|	letw B in T | $\lambda(w_1, \ldots, w_n)\ .\ T$ | $T\ (W_1, \ldots, W_n)$
		|	let $x = U$ in T | letrec $x = U$ in T
		|	$\langle t_1 \rangle$ | lift T using W
		|	unit T | let $x \leftarrow U$ in T | run T at W

Figure 9.2: Syntax of λ^{SC} run-time terms

the type `{{ a }}` assigned to y is well-kinded since a is introduced at stage 1. However, in the term

```
{{ \ x : a . ~( fst ( {{ x }}, \y : a . y ) ) }}
```

the type a assigned to y is binding-time incorrect. This term will be rejected by the VAR rule.

Notice that type variables may be implicitly lifted across stages. For example, in

```
{{ \x : a . {{ \y : a . y }} }}
```

the type variable a is introduced at stage 1 and used at stage 2.

Figure 9.3 also extends the well-kinding judgement to type schemes, and constraints. Care must be taken to prevent constraints from containing any type variables from a stage later than the constraint itself: hence the projection $\overline{\Delta}^{\leq n}$ in rules RTTYPE and LIFTABLE. Without this restriction, it is possible for a type variable to *leak* from a later stage to an earlier stage via the constraint simplification system. For example, in

```
{{ \x : a . ~( fst ({{ x }}, run {{ x }})) }}
```

the **run** (at stage 0) would introduce the constraint `rttype {{ a }}` (also at stage 0). Though we shall not present constraint simplification rules for λ^{SC}, any reasonable implementation would simplify this constraint to `rttype a`, which would be ill-kinded at stage 0. Hence the term above should be rejected.

We extend well-kinding of type schemes to type contexts pointwise.

We let θ range over substitutions, which are idempotent maps from type variables to types such that only a finite number of variables are mapped away from themselves. In the following, let $\Delta \vdash \theta$ gsubst (read "θ is a ground substitution for Δ") if $dom(\theta) \subseteq dom(\Delta)$ and $\forall (a : \kappa) \in \Delta\ .\ \overline{\Delta_{init}} \vdash^0 \theta\ a : \kappa$.

$$\boxed{\overline{\Delta} \vdash^n \tau : \kappa}$$

$$\frac{}{\overline{\Delta} \vdash^n \texttt{Int} : \mathsf{Type}}\ \text{INT} \qquad \frac{\overline{\Delta} \vdash^n \tau : \mathsf{Type} \quad \overline{\Delta} \vdash^n \upsilon : \mathsf{Type}}{\overline{\Delta} \vdash^n \tau \texttt{ -> } \upsilon : \mathsf{Type}}\ \text{FUN}$$

$$\frac{\overline{\Delta} \vdash^{n+1} \tau : \mathsf{Type}}{\overline{\Delta} \vdash^n \texttt{\{\{}\ \tau\ \texttt{\}\}} : \mathsf{Type}}\ \text{CODET} \qquad \frac{}{\overline{\Delta} \vdash^n \texttt{\{?\}} : \mathsf{Type}}\ \text{CODEU}$$

$$\frac{\overline{\Delta} \vdash^n \tau : \mathsf{Type}}{\overline{\Delta} \vdash^n \texttt{IO}\ \tau : \mathsf{Type}}\ \text{IO} \qquad \frac{(a : \mathsf{Type}) \in \overline{\Delta}^m \quad m \leq n}{\overline{\Delta} \vdash^n a : \mathsf{Type}}\ \text{VAR}$$

$$\boxed{\overline{\Delta} \vdash^n \sigma\ \mathsf{scheme}}$$

$$\frac{\overline{\Delta} +\!\!+^n \Delta' \vdash^n C\ \mathsf{constraint} \quad \overline{\Delta} +\!\!+^n \Delta' \vdash^n \tau : \mathsf{Type}}{\overline{\Delta} \vdash^n \texttt{forall}\ \Delta'\ \texttt{.}\ C \texttt{ => } \tau\ \mathsf{scheme}}\ \text{SCHEME}$$

$$\boxed{\overline{\Delta} \vdash^n C\ \mathsf{constraint}}$$

$$\frac{\overline{\Delta}^{\leq n} \vdash^n \tau : \mathsf{Type}}{\overline{\Delta} \vdash^n \texttt{rttype}\ \tau\ \mathsf{constraint}}\ \text{RTTYPE} \qquad \frac{\overline{\Delta}^{\leq n} \vdash^n \tau : \mathsf{Type}}{\overline{\Delta} \vdash^n \texttt{liftable}\ \tau\ \mathsf{constraint}}\ \text{LIFTABLE}$$

$$\frac{\overline{\Delta} +\!\!+^n \Delta' \vdash^n C\ \mathsf{constraint}}{\overline{\Delta} \vdash^n \texttt{exists}\ \Delta'\ \texttt{.}\ C\ \mathsf{constraint}}\ \text{EXISTS} \qquad \frac{\forall i\ .\ (\overline{\Delta} \vdash^n c_i\ \mathsf{constraint})}{\overline{\Delta} \vdash^n c_1, \ldots, c_m\ \mathsf{constraint}}\ \text{CONSTRAINTS}$$

Figure 9.3: Well-kinded λ^{SC} types, type schemes and constraints

9.3 Constraint Entailment

The well-typing rules require a notion of constraint *entailment*. For example, `lift` t will be well typed if t has type τ and the current constraint context entails `liftable` τ. Roughly, C entails D when every satisfying substitution for C also satisfies D. However, as explained in Section 7.7, entailment must also construct a *witness* for each primitive constraint in D.

In the following, we will associate witness variables with primitive constraints. Constraints containing such names are termed *constraint contexts* by analogy with ordinary contexts: $w : c$ means "w is bound to a witness of c at run-time" just as $x : \sigma$ means "x is bound to a value of type σ at run-time." To avoid unnecessary syntactic clutter, we shall use C and D to range over both constraints (as defined in Figure 9.1) and constraint contexts. We write *named*(C) for the constraint context formed by associating fresh witness names with each primitive constraint in constraint C. We write *names*(C) for the tuple of witness names in constraint context C. We write *anon*(C) for the constraint formed from constraint context C by erasing all witness names.

Figure 9.4 presents rules for deciding the judgement $C \vdash^e d \hookrightarrow W$, with intended interpretation: "C entails primitive constraint d, with witness W." Notice that W may mention

$$\boxed{C \vdash^e d \hookrightarrow W}$$

$$\frac{d = \texttt{rttype}\ \tau \lor d = \texttt{liftable}\ \tau}{C, w : d \vdash^e d \hookrightarrow w}\ \text{REF}$$

$$\frac{}{C \vdash^e \texttt{liftable Int} \hookrightarrow \mathsf{Int}}\ \text{LIFTINT}$$

$$\frac{}{C \vdash^e \texttt{rttype Int/\{?\}} \hookrightarrow \mathsf{Int}/\{?\}}\ \text{RTTYPEINT/RTTYPECODEU}$$

$$\frac{C \vdash^e \texttt{rttype}\ \tau \hookrightarrow W}{C \vdash^e \texttt{rttype \{\{}\ \tau\ \texttt{\}\}} \hookrightarrow \{\{\ W\ \}\}}\ \text{RTTYPECODET}$$

$$\frac{C \vdash^e \texttt{rttype}\ \upsilon \hookrightarrow W \quad C \vdash^e \texttt{rttype}\ \tau \hookrightarrow W'}{C \vdash^e \texttt{rttype}\ (\upsilon\ \texttt{->}\ \tau) \hookrightarrow (W \to W')}\ \text{RTTYPEFUN}$$

$$\frac{C \vdash^e \texttt{rttype}\ \tau \hookrightarrow W}{C \vdash^e \texttt{rttype (IO}\ \tau\texttt{)} \hookrightarrow \mathsf{IO}\ W}\ \text{RTTYPEIO}$$

$$\frac{}{C \vdash^e \texttt{exists}\ \Delta\ \texttt{. true} \hookrightarrow \mathsf{True}}\ \text{EXISTSTRIV}$$

$$\frac{C \vdash^e \texttt{rttype}\ \mathit{anyground}(\Delta, \tau) \hookrightarrow _ \quad C \vdash^e \texttt{exists}\ \Delta\ \texttt{.}\ D \hookrightarrow \mathsf{True}}{C \vdash^e \texttt{exists}\ \Delta\ \texttt{. (rttype}\ \tau\texttt{,}\ D\texttt{)} \hookrightarrow \mathsf{True}}\ \text{EXISTSRTTYPE}$$

$$\frac{a \in \mathit{dom}(\Delta) \quad C \vdash^e \texttt{exists}\ \Delta\ \texttt{.}\ D \hookrightarrow \mathsf{True}}{C \vdash^e \texttt{exists}\ \Delta\ \texttt{. (liftable}\ a\texttt{,}\ D\texttt{)} \hookrightarrow \mathsf{True}}\ \text{EXISTSLIFTA}$$

$$\frac{\mathit{fv}(d) \cap \mathit{dom}(\Delta) = \emptyset \quad C \vdash^e d \hookrightarrow _ \quad C \vdash^e \texttt{exists}\ \Delta\ \texttt{.}\ D \hookrightarrow \mathsf{True}}{C \vdash^e \texttt{exists}\ \Delta\ \texttt{. (d, D)} \hookrightarrow \mathsf{True}}\ \text{EXISTSLIFT}$$

$$\boxed{C \vdash^e D \hookrightarrow B}$$

$$\frac{\forall i\ .\ (C \vdash^e w_i : d_i \hookrightarrow W_i)}{C \vdash^e \overline{w : d} \hookrightarrow \overline{w = W}}\ \text{CONJ}$$

Figure 9.4: Entailment of λ^{SC} constraints

$$
\begin{array}{rlrl}
[\![w]\!]_\eta &= \eta\ w & [\![\mathsf{True}]\!]_\eta &= \mathsf{ttrue} : * \\
[\![\mathsf{Int}]\!]_\eta &= \mathsf{tint} : * & [\![W \to W']\!]_\eta &= \mathsf{tfun} : ([\![W]\!]_\eta, [\![W']\!]_\eta) \\
[\![\{\{\ W\ \}\}]\!]_\eta &= \mathsf{tcodet} : [\![W]\!]_\eta & [\![\{?\}]\!]_\eta &= \mathsf{tcodeu} : * \\
[\![\mathsf{IO}\ W]\!]_\eta &= \mathsf{tio} : [\![W]\!]_\eta & &
\end{array}
$$

$$
\begin{aligned}
env(B) &= env(B, \cdot) \\
env(\cdot, \eta) &= \eta \\
env((w = W, B), \eta) &= env(B, (\eta, w \mapsto [\![W]\!]_\eta))
\end{aligned}
$$

Figure 9.5: Denotation of λ^{SC} witnesses into $\mathcal{T}$, and the definition of *env*

$$
\begin{aligned}
[\![\texttt{rttype Int}]\!] &= \{\mathsf{tint} : *\} \\
[\![\texttt{rttype (}\upsilon\ \texttt{->}\ \tau\texttt{)}]\!] &= \{\mathsf{tfun} : (t, t') \mid t \in [\![\texttt{rttype}\ \upsilon]\!], t' \in [\![\texttt{rttype}\ \tau]\!]\} \\
[\![\texttt{rttype \{\{}\ \tau\ \texttt{\}\}}]\!] &= \{\mathsf{tcodet} : t \mid t \in [\![\texttt{rttype}\ \tau]\!]\} \\
[\![\texttt{rttype \{?\}}]\!] &= \{\mathsf{tcodeu} : *\} \\
[\![\texttt{rttype (IO}\ \tau\texttt{)}]\!] &= \{\mathsf{tio} : t \mid t \in [\![\texttt{rttype}\ \tau]\!]\} \\
\\
[\![\texttt{liftable Int}]\!] &= \{\mathsf{tint} : *\} \\
[\![\texttt{liftable _}]\!] &= \emptyset \\
\\
[\![\texttt{exists}\ \overline{a : \kappa}\ \texttt{.}\ \overline{c}]\!] &= \{\mathsf{ttrue} : * \mid \overline{\Delta_{init}} \vdash^0 \overline{\upsilon : \kappa}, \textstyle\prod_i [\![c_i[\overline{a \mapsto \upsilon}]]\!] \neq \emptyset\}
\end{aligned}
$$

Figure 9.6: Denotation of λ^{SC} ground primitive constraints as subsets of $\mathcal{T}$

the witness variables of C. This judgement is extended pointwise to general constraint contexts by the CONJ rule.

In rule EXISTSRTTYPE we write $anyground(\Delta, \tau)$ to denote the type $\tau[\overline{a \mapsto \upsilon}]$, where $\Delta = a_1 : \kappa_1, \ldots, a_n : \kappa_n$ and υ_i is a dummy type such that $\overline{\Delta_{init}} \vdash^0 \upsilon_i : \kappa_i$. (Since our only kind is Type, each υ_i may simply be Int). The function *anyground* is a degenerate form of skolemisation.

9.3.1 Soundness of Entailment

Witnesses may be given a trivial denotation in the set $\mathcal{T}$ defined by:

$$\mathcal{T} = (\mathsf{ttrue} : \mathbf{1} + \mathsf{tint} : \mathbf{1} + \mathsf{tfun} : \mathcal{T} \times \mathcal{T} + \mathsf{tcodet} : \mathcal{T} + \mathsf{tcodeu} : \mathbf{1} + \mathsf{tio} : \mathcal{T})$$

Notice there is an injector for each monotype form, in addition to an injector representing the trivial witness True.

The semantics is given by Figure 9.5. We let η range over valuation environments mapping witness names to witnesses in $\mathcal{T}$ (and in the sequel, variable names to values in $\mathbf{E}\ \mathcal{V}$). Figure 9.5 also defines the ancillary function *env* to convert a witness binding B to an environment.

Given $t \in \mathcal{T}$, let $typeOf(t)$ be the unique ground type τ such that $[\![\texttt{rttype}\ \tau]\!] = t$. This function is undefined if t is or contains $\mathsf{ttrue} : *$.

We now wish to check that witnesses built by the entailment relation do indeed "witness"

$$
\begin{aligned}
i &: \texttt{Int} \\
\texttt{throw} &: \texttt{forall a : Type . IO a} \\
\texttt{(try _ catch _)} &: \texttt{forall a : Type . IO a -> IO a -> IO a} \\
\texttt{putint} &: \texttt{Int -> IO Int} \\
\texttt{getint} &: \texttt{IO Int}
\end{aligned}
$$

Figure 9.7: Types for λ^{SC} constants in Γ_{init}

their corresponding constraints. Figure 9.6 defines the meaning of a ground constraint as either the empty set (the constraint is unsatisfiable) or a singleton set containing the sole witness.

We say $\eta \models w : c$ if $\eta\ w \in [\![c]\!]$. This definition is extended pointwise to $\eta \models C$.

Lemma 9.1 (Soundness of Entailment) Let $\Delta\,;\overline{\Delta_{init}} \vdash^0 C$ constraint and $\Delta\,;\overline{\Delta_{init}} \vdash^0 d$ constraint and $\Delta \vdash \theta$ gsubst and $\eta \models \theta\ C$. Then

(i) $C \vdash^e d \hookrightarrow W$ implies $[\![W]\!]_\eta \in [\![\theta\ d]\!]$

(ii) $C \vdash^e \overline{w : d} \hookrightarrow \overline{w = W}$ implies $\forall i\ .\ [\![W_i]\!]_\eta \in [\![\theta\ d_i]\!]$

Proof See Lemma D.1. □

Lemma 9.2 (Transitivity) Let θ be a well-kinded grounding substitution. If $\mathbf{true} \vdash^e \theta\ C \hookrightarrow B$ and $C \vdash^e D \hookrightarrow B'$ then $\mathbf{true} \vdash^e \theta\ D \hookrightarrow B''$ and $env(B'') = env(B', env(B))_{\restriction names(D)}$.

Proof See Lemma D.2. □

Lemma 9.3 (Closure of Entailment) If $\Delta\ ;\overline{\Delta'} \vdash^n C/D$ constraint and $\Delta \vdash \theta$ gsubst and $C \vdash^e D$ then $\theta\ C \vdash^e \theta\ D$

Proof See Lemma D.3. □

Lemma 9.4 Let c be a primitive constraint such that $\Delta_{init};\overline{\Delta'} \vdash^0 c$ constraint and $\mathbf{true} \vdash^e c \hookrightarrow W$.

(i) If $c = w : \texttt{rttype}\ \tau$ then $typeOf([\![W]\!].) = \tau$.

(ii) If $c = w : \texttt{liftable}\ \tau$ then $typeOf([\![W]\!].) = \tau$ and $\tau \in \{\texttt{Int}\}$.

(iii) If $c = \texttt{exists}\ \Delta\ .\ C$ and $\Delta = a_1 : \kappa_1, \ldots, a_n : \kappa_n$ then there exists $\overline{v}$ s.t. $\forall i\ .\ \overline{\Delta_{init}} \vdash^0 v_i : \kappa_i$ and $\mathbf{true} \vdash^e C[\overline{a \mapsto v}]$.

Proof Immediate from Lemma D.1. □

9.4 Well-typed Terms

We write $\overline{\Gamma}$ to denote the ω-vector $\Gamma_0 ; \Gamma_1 ; \ldots$, which enjoys the same conventions as for $\overline{\Delta}$. Γ_{init} contains type schemes for the constants, as defined in Figure 9.7.

We write $\overline{C}$ to denote the n-vector $C_0 ; C_1 ; \ldots ; C_n$, where each C_i is a constraint context. Here n is typically the "current" stage number and hence implied by context.

It is important to notice that $\overline{\Gamma}$ is vector-like, whereas $\overline{C}$ is stack-like. This difference is because free variables persist across stages, whereas constraints must not.

Figure 9.8 presents rules for deciding the judgement $\overline{\Delta} \mid C \mid \overline{\Gamma} \vdash^0 t : \tau \hookrightarrow T$ with intended interpretation:

> "Term t is a stage 0 term of type τ, and is translated to the run-time term T, assuming (for every $i \geq 0$) variables in $\overline{\Gamma}^i$ are bound at stage i to values of their assigned type, and assuming the satisfiability of the constraint C, both of which assume the type variables in $\overline{\Delta}^i$ are substituted at stage i with types of their assigned kind. Furthermore, T assumes the witness names in C to be bound at stage 0 to witnesses."

Two more judgements are required to extend the notion of well-typing to all stages. The rules for these judgements are shown in Figures 9.9 and 9.10.

The judgement $\overline{\Delta} \mid \overline{C} \mid \overline{\Gamma} \vdash^{n+1}_{\mathbf{tt}} t : \tau \hookrightarrow t'_{n+1}$ is true when t is code at stage $n+1$ of type τ. This term is rewritten to the same term, except with any stage 0 sub-terms within it rewritten according to the stage-0 judgement given above.

The judgement $\overline{\Delta} \mid \overline{C} \mid \overline{\Gamma} \vdash^{n+1}_{\mathbf{ff}} t : \tau \hookrightarrow t'_{n+1}$ is similar, except that the type τ assigned to t is "advisory." That is, it is possible for t to evaluate, at stage $n+1$, to code of any type, or even be ill-typed. However, it is also possible that t may be well-typed with type τ. The purpose of this judgement is to attempt to reject at compile-time dynamically typed code which can never yield well-typed code at run-time. As mentioned in Section 7.5, this checking is unnecessary, and is included only as an additional aid to program correctness.

Since these two judgements differ in only 6 places we present most of the rules as a rule schema, using b to range over $\{\mathbf{tt}, \mathbf{ff}\}$.

Rules ABS0, APP0 and LETREC are those of a conventional polymorphic λ-calculus, except with contexts extended to all stages. Similarly, rules UNITM0 and LETM0 type the two monadic constructs.

Rules LET0 and VAR0 respectively introduce and eliminate constrained type schemes. The hypotheses for rule LET0 are somewhat daunting! We explain the situation as follows. The `let`-bound term u may *inherit* the constraints in D_1 from its context C. These constraints must be entailed by C, and must not mention any type variables which u's type will universally quantify. However, u may also require an arbitrary additional constraint D_2, and both D_2 and u's type v may require an arbitrary additional type variable context Δ'. However, for semantic reasons which will become clear in the sequel, we must ensure that D_2 is satisfiable. Hence we also ask that C entails the constraint `exists` Δ' . D_2.

One more subtlety with rule LET0 remains. Some constraints should never be inherited from C. For example, implicit parameters [57] cannot be inherited, otherwise they would become lexically rather than dynamically bound. We let *inhert*(D_1) be true if all the

$$\boxed{\overline{\Delta} \mid \overline{C} \mid \overline{\Gamma} \vdash^0 t : \tau \hookrightarrow T}$$

$$\frac{\begin{array}{c}(x/k : \texttt{forall}\ \overline{a : \kappa}\ .\ D \texttt{ => } \tau) \in \overline{\Gamma}^0 \quad \overline{\Delta} \vdash^0 \overline{v : \kappa} \\ D' = \mathit{named}(D) \quad C \vdash^e D'[\overline{a \mapsto v}] \hookrightarrow B\end{array}}{\overline{\Delta} \mid C \mid \overline{\Gamma} \vdash^0 x/k : \tau[\overline{a \mapsto v}] \hookrightarrow \mathsf{letw}\ B\ \mathsf{in}\ x/k\ \mathit{names}(D')}\ \text{VAR0}$$

$$\frac{\begin{array}{c}\overline{\Delta} \vdash^0 v : \mathsf{Type} \\ \overline{\Delta} \mid C \mid \overline{\Gamma} +\!\!+^0 x : v \vdash^0 t : \tau \hookrightarrow T\end{array}}{\overline{\Delta} \mid C \mid \overline{\Gamma} \vdash^0 \texttt{\textbackslash}x\ .\ t : (v \texttt{ -> } \tau) \hookrightarrow \lambda x\ .\ T}\ \text{ABS0}$$

$$\frac{\begin{array}{c}\overline{\Delta} \mid C \mid \overline{\Gamma} \vdash^0 t : (v \texttt{ -> } \tau) \hookrightarrow T \\ \overline{\Delta} \mid C \mid \overline{\Gamma} \vdash^0 u : v \hookrightarrow U\end{array}}{\overline{\Delta} \mid C \mid \overline{\Gamma} \vdash^0 t\ u : \tau \hookrightarrow T\ U}\ \text{APP0}$$

$$\frac{\begin{array}{c}\overline{\Delta} \vdash^0 D_1\ \mathsf{constraint} \quad \overline{\Delta} +\!\!+^0 \Delta' \vdash^0 D_2\ \mathsf{constraint} \quad \mathit{inherit}(D_1) \\ C \vdash^e D_1 \hookrightarrow B \quad C \vdash^e \texttt{exists}\ \Delta'\ .\ D_2 \hookrightarrow \mathsf{True} \\ \overline{\Delta} +\!\!+^0 \Delta' \mid D_1 +\!\!+ D_2 \mid \overline{\Gamma} \vdash^0 u : v \hookrightarrow U \\ \overline{\Delta} \mid C \mid \overline{\Gamma} +\!\!+^0 x : (\texttt{forall}\ \Delta'\ .\ \mathit{anon}(D_2) \texttt{ => } v) \vdash^0 t : \tau \hookrightarrow T\end{array}}{\overline{\Delta} \mid C \mid \overline{\Gamma} \vdash^0 \texttt{let}\ x \texttt{ = } u\ \texttt{in}\ t : \tau \hookrightarrow \mathsf{let}\ x = (\mathsf{letw}\ B\ \mathsf{in}\ \lambda \mathit{names}(D_2)\ .\ U)\ \mathsf{in}\ T}\ \text{LET0}$$

$$\frac{\begin{array}{c}\overline{\Delta} \vdash^0 v : \mathsf{Type} \quad \overline{\Delta} \mid C \mid \overline{\Gamma} +\!\!+^0 x : v \vdash^0 u : v \hookrightarrow U \\ \overline{\Delta} \mid C \mid \overline{\Gamma} +\!\!+^0 x : v \vdash^0 t : \tau \hookrightarrow T\end{array}}{\overline{\Delta} \mid C \mid \overline{\Gamma} \vdash^0 \texttt{letrec}\ x \texttt{ = } u\ \texttt{in}\ t : \tau \hookrightarrow \mathsf{letrec}\ x = U\ \mathsf{in}\ T}\ \text{LETREC0}$$

$$\frac{\overline{\Delta} \mid C \mid \overline{\Gamma} \vdash^0 t : \tau \hookrightarrow T}{\overline{\Delta} \mid C \mid \overline{\Gamma} \vdash^0 \texttt{unit}\ t : \texttt{IO}\ \tau \hookrightarrow \mathsf{unit}\ T}\ \text{UNITM0}$$

$$\frac{\overline{\Delta} \mid C \mid \overline{\Gamma} \vdash^0 u : \texttt{IO}\ v \hookrightarrow U \quad \overline{\Delta} \mid C \mid \overline{\Gamma} +\!\!+^0 x : v \vdash^0 t : \texttt{IO}\ \tau \hookrightarrow T}{\overline{\Delta} \mid C \mid \overline{\Gamma} \vdash^0 \texttt{let}\ x \texttt{ <- } u\ \texttt{in}\ t : \texttt{IO}\ \tau \hookrightarrow \mathsf{let}\ x \leftarrow U\ \mathsf{in}\ T}\ \text{LETM0}$$

$$\frac{\overline{\Delta} \mid C\,;\texttt{true} \mid \overline{\Gamma} \vdash^1_{\texttt{tt}} t : \tau \hookrightarrow t'}{\overline{\Delta} \mid C \mid \overline{\Gamma} \vdash^0 \texttt{\{\{}\ t\ \texttt{\}\}} : \texttt{\{\{}\ \tau\ \texttt{\}\}} \hookrightarrow \langle t' \rangle}\ \text{DEFERT0}$$

$$\frac{\overline{\Delta} +\!\!+^1 \Delta' \vdash^1 D\ \mathsf{constraint} \quad \overline{\Delta} +\!\!+^1 \Delta' \mid C\,; D \mid \overline{\Gamma} \vdash^1_b t : \tau \hookrightarrow t'}{\overline{\Delta} \mid C \mid \overline{\Gamma} \vdash^0 \texttt{\{?}\ t\ \texttt{?\}} : \texttt{\{?\}} \hookrightarrow \langle t' \rangle}\ \text{DEFERU0}$$

$$\frac{\overline{\Delta} \mid C \mid \overline{\Gamma} \vdash^0 t : \tau \hookrightarrow T \quad \overline{\Delta}^0 \vdash^0 \tau : \mathsf{Type} \quad C \vdash^e \texttt{liftable}\ \tau \hookrightarrow W}{\overline{\Delta} \mid C \mid \overline{\Gamma} \vdash^0 \texttt{lift}\ t : \texttt{\{\{}\ \tau\ \texttt{\}\}} \hookrightarrow \mathsf{lift}\ T\ \mathsf{using}\ W}\ \text{LIFT0}$$

$$\frac{\overline{\Delta} \mid C \mid \overline{\Gamma} \vdash^0 t : \texttt{\{\{}\ \tau\ \texttt{\}\}} \hookrightarrow T \quad \overline{\Delta}^0 \vdash^0 \tau : \mathsf{Type} \quad C \vdash^e \texttt{rttype}\ \tau \hookrightarrow W}{\overline{\Delta} \mid C \mid \overline{\Gamma} \vdash^0 \texttt{run}\ t : \texttt{IO}\ \tau \hookrightarrow \mathsf{run}\ T\ \mathsf{at}\ W}\ \text{RUNT0}$$

$$\frac{\overline{\Delta} \mid C \mid \overline{\Gamma} \vdash^0 t : \texttt{\{?\}} \hookrightarrow T \quad \overline{\Delta}^0 \vdash^0 \tau : \mathsf{Type} \quad C \vdash^e \texttt{rttype}\ \tau \hookrightarrow W}{\overline{\Delta} \mid C \mid \overline{\Gamma} \vdash^0 \texttt{run}\ t : \texttt{IO}\ \tau \hookrightarrow \mathsf{run}\ T\ \mathsf{at}\ W}\ \text{RUNU0}$$

Figure 9.8: Well-typed λ^{SC} stage 0 terms

$$\boxed{\overline{\Delta} \mid \overline{C} \mid \overline{\Gamma} \vdash_b^{n+1} t : \tau \hookrightarrow t'_{n+1}}$$

$$\frac{\overline{\Delta} \mid \overline{C} \,;\, C' \mid \overline{\Gamma} \vdash_{\mathbf{tt}}^{n+1} t : \tau \hookrightarrow t'}{\overline{\Delta} \mid \overline{C} \,;\, C' \mid \overline{\Gamma} \vdash_{\mathbf{ff}}^{n+1} t : \tau \hookrightarrow t'} \text{ FORGET1}$$

$$\frac{\begin{array}{c}(x/k : \texttt{forall}\ \overline{a : \kappa}\ .\ D \texttt{ => } \tau) \in \overline{\Gamma}^{n+1} \quad \overline{\Delta} \vdash^{n+1} \overline{v : \kappa} \\ D' = \mathit{named}(D) \quad C' \vdash^e D'[\overline{a \mapsto v}]\end{array}}{\overline{\Delta} \mid \overline{C} \,;\, C' \mid \overline{\Gamma} \vdash_{\mathbf{tt}}^{n+1} x/k : \tau[\overline{a \mapsto v}] \hookrightarrow x/k} \text{ VAR1}$$

$$\frac{\begin{array}{c}\overline{\Delta} \vdash^{n+1} v : \mathsf{Type} \\ \overline{\Delta} \mid \overline{C} \,;\, C' \mid \overline{\Gamma} +\!\!+^{n+1} x : v \vdash_b^{n+1} t : \tau \hookrightarrow t'\end{array}}{\overline{\Delta} \mid \overline{C} \,;\, C' \mid \overline{\Gamma} \vdash_b^{n+1} \texttt{\\}x\ .\ t : (v \texttt{ -> } \tau) \hookrightarrow \texttt{\\}x\ .\ t'} \text{ ABS1}$$

$$\frac{\begin{array}{c}\overline{\Delta} \mid \overline{C} \,;\, C' \mid \overline{\Gamma} \vdash_b^{n+1} t : (v \texttt{ -> } \tau) \hookrightarrow t' \\ \overline{\Delta} \mid \overline{C} \,;\, C' \mid \overline{\Gamma} \vdash_b^{n+1} u : v \hookrightarrow u'\end{array}}{\overline{\Delta} \mid \overline{C} \,;\, C' \mid \overline{\Gamma} \vdash_b^{n+1} t\ u : \tau \hookrightarrow t'\ u'} \text{ APP1}$$

$$\frac{\begin{array}{c}\overline{\Delta} \vdash^{n+1} D_1\ \mathsf{constraint} \quad \overline{\Delta} +\!\!+^{n+1} \Delta' \vdash^{n+1} D_2\ \mathsf{constraint} \quad \mathit{inherit}(D_1) \\ C' \vdash^e D_1 \quad C' \vdash^e \texttt{exists}\ \Delta'\ .\ D_2 \\ \overline{\Delta} +\!\!+^{n+1} \Delta' \mid \overline{C} \,;\, D_1 +\!\!+ D_2 \mid \overline{\Gamma} \vdash_b^{n+1} u : v \hookrightarrow u' \\ \overline{\Delta} \mid \overline{C} \,;\, C' \mid \overline{\Gamma} +\!\!+^{n+1} x : (\texttt{forall}\ \Delta'\ .\ \mathit{anon}(D_2) \texttt{ => } v) \vdash_b^{n+1} t : \tau \hookrightarrow t'\end{array}}{\overline{\Delta} \mid \overline{C} \,;\, C' \mid \overline{\Gamma} \vdash_b^{n+1} \texttt{let}\ x\ \texttt{=}\ u\ \texttt{in}\ t : \tau \hookrightarrow \texttt{let}\ x\ \texttt{=}\ u'\ \texttt{in}\ t'} \text{ LET1}$$

$$\frac{\begin{array}{c}\overline{\Delta} \vdash^{n+1} v : \mathsf{Type} \quad \overline{\Delta} \mid \overline{C} \,;\, C' \mid \overline{\Gamma} +\!\!+^{n+1} x : v \vdash_b^{n+1} u : v \hookrightarrow u' \\ \overline{\Delta} \mid \overline{C} \,;\, C' \mid \overline{\Gamma} +\!\!+^{n+1} x : v \vdash_b^{n+1} t : \tau \hookrightarrow t'\end{array}}{\overline{\Delta} \mid \overline{C} \,;\, C' \mid \overline{\Gamma} \vdash_b^{n+1} \texttt{letrec}\ x\ \texttt{=}\ u\ \texttt{in}\ t : \tau \hookrightarrow \texttt{letrec}\ x\ \texttt{=}\ u'\ \texttt{in}\ t'} \text{ LETREC1}$$

$$\frac{\overline{\Delta} \mid \overline{C} \,;\, C' \mid \overline{\Gamma} \vdash_b^{n+1} t : \tau \hookrightarrow t'}{\overline{\Delta} \mid \overline{C} \,;\, C' \mid \overline{\Gamma} \vdash_b^{n+1} \texttt{unit}\ t : \texttt{IO}\ \tau \hookrightarrow \texttt{unit}\ t'} \text{ UNITM1}$$

$$\frac{\overline{\Delta} \mid \overline{C} \,;\, C' \mid \overline{\Gamma} \vdash_b^{n+1} u : \texttt{IO}\ v \hookrightarrow u' \quad \overline{\Delta} \mid \overline{C} \,;\, C' \mid \overline{\Gamma} +\!\!+^{n+1} x : v \vdash_b^{n+1} t : \texttt{IO}\ \tau \hookrightarrow t'}{\overline{\Delta} \mid \overline{C} \,;\, C' \mid \overline{\Gamma} \vdash_b^{n+1} \texttt{let}\ x\ \texttt{<-}\ u\ \texttt{in}\ t : \texttt{IO}\ \tau \hookrightarrow \texttt{let}\ x\ \texttt{<-}\ u'\ \texttt{in}\ t'} \text{ LETM1}$$

Figure 9.9: Well-typed λ^{sc} stage $n+1$ terms (part 1 of 2)

$$\frac{\overline{\Delta} \mid \overline{C} \,;\, C' \,;\, \mathtt{true} \mid \overline{\Gamma} \vdash^{n+2}_{\mathtt{tt}} t : \tau \hookrightarrow t'}{\overline{\Delta} \mid \overline{C} \,;\, C' \mid \overline{\Gamma} \vdash^{n+1}_{\mathtt{tt}} \mathtt{\{\{}\ t\ \mathtt{\}\}} : \mathtt{\{\{}\ \tau\ \mathtt{\}\}} \hookrightarrow \mathtt{\{\{}\ t'\ \mathtt{\}\}}}\ \textsc{defert1}$$

$$\frac{\overline{\Delta} +\!\!+^{n+2} \Delta' \vdash^{n+2} D \text{ constraint} \quad \overline{\Delta} +\!\!+^{n+2} \Delta' \mid \overline{C} \,;\, C' \,;\, D \mid \overline{\Gamma} \vdash^{n+2}_{b} t : \tau \hookrightarrow t'}{\overline{\Delta} \mid \overline{C} \,;\, C' \mid \overline{\Gamma} \vdash^{n+1}_{b} \mathtt{\{?}\ t\ \mathtt{?\}} : \mathtt{\{?\}} \hookrightarrow \mathtt{\{?}\ t'\ \mathtt{?\}}}\ \textsc{deferu1}$$

$$\frac{\overline{\Delta} \mid C \mid \overline{\Gamma} \vdash^{0} t : \mathtt{\{\{}\ \tau\ \mathtt{\}\}} \hookrightarrow T}{\overline{\Delta} \mid C \,;\, D \mid \overline{\Gamma} \vdash^{1}_{\mathtt{tt}} \mathtt{\sim} t : \tau \hookrightarrow \mathtt{\sim} T}\ \textsc{splicet1}$$

$$\frac{\overline{\Delta} \mid C \mid \overline{\Gamma} \vdash^{0} t : \mathtt{\{?\}} \hookrightarrow T \quad \overline{\Delta} \vdash^{1} \tau : \mathsf{Type}}{\overline{\Delta} \mid C \,;\, D \mid \overline{\Gamma} \vdash^{1}_{\mathtt{ff}} \mathtt{\sim} t : \tau \hookrightarrow \mathtt{\sim} T}\ \textsc{spliceu1}$$

$$\frac{\overline{\Delta} \mid \overline{C} \,;\, C' \mid \overline{\Gamma} \vdash^{n+1}_{b} t : \mathtt{\{\{}\ \tau\ \mathtt{\}\}} \hookrightarrow t'}{\overline{\Delta} \mid \overline{C} \,;\, C' \,;\, D \mid \overline{\Gamma} \vdash^{n+2}_{b} \mathtt{\sim} t : \tau \hookrightarrow \mathtt{\sim} t'}\ \textsc{splicet2}$$

$$\frac{\overline{\Delta} \mid \overline{C} \,;\, C' \mid \overline{\Gamma} \vdash^{n+1}_{b} t : \mathtt{\{?\}} \hookrightarrow t' \quad \overline{\Delta} \vdash^{n+2} \tau : \mathsf{Type}}{\overline{\Delta} \mid \overline{C} \,;\, C' \,;\, D \mid \overline{\Gamma} \vdash^{n+2}_{\mathtt{ff}} \mathtt{\sim} t : \tau \hookrightarrow \mathtt{\sim} t'}\ \textsc{spliceu2}$$

$$\frac{\overline{\Delta} \mid \overline{C} \,;\, C' \mid \overline{\Gamma} \vdash^{n+1}_{b} t : \tau \hookrightarrow t' \quad \overline{\Delta}^{\leq n+1} \vdash^{n+1} \tau : \mathsf{Type} \quad C' \vdash^{e} \mathtt{liftable}\ \tau}{\overline{\Delta} \mid \overline{C} \,;\, C' \mid \overline{\Gamma} \vdash^{n+1}_{b} \mathtt{lift}\ t : \mathtt{\{\{}\ \tau\ \mathtt{\}\}} \hookrightarrow \mathtt{lift}\ t'}\ \textsc{lift1}$$

$$\frac{\overline{\Delta} \mid \overline{C} \,;\, C' \mid \overline{\Gamma} \vdash^{n+1}_{b} t : \mathtt{\{\{}\ \tau\ \mathtt{\}\}} \hookrightarrow t' \quad \overline{\Delta}^{\leq n+1} \vdash^{n+1} \tau : \mathsf{Type} \quad C' \vdash^{e} \mathtt{rttype}\ \tau}{\overline{\Delta} \mid \overline{C} \,;\, C' \mid \overline{\Gamma} \vdash^{n+1}_{b} \mathtt{run}\ t : \mathtt{IO}\ \tau \hookrightarrow \mathtt{run}\ t'}\ \textsc{runt1}$$

$$\frac{\overline{\Delta} \mid \overline{C} \,;\, C' \mid \overline{\Gamma} \vdash^{n+1}_{b} t : \mathtt{\{?\}} \hookrightarrow t' \quad \overline{\Delta}^{\leq n+1} \vdash^{n+1} \tau : \mathsf{Type} \quad C' \vdash^{e} \mathtt{rttype}\ \tau}{\overline{\Delta} \mid \overline{C} \,;\, C' \mid \overline{\Gamma} \vdash^{n+1}_{b} \mathtt{run}\ t : \mathtt{IO}\ \tau \hookrightarrow \mathtt{run}\ t'}\ \textsc{runu1}$$

Figure 9.10: Well-typed λ^{SC} stage $n+1$ terms (part 2 of 2)

constraints in D_1 may be satisfied by the context of the let-binding rather than the context of each occurrence of the let-bound variable. We assume *inherit*(D_1) implies *inherit*$(\theta\ D_1)$ for any θ. It is because of *inherit* that we require only that C entail D_1, rather than the stronger $C = D_1$. Otherwise, for example, any implicit parameters in C would cause *inherit*(C) to fail, regardless of whether u mentioned these parameters. Of course, in λ^{SC} *inherit*(D) may be true for every D.

The rules DEFERT0 and DEFERU0 are responsible for all of the additional complexity of λ^{SC}. In DEFERT0, an expression `{{ t }}` at stage 0 is well-typed if t is (definitely) well-typed at stage 1 with no residual constraint context. Similarly, in DEFERU0, an expression `{? t ?}` at stage 0 is well-typed if t can be assigned some type under an arbitrary constraint context. Notice there is no requirement that D even be satisfiable.

Rule LIFT0 allows a term to be lifted by one stage if it is of a suitable type. Note that a term may be lifted to an arbitrary stage by nesting splice and lift expressions. The check

that τ be well-kinded using only free type variables from stage 0 prevents the type variable leakage problem mentioned above.

Rules RUNT0 and RUNU0 are identical, save for the type of code being run. Notice the inclusion of the constraint `rttype` τ. As with rule LIFT0, these rules must also check for possible type variable leakage.

The typing rules for terms at stages above zero are for the most part a direct lift of those at stage zero. We shall consider only the exceptions.

Rule FORGET1 allows a definitely well-typed term to be coerced to a possibly well-typed term, and is included only to avoid duplicating rules VAR1, DEFERT1 and SPLICET1. (This rule saves quite some effort later.)

The reader may wonder why the conclusion in rule DEFERU1 uses the $\vdash_b^{n+1}$ judgement rather than $\vdash_{\mathtt{tt}}^{n+1}$, since once code t is wrapped as `{?` t `?}` its type is no longer visible. Unfortunately, such a variation would complicate the proof of soundness, since it is possible for t to evaluate to an untypable piece of code at run-time.

Rule SPLICET1 is the dual to DEFERT0. Notice that the current constraint context is dropped when moving down a stage. This rule must also be replicated over all higher stages, hence SPLICET2.

Rules SPLICEU1 and SPLICEU2 are similar, but allow the type of spliced code to be chosen arbitrarily. In this way, terms such as

```
let f = \code : {?} . {{ ~code + 1 }}
```

may be type checked by assuming `code` will yield an expression of type `Int` at run-time.

9.5 Denotational Semantics

We now turn our attention to the precise semantics of λ^{sc} programs. There are three aspects which make it somewhat complicated.

Firstly, because generated code may contain free variables, care must be taken to avoid *name capture*. For example in:

```
let f = \code . {{ \x . ~code + x }}
in {{ \x . ~(f {{ x }}) }}
```

applying `f` to `{{ x }}` should yield

```
{{ \y . \x . y + x }}
```

and *not*

```
{{ \x . \x . x + x }}
```

Furthermore, there is no way to bound the amount of renaming at compile-time. Consider:

```
letrec f : Int -> List {{ Int }} -> {{ Int -> Int }}
  = \n vs . if n = 0 then
              {{ \x . ~(foldr (\v c . {{ ~v + ~c }}) {{ x }} vs) }}
            else
              {{ \x . ~(f (n - 1) ({{ x }} :: vs)) 1 }}
```

Then `f 2 []` should evaluate to

```
{{ \a . (\b . (\c . b + (a + c)) 1) 1 }}
```

and, in general, `f` n `[]` requires $n + 1$ fresh names.

Thus, any implementation of λ^{SC} must carry around a *fresh name supply* while rebuilding code, and any honest semantics should model this behaviour.

Secondly, in an implementation, eagerly renaming bound variables as they are encountered while generating code would be of quadratic complexity. Instead, renaming should be performed *incrementally* as code is generated by carrying around a *renaming environment.* Notice that since variables are lexically rather than dynamically scoped, incremental renaming requires the construction of "renaming closures," analogous to the value closures already required for partial applications. In order to show the correctness of this optimisation, the semantics should do likewise.

The final source of complexity stems from our desire to apply laziness to all aspects of execution of λ^{SC} programs. For example, since programmers are accustomed to `let x = undefined in 1` evaluating to `1`, they most likely expect `let x = {{ ~undefined }} in 1` and `let x <- unit undefined in unit 1` to do likewise. The former implies code rebuilding must be done lazily, and the second implies monadic commands require a two-level semantics. Modelling lazy rebuilding, whilst also capturing the renaming behaviour above, involves some subtlety.

Moggi [73] has developed a functor-category semantics for two-level languages, which in turn follows the pioneering work of Oles [81] on the semantics of block-structured variables in Algol. This style of semantics is also suitable for λ^{SC}, since we way regard all stages greater than zero to be a single "dynamic stage." However, it suffers two drawbacks. Firstly, because λ^{SC} types and type contexts are indexed by a kind context, a functor-category presentation would require an indexed base category, and hence the calculations could become fairly involved. Secondly, and more importantly, we would like to be able to extend λ^{SC} with the constructs of λ^{TIR} developed in Part I. Since the types of values passed at run-time often depend on indices generated at run-time, λ^{TIR} is most conveniently given an untyped semantics. Thus we would like λ^{SC}'s semantics to be similarly untyped.

Our semantics of terms will be pleasingly close to that of a practical implementation, and will make explicit the name generation and renaming mentioned above. Many aspects of the functor-category semantics reappear within the semantics of types. For example, the semantics will be indexed by a kind and type context vector, and great care will be taken to exclude terms which do not behave uniformly upon renaming. However, we should stress that this connection is, at present, purely informal.

9.5.1 Monads

The denotational semantics will be given in a monadic meta-language [72] over five computational monads [70]. Though each monad is very simple, and thus a direct semantics would also be quite feasible, this approach has three advantages:

- It helps clarify the overall structure of the semantics, and makes, for example, the difference between values, computations, and closures explicit;

$$
\begin{aligned}
\mathbf{E}\ A &= \{\bot\} \cup \{[a] \mid a \in A\} \\
\mathbf{unit_E} &: A \to \mathbf{E}\ A \\
&= \lambda a\ .\ [a] \\
\mathbf{bind_E} &: \mathbf{E}\ A \to (A \to \mathbf{E}\ B) \to \mathbf{E}\ B \\
&= \lambda ea\ f\ .\ \mathbf{case}\ ea\ \mathbf{of}\ \{\bot \to \bot\ ;\ [a] \to f\ a\} \\
\mathbf{strength_E} &: A \times \mathbf{E}\ B \to \mathbf{E}\ (A \times B) \\
&= \lambda a\ eb\ .\ \mathbf{case}\ eb\ \mathbf{of}\ \{\bot \to \bot\ ;\ [b] \to [(a, b)]\} \\
\mathbf{fix_E} &: (\mathbf{E}\ A \to \mathbf{E}\ A) \to \mathbf{E}\ A \\
&= \lambda f\ .\ \mu ea\ .\ f\ ea
\end{aligned}
$$

Figure 9.11: Evaluation monad **E**

- It factors the semantics so that extensions such as imprecise exceptions [86] or mutable references may be added without the need to restructure the semantics as a whole; and
- It may be possible to replace the definitions of these monads with ones which generate code to perform a command, rather than perform the command directly, thus yielding a simple compiler [36].

In the following, we shall work both in **PDom** (pre-domains and continuous functions) and **Dom** (domains and continuous functions).

Figure 9.11 presents the monad **E** of possibly-diverging computations. In this and all subsequent monad definitions we use A and B to, informally, range over all (pre)domains. We assume the usual order-theoretic structure on the result. We write $\mu x\ .\ F[x]$ to denote $\bigsqcup_{i\in\omega}(\lambda x\ .\ F[x])^i\ \bot_D)$, where $(\lambda x\ .\ F[x]) : D \to D$ for some domain D with least-element $\bot_D$. In the definition of $\mathbf{fix_E}$, clearly $\bot_{\mathbf{E}}=\bot$.

Figure 9.12 defines a family of reader monads, which is instantiated in Figures 9.13, 9.14, and 9.15 for reader monads over a renaming environment (**R**), a fresh-name supply (**M**), and both of the above (**N**). We assume $\mathcal{S}$, the set of all variable names, is countably infinite. The empty renaming is denoted by $\emptyset$, and *Names* is all infinite lists of *distinct* variable names. We write $Names_{\setminus\overline{\Gamma}}$ to denote only those lists which do not contain any variable in $\bigcup_i dom(\overline{\Gamma}^i)$.

We write $\mathbf{let}_M\ x \leftarrow u\ \mathbf{in}\ t$ as shorthand for $\mathbf{bind}_M\ u\ (\lambda x\ .\ t)$, and assume $\mathbf{strength}_M$ is used to distribute variables over multiple let-bindings as required (see Moggi [72] for the precise construction.)

Figure 9.16 defines the monad **IO** of integer Input/Output with a single exception using a resumptions-style semantics [90]. The local domain equation for $\mathcal{IO}$ is solved in **Dom**, but **IO** itself is a functor in both **Dom** and **PDom**. Notice we have elided all applications of the *in* and *out* functions mediating the isomorphism between $\mathcal{IO}$ and its one-step unfolding.

The operator $\mathbf{bind_{IO}}$ performs a fold over its first argument looking for the final $(\mathsf{unit} : a)$ to pass to its second argument. Notice the body of the μ binding of l is a function from $\mathbf{IO}\ A \to \mathbf{IO}\ B$, and thus

$$\bot_{\mathbf{IO}\ A\to\mathbf{IO}\ B} = \lambda x\ .\bot_{\mathbf{IO}\ B} = \lambda x\ .\bot_{\mathbf{E}} = \lambda x\ .\bot$$

$$
\begin{aligned}
\mathbf{D}\ E\ A &= E \to \mathbf{E}\ A \\
\mathbf{unit}_{\mathbf{D}\ E} &: A \to \mathbf{D}\ E\ A \\
&= \lambda a\ .\ \lambda e\ .\ \mathbf{unit}_{\mathbf{E}}\ a \\
\mathbf{bind}_{\mathbf{D}\ E} &: \mathbf{D}\ E\ A \to (A \to \mathbf{D}\ E\ B) \to \mathbf{D}\ E\ B \\
&= \lambda ra\ f\ .\ \lambda e\ .\ \mathbf{let}_{\mathbf{E}}\ a \leftarrow ra\ e\ \mathbf{in}\ f\ a\ e \\
\mathbf{strength}_{\mathbf{D}\ E} &: A \times \mathbf{D}\ E\ B \to \mathbf{D}\ E\ (A \times B) \\
&= \lambda a\ rb\ .\ \lambda e\ .\ \mathbf{strength}_{\mathbf{E}}\ (a, rb\ e) \\
\mathbf{closure}^{\mathbf{E}}_{\mathbf{D}\ E} &: \mathbf{D}\ E\ A \to \mathbf{D}\ E\ (\mathbf{E}\ A) \\
&= \lambda ra\ .\ \lambda e\ .\ \mathbf{unit}_{\mathbf{E}}\ (ra\ e) \\
\mathbf{closurefun}^{\mathbf{E}}_{\mathbf{D}\ E} &: (A \to \mathbf{D}\ E\ B) \to \mathbf{D}\ E\ (A \to \mathbf{E}\ B) \\
&= \lambda frb\ .\ \lambda e\ .\ \mathbf{unit}_{\mathbf{E}}\ (\lambda a\ .\ frb\ a\ e) \\
\mathbf{closurefix}^{\mathbf{E}}_{\mathbf{D}\ E} &: (\mathbf{E}\ A \to \mathbf{D}\ E\ A) \to \mathbf{D}\ E\ (\mathbf{E}\ A) \\
&= \lambda fra\ .\ \lambda e\ .\ \mathbf{fix}_{\mathbf{E}}\ (\lambda ea\ .\ fra\ ea\ e) \\
\mathbf{lift}^{\mathbf{D}\ E}_{\mathbf{E}} &: \mathbf{E}\ A \to \mathbf{D}\ E\ A \\
&= \lambda ea\ .\ \lambda e\ .\ ea \\
\mathbf{prod}_{\mathbf{D}\ E} &: \mathbf{D}\ E\ A \to \mathbf{D}\ E\ B \to \mathbf{D}\ E\ (A \times B) \\
&= \lambda ra\ rb\ .\ \lambda e\ .\ \mathbf{let}_{\mathbf{E}}\ a \leftarrow ra\ e \\
&\qquad\qquad\qquad \mathbf{in}\ \mathbf{let}_{\mathbf{E}}\ b \leftarrow rb\ e \\
&\qquad\qquad\qquad \mathbf{in}\ \mathbf{unit}_{\mathbf{E}}\ (a, b)
\end{aligned}
$$

Figure 9.12: Reader monad $\mathbf{D}\ E$

$$
\begin{aligned}
\mathcal{S} &= \text{all variable names} \\
\mathit{RenEnv} &= \mathcal{S} \to^{\mathit{fin}} \mathcal{S} \\
\\
\mathbf{R}\ A &= \mathbf{D}\ \mathit{RenEnv}\ A \\
\mathbf{unit}_{\mathbf{R}} &= \mathbf{unit}_{\mathbf{D}\ \mathit{RenEnv}} \\
\mathbf{bind}_{\mathbf{R}} &= \mathbf{bind}_{\mathbf{D}\ \mathit{RenEnv}} \\
\mathbf{strength}_{\mathbf{R}} &= \mathbf{strength}_{\mathbf{D}\ \mathit{RenEnv}} \\
\mathbf{closure}^{\mathbf{E}}_{\mathbf{R}} &= \mathbf{closure}^{\mathbf{E}}_{\mathbf{D}\ \mathit{RenEnv}} \\
\mathbf{closurefun}^{\mathbf{E}}_{\mathbf{R}} &= \mathbf{closurefun}^{\mathbf{E}}_{\mathbf{D}\ \mathit{RenEnv}} \\
\mathbf{closurefix}^{\mathbf{E}}_{\mathbf{R}} &= \mathbf{closurefix}^{\mathbf{E}}_{\mathbf{D}\ \mathit{RenEnv}} \\
\mathbf{lift}^{\mathbf{R}}_{\mathbf{E}} &= \mathbf{lift}^{\mathbf{D}\ \mathit{RenEnv}}_{\mathbf{E}} \\
\mathbf{get}_{\mathbf{R}} &: \mathcal{S} \to \mathbf{R}\ (\mathsf{name} : \mathcal{S} + \mathsf{undef} : \mathbf{1}) \\
&= \lambda nm\ .\ \lambda env\ .\ \mathbf{unit}_{\mathbf{E}}\ (\mathbf{if}\ nm \in dom(env)\ \mathbf{then}\ (\mathsf{name} : env\ nm) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ (\mathsf{undef} : *)) \\
\mathbf{run}_{\mathbf{R}} &: \mathbf{R}\ A \to \mathbf{E}\ A \\
&= \lambda ra\ .\ ra\ \emptyset
\end{aligned}
$$

Figure 9.13: Renaming monad $\mathbf{R}$

$$Names = List\ \mathcal{S}$$

$$\begin{aligned}
\mathbf{M}\ A &= \mathbf{D}\ Names\ A \\
\mathbf{unit}_{\mathbf{M}} &= \mathbf{unit}_{\mathbf{D}\ Names} \\
\mathbf{bind}_{\mathbf{M}} &= \mathbf{bind}_{\mathbf{D}\ Names} \\
\mathbf{strength}_{\mathbf{M}} &= \mathbf{strength}_{\mathbf{D}\ Names} \\
\mathbf{lift}^{\mathbf{M}}_{\mathbf{E}} &= \mathbf{lift}^{\mathbf{D}\ Names}_{\mathbf{E}}
\end{aligned}$$

Figure 9.14: Name supply monad **M**

$$\begin{aligned}
\mathbf{N}\ A &= \mathbf{D}\ (Names \times RenEnv)\ A \\
\mathbf{unit}_{\mathbf{N}} &= \mathbf{unit}_{\mathbf{D}\ (Names \times RenEnv)} \\
\mathbf{bind}_{\mathbf{N}} &= \mathbf{bind}_{\mathbf{D}\ (Names \times RenEnv)} \\
\mathbf{strength}_{\mathbf{N}} &= \mathbf{strength}_{\mathbf{D}\ (Names \times RenEnv)} \\
\mathbf{prod}_{\mathbf{N}} &= \mathbf{prod}_{\mathbf{D}\ (Names \times RenEnv)} \\
\mathbf{rename}_{\mathbf{N}} &: \mathcal{S} \to \mathbf{N}\ A \to \mathbf{N}\ (\mathcal{S} \times A) \\
&= \lambda nm\ na\ .\ \lambda((nm' : nms), env)\ .\ \mathbf{let}_{\mathbf{E}}\ a \leftarrow na\ (nms, (env[nm \mapsto nm']))\ \mathbf{in}\ \mathbf{unit}_{\mathbf{E}}\ (nm', a) \\
\mathbf{closure}^{\mathbf{M}}_{\mathbf{N}} &: \mathbf{N}\ A \to \mathbf{R}\ (\mathbf{M}\ A) \\
&= \lambda na\ .\ \lambda env\ .\ \mathbf{unit}_{\mathbf{E}}\ (\lambda nms\ .\ na\ (nms, env)) \\
\mathbf{lift}^{\mathbf{N}}_{\mathbf{R}} &: \mathbf{R}\ A \to \mathbf{N}\ A \\
&= \lambda ra\ .\ \lambda(nms, env)\ .\ ra\ env \\
\mathbf{lift}^{\mathbf{N}}_{\mathbf{M}} &: \mathbf{M}\ A \to \mathbf{N}\ A \\
&= \lambda ma\ .\ \lambda(nms, env)\ .\ ma\ nms
\end{aligned}$$

Figure 9.15: Name supply and renaming monad **N**

The operator $\mathbf{trycatch}_{\mathbf{IO}}$ is similar to $\mathbf{bind}_{\mathbf{IO}}$, except that if the first argument yields an $(\mathsf{exception} : *)$, the second argument is spliced into the resumption. In effect, this runs the first command till completion, unless an exception is raised, in which case execution switches to the second command.

We say *ea evaluates to a* (in the **E** monad), written $ea \Downarrow_{\mathbf{E}} a$, if $ea = [a]$. Similarly, we say *ioa evaluates to a* (in the **IO** monad), written $ioa \Downarrow_{\mathbf{IO}} a$, if

$$\begin{aligned}
& ioa \Downarrow_{\mathbf{E}} (\mathsf{unit} : a) \\
\vee\ & \exists z, ioa'\ .\ ioa \Downarrow_{\mathbf{E}} (\mathsf{putint} : (z, ioa')) \wedge ioa' \Downarrow_{\mathbf{IO}} a \\
\vee\ & ioa \Downarrow_{\mathbf{E}} (\mathsf{getint} : f) \wedge \exists z \in \mathcal{Z}\ .\ (f\ z) \Downarrow_{\mathbf{IO}} a
\end{aligned}$$

Notice that

$$\mathbf{unit}_{\mathbf{IO}}\ a \Downarrow_{\mathbf{IO}} a$$

and

$$\mathbf{let}_{\mathbf{IO}}\ x \leftarrow u\ \mathbf{in}\ t \Downarrow_{\mathbf{IO}} b \iff \exists a\ .\ u \Downarrow_{\mathbf{IO}} a \wedge t[x \mapsto a] \Downarrow_{\mathbf{IO}} b$$

$$
\begin{aligned}
&\mathbf{IO}\ A = \mathcal{IO} \\
&\quad \text{where } \mathcal{IO} = \mathbf{E}\ (\mathsf{unit} : A + \mathsf{exception} : \mathbf{1} + \mathsf{putint} : \mathcal{Z} \times \mathcal{IO} + \mathsf{getint} : \mathcal{Z} \to \mathcal{IO}) \\
\mathbf{unit}_{\mathbf{IO}} &: A \to \mathbf{IO}\ A \\
&= \lambda a\ .\ \mathbf{unit}_{\mathbf{E}}\ (\mathsf{unit} : a) \\
\mathbf{bind}_{\mathbf{IO}} &: \mathbf{IO}\ A \to (A \to \mathbf{IO}\ B) \to \mathbf{IO}\ B \\
&= \lambda ioa_1\ f\ .\ (\mu l\ .\ \lambda ioa_2\ .\ \mathbf{let}_{\mathbf{E}}\ v \leftarrow ioa_2 \\
&\qquad \mathbf{in\ case}\ v\ \mathbf{of}\ \{ \\
&\qquad\quad \mathsf{unit} : a \to f\ a; \\
&\qquad\quad \mathsf{exception} : * \to \mathbf{unit}_{\mathbf{E}}\ (\mathsf{exception} : *); \\
&\qquad\quad \mathsf{putint} : (z, ioa_3) \to \mathbf{unit}_{\mathbf{E}}\ (\mathsf{putint} : (z, l\ ioa_3)); \\
&\qquad\quad \mathsf{getint} : g \to \mathbf{unit}_{\mathbf{E}}\ (\mathsf{getint} : \lambda z\ .\ l\ (g\ z)) \\
&\qquad \})\ ioa_1 \\
\mathbf{strength}_{\mathbf{IO}} &: A \times \mathbf{IO}\ B \to \mathbf{IO}\ (A \times B) \\
&= \lambda a\ iob_1\ .\ (\mu l\ .\ \lambda iob_2\ .\ \mathbf{let}_{\mathbf{E}}\ v \leftarrow iob_2 \\
&\qquad \mathbf{in\ case}\ v\ \mathbf{of}\ \{ \\
&\qquad\quad \mathsf{unit} : b \to \mathbf{unit}_{\mathbf{E}}\ (a, b); \\
&\qquad\quad \mathsf{exception} : * \to \mathbf{unit}_{\mathbf{E}}\ (\mathsf{exception} : *); \\
&\qquad\quad \mathsf{putint} : (z, iob_3) \to \mathbf{unit}_{\mathbf{E}}\ (\mathsf{putint} : (z, l\ iob_3)); \\
&\qquad\quad \mathsf{getint} : g \to \mathbf{unit}_{\mathbf{E}}\ (\mathsf{getint} : \lambda z\ .\ l\ (g\ z)) \\
&\qquad \})\ iob_1 \\
\mathbf{putint}_{\mathbf{IO}} &: \mathcal{Z} \to \mathbf{IO}\ \mathbf{1} \\
&= \lambda z\ .\ \mathbf{unit}_{\mathbf{E}}\ (\mathsf{putint} : (z, \mathbf{unit}_{\mathbf{E}}\ (\mathsf{unit} : *))) \\
\mathbf{getint}_{\mathbf{IO}} &: \mathbf{IO}\ \mathcal{Z} \\
&= \mathbf{unit}_{\mathbf{E}}\ (\mathsf{getint} : \lambda z\ .\ \mathbf{unit}_{\mathbf{E}}\ (\mathsf{unit} : z)) \\
\mathbf{throw}_{\mathbf{IO}} &: \mathbf{IO}\ A \\
&= \mathbf{unit}_{\mathbf{E}}\ (\mathsf{exception} : *) \\
\mathbf{trycatch}_{\mathbf{IO}} &: \mathbf{IO}\ A \to \mathbf{IO}\ A \to \mathbf{IO}\ A \\
&= \lambda ioa_1\ ioa_2\ .\ (\mu l\ .\ \lambda ioa_3\ .\ \mathbf{let}_{\mathbf{E}}\ v \leftarrow ioa_3 \\
&\qquad \mathbf{in\ case}\ v\ \mathbf{of}\ \{ \\
&\qquad\quad \mathsf{unit} : a \to \mathbf{unit}_{\mathbf{IO}}\ (\mathsf{unit} : a); \\
&\qquad\quad \mathsf{exception} : * \to ioa_2; \\
&\qquad\quad \mathsf{putint} : (z, ioa_4) \to \mathbf{unit}_{\mathbf{E}}\ (\mathsf{putint} : (z, l\ ioa_4)); \\
&\qquad\quad \mathsf{getint} : g \to \mathbf{unit}_{\mathbf{E}}\ (\mathsf{getint} : \lambda z\ .\ l\ (g\ z)) \\
&\qquad \})\ ioa_1 \\
\mathbf{lift}_{\mathbf{E}}^{\mathbf{IO}} &: \mathbf{E}\ A \to \mathbf{IO}\ A \\
&= \lambda ea\ .\ \mathbf{let}_{\mathbf{E}}\ a \leftarrow ea\ \mathbf{in}\ \mathbf{unit}_{\mathbf{E}}\ (\mathsf{unit} : a)
\end{aligned}
$$

Figure 9.16: I/O monad **IO**

$$
\begin{aligned}
\mathbf{MIO}\ A &= \mathit{Names} \to \mathbf{IO}\ A \\
\mathbf{unit}_{\mathbf{MIO}} &: A \to \mathbf{MIO}\ A \\
&= \lambda a\ .\ \lambda \mathit{nms}\ .\ \mathbf{unit}_{\mathbf{IO}}\ a \\
\mathbf{bind}_{\mathbf{MIO}} &: \mathbf{MIO}\ A \to (A \to \mathbf{MIO}\ B) \to \mathbf{MIO}\ B \\
&= \lambda \mathit{mioa}\ f\ .\ \lambda \mathit{nms}\ .\ \mathbf{bind}_{\mathbf{IO}}\ (\mathit{mioa}\ \mathit{nms})\ (\lambda a\ .\ f\ a\ \mathit{nms}) \\
\mathbf{strength}_{\mathbf{MIO}} &: A \times \mathbf{MIO}\ B \to \mathbf{MIO}\ (A \times B) \\
&= \lambda a\ \mathit{miob}\ .\ \lambda \mathit{nms}\ .\ \mathbf{strength}_{\mathbf{IO}}\ (a, \mathit{miob}\ \mathit{nms}) \\
\mathbf{putint}_{\mathbf{MIO}} &: \mathcal{Z} \to \mathbf{MIO}\ \mathbf{1} \\
&= \lambda z\ .\ \lambda \mathit{nms}\ .\ \mathbf{putint}_{\mathbf{IO}}\ z \\
\mathbf{getint}_{\mathbf{MIO}} &: \mathbf{MIO}\ \mathcal{Z} \\
&= \lambda \mathit{nms}\ .\ \mathbf{getint}_{\mathbf{IO}} \\
\mathbf{throw}_{\mathbf{MIO}} &: \mathbf{MIO}\ A \\
&= \lambda \mathit{nms}\ .\ \mathbf{throw}_{\mathbf{IO}} \\
\mathbf{trycatch}_{\mathbf{MIO}} &: \mathbf{MIO}\ A \to \mathbf{MIO}\ A \to \mathbf{MIO}\ A \\
&= \lambda \mathit{mioa}_1\ \mathit{mioa}_2\ .\ \lambda \mathit{nms}\ .\ \mathbf{trycatch}_{\mathbf{IO}}\ (\mathit{mioa}_1\ \mathit{nms})\ (\mathit{mioa}_2\ \mathit{nms}) \\
\mathbf{lift}_{\mathbf{E}}^{\mathbf{MIO}} &: \mathbf{E}\ A \to \mathbf{MIO}\ A \\
&= \lambda \mathit{ea}\ .\ \lambda \mathit{nms}\ .\ \mathbf{lift}_{\mathbf{E}}^{\mathbf{IO}}\ \mathit{ea} \\
\mathbf{lift}_{\mathbf{M}}^{\mathbf{MIO}} &: \mathbf{M}\ A \to \mathbf{MIO}\ A \\
&= \lambda \mathit{ma}\ .\ \lambda \mathit{nms}\ .\ \mathbf{lift}_{\mathbf{E}}^{\mathbf{IO}}\ (\mathit{ma}\ \mathit{nms})
\end{aligned}
$$

Figure 9.17: Name supply and I/O monad **MIO**

Finally, Figure 9.17 defines the monad **MIO** of Input/Output with a fresh name supply. All of these monads obey the laws:

$$
\begin{array}{rcll}
\mathbf{unit}_M\ t = \mathbf{unit}_M\ u & \Longrightarrow & t = u & \\
\mathbf{let}_M\ x \leftarrow \mathbf{unit}_M\ u\ \mathbf{in}\ t & = & t[x \mapsto u] & \\
\mathbf{let}_M\ x \leftarrow u\ \mathbf{in}\ \mathbf{unit}_M\ x & = & u & \\
\mathbf{let}_M\ y \leftarrow (\mathbf{let}_M\ x \leftarrow u\ \mathbf{in}\ w)\ \mathbf{in}\ t & = & \mathbf{let}_M\ x \leftarrow u\ \mathbf{in}\ y \leftarrow w\ \mathbf{in}\ t & \text{where } x \notin \mathit{fv}(t) \\
\mathbf{lift}_{N'}^{N}\ (\mathbf{lift}_M^{N'}\ t) & = & \mathbf{lift}_M^N\ t & \\
\mathbf{lift}_M^N\ (\mathbf{unit}_M\ t) & = & \mathbf{unit}_N\ t & \\
\mathbf{let}_N\ x \leftarrow \mathbf{lift}_M^N\ u\ \mathbf{in}\ \mathbf{lift}_M^N\ t & = & \mathbf{lift}_M^N\ (\mathbf{let}_M\ x \leftarrow u\ \mathbf{in}\ t) &
\end{array}
$$

Here M and N range over all monad functors, and t, u and w denote meta-terms (and not terms of $\lambda^{\mathrm{SC}}!$). We shall exploit these equalities in the sequel, generally without special mention.

The $\mathbf{strength}_M$ operator also obeys:

$$
\begin{aligned}
\mathbf{strength}_M\ (t, \mathbf{unit}_M\ u) &= \mathbf{unit}_M\ (t, u) \\
\mathbf{let}\ v \leftarrow \mathbf{strength}_M\ (t, u)\ \mathbf{in}\ \mathbf{strength}_M\ v &= \mathbf{strength}_M\ (t, \mathbf{let}_M\ v \leftarrow u\ \mathbf{in}\ v) \\
\mathbf{let}\ (_, x) \leftarrow \mathbf{strength}_M\ (*, t)\ \mathbf{in}\ \mathbf{unit}_M\ x &= t \\
\mathbf{strength}_M\ (t, \mathbf{strength}_M\ (u, w)) &= \mathbf{let}\ ((x, y), z) \leftarrow \mathbf{strength}_M\ ((t, u), w) \\
&\quad\ \mathbf{in}\ \mathbf{unit}_M\ (x, (y, z))
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{Z} &= \text{all integers} \\
\mathcal{D} &= (\ \mathsf{dwrong} : \mathbf{1} + \mathsf{dvar} : \mathcal{S} + \mathsf{dconst} : k + \mathsf{dabs} : \mathcal{S} \times \mathcal{D} + \mathsf{dapp} : \mathcal{D} \times \mathcal{D} \\
&\quad + \mathsf{dlet} : \mathcal{S} \times \mathcal{D} \times \mathcal{D} + \mathsf{dletrec} : \mathcal{S} \times \mathcal{D} \times \mathcal{D} \\
&\quad + \mathsf{ddeft} : \mathcal{D} + \mathsf{ddefu} : \mathcal{D} + \mathsf{dsplice} : \mathcal{D} + \mathsf{dlift} : \mathcal{D} \\
&\quad + \mathsf{dunitm} : \mathcal{D} + \mathsf{dletm} : \mathcal{S} \times \mathcal{D} \times \mathcal{D} + \mathsf{drun} : \mathcal{D}\ ) \\
\mathcal{V} &= (\ \mathsf{wrong} : \mathbf{1} + \mathsf{int} : \mathcal{Z} + \mathsf{func} : \mathbf{E}\ \mathcal{V} \to \mathbf{E}\ \mathcal{V} \\
&\quad + (\textstyle\sum_{n \geq 0} \mathsf{tfunc}_n : (\prod_{1 \leq i \leq n} \mathcal{T}) \to \mathbf{E}\ \mathcal{V}) \\
&\quad + \mathsf{code} : \mathbf{M}\ \mathcal{D} + \mathsf{cmd} : \mathbf{MIO}\ (\mathbf{E}\ \mathcal{V})\ )
\end{aligned}
$$

Figure 9.18: The semantic sets $\mathcal{Z}$ and $\mathcal{D}$, and the predomain $\mathcal{V}$

Since all uses of $\mathbf{strength}_M$ are implicit, all uses of these equalities are similarly left implicit.

9.5.2 Semantic Sets and Predomains

Figure 9.18 defines the set $\mathcal{D}$ and the pre-domain $\mathcal{V}$. Source terms of λ^{sc} generated at run-time will be given a denotation in $\mathcal{D}$. Notice it contains an injector for each source-term construct, and the additional injector dwrong to signal a catastrophic run-time type error during code construction. By "catastrophic," we mean not the failure of type-checking within **run**, which is signalled by an exception in the **MIO** monad, but rather a fundamental type error such as application of an integer.

Given $d \in \mathcal{D}$, we write *termOf*(d) to denote the term t represented by d; it is undefined if d is or contains $\mathsf{dwrong} : *$.

Values, the result of evaluation, will be given a denotation in $\mathcal{V}$. We have presented its semantic equation in a form convenient for the model of types and terms to follow, however since $\mathcal{V}$ is not pointed a little care must be taken to see it has a solution. Consider the domain $\mathbf{E}\ \mathcal{V}$. By pushing $\mathbf{E}$ into the summands, and switching from categorical coproduct, $+$, to coalescing sum $\oplus$, we have $\mathbf{E}\ \mathcal{V} = \mathcal{V}'$, where

$$
\begin{aligned}
\mathcal{V}' = &\ (\ \mathsf{wrong} : \mathbf{E}\ \mathbf{1} \oplus \mathsf{int} : \mathbf{E}\ \mathcal{Z} \oplus \mathsf{func} : \mathbf{E}\ (\mathcal{V}' \to \mathcal{V}') \\
&\oplus (\textstyle\bigoplus_{n \geq 0} \mathsf{tfunc}_n : \mathbf{E}\ ((\prod_{1 \leq i \leq n} \mathcal{T}) \to \mathcal{V}')) \\
&\oplus \mathsf{code} : \mathbf{E}\ (\mathbf{M}\ \mathcal{D}) \oplus \mathsf{cmd} : \mathbf{E}\ (\mathbf{MIO}\ \mathcal{V}')\ )
\end{aligned}
$$

This equation may be solved in **Dom**. Then $\mathcal{V} \cong \downarrow \mathcal{V}'$, where $\downarrow$ removes the least element from a domain. Again, we shall ignore the functions mediating this isomorphism.

Values include the usual integers, functions and $(\mathsf{wrong} : *)$, signalling a catastrophic run-time type error. Notice that functions are call-by-name. The injectand $(\mathsf{tfunc}_n : f)$ is a witness function taking a tuple of n witnesses to a computation of a value. In practice, these witnesses will be run-time representations of monotypes.

The injectand $(\mathsf{code} : md)$ represents a piece of code, which is modelled as a function accepting a fresh name supply and yielding a computation of a run-time representation of a λ^{sc} source term. When code is copied from its point of definition to its final destination, any binders within it will be renamed away from any variables in its new lexical scope by applying the appropriate fresh name supply.

The injectand $(\mathsf{cmd} : mio)$ represents an I/O computation. It accepts a fresh name supply, and yields a computation in the **IO** monad. Notice that the I/O computation yields a computation of a value, rather than a value directly. Otherwise (the denotation of) `unit` t would be strict in t.

9.5.3 Denotation of Types

Figure 9.19 presents the denotation of stage 0 types and type schemes as ideals [59] of $\mathbf{E}\,\mathcal{V}$.

To motivate the definitions, consider how to assign a meaning to the type `{{ Int }}`. Clearly it should contain all functions which, given a fresh name supply, return a computation yielding a run-time generated piece of syntax. Hence, as a first approximation:

$$[\![\{\!\{\ \mathtt{Int}\ \}\!\}]\!] = \mathbf{E}\ \{\mathsf{code} : md \mid md \in \mathbf{M}\ \mathcal{D}, nms \in Names \Longrightarrow md\ nms \in \mathbf{E}\ \mathcal{D}\}$$

Of course, we also wish to ensure (in this case) only *integers* are generated at run-time, suggesting the smaller denotation:

$$[\![\{\!\{\ \mathtt{Int}\ \}\!\}]\!] = \mathbf{E}\ \{\mathsf{code} : md \mid md \in \mathbf{M}\ \mathcal{D}, nms \in Names \Longrightarrow md\ nms \in \mathbf{E}\ \mathcal{D}_{wt}\}$$

where

$$\mathcal{D}_{wt} = \{d \in \mathcal{D} \mid (\overline{\Delta_{init}} \mid \mathtt{true} \mid \overline{\Gamma_{init}} \vdash^0 termOf(d) : \mathtt{Int})\}$$

However, now the denotation is too small, as it forbids the run-time generation of open-code; that is, code containing free variables. Thus we must index the denotation by an appropriate kind and type context for use within the well-typing judgement:

$$[\![\{\!\{\ \mathtt{Int}\ \}\!\}]\!]_{(\overline{\Delta},\overline{\Gamma})} = \mathbf{E}\ \{\mathsf{code} : md \mid md \in \mathbf{M}\ \mathcal{D}, nms \in Names \Longrightarrow md\ nms \in \mathbf{E}\ \mathcal{D}_{wt}\}$$

where

$$\mathcal{D}_{wt} = \{d \in \mathcal{D} \mid (\overline{\Delta} \mid \mathtt{true} \mid \overline{\Gamma} \vdash^0 termOf(d) : \mathtt{Int})\}$$

Now, however, we must be more precise about exactly which lists of "fresh" variable names within *Names* are suitable. To prevent name-capture (which is the whole point of including the machinery for renaming in the first place!), *nms* cannot contain any names within $\overline{\Gamma}$:

$$[\![\{\!\{\ \mathtt{Int}\ \}\!\}]\!]_{(\overline{\Delta},\overline{\Gamma})} = \mathbf{E}\ \{\mathsf{code} : md \mid md \in \mathbf{M}\ \mathcal{D}, nms \in Names_{\setminus\overline{\Gamma}} \Longrightarrow md\ nms \in \mathbf{E}\ \mathcal{D}_{wt}\}$$

where $\mathcal{D}_{wt}$ is as above.

Alas, this denotation is still too large, for it includes members of $\mathbf{M}\ \mathcal{D}$ which simply ignore *nms* and rename bound variables arbitrarily, or not at all. For example, imagine an $md \in \mathbf{M}\ \mathcal{D}$ which produces a $d \in \mathcal{D}_{wt}$ in which a bound variable has been renamed arbitrarily so as to clash with the type context $\overline{\Gamma_e}$ (with new type variables in $\overline{\Delta_e}$). In that case, however, the derivation

$$\overline{\Delta} +\!\!+ \overline{\Delta_e} \mid \mathtt{true} \mid \overline{\Gamma} +\!\!+ \overline{\Gamma_e} \vdash^0 termOf(d) : \mathtt{Int}$$

would *fail*, since shadowed variables are forbidden. This observation suggests misbehaving computations may be rejected if we require their results to be well-typed for *arbitrary*

$$[\![\texttt{Int}]\!]_{(\overline{\Delta},\overline{\Gamma})} = \mathbf{E}\,\{\mathsf{int} : i \mid i \in \mathcal{Z}\}$$

$$[\![\tau \texttt{ -> } \upsilon]\!]_{(\overline{\Delta},\overline{\Gamma})} = \bigcap \left\{ S_{(\overline{\Delta_e},\overline{\Gamma_e})} \mid (\overline{\Delta_e},\overline{\Gamma_e}) \text{ extends } (\overline{\Delta},\overline{\Gamma}) \right\}$$

$$\text{where } S_{(\overline{\Delta_e},\overline{\Gamma_e})} = \mathbf{E} \left\{ \mathsf{func} : f \;\middle|\; \begin{array}{l} f \in \mathbf{E}\,\mathcal{V} \to \mathbf{E}\,\mathcal{V}, \\ ev \in [\![\tau]\!]_{(\overline{\Delta} + \overline{\Delta_e}, \overline{\Gamma} + \overline{\Gamma_e})} \\ \quad \Longrightarrow f\ ev \in [\![\upsilon]\!]_{(\overline{\Delta} + \overline{\Delta_e}, \overline{\Gamma} + \overline{\Gamma_e})} \end{array} \right\}$$

$$[\![\texttt{\{?\}}]\!]_{(\overline{\Delta},\overline{\Gamma})} = \bigcap \left\{ S_{(\overline{\Delta_e},\overline{\Gamma_e})} \mid (\overline{\Delta_e},\overline{\Gamma_e}) \text{ extends } (\overline{\Delta},\overline{\Gamma}) \right\}$$

$$\text{where } S_{(\overline{\Delta_e},\overline{\Gamma_e})} = \mathbf{E} \left\{ \mathsf{code} : md \;\middle|\; \begin{array}{l} md \in \mathbf{M}\,\mathcal{D}, \\ nms \in Names_{\backslash \overline{\Gamma} + \overline{\Gamma_e}} \\ \quad \Longrightarrow md\ nms \in \mathbf{E}\,\mathcal{D}_{wd} \end{array} \right\}$$

$$\text{and } \mathcal{D}_{wd} = \left\{ d \in \mathcal{D} \;\middle|\; \begin{array}{l} termOf(d) \text{ well-defined}, \\ \forall i\ .\ vars(i, termOf(d)) \subseteq dom(\overline{\Gamma}^{i}) \end{array} \right\}$$

$$[\![\texttt{\{\{ } \tau \texttt{ \}\}}]\!]_{(\overline{\Delta},\overline{\Gamma})} = \bigcap \left\{ S_{(\overline{\Delta_e},\overline{\Gamma_e})} \mid (\overline{\Delta_e},\overline{\Gamma_e}) \text{ extends } (\overline{\Delta},\overline{\Gamma}) \right\}$$

$$\text{where } S_{(\overline{\Delta_e},\overline{\Gamma_e})} = \mathbf{E} \left\{ \mathsf{code} : md \;\middle|\; \begin{array}{l} md \in \mathbf{M}\,\mathcal{D}, \\ nms \in Names_{\backslash \overline{\Gamma} + \overline{\Gamma_e}} \\ \quad \Longrightarrow md\ nms \in \mathbf{E}\,\mathcal{D}_{wt(\overline{\Delta_e},\overline{\Gamma_e})} \end{array} \right\}$$

$$\text{and } \mathcal{D}_{wt(\overline{\Delta_e},\overline{\Gamma_e})} = \left\{ d \in \mathcal{D} \;\middle|\; \begin{array}{l} t = termOf(d) \text{ well-defined}, \\ \overline{\Delta} + \overline{\Delta_e} \mid \mathbf{true} \mid \overline{\Gamma} + \overline{\Gamma_e} \vdash^0 t : \tau \end{array} \right\}$$

$$[\![\texttt{IO } \tau]\!]_{(\overline{\Delta},\overline{\Gamma})} = \bigcap \left\{ S_{(\overline{\Delta_e},\overline{\Gamma_e})} \mid (\overline{\Delta_e},\overline{\Gamma_e}) \text{ extends } (\overline{\Delta},\overline{\Gamma}) \right\}$$

$$\text{where } S_{(\overline{\Delta_e},\overline{\Gamma_e})} = \mathbf{E} \left\{ \mathsf{cmd} : io \;\middle|\; \begin{array}{l} io \in \mathbf{MIO}\ (\mathbf{E}\,\mathcal{V}), \\ nms \in Names_{\backslash \overline{\Gamma} + \overline{\Gamma_e}} \wedge (io\ nms) \Downarrow_{\mathbf{IO}} ea \\ \quad \Longrightarrow ea \in [\![\tau]\!]_{(\overline{\Delta},\overline{\Gamma})} \end{array} \right\}$$

$$[\![\texttt{forall } \overline{a : \kappa}\ .\ C \texttt{ => } \tau]\!]_{(\overline{\Delta},\overline{\Gamma})} = \bigcap \left\{ S_{(\overline{\upsilon},B)} \;\middle|\; \begin{array}{l} \overline{\Delta_{init}} \vdash^0 \overline{\upsilon : \kappa}, \\ \mathbf{true} \vdash^e D[\overline{a \mapsto \upsilon}] \hookrightarrow B \end{array} \right\}$$

$$\text{where } D = named(C)$$

$$\text{and } names(D) = (w_1, \ldots, w_n)$$

$$\text{and } S_{(\overline{\upsilon},B)} = \mathbf{E} \left\{ \mathsf{tfunc}_n : f \;\middle|\; \begin{array}{l} f \in (\prod_{1 \le i \le n} \mathcal{T}) \to \mathbf{E}\,\mathcal{V}, \\ f\ ([\![w_1]\!]_{env(B)}, \ldots, [\![w_n]\!]_{env(B)}) \\ \quad \in [\![\tau[\overline{a \mapsto \upsilon}]]\!]_{(\overline{\Delta},\overline{\Gamma})} \end{array} \right\}$$

Figure 9.19: Denotation of λ^{SC} types as ideals of $\mathbf{E}\,\mathcal{V}$

(well-kinded) $\overline{\Gamma_e}$ and $\overline{\Delta_e}$ extending $\overline{\Gamma}$ and $\overline{\Delta}$.

Since ideals are closured under intersection, this condition is easily enforced using the definition as it appears in Figure 9.19. We write $(\overline{\Delta_e}, \overline{\Gamma_e})$ extends $(\overline{\Delta}, \overline{\Gamma})$ to denote that $dom(\overline{\Delta}) \cap dom(\overline{\Delta_e}) = \emptyset$ and $dom(\overline{\Gamma}) \cap dom(\overline{\Gamma_e}) = \emptyset$. Furthermore, we require that Δ_{init} ; $(\overline{\Delta} +\!\!+ \overline{\Delta_e}) \vdash^0 \Gamma_{init} ; \overline{\Gamma_e}$ context, though for readability we shall often leave such well-kinding assumptions implicit. We also implicitly assume Γ_{init} ; $\overline{\Gamma}$ is well-kinded in Δ_{init} ; $\overline{\Delta}$.

The denotations for the remaining types must similarly take into account this uniform renaming behaviour. For `{?}` we obviously cannot require generated terms to be well-typed, but instead only require their free-variables to be contained within $\overline{\Gamma}$. To this end, if $z \in \mathcal{Z}$ we write $vars(z, t)$ for all free variables at stage z in term t. Notice that z may be negative: for example $vars(-1, \texttt{\{\{ \~{}\~{}x \}\}}) = \{\texttt{x}\}$.

A function must behave uniformly regardless of the lexical scope it is applied within, even though that scope will generally be deeper than the scope of its definition. Hence the denotation of function spaces is similarly an intersection over all kind and type context extensions. I/O computations must also be uniform over all extensions.

Finally, the denotation of a type scheme includes only those witness functions which behave correctly for any (ground) types satisfying the scheme's constraint. This use of intersection of ideals is familiar from the semantics of polymorphism given by MacQueen *et al.* [59]. Notice that if C is unsatisfiable, the denotation of `forall` $\overline{a : \kappa}$ `.` C `=>` τ will be all of $\mathbf{E}\,\mathcal{V}$. This fact will be important when we come to show type soundness in the sequel.

Notice that $[\![\sigma]\!]_{(\overline{\Delta}, \overline{\Gamma})}$ is well-defined if $\Delta_{init} ; \overline{\Delta} \vdash^0 \tau$ scheme and $\Delta_{init} ; \overline{\Delta} \vdash^0 \Gamma_{init} ; \overline{\Gamma}$ context. That is, σ must be closed at stage 0, but may contain type variables from $\overline{\Delta}$ at higher stages.

9.5.4 Denotation of Run-Time Terms

The denotation of run-time terms naturally divides into two halves. For higher-stage terms the semantics describes how run-time terms are rebuilt by splicing and renaming. This semantics is defined in Figure 9.20. We let η range over run-time environments mapping both witness variables, w, to witnesses in $\mathcal{T}$, and variable names, x, to computations of values in $\mathbf{E}\,\mathcal{V}$. Then, given a stage-$(n+1)$ term t_{n+1}, we have $[\![t_{n+1}]\!]^{n+1}_{\eta} \in \mathbf{N}\,\mathcal{D}$.

Notice that each occurrence of a variable is renamed as it is encountered by looking up its name in the renaming environment. Dually, each binding occurrence of a variable results in the renaming environment being extended. The fresh name supply is not threaded throughout the computation, but rather inherited according to λ^{SC}'s scope rules. In the splice expression `~` T, T must be evaluated to yield a code value, which is then rebuilt to yield the result.

For stage 0 terms, the semantics is the familiar untyped semantics of the call-by-name λ-calculus, augmented with witness passing, I/O, and the propagation of the renaming environment. Given a run-time term T, we have $[\![T]\!]^0_{\eta} \in \mathbf{R}\,\mathcal{V}$. As usual for denotational semantics, we ignore the sharing of computation which would take place in a call-by-need operational semantics for λ^{SC}.

Notice that, unlike for higher-staged terms, there is no need to propagate a fresh name supply within the semantics of stage 0 terms. Since only `run` rebuilds code, and `run` is

$$
\begin{array}{rl}
[\![x]\!]_\eta^{n+1} = & \mathbf{let_N}\ res \leftarrow \mathbf{lift_R^N}\ (\mathbf{get_R}\ \text{``}x\text{''}) \\
& \mathbf{in\ unit_N}\ (\mathbf{case}\ res\ \mathbf{of}\ \{ \\
& \quad \mathsf{name} : nm \rightarrow \mathsf{dvar} : nm \\
& \quad \mathbf{otherwise} \rightarrow \mathsf{dwrong} : * \\
& \}) \\
[\![k]\!]_\eta^{n+1} = & \mathbf{unit_N}\ (\mathsf{dconst} : k) \\
[\![\texttt{\textbackslash}x\ .\ t]\!]_\eta^{n+1} = & \mathbf{let_N}\ (nm, d) \leftarrow \mathbf{rename_N}\ \text{``}x\text{''}\ [\![t]\!]_\eta^{n+1} \\
& \mathbf{in\ unit_N}\ (\mathsf{dabs} : (nm, d)) \\
[\![t\ u]\!]_\eta^{n+1} = & \mathbf{let_N}\ d \leftarrow [\![t]\!]_\eta^{n+1} \\
& \mathbf{in\ let_N}\ d' \leftarrow [\![u]\!]_\eta^{n+1} \\
& \mathbf{in\ unit_N}\ (\mathsf{dapp} : (d, d')) \\
[\![\texttt{let}\ x\ \texttt{=}\ u\ \texttt{in}\ t]\!]_\eta^{n+1} = & \mathbf{let_N}\ d \leftarrow [\![u]\!]_\eta^{n+1} \\
& \mathbf{in\ let_N}\ (nm, d') \leftarrow \mathbf{rename_N}\ \text{``}x\text{''}\ [\![t]\!]_\eta^{n+1} \\
& \mathbf{in\ unit_N}\ (\mathsf{dlet} : (nm, d, d')) \\
[\![\texttt{letrec}\ x\ \texttt{=}\ u\ \texttt{in}\ t]\!]_\eta^{n+1} = & \mathbf{let_N}\ (nm, (d, d')) \leftarrow \mathbf{rename_N}\ \text{``}x\text{''} \\
& \qquad (\mathbf{prod_N}\ [\![u]\!]_\eta^{n+1}\ [\![t]\!]_\eta^{n+1}) \\
& \mathbf{in\ unit_N}\ (\mathsf{dletrec} : (nm, d, d')) \\
[\![\texttt{\{\{}\ t\ \texttt{\}\}}]\!]_\eta^{n+1} = & \mathbf{let_N}\ d \leftarrow [\![t]\!]_\eta^{n+2} \\
& \mathbf{in\ unit_N}\ (\mathsf{ddeft} : d) \\
[\![\texttt{\{?}\ t\ \texttt{?\}}]\!]_\eta^{n+1} = & \mathbf{let_N}\ d \leftarrow [\![t]\!]_\eta^{n+2} \\
& \mathbf{in\ unit_N}\ (\mathsf{ddefu} : d) \\
[\![\texttt{\textasciitilde}T]\!]_\eta^{1} = & \mathbf{let_N}\ v \leftarrow \mathbf{lift_R^N}\ [\![T]\!]_\eta^{0} \\
& \mathbf{in\ case}\ v\ \mathbf{of}\ \{ \\
& \quad \mathsf{code} : md \rightarrow \mathbf{lift_M^N}\ md; \\
& \quad \mathbf{otherwise} \rightarrow \mathbf{unit_N}\ (\mathsf{dwrong} : *) \\
& \} \\
[\![\texttt{\textasciitilde}t]\!]_\eta^{n+2} = & \mathbf{let_N}\ d \leftarrow [\![t]\!]_\eta^{n+1} \\
& \mathbf{in\ unit_N}\ (\mathsf{dsplice} : d) \\
[\![\texttt{lift}\ t\ \texttt{by}\ n]\!]_\eta^{n+1} = & \mathbf{let_N}\ d \leftarrow [\![t]\!]_\eta^{n+1} \\
& \mathbf{in\ unit_N}\ (\mathsf{dlift} : (d, n)) \\
[\![\texttt{unit}\ t]\!]_\eta^{n+1} = & \mathbf{let_N}\ d \leftarrow [\![t]\!]_\eta^{n+1} \\
& \mathbf{in\ unit_N}\ (\mathsf{dunitm} : d) \\
[\![\texttt{let}\ x\ \texttt{<-}\ u\ \texttt{in}\ t]\!]_\eta^{n+1} = & \mathbf{let_N}\ d \leftarrow [\![u]\!]_\eta^{n+1} \\
& \mathbf{in\ let_N}\ (nm, d') \leftarrow \mathbf{rename_N}\ \text{``}x\text{''}\ [\![t]\!]_\eta^{n+1} \\
& \mathbf{in\ unit_N}\ (\mathsf{dletm} : (nm, d, d')) \\
[\![\texttt{run}\ t]\!]_\eta^{n+1} = & \mathbf{let_N}\ d \leftarrow [\![t]\!]_\eta^{n+1} \\
& \mathbf{in\ unit_N}\ (\mathsf{drun} : d)
\end{array}
$$

Figure 9.20: Denotation of λ^{SC} stage $n+1$ terms

$$
\begin{aligned}
[\![x]\!]^0_\eta &= \mathbf{lift}^{\mathbf{R}}_{\mathbf{E}}\ (\eta\ x) \\
[\![\lambda x\ .\ T]\!]^0_\eta &= \mathbf{let}_{\mathbf{R}}\ f \leftarrow \mathbf{closurefun}^{\mathbf{E}}_{\mathbf{R}}\ (\lambda ev\ .\ [\![T]\!]^0_{\eta, x \mapsto ev}) \\
&\quad\ \mathbf{in}\ \mathbf{unit}_{\mathbf{R}}\ (\mathsf{func} : f) \\
[\![T\ U]\!]^0_\eta &= \mathbf{let}_{\mathbf{R}}\ v \leftarrow [\![T]\!]^0_\eta \\
&\quad\ \mathbf{in}\ \mathbf{let}_{\mathbf{R}}\ ev \leftarrow \mathbf{closure}^{\mathbf{E}}_{\mathbf{R}}\ [\![U]\!]^0_\eta \\
&\quad\ \mathbf{in}\ \mathbf{lift}^{\mathbf{R}}_{\mathbf{E}}\ (\mathbf{case}\ v\ \mathbf{of}\ \{ \\
&\quad\quad \mathsf{func} : f \to f\ ev; \\
&\quad\quad \mathbf{otherwise} \to \mathbf{unit}_{\mathbf{E}}\ (\mathsf{wrong} : *) \\
&\quad\ \}) \\
[\![\mathsf{letw}\ B\ \mathsf{in}\ T]\!]^0_\eta &= [\![T]\!]^0_{env(B,\eta)} \\
[\![\lambda(w_1, \ldots, w_n)\ .\ T]\!]^0_\eta &= \mathbf{let}_{\mathbf{R}}\ f \leftarrow \mathbf{closurefun}^{\mathbf{E}}_{\mathbf{R}}\ (\lambda(y_1, \ldots, y_n)\ .\ [\![T]\!]^0_{\eta, w_1 \mapsto y_1, \ldots, w_n \mapsto y_1}) \\
&\quad\ \mathbf{in}\ \mathbf{unit}_{\mathbf{R}}\ (\mathsf{tfunc}_n : f) \\
[\![T\ (W_1, \ldots, W_n)]\!]^0_\eta &= \mathbf{let}_{\mathbf{R}}\ v \leftarrow [\![T]\!]^0_\eta \\
&\quad\ \mathbf{in}\ \mathbf{lift}^{\mathbf{R}}_{\mathbf{E}}\ (\mathbf{case}\ v\ \mathbf{of}\ \{ \\
&\quad\quad \mathsf{tfunc}_n : f \to f\ ([\![W_1]\!]_\eta, \ldots, [\![W_n]\!]_\eta); \\
&\quad\quad \mathbf{otherwise} \to \mathbf{unit}_{\mathbf{E}}\ (\mathsf{wrong} : *) \\
&\quad\ \}) \\
[\![\mathsf{let}\ x = U\ \mathsf{in}\ T]\!]^0_\eta &= \mathbf{let}_{\mathbf{R}}\ ev \leftarrow \mathbf{closure}^{\mathbf{E}}_{\mathbf{R}}\ [\![U]\!]^0_\eta \\
&\quad\ \mathbf{in}\ [\![T]\!]^0_{\eta, x \mapsto ev} \\
[\![\mathsf{letrec}\ x = U\ \mathsf{in}\ T]\!]^0_\eta &= \mathbf{let}_{\mathbf{R}}\ ev \leftarrow \mathbf{closurefix}^{\mathbf{E}}_{\mathbf{R}}\ (\lambda ev\ .\ [\![U]\!]^0_{\eta, x \mapsto ev}) \\
&\quad\ \mathbf{in}\ [\![T]\!]^0_{\eta, x \mapsto ev} \\
[\![\langle t \rangle]\!]^0_\eta &= \mathbf{let}_{\mathbf{R}}\ md \leftarrow \mathbf{closure}^{\mathbf{M}}_{\mathbf{N}}\ [\![t]\!]^1_\eta \\
&\quad\ \mathbf{in}\ \mathbf{unit}_{\mathbf{R}}\ (\mathsf{code} : md) \\
[\![\mathsf{lift}\ U\ \mathsf{using}\ W]\!]^0_\eta &= \mathbf{let}_{\mathbf{R}}\ v \leftarrow [\![U]\!]^0_\eta \\
&\quad\ \mathbf{in}\ \mathbf{case}\ (v, [\![W]\!]_\eta)\ \mathbf{of}\ \{ \\
&\quad\quad (\mathsf{int} : i, \mathsf{tint} : *) \to \mathbf{unit}_{\mathbf{R}}\ (\mathsf{code} : \mathbf{unit}_{\mathbf{M}}\ (\mathsf{dconst} : i)); \\
&\quad\quad \mathbf{otherwise} \to \mathbf{unit}_{\mathbf{R}}\ (\mathsf{wrong} : *) \\
&\quad\ \}
\end{aligned}
$$

Figure 9.21: Denotation of λ^{SC} stage 0 pure terms

performed only within the **IO** monad, the semantics may push the fresh name supply into this monad. (We could go even further and eliminate the fresh name supply altogether by simply re-using a global fresh name supply within the denotation of run. However, to do so would complicate the proof of type soundness.)

Figures 9.21 presents the denotation of non-monadic terms. Notice the use of **closure**, **closurefun** and **closurefix** (over various monads) to ensure the renaming environment is propagated to match the *static* lexical scope of the program. A semantics in which these closures also capture the environment η is also possible. (We chose not to do so because the present version forms the basis of a translation from λ^{SC} into a "vanilla" higher-order functional programming language lacking any staging constructs. In this case the target language provides partial-application closures implicitly.)

The denotation for deferred expressions makes it clear that $\langle t \rangle$ is a value, and thus t is *not* rebuilt when $\langle t \rangle$ is evaluated.

$$
\begin{array}{rl}
[\![\mathsf{unit}\ T]\!]^0_\eta = & \mathbf{let_R}\ ev \leftarrow \mathbf{closure^E_R}\ [\![T]\!]^0_\eta \\
& \mathbf{in\ unit_R}\ (\mathsf{cmd} : \mathbf{unit_{MIO}}\ ev) \\
[\![\mathsf{let}\ x \leftarrow U\ \mathsf{in}\ T]\!]^0_\eta = & \mathbf{let_R}\ ev \leftarrow \mathbf{closure^E_R}\ [\![U]\!]^0_\eta \\
& \mathbf{in\ let_R}\ f \leftarrow \mathbf{closurefun^E_R}\ (\lambda ev' \,.\, [\![T]\!]^0_{\eta, x \mapsto ev'}) \\
& \mathbf{in\ unit_R}\ (\mathsf{cmd} : \mathbf{let_{MIO}}\ v \leftarrow \mathbf{lift^{MIO}_E}\ ev \\
& \qquad \mathbf{in\ case}\ v\ \mathbf{of}\ \{ \\
& \qquad\quad \mathsf{cmd} : ioev \rightarrow \\
& \qquad\qquad \mathbf{let_{MIO}}\ ev' \leftarrow ioev \\
& \qquad\qquad \mathbf{in\ let_{MIO}}\ v' \leftarrow \mathbf{lift^{MIO}_E}\ (f\ ev') \\
& \qquad\qquad \mathbf{in\ case}\ v'\ \mathbf{of}\ \{ \\
& \qquad\qquad\quad \mathsf{cmd} : ioev' \rightarrow ioev'; \\
& \qquad\qquad\quad \mathbf{otherwise} \rightarrow \mathbf{unit_{MIO}}\ (\mathbf{unit_E}\ (\mathsf{wrong} : *)) \\
& \qquad\qquad \}; \\
& \qquad\quad \mathbf{otherwise} \rightarrow \mathbf{unit_{MIO}}\ (\mathbf{unit_E}\ (\mathsf{wrong} : *)) \\
& \qquad \}) \\
[\![\mathsf{run}\ T\ \mathsf{at}\ W]\!]^0_\eta = & \mathbf{let_R}\ ev \leftarrow \mathbf{closure^E_R}\ [\![T]\!]^0_\eta \\
& \mathbf{in\ unit_R}\ (\mathsf{cmd} : \mathbf{let_{MIO}}\ v \leftarrow \mathbf{lift^{MIO}_E}\ ev \\
& \qquad \mathbf{in\ case}\ v\ \mathbf{of}\ \{ \\
& \qquad\quad \mathsf{code} : md \rightarrow \\
& \qquad\qquad \mathbf{let_{MIO}}\ d \leftarrow \mathbf{lift^{MIO}_M}\ md \\
& \qquad\qquad \mathbf{in\ if}\ termOf(d)\ \text{well-defined} \\
& \qquad\qquad\quad \mathbf{and}\ (\overline{\Delta_{init}} \mid \mathbf{true} \mid \overline{\Gamma_{init}} \vdash^0 \\
& \qquad\qquad\qquad termOf(d) : typeOf([\![W]\!]_\eta) \\
& \qquad\qquad\qquad \hookrightarrow T')\ \mathbf{then} \\
& \qquad\qquad\qquad \mathbf{unit_{MIO}}\ (\mathbf{run_R}\ [\![T']\!]^0_\cdot) \\
& \qquad\qquad\quad \mathbf{else} \\
& \qquad\qquad\qquad \mathbf{throw_{MIO}}; \\
& \qquad\quad \mathbf{otherwise} \rightarrow \\
& \qquad\qquad \mathbf{unit_{MIO}}\ (\mathbf{unit_E}\ (\mathsf{wrong} : *)) \\
& \qquad \})
\end{array}
$$

Figure 9.22: Denotation of λ^{SC} stage 0 monadic terms

Figure 9.22 presents the denotation of the monadic constructs. The semantics of $\mathsf{let}\ x \leftarrow U\ \mathsf{in}\ T$ is complicated by the two-level nature of I/O computations. That is to say, we must be careful to distinguish evaluating an I/O command ("Is it defined?") from performing an I/O command ("What does it do?"). Most interesting is the semantics for $\mathsf{run}\ T\ \mathsf{at}\ W$. It creates an I/O command which, when performed, will rebuild T to a representation, d, of a run-time term, then check if $termOf(d)$ is a well-typed term in the empty kind and type contexts. If so, the type judgement will return a new run-time term T', which is then evaluated in the empty environment. Otherwise, an exception is thrown by $\mathbf{throw_{MIO}}$.

Finally, Figure 9.23 presents the denotation of the constants, which are straightforward (if somewhat tedious).

It is possible to refine the semantics of statically typed code in a number of ways. Firstly, because no constraints cross statically typed code boundaries, it is possible to translate

$$
\begin{array}{rl}
[\![i]\!]^0_\eta = & \mathbf{unit_R}\ (\mathsf{int} : i) \\
[\![\mathsf{throw}]\!]^0_\eta = & \mathbf{unit_R}\ (\mathsf{cmd} : \mathbf{throw_{MIO}}) \\
[\![(\mathsf{try}\ _\ \mathsf{catch}\ _)]\!]^0_\eta = & \lambda rio_1\ rio_2\ . \\
& \quad \mathbf{let_R}\ ev_1 \leftarrow \mathbf{closure^E_R}\ rio_1 \\
& \quad \mathbf{in\ let_R}\ ev_2 \leftarrow \mathbf{closure^E_R}\ rio_2 \\
& \quad \mathbf{in\ unit_R}\ (\mathsf{cmd} : \mathbf{let_{MIO}}\ v_1 \leftarrow \mathbf{lift^{MIO}_E}\ ev_1 \\
& \qquad \mathbf{in\ case}\ v_1\ \mathbf{of}\ \{ \\
& \qquad\quad \mathsf{cmd} : ioev_1 \rightarrow \\
& \qquad\qquad \mathbf{trycatch_{MIO}}\ ioev_1 \\
& \qquad\qquad\quad (\mathbf{let_{MIO}}\ v_2 \leftarrow \mathbf{lift^{MIO}_E}\ ev_2 \\
& \qquad\qquad\quad \mathbf{in\ case}\ v_2\ \mathbf{of}\ \{ \\
& \qquad\qquad\qquad \mathsf{cmd} : ioev_2 \rightarrow ioev_2; \\
& \qquad\qquad\qquad \mathbf{otherwise} \rightarrow \\
& \qquad\qquad\qquad\quad \mathbf{unit_{MIO}}\ (\mathbf{unit_E}\ (\mathsf{wrong} : *)) \\
& \qquad\qquad\quad \}); \\
& \qquad\quad \mathbf{otherwise} \rightarrow \mathbf{unit_{MIO}}\ (\mathbf{unit_E}\ (\mathsf{wrong} : *)) \\
& \qquad \}) \\
[\![\mathsf{putint}]\!]^0_\eta = & \lambda ri\ .\ \mathbf{let_R}\ ev \leftarrow \mathbf{closure^E_R}\ ri \\
& \quad \mathbf{in\ unit_R}\ (\mathsf{cmd} : \mathbf{let_{MIO}}\ v \leftarrow \mathbf{lift^{MIO}_E}\ ev \\
& \qquad \mathbf{in\ case}\ v\ \mathbf{of}\ \{ \\
& \qquad\quad \mathsf{int} : i \rightarrow \mathbf{let_{MIO}}\ _ \leftarrow \mathbf{putint_{MIO}}\ i \\
& \qquad\qquad\qquad \mathbf{in\ unit_{MIO}}\ (\mathbf{unit_E}\ (\mathsf{int} : 0)); \\
& \qquad\quad \mathbf{otherwise} \rightarrow \mathbf{unit_{MIO}}\ (\mathbf{unit_E}\ (\mathsf{wrong} : *)) \\
& \qquad \}) \\
[\![\mathsf{getint}]\!]^0_\eta = & \mathbf{unit_R}\ (\mathsf{cmd} : \mathbf{let_{MIO}}\ i \leftarrow \mathbf{getint_{MIO}} \\
& \qquad \mathbf{in\ unit_{MIO}}\ (\mathbf{unit_E}\ (\mathsf{int} : i)))
\end{array}
$$

Figure 9.23: Denotation of λ^{SC} stage 0 constants

these source terms to run-time terms during type checking. This compile-time translation is in contrast to the present approach which performs this translation only when such (rebuilt) code is to be run. Secondly, as a result of the first refinement, there is no need to type check statically typed code at run time at all.

To implement these refinements would unfortunately require duplicating much of the machinery currently shared between dynamically- and statically typed code. Because dynamically typed code *does* require the construction and type-checking of members of $\mathcal{D}$, it is easiest to make statically typed code do likewise.

9.5.5 Type Soundness

Run-time terms which encounter a catastrophic run-time type error are denoted by $[\mathsf{wrong} : *]$. We first show the denotation of every well-kinded type does not include such a value.

Lemma 9.5 If $\Delta_{init}\ ;\ \overline{\Delta'} \vdash^0 \tau : \mathsf{Type}$ and $\Delta_{init}\ ;\ \overline{\Delta'} \vdash^0 \Gamma_{init}\ ;\ \overline{\Gamma'}$ context, then $[\mathsf{wrong} : *] \notin$

$[\![\tau]\!]_{(\overline{\Delta'},\overline{\Gamma'})}$.

Proof By induction on derivation of $\Delta_{init}\,;\overline{\Delta'} \vdash^0 \tau : \mathsf{Type}$. □

Given a constraint C s.t. $\Delta\,;\overline{\Delta_{init}} \vdash^0 C\ \mathsf{constraint}$, we say C is *satisfiable in* Δ if $\texttt{true} \vdash^e \texttt{exists}\ \Delta\ .\ C$. We say a type scheme $\texttt{forall}\ \Delta\ .\ C \texttt{ => } \tau$ s.t. $\Delta\,;\overline{\Delta_{init}} \vdash^0 C\ \mathsf{constraint}$ is satisfiable iff C is satisfiable in Δ. Finally, we say a type context Γ is satisfiable if $\forall (x:\sigma) \in \Gamma$, σ is satisfiable.

We say η *models* Γ *with respect to* $(\overline{\Delta'},\overline{\Gamma'})$, written $\eta \models_{(\overline{\Delta'},\overline{\Gamma'})} \Gamma$, if Γ is satisfiable and $dom(\Gamma) \subseteq dom(\eta)$ and $\forall (x:\tau) \in \Gamma\ .\ \eta\ x \in \mathbf{E}\ [\![\tau]\!]_{(\overline{\Delta'},\overline{\Gamma'})}$.

Let ρ range over injective finite renaming environments mapping variable names to (fresh) names. Note that ρ need not be idempotent.

We now show that the denotation of the translation of a well-typed term is a member of the denotation of its type. The theorem statement is quite complex, since it must tie together:

- the static kind context $(\Delta\,;\overline{\Delta'})$ and static type contexts $(\Gamma\,;\overline{\Gamma'})$;
- the current renaming ρ, which has domain $\overline{\Gamma'}$;
- the current dynamic type context $\overline{\Gamma_r}$, which assigns a type to all the variables in the range of ρ; and
- for higher staged terms, an arbitrary extension $(\overline{\Delta_e},\overline{\Gamma_e})$ to $(\overline{\Delta'},\overline{\Gamma_r})$, and the current fresh name supply nms, which cannot contain any names from $\overline{\Gamma_r} +\!\!+ \overline{\Gamma_e}$.

Theorem 9.6 (Soundness)

(i) If $\Delta\,;\overline{\Delta'} \mid C \mid \Gamma\,;\overline{\Gamma'} \vdash^0 t : \tau \hookrightarrow T$, and $\Delta \vdash \theta\ \mathsf{gsubst}$, and $\texttt{true} \vdash^e \theta\ C \hookrightarrow B$, and $\rho\,\overline{\Gamma'} \subseteq \overline{\Gamma_r}$, and $\eta \models_{(\overline{\Delta'},\theta\,\overline{\Gamma_r})} \theta\,\Gamma$ then $[\![T]\!]^0_{\eta +\!\!+ env(B)}\ \rho \in [\![\theta\ \tau]\!]_{(\overline{\Delta'},\theta\,\overline{\Gamma_r})}$

(ii) If $\Delta\,;\overline{\Delta'} \mid C\,;\overline{C'} \mid \Gamma\,;\overline{\Gamma'} \vdash^{n+1}_b t : \tau \hookrightarrow t'$, and $\Delta \vdash \theta\ \mathsf{gsubst}$, and $\texttt{true} \vdash^e \theta\ C \hookrightarrow B$, and $\rho\ \overline{\Gamma'} \subseteq \overline{\Gamma_r}$, and $\eta \models_{(\overline{\Delta'},\theta\,\overline{\Gamma_r})} \theta\ \Gamma$, and $nms \in Names_{\setminus \overline{\Gamma_r} +\!\!+ \overline{\Gamma_e}}$ and $(\overline{\Delta_e},\overline{\Gamma_e})$ extends $(\overline{\Delta'},\theta\,\overline{\Gamma_r})$ then $[\![t']\!]^{n+1}_{\eta +\!\!+ env(B)}\ (nms,\rho) \in \mathbf{E}\ \mathcal{D}_{wd}$, where

$$\mathcal{D}_{wd} = \left\{ d \in \mathcal{D} \;\middle|\; \begin{array}{l} termOf(d)\ \text{well-defined}, \\ \forall i\ .\ vars(i-n, termOf(d)) \subseteq dom(\overline{\Gamma_r}^{\,i}) \end{array} \right\}$$

(iii) If, in addition to the hypotheses of (ii), we also have $b = \mathbf{tt}$ then $[\![t']\!]^{n+1}_{\eta +\!\!+ env(B)}\ (nms,\rho) \in \mathbf{E}\ \mathcal{D}_{wt}$, where, if $n > 0$ then

$$\mathcal{D}_{wt} = \mathbf{E} \left\{ d \in \mathcal{D} \;\middle|\; \begin{array}{l} termOf(d)\ \text{well-defined}, \\ \overline{\Delta'} +\!\!+ \overline{\Delta_e} \mid \theta\ \overline{C'} \mid (\theta\ \overline{\Gamma_r}) +\!\!+ \overline{\Gamma_e} \vdash^n_{\mathbf{tt}} termOf(d) : \theta\ \tau \end{array} \right\}$$

otherwise

$$\mathcal{D}_{wt} = \mathbf{E} \left\{ d \in \mathcal{D} \;\middle|\; \begin{array}{l} termOf(d)\ \text{well-defined}, \\ \overline{\Delta'} +\!\!+ \overline{\Delta_e} \mid \theta\ \overline{C'} \mid (\theta\ \overline{\Gamma_r}) +\!\!+ \overline{\Gamma_e} \vdash^0 termOf(d) : \theta\ \tau \end{array} \right\}$$

Proof See Theorem D.8. □

Finally, the translation of a well-typed term never encounters a catastrophic type error.

Corollary 9.7 If $\overline{\Delta_{init}} \mid \mathtt{true} \mid \overline{\Gamma_{init}} \vdash^0 t : \tau \hookrightarrow T$ then $[\![T]\!]^0_{\cdot}\ \emptyset \neq [\mathsf{wrong} : *]$.

Proof Immediate from Theorem 9.6 and and Lemma 9.5. $\square$

Chapter 10

Conclusions to Part II

10.1 Related Work

Two-stage functional languages were first developed to express the results of binding time analysis in preparation for partial evaluation [51, 75, 76]. Nielson and Nielson generalised the concept to arbitrary stages [77]. The syntax of λ^{SC} has its origin in Lisp [61]: our `{? ... ?}`, `~` and **run** operators roughly correspond with Lisp's `'( ... )`, `‘` and `eval` operators. Of course Lisp must perform run-time type checking for *every* expression, dynamically generated or otherwise.

Davies and Pfenning demonstrated two Curry-Howard correspondences for staged languages. Staging restricted to closed-code corresponds with the modal calculus S4 [21], while staging with open-code but without a **run** operator corresponds with a linear-time temporal logic [20]. A naïve combination of these two calculi in which the distinction between closed and open code is forgotten is unsound: *viz* **run** may encounter unbound variables [107]. Motivated by the categorical framework of Benaissa *et al.* [10], Taha *et al.* [106] have developed a sound calculus which supports both closed and open code, but at the cost of a somewhat clumsy syntax in which the free variables of open code must be explicitly "reconnected" whenever code is spliced.

The statically typed code fragment of λ^{SC} is based upon MetaML [104, 107, 97]. However, unlike MetaML, λ^{SC} is careful to restrict the use of **run** so as to avoid the open-code problem mentioned above.

The dynamically typed code fragment of λ^{SC} is an extensive reworking of the calculus, λdyn, presented in Shields, Sheard and Peyton Jones [99]. The differences are significant:

- λdyn supports only unconstrained parametric polymorphism, whereas λ^{SC} supports arbitrary constrained polymorphism.
- λdyn is call-by-value, λ^{SC} is call-by-name or call-by-need.
- λdyn assumes a full type-passing-based implementation, whereas λ^{SC} passes run-time representations of types only where they are required by **run**.
- λdyn's type system does not prevent the application of `run` to open code. Instead, such code is regarded as ill-typed at run-time. By contrast, λ^{SC} places `run` in the `IO` monad, which restricts its use, but also ensures at compile-time that only closed code may be run.
- λdyn uses Wright and Felleisen's [115] style of context-based small-step operational

semantics. This semantics requires an (infinite) family of mutually-recursively defined rewrite contexts. Type soundness is shown by subject reduction. In contrast, λ^{sc} uses a denotational semantics and type soundness is shown model-theoretically.

- λdyn's operational semantics handled the problem of variable renaming implicitly: β-reduction is assumed to avoid name capture by "inventing" a fresh name as required. In λ^{sc}, this aspect is modelled explicitly, and hence we believe, more honestly.

- Both λdyn and λ^{sc} use a type-directed translation into a run-time language. However, because λdyn does not support type constraints, its translation does not need to introduce any witness passing. Thus it is possible to translate all λdyn code fragments into the run-time language at compile-time, and execution need only splice these fragments together to generate a final run-time term. The operational semantics for λdyn exploits this compile-time translation by simplifying the run-time type checking problem to a series of *residual unification problems* performed at each splice point. In contrast, dynamically typed code in λ^{sc} cannot be translated to the run-time language at compile-time, since the translation depends on which constraints arise at run-time. As a result, λ^{sc} requires full type checking of terms at run-time, and no simplification is possible.

The denotational semantics of Sections 9.5.3 and 9.5.4 is somewhat of a chimera. Its motivation is the functor-category semantics for two-level languages of Moggi [73], but re-presented in a point-full form with a concrete base category of well-kinded type contexts and well-typed environments. The beauty of this semantics is that, at the term level, it has the simplicity of Gomard and Jones' [33] original denotational semantics for two-level languages (though with the fresh names supply made explicit rather than left implicit as in their work).

It is unclear whether the statically typed code fragment of λ^{sc} could be given a categorical semantics within the framework of Benaissa *et al.* [10].

More recently, Gabbay and Pitts [30] have developed a non-standard set-theoretic foundation, and Fiore, Plotkin and Turi [28] a category-theoretic framework, for inductive datatypes involving name binders. In both theories, α-conversion is "built-in." Representing the semantics of λ^{sc} in one of these settings would effectively factor out all explicit manipulation of variable names, resulting in a tremendous simplification.

10.2 Conclusions and Future Work

We presented λ^{sc}, a calculus supporting the run-time generation of both statically and dynamically typed code. It is flexible enough to allow code fragments to contain free variables, while also ensuring such variables are always bound within result code. For dynamically typed code, type checking is deferred until just before the code is to be run. Run-time generated code which is ill-typed raises an exception, and hence may be handled gracefully. On the other hand, we have shown that statically typed code is always well-typed, and hence requires no run-time check. We demonstrated the utility of mixing both statically and dynamically typed code within a single program.

The calculus also supports constrained polymorphism, and hence many other type features such as the type-indexed-rows of Part I, implicit parameters [57], and type classes [47, 109].

This suggests λ^{sc} is a suitable foundation on which to implement full-scale multi-staged languages.

To the author's knowledge, λ^{sc} is the first system to combine all of these features.

We have not yet developed a type inference system for λ^{sc}. Because both statically and dynamically typed code share the same three constructs, inference may be problematic. In this case we may need to syntactically distinguish these constructs.

On the theoretical side, though we have shown (model-theoretic) type soundness for λ^{sc}, we have not shown *staging-correctness*. The usual approach [71, 77] is to first define an erasure function taking a multi-stage term to a single-stage term by erasing all `{{ }}` and `~` operators. Then a logical relation [69] is constructed between multi-stage terms and their stage-erasure. By the logical relations lemma, correctness follows if all constants are related. It is not at all obvious such an approach will work for λ^{sc}, particularly given its rich type structure and the remarks of Moggi [71].

Appendix A

Recognising XML Elements

This appendix shows how to extend the syntax and typing rules of λ^{TIR} (as presented in Chapters 4 and 5) to handle terms in native XML syntax (as outlined in Section 3.4). Our exposition is extremely brief, and no proofs of correctness are provided. Though awkward to express formally, the material of this appendix is for the most part a straightforward application of automata theory.

(The reader interested in what my long suffering supervisors, John Launchbury and Simon Peyton Jones, have had to put up with over the years is invited to attempt to decipher this material without the aid of the explanatory text.)

Recall from Section 3.4 that τ `*` is shorthand for `List` τ, where `List` is the datatype:

```
data List = \a . Cons (a, List a) | Nil
```

Similarly, τ `?` is shorthand for `Option` τ, where `Option` is the datatype:

```
data Option = \a . Some a | None
```

Also recall (τ_1 `|` ... `|` τ_n) abbreviates `One` (τ_1 `#` ... `#` τ_n `# Empty`), and dually, (τ_1 `&` ... `&` τ_n) abbreviates `All` (τ_1 `#` ... `#` τ_n `# Empty`).

Figure A.1 presents the required extensions to λ^{TIR} types, terms and patterns. An element, e, is a tag delimited sequence of element items, ei. We allow an element item to "escape" from XML syntax back to native λ^{TIR} syntax by using the special `<<...>>` form. A tag is a saturated newtype of the form $A\ \tau_1 \ldots \tau_n$. Each τ_i must be a monotype, and the application must have kind `Type`. This restriction is necessary in order to be able to construct an automaton for the body of A (see Section 3.4). We extend the language of λ^{TIR} terms with strings, elements, and the data constructors of the above datatype declarations.

XML elements may also appear within patterns. For the most part XML patterns are handled analogously to XML elements within terms, hence we shall elide the rules dealing with them.

Figure A.1 also presents some additional structure required by recognisers. We shall be constructing Glushkov automata [17] which have as states the *positions* of a regular expression (type). Hence we take as the set of states for type τ all ways of delimiting the sub-terms of τ by $[\cdot]$. In other words, a position, p, is a factorisation of τ into a context, P, and a sub-term, v, of τ such that $\tau = P[v]$.

The Glushkov automata we shall construct will be augmented to rewrite a sequence of XML sub-elements into a λ^{TIR} term in native syntax. To this end, the automata include a stack, *st*, of intermediate run-time terms, and the transition function specifies a sequence of stack actions, *acts*, to be performed on the stack when making a transition.

Types	$\tau, v ::= \texttt{String} \mid \ldots$	
Strings	str	
Elements	$e ::= \texttt{<}A\ \tau_1 \ldots \tau_m\texttt{>}\ ei_1 \ldots ei_n\ \texttt{</}A\texttt{>}$	$m, n \geq 0$
Element patterns	$ep ::= \texttt{<}A\ \tau_1 \ldots \tau_m\texttt{>}\ eip_1 \ldots eip_n\ \texttt{</}A\texttt{>}$	$m, n \geq 0$
Element items	$ei ::= str \mid e \mid \texttt{<<}\ t\ \texttt{>>}$	
Element pattern items	$eip ::= str \mid ep \mid \texttt{<<}\ p\ \texttt{>>}$	
Terms	$t, u ::= \texttt{"}str\texttt{"} \mid e \mid \texttt{Cons} \mid \texttt{Nil} \mid \texttt{Some} \mid \texttt{None} \mid \ldots$	
Patterns	$p, q ::= \texttt{"}str\texttt{"} \mid ep \mid \texttt{Cons}\ p\ q \mid \texttt{Nil} \mid \texttt{Some}\ p \mid \texttt{None} \mid \ldots$	
Recogniser position contexts	$P[\bullet] ::= \bullet \mid P[\bullet]\ \texttt{*} \mid P[\bullet]\ \texttt{?}$	
	$\mid (\tau_1\texttt{,} \ldots\texttt{,}\ P[\bullet]\texttt{,} \ldots\texttt{,}\ \tau_n)$	$n \geq 0$
	$\mid (\tau_1\ \texttt{\|} \ldots \texttt{\|}\ P[\bullet]\ \texttt{\|} \ldots \texttt{\|}\ \tau_n)$	$n \geq 2$
	$\mid (\tau_1\ \texttt{\&} \ldots \texttt{\&}\ P[\bullet]\ \texttt{\&} \ldots \texttt{\&}\ \tau_n)$	$n \geq 2$
Recogniser positions	$p ::= P[\tau]$	
Recogniser actions	$act ::= \texttt{null} \mid \texttt{tuple}(n) \mid \texttt{[} \mid \texttt{]} \mid \texttt{none} \mid \texttt{some}$	
	$\mid \texttt{inj}(i) \mid \texttt{<} \mid \texttt{unseen}(i) \mid \texttt{seen}(i) \mid \texttt{prod}(n)$	
Action sequence	$acts ::= \cdot \mid act, acts$	
Special stack term	$special ::= \texttt{[} \mid \texttt{<} \mid \texttt{unseen}(i) \mid \texttt{seen}(i)$	
Term recogniser stack	$st ::= \cdot \mid st, special \mid st, T$	

Figure A.1: Extensions to λ^{TIR} types, terms and patterns for handling XML elements, and syntax for recogniser components

Thus, each automaton includes a simple stack machine with the following operators:

- `null` pushes the empty string "".
- `tuple(`n`)` pops n terms and pushes their aggregation as a tuple.
- `[` pushes itself as a "start of list" marker.
- `]` pops the stack back to and including the last `[` marker, and pushes the aggregation of all popped terms as a list.
- `none` pushes None.
- `some` pops a term T and pushes Some T.
- `inj(`i`)` pops a term T and pushes Inj (Inc^{i-1} One) T. We write Inc^j to denote j applications of Inc.
- `<` pushes itself as a "start of unordered sequence" marker.
- `unseen(`i`)` pushes itself to signal the topmost term as a "default" to use if the i'th (in canonical order) member of an unordered sequence is missing.
- `seen(`i`)` pushes itself to signal the topmost term has occurred as the i'th (in canonical order) member of an unordered sequence.

$$\boxed{\mathcal{S}(st \mid acts) = st'}$$

$$
\begin{aligned}
\mathcal{S}(st \mid \cdot) &= st \\
\mathcal{S}(st \mid \texttt{null} +\!\!+ acts) &= \mathcal{S}(st +\!\!+ \texttt{""} \mid acts) \\
\mathcal{S}(st +\!\!+ U_1, \ldots, U_n \mid \texttt{tuple}(n) +\!\!+ acts) &= \mathcal{S}(st +\!\!+ \langle U_1, ..., U_n \rangle \mid acts) \\
\mathcal{S}(st \mid \texttt{[} +\!\!+ acts) &= \mathcal{S}(st +\!\!+ \texttt{[} \mid acts) \\
\mathcal{S}(st +\!\!+ \texttt{[}, U_1, \ldots, U_n \mid \texttt{]} +\!\!+ acts) &= \mathcal{S}(st +\!\!+ (\mathsf{Cons}\ U_1\ (\ldots\ (\mathsf{Cons}\ U_n\ \mathsf{Nil})\ldots)) \mid acts) \\
\mathcal{S}(st \mid \texttt{none} +\!\!+ acts) &= \mathcal{S}(st +\!\!+ \mathsf{None} \mid acts) \\
\mathcal{S}(st +\!\!+ U \mid \texttt{some} +\!\!+ acts) &= \mathcal{S}(st +\!\!+ (\mathsf{Some}\ U) \mid acts) \\
\mathcal{S}(st +\!\!+ U \mid \texttt{inj}(i) +\!\!+ acts) &= \mathcal{S}(st +\!\!+ (\mathsf{Inj}\ (\mathsf{Inc}^{i-1}\ \mathsf{One})\ U) \mid acts) \\
\mathcal{S}(st \mid \texttt{<} +\!\!+ acts) &= \mathcal{S}(st +\!\!+ \texttt{<} \mid acts) \\
\mathcal{S}(st \mid \texttt{unseen}(i) +\!\!+ acts) &= \mathcal{S}(st +\!\!+ \texttt{unseen}(i) \mid acts) \\
\mathcal{S}(st \mid \texttt{seen}(i) +\!\!+ acts) &= \mathcal{S}(st +\!\!+ \texttt{seen}(i) \mid acts)
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{S}(st +\!\!+ \texttt{<}, U_1, \texttt{unseen}(i_1), \ldots, U_n, \texttt{unseen}(i_n), \\
&\quad T_1, \texttt{seen}(j_1), \ldots, T_m, \texttt{seen}(j_m) \mid \texttt{prod}(n') +\!\!+ acts) = \\
&\qquad\qquad \mathcal{S}(st +\!\!+ \langle T'_1, ..., T'_{n'} \rangle \mid acts) \\
&\text{where} \\
&\quad \forall 0 < k, k' \leq m \,.\, i_k = i_{k'} \implies k = k' \\
&\quad \forall 0 < k, k' \leq m \,.\, j_k = j_{k'} \implies k = k' \\
&\quad \forall 0 < k \leq n' \,.\, T'_k = \begin{cases} T_{k'}, & \textbf{if } k = j_{k'} \\ U_{k'}, & \textbf{if } \nexists k'' \,.\, k = j_{k''} \wedge k = i_{k'} \\ \textbf{undefined}, & \textbf{otherwise} \end{cases}
\end{aligned}
$$

Figure A.2: Executing a sequence of recogniser actions upon a stack of λ^{TIR} run-time terms

- `prod(n)` pops the stack back to and including the last `<` marker, and pushes a tuple. The term in position i of the tuple is either the popped term which was marked by `seen(i)`, or if no such term exists, the popped term which was marked by `unseen(i)`.

These actions are formalised in Figure A.2.

Figures A.3, A.4 and A.5 present the definition of the function $\mathcal{G}$. Given a position $P[\tau]$, where τ is a monotype, $\mathcal{G}$ constructs a transition function for an augmented Glushkov automaton recognising the language of τ when viewed as a regular expression. The alphabet of this language is λ^{TIR} monotypes. $\mathcal{G}$ is undefined if τ is not 1-unambiguous as a regular expression.

In XMλ, the constraint `readable` τ is true if τ is a newtype application whose normalised body is in the domain of $\mathcal{G}$. The constraint `writable` τ is true if, furthermore, this body type is first-order. The witness for both of these constraints is the automaton constructed by $\mathcal{G}$.

A version of $\mathcal{G}$ for constructing un-augmented Glushkov automata was presented as an example in Section 8.2. A simpler version of this construction may also be found in Brüggemann-Klein *et al.* [13].

To ease the notation we shall adopt an informal record-like syntax. Given a position $P[\tau]$,

$$\boxed{\mathcal{G}[\![P[\tau]]\!] = \{pos\ ;\ empty\ ;\ first\ ;\ last\ ;\ follow\}}$$

$$\begin{array}{lll}
\mathcal{G}[\![P[\texttt{String}]]\!] = \{ & \mathcal{G}[\![P[\upsilon \texttt{ -> } \tau]]\!] = \{ & \mathcal{G}[\![P[A]]\!] = \{ \\
\quad pos = \{P[\texttt{String}]\}; & \quad pos = \{P[\upsilon \texttt{ -> } \tau]\}; & \quad pos = \{P[A]\}; \\
\quad empty = \mathbf{null}; & \quad empty = \cdot; & \quad empty = \cdot; \\
\quad first = \{P[\texttt{String}](\cdot)\}; & \quad first = \{P[\upsilon \texttt{ -> } \tau](\cdot)\}; & \quad first = \{P[A](\cdot)\}; \\
\quad last = \{P[\texttt{String}](\cdot)\}; & \quad last = \{P[\upsilon \texttt{ -> } \tau](\cdot)\}; & \quad last = \{P[A](\cdot)\}; \\
\quad follow = \lambda p\ .\ \emptyset & \quad follow = \lambda p\ .\ \emptyset & \quad follow = \lambda p\ .\ \emptyset \\
\} & \} & \}
\end{array}$$

$$\begin{array}{l}
\mathcal{G}[\![P[(\tau_1\texttt{,}\ \ldots\texttt{,}\ \tau_n)]]\!] = \{ \\
\quad pos = \{thispos\} \cup \bigcup_{0<i\leq n} pos'_i; \\
\quad empty = emptyacts_{0,n+1}; \\
\quad first = \{thispos(\cdot)\} \cup \bigcup_{0<i\leq initne} \{p(emptyacts_{0,i} +\!\!+ acts) \mid p(acts) \in first'_i\}; \\
\quad last = \{thispos(\cdot)\} \cup \bigcup_{finalne\leq i\leq n} \{p(acts +\!\!+ emptyacts_{i,n+1}) \mid p(acts) \in last'_i\}; \\
\quad follow = \lambda p\ .\ \bigcup_{0<i\leq n} \begin{cases} follow'_i\ p \cup \bigcup_{i<j\leq \mathbf{min}(nextne_i,n)} & \\ \quad \{p'(acts +\!\!+ emptyacts_{i,j} +\!\!+ acts') \mid p'(acts') \in first'_j\}, & \\ \qquad\qquad \mathbf{if}\ p(acts) \in last'_i & \\ follow'_i\ p, \quad \mathbf{if}\ p \in pos'_i & \\ \emptyset, \quad \mathbf{otherwise} & \end{cases} \\
\}
\end{array}$$

where

$$\begin{array}{l}
\forall\, 0 < i \leq n\ .\ P'_i[\bullet] = P[(\tau_1\texttt{,}\ \ldots\texttt{,}\ \tau_{i-1}\texttt{,}\ \bullet\texttt{,}\ \tau_{i+1}\texttt{,}\ \ldots\texttt{,}\ \tau_n)] \\
\forall\, 0 < i \leq n\ .\ \{pos'_i\ ;\ empty'_i\ ;\ first'_i\ ;\ last'_i\ ;\ follow'_i\} = \mathcal{G}[\![P'_i[\tau_i]]\!] \\
\forall\, 0 \leq i \leq n\ .\ nextne_i = \mathbf{max}\ i < j \leq n+1\ .\ \forall i < k < j\ .\ empty'_k \neq \cdot \\
\forall\, 0 < i \leq n+1\ .\ prevne_i = \mathbf{min}\ 0 \leq j < i\ .\ \forall j < k < i\ .\ empty'_k \neq \cdot \\
\forall\, 0 \leq i < j \leq n+1\ .\ emptyacts_{i,j} \\
\qquad = \begin{cases} +\!\!+_{i<k<j}\, empty'_k +\!\!+ \texttt{tuple}(n), & \mathbf{if}\ \forall i < k < j\ .\ empty'_k \neq \cdot \wedge j = n+1 \\ +\!\!+_{i<k<j}\, empty'_k, & \mathbf{if}\ \forall i < k < j\ .\ empty'_k \neq \cdot \\ \cdot, & \mathbf{otherwise} \end{cases} \\
\quad thispos = P[(\tau_1\texttt{,}\ \ldots\texttt{,}\ \tau_n)] \\
\quad initne = \mathbf{min}(nextne_0, n) \\
\quad finalne = \mathbf{max}(prevne_{n+1}, 1)
\end{array}$$

Figure A.3: Building a recogniser from a λ^{TIR} type (part 1 of 3)

$$\mathcal{G}[\![P[(\tau_1 \mid \ldots \mid \tau_n)]]\!] = \{$$

$$pos = \{thispos\} \cup \bigcup_{0<i\leq n} pos'_i;$$

$$empty = \begin{cases} empty'_j \mathbin{+\!\!+} \mathtt{inj}(\pi^{-1}\ j), & \textbf{if } \forall\, 0 < k \leq n \;.\; empty'_k \neq \cdot \implies k = j \\ \cdot, & \textbf{if } \forall\, 0 < k \leq n \;.\; empty'_k = \cdot \\ \textbf{undefined}, & \textbf{otherwise}; \end{cases}$$

$$first = \{thispos(\cdot)\} \cup \bigcup_{0<i\leq n} first'_i;$$

$$last = \{thispos(\cdot)\} \cup \bigcup_{0<i\leq n} \{p(acts \mathbin{+\!\!+} \mathtt{inj}(\pi^{-1}\ i)) \mid p(acts) \in last'_i\};$$

$$follow = \lambda p \;.\; \bigcup_{0<i\leq n} \begin{cases} follow'_i\ p, \textbf{if } p \in pos'_i \\ \emptyset \qquad\quad ,\textbf{otherwise} \end{cases}$$

$$\}$$

where

$$\forall\, 0 < i \leq n \;.\; P'_i[\bullet] = P[(\tau_1 \mid \ldots \mid \tau_{i-1} \mid \bullet \mid \tau_{i+1} \mid \ldots \mid \tau_n)]$$

$$\forall\, 0 < i \leq n \;.\; \{pos'_i\,;\, empty'_i\,;\, first'_i\,;\, last'_i\,;\, follow'_i\} = \mathcal{G}[\![P'_i[\tau_i]]\!]$$

$$thispos = P[(\tau_1 \mid \ldots \mid \tau_n)]$$

$$\{\pi\} = sortingPerms(\tau_1, \ldots, \tau_n)$$

$$\mathcal{G}[\![P[(\tau_1 \,\&\, \ldots \,\&\, \tau_n)]]\!] = \{$$

$$pos = \{thispos\} \cup \bigcup_{0<i\leq n} pos'_i;$$

$$empty = \begin{cases} emptyacts \mathbin{+\!\!+} \mathtt{prod}(n), & \textbf{if } \forall\, 0 < i \leq n \;.\; empty'_i \neq \cdot \\ \cdot, & \textbf{otherwise}; \end{cases}$$

$$first = \{thispos(\cdot)\} \cup \bigcup_{0<i\leq n} \{p(emptyacts \mathbin{+\!\!+} acts) \mid p(acts) \in first'_i\};$$

$$last = \{thispos(\cdot)\} \cup \bigcup_{0<i\leq n} \{p(acts \mathbin{+\!\!+} \mathtt{seen}(\pi^{-1}\ i) \mathbin{+\!\!+} \mathtt{prod}(n)) \mid p(acts) \in last'_i\};$$

$$follow = \lambda p \;.\; \bigcup_{0<i\leq n} \begin{cases} \bigcup_{0<j\leq n} \{p'(acts \mathbin{+\!\!+} \mathtt{seen}(\pi^{-1}\ i) \mathbin{+\!\!+} acts') \mid p'(acts') \in first_j\}, \\ \qquad\qquad \textbf{if } p(acts) \in last'_i \\ follow'_i\ p, \quad \textbf{if } p \in pos'_i \\ \emptyset, \qquad\quad \textbf{otherwise} \end{cases}$$

$$\}$$

where

$$\forall\, 0 < i \leq n \;.\; P'_i[\bullet] = P[(\tau_1 \,\&\, \ldots \,\&\, \tau_{i-1} \,\&\, \bullet \,\&\, \tau_{i+1} \,\&\, \ldots \,\&\, \tau_n)]$$

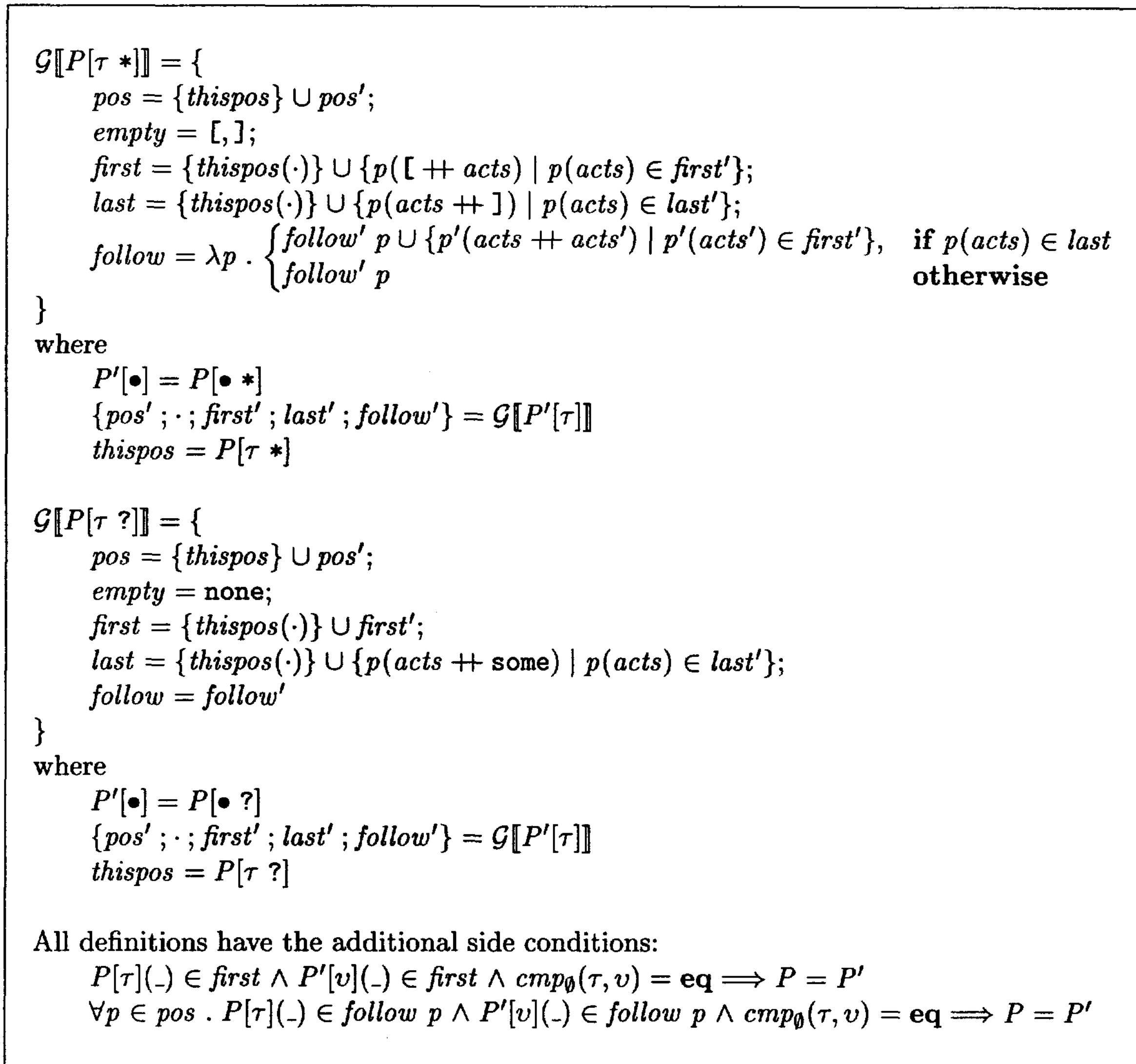
$$\forall\, 0 < i \leq n \;.\; \{pos'_i\,;\, empty'_i\,;\, first'_i\,;\, last'_i\,;\, follow'_i\} = \mathcal{G}[\![P'_i[\tau_i]]\!]$$

$$thispos = P[(\tau_1 \,\&\, \ldots \,\&\, \tau_n)]$$

$$\{\pi\} = sortingPerms(\tau_1, \ldots, \tau_n)$$

$$emptyacts = \mathtt{<} \mathbin{+\!\!+} \mathop{+\!\!+}_{0<i\leq n} \begin{cases} empty'_i \mathbin{+\!\!+} \mathtt{unseen}(\pi^{-1}\ i), & \textbf{if } empty'_i \neq \cdot \\ \cdot, & \textbf{otherwise} \end{cases}$$

Figure A.4: Building a recogniser from a λ^{TIR} type (part 2 of 3)



$$\mathcal{G}[\![P[\tau\ *]]\!] = \{$$
$$pos = \{thispos\} \cup pos';$$
$$empty = [,];$$
$$first = \{thispos(\cdot)\} \cup \{p([\ +\!\!+\ acts) \mid p(acts) \in first'\};$$
$$last = \{thispos(\cdot)\} \cup \{p(acts +\!\!+\ ]) \mid p(acts) \in last'\};$$
$$follow = \lambda p\ .\ \begin{cases} follow'\ p \cup \{p'(acts +\!\!+ acts') \mid p'(acts') \in first'\}, & \textbf{if } p(acts) \in last \\ follow'\ p & \textbf{otherwise} \end{cases}$$
$$\}$$

where

$$P'[\bullet] = P[\bullet\ *]$$
$$\{pos'\ ;\cdot\ ;first'\ ;last'\ ;follow'\} = \mathcal{G}[\![P'[\tau]]\!]$$
$$thispos = P[\tau\ *]$$

$$\mathcal{G}[\![P[\tau\ ?]]\!] = \{$$
$$pos = \{thispos\} \cup pos';$$
$$empty = \textbf{none};$$
$$first = \{thispos(\cdot)\} \cup first';$$
$$last = \{thispos(\cdot)\} \cup \{p(acts +\!\!+ \texttt{some}) \mid p(acts) \in last'\};$$
$$follow = follow'$$
$$\}$$

where

$$P'[\bullet] = P[\bullet\ ?]$$
$$\{pos'\ ;\cdot\ ;first'\ ;last'\ ;follow'\} = \mathcal{G}[\![P'[\tau]]\!]$$
$$thispos = P[\tau\ ?]$$

All definitions have the additional side conditions:

$$P[\tau](_) \in first \wedge P'[v](_) \in first \wedge cmp_{\emptyset}(\tau, v) = \textbf{eq} \Longrightarrow P = P'$$
$$\forall p \in pos\ .\ P[\tau](_) \in follow\ p \wedge P'[v](_) \in follow\ p \wedge cmp_{\emptyset}(\tau, v) = \textbf{eq} \Longrightarrow P = P'$$

Figure A.5: Building a recogniser from a λ^{TIR} type (part 3 of 3)

$\mathcal{G}$ constructs a record of the form:

$$\begin{array}{l} \{ \\ \quad pos = \{p_1, \ldots, p_n\}; \\ \quad empty = acts; \\ \quad first = \{p_1(acts_1), \ldots, p_n(acts_n)\}; \\ \quad last = \{p_1(acts_1), \ldots, p_n(acts_n)\}; \\ \quad follow = f \\ \} \end{array}$$

where

- *pos* is the set of sub-positions of τ, including τ itself.
- *empty* is a sequence of actions which will construct (on the top of the stack) a run-time term of type τ to represent that τ is "missing." For example, a missing `Just` τ is

$$
\begin{aligned}
&\mathcal{R}\ \{pos'\ ;\ empty'\ ;\ first'\ ;\ last'\ ;\ follow'\} = \{\\
&\quad last = last' \cup \textbf{if}\ empty \neq \cdot\ \textbf{then start}(empty)\ \textbf{else}\ \emptyset;\\
&\quad follow = \lambda p\ .\ \textbf{if}\ p = \textbf{start then}\ first'\ \textbf{else}\ follow'\ p\\
&\}
\end{aligned}
$$

Figure A.6: Converting a recogniser to use **start**

constructed by the action none, which constructs the run-time term None. Similarly, a missing $\tau *$ is constructed by the actions [,], which construct Nil. If τ must be present, then *empty* will be empty.

- *first* is a set of $(p, acts)$ pairs representing all possible first positions of τ, and for each position, the actions to perform *before* moving to the position. We shall write these pairs in the form $p(acts)$, to signal that *acts* is determined from p.
- *last* is the dual to *first*. It contains all possible last positions of τ, and for each position, the actions to perform *after* leaving the position.
- *follow* is a function, f, from positions to sets of $(p, acts)$ pairs, representing the automaton's transition function. If $f\ p' = \{p_1(acts_1), \ldots, p_n(acts_n)\}$ then for each i, actions $acts_i$ should be performed if a transition is made from p' to p_i.

The definition of $\mathcal{G}$ is straightforward, but unfortunately very ugly. Since we wish to be able to construct empty terms, such as a tuple, in a single step, we are prevented from using the more elegant recursive decomposition of composite types.

Figure A.6 presents the function $\mathcal{R}$. Given a record constructed by $\mathcal{G}$, $\mathcal{R}$ builds *first* and *follow* members which use the dummy position **start** to signify the initial position. This function allows us to discard *pos*, *last* and *empty* in what follows.

Finally, we come to the problem of type inference for XML elements. Figure A.7 presents the new type inference rule IELEMENT, in addition to two ancillary judgements.

Given an XML element, rule IELEMENT proceeds by inferring the type of, and converting to a run-time term, each element item. This initial conversion is handled by the ancillary, and trivial, $\vdash_{ei}$ judgement. The problem is then to combine a sequence of typed run-time terms $U_1 : v_1, \ldots, U_n : v_n$ into a single run-time term T representing the XML element in native syntax. This combination is done by first expanding the saturated tag name $A\ \tau_1 \ldots \tau_m$ to the normalised type $norm(\tau'\ \tau_1 \ldots \tau_m)$, where τ' is the body of the newtype A. Since each τ_i is ground, so is $norm(\tau'\ \tau_1 \ldots \tau_m)$, hence this type may be used to construct an augmented Glushkov automaton. The automaton is then simulated on the sequence $v_1 \ldots v_n$. If it reaches an accepting state, the desired T will be left on its stack.

The ancillary judgement

$$\theta \mid (C \mid B) \triangleright (D \mid B') \mid st \mid p \vdash_{\mathcal{T}\{last;follow\}} U_1 : \tau_1, \ldots, U_n : \tau_n \hookrightarrow T$$

performs this simulation. It takes as input the sequence $U_1 : \tau_1, \ldots, U_n : \tau_n$, the current position p, the current stack st, the current constraint C, and a coercion, B, accumulated so far for C.

Rule EILAST checks if the empty sequence is acceptable. If so, any final actions are performed to yield a singleton stack containing T.

$$\boxed{\theta \mid C \mid \Gamma \vdash t : \tau \hookrightarrow T}$$

$$\frac{\begin{array}{c}(\texttt{newtype}\ \{\texttt{opaque}\}^{opt}\ A = \tau') \in tdecls \\ A : \kappa_1\ \texttt{->}\ \ldots\ \texttt{->}\ \kappa_m\ \texttt{->}\ \texttt{Type} \in \Delta_{init} \\ \forall i\ .\ \Delta_{init} \vdash \tau_i : \kappa_i \\ \theta_1 \mid C_1 \mid \Gamma \vdash_{ei} ei_1 : \upsilon_1 \hookrightarrow U_1 \\ \theta_2 \mid C_2 \mid \theta_1\ \Gamma \vdash_{ei} ei_2 : \upsilon_2 \hookrightarrow U_2 \\ \vdots \\ \theta_n \mid C_n \mid \theta_{n-1} \circ \ldots \circ \theta_1\ \Gamma \vdash_{ei} ei_n : \upsilon_n \hookrightarrow U_n \\ \{last\,; follow\} = \mathcal{R}\ \mathcal{G}[\![\bullet[norm(\tau'\ \tau_1 \ldots \tau_m)]]\!] \\ \theta' = \theta_n \circ \ldots \circ \theta_1 \quad C' = C_1 +\!\!+ \ldots +\!\!+ C_n \\ \theta'' \mid (C' \mid \cdot) \triangleright (D \mid B) \mid \cdot \mid \textbf{start} \vdash_{\mathcal{T}\{last;follow\}} U_1 : \theta'\ \upsilon_1, \ldots, U_n : \theta'\ \upsilon_n \hookrightarrow T\end{array}}{(\theta'' \circ \theta')_{|fv(\Gamma)} \mid D \mid \Gamma \vdash \texttt{<}A\ \tau_1 \ldots \tau_m\texttt{>}\ ei_1 \ldots ei_n\ \texttt{</}A\texttt{>} : A\ \tau_1 \ldots \tau_m \hookrightarrow \mathsf{letw}\ B\ \mathsf{in}\ T}\ \text{IELEMENT}$$

$$\boxed{\theta \mid C \mid \Gamma \vdash_{ei} ei : \tau \hookrightarrow T}$$

$$\frac{\theta \mid C \mid \Gamma \vdash str : \tau \hookrightarrow T}{\theta \mid C \mid \Gamma \vdash_{ei} str : \tau \hookrightarrow T}\ \text{EISTR} \qquad \frac{\theta \mid C \mid \Gamma \vdash e : \tau \hookrightarrow T}{\theta \mid C \mid \Gamma \vdash_{ei} e : \tau \hookrightarrow T}\ \text{EIELEM}$$

$$\frac{\theta \mid C \mid \Gamma \vdash t : \tau \hookrightarrow T}{\theta \mid C \mid \Gamma \vdash_{ei} \texttt{<<}\ t\ \texttt{>>} : \tau \hookrightarrow T}\ \text{EITERM}$$

$$\boxed{\theta \mid (C \mid B) \triangleright (D \mid B') \mid st \mid p \vdash_{\mathcal{T}\{last;follow\}} U_1 : \tau_1, \ldots, U_n : \tau_n \hookrightarrow T}$$

$$\frac{p(acts) \in last \quad \mathcal{S}(st \mid acts) = T}{\textbf{Id} \mid (C \mid B) \triangleright (C \mid B) \mid st \mid p \vdash_{\mathcal{T}\{last;follow\}} \cdot \hookrightarrow T}\ \text{EILAST}$$

$$\frac{\begin{array}{c}P[\upsilon](acts) \in follow\ p \quad cmp_{\emptyset}(\upsilon, \tau) \in \{\textbf{eq}, \textbf{unk}\} \\ \forall P'[\upsilon'](acts) \in first\ .\ cmp_{\emptyset}(\upsilon', \tau) \in \{\textbf{eq}, \textbf{unk}\} \Longrightarrow P'[\upsilon'] = P[\upsilon] \\ \theta \mid (C +\!\!+ \upsilon\ \textbf{\textit{eq}}\ \tau \mid B) \triangleright (D \mid B') \mid \mathcal{S}(st \mid acts) +\!\!+ U \mid P[\upsilon] \vdash_{\mathcal{T}\{last;follow\}} rest \hookrightarrow T\end{array}}{\theta \mid (C \mid B) \triangleright (D \mid B') \mid st \mid p \vdash_{\mathcal{T}\{last;follow\}} U : \tau, rest \hookrightarrow T}\ \text{EIFOL}$$

$$\frac{\begin{array}{c}\langle fv(C) \mid C \rangle \triangleright^{*} \langle \theta_1 \mid C_1 \mid B' \rangle \\ \theta_2 \mid (C_1 \mid B' +\!\!+ B) \triangleright (D \mid B'') \mid st \mid p \vdash_{\mathcal{T}\{last;follow\}} \theta_1\ rest \hookrightarrow T\end{array}}{\theta_2 \circ \theta_1 \mid (C \mid B) \triangleright (D \mid B'') \mid st \mid p \vdash_{\mathcal{T}\{last;follow\}} rest \hookrightarrow T}\ \text{EISIMP}$$

Figure A.7: Extensions to λ^{TIR} type inference rules to recognise and convert XML elements to λ^{TIR} run-time terms

Rule EIFOL attempts to make a transition based on the current type τ. The rule succeeds if there is exactly one possible follow position with a type unifiable with τ. If there are no such follow positions, the sequence is ill-typed. If there is more than one follow position, the programmer must supply some type annotations, or reorder the sub-terms of the program, so that the transition may be uniquely determined. (A possible refinement of this rule is to allow speculative choices and backtracking.) If all is well, the appropriate equality constraint is added to the current constraint context, the transition actions are performed, the next run-time term is pushed onto the stack, and the rule proceeds recursively.

Rule EISIMP is the analogue of rule ISIMP, and allows constraints to be simplified whilst in the middle of recognising an XML element. This simplification step is vital to allow transitions made for earlier element items to guide the transitions for later element items.

Appendix B

Proofs for Chapter 4

B.1 Type Order

Lemma B.1 Given $\Delta \vdash \tau/\tau'/\upsilon/\upsilon' : \mathsf{Type}$, and $\Delta \vdash \theta\ \mathsf{subst}$, then:

(i) If $cmp_O(\tau,\tau') = \mathbf{eq}$ and $cmp^t_O(\tau,\upsilon) = \mathbf{lt}$ and $cmp^t_O(\theta\ \upsilon, \theta\ \tau) \in \{\mathbf{lt},\mathbf{eq}\}$ then $cmp_O(\upsilon,\tau') = \mathbf{unk}$ and $cmp_O(\theta\ \upsilon, \theta\ \tau') \in \{\mathbf{unk},\mathbf{lt}\}$.

(ii) If $cmp_O(\tau,\tau') = x \in \{\mathbf{lt},\mathbf{gt}\}$ and $cmp^t_O(\tau,\upsilon) = \mathbf{lt}$ and $cmp^t_O(\theta\ \upsilon,\theta\ \tau) \in \{\mathbf{lt},\mathbf{gt}\}$ then $cmp_O(\upsilon,\tau') = x$ (and thus $cmp_O(\theta\ \upsilon,\theta\ \tau') = x$).

(iii) If $cmp_O(\tau,\tau') = \mathbf{eq}$ and $cmp^t_O(\tau,\upsilon) = \mathbf{lt}$ and $cmp^t_O(\tau',\upsilon') = \mathbf{lt}$ and $cmp^t_O(\theta\ \upsilon,\theta\ \tau) \in \{\mathbf{lt},\mathbf{eq}\}$ and $cmp^t_O(\theta\ \upsilon',\theta\ \tau') \in \{\mathbf{lt},\mathbf{eq}\}$ then ($cmp_O(\upsilon,\upsilon') = \mathbf{unk}$ and $cmp_O(\theta\ \upsilon,\theta\ \upsilon') \in \{\mathbf{lt},\mathbf{eq},\mathbf{gt}\}$), or $cmp_O(\upsilon,\upsilon') = cmp_O(\theta\ \upsilon,\theta\ \upsilon')$.

(iv) If $cmp_O(\tau,\tau') = x \in \{\mathbf{lt},\mathbf{gt}\}$ and $cmp^t_O(\tau,\upsilon) = \mathbf{lt}$ and $cmp^t_O(\tau',\upsilon') = \mathbf{lt}$ and $cmp^t_O(\theta\ \upsilon,\theta\ \tau) \in \{\mathbf{lt},\mathbf{eq}\}$ and $cmp^t_O(\theta\ \upsilon',\theta\ \tau') \in \{\mathbf{lt},\mathbf{eq}\}$ then $cmp_O(\upsilon,\upsilon') = x$ (and thus $cmp_O(\theta\ \upsilon,\theta\ \upsilon') = x$).

Proof

(i) We have

$$preorder^p_O(\tau) = r +\!\!+ [a] +\!\!+ r'$$
$$preorder^p_O(\upsilon) = r +\!\!+ [y] +\!\!+ r''$$
$$preorder^p_O(\tau') = r +\!\!+ [a] +\!\!+ r'$$

case $y = b$ for $a <^{\mathsf{a}} b$. Thus $cmp_O(\upsilon,\tau') = \mathbf{unk}$. If $\theta\ a = d$, $\theta\ b = c$ for $c \leq^{\mathsf{a}} d$, then $cmp_O(\theta\ \upsilon,\theta\ \tau') = \mathbf{unk}$ as required. If $\theta\ a = G$, $\theta\ b = F$ for $F \leq^F G$, then $cmp_O(\theta\ \upsilon,\theta\ \tau') = \mathbf{lt}$ as required.

case $y = F$. Thus $cmp_O(\upsilon,\tau') = \mathbf{unk}$. Then $\theta\ a = G$ for $F \leq^F G$, and $cmp_O(\theta\ \upsilon,\theta\ tau') = \mathbf{lt}$ as required.

(ii) Let $x = \mathbf{lt}$. We have

$$preorder^{p}_{O}(\tau) = r ++ [F'] ++ r' ++ [a] ++ r''$$
$$preorder^{p}_{O}(\upsilon) = r ++ [F'] ++ r' ++ [y] ++ r'''$$
$$preorder^{p}_{O}(\tau') = r ++ [G'] ++ r''''$$

where $F' <^{F} G'$. Then $y = b$ for $a <^{\mathtt{a}} b$, or $y = F$. Then, regardless of θ, $cmp_O(\upsilon, \tau') = cmp_O(\theta\ \upsilon, \theta\ \tau') = \mathbf{lt}$ as required.

The case for $x = \mathbf{gt}$ is similar.

(iii) We have

$$preorder^{p}_{O}(\tau) = r ++ [a] ++ r'$$
$$preorder^{p}_{O}(\upsilon) = r ++ [y] ++ r''$$
$$preorder^{p}_{O}(\tau') = r ++ [a] ++ r'$$
$$preorder^{p}_{O}(\upsilon') = r ++ [z] ++ r'''$$

case $y = b$, $z = c$ for $a <^{\mathtt{a}} b$ and $a <^{\mathtt{a}} c$. Then if $\theta\ a = f$, $\theta\ b = e$, $\theta\ c = d$ for $d \leq^{\mathtt{a}} f$ and $e \leq^{\mathtt{a}} f$, then $cmp_O(\upsilon, \upsilon') = cmp_O(\theta\ \upsilon, \theta\ \upsilon') = \mathbf{unk}$ as required. If $\theta\ a = H$, $\theta\ b = G$, $\theta\ c = F$ for $F \leq^{F} H$ and $G \leq^{F} H$ then $cmp_O(\upsilon, \upsilon') = \mathbf{unk}$ and $cmp_O(\theta\ \upsilon, \theta\ \upsilon') \in \{\mathbf{lt}, \mathbf{eq}, \mathbf{gt}\}$, depending on relation between F and G, as required.

case $y = F$, $z = G$: Then $\theta\ a = H$ for $F \leq^{F} H$ and $G \leq^{F} H$. Thus $cmp_O(\upsilon, \upsilon') = cmp_O(\theta\ \upsilon, \theta\ \upsilon') \in \{\mathbf{lt}, \mathbf{eq}, \mathbf{gt}\}$, depending on relation between F and G, as required.

(iv) Let $x = \mathbf{lt}$. Then we have

$$preorder^{p}_{O}(\tau) = r ++ [F'] ++ r' ++ [a] ++ r'''_1$$
$$preorder^{p}_{O}(\upsilon) = r ++ [F'] ++ r' ++ [y] ++ r'''_2$$
$$preorder^{p}_{O}(\tau') = r ++ [G'] ++ r'' ++ [c] ++ r'''_3$$
$$preorder^{p}_{O}(\upsilon') = r ++ [G'] ++ r'' ++ [z] ++ r'''_4$$

where $F' <^{F} G'$, and $y = b$ for $a <^{\mathtt{a}} b$ or $y = F$, and $z = d$ for $c <^{\mathtt{a}} d$, or $z = G$. In all four cases, regardless of θ, $cmp_O(\upsilon, \upsilon') = cmp_O(\theta\ \upsilon, \theta\ \upsilon') = \mathbf{lt}$ as required.

The case for $x = \mathbf{gt}$ is similar.

□

Lemma B.2 Let $\Delta \vdash \tau, \upsilon, \tau' : \mathtt{Type}$ s.t. $cmp^{t}_{O}(\tau, \tau') \in \{\mathbf{lt}, \mathbf{eq}\}$ and $cmp^{t}_{O}(\tau', \upsilon) \in \{\mathbf{lt}, \mathbf{eq}\}$. Then

(i) If $cmp_O(\tau, \upsilon) = \mathbf{eq}$ then $cmp_O(\tau, \tau') = \mathbf{eq}$.

(ii) If $cmp_O(\tau, \upsilon) = \mathbf{lt}$ then $cmp_O(\tau, \tau') = \mathbf{lt}$.

Proof Straigtforward reasoning with $preorder^{p}_{O}(\tau)$, $preorder^{p}_{O}(\upsilon)$ and $preorder^{p}_{O}(\tau')$. □

Lemma B.3 Let $\kappa \in \{\texttt{Type}, \texttt{Row}\}$ and $\Delta \vdash \tau/\upsilon'/\upsilon : \kappa$ and $\Delta \vdash \theta$ subst. If $cmp_O(\tau, \upsilon) = x$ for $x \neq \textbf{unk}$ then $cmp_O(\theta\ \tau, \theta\ \upsilon) = x$.

Proof The result is immediate from the definition of $lexcmp^{p}$ when τ and υ are types. Consider the case for rows, and assume:

$$cmp_O((\#)_m\ \overline{\tau}\ l,\ (\#)_n\ \overline{\upsilon}\ l') = x \neq \textbf{unk}\ .$$

By inspection of $preorder^{p}_{O}$ and $lexcmp^{p}$, $l = l'$. If $m \neq n$ then $\overline{\tau}$ and $\overline{\upsilon}$ do not affect the result and we are done. Now assume $m = n$. Then

$$lexcmp^{p}\begin{pmatrix} preorder^{p}_{O}(\tau_{\pi\, 1}) + \!\!+ \ldots + \!\!+ preorder^{p}_{O}(\tau_{\pi\, n}), \\ preorder^{p}_{O}(\upsilon_{\pi'\, 1}) + \!\!+ \ldots + \!\!+ preorder^{p}_{O}(\upsilon_{\pi'\, n}) \end{pmatrix} = x \neq \textbf{unk}$$

where π and π' are the sorting permutations under cmp^{t}_{O}. We need to show:

$$lexcmp^{p}\begin{pmatrix} preorder^{p}_{O}(\tau'_{\pi''\, 1}) + \!\!+ \ldots + \!\!+ preorder^{p}_{O}(\tau'_{\pi''\, (n+n')}), \\ preorder^{p}_{O}(\upsilon'_{\pi'''\, 1}) + \!\!+ \ldots + \!\!+ preorder^{p}_{O}(\upsilon'_{\pi'''\, (n+n')}) \end{pmatrix} = x$$

where

$$\begin{aligned} \theta\ l &= (\#)_{n'}\ \overline{\tau''}\ l'' \\ \overline{\tau'} &= (\theta\ \overline{\tau}) + \!\!+ \overline{\tau''} \\ \overline{\upsilon'} &= (\theta\ \overline{\upsilon}) + \!\!+ \overline{\tau''} \end{aligned}$$

and where π'' and π''' are the obvious sorting permutations under cmp^{t}_{O}.

There are two possibilites:

(i) It is possible that θ may "flip" the relative ordering of two types in $\overline{\tau}$, $\overline{\upsilon}$, or both. To be more precise, given $i \neq j$, it is possible for $\pi^{-1}\ i < \pi^{-1}\ j$, but $\pi''^{-1}\ i > \pi''^{-1}\ j$ (and similarly for π' and π''').

However, Lemma B.1 shows that in all such cases the result of $lexcmp^{p}$ against the reordered types remains unchanged.

(ii) It is also possible that a type in $\overline{\tau''}$ may be inserted into $\overline{\tau}$ and $\overline{\upsilon}$ so as to be placed before a type in $\overline{\tau}$ and after a type in $\overline{\upsilon}$ which were previously matched against by $lexcmp^{p}$ (or vice-versa).

However, Lemma B.2 shows such insertions cannot change the result of $lexcmp^{p}$.

□

The following lemma is required in Section 4.4.

Lemma B.4 Let, for all (finite) i, $\Delta \vdash \tau_i/\upsilon_i : \kappa_i$ and $\kappa_i \in \{\texttt{Type}, \texttt{Row}\}$ and $cmp_O(\tau_i, \upsilon_i) = \textbf{unk}$. Then there exists a $\vdash \theta : \Delta \to \Delta_{init}$ s.t. for all i, $cmp_O(\theta\ \tau_i, \theta\ \upsilon_i) \in \{\textbf{lt}, \textbf{gt}\}$.

Proof Let θ be the substitution $\overline{a \mapsto \tau}$, where for each $(a_i : \kappa_i) \in \Delta$, A_i is a fresh newtype declared as

$$\texttt{newtype } A_i \texttt{ = Int}$$

and τ_i is the type

$$\tau_i = \begin{cases} A_i, & \text{if } \kappa_i = \texttt{Type} \\ A_i \texttt{ \# Empty}, & \text{if } \kappa_i = \texttt{Row} \end{cases}$$

Then the result follows from inspection of cmp_O. □

Note that $cmp_O(\tau, v) = \mathbf{unk}$ *does not* imply there exists a θ s.t. $cmp_O(\tau, v) = \mathbf{eq}$. For example, notice $cmp_O(a, a \texttt{ -> } a) = \mathbf{unk}$, but $cmp_O(\theta\ a, \theta\ a \texttt{ -> } \theta\ a) = \mathbf{eq}$ implies $\theta\ a$ is an infinite type, which is not allowed. This will be important in Chapter 5.

B.2 Unification

Lemma B.5 If $\Delta \vdash \theta$ subst and $\Delta \vdash C$ constraint and $eqs(C) = C$ and $\theta' \in mgus_O(\theta \vdash C)$ then there exists a Δ' s.t. $\Delta \mathbin{+\!\!+} \Delta' \vdash \theta'$ subst.

Proof By inspection. The rule for unification of rows may introduce a fresh variable, which should appear within Δ' with kind `Row`. □

Lemma B.6 (Soundness of Unification) If $\forall i\ .\ \theta\ \tau_i = \tau_i \wedge \theta\ v_i = v_i$ then $\theta' \in mgus_O(\theta \vdash \overline{\tau\ eq\ v})$ implies $\exists \theta''\ .\ \theta' = \theta'' \circ \theta$ and $\forall i\ .\ cmp_O(\theta''\ \tau_i, \theta''\ v_i) = \mathbf{eq}$.

Proof Let $size(\tau)$ denote the size of a type τ, which is defined as:

$$\begin{aligned} size(a) &= 1 \\ size(F\ \overline{\tau}) &= 1 + \textstyle\sum_i size(\tau_i) \\ size(\texttt{(\#)}_n\ \overline{\tau}\ l) &= n + size(l) + \textstyle\sum_i size(\tau_i) \end{aligned}$$

We extend $size$ to equality primitive constraints by

$$size(\tau\ eq\ v) = size(\tau) + size(v)$$

and to sets of primitive equality constraints as the sum of each member constraint size.

Then the theorem follows by an easy induction on $size(\overline{\tau\ eq\ v})$ using Lemma 4.2 (iv). □

Lemma B.7 (Completeness of Unification) If $\forall i\ .\ cmp_O(\theta\ \tau_i, \theta\ v_i) = \mathbf{eq}$ then $\exists \theta' \in mgus_O(\mathbf{Id} \vdash \overline{\tau\ eq\ v})$ and θ'' s.t. $\theta'' \circ \theta'|_{dom(\theta)} \equiv_O \theta$.

Proof W.l.o.g. we assume only a single primitive equality constraint $\tau\ eq\ v$. (Multiple constraints may always be collapsed to one by constructing a suitable pair of function types.)

Then we proceed by pairwise induction on the structure of (τ, v):

case (a, a): Immediate.
case $(a, v), a \notin fv_O(v)$: Then $mgus_O(\mathbf{Id} \vdash a\ eq\ v) = \{[a \mapsto v]\}$. Let $\theta'' = \theta_{\backslash a}$. Then since $\theta\ a = \tau$ s.t. $cmp_O(\tau, \theta\ v) = \mathbf{eq}$, we have $\theta'' \circ [a \mapsto v] \equiv_O \theta$.

case $(\tau, a), a \notin fv_O(v)$: As above.
case $(a, v), a \in fv_O(v)$: Then $cmp_O(\theta\ a, \theta\ v) = \mathbf{eq}$ implies θ is not idempotent.
case $(F\ \overline{\tau}, F\ \overline{v})$: W.l.o.g. assume F has arity 2.
By definition $cmp_O(\theta\ (F\ \tau_1\ \tau_2), \theta\ (F\ v_1\ v_2)) = \mathbf{eq}$ implies

$$cmp_O(\theta\ \tau_1, \theta\ v_1) = \mathbf{eq} \tag{a}$$
$$cmp_O(\theta\ \tau_2, \theta\ v_2) = \mathbf{eq} \tag{b}$$

By I.H. on (a) there exists a $\theta'_1 \in mgus_O(\mathbf{Id} \vdash \tau_1\ eq v_1)$ and a θ''_1 s.t. $(\theta''_1 \circ \theta'_1)_{\restriction dom(\theta)} \equiv_O \theta$.
Then by (b)

$$cmp_O(\theta''_1\ \theta'_1\ \tau_2, \theta''_1\ \theta'_1\ v_2) = \mathbf{eq} \tag{c}$$

By I.H. on (c) there exists a $\theta'_2 \in mgus_O(\mathbf{Id} \vdash \theta'_1\ \tau_2\ eq\ \theta'_1\ v_2)$ and a θ''_2 s.t. $(\theta''_2 \circ \theta'_2)_{\restriction dom(\theta''_1)} \equiv_O \theta''_1$.
Let $\theta'' = \theta''_2$. Then

$$\begin{aligned}
\theta'' \circ \theta'_2 \circ \theta'_{1\restriction dom(\theta)} &\equiv_\emptyset \theta''_2 \circ \theta'_{2\restriction dom(\theta''_1)} \circ \theta'_{1\restriction dom(\theta)} \\
&\equiv_\emptyset (\theta''_1 \circ \theta'_1)_{\restriction dom(\theta)} \\
&\equiv_\emptyset \theta
\end{aligned}$$

By definition

$$\begin{aligned}
&mgus_O(\mathbf{Id} \vdash F\ \tau_1\ \tau_2\ eq\ F\ v_1\ v_2) \\
&= mgus_O(\mathbf{Id} \vdash \tau_1\ eq\ v_1, \tau_2\ eq\ v_2) \\
&= \left\{ \theta'_2 \circ \theta'_1 \;\middle|\; \begin{array}{l} \theta'_1 \in mgus_O(\mathbf{Id} \vdash \tau_1\ eq\ v_1), \\ \theta'_2 \in mgus_O(\mathbf{Id} \vdash \theta'_1\ \tau_2\ eq\ \theta'_1\ v_2) \end{array} \right\}
\end{aligned}$$

Then the result follows since all such θ'_1 and θ'_2 are collected.
case $(F\ \overline{\tau}, G\ \overline{v}), F \neq G$: Then by definition $cmp(\theta\ (F\ \overline{\tau}), \theta\ (G\ \overline{v})) \in \{\mathbf{lt}, \mathbf{gt}\}$.
case $((\#)_m\ \overline{\tau}\ l, (\#)_n\ \overline{v}\ l')$: Notice if $m = 0$ or $n = 0$ then an earlier case will apply.
W.l.o.g. assume

$$\theta\ ((\#)_m\ \overline{\tau}\ l) = (\#)_{m+m'}\ \overline{\tau'}\ l''$$
$$\theta\ ((\#)_n\ \overline{v}\ l') = (\#)_{n+n'}\ \overline{v'}\ l'''$$

where

$$\tau'_k = \begin{cases} \theta'\ \tau_k, & \textbf{if } 1 \le k \le m \\ \tau''_{k-m}, & \textbf{if } m < k \le m + m' \end{cases}$$
$$v'_k = \begin{cases} \theta'\ v_i, & \textbf{if } 1 \le k \le n \\ v''_{k-n}, & \textbf{if } n < k \le n + n' \end{cases}$$

Then since $cmp_O(\theta\ ((\#)_m\ \overline{\tau}\ l), \theta\ ((\#)_n\ \overline{v}\ l')) = \mathbf{eq}$, by Lemma 4.2 (iv)

$$\begin{gathered} m + m' = n + n' \\ l'' = l''' \\ \exists \pi : m + m' \to m + m'\ .\ \forall i\ .\ cmp_O(\tau'_i, v'_{\pi\ i}) = \mathbf{eq} \end{gathered}$$

Consider $j = \pi\ 1$:

case $1 \leq j \leq n$: Thus we have

$$cmp_O(\theta\ \tau_1, \theta\ v_j) = \mathbf{eq} \tag{a}$$

$$cmp_O((\#)_{m+m'-1}\ \overline{\tau'}_{\backslash 1}\ l'', (\#)_{n+n'-1}\ \overline{v'}_{\backslash j}\ l'') = \mathbf{eq} \tag{b}$$

By I.H. on (a) there exists a $\theta'_1 \in mgus_O(\mathbf{Id} \vdash \tau_1, v_j)$ and a θ''_1 s.t. $\theta''_1 \circ \theta'_{1\restriction dom(\theta)} \equiv_O \theta$.

Then by (b)

$$cmp_O((\#)_{m+m'-1}\ \overline{\tau'''}_{\backslash 1}\ l'', (\#)_{n+n'-1}\ \overline{v'''}_{\backslash j}\ l'') = \mathbf{eq} \tag{c}$$

where $\overline{\tau'''}$ and $\overline{v'''}$ are defined as for $\overline{\tau'}$ and $\overline{v'}$, but using $\theta''_1 \circ \theta'_1$ instead of θ.

Then by I.H. on (c), there exist a $\theta'_2 \in mgus_O(\mathbf{Id} \vdash \theta''_1\ \theta'_1\ ((\#)_{m-1}\ \overline{\tau}_{\backslash 1}\ l)\ \mathit{eq}\ \theta''_1\ \theta'_1\ ((\#)_{n-1}\ \overline{v}_{\backslash j}\ l'))$ and a θ''_2 s.t. $\theta''_2 \circ \theta'_{2\restriction dom(\theta''_1)} \equiv_O \theta''_1$.

Let $\theta'' = \theta''_2$. Then

$$\begin{aligned} \theta'' \circ \theta'_2 \circ \theta'_{1\restriction dom(\theta)} &\equiv_O \theta''_2 \circ \theta'_{2\restriction dom(\theta''_1)} \circ \theta'_{1\restriction dom(\theta)} \\ &\equiv_O \theta''_1 \circ \theta'_{1\restriction dom(\theta)} \\ &\equiv_O \theta \end{aligned}$$

By definition of S_j

$$\begin{aligned} & mgus_O(\mathbf{Id} \vdash \tau_1\ \mathit{eq}\ v_j, (\#)_{m-1}\ \overline{\tau}_{\backslash 1}\ l\ \mathit{eq}\ (\#)_{n-1}\ \overline{v}_{\backslash j}\ l') \\ = & \left\{ \theta'_2 \circ \theta'_1 \;\middle|\; \begin{array}{l} \theta'_1 \in mgus_O(\mathbf{Id} \vdash \tau_1\ \mathit{eq}\ v_j), \\ \theta'_2 \in mgus_O(\mathbf{Id} \vdash \theta'_1\ ((\#)_{m-1}\ \overline{\tau}_{\backslash 1}\ l)\ \mathit{eq}\ \theta'_1\ ((\#)_{n-1}\ \overline{v}_{\backslash j}\ l')) \end{array} \right\} \end{aligned}$$

Then the result follows since all such θ'_1 and θ'_2 are collected, and $mgus_O(\mathbf{Id} \vdash (\#)_{m-1}\ \overline{\tau}\ \mathit{eq}\ (\#)_{n-1}\ \overline{v}\ l')$ includes S_j.

case $n + 1 \leq j \leq n + n'$: Thus we have

$$cmp_O(\theta\ \tau_1, v''_{j-n}) = \mathbf{eq} \tag{d}$$

$$cmp_O((\#)_{m+m'-1}\ \overline{\tau'}_{\backslash 1}\ l'', (\#)_{n+n'-1}\ \overline{v'}_{\backslash j}\ l'') = \mathbf{eq} \tag{e}$$

and $l' = a$ for some a s.t. $\theta\ a = (\#)_{n'}\ \overline{v''}\ l''$.

Furthermore, if $a \in fv_O(\tau_1)$ then by (d) $a \in fv_O(v''_{j-n})$, and thus θ would not be idempotent.

Let $\theta'_1 = \theta_{\backslash a} \circ [b \mapsto (\#)_{n'-1}\ \overline{v''}_{\backslash j-n}\ l'']$. Then since b fresh, by (d) and Lemma 4.2 (iv)

$$(\theta'_1 \circ [a \mapsto \tau_1\ \#\ b])_{\backslash b} \equiv_O \theta \tag{f}$$

Thus

$$cmp_O(({\#})_{m+m'-1}\ \overline{\tau'}_{\backslash 1}\ l'', \theta'_1 \circ [a \mapsto \tau_1 \ \# \ b]\ (({\#})_{m-1}\ \overline{\tau}_{\backslash 1}\ l)) = \mathbf{eq}$$
$$cmp_O(({\#})_{n+n'-1}\ \overline{v'}_{\backslash j}\ l'', \theta'_1 \circ [a \mapsto \tau_1 \ \# \ b]\ (({\#})_{n}\ \overline{v}\ b)) = \mathbf{eq}$$

which is to say

$$cmp_O(\theta'_1 \circ [a \mapsto \tau_1 \ \# \ b]\ (({\#})_{m-1}\ \overline{\tau}_{\backslash 1}\ l), \theta'_1 \circ [a \mapsto \tau_1 \ \# \ b]\ (({\#})_{n}\ \overline{v}\ b)) = \mathbf{eq} \qquad \text{(g)}$$

Then by I.H. on (g), there exists a $\theta'_2 \in mgus_O(\mathbf{Id} \vdash (({\#})_{m-1}\ \overline{\tau}_{\backslash 1}\ l\ \boldsymbol{eq}\ ({\#})_n\ \overline{v}\ b)[a \mapsto \tau \ \# \ b])$ and θ''_2 s.t. $(\theta''_2 \circ \theta'_2)_{\restriction dom(\theta'_1)} \equiv_O \theta'_1$.
Then by (f)

$$\begin{aligned}(\theta''_2 \circ \theta'_2 \circ [a \mapsto \tau_1 \ \# \ b])_{\restriction dom(\theta)} &\equiv_O ((\theta''_2 \circ \theta'_2)_{\restriction dom(\theta) \cup \{b\}} \circ [a \mapsto \tau_1 \ \# \ b])_{\backslash b} \\ &\equiv_O ((\theta''_2 \circ \theta'_2)_{\restriction dom(\theta'_1)} \circ [a \mapsto \tau_1 \ \# \ b])_{\backslash b} \\ &\equiv_O (\theta'_1 \circ [a \mapsto \tau_1 \ \# \ b])_{\backslash b} \\ &\equiv_O \theta\end{aligned}$$

The result follows from the definition of S' and that $mgus_O(\mathbf{Id} \vdash ({\#})_{m-1}\ \overline{\tau}\ \boldsymbol{eq}\ ({\#})_{n-1}\ \overline{v}\ l')$ includes S'.

□

For the most part we shall supress the projection required in the above lemma.

Corollary B.8 (Most General Unifiers) For all $\theta' \in mgus_O(\mathbf{Id} \mid \overline{\theta\ \tau\ \boldsymbol{eq}\ \theta\ v})$ there exists $\theta'' \in mgus_O(\mathbf{Id} \mid \overline{\tau\ \boldsymbol{eq}\ v})$ and θ''' s.t. $\theta' \circ \theta \equiv_O \theta''' \circ \theta''$.

Proof If $\theta' \in mgus_O(\mathbf{Id} \mid \overline{\theta\ \tau\ \boldsymbol{eq}\ \theta\ v})$ then by Lemma B.6 $\forall i\ .\ cmp_O(\theta'\ \theta\ \tau_i, \theta'\ \theta\ v_i) = \mathbf{eq}$. Then the result follows from Lemma B.7. □

B.3 Entailment

Lemma B.9 Let $\Delta \vdash C$ constraint s.t. $C = eqs(C)$, $\Delta \vdash \tau\ \boldsymbol{ins}\ \rho$ constraint and $\vdash \theta : \Delta \to \Delta_{init}$. Then if (a) $\eta \models \theta\ C$ and (b) $C \vdash^m \tau\ \boldsymbol{ins}\ \rho \hookrightarrow W$ then $[\![W]\!]_\eta \in [\![\theta\ \tau\ \boldsymbol{ins}\ \theta\ \rho]\!]$.

Proof By induction on derivation of (b):

case MEMPTY: Let (b) be

$$C \vdash^m \tau\ \boldsymbol{ins}\,\mathsf{Empty} \hookrightarrow \mathsf{One}$$

Since $sortingPerms(\theta\ \tau) = \{id\}$ we have

$$[\![\mathsf{One}]\!]_\eta = \mathsf{iind} : 1 \in \{\mathsf{iind} : id^{-1}\ 1\} = [\![\theta\ \tau\ \boldsymbol{ins}\,\mathsf{Empty}]\!]$$

as required.

case MREF: Let (b) be

$$C \vdash^m \tau_1\ \boldsymbol{ins}\ \rho \hookrightarrow w$$

Then by MREF

$$cmp_{opaque}(\tau_1, \tau_1') = \mathbf{eq} \wedge cmp_{opaque}(\rho, \rho') = \mathbf{eq} \wedge (w : \tau_1' \; \boldsymbol{ins}\, \rho') \in C \tag{c}$$

W.l.o.g. let $\rho = \tau_2 \,\#\, \ldots \,\#\, \tau_n \,\#\, l$ and $\rho' = \tau_2' \,\#\, \ldots \,\#\, \tau_m' \,\#\, l'$ where $n, m > 0$.
By (c) and Lemma 4.2, $n = m$, $l = l'$ and there exists a permutation $\pi : n-1 \to n-1$ s.t. $\forall i \,.\, cmp_{opaque}(\tau_{i+1}, \tau'_{(\pi\, i)+1}) = \mathbf{eq}$. Again w.l.o.g., we may assume

$$\theta\, l = \tau_{n+1} \,\#\, \ldots \,\#\, \tau_{n+n'} \,\#\, \mathbf{Empty} = \tau'_{n+1} \,\#\, \ldots \,\#\, \tau'_{n+n'} \,\#\, \mathbf{Empty} = \theta\, l'$$

for $n' \geq 0$. By idempotency of θ, $\forall i \,.\, \theta\, \tau_{n+i} = \tau_{n+i}$.
Then define $\pi' : n + n' \to n + n'$ as

$$\pi'\, i = \begin{cases} 1, & \textbf{if } i = 1 \\ (\pi\, (i-1)) + 1, & \textbf{if } 2 \leq i \leq n \\ i, & \textbf{if } n < i \leq n + n' \end{cases}$$

Then

$$\forall i \,.\, cmp_{opaque}(\theta\, \tau_i, \theta\, \tau'_{\pi'\, i}) = \mathbf{eq} \wedge \pi'\, 1 = 1 \tag{d}$$

By (a), $[\![w]\!]_\eta \in [\![\theta\, \tau' \; \boldsymbol{ins}\, \theta\, \rho']\!]$, which implies there exists a j s.t.

$$S \neq \emptyset \wedge \pi''' \in S \Longrightarrow \pi'''^{-1}\, 1 = j \tag{e}$$

$$[\![w]\!]_\eta = \mathsf{iind} : j \tag{f}$$

where $S = sortingPerms(\theta\, \tau_1', \ldots, \theta\, \tau'_{n+n'})$.
Now let $\pi'' \in sortingPerms(\theta\, \tau_1, \ldots, \theta\, \tau_{n+n'})$. Then there exists a $\pi''' \in S$ s.t.

$$\pi'' = \pi' \circ \pi'''$$

Then by (d) and (e)

$$\pi''^{-1}\, 1 = (\pi' \circ \pi''')^{-1}\, 1 = \pi'''^{-1} \circ \pi'^{-1}\, 1 = \pi'''^{-1}\, 1 = j$$

Since this holds for every π'', by (f) $[\![w]\!]_\eta \in [\![\theta\, \tau_1 \; \boldsymbol{ins}\, \theta\, \rho]\!]$ as required.

case MCONT: Let (b) be

$$C \vdash^m \tau_1 \; \boldsymbol{ins}\, \rho \hookrightarrow W$$

where w.l.o.g. assume $\rho = \tau_2 \,\#\, \ldots \tau_{i-1} \,\#\, \tau_{i+1} \,\#\, \ldots \,\#\, \tau_n \,\#\, l$ for $i > 1$.
Then by MCONT

$$cmp_{opaque}(\tau_1, \tau_i) = \mathbf{lt} \tag{c}$$

$$C \vdash^m \tau_1 \; \boldsymbol{ins}\, \rho' \hookrightarrow W \tag{d}$$

where $\rho' = \tau_2 \,\#\, \ldots \,\#\, \tau_n \,\#\, l$.
W.l.o.g., we may assume $\theta\, l = \tau_{n+1} \,\#\, \ldots \,\#\, \tau_{n+n'} \,\#\, \mathbf{Empty}$ for $n' \geq 0$. By idempotency of θ, $\forall i' \,.\, \theta\, \tau_{n+i'} = \tau_{n+i'}$.
By I.H. on (d) $[\![W]\!]_\eta \in [\![\theta\, \tau_1 \; \boldsymbol{ins}\, \theta\, \tau_2 \,\#\, \ldots \,\#\, \theta\, \tau_{n+n'} \,\#\, l]\!]$, which implies there exists a

j s.t.

$$S \neq \emptyset \wedge \pi \in S \Longrightarrow \pi^{-1}\,1 = j \tag{e}$$
$$[\![W]\!]_\eta = \mathsf{iind} : j \tag{f}$$

where $S = \mathit{sortingPerms}(\theta\ \tau_1, \ldots, \theta\ \tau_{n+n'})$.
Let $\pi' \in \mathit{sortingPerms}(\theta\ \tau_1, \theta\ \tau_2, \ldots, \theta\ \tau_{i-1}, \theta\ \tau_{i+1}, \ldots, \theta\ \tau_{n+n'})$. Then there exists a $\pi \in S$ s.t.

$$\begin{array}{ll} \pi'\ i' = & \textbf{if } i' < k \textbf{ then} \\ & \quad \textbf{if } \pi\ i' < i \textbf{ then } \pi\ i' \textbf{ else } (\pi\ i') - 1 \\ & \textbf{else} \\ & \quad \textbf{if } \pi\ (i'+1) < i \textbf{ then } \pi\ (i'+1) \textbf{ else } (\pi\ (i'+1)) - 1 \end{array}$$

where $k = \pi^{-1}\ i$.
By (c) and stability of $\mathit{cmp}_{\mathit{opaque}}$ $\mathit{cmp}_{\mathit{opaque}}(\theta\ \tau_1, \theta\ \tau_i) = \mathbf{lt}$, and thus $j < k$.
Let $j' = \pi'^{-1}\ 1$. Then since $\pi'\ j = \pi\ j = 1$, we have $j = j'$.
Since this holds for every π', we have $[\![W]\!]_\eta \in [\![\theta\ \tau_1\ \boldsymbol{ins}\ \theta\ \rho]\!]$ as required.

case MDEC: As for case MCONT, but this time since $\mathit{cmp}_{\mathit{opaque}}(\tau_1, \tau_i) = \mathbf{gt}$, $j > k$, and thus $j > 1$. Then $\pi'\ (j-1) = \pi\ j = 1$, so $j' = j - 1$, which is to say $j = j' + 1$. Thus

$$\begin{array}{rl} [\![\mathsf{Dec}\ W]\!]_\eta = & \textbf{case } [\![W]\!]_\eta \textbf{ of } \{ \\ & \quad \mathsf{iind} : i' \to \textbf{if } i' > 1 \textbf{ then } \mathsf{iind} : i' - 1 \textbf{ else } \mathsf{iwrong} : *; \\ & \quad \textbf{otherwise} \to \mathsf{iwrong} : * \\ & \} \\ = & \mathsf{iind} : j' \\ \in & [\![\theta\ \tau_1\ \boldsymbol{ins}\ \theta\ \rho]\!] \end{array}$$

as required.

case MEXP: Let (b) be

$$C \vdash^m \tau_1\ \boldsymbol{ins}\ \rho \hookrightarrow W$$

where w.l.o.g. assume $\rho = \tau_2\ \#\ \ldots\ \#\ \tau_n\ \#\ l$.
Then by MEXP

$$\mathit{cmp}_{\mathit{opaque}}(\tau_1, \tau_i) = \mathbf{lt} \tag{c}$$
$$C \vdash^m \tau_1\ \boldsymbol{ins}\ \rho' \hookrightarrow W \tag{d}$$

where $\rho' = \tau_2\ \#\ \ldots\ \#\ \tau_{i-1}\ \#\ \tau_{i+1}\ \#\ \ldots\ \#\ \tau_n\ \#\ l$ for $i > 1$.
W.l.o.g., we may assume $\theta\ l = \tau_{n+1}\ \#\ \ldots\ \#\ \tau_{n+n'}\ \#\ \mathtt{Empty}$ for $n' \geq 0$. By idempotency of θ, $\forall i\ .\ \theta\ \tau_{n+i} = \tau_{n+i}$.
By I.H. on (d) $[\![W]\!]_\eta \in [\![\theta\ \tau_1\ \boldsymbol{ins}\ \theta\ \rho']\!]$, which implies there exists a j s.t.

$$S \neq \emptyset \wedge \pi \in S \Longrightarrow \pi^{-1}\,1 = j \tag{e}$$
$$[\![W]\!]_\eta = \mathsf{iind} : \pi^{-1}\ 1 \tag{f}$$

where $S = \mathit{sortingPerms}(\theta\ \tau_1, \theta\ \tau_2, \ldots, \theta\ \tau_{i-1}, \theta\ \tau_{i+1}, \ldots, \theta\ \tau_{n+n'})$.

Let $\pi' \in sortingPerms(\theta\ \tau_1, \ldots, \theta\ \tau_{n+n'})$. Then there exists a $\pi \in S$ s.t.

$$\begin{aligned}\pi\ i' = \ & \textbf{if } i' < k \textbf{ then} \\ & \quad \textbf{if } \pi'\ i' < i \textbf{ then } \pi'\ i' \textbf{ else } (\pi'\ i') - 1 \\ & \textbf{else} \\ & \quad \textbf{if } \pi'\ (i'+1) < i \textbf{ then } \pi'\ (i'+1) \textbf{ else } (\pi'\ (i'+1)) - 1\end{aligned}$$

where $k = \pi'^{-1}\ i$.
Let $j' = \pi'^{-1}\ 1$. By (c) and stability of cmp_{opaque}, $cmp_{opaque}(\theta\ \tau_1, \theta\ \tau_i) = \mathbf{lt}$, thus $j' < k$.
Then $\pi\ j' = \pi'\ j' = 1$, so $j = j'$.
Since this holds for every π', we have $[\![W]\!]_\eta \in [\![\theta\ \tau_1\ \textit{ins}\ \theta\ \rho]\!]$ as required.

case MINC: As for case MEXP, but this time since $cmp_{opaque}(\tau_1, \tau_i) = \mathbf{gt}$, $j' > k$.
Then $\pi\ (j'-1) = \pi'\ j' = 1$, thus $j = j' - 1$.
Thus

$$\begin{aligned}[\![\mathsf{Inc}\ W]\!]_\eta = \ & \textbf{case } [\![W]\!]_\eta \textbf{ of } \{ \\ & \quad \mathsf{iind} : i' \to \mathsf{iind} : i' + 1; \\ & \quad \textbf{otherwise} \to \mathsf{iwrong} : * \\ & \} \\ = \ & \mathsf{iint} : j' \\ \in \ & [\![\theta\ \tau_1\ \textit{ins}\ \theta\ \rho]\!]\end{aligned}$$

as required.

□

Lemma B.10 (i) If $\Delta_{init} \vdash c/d$ constraint and $c \equiv d$ then $[\![c]\!] = [\![d]\!]$.

(ii) If $\Delta_{init} \vdash C/D$ constraint and $\eta \models C$ and $C \equiv D$ then $\eta \models D$.

Proof

(i) **case** $c = \tau\ \textit{eq}\ \upsilon$. Then $d = \tau'\ \textit{eq}\ \upsilon'$ where $cmp_\emptyset(\tau, \tau') = \mathbf{eq}$ and $cmp_\emptyset(\upsilon, \upsilon') = \mathbf{eq}$, or vise versa.
If $cmp_\emptyset(\tau, \upsilon) = \mathbf{eq}$ then by Lemma 4.2 $cmp_\emptyset(\tau', \upsilon') = \mathbf{eq}$. Thus $[\![d]\!] = \{\mathsf{itrue} : *\} = [\![c]\!]$.
Otherwise if $cmp_\emptyset(\tau, \upsilon) \in \{\mathbf{lt}, \mathbf{gt}\}$ then by Lemma 4.2 $cmp_\emptyset(\tau', \upsilon') \in \{\mathbf{lt}, \mathbf{gt}\}$. Thus $[\![d]\!] = \emptyset = [\![c]\!]$.

case $c = \tau\ \textit{ins}\ (\#)_n\ \overline{\upsilon}\ \mathtt{Empty}$. Then $d = \tau'\ \textit{ins}\ (\#)_n\ \overline{\upsilon'}\ \mathtt{Empty}$ where $cmp_\emptyset(\tau, \tau) = \mathbf{eq}$ and $cmp_\emptyset((\#)_n\ \overline{\upsilon}\ \mathtt{Empty}, (\#)_n\ \overline{\upsilon'}\ \mathtt{Empty}) = \mathbf{eq}$.
If $\forall \pi \in sortingPerms(\tau, \upsilon_1, \ldots, \upsilon_n)$ we have $\pi^{-1}\ 1 = j$ for some j. Then by the same reasoning as for case MREF in Lemma B.9 $sortingPerms(\tau, \upsilon_1, \ldots, \upsilon_n) = sortingPerms(\tau', \upsilon'_1, \ldots, \upsilon'_n)$. Thus $[\![d]\!] = \{\mathsf{iind} : j\} = [\![d]\!]$.
Otherwise, there exists an i s.t. $cmp_\emptyset(\tau, \upsilon_i) = \mathbf{eq}$. Thus there exists an i' s.t. $cmp_\emptyset(\tau, \upsilon_{i'}) = \mathbf{eq}$. Thus $[\![d]\!] = \emptyset = [\![c]\!]$.

(ii) Let $(w : c) \in C$, and let $(w : d) \in D$ be the coresponding primitive constraint s.t. $c \equiv d$. Since $\eta \models C$, $\eta\ w \in [\![c]\!] = [\![d]\!]$, so $\eta \models D$.

□

Lemma B.11 Let $\Delta \vdash C/C'$ constraint and $C = inss(C)$ and $\Delta \vdash \theta$ subst. Then

(i) $satisfied(\theta\ C) \Longrightarrow satisfied(C)$

(ii) $satisfied(C) \wedge C \equiv C' \Longrightarrow satisfied(C')$

Proof

(i) Assume $satisfied(\theta\ C)$ and $\neg satisfied(C)$. Then there exists $(\tau\ \mathbf{ins}\ (\#)_n\ \overline{v}\ l) \in C$ and an i s.t. $cmp_{opaque}(\tau, v_i) = \mathbf{eq}$. But then by stability of cmp_{opaque}, $cmp_{opaque}(\theta\ \tau, \theta\ v_i) = \mathbf{eq}$, and hence $\neg satisfied(\theta\ C)$.

(ii) Similar.

□

Lemma B.12 Let $\Delta \vdash C$ constraint and $\Delta \vdash d$ constraint and (a) $C \vdash^e d \hookrightarrow W$ and $\vdash \theta : \Delta \to \Delta_{init}$ and (b) $\eta \models \theta\ C$. Then $[\![W]\!]_\eta \in [\![\theta\ d]\!]$.

Proof By case analysis on d:

case EQUALS: Let $d = \tau\ \mathbf{eq}\, v$. Then by EQUALS:

$$\forall \theta' \in saturate(C)\ .\ cmp_\emptyset(\theta'\ \tau, \theta'\ v) = \mathbf{eq}$$

and $W = \mathsf{True}$.
Then by definition of *saturate*:

$$\begin{array}{l} \forall \theta' \in mgus_\emptyset(\mathbf{Id} \vdash eqs(C))\ . \\ \quad satisfied(\theta'\ inss(C)) \Longrightarrow \\ \quad\quad cmp_\emptyset(\theta'\ \tau, \theta'\ v) = \mathbf{eq} \end{array}$$

Then by Lemma B.8:

$$\begin{array}{l} \forall \theta'' \in mgus_\emptyset(\mathbf{Id} \vdash \theta\ eqs(C))\ . \\ \quad \exists \theta' \in mgus_\emptyset(\mathbf{Id} \vdash eqs(C))\ . \\ \quad\quad \exists \theta'''\ .\ \theta'' \circ \theta \equiv_\emptyset \theta''' \circ \theta' \\ \quad\quad\quad \wedge\ satisfied(\theta'\ inss(C)) \Longrightarrow \\ \quad\quad\quad\quad cmp_\emptyset(\theta'\ \tau, \theta'\ v) = \mathbf{eq} \end{array} \qquad (c)$$

By (b)

$$(\tau'\ \mathbf{eq}\, v') \in C \Longrightarrow cmp_\emptyset(\theta\ \tau', \theta\ v') = \mathbf{eq} \qquad (d)$$

and

$$(\tau'\ \mathbf{ins}\, \rho') \in C \Longrightarrow \neg isIn(\theta\ \tau', \theta\ \rho') \qquad (e)$$

Then by (d) and Lemma B.7

$$\mathbf{Id} \in mgus_\emptyset(\mathbf{Id} \vdash \theta\ eqs(C))$$

and by (e)

$$satisfied(\theta\ inss(C)) \qquad (f)$$

Thus, by (c), taking $\theta'' = \mathbf{Id}$

$$\begin{array}{l} \exists\theta' \in \mathit{mgus}_\emptyset(\mathbf{Id} \vdash \mathit{eqs}(C)) \; . \\ \quad \exists\theta''' \; . \; \theta \equiv_\emptyset \theta''' \circ \theta' \\ \qquad \wedge\; \mathit{satisfied}(\theta' \; \mathit{inss}(C)) \Longrightarrow \\ \qquad\quad \mathit{cmp}_\emptyset(\theta' \; \tau, \theta' \; v) = \mathbf{eq} \end{array}$$

which by Lemma B.11 (i) and stability of $\mathit{cmp}_\emptyset$ implies

$$\begin{array}{l} \exists\theta' \in \mathit{mgus}_\emptyset(\mathbf{Id} \vdash \mathit{eqs}(C)) \; . \\ \quad \exists\theta''' \; . \; \theta \equiv_\emptyset \theta''' \circ \theta' \\ \qquad \wedge\; \mathit{satisfied}(\theta''' \; \theta' \; \mathit{inss}(C)) \Longrightarrow \\ \qquad\quad \mathit{cmp}_\emptyset(\theta''' \; \theta' \; \tau, \theta''' \; \theta' \; v) = \mathbf{eq} \end{array}$$

Then by (f) and Lemma B.11 (ii) $\mathit{satisfied}(\theta''' \; \theta' \; \mathit{inss}(C))$, thus $\mathit{cmp}_\emptyset(\theta \; \tau, \theta \; v) = \mathbf{eq}$ and thus

$$\mathsf{True} \in [\![\theta \; \tau \; \boldsymbol{eq} \, \theta \; v]\!]$$

as required.

case INSERT: Let $d = \tau \; \boldsymbol{ins} \, \rho$. Then by INSERT:

$$\forall\theta' \in \mathit{saturate}(C) \; . \; \theta' \; \mathit{inss}(C) \vdash^m \theta' \; \tau \; \boldsymbol{ins} \, \theta' \; \rho \hookrightarrow W$$

Then, by same argument as for case EQUALS:

$$\begin{array}{l} \exists\theta' \in \mathit{mgus}_\emptyset(\mathbf{Id} \vdash \mathit{eqs}(C)) \; . \\ \quad \exists\theta''' \; . \; \theta \equiv_\emptyset \theta''' \circ \theta' \\ \qquad \wedge\; \mathit{satisfied}(\theta''' \; \theta' \; \mathit{inss}(C)) \Longrightarrow \\ \qquad\quad \theta' \; \mathit{inss}(C) \vdash^m \theta' \; \tau \; \boldsymbol{ins} \, \theta' \; \rho \hookrightarrow W \end{array} \tag{c}$$

By Lemma B.9, if $\eta' \models \theta''' \; \theta' \; \mathit{inss}(C)$ then

$$[\![W]\!]_{\eta'} \in [\![\theta''' \; \theta' \; \tau \; \boldsymbol{ins} \, \theta''' \; \theta' \; \rho]\!]$$

Thus by (b) and Lemma B.10 (i)

$$[\![W]\!]_\eta \in [\![\theta \; \tau \; \boldsymbol{ins} \, \theta \; \rho]\!]$$

as required.

□

Lemma B.13 Let $\Delta \vdash C$ constraint and $\Delta \vdash D$ constraint and $C \vdash^e D \hookrightarrow B$. Then $C \Vdash^e D \hookrightarrow B$.

Proof Let $\vdash \theta : \Delta \longrightarrow \Delta_{init}$ and η be s.t. $\eta \models \theta \; C$. Then by rule CONJ $C \vdash^e w : d \hookrightarrow W$ for each $(w : d) \in D$, where by Lemma B.12 $[\![W]\!]_\eta \in [\![\theta \; d]\!]$. Thus $\mathit{env}(B, \eta) \models \theta \; D$. □

Lemma B.14 Let $\Delta \vdash C$ constraint and $\Delta \vdash d$ constraint and $C \vdash^e d \hookrightarrow W$ and $\vdash \theta : \Delta \rightarrow \Delta_{init}$ and $\eta \models \theta \; C$. Then

(i) If $d = \tau \; \boldsymbol{eq} \, v$ then $[\![W]\!]_\eta = \mathsf{itrue} : *$ and $\mathit{eq}^m_\emptyset(\theta \; \tau, \theta \; v)$.

(ii) If $d = \tau\ \textit{ins}\ \rho$ then $[\![W]\!]_\eta = \text{iind} : i$, and if $\theta\ \rho = (\texttt{\#})_n\ \overline{\upsilon}\ \texttt{Empty}$ then $S \neq \emptyset$ and $\forall \pi \in S\ .\ \pi^{-1}\ 1 = i$, where $S = \textit{sortingPerms}(\theta\ \tau, \upsilon_1, \ldots, \upsilon_n)$.

Proof By Lemma B.12 $[\![W]\!]_\eta \in [\![\theta\ d]\!]$.

(i) Then $[\![\text{True}]\!]_\eta \in [\![\theta\ \tau\ \textit{eq}\,\theta\ \upsilon]\!]$ and so, since θ is grounding, by Lemma 4.2 $eq^m_\emptyset(\theta\ \tau, \theta\ \upsilon)$ as required.

(ii) Then $[\![W]\!]_\eta \in [\![\theta\ \tau\ \textit{ins}\ \theta\ \rho]\!]$ where $\theta\ \rho = (\texttt{\#})_n\ \overline{\upsilon}\ \texttt{Empty}$. Let $S = \textit{sortingPerms}(\theta\ \tau, \upsilon_1, \ldots, \upsilon_n)$. Then $S \neq \emptyset$ and $\forall \pi \in S\ .\ \pi^{-1}\ 1 = i$. Thus $[\![W]\!]_\eta \in \{\text{iind} : i\}$ as required.

□

Lemma B.15 Let $\Delta \vdash C$ constraint.

(i) Let $\vdash \theta' : \Delta \to \Delta_{init}$ and η be s.t. $\eta \models \theta'\ C$. Then there exists a $\theta \in \textit{saturate}(C)$ and a θ'' s.t. $\theta' \equiv_\emptyset \theta'' \circ \theta$.

(ii) Let $\theta \in \textit{saturate}(C)$. Then there exists a $\vdash \theta' : \Delta \to \Delta_{init}$, θ'' and η s.t. $\eta \models \theta'\ C$ and $\theta' \equiv_\emptyset \theta'' \circ \theta$.

Proof

(i) Let $\vdash \theta' : \Delta' \to \Delta_{init}$ and η be s.t.

$$\eta \models \theta'\ C$$

Then by definition of $\models$ we have

$$\begin{array}{l} \forall (\tau\ \textbf{eq}\ \upsilon) \in \textit{eqs}(C)\ . \\ \quad \textit{cmp}_\emptyset(\theta'\ \tau, \theta'\ \upsilon) = \textbf{eq} \end{array}$$

and thus by Lemma B.7

$$\begin{array}{l} \exists \theta \in \textit{mgus}_\emptyset(\textbf{Id} \vdash \textit{eqs}(C))\ . \\ \quad \exists \theta''\ .\ \theta' \equiv_\emptyset \theta'' \circ \theta \end{array} \tag{a}$$

Also by definition of $\models$ we have

$$\begin{array}{l} \forall (\tau\ \textit{ins}\ \rho) \in \textit{inss}(C)\ . \\ \quad \exists \overline{\upsilon}\ . \\ \quad\quad \theta'\ \rho = (\texttt{\#})_n\ \overline{\upsilon}\ \texttt{Empty} \\ \quad\quad \wedge\ S = \textit{sortingPerms}(\theta'\ \tau, \upsilon_1, \ldots, \upsilon_n) \neq \emptyset \\ \quad\quad \wedge\ \pi_1, \pi_2 \in S \Longrightarrow \pi_1^{-1}\ 1 = \pi_2^{-1}\ 1 \end{array}$$

In the following, let τ, ρ and $\overline{\upsilon}$ be drawn from one of the insertion constraints in C. Assume that $\textit{cmp}_{\textit{opaque}}(\theta'\ \tau, \upsilon_i) = \textbf{eq}$ for some i. But then *sortingPerms* would contain at least two permutations, π_1 and π_2, differing in their ordering of $\theta'\ \tau$ and υ_i. Thus $\pi_1^{-1}\ 1 \neq \pi_2^{-1}\ 1$, which contradicts the assumption. Thus

$$\forall i\ .\ \textit{cmp}_{\textit{opaque}}(\theta'\ \tau, \upsilon_i) \in \{\textbf{lt}, \textbf{gt}\} \tag{b}$$

Now assume $isIn(\theta\ \tau, \theta\ \rho)$, where θ is as given in (a). Then if $\theta\ \rho = (\#)_m\ \overline{v'}\ l$ we have

$$\exists i\ .\ cmp_{opaque}(\theta\ \tau, v'_i) = \mathbf{eq}$$

which by transitivity and stability of cmp_{opaque} implies

$$\exists i\ .\ cmp_{opaque}(\theta''\ \theta\ \tau, \theta''\ v'_i) = \mathbf{eq}$$

where θ'' is as given in (a). But then since $m \leq n$

$$cmp_{opaque}(\theta\ \tau, v_i) = \mathbf{eq}$$

which contradicts (b). Thus we conclude $\neg isIn(\theta\ \tau, \theta\ \rho)$.

Thus by (b), above argument, and definition of $isIn$

$$\begin{array}{l} \exists \theta \in mgus_{\emptyset}(\mathbf{Id} \vdash eqs(C))\ . \\ \quad \forall (\tau\ \boldsymbol{ins}\ \rho) \in inss(C)\ .\ \neg isIn(\theta\ \tau, \theta\ \rho) \\ \wedge\ \exists \theta''\ .\ \theta' \equiv_{\emptyset} \theta'' \circ \theta \end{array}$$

which is equivalent to

$$\begin{array}{l} \exists \theta \in mgus_{\emptyset}(\mathbf{Id} \vdash eqs(C))\ . \\ \quad satisfied(\theta\ inss(C)) \\ \wedge\ \exists \theta''\ .\ \theta' \equiv_{\emptyset} \theta'' \circ \theta \end{array}$$

from which the result follows by definition of *saturate*.

(ii) Let $\theta \in saturate(C)$.

By definition of *saturate*, we have

$$\begin{array}{l} \theta \in mgus_{\emptyset}(\mathbf{Id} \vdash eqs(C))\ . \\ \quad \forall (\tau\ \boldsymbol{ins}\ \rho) \in inss(C)\ .\ \neg isIn(\theta\ \tau, \theta\ \rho) \end{array} \tag{c}$$

which is to say, for each $(\tau\ \boldsymbol{ins}\ \rho) \in inss(C)$, if $\theta\ \rho = (\#)_m\ \overline{v}\ l$ then

$$\forall i\ .\ cmp_{opaque}(\theta\ \tau, v_i) \neq \mathbf{eq} \tag{d}$$

We seek a θ'' and η s.t. $(\theta'' \circ \theta) : \Delta \rightarrow \Delta_{init}$ and $\eta \models \theta'' \circ \theta\ C$.

By Lemma B.6 and the stability of $cmp_{\emptyset}$, we have

$$\forall (\tau\ \boldsymbol{eq}\ v) \in eqs(C)\ .\ cmp_{\emptyset}(\theta''\ \theta\ \tau, \theta''\ \theta\ v) = \mathbf{eq}$$

regardless of θ'', hence the equality constraints in C do not restrict our choice of θ''.

Similarly, by the stability of cmp_{opaque}, $cmp_{opaque}(\theta\ \tau, v_i) \in \{\mathbf{lt}, \mathbf{gt}\}$ implies $cmp_{opaque}(\theta''\ \theta\ \tau, \theta''\ v_i) \in \{\mathbf{lt}, \mathbf{gt}\}$ for any θ'', hence these pairs of types within insertion constraints in (d) also do not restrict our choice of θ''.

Hence θ'' is constrained only by those insertion constraints s.t.

$$\begin{array}{l} (\tau\ \boldsymbol{ins}\ \rho) \in inss(C) \\ \wedge\ \theta\ \rho = (\#)_m\ \overline{v}\ l \\ \wedge\ \exists i\ .\ cmp_{opaque}(\theta\ \tau, v_i) = \mathbf{unk} \end{array}$$

Collect all such pairs of types as $\overline{\tau'}$ and $\overline{\upsilon'}$. Thus $\forall i \,.\, cmp_{opaque}(\theta\ \tau'_i, \upsilon'_i) = \mathbf{unk}$. Then by Lemma B.4, there exists a θ'' (constructed within the proof) s.t.

$$\forall i \,.\, cmp_{opaque}(\theta''\ \theta\ \tau'_i, \theta''\ \upsilon'_i) \in \{\mathbf{lt}, \mathbf{gt}\} \quad \text{(e)}$$

Now consider again each insertion constraint $(w : \tau\ \boldsymbol{ins}\ \rho) \in inss(C)$, where $\theta'\ \rho = (\texttt{\#})_m\ \overline{\upsilon}\ l$. From (d) and (e) we have

$$\forall i \,.\, cmp_{opaque}(\theta''\ \theta\ \tau, \theta''\ \upsilon_i) \in \{\mathbf{lt}, \mathbf{gt}\}$$

Furthermore, if $l = a$, then by the construction of θ'' we have $\theta''\ a = A\ \texttt{\#}\ \texttt{Empty}$ for a fresh newtype A. That is, $\theta''\ \theta\ \rho = (\texttt{\#})_{m+1}\ (\overline{\theta''\ \upsilon} +\!\!+ A)\ \texttt{Empty}$. Since $\Delta \vdash a : \texttt{Row}$, $\Delta \vdash \tau : \mathsf{Type}$, and $\Delta \vdash \theta$ subst, $\theta\ \tau \neq a$, and so since $\theta'' \circ \theta$ is a grounding substitution

$$cmp_{opaque}(\theta''\ \theta\ \tau, A) \in \{\mathbf{lt}, \mathbf{gt}\}$$

Define S as either (if $l = \texttt{Empty}$)

$$S = sortingPerms(\theta''\ \theta\ \tau, \theta''\ \upsilon_1, \ldots, \theta''\ \upsilon_m)$$

or (if $l = a$)

$$S = sortingPerms(\theta''\ \theta\ \tau, \theta''\ \upsilon_1, \ldots, \theta''\ \upsilon_m, A)$$

Then we have $S \neq \emptyset$ and $\pi_1, \pi_2 \in S \Longrightarrow \pi_1^{-1}\ 1 = \pi_2^{-1}\ 1$. Thus

$$[\![\theta''\ \theta\ \tau\ \boldsymbol{ins}\ \theta''\ \theta\ \rho]\!] = \{iind : j\}$$

where $j = \pi^{-1}\ 1$ for every $\pi \in S$. We thus take $\eta\ w = \mathsf{Inc}^j\ \mathsf{One}$.

Taking $\theta' = \theta'' \circ \theta$, we have $\eta \models \theta\ C$ as required.

□

Lemma B.16 *satisfiable*(C) iff *saturate*$(C) \neq \emptyset$.

Proof Immediate by Lemma B.15. □

Lemma B.17 If $C \vdash^e D$ and *satisfiable*(C) then *satisfiable*(D).

Proof Let $C \vdash^e D \hookrightarrow B$. Since *satisfiable*$(C)$ there exists a θ and η s.t. $\eta \models \theta\ C$. Then by Lemma B.13 $env(B, \eta) \models \theta\ D$. Thus *satisfiable*$(D)$. □

Lemma B.18 If $\Delta \vdash C$ constraint and $\theta \in saturate(C)$ then there exists a Δ' s.t. $\Delta +\!\!+ \Delta' \vdash \theta$ subst.

Proof By definition of *saturate* and Lemma B.5. □

Lemma B.19 If $\Delta \vdash C/D$ constraint and $\Delta \vdash \theta$ subst then

(i) *saturate*$(C) = \emptyset$ implies *saturate*$(\theta\ C +\!\!+ D) = \emptyset$

(ii) *saturate*$(\theta\ C +\!\!+ D) \neq \emptyset$ implies *saturate*$(C) \neq \emptyset$

Proof From definition of *saturate*, Lemma B.7 and stability of cmp_{opaque}. □

Lemma B.20 Let $\Delta \vdash C$ constraint and $\vdash \theta : \Delta \to \Delta_{init}$ and $C = inhs(C)$ and $\eta \models \theta\ C$. Then $\mathtt{true} \vdash^e \theta\ C \hookrightarrow B$ and $env(B) = \eta_{\restriction names(C)}$.

Proof Notice the restriction of C to only include inheritable constraints. This restriction is necessary because $\mathtt{true}$ can never entail $\theta\ C$ if C contains implicit parameter constraints.

Let $\eta \models \theta\ C$. By CONJ, it is sufficient to show for each $(w : c) \in C$ that $\mathtt{true} \vdash^e \theta\ c \hookrightarrow W$ for $[\![W]\!]. = \eta\ w$. However, by Lemma B.12 we already know $[\![W]\!]. \in [\![\theta\ c]\!] \ni \eta\ w$, and thus $[\![W]\!]. = \eta\ w$. Hence we need only show existance of a derivation.

We proceed by case analysis on each $(w : c) \in C$.

case $c = \tau\ \boldsymbol{eq}\, v$. Then by definition $cmp_{\emptyset}(\theta\ \tau, \theta\ v) = \mathbf{eq}$. Thus by CONJ and EQUALITY $\mathtt{true} \vdash^e \theta\, \tau\ \boldsymbol{eq}\, \theta\ v$.

case $c = \tau\ \boldsymbol{ins}\ \rho$. W.l.o.g. assume $\theta\ \rho = (\mathbf{\#})_n\ \overline{v}\ \mathbf{Empty}$. Then by definition $sortingPerms(\theta\ \tau, v_1, \ldots, v_n) = S$ where $S \neq \emptyset$ and $\pi_1, \pi_2 \in S \implies \pi_1^{-1}\ 1 = \pi_2^{-1}\ 1$. Thus $\forall i\ .\ cmp_{opaque}(\theta\ \tau, v_i) \in \{\mathbf{lt}, \mathbf{gt}\}$. Thus by inspection of rules for $\vdash^m$, $\mathtt{true} \vdash^m \theta\, \tau\ \boldsymbol{ins}\, \theta\ \rho$. Then by CONJ and MEMBER $\mathtt{true} \vdash^e \theta\, \tau\ \boldsymbol{ins}\, \theta\ \rho$.

□

Lemma B.21 Let $\Delta \vdash C/D$ constraint. If $C \vdash^e D \hookrightarrow B_1$ and $C \vdash^e D \hookrightarrow B_2$ then for every $\vdash \theta : \Delta \to \Delta_{init}$ and η s.t. $\eta \models \theta\ C$, $env(B_1, \eta) = env(B_2, \eta)$.

Proof By Lemma B.13 $env(B_1, \eta) \models C$ and $env(B_2, \eta) \models C$. By the definitions of Figure 4.14, the denotation of each member of C is a singleton. Hence $env(B_1, \eta) = env(B_2, \eta)$. □

Lemma B.22 If $C \vdash^m \tau\ \boldsymbol{ins}\, \rho \hookrightarrow W$ and $cmp_{opaque}(\tau, \tau') = \mathbf{eq}$ and $cmp_{opaque}(\rho, \rho') = \mathbf{eq}$ and $C \equiv C'$ then $C' \vdash^m \tau'\ \boldsymbol{ins}\, \rho' \hookrightarrow W$.

Proof Straightforward induction. □

Lemma B.23 If $C \vdash^e d \hookrightarrow W$ and $d \equiv d'$ and $C \equiv C'$ then $C' \vdash^e d' \hookrightarrow W$.

Proof From Lemma B.22 and Lemma 4.2 (xiii) if d is an insertion constraint. Otherwise, result is immediate from transitivity of $cmp_{\emptyset}$. □

Lemma B.24 If $C \vdash^e D \hookrightarrow B$ and $D \equiv D'$ then $C \vdash^e D' \hookrightarrow B$.

Proof From Lemma B.23. □

Lemma B.25 Let $\Delta \vdash C$ constraint and $\Delta \vdash \tau : \mathtt{Type}$ and $\Delta \vdash \rho : \mathtt{Row}$ and $\Delta \vdash \theta$ subst. Then (a) $C \vdash^m \tau\ \boldsymbol{ins}\, \rho \hookrightarrow W$ implies $\theta\ C \vdash^m \theta\, \tau\ \boldsymbol{ins}\, \theta\ \rho \hookrightarrow W$.

Proof By induction on derivation of (a):

case MEMPTY: Immediate.
case MREF: Let (a) be

$$C \vdash^m \tau\ \boldsymbol{ins}\, \rho \hookrightarrow w$$

Then by MREF

$$cmp_{opaque}(\tau, \tau') = \mathbf{eq} \wedge cmp_{opaque}(\rho, \rho') = \mathbf{eq} \wedge (w : \tau'\ \boldsymbol{ins}\, \rho') \in C$$

and thus by stability of cmp_{opaque}

$$cmp_{opaque}(\theta\,\tau, \theta\,\tau') = \mathbf{eq} \wedge cmp_{opaque}(\theta\,\rho, \theta\,\rho') = \mathbf{eq}$$

Hence, by MREF

$$\theta\ C \vdash^m \theta\,\tau\ \mathit{ins}\,\theta\,\rho \hookrightarrow w$$

as required.

case MCONT: Let (a) be

$$C \vdash^m \tau\ \mathit{ins}\ (\#)_{n-1}\ \overline{v}_{\backslash i}\ l \hookrightarrow W$$

Then by MCONT

$$\begin{array}{c} cmp_{opaque}(\tau, v_i) = \mathbf{lt} \\ C \vdash^m \tau\ \mathit{ins}\ (\#)_n\ \overline{v}\ l \hookrightarrow W \end{array}$$

Thus by stability of cmp_{opaque} and I.H. on (c)

$$cmp_{opaque}(\theta\,\tau, \theta\,v_i) = \mathbf{lt}\theta\ C \vdash^m \theta\,\tau\ \mathit{ins}\ (\#)_n\ (\theta\,\overline{v})\ (\theta\,l) \hookrightarrow W$$

hence by MCONT

$$\theta\ C \vdash^m \theta\,\tau\ \mathit{ins}\ (\#)_{n-1}\ (\theta\,\overline{v}_{\backslash i})\ (\theta\,l) \hookrightarrow W$$

as required.

case MDEC, MEXP, MINC: Similar to case MCONT.

□

Lemma B.26 Let $\Delta \vdash C$ constraint and $\Delta \vdash d$ constraint and $\Delta \vdash \theta$ subst. Then $C \vdash^m d \hookrightarrow W$ implies $\theta\ C \vdash^m \theta d \hookrightarrow W$.

Proof By case analysis on d:

case $d = \tau\ \mathit{eq}\,v$. Then by EQUALS

$$\forall \theta' \in \mathit{saturate}(C)\ .\ cmp_{\emptyset}(\theta'\,\tau, \theta'\,v) = \mathbf{eq}$$

Then by definition of *saturate*:

$$\begin{array}{l} \forall \theta' \in \mathit{mgus}_{\emptyset}(\mathbf{Id} \vdash \mathit{eqs}(C))\ . \\ \quad \mathit{satisfied}(\theta'\ \mathit{inss}(C)) \Longrightarrow \\ \quad\quad cmp_{\emptyset}(\theta'\,\tau, \theta'\,v) = \mathbf{eq} \end{array}$$

Then by Lemma B.8

$$\begin{array}{l} \forall \theta'' \in \mathit{mgus}_{\emptyset}(\mathbf{Id} \vdash \theta\ \mathit{eqs}(C))\ . \\ \quad \exists \theta' \in \mathit{mgus}_{\emptyset}(\mathbf{Id} \vdash \mathit{eqs}(C))\ . \\ \quad\quad \exists \theta'''\ .\ \theta'' \circ \theta \equiv_{\emptyset} \theta''' \circ \theta' \\ \quad\quad\quad \wedge\ \mathit{satisfied}(\theta'\ \mathit{inss}(C)) \Longrightarrow \\ \quad\quad\quad\quad cmp_{\emptyset}(\theta'\,\tau, \theta'\,v) = \mathbf{eq} \end{array}$$

Then by stability of $cmp_\emptyset$ and Lemma B.11 (i)

$$\begin{array}{l}\forall\theta'' \in mgus_\emptyset(\mathbf{Id} \vdash \theta\ eqs(C))\ .\\ \quad \exists\theta' \in mgus_\emptyset(\mathbf{Id} \vdash eqs(C))\ .\\ \quad\quad \exists\theta'''\ .\ \theta'' \circ \theta \equiv_\emptyset \theta''' \circ \theta'\\ \quad\quad\quad \wedge\ satisfied(\theta'''\ \theta'\ inss(C)) \Longrightarrow\\ \quad\quad\quad\quad cmp_\emptyset(\theta'''\ \theta'\ \tau, \theta'''\ \theta'\ \upsilon) = \mathbf{eq}\end{array}$$

Then by Lemma B.11 (ii)

$$\begin{array}{l}\forall\theta'' \in mgus_\emptyset(\mathbf{Id} \vdash \theta\ eqs(C))\ .\\ \quad satisfied(\theta''\ \theta\ inss(C)) \Longrightarrow\\ \quad\quad cmp_\emptyset(\theta''\ \theta\ \tau, \theta''\ \theta\ \upsilon) = \mathbf{eq}\end{array}$$

and hence

$$\forall\theta'' \in saturate(\theta\ C)\ .\ cmp_\emptyset(\theta''\ \theta\ \tau, \theta''\ \theta\ \upsilon) = \mathbf{eq}$$

which by EQUALS implies

$$\theta\ C \vdash^e \theta\ \tau\ \boldsymbol{eq}\ \theta\ \upsilon \hookrightarrow \mathsf{True}$$

as required.

case $d = \tau\ \boldsymbol{ins}\ \rho$: Then by INSERT

$$\forall\theta' \in saturate(C)\ .\ \theta'\ inss(C) \vdash^m \theta'\ \tau\ \boldsymbol{ins}\ \theta'\ \rho \hookrightarrow W$$

By the same reasoning as for case EQUALS

$$\begin{array}{l}\forall\theta'' \in mgus_\emptyset(\mathbf{Id} \vdash \theta\ eqs(C))\ .\\ \quad \exists\theta' \in mgus_\emptyset(\mathbf{Id} \vdash eqs(C))\ .\\ \quad\quad \exists\theta'''\ .\ \theta'' \circ \theta \equiv_\emptyset \theta''' \circ \theta'\\ \quad\quad\quad \wedge\ satisfied(\theta'\ inss(C)) \Longrightarrow\\ \quad\quad\quad\quad \theta'\ inss(C) \vdash^m \theta'\ \tau\ \boldsymbol{ins}\ \theta'\ \rho \hookrightarrow W\end{array}$$

Then by Lemma B.25 and Lemma B.11 (i)

$$\begin{array}{l}\forall\theta'' \in mgus_\emptyset(\mathbf{Id} \vdash \theta\ eqs(C))\ .\\ \quad \exists\theta' \in mgus_\emptyset(\mathbf{Id} \vdash eqs(C))\ .\\ \quad\quad \exists\theta'''\ .\ \theta'' \circ \theta = \theta''' \circ \theta'\\ \quad\quad\quad \wedge\ satisfied(\theta'''\ \theta'\ inss(C)) \Longrightarrow\\ \quad\quad\quad\quad \theta'''\ \theta'\ inss(C) \vdash^m \theta'''\ \theta'\ \tau\ \boldsymbol{ins}\ \theta'''\ \theta'\ \rho \hookrightarrow W\end{array}$$

Thus by Lemma B.11 (ii) and Lemma B.22

$$\begin{array}{l}\forall\theta'' \in mgus_\emptyset(\mathbf{Id} \vdash \theta\ eqs(C))\ .\\ \quad \wedge\ satisfied(\theta''\ \theta\ inss(C)) \Longrightarrow\\ \quad\quad \theta''\ \theta\ inss(C) \vdash^m \theta''\ \theta\ \tau\ \boldsymbol{ins}\ \theta''\ \theta\ \rho \hookrightarrow W\end{array}$$

which by INSERT implies

$$\theta\ C \vdash^e \theta\ \tau\ \boldsymbol{ins}\ \theta\ \rho \hookrightarrow W$$

as required.

□

Lemma B.27 Let $\Delta \vdash C$ constraint and $\Delta \vdash D$ constraint. and $\Delta \vdash \theta$ subst. Then $C \vdash^m D \hookrightarrow B$ implies $\theta\, C \vdash^m \theta D \hookrightarrow B$.

Proof Straightforward application of Lemma B.26. □

Lemma B.28 $C \vdash^e C \hookrightarrow \cdot$

Proof Straightforward from definition of rule MREF and definition of *saturate*. □

Lemma B.29 If (a) $C \vdash^m \tau\ \textbf{\textit{ins}}\ \rho \hookrightarrow W$ and $\mathit{satisfied}(C)$ then $\neg \mathit{isIn}(\tau, \rho)$.

Proof By definition of *satisfied* and straightforward induction on (a). □

Lemma B.30 If $\Delta \vdash C/D$ constraint and $\Delta \vdash d$ constraint and (a) $C \vdash^e D \hookrightarrow B$ and (b) $D \vdash^e d \hookrightarrow W$ then $C \vdash^e d \hookrightarrow W'$.

Furthermore, if $\vdash \theta : \Delta \rightarrow \Delta_{init}$ and $\eta \models \theta\ C$ then $[\![W]\!]_{env(B,\eta)} = [\![W']\!]_{\eta}$.

Proof The first part procceds by case analysis on d:

case $d = \tau\ \textbf{\textit{eq}}\ \upsilon$: By (a) and definition of *saturate*

$$\begin{array}{l} \forall \theta'' \in \mathit{mgus}_{\emptyset}(\mathbf{Id} \vdash \mathit{eqs}(C))\ . \\ \quad \mathit{satisfied}(\theta''\ \mathit{inss}(C)) \Longrightarrow \\ \quad\quad \forall (\tau'\ \textbf{\textit{eq}}\ \upsilon') \in \mathit{eqs}(D)\ .\ \mathit{cmp}_{\emptyset}(\theta''\ \tau', \theta''\ \upsilon') = \mathbf{eq} \\ \quad\quad \wedge\ \forall (\tau'\ \textbf{\textit{ins}}\ \rho') \in \mathit{inss}(D)\ .\ \theta''\ \mathit{inss}(C) \vdash^m \theta''\ \tau'\ \textbf{\textit{ins}}\ \theta''\ \rho' \hookrightarrow _ \end{array}$$

Then by Lemma B.7 and Lemma B.29

$$\begin{array}{l} \forall \theta'' \in \mathit{mgus}_{\emptyset}(\mathbf{Id} \vdash \mathit{eqs}(C))\ . \\ \quad \mathit{satisfied}(\theta''\ \mathit{inss}(C)) \Longrightarrow \\ \quad\quad \mathbf{Id} \in \mathit{mgus}_{\emptyset}(\mathbf{Id} \vdash \theta''\ \mathit{eqs}(D)) \wedge \mathit{satisfied}(\theta''\ \mathit{inss}(D)) \end{array} \tag{d}$$

Since by Lemma B.26 on (b)

$$\theta''\ D \vdash^e \theta''\ d$$

we have

$$\begin{array}{l} \forall \theta' \in \mathit{mgus}_{\emptyset}(\mathbf{Id} \vdash \theta''\ \mathit{eqs}(D))\ . \\ \quad \mathit{satisfied}(\theta'\ \theta''\ \mathit{inss}(D)) \Longrightarrow \\ \quad\quad \mathit{cmp}_{\emptyset}(\theta'\ \theta''\ \tau, \theta'\ \theta''\ \upsilon) = \mathbf{eq} \end{array}$$

then by (d) we may take $\theta' = \mathbf{Id}$ so that

$$\begin{array}{l} \mathit{satisfied}(\theta''\ \mathit{inss}(D)) \\ \wedge\ \mathit{cmp}_{\emptyset}(\theta''\ \tau, \theta''\ \upsilon) = \mathbf{eq} \end{array}$$

Thus

$$\begin{array}{l} \forall \theta'' \in \mathit{mgus}_{\emptyset}(\mathbf{Id} \vdash \mathit{eqs}(C))\ . \\ \quad \mathit{satisfied}(\theta''\ \mathit{inss}(C)) \Longrightarrow \\ \quad\quad \mathit{cmp}_{\emptyset}(\theta''\ \tau, \theta''\ \upsilon) = \mathbf{eq} \end{array}$$

so by EQUALS

$$C \vdash^e \tau\ \textbf{\textit{eq}}\ \upsilon \hookrightarrow \mathsf{True}$$

as required.

case $d = \tau \ \boldsymbol{ins}\ \rho$: By (a) and definition of *saturate*:

$$\begin{array}{l} \forall \theta'' \in mgus_{\emptyset}(\mathbf{Id} \vdash eqs(C)) \ . \\ \quad satisfied(\theta''\ inss(C)) \Longrightarrow \\ \quad\quad \forall (\tau'\ \boldsymbol{eq}\, v') \in eqs(D) \ . \ cmp_{\emptyset}(\theta''\ \tau', \theta''\ v') = \mathbf{eq} \\ \quad\quad \wedge\ \forall (w : \tau'\ \boldsymbol{ins}\, \rho') \in inss(D) \ . \ \theta''\ inss(C) \vdash^{m} \theta''\ \tau'\ \boldsymbol{ins}\, \theta''\ \rho' \hookrightarrow W''_{w} \end{array} \tag{d}$$

where $B = \overline{w = W''_w}$.

By the same arguments as above we have

$$\begin{array}{l} \forall \theta'' \in mgus_{\emptyset}(\mathbf{Id} \vdash eqs(C)) \ . \\ \quad satisfied(\theta''\ inss(C)) \Longrightarrow \\ \quad\quad \mathbf{Id} \in mgus_{\emptyset}(\mathbf{Id} \vdash \theta''\ eqs(D)) \wedge satisfied(\theta''\ inss(D)) \end{array}$$

Since by Lemma B.26 on (b)

$$\theta''\ D \vdash^{e} \theta''\ d \hookrightarrow W$$

we have

$$\begin{array}{l} \forall \theta' \in mgus_{\emptyset}(\mathbf{Id} \vdash \theta''\ eqs(D)) \ . \\ \quad satisfied(\theta'\ \theta''\ inss(D)) \Longrightarrow \\ \quad\quad \theta'\ \theta''\ inss(D) \vdash^{m} \theta'\ \theta''\ \tau\ \boldsymbol{ins}\, \theta'\ \theta''\ \rho \hookrightarrow W \end{array}$$

then by (d) we may take $\theta' = \mathbf{Id}$ so that

$$\begin{array}{l} \forall \theta'' \in mgus_{\emptyset}(\mathbf{Id} \vdash eqs(C)) \ . \\ \quad satisfied(\theta''\ inss(C)) \Longrightarrow \\ \quad\quad \theta''\ inss(D) \vdash^{m} \theta''\ \tau\ \boldsymbol{ins}\, \theta''\ \rho \hookrightarrow W \end{array} \tag{e}$$

By inspection, each rule for deciding $\vdash^m$ has zero or one invocation of $\vdash^m$ in its hypotheses. Hence, a derivation of

$$\theta''\ inss(D) \vdash^{m} \theta''\ \tau\ \boldsymbol{ins}\, \theta''\ \rho \hookrightarrow W$$

is a chain with leaf an instance of rule MEMPTY or MREF. We consider each case:

case MEMPTY: Replace the leaf

$$\frac{}{\theta''\ inss(D) \vdash^{m} \tau'\ \boldsymbol{ins}\, \mathtt{Empty} \hookrightarrow \mathsf{One}}\ \text{MEMPTY}$$

with

$$\frac{}{\theta''\ inss(C) \vdash^{m} \tau'\ \boldsymbol{ins}\, \mathtt{Empty} \hookrightarrow \mathsf{One}}\ \text{MEMPTY}$$

Then $\theta''\ inss(C) \vdash^{m} \theta''\ \tau\ \boldsymbol{ins}\, \theta''\ \rho \hookrightarrow W$, and so

$$\begin{array}{l} \forall \theta'' \in mgus_{\emptyset}(\mathbf{Id} \vdash eqs(C)) \ . \\ \quad satisfied(\theta''\ inss(C)) \Longrightarrow \\ \quad\quad \theta'' inss(C) \vdash^{m} \theta''\ \tau\ \boldsymbol{ins}\, \theta''\ \rho \hookrightarrow W \end{array}$$

which by INSERT implies

$$C \vdash^e \tau \; \textbf{\textit{ins}} \; \rho \hookrightarrow W$$

as required.

case MREF: The leaf is of the form

$$\frac{\begin{array}{c}(w : \theta'' \; \tau' \; \textbf{\textit{ins}} \; \theta'' \; \rho') \in \theta'' \; inss(D) \\ cmp_{opaque}(\theta'' \; \tau', \tau'') = \textbf{eq} \\ cmp_{opaque}(\theta'' \; \rho', \rho'') = \textbf{eq}\end{array}}{\theta'' \; inss(D) \vdash^m \tau'' \; \textbf{\textit{ins}} \; \rho'' \hookrightarrow w} \text{ MREF}$$

By (e)

$$\theta'' \; inss(C) \vdash^m \theta'' \; \tau' \; \textbf{\textit{ins}} \; \theta'' \; \rho' \hookrightarrow W''_w$$

and so by Lemma B.22

$$\theta'' \; inss(C) \vdash^m \tau'' \; \textbf{\textit{ins}} \; \rho'' \hookrightarrow W''_w$$

Hence

$$\theta'' \; inss(C) \vdash^m \theta'' \; \tau \; \textbf{\textit{ins}} \; \theta'' \; \rho \hookrightarrow W[w \mapsto W''_w]$$

and thus

$$\begin{array}{l}\forall \theta'' \in mgus_{\emptyset}(\textbf{Id} \vdash eqs(C)) \; . \\ \quad satisfied(\theta'' \; inss(C)) \Longrightarrow \\ \qquad \theta'' inss(C) \vdash^m \theta'' \; \tau \; \textbf{\textit{ins}} \; \theta'' \; \rho \hookrightarrow W[w \mapsto W''_w]\end{array}$$

which by INSERT implies

$$C \vdash^e \tau \; \textbf{\textit{ins}} \; \rho \hookrightarrow W[w \mapsto W''_w]$$

For the second part, notice that by Lemma B.12 $[\![W']\!]_\eta \in [\![\theta \; d]\!]$, $env(B, \eta) \models D$, and thus $[\![W]\!]_{env(B,\eta)} \in [\![\theta \; d]\!]$. Then $[\![W']\!]_\eta = [\![W]\!]_{env(B,\eta)}$. □

Lemma B.31 If $\Delta \vdash C/D'/D$ constraint and $C \vdash^e D' \hookrightarrow B$ and $D' \vdash^e D \hookrightarrow B'$ then $C \vdash^e D \hookrightarrow B''$.

Furthermore, if $\vdash \theta : \Delta \to \Delta_{init}$ and $\eta \models \theta \; C$ then $env(B +\!\!+ B', \eta)_{\restriction names(D)} = env(B'', \eta)_{\restriction names(D)}$

Proof By Lemma B.30 and definition of *env*. □

Lemma B.32 If $\Delta \vdash C/D/d$ constraint and $C \vdash^m d \hookrightarrow W$ then $C +\!\!+ D \vdash^m d \hookrightarrow W$

Proof Straightforward induction. □

Lemma B.33 If $\Delta \vdash C/D/d$ constraint and (a) $C \vdash^e d \hookrightarrow W$ then $C +\!\!+ D \vdash^e d \hookrightarrow W$.

Proof Notice that if $\theta \in mgus_{\emptyset}(\textbf{Id} \vdash eqs(C))$ and $\theta' \in mgus_{\emptyset}(\textbf{Id} \vdash eqs(C) +\!\!+ eqs(D))$ then by Lemma B.6 and Lemma B.7 there exists a θ'' s.t. $\theta' \equiv_{\emptyset} \theta'' \circ \theta$.

Furthermore, by stability of cmp_{opaque}, if $\neg isIn(\theta' \; \tau, \theta' \; \rho)$ then $\neg isIn(\theta \; \tau, \theta \; \rho)$.

We proceed by case analysis on d:

case $d = (\tau\ eq\ \upsilon)$, $W = \mathsf{True}$: Then from (a)

$$\begin{array}{l}\forall\theta \in mgus_{\emptyset}(\mathbf{Id} \vdash eqs(C))\ .\\ \quad (\forall(\tau'\ \boldsymbol{ins}\ \rho') \in inss(C)\ .\ \neg isIn(\theta\ \tau', \theta\ \rho')) \Longrightarrow\\ \quad\quad cmp_{\emptyset}(\theta\ \tau, \theta\ \upsilon) = \mathbf{eq}\end{array}$$

Then by above results

$$\begin{array}{l}\forall\theta' \in mgus_{\emptyset}(\mathbf{Id} \vdash eqs(C) +\!\!+ eqs(D))\ .\\ \quad \exists\theta \in mgus_{\emptyset}(\mathbf{Id} \vdash eqs(C))\ .\ \exists\theta''\ .\\ \quad\quad \theta' \equiv_{\emptyset} \theta'' \circ \theta\\ \quad\quad \wedge\ (\forall(\tau'\ \boldsymbol{ins}\ \rho') \in inss(C)\ .\ \neg isIn(\theta'\ \tau', \theta'\ \rho')) \Longrightarrow\\ \quad\quad\quad cmp_{\emptyset}(\theta\ \tau, \theta\ \upsilon) = \mathbf{eq}\end{array}$$

Thus by the transitivity and stability of $cmp_{\emptyset}$

$$\begin{array}{l}\forall\theta' \in mgus_{\emptyset}(\mathbf{Id} \vdash eqs(C) +\!\!+ eqs(D))\ .\\ \quad (\forall(\tau'\ \boldsymbol{ins}\ \rho') \in inss(C)\ .\ \neg isIn(\theta'\ \tau', \theta'\ \rho')\\ \quad \wedge\ \forall(\tau'\ \boldsymbol{ins}\ \rho') \in inss(D)\ .\ \neg isIn(\theta'\ \tau', \theta'\ \rho')) \Longrightarrow\\ \quad\quad cmp_{\emptyset}(\theta'\ \tau, \theta'\ \upsilon) = \mathbf{eq}\end{array}$$

which is equivalent to

$$C +\!\!+ D \vdash^{e} d \hookrightarrow \mathsf{True}$$

as required.

case $d = (\tau\ \boldsymbol{ins}\ \rho)$. Then from (a)

$$\begin{array}{l}\forall\theta \in mgus_{\emptyset}(\mathbf{Id} \vdash eqs(C))\ .\\ \quad (\forall(\tau'\ \boldsymbol{ins}\ \rho') \in inss(C)\ .\ \neg isIn(\theta\ \tau', \theta\ \rho')) \Longrightarrow\\ \quad\quad \theta\ inss(C) \vdash^{m} \theta\ \tau\ \boldsymbol{ins}\ \theta\ \rho \hookrightarrow W\end{array}$$

Then by above results

$$\begin{array}{l}\forall\theta' \in mgus_{\emptyset}(\mathbf{Id} \vdash eqs(C) +\!\!+ eqs(D))\ .\\ \quad \exists\theta \in mgus_{\emptyset}(\mathbf{Id} \vdash eqs(C))\ .\ \exists\theta''\ .\\ \quad\quad \theta' \equiv_{\emptyset} \theta'' \circ \theta\\ \quad\quad \wedge\ (\forall(\tau'\ \boldsymbol{ins}\ \rho') \in inss(C)\ .\ \neg isIn(\theta'\ \tau', \theta'\ \rho')) \Longrightarrow\\ \quad\quad\quad \theta\ inss(C) \vdash^{m} \theta\ \tau\ \boldsymbol{ins}\ \theta\ \rho \hookrightarrow W\end{array}$$

Thus by Lemma B.25, Lemma B.22 and Lemma B.32

$$\begin{array}{l}\forall\theta' \in mgus_{\emptyset}(\mathbf{Id} \vdash eqs(C) +\!\!+ eqs(D))\ .\\ \quad (\forall(\tau'\ \boldsymbol{ins}\ \rho') \in inss(C)\ .\ \neg isIn(\theta'\ \tau', \theta'\ \rho')\\ \quad \wedge\ \forall(\tau'\ \boldsymbol{ins}\ \rho') \in inss(D)\ .\ \neg isIn(\theta'\ \tau', \theta'\ \rho')) \Longrightarrow\\ \quad\quad \theta'\ inss(C) +\!\!+ \theta'\ inss(D) \vdash^{m} \theta'\ \tau\ \boldsymbol{ins}\ \theta'\ \rho \hookrightarrow W\end{array}$$

which is equivalent to

$$C +\!\!+ D \vdash^{e} d \hookrightarrow W$$

as required.

□

Lemma B.34 If $\Delta \vdash C/D/D'$ constraint and $C \vdash^e D' \hookrightarrow B$ then $C +\!\!+ D \vdash^e D' \hookrightarrow B$.

Proof Straightforward application of Lemma B.33. □

B.4 Type Soundness

Lemma B.35 If $\Delta \vdash C$ constraint and $\Delta \vdash \Gamma$ context and $\Delta \mid C \mid \Gamma \vdash t : \tau$ then $\Delta \vdash \tau :$ Type.

Proof Easy induction. Notice the use of well-kinding judgements within VAR, LET, P3, P4, P5 and P7. □

Lemma B.36 Let (a) $\Delta \mid C \mid \Gamma \vdash t : \tau$ and $(x : \texttt{forall}\ \Delta'\ .\ C' \texttt{=>} \tau') \in \Gamma$.

(i) For every (run-time) specialisation of x within t there exists a $\vdash \theta : \Delta' \rightarrow \Delta$ and $\overline{w}$ s.t. $D = \mathit{named}(C')$, $\mathit{names}(D) = \overline{w}$ and $C \vdash^e \theta\, D$.

(ii) If $x \in \mathit{fv}(t)$ then there exists a $\vdash \theta : \Delta' \rightarrow \Delta$ and $\overline{w}$ s.t. $D = \mathit{named}(C')$, $\mathit{names}(D) = \overline{w}$ and $C \vdash^e \theta\, D$.

Proof For (i), by induction on derivation of (a):

case VAR: Let (a) be

$$\Delta \mid C \mid \Gamma \vdash y : \tau''[\overline{a \mapsto v}]$$

If $y \neq x$ then the result holds vacuously. Otherwise, we have $y = x$, and by VAR

$$\tau'' = \tau'$$
$$\vdash [\overline{a \mapsto v}] : \Delta' \rightarrow \Delta$$
$$C \vdash^e D[\overline{a \mapsto v}]$$

where $D = \mathit{named}(C')$. The result is immediate.

case APP: Let (a) be

$$\Delta \mid C \mid \Gamma \vdash t\ u : \tau$$

where by APP

$$\Delta \mid C \mid \Gamma \vdash t : v \quad \text{(b)}$$
$$\Delta \mid C \mid \Gamma \vdash u : v' \quad \text{(c)}$$

case (in t) By I.H. on (b) the result holds for each specialisation of x in t.

case (in u) Similary, by I.H. on (c) it holds for each specialisation of x in u.

case LET: Let (a) be

$$\Delta \mid C \mid \Gamma \vdash \texttt{let}\ y = u\ \texttt{in}\ t : \tau$$

Then by LET

$$y \in fv(t) \tag{b}$$
$$D'_1 = inhs(C) \tag{c}$$
$$saturate(D'_1 +\!\!+ D'_2) \neq \emptyset \tag{d}$$
$$\Delta +\!\!+ \Delta'' \mid D'_1 +\!\!+ D'_2 \mid \Gamma \vdash u : \tau'' \tag{e}$$
$$\Delta \mid C \mid \Gamma, y : \sigma \vdash t : \tau \tag{f}$$
$$\Delta \vdash D'_1 \text{ constraint} \tag{g}$$
$$\Delta +\!\!+ \Delta'' \vdash D'_2 \text{ constraint} \tag{h}$$

where $\sigma = \texttt{forall}\ \Delta''\ .\ anon(D'_2)\ \texttt{=>}\ \tau''$. (We shall ignore shadowing, thus $x \neq y$.)

case (in u) W.l.o.g. assume $dom(\Delta') \cap dom(\Delta'') = \emptyset$ and that $named(anon(D'_2)) = D'_2$.
By I.H. on (e), for each specialisation of x in u, there exists $\vdash \theta : \Delta' \to \Delta +\!\!+ \Delta''$ s.t.

$$D'_1 +\!\!+ D'_2 \vdash^e \theta\ D \tag{i}$$

where $D = named(C')$.
By I.H. (this time using y, which is known to occur at least once within t by (b)) on (f) for each specialisation of y in t there exists at least one $\vdash \theta' : \Delta'' \to \Delta$ s.t.

$$C \vdash^e \theta'\ D'_2 \tag{j}$$

Notice by (g) $\theta'\ D'_1 = D'_1$. Then by (c)

$$C \vdash^e \theta'\ D'_1$$

and thus by (j) and CONJ

$$C \vdash^e \theta'\ (D'_1 +\!\!+ D'_2)$$

Then by (i), Lemma B.27 and Lemma B.31

$$C \vdash^e \theta' \circ \theta\ D$$

which is equivalent to

$$C \vdash^e (\theta' \circ \theta)_{\restriction dom(\Delta')} D$$

Furthermore, we have $\vdash (\theta' \circ \theta)_{\restriction dom(\Delta')} : \Delta' \to \Delta$.
Thus the result holds for each specialistion of x via y in t.

case (in t) By I.H. on (h) the result holds for each specialisation of x in t.

Notice that (d) plays no part in this result. Indeed, the test for satisfiability in rule LET is purely to aid the locality of error diagnostics.

Other cases proceed similarly.

For (ii), notice that if $x \in fv(t)$ then it must be specialised at least once. □

Lemma B.37 (i) If $\Delta \mid C \mid \Gamma \vdash t : \tau \hookrightarrow T$ then $\Delta +\!\!+ \Delta' \mid C \mid \Gamma \vdash t : \tau \hookrightarrow T$.

(ii) If $\Delta +\!\!+ \Delta' \mid C \mid \Gamma \vdash_n t : \tau \hookrightarrow T[\bullet]$ then $\Delta \mid C \mid \Gamma \vdash_n t : \tau \hookrightarrow T[\bullet]$.

Proof Straightforward induction. □

Lemma B.38 (i) If $\Delta \mid C \mid \Gamma \vdash t : \tau \hookrightarrow T$ then $\Delta \mid C \mid \Gamma +\!\!+ \Gamma' \vdash t : \tau \hookrightarrow T$.

(ii) If $\Delta \mid C \mid \Gamma \vdash_n t : \tau \hookrightarrow T[\bullet]$ then $\Delta \mid C \mid \Gamma +\!\!+ \Gamma' \vdash_n t : \tau \hookrightarrow T[\bullet]$.

Proof Straightforward induction. □

Theorem B.39 (Type Soundness)

(i) If

(a) $\Delta \mid C \mid \Gamma \vdash t : \tau \hookrightarrow T$

(b) $\vdash \theta : \Delta \to \Delta_{init}$

(c) $env(B) \models \theta\ C$

(d) $\eta \models \theta\ \Gamma$

then $[\![T]\!]_{\eta +\!\!+ env(B)} \in [\![\theta\ \tau]\!]$.

(ii) If

(a) $\Delta \mid C \mid \Gamma \vdash_n t : \tau \hookrightarrow T[\bullet]$

(b) $\vdash \theta : \Delta \to \Delta_{init}$

(c) $env(B) \models \theta\ C$

(d) $\eta \models \theta\ \Gamma$

(e) $[\![U]\!]_{\eta +\!\!+ env(B)} \in [\![\theta\ \tau]\!]$

then $[\![T[U]]\!]_{\eta +\!\!+ env(B)} \in [\![\theta\ \tau]\!]$.

Proof By induction on derivation of (a). (We shall mix the two proofs and rely on the rulename to distinguish between statements (i) and (ii).)

case INT: Let (a) be

$$\Delta \mid C \mid \Gamma \vdash i : \mathbf{Int} \hookrightarrow i$$

Then by definition

$$\begin{aligned} [\![i]\!]_{\eta +\!\!+ env(B)} &= \mathbf{unit_E}\ (\mathsf{int} : i) \\ &\in \mathbf{E}\ \{\mathsf{int} : i \mid i \in Z\} \\ &= [\![\theta\ \mathbf{Int}]\!] \end{aligned}$$

as required.

case APP: Let (a) be

$$\Delta \mid C \mid \Gamma \vdash t\ u : \tau \hookrightarrow T\ U$$

Then by APP we have

$$\Delta \mid C \mid \Gamma \vdash t : \upsilon \hookrightarrow T \qquad \text{(e)}$$

$$\Delta \mid C \mid \Gamma \vdash u : \upsilon' \hookrightarrow U \qquad \text{(f)}$$

$$C \vdash^{e} \upsilon\ eq\, \upsilon' \texttt{ -> } \tau \hookrightarrow \mathsf{True} \qquad \text{(g)}$$

By (c) and Lemma B.14 $eq^m_\emptyset(\theta\, \upsilon, \theta\, \upsilon' \texttt{ -> } \theta\, \tau)$ and thus

$$[\![\theta\, \upsilon]\!] = [\![\theta\, \upsilon' \texttt{ -> } \theta\, \tau]\!]$$

By definition

$$\begin{aligned}[\![T\ U]\!]_{\eta +\!\!+ env(B)} = \ & \mathbf{let_E}\ v \leftarrow [\![T]\!]_{\eta +\!\!+ env(B)} \\ & \mathbf{in\ case}\ v\ \mathbf{of}\ \{ \\ & \quad \mathsf{func} : f \rightarrow f\ [\![U]\!]_{\eta +\!\!+ env(B)}; \\ & \quad \mathbf{otherwise} \rightarrow \mathbf{unit_E}\ (\mathsf{wrong} : *)\ \} \\ = \ & (*)\end{aligned}$$

By I.H. (i) on (f)

$$[\![U]\!]_{\eta +\!\!+ env(B)} \in [\![\theta\, \upsilon']\!]$$

By I.H. (i) on (e)

$$\begin{aligned}[\![T]\!]_{\eta +\!\!+ env(B)} & \in [\![\theta\, \upsilon]\!] \\ & \in [\![\theta\, \upsilon' \texttt{ -> } \theta\, \tau]\!] \\ & = \mathbf{E}\ \{\mathsf{func} : f \mid f \in \mathbf{E}\ \mathcal{V} \rightarrow \mathbf{E}\ \mathcal{V}, v' \in [\![\theta\, \upsilon']\!] \Longrightarrow f\ v' \in [\![\theta\, \tau]\!]\}\end{aligned}$$

thus v is tagged by func, and

$$\begin{aligned}(*) = \ & \mathbf{let_E}\ \mathsf{func} : f \leftarrow [\![T]\!]_{\eta +\!\!+ env(B)} \\ & \mathbf{in}\ f\ [\![U]\!]_{\eta +\!\!+ env(B)} \\ \in \ & [\![\theta\, \tau]\!]\end{aligned}$$

as required.

case VAR (normal case): Let (a) be

$$\Delta \mid C \mid \Gamma \vdash x : \tau[\overline{a \mapsto \upsilon}] \hookrightarrow \mathsf{letw}\ B'\ \mathsf{in}\ x\ names(D')$$

Then by VAR

$$(x : \sigma) \in \Gamma \tag{e}$$

$$\Delta \vdash \overline{\upsilon : \kappa} \tag{f}$$

$$C \vdash^e D'[\overline{a \mapsto \upsilon}] \hookrightarrow B' \tag{g}$$

where $\sigma = \texttt{forall}\ \overline{a : \kappa}\ .\ D \texttt{ => } \tau$, $D' = named(D)$, and $names(D') = (w_1, \ldots, w_n)$. By (c), (g) and Lemma B.13

$$env(B +\!\!+ B') \models \theta\ (D'[\overline{a \mapsto \upsilon}]) \tag{h}$$

By definition

$$\begin{array}{ll} & [\![\mathsf{letw}\ B\ \mathsf{in}\ x\ names(D')]\!]_{\eta +\!\!+ env(B)} \\ = & [\![x\ (w_1, \ldots, w_n)]\!]_{env(B', \eta +\!\!+ env(B))} \\ = & [\![x\ (w_1, \ldots, w_n)]\!]_{\eta +\!\!+ env(B +\!\!+ B')} \\ = & \mathbf{let_E}\ v \leftarrow [\![x]\!]_{\eta +\!\!+ env(B +\!\!+ B')} \\ & \mathbf{in\ case}\ v\ \mathbf{of}\ \{ \\ & \quad \mathsf{ifunc}_n : f \to f\ ([\![w_1]\!]_{\eta +\!\!+ env(B +\!\!+ B')}, \ldots, [\![w_n]\!]_{\eta +\!\!+ env(B +\!\!+ B')}); \\ & \quad \mathbf{otherwise} \to \mathbf{unit_E}\ (\mathsf{wrong} : *)\ \} \\ = & \mathbf{let_E}\ v \leftarrow [\![x]\!]_{\eta} \\ & \mathbf{in\ case}\ v\ \mathbf{of}\ \{ \\ & \quad \mathsf{ifunc}_n : f \to f\ ([\![w_1]\!]_{env(B +\!\!+ B')}, \ldots, [\![w_n]\!]_{env(B +\!\!+ B')}); \\ & \quad \mathbf{otherwise} \to \mathbf{unit_E}\ (\mathsf{wrong} : *)\ \} \\ = & (*) \end{array}$$

W.l.o.g. assume $dom(\theta) \cap \overline{a} = \emptyset$. Then by (d) and (e)

$$\begin{array}{rl} [\![x]\!]_\eta \in & [\![\theta\ \mathtt{forall}\ \overline{a : \kappa}\ .\ D\ \mathtt{=>}\ \tau]\!] \\ = & [\![\mathtt{forall}\ \overline{a : \kappa}\ .\ \theta\ D\ \mathtt{=>}\ \theta\ \tau]\!] \\ = & \bigcap \left\{ S_{(\theta'', B'')} \;\middle|\; \begin{array}{l} \vdash \theta'' : \overline{a : \kappa} \to \Delta_{init}, \\ env(B'') \models \theta''\ (\theta\ D') \end{array} \right\} \end{array}$$

where

$$S_{(\theta'', B'')} = \mathbf{E} \left\{ \mathsf{ifunc}_n : f \;\middle|\; \begin{array}{l} f \in \prod_{1 \leq i \leq n} \mathcal{I} \to \mathbf{E}\ \mathcal{V}, \\ f\ ([\![w_1]\!]_{env(B'')}, \ldots, [\![w_n]\!]_{env(B'')}) \\ \quad \in [\![\theta''\ (\theta\ \tau)]\!] \end{array} \right\}$$

Taking $\theta'' = [\overline{a \mapsto v}]$ and $B'' = B +\!\!+ B'$, by (h) v is tagged by ifunc_n and

$$\begin{array}{rl} (*) = & \mathbf{let_E}\ \mathsf{ifunc}_n : f \leftarrow [\![x]\!]_\eta \\ & \mathbf{in}\ f\ ([\![w_1]\!]_{env(B +\!\!+ B')}, \ldots, [\![w_n]\!]_{env(B +\!\!+ B')}) \end{array}$$

where

$$f\ ([\![w_1]\!]_{env(B +\!\!+ B')}, \ldots, [\![w_n]\!]_{env(B +\!\!+ B')}) \in [\![\theta\ (\tau[\overline{a \mapsto v}])]\!]$$

as required.

case VAR ($f = \mathtt{(Inj\ _)}$): Let (a) be

$$\Delta \mid C \mid \Gamma \vdash \mathtt{(Inj\ _)} : (a\ \mathtt{->}\ \mathtt{One}\ (a\ \mathtt{\#}\ b))[a \mapsto \tau, b \mapsto \rho] \hookrightarrow \mathsf{letw}\ w = W\ \mathsf{in}\ \mathtt{(Inj\ _)}\ w$$

Then by VAR

$$\Delta \vdash \tau : \mathsf{Type} \tag{e}$$

$$\Delta \vdash \rho : \mathsf{Row} \tag{f}$$

$$C \vdash^e w : (a\ ins\ b)[a \mapsto \tau, b \mapsto \rho] \hookrightarrow w = W \tag{g}$$

By definition

$$
\begin{aligned}
& [\![\mathsf{letw}\ w = W\ \mathsf{in}\ (\mathtt{Inj}\ _)\ w]\!]_{\eta +\!\!+ env(B)} \\
&= [\![(\mathtt{Inj}\ _)\ (w)]\!]_{env(w=W, \eta +\!\!+ env(B))} \\
&= [\![(\mathtt{Inj}\ _)\ (w)]\!]_{\eta +\!\!+ env(w=W, env(B))} \\
&= [\![(\lambda(w')\ .\ \lambda x\ .\ \mathsf{Inj}\ w'\ x)\ (w)]\!]_{\eta +\!\!+ env(w=W, env(B))} \\
&= [\![\lambda x\ .\ \mathsf{Inj}\ w\ x]\!]_{\eta +\!\!+ env(w=W, env(B))} \\
&= \mathbf{unit_E}\ (\mathsf{func} : \lambda y\ .\ \ \mathbf{case}\ [\![W]\!]_{env(B)}\ \mathbf{of}\ \{ \\
&\qquad\qquad \mathsf{iind} : i \to \mathbf{unit_E}\ (\mathsf{inj} : \langle i, y\rangle) \\
&\qquad\qquad \mathbf{otherwise} \to \mathbf{unit_E}\ (\mathsf{wrong} : *)\ \}\ ) \\
&= (*)
\end{aligned}
$$

By (c), (g) and Lemma B.14, if $\theta\ \rho = (\mathtt{\#})_n\ \overline{v}\ \mathtt{Empty}$ then

$$S' \neq \emptyset \wedge \forall \pi \in S'\ .\ \pi^{-1}\ 1 = j \wedge [\![W]\!]_{env(B)} = \mathsf{iind} : j$$

where $S' = \mathit{sortingPerms}(\theta\ \tau, v_1, \ldots, v_n)$.
Let $\pi \in S'$. Then

$$
\begin{aligned}
(*) &= \mathbf{unit_E}\ (\mathsf{func} : \lambda y\ .\ \mathbf{unit_E}\ (\mathsf{inj} : \langle j, y\rangle)) \\
&\in \mathbf{E}\ \left\{ \mathsf{func} : f \,\middle|\, \begin{array}{l} f \in \mathbf{E}\ \mathcal{V} \to \mathbf{E}\ \mathcal{V}, \\ v \in [\![\theta\ \tau]\!] \Longrightarrow f\ v \in S \end{array} \right\} \\
&\quad \text{where S} = \mathbf{E}\ \left\{ \mathsf{inj} : \langle i, v'\rangle \,\middle|\, \begin{array}{l} 1 \le i \le (n+1), \\ \mathbf{if}\ i = j\ \mathbf{then}\ v' \in [\![\theta\ \tau]\!] \\ \mathbf{else}\ v' \in [\![v_{(\pi\ i)-1}]\!] \end{array} \right\} \\
&= [\![\theta\ \tau\ \mathtt{->}\ \mathtt{One}\ (\theta\ \tau\ \mathtt{\#}\ \theta\ \rho)]\!] \\
&= [\![\theta\ (\tau\ \mathtt{->}\ \mathtt{One}\ (\tau\ \mathtt{,}\ \rho))]\!] \\
&= [\![\theta\ ((a\ \mathtt{->}\ \mathtt{One}\ (a\ \mathtt{\#}\ b)[a \mapsto \tau, b \mapsto \rho]))]\!]
\end{aligned}
$$

as required.

case VAR ($f = \mathtt{(Triv)}$): Let (a) be

$$\Delta \mid C \mid \Gamma \vdash \mathtt{(Triv)} : \mathtt{All\ Empty} \hookrightarrow \mathsf{letw} \cdot \mathsf{in}\ \mathtt{(Triv)}$$

Then by definition

$$
\begin{aligned}
& [\![\mathsf{letw} \cdot \mathsf{in}\ \mathtt{(Triv)}]\!]_{\eta +\!\!+ env(B)} \\
&= [\![\mathtt{(Triv)}]\!]_{\eta +\!\!+ env(B)} \\
&= [\![\langle\rangle]\!]_{\eta +\!\!+ env(B)} \\
&= \mathbf{unit_E}\ (\mathsf{prod}_0 : \langle\rangle) \\
&\in \mathbf{E}\ \{\mathsf{prod}_0 : \langle\rangle\} \\
&= [\![\theta\ \mathtt{All\ Empty}]\!]
\end{aligned}
$$

as required.

case VAR ($f = $ `(_ && _)`): Let (a) be

$$\Delta \mid C \mid \Gamma \vdash \texttt{(_ \&\& _)} : (a \texttt{ -> All } b \texttt{ -> All } (a \texttt{ \# } b))[a \mapsto \tau, b \mapsto \rho] \hookrightarrow \mathsf{letw}\ w = W \ \mathsf{in}\ \texttt{(_ \&\& _)}\ w$$

Then by VAR

$$\Delta \vdash \tau : \mathsf{Type} \tag{e}$$

$$\Delta \vdash \rho : \mathsf{Row} \tag{f}$$

$$C \vdash^e w : (a\ \mathbf{\mathit{ins}}\ b)[a \mapsto \tau, b \mapsto \rho] \hookrightarrow w = W \tag{g}$$

By definition

$$\begin{aligned}
& [\![\mathsf{letw}\ w = W\ \mathsf{in}\ \texttt{(_ \&\& _)}\ w]\!]_{\eta +\!\!+ env(B)} \\
&= [\![\texttt{(_ \&\& _)}\ (w)]\!]_{env(w=W, \eta +\!\!+ env(B))} \\
&= [\![\texttt{(_ \&\& _)}\ (w)]\!]_{\eta +\!\!+ env(w=W, env(B))} \\
&= [\![(\lambda(w')\ .\ \lambda x\ .\ \lambda y\ .\ \mathsf{insert}\ x\ \mathsf{at}\ w'\ \mathsf{into}\ y)\ (w)]\!]_{\eta +\!\!+ env(w=W, env(B))} \\
&= [\![\lambda x\ .\ \lambda y\ .\ \mathsf{insert}\ x\ \mathsf{at}\ w\ \mathsf{into}\ y]\!]_{\eta +\!\!+ env(w=W, env(B))} \\
&= \ \mathbf{unit_E}\ (\mathsf{func} : \lambda x'\ . \\
&\qquad \mathbf{unit_E}\ (\mathsf{func} : \lambda y'\ . \\
&\qquad\quad \mathbf{let_E}\ v \leftarrow y' \\
&\qquad\quad \mathbf{in\ case}\ (v, [\![W]\!]_{env(B)})\ \mathbf{of}\ \{ \\
&\qquad\qquad (\mathsf{prod}_{n'} : \langle v'_1, \ldots, v'_{n'} \rangle, \mathsf{iind} : i) \rightarrow \\
&\qquad\qquad\quad \mathbf{unit_E}\ (\mathbf{if}\ 1 \le i \le (n'+1)\ \mathbf{then}\ v''\ \mathbf{else}\ \mathsf{wrong} : *); \\
&\qquad\qquad \mathbf{otherwise} \rightarrow \mathbf{unit_E}\ (\mathsf{wrong} : *)\ \})) \\
&\quad \text{where } v'' = \mathsf{prod}_{n+1} : \langle v'_1, \ldots, v'_{i-1}, x', v'_i, \ldots, v'_{n'} \rangle \\
&= (*)
\end{aligned}$$

By (c), (g) and Lemma B.14, if $\theta\ \rho = \texttt{(\#)}_n\ \overline{v'}\ \texttt{Empty}$ then

$$S' \neq \emptyset \wedge \forall \pi \in S'\ .\ \pi^{-1}\ 1 = j \wedge [\![W]\!]_{env(B)} = \mathsf{iind} : j$$

where $S' = \mathit{sortingPerms}(\theta\ \tau, v_1, \ldots, v_n)$.
Let $\pi \in S'$ and

$$\pi'\ i = \begin{cases} (\pi\ i) - 1, & \textbf{if } i < j \\ (\pi\ (i+1)) - 1, & \textbf{otherwise} \end{cases}$$

Then $\pi' \in \mathit{sortingPerms}(v_1, \ldots, v_n)$.

Thus

$$
\begin{aligned}
(*) = \;& \mathbf{unit_E}\ (\mathsf{func} : \lambda x' . \\
& \quad \mathbf{unit_E}\ (\mathsf{func} : \lambda y' . \\
& \quad\quad \mathbf{let_E}\ v \leftarrow y' \\
& \quad\quad \mathbf{in\ case}\ v\ \mathbf{of}\ \{ \\
& \quad\quad\quad \mathsf{prod}_n : \langle v_1', \ldots, v_n' \rangle \rightarrow \mathbf{unit_E}\ v''; \\
& \quad\quad\quad \mathbf{otherwise} \rightarrow \mathbf{unit_E}\ (\mathsf{wrong} : *)\ \}))
\end{aligned}
$$

$$
\begin{aligned}
& \text{where } v'' = \mathsf{prod}_{n+1} : \langle v_1', \ldots, v_{j-1}', x', v_j', \ldots, v_n' \rangle \\
& \in \mathbf{E} \left\{ \mathsf{func} : f \;\middle|\; \begin{array}{l} f \in \mathbf{E}\,\mathcal{V} \rightarrow \mathbf{E}\,\mathcal{V}, \\ v''' \in [\![\theta\ \tau]\!] \Longrightarrow f\ v''' \in S \end{array} \right\} \\
& \text{where } S = \mathbf{E} \left\{ \mathsf{func} : g \;\middle|\; \begin{array}{l} g \in \mathbf{E}\,\mathcal{V} \rightarrow \mathbf{E}\,\mathcal{V}, \\ v'''' \in T \Longrightarrow g\ v'''' \in U \end{array} \right\} \\
& \text{and } T = \mathbf{E}\,\{\mathsf{prod}_n : \langle v_1', \ldots, v_n' \rangle \mid v_1' \in [\![v_{\pi' 1}]\!], \ldots v_n' \in [\![v_{\pi' n}]\!]\} \\
& \text{and } U = \mathbf{E} \left\{ \mathsf{prod}_{n+1} : \langle v_1', \ldots, v_{j-1}', v''', v_j', \ldots, v_n' \rangle \;\middle|\; \begin{array}{l} v_1' \in [\![v_{\pi' 1}]\!], \ldots, \\ v_n' \in [\![v_{\pi' n}]\!] \end{array} \right\} \\
& = [\![\theta\ \tau\ \texttt{->}\ \texttt{All}\ (\theta\ \rho)\ \texttt{->}\ \texttt{All}\ (\theta\ \tau\ \texttt{\#}\ \theta\ \rho)]\!] \\
& = [\![\theta\ (\tau\ \texttt{->}\ \texttt{All}\ \rho\ \texttt{->}\ \texttt{All}\ (\tau\ \texttt{\#}\ \rho))]\!] \\
& = [\![\theta\ ((a\ \texttt{->}\ \texttt{All}\ b\ \texttt{->}\ \texttt{All}\ (a\ \texttt{\#}\ b))[a \mapsto \tau, b \mapsto \rho])]\!]
\end{aligned}
$$

as required.

case VAR (f = A): Let (newtype A = τ) $\in$ *tdecls* and let (a) be

$$\Delta \mid C \mid \Gamma \vdash \texttt{A} : (norm(\tau\ a_1 \ldots a_n)\ \texttt{->}\ \texttt{A}\ a_1 \ldots a_n)[\overline{a \mapsto v}] \hookrightarrow \mathsf{letw} \cdot \mathsf{in}\ \texttt{A}$$

Then by VAR

$$\Delta \vdash \overline{v : \kappa} \tag{e}$$

By well-kinding of τ, $\theta\ \tau = \tau$. Then

$$
\begin{aligned}
\theta\ (norm(\tau\ a_1 \ldots a_n)[\overline{a \mapsto v}]) &= \theta\ norm(\tau\ v_1 \ldots v_n) \\
&= norm(\tau\ (\theta\ v_1) \ldots (\theta\ v_n))
\end{aligned}
$$

By definition

$$
\begin{aligned}
& [\![\mathsf{letw} \cdot \mathsf{in}\ \texttt{A}]\!]_{\eta +\!\!+ env(B)} \\
& = [\![\texttt{A}]\!]_{\eta +\!\!+ env(B)} \\
& = [\![\lambda x\ .\ A\ x]\!]_{\eta +\!\!+ env(B)} \\
& = \mathbf{unit_E}\ (\mathsf{func} : \lambda y\ .\ \mathbf{fold}_A\ y) \\
& \in \mathbf{E} \left\{ \mathsf{func} : f \;\middle|\; \begin{array}{l} f \in \mathbf{E}\,\mathcal{V} \rightarrow \mathbf{E}\,\mathcal{V}, \\ v \in [\![norm(\tau\ (\theta\ v_1) \ldots (\theta\ v_n))]\!] \Longrightarrow f\ v \in [\![\texttt{A}\ (\theta\ v_1) \ldots (\theta\ v_n)]\!] \end{array} \right\} \\
& = [\![norm(\tau\ (\theta\ v_1) \ldots (\theta\ v_n)) \rightarrow \texttt{A}\ (\theta\ v_1) \ldots (\theta\ v_n)]\!] \\
& = [\![\theta\ (norm(\tau\ v_1 \ldots v_n)\ \texttt{->}\ \texttt{A}\ v_1 \ldots v_n)]\!] \\
& = [\![\theta\ ((norm(\tau\ a_1 \ldots a_n)\ \texttt{->}\ \texttt{A}\ a_1 \ldots a_n)[\overline{a \mapsto v}])]\!]
\end{aligned}
$$

as required.
case ABS: Let (a) be

$$\Delta \mid C \mid \Gamma \vdash \{abs\} : \tau \hookrightarrow T[\mathsf{undefined}]$$

Then by ABS

$$\Delta \mid C \mid \Gamma \vdash_1 abs : \tau \hookrightarrow T[\bullet] \tag{e}$$

Notice

$$\begin{aligned} [\![\mathsf{undefined}]\!]_{\eta +\!\!+ env(B)} &= \bot \\ &\in [\![\theta\ \tau]\!] \end{aligned}$$

Then by I.H. (ii) on (e)

$$[\![T[\mathsf{undefined}]]\!]_{\eta +\!\!+ env(B)} \in [\![\theta\ \tau]\!]$$

as required.
case DISC: Let (a) be

$$\Delta \mid C \mid \Gamma \vdash \{abs_1, \ldots, abs_{n+1}\} : \tau \hookrightarrow \mathsf{let}\ z = U\ \mathsf{in}\ T[z]$$

Then by DISC

$$\Delta \mid C \mid \Gamma \vdash_1 abs_1 : \tau \hookrightarrow T[\bullet] \tag{e}$$
$$\Delta \mid C \mid \Gamma \vdash \{abs_2, \ldots, abs_{n+1}\} : \tau' \hookrightarrow U \tag{f}$$
$$C \vdash^e \tau\ eq\ \tau' \hookrightarrow \mathsf{True} \tag{g}$$

By (c), (g) and Lemma B.14 $eq_{\emptyset}^{m}(\theta\ \tau, \theta\ \tau')$, hence

$$[\![\theta\ \tau]\!] = [\![\theta\ \tau']\!]$$

By definition

$$\begin{aligned} & [\![\mathsf{let}\ z = U\ \mathsf{in}\ T[z]]\!]_{\eta +\!\!+ env(B)} \\ = {} & [\![T[z]]\!]_{\eta +\!\!+ env(B), z \mapsto v} \\ & \text{where } v = [\![U]\!]_{\eta +\!\!+ env(B)} \\ = {} & [\![T[U]]\!]_{\eta +\!\!+ env(B)} \\ = {} & (*) \end{aligned}$$

(Notice the translation let-binds U so as to avoid duplicating it within the body of T. Since our semantics is call-by-name, we may safely undo this.)
By I.H. (i) on (f)

$$\begin{aligned} [\![U]\!]_{\eta +\!\!+ env(B)} &\in [\![\theta\ \tau']\!] \\ &= [\![\theta\ \tau]\!] \end{aligned}$$

Then, by I.H. (ii) on (e)

$$(*) \in [\![\theta\ \tau]\!]$$

as required.

case LET: Let (a) be

$$\Delta \mid C \mid \Gamma \vdash \texttt{let}\ x = u\ \texttt{in}\ t : \tau \hookrightarrow \mathsf{let}\ x = \lambda names(D_2)\ .\ U\ \mathsf{in}\ T$$

Then by LET

$$x \in fv(t) \tag{e}$$
$$\Delta \vdash D_1\ \mathsf{constraint} \tag{f}$$
$$\Delta +\!\!+ \Delta' \vdash D_2\ \mathsf{constraint} \tag{g}$$
$$D_1 = inhs(C) \tag{h}$$
$$saturate(D_1 +\!\!+ D_2) \neq \emptyset \tag{j}$$
$$\Delta +\!\!+ \Delta' \mid D_1 +\!\!+ D_2 \mid \Gamma \vdash u : v \hookrightarrow U \tag{k}$$
$$\Delta \mid C \mid \Gamma, x : \sigma \vdash t : \tau \hookrightarrow T \tag{l}$$

where $\sigma = \texttt{forall}\ \overline{a : \kappa}\ .\ anon(D_2)\ \texttt{=>}\ v$, $names(D_2) = (w_1, \ldots, w_n)$, and $\Delta' = \overline{a : \kappa}$. By definition

$$\begin{aligned} & [\![\mathsf{let}\ x = \lambda\ names(D_2)\ .\ U\ \mathsf{in}\ T]\!]_{\eta +\!\!+ env(B)} \\ = {} & [\![T]\!]_{\eta +\!\!+ env(B) +\!\!+ x \mapsto v} \\ = {} & (*) \end{aligned}$$

where

$$\begin{aligned} v &= [\![\lambda names(D_2)\ .\ U]\!]_{\eta +\!\!+ env(B)} \\ &= \mathbf{unit_E}\ (\mathsf{ifunc}_n : \lambda(y_1, \ldots, y_n)\ .\ [\![U]\!]_{\eta +\!\!+ env(B), w_1 \mapsto y_1, \ldots, w_n \mapsto y_n}) \end{aligned}$$

Since by (b) $dom(\theta) \cap dom(\Delta') = \emptyset$, by (f)

$$\Delta' \vdash \theta\ D_2\ \mathsf{constraint}$$

By (e) and Lemma B.36 (ii) there exists $\vdash \theta' : \Delta' \to \Delta$ s.t. $C \vdash^e \theta'\ D_2 \hookrightarrow B'$. Then by Lemma B.13 $env(B +\!\!+ B') \models \theta \circ \theta'\ D_2$. By (b) this is equivalent to

$$env(B +\!\!+ B') \models (\theta \circ \theta')_{\restriction dom(\Delta')}\ (\theta\ D_2)$$

where $\vdash (\theta \circ \theta')_{\restriction dom(\Delta')} : \Delta' \to \Delta_{init}$.

Now let θ'' and B'' be s.t.

$$\vdash \theta'' : \Delta' \to \Delta_{init} \wedge env(B'') \models \theta''\ (\theta\ D_2) \tag{m}$$

By above argument at least one such θ'' and B'' exists.

Then, since $\theta'' \circ \theta\ D_1 = \theta\ D_1$, by (c) and (h) we have

$$env(B) +\!\!+ env(B'') \models \theta'' \circ \theta\ (D_1 +\!\!+ D_2)$$

and so by I.H. (i) on (k)

$$\begin{aligned}[\![U]\!]_{\eta +\!\!+ env(B +\!\!+ B'')} &= [\![U]\!]_{\eta +\!\!+ env(B) +\!\!+ env(B'')} \\ &\in [\![\theta'' \ (\theta\ v)]\!]\end{aligned}$$

Since this holds for any choice of θ'' and B'' s.t. (m) holds

$$\begin{aligned} v \in \bigcap \left\{ S_{(\theta'', B'')} \;\middle|\; \begin{array}{l} \vdash \theta'' : \Delta' \to \Delta_{init}, \\ env(B'') \models \theta''\ (\theta\ D_2) \end{array} \right\} \\ = [\![\texttt{forall}\ \overline{a : \kappa}\ .\ (\theta\ anon(D_2))\ \texttt{=>}\ (\theta\ v)]\!] \\ = [\![\theta\ (\texttt{forall}\ \overline{a : \kappa}\ .\ anon(D_2)\ \texttt{=>}\ v)]\!] \end{aligned}$$

where

$$S_{(\theta'', B'')} = \mathbf{E} \left\{ \mathsf{ifunc}_n : f \;\middle|\; \begin{array}{l} f \in \prod_{1 \le i \le n} \mathcal{I} \to \mathbf{E}\ \mathcal{V}, \\ f\ ([\![w_1]\!]_{env(B'')}, \ldots, [\![w_n]\!]_{env(B'')}) \\ \quad \in [\![\theta''\ (\theta\ v)]\!] \end{array} \right\}$$

Now, let $\eta' = \eta, x \mapsto v$. Then $\eta' \models (\theta\,\Gamma), x : (\theta\,\sigma)$. Thus by I.H. (i) on (l)

$$(*) \in [\![\theta\ \tau]\!]$$

as required.

Notice (h) and (j) play no part in soundness. The former is always true in λ^{TIR} as presented, and the latter serves only to detect unsatisfiability as early as possible.

case P1: Let (a) be

$$\Delta \mid C \mid \Gamma \vdash_0 t : \tau \hookrightarrow T$$

Then by P1

$$\Delta \mid C \mid \Gamma \vdash t : \tau \hookrightarrow T$$

which by I.H. (i) implies $[\![T]\!]_{\eta +\!\!+ env(B)} \in [\![\theta\ \tau]\!]$. Since T has no "hole", the result is immediate.

case P2: Let (a) be

$$\Delta \mid C \mid \Gamma \vdash_{n+1} \texttt{\textbackslash}i\ .\ t : \texttt{Int -> } \tau \hookrightarrow \lambda x\ .\ \mathsf{case}\ x\ \mathsf{of}\ \{\ i \to T[\bullet\ x];\ \mathsf{otherwise} \to \bullet\ x\ \}$$

Then by P2 $\Delta \mid C \mid \Gamma \vdash_n t : \tau \hookrightarrow T[\bullet]$, and since $x \notin dom(\Gamma)$, by Lemma B.38

$$\Delta \mid C \mid \Gamma, x : \texttt{Int} \vdash_n t : \tau \hookrightarrow T[\bullet] \tag{f}$$

Let v be s.t.

$$\begin{aligned} v &\in [\![\texttt{Int}]\!] \\ &= \mathbf{E}\ \{\mathsf{int} : i \mid i \in \mathcal{Z}\} \end{aligned} \tag{g}$$

and let $\eta' = \eta, x \mapsto v$. Then by (d) $\eta' \models (\theta\,\Gamma), x : \texttt{Int}$.

By (e)

$$\llbracket U \rrbracket_{\eta + env(B)} \in \llbracket \texttt{Int -> } \theta\ \tau \rrbracket = \mathbf{E}\ \{\mathsf{func} : f \mid f \in \mathbf{E}\ \mathcal{V} \to \mathbf{E}\ \mathcal{V}, v' \in \llbracket \mathbf{Int} \rrbracket \Longrightarrow f\ v' \in \llbracket \theta\ \tau \rrbracket \}$$

Thus

$$\begin{aligned} & \llbracket U\ x \rrbracket_{\eta' + env(B)} \\ = \ & \mathbf{let_E}\ v' \leftarrow \llbracket U \rrbracket_{\eta' + env(B)} \\ & \mathbf{in\ case}\ v'\ \mathbf{of}\ \{ \\ & \quad \mathsf{func} : f \to f\ \llbracket x \rrbracket_{\eta' + env(B)}; \\ & \quad \mathbf{otherwise} \to \mathbf{unit_E}\ (\mathsf{wrong} : *)\ \} \\ = \ & \mathbf{let_E}\ \mathsf{func} : f \leftarrow \llbracket U \rrbracket_{\eta + env(B)} \\ & \mathbf{in}\ f\ v \\ \in \ & \llbracket \theta\ \tau \rrbracket \end{aligned} \quad \text{(h)}$$

Then by I.H. (ii) (using η') on (f)

$$\llbracket T[U\ x] \rrbracket_{\eta' + env(B)} \in \llbracket \theta\ \tau \rrbracket \quad \text{(i)}$$

By definition

$$\begin{aligned} & \llbracket \lambda x\ .\ \mathsf{case}\ x\ \mathsf{of}\ \{\ i \to T[U\ x];\ \mathsf{otherwise} \to U\ x\ \} \rrbracket_{\eta + env(B)} \\ = \ & \mathbf{unit_E}\ (\mathsf{func} : \lambda x'\ .\ \ \mathbf{let_E}\ v' \leftarrow \llbracket x \rrbracket_{\eta + env(B), x \mapsto x'} \\ & \qquad\qquad\qquad\quad \mathbf{in\ case}\ v'\ \mathbf{of}\ \{ \\ & \qquad\qquad\qquad\qquad \mathsf{int} : j \to\ \ \mathbf{if}\ i = j\ \mathbf{then}\ \llbracket T[U\ x] \rrbracket_{\eta + env(B), x \mapsto x'} \\ & \qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ \llbracket U\ x \rrbracket_{\eta + env(B), x \mapsto x'}; \\ & \qquad\qquad\qquad\qquad \mathbf{otherwise} \to \mathbf{unit_E}\ (\mathsf{wrong} : *)\ \}\ ) \\ = \ & (*) \end{aligned}$$

Then, since the choice of v was arbitrary s.t. (g) holds, by (h) and (i)

$$\begin{aligned} (*) & \in \mathbf{E}\ \{\mathsf{func} : f \mid f \in \mathbf{E}\ \mathcal{V} \to \mathbf{E}\ \mathcal{V}, v'' \in \llbracket \mathbf{Int} \rrbracket \Longrightarrow f\ v'' \in \llbracket \theta\ \tau \rrbracket \} \\ & = \llbracket \texttt{Int -> } \theta\ \tau \rrbracket \\ & = \llbracket \theta\ (\texttt{Int -> } \tau) \rrbracket \end{aligned}$$

as required.

case P3: Let (a) be

$$\begin{aligned} & \Delta \mid C \mid \Gamma \vdash_{n+1} \backslash A\ p\ .\ t : A\ v_1 \ldots v_n \texttt{ -> } \tau \\ & \hookrightarrow \lambda x\ .\ \mathsf{let}\ y = A^{-1}\ x\ \mathsf{in}\ T[\lambda y\ .\ \bullet\ (A\ y)]\ y \end{aligned}$$

Then by P3

$$\Delta \vdash \overline{v : \kappa} \quad \text{(f)}$$

$$C \vdash^e norm(v'\ v_1 \ldots v_n)\ eq\ \tau' \hookrightarrow \mathsf{True} \quad \text{(g)}$$

$$\Delta \mid C \mid \Gamma \vdash_{n+1} \backslash p\ .\ t : \tau' \texttt{ -> } \tau \hookrightarrow T[\bullet] \quad \text{(h)}$$

Notice by well-kinding of v'

$$\begin{aligned}\theta\ norm(v'\ v_1 \ldots v_n) &= norm((\theta\ v')\ (\theta\ v_1) \ldots (\theta\ v_n)) \\ &= norm(v'\ (\theta\ v_1) \ldots (\theta\ v_n))\end{aligned}$$

By (c), (g) and Lemma B.14 $eq^m_\emptyset(\theta\ norm(v'\ v_1 \ldots v_n), \theta\ \tau')$, and thus

$$[\![norm(v'\ (\theta\ v_1) \ldots (\theta\ v_n))]\!] = [\![\theta\ \tau']\!] \tag{i}$$

By (e)

$$\begin{aligned}[\![U]\!]_{\eta +\!\!+ env(B)} &\in [\![\theta\ ((A\ v_1 \ldots v_n)\ \texttt{->}\ \tau)]\!] \\ &= [\![(A\ (\theta\ v_1) \ldots (\theta\ v_n))\ \texttt{->}\ (\theta\ \tau)]\!] \\ &= \mathbf{E}\left\{ \mathsf{func} : f \;\middle|\; \begin{array}{l} f \in \mathbf{E}\,\mathcal{V} \to \mathbf{E}\,\mathcal{V}, \\ v' \in [\![(A\ (\theta\ v_1) \ldots (\theta\ v_n))]\!] \Longrightarrow f\ v' \in [\![\theta\ \tau]\!] \end{array} \right\}\end{aligned}$$

Then by definition

$$\begin{aligned}&[\![\lambda y\ .\ U\ (A\ y)]\!]_{\eta +\!\!+ env(B)} \\ &= \mathbf{unit_E}\ (\mathsf{func} : \lambda y'\ .\ \begin{array}[t]{l} \mathbf{let_E}\ v \leftarrow [\![U]\!]_{\eta +\!\!+ env(B), y \mapsto y'} \\ \mathbf{in\ case}\ v\ \mathbf{of}\ \{ \\ \quad \mathsf{func} : f \to f\ (\mathbf{fold}_A\ [\![y]\!]_{\eta +\!\!+ env(B), y \mapsto y'}); \\ \quad \mathbf{otherwise} \to \mathbf{unit_E}\ (\mathsf{wrong} : *)\ \}\ ) \end{array} \\ &= \mathbf{unit_E}\ (\mathsf{func} : \lambda y'\ .\ \begin{array}[t]{l} \mathbf{let_E}\ \mathsf{func} : f \leftarrow [\![U]\!]_{\eta +\!\!+ env(B)} \\ \mathbf{in}\ f\ (\mathbf{fold}_A\ y')\ ) \end{array} \\ &\in \mathbf{E}\ \{\mathsf{func} : g \mid g \in \mathbf{E}\,\mathcal{V} \to \mathbf{E}\,\mathcal{V}, v'' \in [\![norm(v'\ (\theta\ v_1) \ldots (\theta\ v_n))]\!] \Longrightarrow g\ v'' \in [\![\theta\ \tau]\!]\} \\ &= [\![norm(v'\ (\theta\ v_1) \ldots (\theta\ v_n))\ \texttt{->}\ \theta\ \tau]\!] \\ &= [\![\theta\ \tau'\ \texttt{->}\ \theta\ \tau]\!] \\ &= [\![\theta\ (\tau'\ \texttt{->}\ \tau)]\!]\end{aligned}$$

Thus by I.H. (ii) on (h)

$$[\![T[\lambda y\ .\ U\ (A\ y)]]\!]_{\eta +\!\!+ env(B)} \in [\![\theta\ (\tau'\ \texttt{->}\ \tau)]\!] \tag{j}$$

By definition

$$\begin{aligned}
&[\![\lambda x \;.\; \mathsf{let}\; y = A^{-1}\; x\; \mathsf{in}\; T[\lambda y \;.\; U\;(A\; y)]\; y]\!]_{\eta +\!\!+ env(B)} \\
&= \mathbf{unit_E}\;(\mathsf{func} : \lambda x' \;.\;\; \mathbf{let}\; v \leftarrow [\![T[\lambda y \;.\; U\;(A\; y)]]\!]_{\eta +\!\!+ env(B), x \mapsto x', y \mapsto \mathbf{unfold}_A\; x'} \\
&\qquad\qquad\qquad\qquad\quad \mathbf{in}\;\; \mathbf{case}\; v\; \mathbf{of}\; \{ \\
&\qquad\qquad\qquad\qquad\qquad \mathsf{func} : f \to f\; \mathbf{unfold}_A\; x'; \\
&\qquad\qquad\qquad\qquad\qquad \mathbf{otherwise} \to \mathbf{unit_E}\;(\mathsf{wrong} : *)\;\}\;) \\
&= \mathbf{unit_E}\;(\mathsf{func} : \lambda x' \;.\;\; \mathbf{let}\; \mathsf{func} : f \leftarrow [\![T[\lambda y \;.\; U\;(A\; y)]]\!]_{\eta +\!\!+ env(B)} \\
&\qquad\qquad\qquad\qquad\quad \mathbf{in}\; f\;(\mathbf{unfold}_A\; x')\;) \\
&\in \mathbf{E}\;\{\mathsf{func} : g \mid g \in \mathbf{E}\,\mathcal{V} \to \mathbf{E}\,\mathcal{V}, v' \in [\![A\;(\theta\, v_1) \ldots (\theta\, v_n)]\!] \Longrightarrow f\; v' \in [\![\theta\; \tau]\!]\} \\
&= [\![A\;(\theta\, v_1) \ldots (\theta\, v_n)\; \texttt{->}\; (\theta\; \tau)]\!] \\
&= [\![\theta\;((A\; v_1 \ldots v_n)\; \texttt{->}\; \tau)]\!]
\end{aligned}$$

as required.

case P4: Let (a) be

$$\begin{aligned}
&\Delta \mid C \mid \Gamma \vdash_{n+1} \texttt{\textbackslash Inj}\; p\; .\; t : \texttt{One}\;(v\;\texttt{\#}\;\rho)\;\texttt{->}\;\tau \\
&\hookrightarrow \lambda x \;.\; \mathsf{case}\; x\; \mathsf{of}\; \{\mathsf{Inj}\; W\; y\; \to T[\lambda y \;.\; \bullet\;(\mathsf{Inj}\; W\; y)]\; y; \\
&\qquad\qquad\qquad\qquad \mathsf{otherwise} \to \bullet\; x\}
\end{aligned}$$

Then by P4

$$\Delta \mid C \mid \Gamma \vdash_{n+1} \texttt{\textbackslash}p\; .\; t : v\;\texttt{->}\;\tau \hookrightarrow T[\bullet] \tag{f}$$

$$C \vdash^e v\; \mathit{ins}\; \rho \hookrightarrow W \tag{g}$$

$$\Delta \vdash \rho : \texttt{Row}$$

By (c), (g) and Lemma B.14, if $\theta\, \rho = (\texttt{\#})_n\; \overline{v'}\; \texttt{Empty}$ then

$$S' \neq \emptyset \wedge \forall \pi \in S' \;.\; \pi^{-1}\; 1 = j \wedge [\![W]\!]_{env(B)} = \mathsf{iint} : j \tag{h}$$

where $S' = \mathit{sortingPerms}(\theta\; v, v'_1, \ldots, v'_n)$.
Let $\pi \in S'$. By (e)

$$\begin{aligned}
[\![U]\!]_{\eta +\!\!+ env(B)} &\in [\![\theta\;(\texttt{One}\;(v\;\texttt{\#}\;\rho)\;\texttt{->}\;\tau)]\!] \\
&\in [\![\texttt{One}\;(\theta\; v\;\texttt{\#}\;\theta\;\rho)\;\texttt{->}\;\theta\;\tau]\!] \\
&= \mathbf{E}\;\{\mathsf{func} : f \mid f \in \mathbf{E}\,\mathcal{V} \to \mathbf{E}\,\mathcal{V}, v' \in S \Longrightarrow f\; v' \in [\![\theta\;\tau]\!]\}
\end{aligned} \tag{i}$$

where

$$S = \mathbf{E}\left\{ \mathsf{inj} : \langle i, v' \rangle \;\middle|\; \begin{array}{l} 1 \le i \le n, \\ \mathbf{if}\; i = j\; \mathbf{then}\; v' \in [\![\theta\; v]\!] \\ \mathbf{else}\; v' \in [\![v'_{(\pi\; i)-1}]\!] \end{array} \right\}$$

Then by (h) and (i)

$$\begin{aligned}
&[\![\lambda y\ .\ U\ (\mathsf{Inj}\ W\ y)]\!]_{\eta +\!\!+ env(B)} \\
&= \mathbf{unit_E}\ (\mathsf{func} : \lambda y'\ .\ \mathbf{let_E}\ v \leftarrow [\![U]\!]_{\eta +\!\!+ env(B)} \\
&\qquad\qquad \mathbf{in\ case}\ v\ \mathbf{of}\ \{ \\
&\qquad\qquad\quad \mathsf{func} : f \to f\ (\ \mathbf{case}\ [\![W]\!]_\eta\ \mathbf{of}\ \{ \\
&\qquad\qquad\qquad\qquad \mathsf{iind} : i \to \mathbf{unit_E}\ (\mathsf{inj} : \langle i, y' \rangle); \\
&\qquad\qquad\qquad\qquad \mathbf{otherwise} \to \mathbf{unit_E}\ (\mathsf{wrong} : *)\ \}\ ) \\
&\qquad\qquad\quad \mathbf{otherwise} \to \mathbf{unit_E}\ (\mathsf{wrong} : *)\ \}\ ) \\
&= \mathbf{unit_E}\ (\mathsf{func} : \lambda y'\ .\ \mathbf{let_E}\ \mathsf{func} : f \leftarrow [\![U]\!]_{\eta +\!\!+ env(B)} \\
&\qquad\qquad \mathbf{in}\ f\ (\mathbf{unit_E}\ (\mathsf{inj} : \langle j, y' \rangle)))\ ) \\
&\in \mathbf{E}\ \{\mathsf{func} : g \mid g \in \mathbf{E}\ \mathcal{V} \to \mathbf{E}\ \mathcal{V}, v' \in [\![\theta\ v]\!] \Longrightarrow g\ v' \in [\![\theta\ \tau]\!]\} \\
&= [\![\theta\ v\ \texttt{->}\ \theta\ \tau]\!] \\
&= [\![\theta\ (v\ \texttt{->}\ \tau)]\!]
\end{aligned}$$

Then by I.H. (ii) on (f)

$$[\![T[\lambda y\ .\ U\ (\mathsf{Inj}\ W\ y)]]\!]_{\eta +\!\!+ env(B)} \in [\![\theta\ (v\ \texttt{->}\ \tau)]\!] \tag{j}$$

Let v'' be s.t.

$$v'' \in [\![\texttt{One}\ (\theta\ v\ \texttt{\#}\ \theta\ \rho)]\!] \tag{k}$$

and let $\eta' = \eta, x \mapsto v''$. Then by (d) $\eta' \models (\theta\ \Gamma), x : \theta\ (\texttt{One}\ (v\ \texttt{\#}\ \rho))$.
By definition

$$\begin{aligned}
&[\![U\ x]\!]_{\eta' +\!\!+ env(B)} \\
=\ & \mathbf{let_E}\ v \leftarrow [\![U]\!]_{\eta' +\!\!+ env(B)} \\
& \mathbf{in\ case}\ v\ \mathbf{of}\ \{ \\
& \quad \mathsf{func} : f \to f\ [\![x]\!]_{\eta' +\!\!+ env(B)}; \\
& \quad \mathbf{otherwise} \to \mathbf{unit_E}\ (\mathsf{wrong} : *)\ \} \\
=\ & \mathbf{let_E}\ \mathsf{func} : f \leftarrow [\![U]\!]_{\eta +\!\!+ env(B)} \\
& \mathbf{in}\ f\ v \\
\in\ & [\![\theta\ \tau]\!]
\end{aligned} \tag{l}$$

Since x is fresh, by (f) and Lemma B.38

$$\Delta \mid C \mid \Gamma, x : \texttt{One}\ (v\ \texttt{\#}\ \rho) \vdash_{n+1} \backslash p\ .\ t : v\ \texttt{->}\ \tau \hookrightarrow T[\bullet]$$

Thus by (l) and I.H. (ii) (using η')

$$[\![T[U\ x]]\!]_{\eta' +\!\!+ env(B)} \in [\![\theta\ \tau]\!]$$

By (h) and by definition

$$
\begin{aligned}
&[\![\lambda x \,.\, \mathsf{case}\ x\ \mathsf{of}\ \{\ \mathsf{Inj}\ W\ y\ \to T[\lambda y \,.\, U\ (\mathsf{Inj}\ W\ y)]\ y; \\
&\qquad\qquad\qquad \mathsf{otherwise} \to U\ x\}]\!]_{\eta +\!\!+ env(B)} \\
&= \mathbf{unit_E}\ (\mathsf{func} : \lambda x' \,.\ \mathbf{let_E}\ v \leftarrow [\![x]\!]_{\eta +\!\!+ env(B), x \mapsto x'} \\
&\qquad \mathbf{in\ case}\ (v, [\![W]\!]_\eta)\ \mathbf{of}\ \{ \\
&\qquad\quad (\mathsf{inj} : \langle j', v' \rangle, \mathsf{iind} : i) \to \\
&\qquad\qquad \mathbf{if}\ i = j'\ \mathbf{then} \\
&\qquad\qquad\quad \mathbf{let_E}\ v'' \leftarrow [\![T[\lambda y \,.\, U\ (\mathsf{Inj}\ W\ y)]]\!]_{\eta +\!\!+ env(B), x \mapsto x', y \mapsto v'} \\
&\qquad\qquad\quad \mathbf{in\ case}\ v''\ \mathbf{of}\ \{ \\
&\qquad\qquad\qquad \mathsf{func} : f \to f\ [\![y]\!]_{\eta +\!\!+ env(B), x \mapsto x', y \mapsto v'}; \\
&\qquad\qquad\qquad \mathbf{otherwise} \to \mathbf{unit_E}\ (\mathsf{wrong} : *)\ \} \\
&\qquad\qquad \mathbf{else}\ [\![U\ x]\!]_{\eta +\!\!+ env(B), x \mapsto x', y \mapsto v'}; \\
&\qquad\quad \mathbf{otherwise} \to \mathbf{unit_E}\ (\mathsf{wrong} : *)\ \} \\
&= \mathbf{unit_E}\ (\mathsf{func} : \lambda x' \,.\ \mathbf{let_E}\ v \leftarrow x' \\
&\qquad \mathbf{in\ case}\ v\ \mathbf{of}\ \{ \\
&\qquad\quad \mathsf{inj} : \langle j', v' \rangle \to \\
&\qquad\qquad \mathbf{if}\ j = j'\ \mathbf{then} \\
&\qquad\qquad\quad \mathbf{let_E}\ \mathsf{func} : f \leftarrow [\![T[\lambda y \,.\, U\ (\mathsf{Inj}\ W\ y)]]\!]_{\eta +\!\!+ env(B)} \\
&\qquad\qquad\quad \mathbf{in}\ f\ v' \\
&\qquad\qquad \mathbf{else}\ [\![U\ x]\!]_{\eta +\!\!+ env(B), x \mapsto x'}; \\
&\qquad\quad \mathbf{otherwise} \to \mathbf{unit_E}\ (\mathsf{wrong} : *)\ \} \\
&= (*)
\end{aligned}
$$

Then since the choice of v'' was arbitrary s.t. (k) holds, by (j) and (l)

$$
\begin{aligned}
(*) &\in \mathbf{E}\ \{\mathsf{func} : f \mid f \in \mathbf{E}\ \mathcal{V} \to \mathbf{E}\ \mathcal{V}, v'' \in [\![\mathtt{One}\ (\theta\ \upsilon\ \texttt{\#}\ \theta\ \rho)]\!] \Longrightarrow f\ v'' \in [\![\theta\ \tau]\!]\} \\
&= [\![\mathtt{One}\ (\theta\ \upsilon\ \texttt{\#}\ \theta\ \rho)\ \texttt{->}\ (\theta\ \tau)]\!] \\
&= [\![\theta\ (\mathtt{One}\ (\upsilon\ \texttt{\#}\ \rho)\ \texttt{->}\ \tau)]\!]
\end{aligned}
$$

as required.

case P5: Let (a) be

$$
\begin{aligned}
&\Delta \mid C \mid \Gamma \vdash_{n+1} \texttt{\textbackslash}p\ \texttt{\&\&}\ q\ .\ t : \mathtt{All}\ (\upsilon_1\ \texttt{\#}\ \rho)\ \texttt{->}\ \tau \\
&\hookrightarrow \lambda x \,.\, \mathsf{let}\ y\ z = \mathsf{remove}\ W\ \mathsf{from}\ x \\
&\qquad\quad \mathsf{in}\ T[\lambda y \,.\, \lambda z \,.\, \bullet\ (\mathsf{insert}\ y\ \mathsf{at}\ W\ \mathsf{into}\ z)]\ y\ z
\end{aligned}
$$

Then by P5

$$\Delta \mid C \mid \Gamma \vdash_{n+2} \texttt{\textbackslash}p\ .\ \texttt{\textbackslash}q\ .\ t : \upsilon_1\ \texttt{->}\ \upsilon_2\ \texttt{->}\ \tau \hookrightarrow T[\bullet] \tag{f}$$

$$C \vdash^e \mathtt{All}\ \rho\ eq\ \upsilon_2 \hookrightarrow \mathsf{True} \tag{g}$$

$$C \vdash^e \upsilon_1\ ins\ \rho \hookrightarrow W \tag{h}$$

$$\Delta \vdash \rho : \mathtt{Row}$$

By (c), (g) and Lemma B.14, $eq^m_\emptyset(\mathtt{All}\ (\theta\ \rho), \theta\ \upsilon_2)$, and thus

$$[\![\mathtt{All}\ (\theta\ \rho)]\!] = [\![\theta\ \upsilon_2]\!] \tag{i}$$

Similarly, by (c), (h) and Lemma B.14, if $\theta\ \rho = \texttt{(\#)}_n\ \overline{v'}\ \texttt{Empty}$ then

$$S' \neq \emptyset \wedge \forall \pi \in S' \ .\ \pi^{-1}\ 1 = j \wedge [\![W]\!]_{env(B)} = \mathsf{iind} : j \tag{j}$$

where $S' = sortingPerms(\theta\ v_1, v'_1, \ldots, v'_n)$.
Let $\pi \in S'$, and let

$$\pi'\ i = \begin{cases} (\pi\ i) - 1, & \textbf{if } i < j \\ (\pi\ (i+1)) - 1, & \textbf{otherwise} \end{cases}$$

Then $\pi' \in sortingPerms(v'_1, \ldots, v'_n)$.
By (e)

$$\begin{aligned} [\![U]\!]_{\eta +\!\!+ env(B)} &\in [\![\theta\ (\texttt{All}\ (v_1\ \texttt{\#}\ \rho)\ \texttt{->}\ \tau)]\!] \\ &= [\![\texttt{All}\ (\theta\ v_1\ \texttt{\#}\ \theta\ \rho)\ \texttt{->}\ (\theta\ \tau)]\!] \\ &= \mathbf{E}\ \{\mathsf{func} : f \mid f \in \mathbf{E}\ \mathcal{V} \to \mathbf{E}\ \mathcal{V}, v' \in [\![\texttt{All}\ (\theta\ v_1\ \texttt{\#}\ \theta\ \rho)]\!] \Longrightarrow f\ v' \in [\![\theta\ \tau]\!]\} \end{aligned} \tag{k}$$

By definition

$$\begin{aligned} & [\![\lambda y\ .\ \lambda z\ .\ U\ (\mathsf{insert}\ y\ \mathsf{at}\ W\ \mathsf{into}\ z)]\!]_{\eta +\!\!+ env(B)} \\ = \ & \mathbf{unit_E}\ (\lambda y'\ .\ \mathbf{unit_E}\ (\lambda z'\ . \\ & \quad \mathbf{let_E}\ v \leftarrow [\![U]\!]_{\eta +\!\!+ env(B), y \mapsto y', z \mapsto z'} \\ & \quad \mathbf{in\ case}\ v\ \mathbf{of}\ \{ \\ & \qquad \mathsf{func} : f \to f\ (\ \mathbf{let_E}\ v' \leftarrow [\![z]\!]_{\eta +\!\!+ env(B), y \mapsto y', z \mapsto z'} \\ & \qquad\qquad \mathbf{in\ case}\ (v', [\![W]\!]_{\eta +\!\!+ env(B), y \mapsto y', z \mapsto z'})\ \mathbf{of}\ \{ \\ & \qquad\qquad\quad (\mathsf{prod}_n : \langle v''_1,\ \ldots, v''_n\rangle, \mathsf{iind} : i) \to \\ & \qquad\qquad\qquad \mathbf{unit_E}\ (\mathbf{if}\ 1 \leq i \leq n+1\ \mathbf{then}\ v'''\ \mathsf{else}\ \mathsf{wrong} : *); \\ & \qquad\qquad\quad \mathbf{otherwise} \to \mathbf{unit_E}\ (\mathsf{wrong} : *)\ \}\ ); \\ & \qquad \mathbf{otherwise} \to \mathbf{unit_E}\ (\mathsf{wrong} : *)\ \}\ )\ ) \\ = \ & (*) \end{aligned}$$

where

$$\begin{aligned} v''' &= \mathsf{prod}_{n+1} : \langle v''_1, \ldots, v''_{i-1}, [\![y]\!]_{\eta +\!\!+ env(B), y \mapsto y', z \mapsto z'}, v''_i, \ldots, v''_n\rangle \\ &= \mathsf{prod}_{n+1} : \langle v''_1, \ldots, v''_{i-1}, y', v''_i, \ldots, v''_n\rangle \end{aligned}$$

Then by (j) and (k)

$$\begin{aligned} (*) = \ & \mathbf{unit_E}\ (\lambda y'\ .\ \mathbf{unit_E}\ (\lambda z'\ . \\ & \quad \mathbf{let_E}\ \mathsf{func} : f \leftarrow [\![U]\!]_{\eta +\!\!+ env(B)} \\ & \quad \mathbf{in}\ f\ (\ \mathbf{let_E}\ v' \leftarrow z' \\ & \qquad \mathbf{in\ case}\ v'\ \mathbf{of}\ \{ \\ & \qquad\quad \mathsf{prod}_n : \langle v''_1, \ldots, v''_n\rangle \to \\ & \qquad\qquad \mathbf{unit_E}\ (\mathsf{prod}_{n+1} : \langle v''_1, \ldots, v''_{j-1}, y', v''_j, \ldots, v''_n\rangle); \\ & \qquad\quad \mathbf{otherwise} \to \mathbf{unit_E}\ (\mathsf{wrong} : *)\ \}\ )\ )\ ) \end{aligned}$$

Notice if $\forall i \,.\, v''_i \in v'_{\pi' i}$ and $y' \in [\![\theta\, v_1]\!]$ then

$$\begin{aligned}
&\mathbf{unit_E}\ (\mathsf{prod}_{n+1} : \langle v''_1, \ldots, v''_{j-1}, y', v''_j, \ldots, v''_n\rangle) \\
&\in \mathbf{E}\,\{\mathsf{prod}_{n+1} : \langle v_1, \ldots, v_{j-1}, v', v_j, \ldots, v_n\rangle \mid v' \in [\![\theta\, v_1]\!], v_1 \in [\![v'_{\pi' 1}]\!], \ldots, v_n \in [\![v'_{\pi' n}]\!]\} \\
&= [\![\texttt{All}\ (\theta\, v_1\ \texttt{\#}\ \theta\, \rho)]\!]
\end{aligned}$$

So by (k)

$$(*) \in \mathbf{E}\,\{\mathsf{func} : g \mid g \in \mathbf{E}\,\mathcal{V} \to \mathbf{E}\,\mathcal{V}, v \in [\![\theta\, v_1]\!] \Longrightarrow g\ v \in S\}$$

where

$$\begin{aligned}
S &= \mathbf{E}\,\{\mathsf{func} : h \mid h \in \mathbf{E}\,\mathcal{V} \to \mathbf{E}\,\mathcal{V}, v \in T \Longrightarrow h\ v \in [\![\theta\, \tau]\!]\} \\
T &= \mathbf{E}\,\{\mathsf{prod}_n : \langle v_1, \ldots, v_n\rangle \mid v_1 \in [\![v'_{\pi' 1}]\!], \ldots, v_n \in [\![v'_{\pi' n}]\!]\}
\end{aligned}$$

and thus by (i)

$$\begin{aligned}
(*) &\in [\![(\theta\, v_1)\ \texttt{->}\ (\theta\, v_2)\ \texttt{->}\ (\theta\, \tau)]\!] \\
&= [\![\theta\, (v_1\ \texttt{->}\ v_2\ \texttt{->}\ \tau)]\!]
\end{aligned}$$

Then by I.H. (ii)

$$[\![T[\lambda y\,.\,\lambda z\,.\,U\ (\mathsf{insert}\ y\ \mathsf{at}\ W\ \mathsf{into}\ z)]\!]_{\eta +\!\!+ env(B)} \in [\![\theta\, (v_1\ \texttt{->}\ v_2\ \texttt{->}\ \tau)]\!] \tag{1}$$

By definition

$$\begin{aligned}
&[\![\lambda x\,.\ \mathsf{let}\ y\ z = \mathsf{remove}\ W\ \mathsf{from}\ x \\
&\qquad \mathsf{in}\ T[\lambda y\,.\,\lambda z\,.\,\bullet\ (\mathsf{insert}\ y\ \mathsf{at}\ W\ \mathsf{into}\ z)]\ y\ z]\!]_{\eta +\!\!+ env(B)} \\
&= \mathbf{unit_E}\ (\lambda x'\,.\ \mathbf{let_E}\ v \leftarrow [\![x]\!]_{\eta +\!\!+ env(B), x \mapsto x'} \\
&\qquad \mathbf{in\ case}\ (v, [\![W]\!]_{\eta +\!\!+ env(B), x \mapsto x'})\ \mathbf{of}\ \{ \\
&\qquad\quad (\mathsf{prod}_{n+1} : \langle v'_1, \ldots, v'_{i-1}, v'', v'_i, \ldots, v'_n\rangle, \mathsf{iind} : i) \to \\
&\qquad\qquad \mathbf{if}\ 1 \le i \le (n+1)\ \mathbf{then} \\
&\qquad\qquad\quad [\![T[\lambda y\,.\,\lambda z\,.\,U\ (\mathsf{insert}\ y\ \mathsf{at}\ W\ \mathsf{into}\ z)]\ y\ z]\!]_{\eta'} \\
&\qquad\qquad \mathbf{else\ unit_E}\ (\mathsf{wrong} : *); \\
&\qquad\quad \mathbf{otherwise} \to \mathbf{unit_E}\ (\mathsf{wrong} : *)\ \}\ ) \\
&= (**)
\end{aligned}$$

where

$$\eta' = \eta +\!\!+ env(B), x \mapsto x', y \mapsto v'', z \mapsto \mathbf{unit_E}\ (\mathsf{prod}_n : \langle v'_1, \ldots, v'_{i-1}, v'_i, \ldots, v'_n\rangle)$$

Then by (j)

$$\begin{aligned}(**) = \mathbf{unit_E}\ (\lambda x' .\ & \mathbf{let_E}\ v \leftarrow x' \\ & \mathbf{in\ case}\ v\ \mathbf{of}\ \{ \\ & \quad \mathsf{prod}_{n+1} : \langle v'_1, \ldots, v'_{j-1}, v'', v'_j, \ldots, v'_n \rangle \rightarrow \\ & \quad\quad [\![T[\lambda y\ .\ \lambda z\ .\ U\ (\mathsf{insert}\ y\ \mathsf{at}\ W\ \mathsf{into}\ z)]\ y\ z]\!]_{\eta''} \\ & \quad \mathbf{otherwise} \rightarrow \mathbf{unit_E}\ (\mathsf{wrong} : *)\ \}\ ) \\ \in \mathbf{E}\ \{\mathsf{func} : f \mid f \in \mathbf{E}\,\mathcal{V} \rightarrow \mathbf{E}\,\mathcal{V}, v \in [\![\mathtt{All}\ (\theta\, \upsilon_1\ \#\ \theta\, \rho)]\!] \Longrightarrow f\ v \in [\![\theta\ \tau]\!]\} \\ = [\![\mathtt{All}\ (\theta\, \upsilon_1\ \#\ \theta\, \rho)\ \texttt{->}\ (\theta\ \tau)]\!] \\ = [\![\theta\ (\mathtt{All}\ (\upsilon_1\ \#\ \rho)\ \texttt{->}\ \tau)]\!]\end{aligned}$$

as required, where

$$\eta'' = \eta +\!\!+\ env(B), y \mapsto v'', z \mapsto \mathbf{unit_E}\ (\mathsf{prod}_n : \langle v'_1, \ldots, v'_{j-1}, v'_j, \ldots, v'_n \rangle)$$

case P6: Let (a) be

$$\Delta \mid C \mid \Gamma \vdash_{n+1} \mathtt{\backslash Triv}\ .\ t : \mathtt{All\ Empty}\ \texttt{->}\ \tau \hookrightarrow \lambda x\ .\ \mathsf{let}\ \langle\rangle = x\ \mathsf{in}\ T[\bullet\ x]$$

Then by P6 $\Delta \mid C \mid \Gamma \vdash_n t : \tau \hookrightarrow T[\bullet]$, and so since x is fresh, by Lemma B.38

$$\Delta \mid C \mid \Gamma, x : \mathtt{All\ Empty} \vdash_n t : \tau \hookrightarrow T[\bullet] \qquad \text{(f)}$$

By (e)

$$\begin{aligned}[\![U]\!]_{\eta +\!\!+ env(B)} &\in [\![\mathtt{All\ Empty}\ \texttt{->}\ \theta\ \tau]\!] \\ &= \mathbf{E}\ \{\mathsf{func} : f \mid f \in \mathbf{E}\,\mathcal{V} \rightarrow \mathbf{E}\,\mathcal{V}, v' \in [\![\mathtt{All\ Empty}]\!] \Longrightarrow f\ v' \in [\![\theta\ \tau]\!]\} \qquad \text{(g)}\end{aligned}$$

Let v be s.t.

$$\begin{aligned}v &\in [\![\mathtt{All\ Empty}]\!] \\ &= \mathbf{E}\ \{\mathsf{prod}_0 : \langle\rangle\} \qquad \text{(h)}\end{aligned}$$

and let $\eta' = \eta, x \mapsto v$.
Then by (g) and by definition

$$\begin{aligned}& [\![U\ x]\!]_{\eta' +\!\!+ env(B)} \\ = \ & \mathbf{let_E}\ v \leftarrow [\![U]\!]_{\eta' +\!\!+ env(B)} \\ & \mathbf{in\ case}\ v\ \mathbf{of}\ \{ \\ & \quad \mathsf{func} : f \rightarrow f\ [\![x]\!]_{\eta' +\!\!+ env(B)} \\ & \quad \mathbf{otherwise} \rightarrow \mathbf{unit_E}\ (\mathsf{wrong} : *)\ \} \\ = \ & \mathbf{let_E}\ \mathsf{func} : f \leftarrow [\![U]\!]_{\eta +\!\!+ env(B)} \\ & \mathbf{in}\ f\ v \\ \in \ & [\![\theta\ \tau]\!]\end{aligned}$$

By (d) $\eta' \models (\theta\,\Gamma), x : (\theta\ \mathtt{All\ Empty})$, so by I.H. (ii) (using η') on (f)

$$[\![T[U\ x]]\!]_{\eta' +\!\!+ env(B)} \in [\![\theta\ \tau]\!] \qquad \text{(i)}$$

By definition

$$\begin{aligned}
&[\![\lambda x \text{ . let } \langle\rangle = x \text{ in } T[U\ x]]\!]_{\eta +\!\!+ env(B)} \\
&= \mathbf{unit_E}\ (\lambda x' \text{ . } \mathbf{let_E}\ v' \leftarrow [\![x]\!]_{\eta +\!\!+ env(B), x \mapsto x'} \\
&\qquad \mathbf{in\ case}\ v'\ \mathbf{of}\ \{ \\
&\qquad\quad \mathsf{prod}_0 : \langle\rangle \rightarrow [\![T[U\ x]]\!]_{\eta +\!\!+ env(B), x \mapsto x'}; \\
&\qquad\quad \mathbf{otherwise} \rightarrow \mathbf{unit_E}\ (\mathsf{wrong} : *)\ \}\ ) \\
&= \mathbf{unit_E}\ (\lambda x' \text{ . } \mathbf{let_E}\ v' \leftarrow x' \\
&\qquad \mathbf{in\ case}\ v'\ \mathbf{of}\ \{ \\
&\qquad\quad \mathsf{prod}_0 : \langle\rangle \rightarrow [\![T[U\ x]]\!]_{\eta +\!\!+ env(B), x \mapsto x'}; \\
&\qquad\quad \mathbf{otherwise} \rightarrow \mathbf{unit_E}\ (\mathsf{wrong} : *)\ \}\ ) \\
&= (*)
\end{aligned}$$

Then since the choice of v was aribitrary s.t. (h) holds, by (i)

$$\begin{aligned}
(*) &\in \mathbf{E}\ \{\mathsf{func} : g \mid g \in \mathbf{E}\ \mathcal{V} \rightarrow \mathbf{E}\ \mathcal{V}, v \in [\![\texttt{All Empty}]\!] \Longrightarrow g\ v \in [\![\theta\ \tau]\!]\} \\
&= [\![\texttt{All Empty -> } \theta\ \tau]\!] \\
&= [\![\theta\ (\texttt{All Empty -> } \tau)]\!]
\end{aligned}$$

as required.

case P7: Let (a) be

$$\Delta \mid C \mid \Gamma \vdash_{n+1} \backslash x\ .\ t : \upsilon \texttt{ -> } \tau \hookrightarrow \lambda x\ .\ T[\bullet\ x]$$

Then by P7

$$\Delta \mid C \mid \Gamma, x : \upsilon \vdash_n t : \tau \hookrightarrow T[\bullet] \qquad \text{(f)}$$

$$\Delta \vdash \upsilon : \mathsf{Type} \qquad \text{(g)}$$

By (e)

$$\begin{aligned}
[\![U]\!]_{\eta +\!\!+ env(B)} &\in [\![\theta\ (\upsilon \texttt{ -> } \tau)]\!] \\
&= [\![\theta\ \upsilon \texttt{ -> } \theta\ \tau]\!] \\
&= \mathbf{E}\ \{\mathsf{func} : f \mid f \in \mathbf{E}\ \mathcal{V} \rightarrow \mathbf{E}\ \mathcal{V}, v' \in [\![\theta\ \upsilon]\!] \Longrightarrow f\ v' \in [\![\theta\ \tau]\!]\} \qquad \text{(h)}
\end{aligned}$$

Let v be s.t.

$$v \in [\![\theta\ \upsilon]\!] \qquad \text{(i)}$$

and let $\eta' = \eta, x \mapsto v$.

Then by (h) and by definition

$$
\begin{aligned}
& [\![U\ x]\!]_{\eta' +\!\!+ env(B)} \\
= \ & \mathbf{let_E}\ v \leftarrow [\![U]\!]_{\eta' +\!\!+ env(B)} \\
& \mathbf{in\ case}\ v\ \mathbf{of}\ \{ \\
& \quad \mathsf{func} : f \rightarrow f\ [\![x]\!]_{\eta' +\!\!+ env(B)}; \\
& \quad \mathbf{otherwise} \rightarrow \mathbf{unit_E}\ (\mathsf{wrong} : *)\ \} \\
= \ & \mathbf{let_E}\ \mathsf{func} : f \leftarrow [\![U]\!]_{\eta +\!\!+ env(B)} \\
& \mathbf{in}\ f\ v \\
\in \ & [\![\theta\ \tau]\!]
\end{aligned}
$$

By (d) and (g) $\eta' \models (\theta\ \Gamma), x : (\theta\ v)$. Then by I.H. (ii) (using η') on (f)

$$[\![T[U\ x]]\!]_{\eta' +\!\!+ env(B)} \in [\![\theta\ \tau]\!]$$

By definition

$$
\begin{aligned}
& [\![\lambda x\ .\ T[U\ x]]\!] \\
& = \mathbf{unit_E}\ (\mathsf{func} : \lambda x'\ .\ [\![T[U\ x]]\!]_{\eta +\!\!+ env(B), x \mapsto x'}) \\
& = (*)
\end{aligned}
$$

Since the choice of v was arbitrary s.t. (i) holds

$$
\begin{aligned}
(*) & \in \mathbf{E}\ \{\mathsf{func} : g \mid g \in \mathbf{E}\ \mathcal{V} \rightarrow \mathbf{E}\ \mathcal{V}, v \in [\![\theta\ v]\!] \Longrightarrow g\ v \in [\![\theta\ \tau]\!]\} \\
& = [\![\theta\ v\ \texttt{->}\ \theta\ \tau]\!] \\
& = [\![\theta\ (v\ \texttt{->}\ \tau)]\!]
\end{aligned}
$$

as required.

□

Appendix C

Proofs for Chapter 5

C.1 Simplifier Correctness

Lemma C.1 Let $\Delta \vdash C/D$ constraint and $\Delta \vdash \tau : \mathsf{Type}$ and $\Delta \vdash \rho : \mathsf{Row}$ and $\Delta \vdash \theta'$ subst and $notIn(C \vdash \tau, \rho)$.

Then $\theta \in saturate((\theta' C) \mathbin{+\!\!+} D)$ implies $\neg isIn(\theta\,\theta'\,\tau, \theta\,\theta'\,\rho)$.

Proof By definition of *saturate*

$$\begin{array}{l} \theta \in mgus_{\emptyset}(\mathbf{Id} \vdash eqs((\theta'\ C) \mathbin{+\!\!+} D)) \\ \wedge\, \forall(\tau'\ \boldsymbol{ins}\,\rho') \in inss((\theta'\ C) \mathbin{+\!\!+} D)\ .\ \neg isIn(\theta\,\tau', \theta\,\rho') \end{array} \quad \text{(a)}$$

W.l.o.g. assume $\rho = (\#)_m\ \overline{\upsilon}\ l$, and $\theta\ \theta'\ l = (\#)_n\ \overline{\upsilon''}\ l'$ and thus $\theta\ \theta'\ \rho = (\#)_{m+n}\ \overline{\upsilon'}\ l'$, where

$$\upsilon'_i = \begin{cases} \theta\,\theta'\,\upsilon_i, \textbf{if } 1 \le i \le m \\ \upsilon''_{i-m}, \ \textbf{if } m < i \le (m+n) \end{cases}$$

Now assume $isIn(\theta\,\theta'\,\tau, \theta\,\theta'\,\rho)$, that is, there exists an i s.t.

$$cmp_{opaque}(\theta\,\theta'\,\tau, \upsilon'_i) = \mathbf{eq} \quad \text{(b)}$$

We shall show each possible value for i leads to a contradiction.

case $1 \le i \le m$: Thus $cmp_{opaque}(\theta\,\theta'\,\tau, \theta\,\theta'\,\upsilon_i) = \mathbf{eq}$.
Since by definition of *notIn*, $notEqual(C \vdash \tau, \upsilon_i)$, then by definition of *notEqual*, *satisfied* and *isIn* we have

$$\begin{array}{l} \forall\theta'' \in mgus_{opaque}(\mathbf{Id} \vdash \tau\ \boldsymbol{eq}\,\upsilon_i)\ . \\ \quad \exists(\tau'\ \boldsymbol{ins}\,\rho') \in inss(C)\ .\ isIn(\theta''\,\tau', \theta''\,\rho') \end{array}$$

Then by Lemma B.8 and stability of cmp_{opaque}:

$$\begin{array}{l} \forall\theta''' \in mgus_{opaque}(\mathbf{Id} \vdash \theta\,\theta'\,\tau\ \boldsymbol{eq}\,\theta\,\theta'\,\upsilon_i)\ . \\ \quad \exists\theta'' \in mgus_{opaque}(\mathbf{Id} \vdash \tau\ \boldsymbol{eq}\,\upsilon_i)\ . \\ \quad\quad \exists\theta''''\ .\ \theta''' \circ \theta \circ \theta' \equiv_{\emptyset} \theta'''' \circ \theta''\ . \\ \quad\quad\quad \exists(\tau'\ \boldsymbol{ins}\,\rho') \in inss(C)\ .\ isIn(\theta''''\,\theta''\,\tau', \theta''''\,\theta''\,\rho') \end{array}$$

and thus by transitivity of cmp_{opaque}

$$\begin{array}{l}\forall \theta''' \in mgus_{opaque}(\mathbf{Id} \vdash \theta\, \theta'\, \tau \; \boldsymbol{eq}\, \theta\, \theta'\, \upsilon_i) \;. \\ \quad \exists (\tau' \; \boldsymbol{ins}\, \rho') \in inss(C) \;.\; isIn(\theta'''\, \theta\, \theta'\, \tau', \theta'''\, \theta\, \theta'\, \rho')\end{array}$$

But by (b) and Lemma B.7

$$\exists \theta''' \in mgus_{opaque}(\mathbf{Id} \vdash \theta\, \theta'\, \tau \; \boldsymbol{eq}\, \theta\, \theta'\, \upsilon_i) \;.\; \theta''' = \mathbf{Id}$$

Thus

$$\exists (\tau' \; \boldsymbol{ins}\, \rho') \in inss(C) \;.\; isIn(\theta\, \theta'\, \tau', \theta\, \theta'\, \rho')$$

which contradicts (a).

case $m < i \leq (m + n)$: Thus $cmp_{opaque}(\theta\, \theta'\, \tau, \upsilon''_{i-m}) = \mathbf{eq}$ (and of course $l \neq$ `Empty`.)
Then by definition of *notIn*

$$\exists (\tau' \; \boldsymbol{ins}\; (\#)_{m'}\; \overline{\tau''}\; l'') \in inss(C) \;.\; cmp_{opaque}(\tau, \tau') = \mathbf{eq} \wedge l = l''$$

which by stability of cmp_{opaque} implies

$$\exists (\tau' \; \boldsymbol{ins}\; (\#)_{m'}\; \overline{\tau''}\; l'') \in inss(C) \;.\; cmp_{opaque}(\theta\, \theta'\, \tau, \theta\, \theta'\, \tau') = \mathbf{eq} \wedge l = l''$$

which by (a) implies

$$\exists (\tau' \; \boldsymbol{ins}\; (\#)_{m'}\; \overline{\tau''}\; l'') \in inss(C) \;.\; \theta\, \theta'\, l'' = (\#)_n\; \overline{\upsilon''}\; l' \wedge cmp_{opaque}(\theta\, \theta'\, \tau', \upsilon''_{i-m}) = \mathbf{eq}$$

that is

$$\exists (\tau' \; \boldsymbol{ins}\, \rho') \in inss(C) \;.\; isIn(\theta\, \theta'\, \tau', \theta\, \theta'\, \rho')$$

which contradicts (a).

□

Lemma C.2 Let $\Delta \vdash C$ constraint and $inhs(C) = C$ and $\Delta \vdash \tau : \mathsf{Type}$ and $\Delta \vdash \rho : \mathsf{Row}$ and $notIn(C \vdash \tau, \rho)$ and $\vdash \theta : \Delta \to \Delta_{init}$ and $\eta \models \theta\; C$.
Then $\neg isIn(\theta\, \tau, \theta\, \rho)$.

Proof By Lemma B.15 (i) $\mathbf{Id} \in saturate(\theta\;\; C)$. Then the result is immediate by Lemma C.1. □

Lemma C.3 Let $\Delta \vdash C$ constraint and $\overline{a} \subseteq dom(\Delta)$ and $\langle \overline{a} \mid C \rangle \triangleright \langle \theta \mid C' \mid B \rangle$. Then

(i) $\theta\; C' = C'$

(ii) $\Delta \vdash C'$ constraint

(iii) There exists a Δ' s.t. $\Delta \mathbin{+\!\!+} \Delta' \vdash \theta$ subst

(iv) $\Delta \vdash \theta_{\restriction \overline{a}}$ subst

(v) $C' =$ `false` or there exists D_1, D_2 and D_3 s.t. $C = D_1 \mathbin{+\!\!+} D_2$ and $C' = (\theta\; D_1) \mathbin{+\!\!+} D_3$

Proof

(i) In rule S2, $[b \mapsto \tau]$ is applied to C, so the rusult follows by idempotency. In rule S17, $dom(\theta) \cap fv_\emptyset(C) = \emptyset$. All other rules yield **Id**.

(ii) In rule S17, θ may introduce fresh variables into $\theta\ D$, but this constraint does not appear within the result. All other rules do not introduce fresh variables. The preservation of well-kinding is by inspection.

(iii) For rule S17, Δ' is as given by Lemma B.18. For all other rules, $\Delta' = \cdot$.

(iv) In rule S2, τ is well-kinded by well-kinding of $C, b\ eq\, \tau$. In rule S17, $dom(\theta) \cap \overline{a} = \emptyset$.

(v) $C' = \texttt{false}$ in rules S4–S7, S16 and S18. For the remaining rules, result follows by inspection.

□

Lemma C.4 Let $\Delta \vdash C$ constraint and $\overline{a} \subseteq dom(\Delta)$ and $\langle \overline{a} \mid C \rangle \triangleright \langle \theta \mid C' \mid B \rangle$. Then

(i) $C' \vdash^e \theta\ C \hookrightarrow B$

(ii) $\theta\ C \vdash^e C' \hookrightarrow B'$

(iii) If there exists a $\vdash \theta' : \Delta \to \Delta_{init}$ and η' s.t. $\eta' \models \theta'\ C$, then there exists a $\vdash \theta'' : \Delta \to \Delta_{init}$ s.t.

(iii.1) $\theta'_{\restriction \overline{a} \cup fv_\emptyset(C')} \equiv_\emptyset (\theta'' \circ \theta)_{\restriction \overline{a} \cup fv_\emptyset(C')}$

(iii.2) $\eta' \models \theta'' \circ \theta\ C$

(iii.3) $env(B', \eta') \models \theta''\ C'$ (where B' is from (ii) above)

Proof We may substantially simplify each of these conclusions in specific cases.

(i) If $C' = \texttt{false}$ then the result holds vacuously. Otherwise, by Lemma C.3, $C = D_1 +\!\!+ D_2$ and $C' = (\theta\ D_1) +\!\!+ D_3$. Then it is suficient to show

$$(\theta\ D_1) +\!\!+ D_3 \vdash^e \theta\ D_2 \hookrightarrow B \qquad \text{(iv)}$$

since by Lemmas B.28 and B.34 $(\theta\ D_1) +\!\!+ D_3 \vdash^e \theta\ D_1 \hookrightarrow \cdot$.

(ii) If $C' = \texttt{false}$, then we need show $\theta\ C \vdash^e \texttt{false}$, which is to say

$$saturate(\theta\ C) = \emptyset \qquad \text{(v)}$$

Otherwise, by Lemma C.3, $C = D_1 +\!\!+ D_2$ and $C' = (\theta\ D_1) +\!\!+ D_3$. Then it is sufficient to show

$$\theta\ (D_1 +\!\!+ D_2) \vdash^e D_3 \qquad \text{(vi)}$$

since by Lemmas B.28 and B.34 $\theta\ (D_1 +\!\!+ D_2) \vdash^e \theta\ D_1 \hookrightarrow \cdot$.

(iii) If $C' = \texttt{false}$ then by (ii) $saturate(\theta\ C) = \emptyset$. Thus by Lemma B.15 (i) there is no θ' and η' s.t. $\eta' \models \theta'\ C$.

Otherwise, it is sufficient to show (iii.1) and either one of (iii.2) and (iii.3). To see how (iii.3) follows from (iii.2), notice that given $\eta' \models \theta'\ C$, by (i) and Lemma B.26 $\theta'' \circ \theta\ C \vdash^e \theta''\ C' \hookrightarrow B'$. Thus by (iii.2) and Lemma B.13 $env(B', \eta') \models \theta''\ C'$.

Conversely, to see how (iii.2) follows from (iii.3), notice that given $\eta' \models \theta'\ C$, by (ii) and Lemma B.27 $\theta''\ C' \vdash^e \theta'' \circ \theta\ C \hookrightarrow B$. Thus by (iii.3) and Lemma B.13 $env(B, env(B', \eta')) \models \theta'' \circ \theta\ C$. But by (i), (ii), Lemma B.31 and Lemma B.28 $env(B, env(B', \eta'))_{\restriction names(C)} = \eta$, so that $\eta \models \theta'' \circ \theta\ C$.

Notice that by the above argument, if $\theta = \mathbf{Id}$, then we may take $\theta'' = \theta'$. Thus (iii.1) and (iii.2) are vacuous, and (iii.3) follows from (iii.2).

We procced by case analysis of the rewrite rule:

case S1: We have

$$\langle \overline{a} \mid C, \tau\ \boldsymbol{eq}\, v \rangle \triangleright \langle \mathbf{Id} \mid C, v\ \boldsymbol{eq}\, \tau \mid \cdot \rangle$$

(iv) Since $(\tau\ \boldsymbol{eq}\, v) \equiv (v\ \boldsymbol{eq}\, \tau)$, by Lemmas B.24 and B.34

$$C, v\ \boldsymbol{eq}\, \tau \vdash^e \tau\ \boldsymbol{eq}\, v \hookrightarrow \cdot$$

as required.

(vi) As for (iv).

case S2: We have

$$\langle \overline{a} \mid C, b\ \boldsymbol{eq}\, \tau \rangle \triangleright \langle [b \mapsto \tau] \mid C[b \mapsto \tau] \mid \cdot \rangle$$

where

$$b \notin fv_{\emptyset}(\tau) \tag{a}$$

(iv) Immediate.

(vi) Immediate.

(iii) Let

$$\eta' \models \theta'\ C, \theta'\ b\ \boldsymbol{eq}\, \theta'\ \tau$$

Then $cmp_{\emptyset}(\theta'\ b, \theta'\ \tau) = \mathbf{eq}$. Let $\theta'' = \theta'_{\setminus b}$.

(iii.1) Then by (a)

$$\begin{aligned} \theta'' \circ [b \mapsto \tau] &= \theta'_{\setminus b} \circ [b \mapsto \tau] \\ &= \theta'_{\setminus b} \circ [b \mapsto \theta'\ \tau] \\ &\equiv_{\emptyset} \theta' \end{aligned}$$

(iii.2) Then by Lemma B.10

$$\eta' \models \theta'' \circ [b \mapsto \tau]\ C, \theta'' \circ [b \mapsto \tau]\ b\ \boldsymbol{eq}\, \theta'' \circ [b \mapsto \tau]\ \tau$$

(iii.3) Follows from (iii.2)

case S3: We have

$$\langle \overline{a} \mid C, F\ \overline{\tau}\ \boldsymbol{eq}\, F\ \overline{v} \rangle \triangleright \langle \mathbf{Id} \mid C, \overline{\tau\ \boldsymbol{eq}\, v} \mid \cdot \rangle$$

(iv) If $\theta \in saturate(C, \overline{\tau\ \boldsymbol{eq}\, v})$ then $\forall i\ .\ cmp(\theta\ \tau_i, \theta\ v_i) = \mathbf{eq}$, and thus $cmp(F\ \overline{\theta\ \tau}, F\ \overline{\theta\ v}) = \mathbf{eq}$. Thus by EQUALS and CONJ

$$C, \overline{\tau\ \boldsymbol{eq}\, v} \vdash^e F\ \overline{\tau}\ \boldsymbol{eq}\, F\ \overline{v}$$

as required.

(vi) As for (iv).

case S4: We have

$$\langle \overline{a} \mid C, F\ \overline{\tau}\ eq\ G\ \overline{v} \rangle \triangleright \langle \mathbf{Id} \mid \mathtt{false} \mid \cdot \rangle$$

where $F \neq G$.

(v) Since $cmp_\emptyset(F\ \overline{\tau}, G\ \overline{v}) \in \{\mathbf{lt}, \mathbf{gt}\}]$, by Lemma B.6 $mgus_\emptyset(\mathbf{Id} \vdash F\ \overline{\tau}\ eq\ G\ \overline{v}) = \emptyset$. Thus $saturate(F\ \overline{\tau}\ eq\ G\ \overline{v}) = \emptyset$. Then by Lemma B.19 (i) $saturate(C, F\ \overline{\tau}\ eq\ G\ \overline{v}) = \emptyset$ as required.

case S5: We have

$$\langle \overline{a} \mid C, (\mathtt{\#})_m\ \overline{\tau}\ b\ eq\ (\mathtt{\#})_n\ \overline{v}\ \mathtt{Empty} \rangle \triangleright \langle \mathbf{Id} \mid \mathtt{false} \mid \cdot \rangle$$

where

$$m > n \tag{a}$$

(v) By (a) and Lemma B.6

$$mgus_\emptyset(\mathbf{Id} \vdash (\mathtt{\#})_m\ \overline{\tau}\ b\ eq\ (\mathtt{\#})_n\ \overline{v}\ \mathtt{Empty}) = \emptyset$$

and thus $saturate(C, (\mathtt{\#})_m\ \overline{\tau}\ b\ eq\ (\mathtt{\#})_n\ \overline{v}\ \mathtt{Empty}) = \emptyset$.

case S6: We have

$$\langle \overline{a} \mid C, (\mathtt{\#})_m\ \overline{\tau}\ \mathtt{Empty}\ eq_{\mathtt{Row}}\ (\mathtt{\#})_n\ \overline{v}\ \mathtt{Empty} \rangle \triangleright \langle \mathbf{Id} \mid \mathtt{false} \mid \cdot \rangle$$

where

$$m \neq n \tag{a}$$

(v) By (a) and Lemma B.6

$$mgus_\emptyset(\mathbf{Id} \vdash (\mathtt{\#})_m\ \overline{\tau}\ b\mathtt{Empty}(\mathtt{\#})_n\ \overline{v}\ \mathtt{Empty}) = \emptyset$$

and thus $saturate(C, (\mathtt{\#})_m\ \overline{\tau}\ b\mathtt{Empty}(\mathtt{\#})_n\ \overline{v}\ \mathtt{Empty}) = \emptyset$.

case S7: We have

$$\langle \overline{a} \mid C, (\mathtt{\#})_m\ \overline{\tau}\ l\ eq_{\mathtt{Row}}\ (\mathtt{\#})_n\ \overline{v}\ l' \rangle \triangleright \langle \mathbf{Id} \mid \mathtt{false} \mid \cdot \rangle$$

where

$$notIn(C \vdash \tau_i, (\mathtt{\#})_n\ \overline{v}\ l') \tag{a}$$

(v) Assume there exists a θ s.t.

$$\theta \in saturate(C, (\mathtt{\#})_m\ \overline{\tau}\ l\ eq_{\mathtt{Row}}\ (\mathtt{\#})_n\ \overline{v}\ l') \tag{b}$$

Then by (a) and Lemma C.1 $\neg isIn(\theta\ \tau_i, \theta\ ((\mathtt{\#})_n\ \overline{v}\ l'))$. But then by Lemma 4.2 (iv)

$$cmp_\emptyset(\theta\ ((\mathtt{\#})_m\ \overline{\tau}\ l), \theta\ ((\mathtt{\#})_n\ \overline{v}\ l')) \neq \mathbf{eq}$$

which contradicts (b). Thus $saturate(C, (\mathtt{\#})_m\ \overline{\tau}\ l\ eq_{\mathtt{Row}}\ (\mathtt{\#})_n\ \overline{v}\ l') = \emptyset$.

case S8: We have

$$\langle \overline{a} \mid C, (\#)_m\, \overline{\tau}\; l\; eq\; (\#)_n\, \overline{\upsilon}\; l' \rangle$$
$$\rhd \langle \mathbf{Id} \mid C, \tau_i\; eq_{\mathrm{Type}}\; \upsilon_j, (\#)_{m-1}\, \overline{\tau}_{\backslash i}\; l\; eq\; (\#)_{n-1}\, \overline{\upsilon}_{\backslash j}\; l' \mid \cdot \rangle$$

where

$$notIn(C \vdash \tau_i, (\#)_{n-1}\, \overline{\upsilon}_{\backslash j}\; l') \tag{a}$$
$$cmp_{opaque}(\tau_i, \upsilon_j) \in \{\mathbf{eq}, \mathbf{unk}\} \tag{b}$$

(iv) Let

$$\theta \in saturate(C, \tau_i\; eq_{\mathrm{Type}}\; \upsilon_j, (\#)_{m-1}\, \overline{\tau}_{\backslash i}\; l\; eq\; (\#)_{n-1}\, \overline{\upsilon}_{\backslash j}\; l')$$

Then by Lemma B.6

$$cmp_{\emptyset}(\theta\, \tau_i, \theta\, \upsilon_j) = \mathbf{eq}$$
$$cmp_{\emptyset}(\theta\, ((\#)_{m-1}\, \overline{\tau}_{\backslash i}\; l), \theta\, ((\#)_{n-1}\, \overline{\upsilon}_{\backslash j}\; l')) = \mathbf{eq}$$

and thus by Lemma 4.2 (iv)

$$cmp_{\emptyset}(\theta\, ((\#)_m\, \overline{\tau}\; l), \theta\, ((\#)_n\, \overline{\upsilon}\; l')) = \mathbf{eq}$$

Then by EQUALS and CONJ

$$C, \tau_i\; eq_{\mathrm{Type}}\; \upsilon_j, (\#)_{m-1}\, \overline{\tau}_{\backslash i}\; l\; eq\; (\#)_{n-1}\, \overline{\upsilon}_{\backslash j}\; l' \vdash^e (\#)_m\, \overline{\tau}\; l\; eq\; (\#)_n\, \overline{\upsilon}\; l' \hookrightarrow \cdot$$

as required.

(vi) Let

$$\theta \in saturate(C, (\#)_m\, \overline{\tau}\; l\; eq\; (\#)_n\, \overline{\upsilon}\; l')$$

Then by Lemma B.6

$$cmp_{\emptyset}(\theta\, ((\#)_m\, \overline{\tau}\; l), \theta\, ((\#)_n\, \overline{\upsilon}\; l')) = \mathbf{eq}$$

and thus by Lemma 4.2 (iv)

$$cmp_{\emptyset}(\theta\, \tau_i, \theta\, \upsilon_j) = \mathbf{eq}$$
$$cmp_{\emptyset}(\theta\, ((\#)_{m-1}\, \overline{\tau}_{\backslash i}\; l), \theta\, ((\#)_{n-1}\, \overline{\upsilon}_{\backslash j}\; l')) = \mathbf{eq}$$

Then by EQUALS and CONJ

$$C, (\#)_m\, \overline{\tau}\; l\; eq\; (\#)_n\, \overline{\upsilon}\; l' \vdash^e \tau_i\; eq_{\mathrm{Type}}\; \upsilon_j, (\#)_{m-1}\, \overline{\tau}_{\backslash i}\; l\; eq\; (\#)_{n-1}\, \overline{\upsilon}_{\backslash j}\; l' \hookrightarrow \cdot$$

Notice that (b) plays no part in this result, and serves only to distinguish this rule from rule S7.

case S10: We have

$$\langle \overline{a} \mid C, w : \tau\; ins\, \rho, w' : \tau'\; ins\, \rho' \rangle \rhd \langle \mathbf{Id} \mid C, w : \tau\; ins\, \rho \mid w' = w \rangle$$

where

$$cmp_{opaque}(\tau, \tau') = \mathbf{eq} \quad \text{(a)}$$
$$cmp_{opaque}(\rho, \rho') = \mathbf{eq} \quad \text{(b)}$$

(iv) By (a), (b) and stability of cmp_{opaque}, if $\theta \in saturate(C, w : \tau\ \boldsymbol{ins}\ \rho)$ then

$$cmp_{opaque}(\theta\ \tau, \theta\ \tau') = \mathbf{eq} \wedge cmp_{opaque}(\theta\ \rho, \theta\ \rho') = \mathbf{eq}$$

Then by MREF, INSERT and CONJ

$$C, w : \tau\ \boldsymbol{ins}\ \rho \vdash^e w' : \tau'\ \boldsymbol{ins}\ \rho' \hookrightarrow w' = w$$

as required.

(vi) Vacuously,

$$C, w : \tau\ \boldsymbol{ins}\ \rho, w' : \tau'\ \boldsymbol{ins}\ \rho' \vdash^e \mathtt{true} \hookrightarrow \cdot$$

case S11: We have

$$\langle \overline{a} \mid C, w : \tau\ \boldsymbol{ins}\, \mathtt{Empty} \rangle \triangleright \langle \mathbf{Id} \mid C \mid w = \mathsf{One} \rangle$$

(iv) By MEMPTY, INSERT and CONJ

$$C \vdash^e w : \tau\ \boldsymbol{ins}\, \mathtt{Empty} \hookrightarrow w = \mathsf{One}$$

(vi) Vacuously,

$$C, w : \tau\ \boldsymbol{ins}\, \mathtt{Empty} \vdash^e \mathtt{true}$$

case S13: We have

$$\langle \overline{a} \mid C, w : \tau\ \boldsymbol{ins}\ (\texttt{\#})_n\ \overline{v}\ l \rangle \triangleright \langle \mathbf{Id} \mid C, w' : \tau\ \boldsymbol{ins}\ (\texttt{\#})_{n-1}\ \overline{v}_{\backslash i}\ l \mid w = \mathsf{Inc}\ w' \rangle$$

where

$$cmp_{opaque}(\tau, v_i) = \mathbf{gt} \quad \text{(a)}$$

and w' fresh.

(iv) Let $\theta \in saturate(C, w' : \tau\, \boldsymbol{ins} (\texttt{\#})_{n-1}\ \overline{v}_{\backslash i}\ l)$. Then by (a) and stability of cmp_{opaque}

$$cmp_{opaque}(\theta\ \tau, \theta\ v_i) = \mathbf{gt} \quad \text{(b)}$$

By MREF

$$C, w' : \theta\ \tau\ \boldsymbol{ins}\, \theta\ ((\texttt{\#})_{n-1}\ \overline{v}_{\backslash i}\ l) \vdash^m \theta\ \tau\ \boldsymbol{ins}\, \theta\ ((\texttt{\#})_{n-1}\ \overline{v}_{\backslash i}\ l) \hookrightarrow w'$$

and so by (b) and MINC

$$C, w' : \theta\ \tau\ \boldsymbol{ins}\, \theta\ ((\texttt{\#})_{n-1}\ \overline{v}_{\backslash i}\ l) \vdash^m \theta\ \tau\ \boldsymbol{ins}\, \theta\ ((\texttt{\#})_{n}\ \overline{v}\ l) \hookrightarrow \mathsf{Inc}\ w'$$

Thus by INSERT and CONJ

$$C, w' : \tau\ \boldsymbol{ins}\ (\texttt{\#})_{n-1}\ \overline{v}_{\backslash i}\ l \vdash^e w : \tau\ \boldsymbol{ins}\ (\texttt{\#})_n\ \overline{v}\ l \hookrightarrow w = \mathsf{Inc}\ w'$$

as required.

(vi) Let $\theta \in saturate(C, w : \tau\ \boldsymbol{ins}\ (\texttt{\#})_n\ \overline{v}\ l)$. Then by (a) and stability of cmp_{opaque}

$$cmp_{opaque}(\theta\ \tau, \theta\ v_i) = \mathbf{gt} \tag{c}$$

By MREF

$$C, w : \theta\ \tau\ \boldsymbol{ins}\ \theta\ ((\texttt{\#})_n\ \overline{v}\ l) \vdash^m \theta\ \tau\ \boldsymbol{ins}\ \theta\ ((\texttt{\#})_n\ \overline{v}\ l) \hookrightarrow w$$

and so by (c) and MDEC

$$C, w : \theta\ \tau\ \boldsymbol{ins}\ \theta\ ((\texttt{\#})_n\ \overline{v}\ l) \vdash^m \theta\ \tau\ \boldsymbol{ins}\ \theta\ ((\texttt{\#})_{n-1}\ \overline{v}_{\backslash i}\ l) \hookrightarrow \mathsf{Dec}\ w$$

Thus by INSERT and CONJ

$$C, w : \tau\ \boldsymbol{ins}\ (\texttt{\#})_n\ \overline{v}\ l \vdash^e w' : \tau\ \boldsymbol{ins}\ (\texttt{\#})_{n-1}\ \overline{v}_{\backslash i}\ l \hookrightarrow w' = \mathsf{Dec}\ w$$

as required.

case S12, S14, S15: As for case S13.
case S16: We have

$$\langle \overline{a} \mid C, w : \tau\ \boldsymbol{ins}\ \rho \rangle \rhd \langle \mathbf{Id} \mid \texttt{false} \mid \cdot \rangle$$

where

$$isIn(\tau, \rho) \tag{a}$$

(v) Immediate from (a) and stability of cmp_{opaque}.

case S17: We have

$$\langle \overline{a} \mid C +\!\!+ D \rangle \rhd \langle \theta \mid C \mid B \rangle$$

where

$$fv_\emptyset(D) \cap fv_\emptyset(C) = \emptyset \tag{a}$$
$$fv_\emptyset(D) \cap \overline{a} = \emptyset \tag{b}$$
$$\theta \in saturate(D) \tag{c}$$
$$\forall \theta''' \in saturate(D)\ .\ \texttt{true} \vdash^e \theta'''\ D \hookrightarrow B \tag{d}$$

(iv) By (d) and Lemma B.34

$$\theta\ C \vdash^e \theta\ D \hookrightarrow B$$

as required.

(vi) Trivially, we have

$$\theta\ C +\!\!+ \theta\ D \vdash^e \texttt{true} \hookrightarrow \cdot$$

as required.

(iii) Let

$$\eta' \models \theta'\ C +\!\!+ \theta'\ D$$

and let $\theta'' = \theta'|_{\overline{a} \cup fv_\emptyset(C)}$.

By (a) and (b) we may split Δ into Δ_C and Δ_D s.t.

$$\begin{aligned}\Delta &= \Delta_C ++ \Delta_D \\ \overline{a} &\subseteq dom(\Delta_C) \\ \Delta_C &\vdash C \text{ constraint} \\ \Delta_D &\vdash D \text{ constraint}\end{aligned}$$

Then by Lemma B.18 there exists a Δ'_D s.t.

$$\Delta_D ++ \Delta'_D \vdash \theta \text{ subst} \tag{e}$$

W.l.o.g. we map assume $dom(\Delta'_D) \cap dom(\Delta_C) = \emptyset$.
(iii.1) By (e) $dom(\theta) \cap (\overline{a} \cup fv_\emptyset(C)) = \emptyset$. Thus

$$\begin{aligned}\theta'_{\restriction \overline{a} \cup fv_\emptyset(C)} &= \theta' \circ \theta_{\restriction \overline{a} \cup fv_\emptyset(C)} \\ &= \theta'_{\restriction \overline{a} \cup fv_\emptyset(C)} \circ \theta_{\restriction \overline{a} \cup fv_\emptyset(C)} \\ &= \theta'' \circ \theta_{\restriction \overline{a} \cup fv_\emptyset(C)}\end{aligned}$$

(iii.2) Since

$$\eta'_{\restriction names(D)} \models \theta' \ D \tag{f}$$

by Lemma B.15 (i) there exists a $\theta''' \in saturate(D)$ and a θ'''' s.t. $\theta' \equiv_\emptyset \theta'''' \circ \theta'''$. But by (d)

$$\texttt{true} \vdash^e \theta''' \ D \hookrightarrow B$$

and by Lemma B.27

$$\texttt{true} \vdash^e \theta'''' \circ \theta''' \ D \hookrightarrow B$$

and so by Lemma B.24

$$\texttt{true} \vdash^e \theta' \ D \hookrightarrow B$$

which by Lemma B.13 implies

$$env(B) \models \theta' \ D$$

and thus by (f)

$$env(B) = \eta'_{\restriction namesD} \tag{g}$$

Notice $\theta'' \circ \theta \ C = \theta' \ C$, thus

$$\eta' \models \theta'' \circ \theta \ C$$

Furthermore, by (a) and (b) $\theta'' \circ \theta \ D = \theta \ D$, thus since $env(B) \models \theta \ D$, by (g)

$$\eta' \models \theta'' \circ \theta \ D$$

Taken together, we thus have

$$\eta' \models \theta'' \circ \theta \ (C ++ D)$$

(iii.3) Follows from (iii.2)

case S18: We have

$$\langle \overline{a} \mid C +\!\!+ D\rangle \triangleright \langle \mathbf{Id} \mid \mathtt{false} \mid \cdot \rangle$$

where

$$fv_{\emptyset}(C) \cap fv_{\emptyset}(D) = \emptyset \quad \text{(a)}$$

$$fv_{\emptyset}(D) \cap \overline{a} = \emptyset \quad \text{(b)}$$

$$saturate(D) = \emptyset \quad \text{(c)}$$

(v) By (c) and Lemma B.19 (i)

$$saturate(C +\!\!+ D) = \emptyset$$

as required.

Note that (a) and (b) are unnecessary for this result, and are included only for pragmatic reasons.

□

Lemma C.5 Let $\Delta \vdash C_1$ constraint and $\overline{a} \subseteq dom(\Delta)$ and $\langle \overline{a} \mid C_1\rangle \triangleright^* \langle \theta_1 \mid C_2 \mid B_1\rangle$. Then

(i) $C_2 \vdash^e \theta_1\ C_1 \hookrightarrow B_2$ where if $\theta_2 : \Delta \to \Delta_{init}$ and $\eta_1 \models \theta_2\ C_2$ then $env(B_2, \eta_1)_{\restriction names(C_1)} = env(B_1, \eta_1)_{\restriction names(C_1)}$

(ii) $\theta_1\ C_1 \vdash^e C_2 \hookrightarrow B_3$

(iii) If there exists a $\vdash \theta_3 : \Delta \to \Delta_{init}$ and η_2 s.t. $\eta_2 \models \theta_3\ C_1$, then there exists a $\vdash \theta_4 : \Delta \to \Delta_{init}$ s.t.

(iii.1) $\theta_{3\restriction \overline{a} \cup fv_{\emptyset}(C_2)} \equiv_{\emptyset} (\theta_4 \circ \theta_1)_{\restriction \overline{a} \cup fv_{\emptyset}(C_2)}$

(iii.2) $\eta_2 \models \theta_4 \circ \theta_1\ C_1$

(iii.3) $env(B_3, \eta_2) \models \theta_4\ C_2$ (where B_3 is from (ii) above)

Proof By induction on derivation:

case SDONE: We have

$$\langle \overline{a} \mid C\rangle \triangleright^* \langle \mathbf{Id} \mid C \mid \cdot\rangle$$

Then (i) and (ii) hold by Lemma B.28, and (iii) holds vacuously.

case SSTEP: We have

$$\langle \overline{a} \mid C_1\rangle \triangleright^* \langle \theta_1'' \circ \theta_1' \mid C_2 \mid B_1'' +\!\!+ B_1'\rangle$$

where by SSTEP

$$\langle \overline{a} \mid C_1\rangle \triangleright \langle \theta_1' \mid C_3 \mid B_1'\rangle \quad \text{(a)}$$

$$\langle \overline{a} \cup \textstyle\bigcup_{a \in \overline{a}} fv_{\emptyset}(\theta_1'\ a) \mid C_3\rangle \triangleright^* \langle \theta_1'' \mid C_2 \mid B_1''\rangle \quad \text{(b)}$$

(i) By I.H. (i) on (b) $C_2 \vdash^e \theta_1''\ C_3 \hookrightarrow B_2''$ where if $\vdash \theta_2 : \Delta \to \Delta_{init}$ and $\eta_1 \models \theta_2\ C_2$ then $env(B_2'', \eta_1)_{\restriction names(C_3)} = env(B_1'', \eta_1)_{\restriction names(C_3)}$.
By Lemma C.4 (i) on (a) $C_3 \vdash^e \theta_1'\ C_1 \hookrightarrow B_1'$. Then by Lemma B.27 and Lemma B.31 $C_2 \vdash^e \theta_1' \circ \theta_1''\ C_1 \hookrightarrow B_2'$ where if $\vdash \theta_2 : \Delta \to \Delta_{init}$ and $\eta_1 \models \theta_2\ C_2$

then

$$\begin{aligned}
env(B_2', \eta_1)_{\restriction names(C_1)} &= env(B_2'' +\!\!+ B_1', \eta_1)_{\restriction names(C_1)} \\
&= env(B_1', env(B_2'', \eta_1)_{\restriction names(C_3)})_{\restriction names(C_1)} \\
&= env(B_1', env(B_1'', \eta_1)_{\restriction names(C_3)})_{\restriction names(C_1)} \\
&= env(B_1'' +\!\!+ B_1', \eta_1)_{\restriction names(C_1)}
\end{aligned}$$

as required.

(ii) By Lemma C.4 (ii) on (a) $\theta_1' \ C_1 \vdash^e C_3 \hookrightarrow B_3'$ for some B_3'. By I.H. (ii) on (b) $\theta_1'' \ C_3 \vdash^e C_2 \hookrightarrow B_3''$ for some B_3''. Then by Lemma B.27 and Lemma B.31 $\theta_1'' \circ \theta_1' \ C_1 \vdash^e C_2 \hookrightarrow B_3$ for some B_3, as required.

(iii) Let $\vdash \theta_3 : \Delta \to \Delta_{init}$ and η_2 be s.t. $\eta_2 \models \theta_3 \ C_1$.
Then by Lemma C.4 (iii) on (a) there exists $\vdash \theta_4' : \Delta \to \Delta_{init}$ s.t.

$$\theta_{3\restriction \overline{a} \cup fv_\emptyset(C_3)} \equiv_\emptyset (\theta_4' \circ \theta_1')_{\restriction \overline{a} \cup fv_\emptyset(C_3)} \quad \text{(c)}$$

$$\eta_2 \models \theta_4' \circ \theta_1' \ C_1 \quad \text{(d)}$$

$$env(B_3', \eta_2) \models \theta_4' \ C_3 \quad \text{(e)}$$

where B_3' is from (ii) above.
Then by I.H. (iii) on (b) (using θ_4' on C_3, which is appropriate by (e)) there exists $\vdash \theta_4'' : \Delta \to \Delta_{init}$ s.t.

$$\theta'_{4\restriction \overline{b}} \equiv_\emptyset (\theta_4'' \circ \theta_1'')_{\restriction \overline{b}} \quad \text{(f)}$$

$$env(B_3', \eta_2) \models \theta_4'' \circ \theta_1'' \ C_3 \quad \text{(g)}$$

$$env(B_3'', env(B_3', \eta_2)) \models \theta_4'' \ C_2 \quad \text{(h)}$$

where B_3'' is from (ii) above and $\overline{b} = \overline{a} \cup \bigcup_{a \in \overline{a}} fv_\emptyset(\theta_1' \ a) \cup fv_\emptyset(C_2)$.
(iii.1) By Lemma C.3 on (a) $fv_\emptyset(C_2) \subseteq fv_\emptyset(C_3)$. Then by (c) and (f)

$$\begin{aligned}
&\theta_{3\restriction \overline{a} \cup fv_\emptyset(C_2)} \\
&= (\theta_4' \circ \theta_1')_{\restriction \overline{a} \cup fv_\emptyset(C_2)} \\
&= (\theta_4'' \circ \theta_1'' \circ \theta_1')_{\restriction \overline{a} \cup fv_\emptyset(C_2)}
\end{aligned}$$

(iii.2) By (i) $C_2 \vdash^e \theta_1'' \circ \theta_1' \ C_1 \hookrightarrow B_2'$, so by Lemma B.27 $\theta_4'' \ C_2 \vdash^e \theta_4'' \circ \theta_1'' \circ \theta_1' \ C_1 \hookrightarrow B_2'$. Then by (h) and Lemma B.13 $env(B_2', env(B_3'', env(B_3', \eta_2))) \models \theta_4'' \circ \theta_1'' \circ \theta_1' \ C_1$. But by (i), (ii) and Lemma B.31 $env(B_2', env(B_3'', env(B_3', \eta_2)))_{\restriction names(C_1)} = \eta_2$. Thus $\eta \models \theta_4'' \circ \theta_1'' \circ \theta_1' \ C_1$.
(iii.3) Immediate from (h).

$\square$

C.2 Soundness of Type Inference

Lemma C.6 If $\theta \mid C \mid \Gamma \vdash t : \tau$ or $\theta \mid C \mid \Gamma \vdash_n t : \tau$ then $dom(\theta) \subseteq fv_{\emptyset}(\Gamma)$, $\theta\ C = C$, and $\theta\ \tau = \tau$.

Proof Easy induction. The case for SIMP requires Lemma C.3. Notice the use of restriction or ellimination in rules ISIMP and IP7. ☐

Lemma C.7 If $\Delta \vdash \Gamma$ context and $\theta \mid C \mid \Gamma \vdash t : \tau$ or $\theta \mid C \mid \Gamma \vdash_n t : \tau$ then there exist a Δ' s.t. $\Delta +\!\!+ \Delta' \vdash \theta$ subst, $\Delta +\!\!+ \Delta' \vdash C$ constraint, and $\Delta +\!\!+ \Delta' \vdash \tau : \mathsf{Type}$.

Proof By induction on derivation, using Lemma C.3 (ii) in rule ISIMP, and relying on the freshness of introduced type variables. Notice each fresh type variable is introduced at a specific kind in rules IAPP, IVAR, IP3, IP4, IP5 and IP7. ☐

Lemma C.8 If

(a) $\Delta \mid C \mid \Gamma \vdash t : \tau \hookrightarrow T$ or $\Delta \mid C \mid \Gamma \vdash_n t : \tau \hookrightarrow T[\bullet]$

(b) $\Delta \vdash D$ constraint

(c) $D \vdash^e C \hookrightarrow B$

(d) $saturate(D) \neq \emptyset$

then $\Delta \mid D \mid \Gamma \vdash t : \tau \hookrightarrow T'$ or $\Delta \mid D \mid \Gamma \vdash_n t : \tau \hookrightarrow T'[\bullet]$.
Furthermore, if $\vdash \theta : \Delta \to \Delta_{init}$ and $env(B') \models \theta\ D$ and and $\eta \models \theta\ \Gamma$ then $[\![T]\!]_{\eta +\!\!+ env(B, env(B'))} = [\![T']\!]_{\eta +\!\!+ env(B')}$ or $[\![T[U]]\!]_{\eta +\!\!+ env(B, env(B'))} = [\![T'[U]]\!]_{\eta +\!\!+ env(B')}$ for well-typed U.

Proof By induction on derivation of (a):

case APP: Let (a) be

$$\Delta \mid C \mid \Gamma \vdash t\ u : \tau \hookrightarrow T\ U$$

Then by APP

$$\Delta \mid C \mid \Gamma \vdash t : v \hookrightarrow T \quad \text{(e)}$$
$$\Delta \mid C \mid \Gamma \vdash u : v' \hookrightarrow U \quad \text{(f)}$$
$$C \vdash^e v\ eq_{\mathsf{Type}}\ v' \to \tau \hookrightarrow \mathsf{True}$$

By (c) and Lemma B.31

$$D \vdash^e v\ eq_{\mathsf{Type}}\ v' \to \tau \hookrightarrow \mathsf{True}$$

By I.H. on (e)

$$\Delta \mid D \mid \Gamma \vdash t : v \hookrightarrow T'$$

and $[\![T]\!]_{\eta +\!\!+ env(B, env(B'))} = [\![T']\!]_{\eta +\!\!+ env(B')}$.
Also, by I.H. on (f)

$$\Delta \mid D \mid \Gamma \vdash u : v' \hookrightarrow U'$$

and $[\![U]\!]_{\eta +\!\!+ env(B,env(B'))} = [\![U']\!]_{\eta +\!\!+ env(B')}$.
Then by APP

$$\Delta \mid D \mid \Gamma \vdash t\ u : \tau \hookrightarrow T'\ U'$$

and

$$\begin{array}{l} [\![T\ U]\!]_{\eta +\!\!+ env(B,env(B'))} \\ = \ \mathbf{let_E}\ v \leftarrow [\![T]\!]_{\eta +\!\!+ env(B,env(B'))} \\ \quad \mathbf{in\ case}\ v\ \mathbf{of}\ \{ \\ \quad\quad \mathsf{func} : f \rightarrow f\ [\![U]\!]_{\eta +\!\!+ env(B,env(B'))}; \\ \quad\quad \mathbf{otherwise} \rightarrow \mathbf{unit_E}\ (\mathsf{wrong} : *)\ \} \\ = \ \mathbf{let_E}\ v \leftarrow [\![T']\!]_{\eta +\!\!+ env(B')} \\ \quad \mathbf{in\ case}\ v\ \mathbf{of}\ \{ \\ \quad\quad \mathsf{func} : f \rightarrow f\ [\![U']\!]_{\eta +\!\!+ env(B')}; \\ \quad\quad \mathbf{otherwise} \rightarrow \mathbf{unit_E}\ (\mathsf{wrong} : *)\ \} \\ = [\![T'\ U']\!]_{\eta +\!\!+ env(B')} \end{array}$$

as required.

case VAR: Let (a) be

$$\Delta \mid C \mid \Gamma \vdash x/f : \tau[\overline{a \mapsto v}] \hookrightarrow \mathsf{letw}\ B''\ \mathsf{in}\ x/f\ names(D')$$

Then by VAR

$$C \vdash^e D'[\overline{a \mapsto v}] \hookrightarrow B''$$

where $(x/f : \mathtt{forall}\ \overline{a : \kappa}\ .\ D \Rightarrow \tau) \in \Gamma$ and $D' = named(D)$.
By (c) and Lemma B.31

$$D \vdash^e D'[\overline{a \mapsto v}] \hookrightarrow B'''$$

where $env(B +\!\!+ B'', env(B'))_{\restriction names(D')} = env(B''', env(B'))_{\restriction names(D')}$.
Thus by VAR

$$\Delta \mid D \mid \Gamma \vdash x/f : \tau[\overline{a \mapsto v}] \hookrightarrow \mathsf{letw}\ B'''\ \mathsf{in}\ x/f\ names(D')$$

and

$$\begin{array}{l} [\![\mathsf{letw}\ B''\ \mathsf{in}\ x/f\ names(D')]\!]_{\eta +\!\!+ env(B,env(B'))} \\ = [\![x/f\ names(D')]\!]_{\eta +\!\!+ env(B'',env(B,env(B')))} \\ = [\![x/f\ names(D')]\!]_{\eta +\!\!+ env(B +\!\!+ B'',env(B'))} \\ = [\![x/f\ names(D')]\!]_{\eta +\!\!+ env(B''',env(B'))} \\ = [\![\mathsf{letw}\ B'''\ \mathsf{in}\ x/f\ names(D')]\!]_{\eta +\!\!+ env(B')} \end{array}$$

as required.

Remaining cases are similar. □

Lemma C.9 If

(a) $\Delta \vdash C$ constraint

(b) $\Delta \vdash \Gamma$ context

(c) $\Delta \mid C \mid \Gamma \vdash t : \tau \hookrightarrow T$ or $\Delta \mid C \mid \Gamma \vdash_n t : \tau \hookrightarrow T[\bullet]$

(d) $\Delta +\!\!+ \Delta' \vdash \theta$ subst

(e) $saturate(\theta\ C) \neq \emptyset$

then $\Delta +\!\!+ \Delta' \mid \theta\ C \mid \theta\,\Gamma \vdash t : \theta\,\tau \hookrightarrow T$ or $\Delta +\!\!+ \Delta' \mid \theta\ C \mid \theta\,\Gamma \vdash_n t : \theta\,\tau \hookrightarrow T[\bullet]$.

Proof By induction on derivation of (c):

case VAR: Let (c) be

$$\Delta \mid C \mid \Gamma \vdash x/f : \tau[\overline{a \mapsto v}] \hookrightarrow \mathsf{letw}\ B\ \mathsf{in}\ x/f\ names(D')$$

Then by VAR

$$x/f : \mathtt{forall}\ \overline{a : \kappa}\ .\ D \mathtt{=>} \tau \in \Gamma \quad \text{(f)}$$

$$C \vdash^e D'[\overline{a \mapsto v}] \hookrightarrow B \quad \text{(g)}$$

$$\Delta \vdash \overline{v : \kappa} \quad \text{(h)}$$

where $D' = named(D)$.
W.l.o.g. assume $dom(\theta) \cap \overline{a} = \emptyset$. Then $\theta\ (D'[\overline{a \mapsto v}]) = (\theta\ D')[\overline{a \mapsto \theta\ v}]$ and $\theta\,(\mathtt{forall}\ \overline{a : \kappa}\ .\ D \mathtt{=>} \tau) = \mathtt{forall}\ \overline{a : \kappa}\ .\ (\theta\ D) \mathtt{=>} (\theta\ \tau)$.
By (d) and (h)

$$\Delta +\!\!+ \Delta' \vdash \overline{\theta\ v : \kappa}$$

By (g) and Lemma B.27

$$\theta\ C \vdash^e (\theta\ D')[\overline{a \mapsto \theta\ v}] \hookrightarrow B$$

By (f)

$$\mathtt{forall}\ \overline{a : \kappa}\ .\ (\theta\ D) \mathtt{=>} (\theta\ \tau) \in \theta\,\Gamma$$

Notice $(\theta\ \tau)[\overline{a \mapsto \theta\ v}] = \theta\ (\tau[\overline{a \mapsto v}])$.
Thus by VAR

$$\Delta +\!\!+ \Delta' \mid \theta\ C \mid \theta\,\Gamma \vdash x/f : \theta\ (\tau[\overline{a \mapsto v}]) \hookrightarrow \mathsf{letw}\ B\ \mathsf{in}\ x/f\ names(D')$$

as required.

case LET: Let (c) be

$$\begin{array}{l}\Delta \mid C \mid \Gamma \vdash \mathtt{let}\ x = u\ \mathtt{in}\ t : \tau \\ \quad \hookrightarrow \mathsf{let}\ x = \lambda\ names(D_2)\ .\ U\ \mathsf{in}\ T\end{array}$$

Then by LET

$$
\begin{aligned}
x \in fv(t) && \text{(f)}\\
\Delta \vdash D_1 \text{ constraint} && \text{(g)}\\
\Delta +\!\!+ \Delta'' \vdash D_2 \text{ constraint} && \text{(h)}\\
D_1 = inhs(C) && \text{(i)}\\
saturate(D_1 +\!\!+ D_2) \neq \emptyset && \text{(k)}\\
\Delta +\!\!+ \Delta'' \mid D_1 +\!\!+ D_2 \mid \Gamma \vdash u : v \hookrightarrow U && \text{(l)}\\
\Delta \mid C \mid \Gamma, x : \sigma \vdash t : \tau \hookrightarrow T && \text{(m)}
\end{aligned}
$$

where $\sigma = \texttt{forall}\ \Delta''\ .\ anon(D_2)\ \texttt{=>}\ v$.
W.l.o.g. assume $dom(\theta) \cap dom(\Delta'') = \emptyset$. Then $\theta\ (\texttt{forall}\ \Delta''\ .\ anon(D_2)\ \texttt{=>}\ v) = \texttt{forall}\ \Delta''\ .\ anon(\theta\ D_2)\ \texttt{=>}\ (\theta\ v)$.
By (d), (g) and (h) we have

$$
\begin{aligned}
\Delta +\!\!+ \Delta' \vdash \theta\ D_1 \text{ constraint} && \text{(n)}\\
\Delta +\!\!+ \Delta' +\!\!+ \Delta'' \vdash \theta\ D_2 \text{ constraint} && \text{(o)}
\end{aligned}
$$

By definition of *inheritable* we have $\theta\ D_1 = inhs(\theta\ C)$.
By (f), (m) and Lemma B.36 (ii) there exists $\vdash \theta' : \Delta'' \to \Delta$ s.t. $C \vdash^e \theta'\ D_2$. By Lemma B.27

$$\theta\ C \vdash^e \theta\ (\theta'\ D_2)$$

which, since $dom(\theta) \cap dom(\Delta'') = \emptyset$, is equivalent to

$$\theta\ C \vdash^e \theta''\ (\theta\ D_2) \qquad \text{(p)}$$

where $\theta'' = (\theta \circ \theta')_{\restriction dom(\Delta'')}$.
Thus by Lemma B.17 $saturate(\theta''\ (\theta\ D_2)) \neq \emptyset$, so by Lemma B.19 (ii)

$$saturate(\theta\ D_2) \neq \emptyset$$

By (n) and (d) $\theta'' \circ \theta\ D_1 = \theta\ D_1$, so by (i), (p) and rule CONJ

$$\theta\ C \vdash^e \theta'' \circ \theta\ (D_1 +\!\!+ D_2)$$

Hence by (e), Lemma B.17 and Lemma B.19 (ii)

$$saturate(\theta\ (D_1 +\!\!+ D_2)) \neq \emptyset$$

Now, by I.H. on (l)

$$\Delta +\!\!+ \Delta' +\!\!+ \Delta'' \mid \theta\ (D_1 +\!\!+ D_2) \mid \theta\,\Gamma \vdash u : \theta\ v \hookrightarrow U$$

and by I.H. on (m)

$$\Delta +\!\!+ \Delta' \mid \theta\ C \mid \theta\,\Gamma, x : \theta\,\sigma \vdash t : \theta\,\tau \hookrightarrow T$$

Finally, by LET

$$\Delta +\!\!+ \Delta' \mid \theta\ C \mid \theta\,\Gamma \vdash \mathtt{let}\ x = u\ \mathtt{in}\ t : \theta\,\tau \\ \hookrightarrow \mathsf{let}\ x = \lambda\ names(D_2)\ .\ U\ \mathsf{in}\ T$$

as required.

Remaining cases proceed by Lemma B.27. □

Theorem C.10 (Soundness of Inference) Let $\Delta \vdash \Gamma$ context. If (a) $\theta \mid C \mid \Gamma \vdash t : \tau$ or $\theta \mid C \mid \Gamma \vdash_n t : \tau$ and (b) $saturate(C) \neq \emptyset$ then $\Delta +\!\!+ \Delta' \mid C \mid \theta\ \Gamma \vdash t : \tau$ or $\Delta +\!\!+ \Delta' \mid C \mid \theta\,\Gamma \vdash_n t : \tau$, (where Δ' is as given in Lemma C.7).

Proof By induction on derivation of (a):

case IAPP: Let (a) be

$$\theta_2 \circ \theta_1 \mid C \mid \Gamma \vdash t\ u : b$$

Then by IAPP

$$\theta_1 \mid D \mid \Gamma \vdash t : \tau \qquad \text{(f)}$$

$$\theta_2 \mid D' \mid \theta_1\,\Gamma \vdash u : v \qquad \text{(g)}$$

where

$$C = (\theta_2\ D) +\!\!+ D' +\!\!+ (\theta_2\ \tau)\ eq_{\mathsf{Type}}\ (v\ \texttt{->}\ b)$$

and b : Type fresh.
By (f) and Lemma C.7 there exists a Δ_1 s.t. $\Delta +\!\!+ \Delta_1 \vdash \theta_1$ subst, $\Delta +\!\!+ \Delta_1 \vdash D$ constraint, and $\Delta +\!\!+ \Delta_1 \vdash \tau$: Type. Furthermore, by Lemma C.6 $dom(\theta_1) \subseteq fv_{\emptyset}(\Gamma)$.
Then by (g) and Lemma C.7 there exists a Δ_2 s.t. $\Delta +\!\!+ \Delta_1 +\!\!+ \Delta_2 \vdash \theta_2$ subst, $\Delta +\!\!+ \Delta_1 +\!\!+ \Delta_2 \vdash D'$ constraint, and $\Delta +\!\!+ \Delta_1 +\!\!+ \Delta_2 \vdash v$: Type.
By (b) and Lemma B.19 (ii) $saturate(D) \neq \emptyset$, so by I.H. on (f)

$$\Delta +\!\!+ \Delta_1 \mid D \mid \theta_1\,\Gamma \vdash t : \tau \qquad \text{(k)}$$

Similarly, by (b) and Lemma B.19 (ii) $saturate(D') \neq \emptyset$, so by I.H. on (g)

$$\Delta +\!\!+ \Delta_1 +\!\!+ \Delta_2 \mid D' \mid \theta_2\ \theta_1\ \Gamma \vdash u : v \qquad \text{(o)}$$

Let

$$\Delta' = \Delta_1 +\!\!+ \Delta_2 +\!\!+ b : \mathsf{Type}$$

By (k) and Lemma B.37

$$\Delta +\!\!+ \Delta' \mid D \mid \theta_1\,\Gamma \vdash t : \theta_2\,\tau \hookrightarrow T'$$

and since by (b) and Lemma B.19 (ii) $saturate(\theta_2\ D) \neq \emptyset$, by Lemma C.9

$$\Delta +\!\!+ \Delta' \mid \theta_2\ D \mid \theta_2 \circ \theta_1\,\Gamma \vdash t : \theta_2\,\tau$$

and since $C \vdash^e \theta_2\ D \hookrightarrow \cdot$, by Lemma C.8

$$\Delta +\!\!+ \Delta' \mid C \mid \theta_2 \circ \theta_1\,\Gamma \vdash t : \theta_2\,\tau$$

Similarly, by (o) and Lemma B.37

$$\Delta \mathbin{+\!\!+} \Delta' \mid D' \mid \theta_2 \circ \theta_1\ \Gamma \vdash u : \upsilon$$

and since $C \vdash^e D' \hookrightarrow \cdot$, by Lemma C.8

$$\Delta \mathbin{+\!\!+} \Delta' \mid C \mid \theta_2 \circ \theta_1\ \Gamma \vdash u : \upsilon$$

Then, since $C \vdash^e (\theta_2\ \tau)\ eq_{\mathsf{Type}}\ (\upsilon \texttt{ -> } b) \hookrightarrow \mathsf{True}$, by APP

$$\Delta \mathbin{+\!\!+} \Delta' \mid C \mid \theta_2 \circ \theta_1\ \Gamma \vdash t\ u : b$$

as required.

case IVAR: Let (a) be

$$\mathbf{Id} \mid C \mid \Gamma \vdash x/f : \tau[\overline{a \mapsto b}]$$

Then by IVAR

$$\begin{gathered} (x/f : \texttt{forall}\ \overline{a : \kappa}\ \texttt{.}\ D \texttt{ => } \tau) \in \Gamma \\ \overline{b : \kappa}\ \mathsf{fresh} \\ C = named(D[\overline{a \mapsto b}]) \end{gathered}$$

Let $\Delta' = \overline{b : \kappa}$ and $D' = named(C)$. Then $C \vdash^e D'[\overline{a \mapsto b}]$. Thus by VAR

$$\Delta \mathbin{+\!\!+} \Delta' \mid C \mid \Gamma \vdash x/f : \tau[\overline{a \mapsto b}]$$

as required.

case ILET: Let (a) be

$$\theta_2 \circ \theta_1 \mid C \mid \Gamma \vdash \texttt{let}\ x \texttt{ = } u\ \texttt{in}\ t : \tau$$

Then by ILET

$$x \in fv(t) \tag{f}$$
$$\theta_1 \mid D_1 \mid \Gamma \vdash u : \upsilon \tag{g}$$
$$gen(D_1 \mid \theta_1\ \Gamma \mid \upsilon) = (D_2 \mid \Delta'' \mid D_3) \tag{h}$$
$$saturate((\theta_2\ D_1) \mathbin{+\!\!+} D_4) \neq \emptyset \tag{i}$$
$$\theta_2 \mid D_4 \mid (\theta_1\ \Gamma), x : \sigma \vdash t : \tau \tag{j}$$

where $\sigma = \texttt{forall}\ \Delta''\ \texttt{.}\ anon(D_3) \texttt{ => } \upsilon$ and $C = (\theta_2\ D_2) \mathbin{+\!\!+} D_4$.
By (g) and Lemma C.7 there exists a Δ_1 s.t. $\Delta \mathbin{+\!\!+} \Delta_1 \vdash \theta_1\ \mathsf{subst}$, $\Delta \mathbin{+\!\!+} \Delta_1 \vdash D_1\ \mathsf{constraint}$, and $\Delta \mathbin{+\!\!+} \Delta_1 \vdash \upsilon : \mathsf{Type}$. Furthermore, by Lemma C.6 $dom(\theta_1) \subseteq fv_\emptyset(\Gamma)$. By definition of gen, $D_1 = D_2 \mathbin{+\!\!+} D_3$, $\Delta'' \subseteq \Delta_1$, $fv_\emptyset(D_2) \cap dom(\Delta'') = \emptyset$, $inheritable(D_2)$, and

$$dom(\Delta'') \cap fv_\emptyset(\theta_1\ \Gamma) = \emptyset \tag{k}$$

Then by (k), (j) and Lemma C.7 there exits a Δ_2 s.t. $\Delta \mathbin{+\!\!+} (\Delta_1 \setminus dom(\Delta'')) \mathbin{+\!\!+} \Delta_2 \vdash \theta_2\ \mathsf{subst}$, $\Delta \mathbin{+\!\!+} (\Delta_1 \setminus dom(\Delta'')) \mathbin{+\!\!+} \Delta_2 \vdash D_4\ \mathsf{constraint}$, and $\Delta \mathbin{+\!\!+} (\Delta_1 \setminus dom(\Delta'')) \mathbin{+\!\!+} \Delta_2 \vdash \tau : \mathsf{Type}$. Furthermore, by Lemma C.6

$$dom(\theta_2) \subseteq fv_\emptyset(\theta_1\ \Gamma) \tag{l}$$

Since all type variables in Δ_2 are created fresh, we may also assume $dom(\Delta_2) \cap dom(\Delta'') = \emptyset$. Thus

$$dom(\theta_2) \cap dom(\Delta'') = \emptyset \tag{m}$$

Let $\Delta' = (\Delta_1 \setminus dom(\Delta'')) +\!\!+ \Delta_2$. Then

$$\Delta +\!\!+ \Delta' \vdash (\theta_2\ D_2) +\!\!+ inhs(D_4)\ \textsf{constraint} \tag{n}$$

$$\Delta +\!\!+ \Delta' +\!\!+ \Delta'' \vdash \theta_2\ D_3\ \textsf{constraint} \tag{o}$$

By definition of gen, $ihns(D_2) = D_2$. Thus

$$inhs(C) = (\theta_2\ D_2) +\!\!+ inhs(D_4) \tag{p}$$

and by Lemma B.34

$$inhs(C) +\!\!+ \theta_2\ D_3 \vdash^e \theta_2\ (D_2 +\!\!+ D_3) \hookrightarrow \cdot \tag{q}$$

By (i) and Lemma B.19 $saturate(D_1) \neq \emptyset$, so by I.H. on (g)

$$\Delta +\!\!+ \Delta_1 \mid D_1 \mid \theta_1\ \Gamma \vdash u : v$$

Then by Lemma B.37

$$\Delta +\!\!+ \Delta' +\!\!+ \Delta'' \mid D_2 +\!\!+ D_3 \mid \theta_1\ \Gamma \vdash u : v$$

and by Lemma C.9

$$\Delta +\!\!+ \Delta' +\!\!+ \Delta'' \mid \theta_2\ (D_2 +\!\!+ D_3) \mid \theta_2 \circ \theta_1\ \Gamma \vdash u : \theta_2\ v$$

and (q), (i) and Lemma C.8

$$\Delta +\!\!+ \Delta' +\!\!+ \Delta'' \mid inhs(C) +\!\!+ \theta_2\ D_3 \mid \theta_2 \circ \theta_1\ \Gamma \vdash u : \theta_2\ v \tag{r}$$

Notice by (m) $\theta_2\ \texttt{forall}\ \Delta''\ .\ anon(D_3)\ \texttt{=>}\ v = \texttt{forall}\ \Delta''\ .\ anon(\theta_2\ D_3)\ \texttt{=>}\ (\theta_2\ v)$. By (b) and Lemma B.19 $saturate(D_4) \neq \emptyset$. Then by I.H. on (j)

$$\Delta +\!\!+ \Delta' \mid D_4 \mid \theta_2 \circ \theta_1\ \Gamma, x : \theta_2\ \sigma \vdash t : \tau$$

Since by CONJ $C \vdash^e D_4 \hookrightarrow \cdot$, by Lemma C.8

$$\Delta +\!\!+ \Delta' \mid C \mid \theta_2 \circ \theta_1\ \Gamma, x : \theta_2\ \sigma \vdash t : \tau \tag{s}$$

We may now apply LET using (reading from top-left to bottom-right of the rule's hypotheses) (f), (n), (o), (p), (i), (r) and (s) to give

$$\Delta +\!\!+ \Delta' \mid C \mid \theta_2 \circ \theta_1\ \Gamma \vdash t : \tau$$

as required.

case ISIMP: Let (a) be

$$(\theta_2 \circ \theta_1)_{\lceil fv_\emptyset(\Gamma)} \mid C \mid \Gamma \vdash t : \theta_2\ \tau \hookrightarrow \textsf{letw}\ B'\ \textsf{in}\ T$$

Then by ISIMP

$$\theta_1 \mid C' \mid \Gamma \vdash t : \tau \quad \text{(f)}$$
$$\langle fv_\emptyset(\theta_1\ \Gamma) \cup fv_\emptyset(\tau) \mid C' \rangle \rhd^* \langle \theta_2 \mid C \mid B' \rangle \quad \text{(g)}$$

By (f) and Lemma C.7 there exists a Δ_1 s.t. $\Delta +\!\!+ \Delta_1 \vdash \theta_1$ subst, $\Delta +\!\!+ \Delta_1 \vdash C'$ constraint, and and $\Delta +\!\!+ \Delta_1 \vdash \tau : \mathsf{Type}$. Furthermore, by Lemma C.6 $dom(\theta_1) \subseteq fv_\emptyset(\Gamma)$. Thus $fv_\emptyset(\theta_1\ \Gamma) \cup fv_\emptyset(\tau) \subseteq dom(\Delta +\!\!+ \Delta_1)$.
Then by (g) and Lemma C.3 there exists a Δ_2 s.t. $\Delta +\!\!+ \Delta_1 +\!\!+ \Delta_2 \vdash C$ constraint, and $\Delta +\!\!+ \Delta_1 +\!\!+ \Delta_2 \vdash \theta_2$ subst.
Furthermore, by Lemma C.4 (i)

$$C \vdash^e \theta_2\ C' \hookrightarrow B' \quad \text{(h)}$$

Then by (b), (h) and Lemma B.17 $saturate(\theta_2\ \ C') \neq \emptyset$, so by Lemma B.19 (ii) $saturate(C') \neq \emptyset$. Then by I.H. on (f)

$$\Delta +\!\!+ \Delta_1 \mid C' \mid \theta_1\ \Gamma \vdash t : \tau \quad \text{(i)}$$

Let $\Delta' = \Delta_1 +\!\!+ \Delta_2$. By (i) and Lemma B.37

$$\Delta +\!\!+ \Delta' \mid C' \mid \theta_1\ \Gamma \vdash t : \tau$$

and since $saturate(\theta_2\ \ C') \neq \emptyset$, by Lemma C.9

$$\Delta +\!\!+ \Delta' \mid \theta_2\ C' \mid \theta_2 \circ \theta_1\ \Gamma \vdash t : \theta_2\ \tau$$

and by (h) by Lemma C.8

$$\Delta +\!\!+ \Delta' \mid C \mid \theta_2 \circ \theta_1\ \Gamma \vdash t : \theta_2\ \tau$$

The result follows since $\theta_2 \circ \theta_1\ \Gamma = (\theta_2 \circ \theta_1)_{\restriction fv_\emptyset(\Gamma)}\ \Gamma$.

case IP7: Let (a) be

$$\theta_{\setminus b} \mid C \mid \Gamma \vdash_{n+1} \backslash x\ .\ t : (\theta\ b) \texttt{->} \tau$$

Then by IP7

$$\theta \mid C \mid \Gamma, x : b \vdash_n t : \tau \quad \text{(f)}$$
$$b : \mathsf{Type\ fresh} \quad \text{(g)}$$

By (f) and Lemma C.7 there exists a Δ' s.t. $\Delta +\!\!+ b : \mathsf{Type} +\!\!+ \Delta' \vdash \theta$ subst, $\Delta +\!\!+ b : \mathsf{Type} +\!\!+ \Delta' \vdash C$ constraint, and and $\Delta +\!\!+ b : \mathsf{Type} +\!\!+ \Delta' \vdash \tau : \mathsf{Type}$.
By (g) and idempotency of substitutions, $\Delta +\!\!+ \Delta' \vdash \theta\ b : \mathsf{Type}$.
Then by I.H. on (f)

$$\Delta +\!\!+ \Delta' \mid C \mid \theta\ \Gamma, x : \theta\ b \vdash_n t : \tau$$

and thus by P7

$$\Delta +\!\!+ \Delta' \mid C \mid \theta\ \Gamma \vdash_{n+1} \backslash x\ .\ t : (\theta\ b) \texttt{->} \tau$$

The result follows since b fresh and thus $\theta\ \Gamma = \theta_{\setminus b}\ \Gamma$.

Remaining cases are similar. □

Appendix D

Proofs for Chapter 9

D.1 Entailment

Lemma D.1 Let $\Delta\,;\overline{\Delta_{init}} \vdash^0 C$ constraint and $\Delta\,;\overline{\Delta_{init}} \vdash^0 d$ constraint and $\Delta \vdash \theta$ gsubst and $\eta \models \theta\ C$. Then

(i) $C \vdash^e d \hookrightarrow W$ implies $[\![W]\!]_\eta \in [\![\theta\ d]\!]$

(ii) $C \vdash^e \overline{w : d} \hookrightarrow \overline{w = W}$ implies $\forall i\ .\ [\![W_i]\!]_\eta \in [\![\theta\ d_i]\!]$

Proof (i) By induction on derivation of $C \vdash^e d \hookrightarrow W$.

case EXISTSRTTYPE: First notice that for any ground type v, $[\![\texttt{rttype}\ v]\!]$ is non-empty. W.l.o.g assume $dom(\theta) \cap dom(\Delta') = \emptyset$. By I.H. $[\![\mathsf{True}]\!] \in [\![\texttt{exists}\ \Delta'\ .\ \theta\ \overline{d}]\!]$. Thus there exists θ' s.t. $\Delta' \vdash \theta'$ gsubst and $[\![\theta'\ \theta\ d_i]\!]$ is non-empty for every d_i. Notice $\Delta +\!\!+ \Delta' \vdash \theta' \circ \theta$ gsubst, and thus $\theta'\ \theta\ \tau$ is ground. Then $[\![\theta'\ \theta\ (\texttt{rttype}\ \tau)]\!]$ is also non-empty. Hence $[\![\mathsf{True}]\!] \in [\![\texttt{exists}\ \Delta'\ .\ (\texttt{rttype}\ \tau,\ \overline{d})]\!]$ as required.

Remaining cases straightforward.

(ii) Straightforward. □

Lemma D.2 Let θ be a well-kinded grounding substitution. If $\texttt{true} \vdash^e \theta\ C \hookrightarrow B$ and $C \vdash^e D \hookrightarrow B'$ then $\texttt{true} \vdash^e \theta\ D \hookrightarrow B''$ and $env(B'') = env(B', env(B))_{\restriction names(D)}$.

Proof Let $C = \overline{w : c}$, $B = \overline{w = W}$, $D = \overline{w' : d}$, $B' = \overline{w' = W'}$ and $B'' = \overline{w' = W''}$. By Lemma D.1

$$\forall j\ .\ [\![W_j]\!]_\cdot \in [\![\theta\ c_j]\!]$$
$$\forall i\ .\ [\![W_i'']\!]_\cdot \in [\![\theta\ d_i]\!]$$

Then $env(B) \models \overline{w : \theta\ c}$. Again by Lemma D.1

$$\forall i\ .\ [\![W_i']\!]_{env(B)} \in [\![\theta\ d_i]\!]$$

Then since each $[\![\theta\ d_i]\!]$ must be a singleton, we have for all i

$$[\![w_i']\!]_{env(B',env(B))} = [\![W_i']\!]_{env(B)} = [\![W_i'']\!]_\cdot = [\![w_i']\!]_{env(B'')}$$

as required. □

Lemma D.3 If $\Delta\,;\overline{\Delta'} \vdash^n C/D$ constraint and $\Delta \vdash \theta$ gsubst and $C \vdash^e D$ then $\theta\ C \vdash^e \theta\ D$

Proof By induction on derivation of $C \vdash^e D$.

case EXISTSRTTYPE: We have

$$C \vdash^e \texttt{exists}\ \Delta''\ .\ (\texttt{rttype}\ \tau,\ \texttt{D})$$

and by I.H.

$$\theta\ C \vdash^e \theta\ \texttt{rttype}\ \mathit{anyground}(\Delta'', \tau)$$
$$\theta\ C \vdash^e \theta\ \texttt{exists}\ \Delta''\ .\ D$$

W.l.o.g. assume $dom(\Delta'') \cap dom(\Delta) = \emptyset$. Then

$$\theta\ C \vdash^e \texttt{rttype}\ \mathit{anyground}(\Delta'', \theta\ \tau)$$
$$\theta\ C \vdash^e \texttt{exists}\ \Delta''\ .\ \theta\ D$$

and by EXISTSRTTYPE

$$\theta\ C \vdash^e \texttt{exists}\ \Delta''\ .\ (\texttt{rttype}\ \theta\ \tau,\ \theta\ D)$$

which implies

$$\theta\ C \vdash^e \theta\ \texttt{exists}\ \Delta''\ .\ (\texttt{rttype}\ \tau,\ D)$$

as required.

case EXISTSLIFTA: We have

$$C \vdash^e \texttt{exists}\ \Delta''\ .\ (\texttt{liftable}\ a,\ D)$$

By I.H. we have

$$\theta\ C \vdash^e \theta\ \texttt{exists}\ \Delta''\ .\ D$$

W.l.o.g. assume $dom(\Delta'') \cap dom(\Delta) = \emptyset$. Then since $a \in dom(\Delta'')$, $\theta\ a = a$. Furthermore

$$\theta\ C \vdash^e \texttt{exists}\ \Delta''\ .\ \theta\ D$$

Then by EXISTSLIFTA

$$\theta\ C \vdash^e \texttt{exists}\ \Delta''\ .\ (\texttt{liftable}\ \theta\ a,\ \theta\ D)$$

and thus

$$\theta\ C \vdash^e \theta\ \texttt{exists}\ \Delta''\ .\ (\texttt{liftable}\ a,\ D)$$

as required.

Remaining cases straightforward. □

D.2 Type Soundness

Note that to ease the notation a little, in the following proofs we shall ellide the subscripts on the sets S, $\mathcal{D}_{wt}$ and $\mathcal{D}_{wd}$ used within the definiton of types.

Lemma D.4 (i) If $\overline{\Delta} \mid \overline{C} \mid \overline{\Gamma} \vdash^0 t : \tau$ then $\forall i \,.\, vars(i,t) \subseteq \overline{\Gamma}^i$

(ii) If $\overline{\Delta} \mid \overline{C} \mid \overline{\Gamma} \vdash^n_b t : \tau$ then $\forall i \,.\, vars(i-n,t) \subseteq \overline{\Gamma}^i$

Proof By straightforward induction on derivation. □

Lemma D.5 If $\forall i \,.\, \overline{\Delta} \vdash^i \overline{C}^i$ constraint and $\forall i \,.\, \overline{\Delta} \vdash^i \overline{\Gamma}^i$ context then

(i) $\overline{\Delta} \mid \overline{C} \mid \overline{\Gamma} \vdash^0 t : \tau$ implies $\overline{\Delta} \vdash^0 \tau : \mathsf{Type}$.

(ii) $\overline{\Delta} \mid \overline{C} \mid \overline{\Gamma} \vdash^{n+1}_b t : \tau$ implies $\overline{\Delta} \vdash^{n+1} \tau : \mathsf{Type}$.

Proof By straightforward induction on well-typing derivation. Rules ABS0, LETREC0, LIFT0, RUNT0, RUNU0, SPLICEU1 and their higher-staged counterparts are careful to check the well-kinding of introduced types. □

Lemma D.6 If $v \in [\![\sigma]\!]_{(\overline{\Delta},\overline{\Gamma})}$ and $(\overline{\Delta'},\overline{\Gamma'})$ extends $(\overline{\Delta},\overline{\Gamma})$ and σ is satisfiable, then $v \in [\![\sigma]\!]_{(\overline{\Delta}+\!\!+\overline{\Delta'},\overline{\Gamma}+\!\!+\overline{\Gamma'})}$

Proof

case $\sigma = \tau$ for a simple type τ: Then

$$\overline{\Delta}_{init} \,;\, \overline{\Delta} \vdash^0 \tau : \mathsf{Type} \tag{a}$$

We proceed by induction on derivation of (a):

case INT: Immediate.
case FUN: Let (a) be

$$\overline{\Delta}_{init} \,;\, \overline{\Delta} \vdash^0 \tau \texttt{ -> } \upsilon : \mathsf{Type}$$

Then by FUN

$$\overline{\Delta}_{init} \,;\, \overline{\Delta} \vdash^0 \upsilon : \mathsf{Type} \tag{b}$$

Let $\overline{\Delta_e}$ and $\overline{\Gamma_e}$ be s.t.

$$(\overline{\Delta_e},\overline{\Gamma_e}) \text{ extends } (\overline{\Delta} +\!\!+ \overline{\Delta'}, \overline{\Gamma} +\!\!+ \overline{\Gamma'}) \tag{c}$$

Then

$$(\overline{\Delta_e} +\!\!+ \overline{\Delta'}, \overline{\Gamma_e} +\!\!+ \overline{\Gamma'}) \text{ extends } (\overline{\Delta},\overline{\Gamma})$$

Since by definition

$$\begin{aligned} v &\in [\![\tau \texttt{ -> } \upsilon]\!]_{(\overline{\Delta},\overline{\Gamma})} \\ &= \bigcap \{ S \mid \ (\overline{\Delta'_e},\overline{\Gamma'_e}) \text{ extends } (\overline{\Delta},\overline{\Gamma}) \ \} \end{aligned}$$

where

$$S = \mathbf{E} \left\{ \mathsf{func} : f \;\middle|\; \begin{array}{l} f \in \mathbf{E}\,\mathcal{V} \to \mathbf{E}\,\mathcal{V}, \\ ev \in [\![\tau]\!]_{(\overline{\Delta}+\!\!+\overline{\Delta'_e},\overline{\Gamma}+\!\!+\overline{\Gamma'_e})} \\ \quad \Longrightarrow f \ ev \in [\![\upsilon]\!]_{(\overline{\Delta}+\!\!+\overline{\Delta'_e},\overline{\Gamma}+\!\!+\overline{\Gamma'_e})} \end{array} \right\}$$

Then $v = \bot$ or $v = (\mathsf{func} : f)$ s.t. if

$$ev \in [\![\tau]\!]_{(\overline{\Delta} + \overline{\Delta'} + \overline{\Delta_e}, \overline{\Gamma} + \overline{\Gamma'} + \overline{\Gamma_e})}$$

then

$$f\ ev \in [\![\upsilon]\!]_{(\overline{\Delta} + \overline{\Delta'} + \overline{\Delta_e}, \overline{\Gamma} + \overline{\Gamma'} + \overline{\Gamma_e})}$$

Since the choice of $\overline{\Delta_e}$ and $\overline{\Gamma_e}$ was arbitrary s.t. (c) holds, we have

$$v \in [\![\tau \texttt{ -> } \upsilon]\!]_{(\overline{\Delta} + \overline{\Delta'}, \overline{\Gamma} + \overline{\Gamma'})}$$

as required.

case CODET:

Let $\overline{\Delta_e}$ and $\overline{\Gamma_e}$ be s.t.

$$(\overline{\Delta_e}, \overline{\Gamma_e}) \text{ extends } (\overline{\Delta} + \overline{\Delta'}, \overline{\Gamma} + \overline{\Gamma'}) \qquad \text{(b)}$$

Then

$$(\overline{\Delta'} + \overline{\Delta_e}, \overline{\Gamma'} + \overline{\Gamma_e}) \text{ extends } (\overline{\Delta}, \overline{\Gamma})$$

We have

$$v \in \bigcap \{ S \mid (\overline{\Delta''}, \overline{\Gamma''}) \text{ extends } (\overline{\Delta}, \overline{\Gamma}) \}$$

where

$$S = \mathbf{E} \left\{ \mathsf{code} : md \;\middle|\; \begin{array}{c} md \in \mathbf{M}\ \mathcal{D}, nms \in Names_{\backslash \overline{\Gamma} + \overline{\Gamma''}} \\ \Longrightarrow md\ nms \in \mathbf{E}\ \mathcal{D}_{wt} \end{array} \right\}$$

and

$$\mathcal{D}_{wt} = \left\{ d \in \mathcal{D} \;\middle|\; \begin{array}{l} termOf(d) \text{ well-defined,} \\ \overline{\Delta} + \overline{\Delta''} \mid \mathbf{true} \mid \overline{\Gamma} + \overline{\Gamma''} \vdash^0 termOf(d) : \tau \end{array} \right\}$$

Then $v = \bot$ or $v = (\mathsf{code} : md)$ where if $nms \in Names_{\backslash \overline{\Gamma} + \overline{\Gamma'} + \overline{\Gamma_e}}$ then $md\ nms \in \mathbf{E}\ \mathcal{D}'_{wt}$ where

$$\mathcal{D}'_{wt} = \left\{ d \in \mathcal{D} \;\middle|\; \begin{array}{l} termOf(d) \text{ well-defined,} \\ \overline{\Delta} + \overline{\Delta'} + \overline{\Delta_e} \mid \mathbf{true} \mid \overline{\Gamma} + \overline{\Gamma'} + \overline{\Gamma_e} \vdash^0 termOf(d) : \tau \end{array} \right\}$$

Then since the choice of $\overline{\Delta_e}$ and $\overline{\Gamma_e}$ was arbitrary s.t. (b) holds, we have

$$v \in [\![\{\{\ \tau\ \}\}]\!]_{(\overline{\Delta} + \overline{\Delta'}, \overline{\Gamma} + \overline{\Gamma'})}$$

as required.

case CODEU: Similar to case CODET.

case IO: Let (a) be

$$\Delta_{init} ; \overline{\Delta} \vdash^0 \texttt{IO}\ \tau : \mathsf{Type}$$

Then by IO

$$\Delta_{init} ; \overline{\Delta} \vdash^0 \tau : \mathit{Type} \qquad \text{(b)}$$

Let $\overline{\Delta_e}$, $\overline{\Gamma_e}$ and nms be s.t.

$$(\overline{\Delta_e}, \overline{\Gamma_e}) \text{ extends } (\overline{\Delta} + \overline{\Delta'}, \overline{\Gamma} + \overline{\Gamma'}) \wedge nms \in Names_{\backslash \overline{\Gamma} + \overline{\Gamma'} + \overline{\Gamma_e}} \qquad \text{(c)}$$

Then

$$(\overline{\Delta'} +\!\!+ \overline{\Delta_e}, \overline{\Gamma'} +\!\!+ \overline{\Gamma_e}) \text{ extends } (\overline{\Delta}, \overline{\Gamma})$$

Thus if

$$\begin{aligned} v &\in [\![\texttt{IO}\ \tau]\!]_{(\overline{\Gamma},\overline{\Delta})} \\ &= \bigcap \{ S \mid (\overline{\Delta'_e}, \overline{\Gamma'_e}) \text{ extends } (\overline{\Delta}, \overline{\Gamma}) \} \end{aligned}$$

where

$$S = \mathbf{E} \left\{ \mathsf{cmd} : io \;\middle|\; \begin{array}{l} io \in \mathbf{MIO}\ (\mathbf{E}\ \mathcal{V}), \\ nms' \in Names_{\backslash \overline{\Gamma} +\!\!+ \overline{\Gamma'_e}} \wedge (io\ nms') \Downarrow_{\mathbf{IO}} ea \\ \quad \Longrightarrow ea \in [\![\tau]\!]_{(\overline{\Delta},\overline{\Gamma})} \end{array} \right\}$$

then $v = \bot$ or $v = (\mathsf{cmd} : io)$ and

$$(io\ nms) \Downarrow_{\mathbf{IO}} ea \Longrightarrow ea \in [\![\tau]\!]_{(\overline{\Delta},\overline{\Gamma})}$$

Then by I.H. on (b)

$$(io\ nms) \Downarrow_{\mathbf{IO}} ea \Longrightarrow ea \in [\![\tau]\!]_{(\overline{\Delta} +\!\!+ \overline{\Delta'}, \overline{\Gamma} +\!\!+ \overline{\Gamma'})}$$

Since the choice of $\overline{\Delta_e}$, $\overline{\Gamma_e}$ and nms was arbitrary s.t. (c) holds, we have

$$v \in [\![\texttt{IO}\ \tau]\!]_{(\overline{\Gamma} +\!\!+ \overline{\Gamma'}, \overline{\Delta} +\!\!+ \overline{\Gamma'})}$$

as required.

case VAR: Not possible.

case $\sigma = \texttt{forall}\ \overline{a : \kappa}\ .\ C \texttt{ => } \tau$: Then

$$\Delta_{init}\,; \overline{\Delta} \vdash^0 \texttt{forall}\ \overline{a : \kappa}\ .\ C \texttt{ => } \tau \text{ scheme} \qquad \text{(a)}$$

where C is satisfiable in $\overline{a : \kappa}$.

Let $\overline{v}$ be types s.t.

$$\begin{array}{l} \overline{\Delta_{init}} \vdash^0 \overline{v : \kappa} \\ \texttt{true} \vdash^e D[\overline{a \mapsto v}] \hookrightarrow B \end{array} \qquad \text{(b)}$$

for $D = named(C)$ and $names(D) = (w_1, \ldots, w_n)$.

By SCHEME

$$(\Delta_{init} +\!\!+ \overline{a : \kappa})\,; \overline{\Delta} \vdash^0 \tau : \textsf{Type}$$

and thus

$$\Delta_{init}\,; \overline{\Delta} \vdash^0 \tau[\overline{a \mapsto v}] : \textsf{Type} \qquad \text{(c)}$$

Then

$$\begin{aligned} v &\in [\![\texttt{forall}\ \overline{a : \kappa}\ .\ C \texttt{ => } \tau]\!]_{(\overline{\Delta},\overline{\Gamma})} \\ &= \bigcap \left\{ S \;\middle|\; \begin{array}{l} \overline{\Delta_{init}} \vdash^0 \overline{v' : \kappa}, \\ \texttt{true} \vdash^e D[\overline{a \mapsto v'}] \hookrightarrow B' \end{array} \right\} \end{aligned}$$

where

$$S = \mathbf{E}\left\{ \mathsf{tfunc}_n : f \;\middle|\; \begin{array}{l} f \in \prod_{1 \le i \le n} \mathcal{T} \to \mathbf{E}\,\mathcal{V}, \\ f\ (\llbracket w_1 \rrbracket_{env(B')}, \ldots, \llbracket w_n \rrbracket_{env(B')}) \\ \quad \in \llbracket \tau[\overline{a \mapsto v'}] \rrbracket_{(\overline{\Delta},\overline{\Gamma})} \end{array} \right\}$$

Then $v = \bot$ or $v = (\mathsf{tfunc}_n : f)$ s.t.

$$f\ (\llbracket w_1 \rrbracket_{env(B)}, \ldots, \llbracket w_n \rrbracket_{env(B)}) \in \llbracket \tau[\overline{a \mapsto v}] \rrbracket_{(\overline{\Delta},\overline{\Gamma})}$$

By I.H. on (c)

$$f\ (\llbracket w_1 \rrbracket_{env(B)}, \ldots, \llbracket w_n \rrbracket_{env(B)}) \in \llbracket \tau[\overline{a \mapsto v}] \rrbracket_{(\overline{\Delta} +\!\!+ \overline{\Delta'}, \overline{\Gamma} +\!\!+ \overline{\Gamma'})}$$

Since the choice of $\overline{v}$ was arbitrary s.t. (b) holds, we have

$$v \in \llbracket \texttt{forall}\ \overline{a : \kappa}\ \texttt{.}\ C\ \texttt{=>}\ \tau \rrbracket_{(\overline{\Delta} +\!\!+ \overline{\Delta'}, \overline{\Gamma} +\!\!+ \overline{\Gamma'})}$$

as required.

□

Lemma D.7 If $\eta \models_{(\overline{\Delta},\overline{\Gamma})} \Gamma''$ and $(\overline{\Delta'}, \overline{\Gamma'})$ extends $(\overline{\Delta}, \overline{\Gamma})$ then $\eta \models_{(\overline{\Delta} +\!\!+ \overline{\Delta'}, \overline{\Gamma} +\!\!+ \overline{\Gamma'})} \Gamma''$.

Proof By pointwise application of Lemma D.6. □

Theorem D.8

(i) If

- (a) $\Delta ; \overline{\Delta'} \mid C \mid \Gamma ; \overline{\Gamma'} \vdash^0 t : \tau \hookrightarrow T$
- (b) $\Delta \vdash \theta$ gsubst
- (c) $\texttt{true} \vdash^e \theta\ C \hookrightarrow B$
- (d) $\rho\ \overline{\Gamma'} \subseteq \overline{\Gamma_r}$
- (e) $\eta \models_{(\overline{\Delta'}, \theta \overline{\Gamma_r})} \theta\ \Gamma$

then $\llbracket T \rrbracket^0_{\eta +\!\!+ env(B)}\ \rho \in \llbracket \theta\ \tau \rrbracket_{(\overline{\Delta'}, \theta \overline{\Gamma_r})}$

(ii) If

- (a) $\Delta ; \overline{\Delta'} \mid C ; \overline{C'} \mid \Gamma ; \overline{\Gamma'} \vdash^{n+1}_b t : \tau \hookrightarrow t'$
- (b) $\Delta \vdash \theta$ gsubst
- (c) $\texttt{true} \vdash^e \theta\ C \hookrightarrow B$
- (d) $\rho\ \overline{\Gamma'} \subseteq \overline{\Gamma_r}$
- (e) $\eta \models_{(\overline{\Delta'}, \theta \overline{\Gamma_r})} \theta\ \Gamma$
- (f) $nms \in Names_{\backslash \overline{\Gamma_r} +\!\!+ \overline{\Gamma_e}}$ and $(\overline{\Delta_e}, \overline{\Gamma_e})$ extends $(\overline{\Delta'}, \theta\ \overline{\Gamma_r})$

then $\llbracket t' \rrbracket^{n+1}_{\eta +\!\!+ env(B)}\ (nms, \rho) \in \mathbf{E}\ \mathcal{D}_{wd}$ where

$$\mathcal{D}_{wd} = \left\{ d \in \mathcal{D} \;\middle|\; \begin{array}{l} termOf(d) \text{ well-defined}, \\ \forall i\ .\ vars(i - n, termOf(d)) \subseteq dom(\overline{\Gamma_r}^i) \end{array} \right\}$$

(iii) Furthermore, if the conditions (a)–(f) of (ii) hold and

(g) $b = \mathbf{tt}$

then $[\![t']\!]^{n+1}_{\eta +\!\!+ env(B)}\ (nms, \rho) \in \mathbf{E}\, \mathcal{D}_{wt}$, where, if $n > 0$ then

$$\mathcal{D}_{wt} = \mathbf{E} \left\{ d \in \mathcal{D} \;\middle|\; \begin{array}{l} termOf(d) \text{ well-defined,} \\ \overline{\Delta'} +\!\!+ \overline{\Delta_e} \mid \theta\, \overline{C'} \mid (\theta\, \overline{\Gamma_r}) +\!\!+ \overline{\Gamma_e} \vdash^{n}_{\mathbf{tt}} termOf(d) : \theta\, \tau \end{array} \right\}$$

otherwise

$$\mathcal{D}_{wt} = \mathbf{E} \left\{ d \in \mathcal{D} \;\middle|\; \begin{array}{l} termOf(d) \text{ well-defined,} \\ \overline{\Delta'} +\!\!+ \overline{\Delta_e} \mid \theta\, \overline{C'} \mid (\theta\, \overline{\Gamma_r}) +\!\!+ \overline{\Gamma_e} \vdash^{0} termOf(d) : \theta\, \tau \end{array} \right\}$$

Proof

By induction on derivation:

case FORGET1:

(ii) Let (a) be

$$\Delta\, ; \overline{\Delta'} \mid C\, ; \overline{C'}\, ; C'' \mid \Gamma\, ; \overline{\Gamma'} \vdash^{n+1}_{\mathbf{ff}} t : \tau \hookrightarrow t'$$

Then by FORGET1

$$\Delta\, ; \overline{\Delta'} \mid C\, ; \overline{C'}\, ; C'' \mid \Gamma\, ; \overline{\Gamma'} \vdash^{n+1}_{\mathbf{tt}} t : \tau \hookrightarrow t' \tag{h}$$

Then result follows directly from I.H. (ii) on (h).

case VAR0:

(i) Let (a) be

$$\Delta\, ; \overline{\Delta'} \mid C \mid \Gamma\, ; \overline{\Gamma'} \vdash^{0} x : \tau[\overline{a \mapsto v}] \hookrightarrow \mathsf{letw}\ B'\ \mathsf{in}\ x\ names(D')$$

Then by VAR0

$$(x : \mathtt{forall}\ \overline{a : \kappa}\ .\ D \Rightarrow \tau) \in \Gamma \tag{f}$$

$$\Delta\, ; \overline{\Delta'} \vdash^{0} \overline{v : \kappa} \tag{g}$$

$$C \vdash^{e} D'[\overline{a \mapsto v}] \hookrightarrow B' \tag{h}$$

where $D' = named(D)$. Let $names(D') = (w_1, \ldots, w_m)$.

By definition

$$\begin{aligned}
&[\![\textsf{letw } B' \textsf{ in } x\ names(D')]\!]^0_{\eta +\!\!+ env(B)}\ \rho \\
&= [\![x\ names(D')]\!]^0_{\eta +\!\!+ env(B', env(B))}\ \rho \\
&= (\mathbf{let_R}\ v \leftarrow (\mathbf{lift^R_E}\ ((\eta +\!\!+ env(B', env(B)))\ x)) \\
&\quad \mathbf{in\ lift^R_E}\ (\mathbf{case}\ v\ \mathbf{of}\ \{ \\
&\quad\quad \mathsf{tfunc}_m : f \to f\ ([\![w_1]\!]_{\eta +\!\!+ env(B', env(B))}, \ldots, [\![w_m]\!]_{\eta +\!\!+ env(B', env(B))}); \\
&\quad\quad \mathbf{otherwise} \to \mathbf{unit_E}\ (\mathsf{wrong} : *) \\
&\quad \})\ \rho \\
&= \mathbf{let_E}\ v \leftarrow \eta\ x \\
&\quad \mathbf{in\ case}\ v\ \mathbf{of}\ \{ \\
&\quad\quad \mathsf{tfunc}_m : f \to f\ ([\![w_1]\!]_{env(B', env(B))}, \ldots, [\![w_m]\!]_{env(B', env(B))}); \\
&\quad\quad \mathbf{otherwise} \to \mathbf{unit_E}\ (\mathsf{wrong} : *) \\
&\quad \} \\
&= (*)
\end{aligned}$$

W.l.o.g. assume $\overline{a} \cap dom(\theta) = \emptyset$. Then from (e) and (f)

$$\begin{aligned}
\eta\ x \in\ & [\![\theta\ \texttt{forall}\ \overline{a : \kappa}\ .\ D \texttt{ => } \tau]\!]_{(\overline{\Delta'}, \theta \overline{\Gamma_r})} \\
&= [\![\texttt{forall}\ \overline{a : \kappa}\ .\ (\theta\ D) \texttt{ => } (\theta\ \tau)]\!]_{(\overline{\Delta'}, \theta \overline{\Gamma_r})} \\
&= \bigcap \left\{ S \;\middle|\; \begin{array}{l} \overline{\Delta_{init}} \vdash^0 \overline{v' : \kappa}, \\ \texttt{true} \vdash^e (\theta\ D')[\overline{a \mapsto v'}] \hookrightarrow B'' \end{array} \right\}
\end{aligned} \tag{i}$$

where

$$S = \mathbf{E} \left\{ \mathsf{tfunc}_m : f \;\middle|\; \begin{array}{l} f \in \prod_{1 \le i \le n} \mathcal{T} \to \mathbf{E}\ \mathcal{V}, \\ f\ ([\![w_1]\!]_{env(B'')}, \ldots, [\![w_m]\!]_{env(B'')}) \\ \quad \in [\![(\theta\ \tau)[\overline{a \mapsto v'}]]\!]_{(\overline{\Delta'}, \theta \overline{\Gamma_r})} \end{array} \right\}$$

By (e) and definition of satisfiability

$$\texttt{true} \vdash^e \texttt{exists}\ \overline{a : \kappa}\ .\ (\theta\ D')$$

Then by Lemma 9.4, there exists $\overline{v'}$ s.t. $\overline{\Delta_{init}} \vdash^0 \overline{v' : \kappa}$ and $\texttt{true} \vdash^e (\theta\ D')[\overline{a \mapsto v'}]$.
Hence the intersection in (i) is not all of $\mathbf{E}\ \mathcal{V}$.
Hence v must be tagged by tfunc_m, and

$$\begin{aligned}
(*) = \ & \mathbf{let_E}\ (\mathsf{tfunc}_m : f) \leftarrow \eta\ x \\
& \mathbf{in}\ f\ ([\![w_1]\!]_{env(B', env(B))}, \ldots, [\![w_m]\!]_{env(B', env(B))})
\end{aligned}$$

From (c) and (h) and Lemma D.2

$$\texttt{true} \vdash^e \theta\ (D'[\overline{a \mapsto v}]) \hookrightarrow B''' \wedge \forall i\ .\ [\![w_i]\!]_{env(B''')} = [\![w_i]\!]_{env(B', env(B))} \tag{j}$$

Notice $\theta\ (D'[\overline{a \mapsto v}]) = (\theta D')[\overline{a \mapsto \theta\ v}]$.
From (b) and (g)

$$\Delta_{init}\ ; \overline{\Delta'} \vdash^0 \overline{\theta\ v : \kappa} \tag{k}$$

Then by (i), (j) and (k)

$$f \in \left\{ \mathsf{tfunc}_m : f' \;\middle|\; \begin{array}{l} f' \in \prod_{1 \le i \le n} \mathcal{T} \to \mathbf{E}\, \mathcal{V}, \\ f'\ (\overline{[\![w_1]\!]_{env(B',env(B))}, \cdots, [\![w_m]\!]_{env(B',env(B))}}) \\ \quad\quad \in [\![(\theta\ \tau)[\overline{a \mapsto (\theta\ \upsilon)}]]\!]_{(\overline{\Delta'}, \theta\, \overline{\Gamma_r})} \end{array} \right\}$$

and thus

$$(*) \in [\![(\theta\ \tau)[\overline{a \mapsto (\theta\ \upsilon)}]]\!]_{(\overline{\Delta'}, \theta\, \overline{\Gamma_r})} = [\![\theta\ (\tau[\overline{a \mapsto \upsilon}])]\!]_{(\overline{\Delta'}, \theta\, \overline{\Gamma_r})}$$

as required.

case VAR1:

(ii) Let (a) be

$$\Delta\, ;\, \overline{\Delta'} \mid C\, ;\, \overline{C'}\, ;\, C'' \mid \Gamma\, ;\, \overline{\Gamma'} \vdash^{n+1}_{\mathbf{tt}} x : \tau[\overline{a \mapsto \upsilon}] \hookrightarrow x$$

Then by VAR1

$$(x : \mathtt{forall}\ \overline{a : \kappa}\ .\ D \Rightarrow \tau) \in (\Gamma\, ;\, \overline{\Gamma'})^{n+1} \quad \text{(h)}$$

$$\Delta\, ;\, \overline{\Delta'} \vdash^{n+1} \overline{\upsilon : \kappa} \quad \text{(i)}$$

$$C'' \vdash^{e} D'[\overline{a \mapsto \upsilon}] \quad \text{(j)}$$

where $D' = named(D)$.
Notice $(\Gamma\, ;\, \overline{\Gamma'})^{n+1} = \overline{\Gamma'}^n$. Then by (d) and (h), $\rho\ x \in \overline{\Gamma_r}^n$ and $x \in dom(\rho)$. W.l.o.g. assume $\rho\ x = y$.
By definition

$$\begin{array}{l} [\![x]\!]^{n+1}_{\eta + env(B)}\ (nms, \rho) \\ = (\ \mathbf{let_N}\ res \leftarrow \mathbf{lift^N_R}\ (\mathbf{get_R}\ \text{“}x\text{”}) \\ \quad\ \mathbf{in\ unit_N}\ (\mathbf{case}\ res\ \mathbf{of}\ \{ \\ \quad\quad \mathsf{name} : nm \to \mathsf{dvar} : nm \\ \quad\quad \mathbf{otherwise} \to \mathsf{dwrong} : * \\ \quad\ \}))\ (nms, \rho) \\ =\ \ \mathbf{let_E}\ res \leftarrow \mathbf{unit_E}\ (\rho\ \text{“}x\text{”}) \\ \quad\ \mathbf{in\ unit_E}\ (\mathbf{case}\ res\ \mathbf{of}\ \{ \\ \quad\quad \mathsf{name} : nm \to (\mathsf{dvar} : nm) \\ \quad\quad \mathbf{otherwise} \to (\mathsf{dwrong} : *) \\ \quad\ \}) \\ = \mathbf{unit_E}\ (\mathsf{dvar} : \text{“}y\text{”}) \\ = (*) \end{array}$$

Then $termOf(\mathsf{dvar} : \text{“}y\text{”}) = y$ is well-defined, and $vars(0, y) = \{y\} \subseteq dom(\overline{\Gamma_r}^n)$.
Hence $(\mathsf{dvar} : \text{“}y\text{”}) \in \mathcal{D}_{wd}$, so that $(*) \in \mathbf{E}\ \mathcal{D}_{wd}$ as required.
(iii) W.l.o.g. assume $\overline{a} \cap dom(\theta) = \emptyset$.
From (f) and (h)

$$(y : \mathtt{forall}\ \overline{a : \kappa}\ .\ (\theta\ D) \Rightarrow (\theta\ \tau)) \in ((\theta\, \overline{\Gamma_r}) + \overline{\Gamma_e})^n$$

From (b) and (i)

$$\overline{\Delta'} \vdash^n \overline{\theta\, \upsilon : \kappa}$$

and thus

$$\overline{\Delta'} \mathbin{+\!\!+} \overline{\Delta_e} \vdash^n \overline{\theta\, \upsilon : \kappa}$$

From (j), and Lemma D.3

$$\theta\, C'' \vdash^e (\theta\, D')[\overline{a \mapsto \theta\, \upsilon}]$$

Then if $n > 0$, by VAR1

$$\overline{\Delta'} \mathbin{+\!\!+} \overline{\Delta_e} \mid (\theta\, \overline{C'})\,;(\theta\, C'') \mid (\theta\, \overline{\Gamma_r}) \mathbin{+\!\!+} \overline{\Gamma_e} \vdash^n_{\mathtt{tt}} y : (\theta\, \tau)[\overline{a \mapsto \theta\, \upsilon}]$$

Or, if $n = 0$, then $\overline{C'} = \cdot$ and by VAR0

$$\overline{\Delta'} \mathbin{+\!\!+} \overline{\Delta_e} \mid \theta\, C'' \mid (\theta\, \overline{\Gamma_r}) \mathbin{+\!\!+} \overline{\Gamma_e} \vdash^0 y : (\theta\, \tau)[\overline{a \mapsto \theta\, \upsilon}]$$

Notice $\mathit{termOf}(\mathsf{dvar} : \text{“}y\text{”}) = y$ and $(\theta\, \tau)[\overline{a \mapsto \theta\, \upsilon}] = \theta\, (\tau[\overline{a \mapsto \upsilon}])$. Thus $(*) \in \mathbf{E}\, \mathcal{D}_{wt}$ as required.

case ABS0:

(i) Let (a) be

$$\Delta\,; \overline{\Delta'} \mid C \mid \Gamma\,; \overline{\Gamma'} \vdash^0 \backslash x\ .\ t : (\upsilon \mathbin{\texttt{->}} \tau) \hookrightarrow \lambda x\ .\ T$$

Then by ABS0

$$\Delta\,; \overline{\Delta'} \vdash^0 \upsilon : \mathsf{Type} \tag{f}$$

$$\Delta\,; \overline{\Delta'} \mid C \mid (\Gamma\,; \overline{\Gamma'}) \mathbin{+\!\!+}^0 x : \upsilon \vdash^0 t : \tau \hookrightarrow T \tag{g}$$

Notice $(\Gamma\,; \overline{\Gamma'}) \mathbin{+\!\!+}^0 x : \upsilon = (\Gamma \mathbin{+\!\!+} x : \upsilon)\,; \overline{\Gamma'}$.
From (b) and (f)

$$\Delta_{init}\,; \overline{\Delta'} \vdash^0 \theta\, \upsilon : \mathsf{Type} \tag{h}$$

Let $\overline{\Delta_e}$ and $\overline{\Gamma_e}$ be s.t.

$$(\overline{\Delta_e}, \overline{\Gamma_e})\ \mathsf{extends}\ (\overline{\Delta'}, \theta\, \overline{\Gamma_r}) \tag{i}$$

Then by (b)

$$\theta\, \overline{\Gamma_e} = \overline{\Gamma_e}$$

By (d)

$$\rho\, \overline{\Gamma'} \subseteq (\overline{\Gamma_r} \mathbin{+\!\!+} \overline{\Gamma_e}) \tag{j}$$

Let $ev \in \mathbf{E}\, \mathcal{V}$ be s.t.

$$ev \in [\![\theta\, \upsilon]\!]_{(\overline{\Delta'} \mathbin{+\!\!+} \overline{\Delta_e}, (\theta\, \overline{\Gamma_r}) \mathbin{+\!\!+} \overline{\Gamma_e})} \tag{k}$$

and let $\eta' = \eta, x \mapsto ev$.
Then by Lemma D.7

$$\eta' \models_{(\overline{\Delta'} \mathbin{+\!\!+} \overline{\Delta_e}, (\theta\, \overline{\Gamma_r}) \mathbin{+\!\!+} \overline{\Gamma_e})} \theta\, (\Gamma \mathbin{+\!\!+} x : \upsilon) \tag{l}$$

Using (j) and (l), by I.H. (i) on (g)

$$[\![T]\!]^0_{\eta' \mathbin{+\!\!+} env(B)}\ \rho \in [\![\theta\, \tau]\!]_{(\overline{\Delta'} \mathbin{+\!\!+} \overline{\Delta_e}, (\theta\, \overline{\Gamma_r}) \mathbin{+\!\!+} \overline{\Gamma_e})}$$

By definition

$$\begin{aligned}
&[\![\lambda x \;.\; T]\!]^{0}_{\eta +\!\!+\, env(B)}\;\rho \\
&= (\;\mathbf{let_R}\, f \leftarrow \mathbf{closurefun^E_R}\;(\lambda ev\;.\;[\![T]\!]^{0}_{\eta +\!\!+\, env(B), x \mapsto ev}) \\
&\quad\;\; \mathbf{in\; unit_R}\;(\mathsf{func} : f))\;\rho \\
&= \mathbf{unit_E}\;(\mathsf{func} : \lambda ev\;.\;[\![T]\!]^{0}_{\eta +\!\!+\, env(B), x \mapsto ev}\;\rho) \\
&= \mathbf{unit_E}\;(\mathsf{func} : \lambda ev\;.\;[\![T]\!]^{0}_{\eta' +\!\!+\, env(B)}\;\rho) \\
&= (*)
\end{aligned}$$

Since the choice of ev was arbitrary s.t. (k) holds, we have

$$(*) \in \mathbf{E}\left\{ \mathsf{func} : f \;\middle|\; \begin{array}{l} f \in \mathbf{E}\,\mathcal{V} \to \mathbf{E}\,\mathcal{V}, \\ ev \in [\![\theta\, \upsilon]\!]_{(\overline{\Delta'} +\!\!+ \overline{\Delta_e}, (\theta\, \overline{\Gamma_r}) +\!\!+ \overline{\Gamma_e}} \\ \quad \Longrightarrow f\; ev \in [\![\theta\, \tau]\!]_{(\overline{\Delta'} +\!\!+ \overline{\Delta_e}, (\theta\, \overline{\Gamma_r}) +\!\!+ \overline{\Gamma_e})} \end{array} \right\}$$

Furthermore, since the choice of $\overline{\Delta_e}$ and $\overline{\Gamma_e}$ was arbitrary s.t. (i) holds, we have

$$(*) \in \bigcap \{ S \mid \; (\overline{\Delta'_e}, \overline{\Gamma'_e}) \text{ extends } (\overline{\Delta'}, \theta\, \overline{\Gamma_r}) \;\}$$

where

$$S = \mathbf{E}\left\{ \mathsf{func} : f \;\middle|\; \begin{array}{l} f \in \mathbf{E}\,\mathcal{V} \to \mathbf{E}\,\mathcal{V}, \\ ev \in [\![\theta\, \upsilon]\!]_{(\overline{\Delta'} +\!\!+ \overline{\Delta'_e}, (\theta\, \overline{\Gamma_r}) +\!\!+ \overline{\Gamma'_e})} \\ \quad \Longrightarrow f\; ev \in [\![\theta\, \tau]\!]_{(\overline{\Delta'} +\!\!+ \overline{\Delta'_e}, (\theta\, \overline{\Gamma_r}) +\!\!+ \overline{\Gamma'_e})} \end{array} \right\}$$

Thus

$$(*) \in [\![\theta\, \upsilon \;\texttt{->}\; \theta\, \tau]\!]_{(\overline{\Delta'}, \theta\, \overline{\Gamma_r})}$$

and the result follows from $\theta\, \upsilon \;\texttt{->}\; \theta\, \tau = \theta\, (\upsilon \;\texttt{->}\; \tau)$.

case ABS1:

(ii) Let (a) be

$$\Delta\, ; \overline{\Delta'} \mid C\, ; \overline{C'}\, ; C'' \mid \Gamma\, ; \overline{\Gamma'} \vdash^{n+1}_{b} \backslash x \;.\; t : (\upsilon \;\texttt{->}\; \tau) \hookrightarrow \backslash x \;.\; t'$$

Then by ABS1:

$$\Delta\, ; \overline{\Delta'} \mid C\, ; \overline{C'}\, ; C'' \mid (\Gamma\, ; \overline{\Gamma'}) +\!\!+^{n+1} x : \upsilon \vdash^{n+1}_{b} t : \tau \hookrightarrow t' \qquad \text{(h)}$$

$$\Delta\, ; \overline{\Delta'} \vdash^{n+1} \upsilon : \mathsf{Type} \qquad \text{(i)}$$

Notice $(\Gamma\, ; \overline{\Gamma'}) +\!\!+^{n+1} x : \upsilon = \Gamma\, ; (\overline{\Gamma'} +\!\!+^{n} x : \upsilon)$.

W.l.o.g. assume nms = "y" : nms', where by (f) $y \notin dom(\overline{\Gamma_r} +\!\!+ \overline{\Gamma_e})$. Let $\overline{\Gamma'_r} = \overline{\Gamma_r} +\!\!+^{n} y : \upsilon$ and $\rho' = \rho[x \mapsto y]$. (Note that this renaming of x to y may override a previous renaming of x of ρ).

Since nms contains only distinct variable names,

$$nms' \in Names_{\backslash(\overline{\Gamma_r} +\!\!+ \overline{\Gamma_e} +\!\!+^{n} y:\upsilon)} = Names_{\backslash \overline{\Gamma'_r} +\!\!+ \overline{\Gamma_e}}$$

and

$$(\overline{\Delta_e}, \overline{\Gamma_e}) \text{ extends } (\overline{\Delta'}, \theta\, \overline{\Gamma'_r})$$

By (d)

$$\rho' \, (\overline{\Gamma'} +\!\!+^n \, x : v) \subset \overline{\Gamma_r} +\!\!+^n \, y : v = \overline{\Gamma'_r}$$

From (e) and Lemma D.7

$$\eta \models_{(\overline{\Delta'}, \theta \overline{\Gamma'_r})} \theta \, \Gamma$$

Hence by I.H. (ii) on (h)

$$[\![t']\!]^{n+1}_{\eta +\!\!+ env(B)} \, (nms, \rho') \in \mathbf{E} \, \mathcal{D}'_{wd} \tag{j}$$

where

$$\mathcal{D}'_{wd} = \left\{ d' \in \mathcal{D} \;\middle|\; \begin{array}{l} termOf(d') \text{ well-defined}, \\ \forall i \, . \, vars(i - n, termOf(d')) \subseteq dom(\overline{\Gamma'_r}^i) \end{array} \right\}$$

By definition

$$\begin{array}{l} [\![\backslash x \; . \; t']\!]^{n+1}_{\eta +\!\!+ env(B)} \, (nms, \rho) \\ = (\, \mathbf{let_N} \, (nm, d) \leftarrow \mathbf{rename_N} \text{ ``}x\text{''} \; [\![t']\!]^{n+1}_{\eta +\!\!+ env(B)} \\ \quad \mathbf{in} \; \mathbf{unit_N} \, (\mathsf{dabs} : (nm, d))) \, (nms, \rho) \\ = \mathbf{let_E} \; d \leftarrow [\![t']\!]^{n+1}_{\eta +\!\!+ env(B)} \, (nms, \rho[x \mapsto y]) \; \mathbf{in} \; \mathbf{unit_E} \, (\mathsf{dabs} : (\text{``}y\text{''}, d)) \\ = \mathbf{let_E} \; d \leftarrow [\![t']\!]^{n+1}_{\eta +\!\!+ env(B)} \, (nms, \rho') \; \mathbf{in} \; \mathbf{unit_E} \, (\mathsf{dabs} : (\text{``}y\text{''}, d)) \\ = (*) \end{array}$$

From (j) $d \in \mathcal{D}'_{wd}$. Then $termOf(\mathsf{dabs} : (\text{``}y\text{''}, d)) = \backslash y \; . \; termOf(d)$ is well-defined, and $vars(0, \backslash y \; . \; termOf(d)) = vars(0, termOf(d)) \setminus \{y\} \subseteq dom(\overline{\Gamma_r}^n)$. Hence $(\mathsf{dabs} : (\text{``}y\text{''}, d)) \in \mathcal{D}_{wd}$, so that $(*) \in \mathbf{E} \, \mathcal{D}_{wd}$ as required.
(iii) Furthermore, if $b = \mathbf{tt}$ then by I.H. (iii) on (h)

$$[\![t']\!]^{n+1}_{\eta +\!\!+ env(B)} \, (nms, \rho') \in \mathbf{E} \, \mathcal{D}'_{wt}$$

where if $n > 0$ then

$$\mathcal{D}'_{wt} = \mathbf{E} \left\{ d' \in \mathcal{D} \;\middle|\; \begin{array}{l} termOf(d') \text{ well-defined}, \\ \overline{\Delta'} +\!\!+ \overline{\Delta_e} \mid \theta \, (\overline{C'} \, ; \, C'') \mid (\theta \, \overline{\Gamma'_r}) +\!\!+ \overline{\Gamma_e} \vdash^n_{\mathbf{tt}} termOf(d') : \theta \, \tau \end{array} \right\}$$

otherwise

$$\mathcal{D}'_{wt} = \mathbf{E} \left\{ d' \in \mathcal{D} \;\middle|\; \begin{array}{l} termOf(d') \text{ well-defined}, \\ \overline{\Delta'} +\!\!+ \overline{\Delta_e} \mid \theta \, (\overline{C'} \, ; \, C'') \mid (\theta \, \overline{\Gamma'_r}) +\!\!+ \overline{\Gamma_e}) \vdash^0 termOf(d') : \theta \, \tau \end{array} \right\}$$

Thus $d \in \mathcal{D}'_{wt}$.
From (b) and (i)

$$\overline{\Delta'} \vdash^n \theta \, v : \mathsf{Type}$$

Then, if $n > 0$ by ABS1

$$\overline{\Delta'} +\!\!+ \overline{\Delta_e} \mid \theta \, (\overline{C'} \, ; \, C'') \mid \theta \, \overline{\Gamma_r} +\!\!+ \overline{\Gamma_e} \vdash^n_{\mathbf{tt}} \backslash y \; . \; termOf(d) : \theta \, v \; \texttt{->} \; \theta \, \tau$$

Or if $n = 0$, $\overline{C'} = \cdot$ and by ABS0

$$\overline{\Delta'} \mathbin{+\!\!+} \overline{\Delta_e} \mid \theta\ C'' \mid \theta\ \overline{\Gamma_r} \mathbin{+\!\!+} \overline{\Gamma_e} \vdash^0 \backslash y\ .\ termOf(d) : \theta\ \upsilon \text{ -> } \theta\ \tau$$

Then since $\theta\ \upsilon \text{ -> } \theta\ \tau = \theta\ (\upsilon \text{ -> } \tau)$ we have $(\mathsf{dabs} : (\text{"}y\text{"}, d)) \in \mathcal{D}_{wt}$ and thus $(*) \in \mathbf{E}\ \mathcal{D}_{wt}$ as required.

case APP0:

(i) Let (a) be

$$\Delta\,;\overline{\Delta'} \mid C \mid \Gamma\,;\overline{\Gamma'} \vdash^0 t\ u : \tau \hookrightarrow T\ U$$

Then by APP0

$$\Delta\,;\overline{\Delta'} \mid C \mid \Gamma\,;\overline{\Gamma'} \vdash^0 t : (\upsilon \text{ -> } \tau) \hookrightarrow T \tag{f}$$

$$\Delta\,;\overline{\Delta'} \mid C \mid \Gamma\,;\overline{\Gamma'} \vdash^0 u : \upsilon \hookrightarrow U \tag{g}$$

By definition

$$\begin{aligned}
& [\![T\ U]\!]^0_{\eta \mathbin{+\!\!+} env(B)}\ \rho \\
= \ & \mathbf{let_R}\ v \leftarrow [\![T]\!]^0_{\eta \mathbin{+\!\!+} env(B)} \\
& \mathbf{in\ let_R}\ ev \leftarrow \mathbf{closure^E_R}\ [\![U]\!]^0_{\eta \mathbin{+\!\!+} env(B)} \\
& \mathbf{in\ lift^R_E}\ (\mathbf{case}\ v\ \mathbf{of}\ \{ \\
& \quad \mathsf{func} : f \to f\ ev; \\
& \quad \mathbf{otherwise} \to \mathbf{unit_E}\ (\mathsf{wrong} : *) \\
& \})\ \rho \\
= \ & \mathbf{let_E}\ v \leftarrow [\![T]\!]^0_{\eta \mathbin{+\!\!+} env(B)}\ \rho \\
& \mathbf{in\ let_E}\ ev \leftarrow \mathbf{unit_E}\ [\![U]\!]^0_{\eta \mathbin{+\!\!+} env(B)}\ \rho \\
& \mathbf{in\ case}\ v\ \mathbf{of}\ \{ \\
& \quad \mathsf{func} : f \to f\ ev; \\
& \quad \mathbf{otherwise} \to \mathbf{unit_E}\ (\mathsf{wrong} : *) \\
& \} \\
= \ & \mathbf{let_E}\ v \leftarrow [\![T]\!]^0_{\eta \mathbin{+\!\!+} env(B)}\ \rho \\
& \mathbf{in\ case}\ v\ \mathbf{of}\ \{ \\
& \quad \mathsf{func} : f \to (f\ [\![U]\!]^0_{\eta \mathbin{+\!\!+} env(B)}\ \rho); \\
& \quad \mathbf{otherwise} \to \mathbf{unit_E}\ (\mathsf{wrong} : *) \\
& \} \\
= \ & (*)
\end{aligned}$$

By I.H. (i) on (g)

$$[\![U]\!]^0_{\eta \mathbin{+\!\!+} env(B)}\ \rho \in [\![\theta\ \upsilon]\!]_{(\overline{\Delta'}, \theta \overline{\Gamma_r})}$$

By I.H. (i) on (f)

$$\begin{aligned}
[\![T]\!]^0_{\eta +\!\!+ env(B)}\ \rho \in [\![\theta\ (\upsilon \text{ -> } \tau)]\!]_{(\overline{\Delta'}, \theta\,\overline{\Gamma_r})} \\
&= [\![(\theta\ \upsilon) \text{ -> } (\theta\ \tau)]\!]_{(\overline{\Delta'}, \theta\,\overline{\Gamma_r})} \\
&= \bigcap \{ S \mid\ (\overline{\Delta_e}, \overline{\Gamma_e}) \text{ extends } (\overline{\Delta'}, \theta\,\overline{\Gamma_r})\ \}
\end{aligned}$$

where

$$S = \mathbf{E} \left\{ \mathsf{func} : f \;\middle|\; \begin{array}{l} f \in \mathbf{E}\,\mathcal{V} \to \mathbf{E}\,\mathcal{V}, \\ ev \in [\![\theta\ \upsilon]\!]_{(\overline{\Delta'} +\!\!+ \overline{\Delta_e}, (\theta\,\overline{\Gamma_r}) +\!\!+ \overline{\Gamma_e})} \\ \quad \Longrightarrow f\ ev \in [\![\theta\ \tau]\!]_{(\overline{\Delta'} +\!\!+ \overline{\Delta_e}, (\theta\,\overline{\Gamma_r}) +\!\!+ \overline{\Gamma_e})} \end{array} \right\}$$

Thus υ is tagged by func and

$$(*) = \mathbf{let_E}\ (\mathsf{func} : f) \leftarrow [\![T]\!]^0_{\eta +\!\!+ env(B)}\ \rho\ \mathbf{in}\ f\ ([\![U]\!]^0_{\eta +\!\!+ env(B)}\ \rho)$$

Taking $\overline{\Delta_e} = \overline{\Delta_{init}}$ and $\overline{\Gamma_e} = \overline{\Gamma_{init}}$, we have

$$f\ ([\![U]\!]^0_{\eta +\!\!+ env(B)}\ \rho) \in [\![\theta\ \tau]\!]_{(\overline{\Delta'}, \theta\,\overline{\Gamma_r})}$$

Thus $(*) \in [\![\theta\ \tau]\!]_{(\overline{\Delta'}, \theta\,\overline{\Gamma_r})}$ as required.

case APP1:

(ii) Let (a) be

$$\Delta\,; \overline{\Delta'} \mid C\,; \overline{C'}\,; C'' \mid \Gamma\,; \overline{\Gamma'} \vdash^{n+1}_b t\ u \hookrightarrow t'\ u'$$

By APP1

$$\Delta\,; \overline{\Delta'} \mid C\,; \overline{C'}\,; C'' \mid \Gamma\,; \overline{\Gamma'} \vdash^{n+1}_b t : (\upsilon \text{ -> } \tau) \hookrightarrow t' \tag{h}$$

$$\Delta\,; \overline{\Delta'} \mid C\,; \overline{C'}\,; C'' \mid \Gamma\,; \overline{\Gamma'} \vdash^{n+1}_b u : \upsilon \hookrightarrow u' \tag{i}$$

By definition

$$\begin{aligned}
& [\![t'\ u']\!]^{n+1}_{\eta +\!\!+ env(B)}\ (nms, \rho) \\
= &\ (\ \mathbf{let_N}\ d \leftarrow [\![t']\!]^{n+1}_{\eta +\!\!+ env(B)} \\
&\quad \mathbf{in}\ \mathbf{let_N}\ d' \leftarrow [\![u']\!]^{n+1}_{\eta +\!\!+ env(B)} \\
&\quad \mathbf{in}\ \mathbf{unit_N}\ (\mathsf{dapp} : (d, d')))\ (nms, \rho) \\
= &\ \mathbf{let_E}\ d \leftarrow [\![t']\!]^{n+1}_{\eta +\!\!+ env(B)}\ (nms, \rho) \\
&\ \mathbf{in}\ \mathbf{let_E}\ d' \leftarrow [\![u']\!]^{n+1}_{\eta +\!\!+ env(B)}\ (nms, \rho) \\
&\ \mathbf{in}\ \mathbf{unit_E}\ (\mathsf{dapp} : (d, d')) \\
= &\ (*)
\end{aligned}$$

By I.H. (ii) on (h) and (i)

$$[\![t']\!]^{n+1}_{\eta +\!\!+ env(B)}\ (nms, \rho) \in \mathbf{E}\ \mathcal{D}'_{wd}$$

$$[\![u']\!]^{n+1}_{\eta +\!\!+ env(B)}\ (nms, \rho) \in \mathbf{E}\ \mathcal{D}''_{wd}$$

where $\mathcal{D}'_{wd} = \mathcal{D}''_{wd} = \mathcal{D}_{wd}$.
Thus $termOf(\mathsf{dapp} : (d, d')) = termOf(d)\ termOf(d')$ is well-defined, and $\forall i\ .\ vars(i - n, termOf(d)\ \ termOf(d')) = vars(i - n, termOf(d)) \cup vars(i - n, termOf(d')) \subseteq dom(\overline{\Gamma_r}^i)$. Thus $(\mathsf{dapp} : (d, d')) \in \mathcal{D}_{wd}$, so that $(*) \in \mathbf{E}\ \mathcal{D}_{wd}$ as required.
(iii) Furthermore, if $b = \mathbf{tt}$ then by I.H. (iii) on (h) and (i)

$$[\![t']\!]^{n+1}_{\eta +\!\!+ env(B)}\ (nms, \rho) \in \mathbf{E}\ \mathcal{D}'_{wt}$$
$$[\![u']\!]^{n+1}_{\eta +\!\!+ env(B)}\ (nms, \rho) \in \mathbf{E}\ \mathcal{D}''_{wt}$$

where if $n > 0$ then

$$\mathcal{D}'_{wt} = \left\{ d'' \in \mathcal{D} \,\middle|\, \begin{array}{l} termOf(d'') \text{ well-defined}, \\ \overline{\Delta'} +\!\!+ \overline{\Delta_e} \mid \theta\ (\overline{C'}\ ;\ C'') \mid (\theta\ \overline{\Gamma_r}) +\!\!+ \overline{\Gamma_e} \vdash^n_{\mathbf{tt}} termOf(d'') : \theta\ (\upsilon \texttt{ -> } \tau) \end{array} \right\}$$

otherwise

$$\mathcal{D}'_{wt} = \left\{ d'' \in \mathcal{D} \,\middle|\, \begin{array}{l} termOf(d'') \text{ well-defined}, \\ \overline{\Delta'} +\!\!+ \overline{\Delta_e} \mid \theta\ C'' \mid (\theta\ \overline{\Gamma_r}) +\!\!+ \overline{\Gamma_e} \vdash^0 termOf(d'') : \theta\ (\upsilon \texttt{ -> } \tau) \end{array} \right\}$$

and $d \in \mathcal{D}'_{wt}$.
Similarly, if $n > 0$ then

$$\mathcal{D}''_{wt} = \mathbf{E} \left\{ d''' \in \mathcal{D} \,\middle|\, \begin{array}{l} termOf(d''') \text{ well-defined}, \\ \overline{\Delta'} +\!\!+ \overline{\Delta_e} \mid \theta\ (\overline{C'}\ ;\ C'') \mid (\theta\ \overline{\Gamma_r}) +\!\!+ \overline{\Gamma_e} \vdash^n_{\mathbf{tt}} termOf(d''') : \theta\ \upsilon \end{array} \right\}$$

otherwise

$$\mathcal{D}''_{wt} = \mathbf{E} \left\{ d''' \in \mathcal{D} \,\middle|\, \begin{array}{l} termOf(d''') \text{ well-defined}, \\ \overline{\Delta'} +\!\!+ \overline{\Delta_e} \mid \theta\ C'' \mid (\theta\ \overline{\Gamma_r}) +\!\!+ \overline{\Gamma_e} \vdash^0 termOf(d''') : \theta\ \upsilon \end{array} \right\}$$

and $d' \in \mathcal{D}''_{wt}$.
Notice $\theta\ (\upsilon \texttt{ -> } \tau) = (\theta\ \upsilon) \texttt{ -> } (\theta\ \tau)$. Then if $n > 0$, by APP1

$$\overline{\Delta'} +\!\!+ \overline{\Delta_e} \mid \theta\ (\overline{C'}\ ;\ C'') \mid (\theta\ \overline{\Gamma_r}) +\!\!+ \overline{\Gamma_e} \vdash^n_{\mathbf{tt}} termOf(d)\ termOf(d') : \theta\ \tau$$

or if $n = 0$, by APP0

$$\overline{\Delta'} +\!\!+ \overline{\Delta_e} \mid \theta\ C'' \mid (\theta\ \overline{\Gamma_r}) +\!\!+ \overline{\Gamma_e} \vdash^0 termOf(d)\ termOf(d') : \theta\ \tau$$

Hence $(*) \in \mathbf{E}\ \mathcal{D}_{wt}$ as required.

case DEFERT0:
(i) Let (a) be

$$\Delta\ ;\ \overline{\Delta'} \mid C \mid \Gamma\ ;\ \overline{\Gamma'} \vdash^0 \{\{\ t\ \}\} : \{\{\ \tau\ \}\} \hookrightarrow \langle t' \rangle$$

Then by DEFERT0

$$\Delta\ ;\ \overline{\Delta'} \mid C\ ;\ \mathsf{true} \mid \Gamma\ ;\ \overline{\Gamma'} \vdash^1_{\mathbf{tt}} t : \tau \hookrightarrow t' \tag{f}$$

Let $\overline{\Delta_e}$, $\overline{\Gamma_e}$ be s.t.

$$(\overline{\Delta_e}, \overline{\Gamma_e}) \text{ extends } (\overline{\Delta'}, \theta\ \overline{\Gamma_r}) \tag{g}$$

and nms s.t.

$$nms \in Names_{\backslash \overline{\Gamma_r} +\!\!+ \overline{\Gamma_e}} \tag{h}$$

Then by I.H. (iii) on (f)

$$[\![t']\!]^1_{\eta +\!\!+ env(B)}\ (nms, \rho) \in \mathbf{E}\ \mathcal{D}'_{wt}$$

where

$$\mathcal{D}'_{wt} = \mathbf{E}\ \left\{ d \in \mathcal{D} \;\middle|\; \begin{array}{l} termOf(d) \text{ well-defined}, \\ \overline{\Delta'} +\!\!+ \overline{\Delta_e} \mid \mathbf{true} \mid (\theta\ \overline{\Gamma_r}) +\!\!+ \overline{\Gamma_e} \vdash^0 termOf(d) : \theta\ \tau \end{array} \right\}$$

By definition

$$\begin{aligned} & [\![\langle t' \rangle]\!]^0_{\eta +\!\!+ env(B)}\ \rho \\ &= (\mathbf{let_R}\ md \leftarrow \mathbf{closure^M_N}\ [\![t']\!]^1_{\eta +\!\!+ env(B)}\ \mathbf{in}\ \mathbf{unit_R}\ (\mathsf{code} : md))\ \rho \\ &= \mathbf{let_E}\ md \leftarrow \mathbf{unit_E}\ (\lambda nms\ .\ [\![t']\!]^1_{\eta +\!\!+ env(B)}\ (nms, \rho))\ \mathbf{in}\ \mathbf{unit_E}\ (\mathsf{code} : md) \\ &= \mathbf{unit_E}\ (\mathsf{code} : \lambda nms\ .\ [\![t']\!]^1_{\eta +\!\!+ env(B)}\ (nms, \rho)) \\ &= (*) \end{aligned}$$

Hence, since *nms* is arbitary s.t. (h) holds, we have

$$(*) \in \mathbf{E}\ \left\{ \mathsf{code} : md \;\middle|\; \begin{array}{c} md \in \mathbf{M}\ \mathcal{D}, nms \in Names_{\backslash \overline{\Gamma_r} +\!\!+ \overline{\Gamma_e}} \\ \Longrightarrow md\ nms \in \mathbf{E}\ \mathcal{D}'_{wt} \end{array} \right\}$$

Furthermore, since $\overline{\Delta_e}$ and $\overline{\Gamma_e}$ are arbitrary s.t. (g) holds, we have

$$(*) \in \bigcap \{ S \mid (\overline{\Delta'_e}, \overline{\Gamma'_e})\ \mathsf{extends}\ (\overline{\Delta'}, \theta\ \overline{\Gamma_r}) \}$$

where

$$S = \mathbf{E}\ \left\{ \mathsf{code} : md \;\middle|\; \begin{array}{c} md \in \mathbf{M}\ \mathcal{D}, nms \in Names_{\backslash \overline{\Gamma_r} +\!\!+ \overline{\Gamma'_e}} \\ \Longrightarrow md\ nms \in \mathbf{E}\ \mathcal{D}_{wt} \end{array} \right\}$$

and

$$\mathcal{D}_{wt} = \left\{ d \in \mathcal{D} \;\middle|\; \begin{array}{l} termOf(d) \text{ well-defined}, \\ \overline{\Delta'} +\!\!+ \overline{\Delta'_e} \mid \mathbf{true} \mid (\theta\ \overline{\Gamma_r}) +\!\!+ \overline{\Gamma'_e} \vdash^0 termOf(d) : \theta\ \tau \end{array} \right\}$$

Thus

$$(*) \in [\![\{\!\{\ \theta\ \tau\ \}\!\}]\!]_{(\overline{\Delta'}, \theta \overline{\Gamma_r})} = [\![\theta\ \{\!\{\ \tau\ \}\!\}]\!]_{(\overline{\Delta'}, \theta \overline{\Gamma_r})}$$

as required.

case DEFERT1:

(ii) Let (a) be

$$\Delta\ ; \overline{\Delta'} \mid C\ ; \overline{C'}\ ; C'' \mid \Gamma\ ; \overline{\Gamma'} \vdash^{n+1}_{\mathbf{tt}} \{\!\{\ t\ \}\!\} : \{\!\{\ \tau\ \}\!\} \hookrightarrow \{\!\{\ t'\ \}\!\}$$

Then by DEFERT1

$$\Delta\ ; \overline{\Delta'} \mid C\ ; \overline{C'}\ ; C''\ ; \mathbf{true} \mid \Gamma\ ; \overline{\Gamma'} \vdash^{n+2}_{\mathbf{tt}} t : \tau \hookrightarrow t' \qquad \text{(h)}$$

By definition

$$\begin{aligned}
&[\![\{\!\{\ t'\ \}\!\}]\!]^{n+1}_{\eta +\!\!+ env(B)}\ (nms, \rho) \\
&= (\mathbf{let_N}\ d \leftarrow [\![t']\!]^{n+2}_{\eta +\!\!+ env(B)}\ \mathbf{in}\ \mathbf{unit_N}\ (\mathsf{ddeft} : d))\ (nms, \rho) \\
&= \mathbf{let_E}\ d \leftarrow [\![t']\!]^{n+2}_{\eta +\!\!+ env(B)}\ (nms, \rho)\ \mathbf{in}\ \mathbf{unit_E}\ (\mathsf{ddeft} : d) \\
&= (*)
\end{aligned}$$

Then by I.H. (ii) on (h)

$$[\![t']\!]^{n+2}_{\eta +\!\!+ env(B)}\ (nms, \rho) \in \mathbf{E}\ \mathcal{D}'_{wd}$$

where

$$\mathcal{D}'_{wd} = \left\{ d \in \mathcal{D} \left|\ \begin{array}{l} termOf(d) \text{ well-defined}, \\ \forall i\ .\ vars(i - (n+1), termOf(d)) \subseteq dom(\overline{\Gamma_r}^{\,i}) \end{array} \right. \right\}$$

Since $termOf(\mathsf{ddeft} : d) = \{\!\{\ termOf(d)\ \}\!\}$ is well-defined and $\forall i\ .\ vars(i - n, \{\!\{\ termOf(d)\ \}\!\}) = vars(i - (n+1), termOf(d))$, we have $(\mathsf{ddeft} : d) \in \mathcal{D}_{wd}$ and so $(*) \in \mathcal{D}_{wd}$ as required.
(iii) Furthermore, by I.H. (iii) on (h)

$$[\![t']\!]^{n+2}_{\eta +\!\!+ env(B)}\ (nms, \rho) \in \mathbf{E}\ \mathcal{D}'_{wt}$$

where

$$\mathcal{D}'_{wt} = \left\{ d' \in \mathcal{D} \left|\ \begin{array}{l} termOf(d') \text{ well-defined}, \\ \overline{\Delta'} +\!\!+ \overline{\Delta_e} \mid \theta\ (\overline{C'}\ ;\ C''\ ;\ \mathtt{true}) \mid (\theta\ \overline{\Gamma_r}) +\!\!+ \overline{\Gamma_e} \vdash^{n+1}_{\mathbf{tt}} termOf(d') : \theta\ \tau \end{array} \right. \right\}$$

Hence $d \in \mathcal{D}'_{wt}$.
Then, if $n > 0$, by DEFERT1

$$\overline{\Delta'} +\!\!+ \overline{\Delta_e} \mid \theta\ (\overline{C'}\ ;\ C'') \mid (\theta\ \overline{\Gamma_r}) +\!\!+ \overline{\Gamma_e} \vdash^{n}_{\mathbf{tt}} \{\!\{\ termOf(d)\ \}\!\} : \{\!\{\ \theta\ \tau\ \}\!\}$$

Or, if $n = 0$, then $\overline{C'} = \cdot$ and by DEFERT0

$$\overline{\Delta'} +\!\!+ \overline{\Delta_e} \mid \theta\ C'' \mid (\theta\ \overline{\Gamma_r}) +\!\!+ \overline{\Gamma_e} \vdash^{0} \{\!\{\ termOf(d)\ \}\!\} : \{\!\{\ \theta\ \tau\ \}\!\}$$

Since $\{\!\{\ termOf(d)\ \}\!\} = termOf(\mathsf{ddeft} : d)$ and $\{\!\{\ \theta\ \tau\ \}\!\} = \theta\ \{\!\{\ \tau\ \}\!\}$, we have $(\mathsf{ddeft} : d) \in \mathcal{D}_{wt}$, and so $(*) \in \mathbf{E}\ \mathcal{D}_{wt}$ as required.

case DEFERU0:
(i) Let (a) be

$$\Delta\ ;\ \overline{\Delta'} \mid C \mid \Gamma\ ;\ \overline{\Gamma'} \vdash^{0} \{?\ t\ ?\} : \{?\} \hookrightarrow \langle t' \rangle$$

Then by DEFERU0

$$(\Delta\ ;\ \overline{\Delta'}) +\!\!+^{1} \Delta'' \mid C\ ;\ D \mid \Gamma\ ;\ \overline{\Gamma'} \vdash^{1}_{b} t : \tau \hookrightarrow t' \qquad \text{(f)}$$

$$(\Delta\ ;\ \overline{\Delta'}) +\!\!+^{1} \Delta'' \vdash^{1} D \text{ constraint} \qquad \text{(g)}$$

By definition

$$\begin{aligned} & [\![\langle t' \rangle]\!]^0_{\eta +\!\!+ env(B)}\ \rho \\ &= \mathbf{unit_E}\ (\mathsf{code} : \lambda nms\ .\ [\![t']\!]^1_{\eta +\!\!+ env(B)}\ (nms, \rho)) \\ &= (*) \end{aligned}$$

Notice $(\Delta\, ; \overline{\Delta'}) +\!\!+^1 \Delta'' = \Delta\, ; (\overline{\Delta'} +\!\!+^0 \Delta'')$.
Let $\overline{\Delta_e}$ and $\overline{\Gamma_e}$ be s.t.

$$(\overline{\Delta_e}, \overline{\Gamma_e})\ \mathsf{extends}\ (\overline{\Delta'}, \theta\, \overline{\Gamma_r}) \tag{h}$$

and nms be s.t.

$$nms \in Names_{\backslash \overline{\Gamma_r} +\!\!+ \overline{\Gamma_e}} \tag{i}$$

W.l.o.g. assume $dom(\Delta'') \cap dom(\overline{\Delta'} +\!\!+ \overline{\Delta_e}) = \emptyset$. Then

$$(\overline{\Delta_e}, \overline{\Gamma_e})\ \mathsf{extends}\ (\overline{\Delta'} +\!\!+^0 \Delta'', \theta\, \overline{\Gamma_r})$$

and by (e) and Lemma D.7

$$\eta \models_{(\overline{\Delta'} +\!\!+^0 \Delta'', \theta \overline{\Gamma_r})} \theta\, \Gamma$$

Then by I.H. (ii) on (f)

$$[\![t']\!]^1_{\eta +\!\!+ env(B)}\ (nms, \rho) \in \mathbf{E}\ \mathcal{D}'_{wd}$$

where

$$\mathcal{D}'_{wd} = \left\{ d \in \mathcal{D} \;\middle|\; \begin{array}{l} termOf(d) \text{ well-defined}, \\ \forall i\ .\ vars(i, termOf(d)) \subseteq dom(\overline{\Gamma_r}^i) \end{array} \right\}$$

Since the choice of nms was arbitrary s.t. (i) holds, we have

$$(*) \in \mathbf{E} \left\{ \mathsf{code} : md \;\middle|\; \begin{array}{c} md \in \mathbf{M}\ \mathcal{D}, nms \in Names_{\backslash \overline{\Gamma_r} +\!\!+ \overline{\Gamma_e}} \\ \Longrightarrow md\ nms \in \mathbf{E}\ \mathcal{D}'_{wd} \end{array} \right\}$$

Furthermore, since the choice of $\overline{\Delta_e}$ and $\overline{\Gamma_e}$ was arbitrary s.t. (h) holds, we have

$$(*) \in \bigcap \{ S \mid (\overline{\Delta'_e}, \overline{\Gamma'_e})\ \mathsf{extends}\ (\overline{\Delta'}, \theta\, \overline{\Gamma_r}) \}$$

where

$$S = \mathbf{E} \left\{ \mathsf{code} : md \;\middle|\; \begin{array}{c} md \in \mathbf{M}\ \mathcal{D}, nms \in Names_{\backslash \overline{\Gamma_r} +\!\!+ \overline{\Gamma'_e}} \\ \Longrightarrow md\ nms \in \mathbf{E}\ \mathcal{D}_{wd} \end{array} \right\}$$

and

$$\mathcal{D}_{wd} = \mathbf{E} \left\{ d \in \mathcal{D} \;\middle|\; \begin{array}{l} termOf(d) \text{ well-defined}, \\ \forall i\ .\ vars(i, termOf(d)) \subseteq dom(\overline{\Gamma_r}^i) \end{array} \right\}$$

Hence

$$\begin{aligned} & (*) [\![\{?\}]\!]_{(\overline{\Delta'}, \theta \overline{\Gamma_r})} \\ &\quad = [\![\theta\ \{?\}]\!]_{(\overline{\Delta'}, \theta \overline{\Gamma_r})} \end{aligned}$$

as required.

case DEFERU1:

(ii) Let (a) be

$$\Delta\,;\overline{\Delta'} \mid C\,;\overline{C'}\,;C'' \mid \Gamma\,;\overline{\Gamma'} \vdash^{n+1}_{\mathbf{tt}} \{?\ t\ ?\} : \{?\} \hookrightarrow \{?\ t'\ ?\}$$

Then by DEFERU1

$$(\Delta\,;\overline{\Delta'}) +\!\!+^{n+2} \Delta'' \mid C\,;\overline{C'}\,;C''\,;D \mid \Gamma\,;\overline{\Gamma'} \vdash^{n+2}_{b} t : \tau \hookrightarrow t' \qquad \text{(h)}$$

$$(\Delta\,;\overline{\Delta'}) +\!\!+^{n+2} \Delta'' \vdash^{n+2} D \text{ contraint} \qquad \text{(i)}$$

By definition

$$\begin{aligned}
&[\![\{?\ t'\ ?\}]\!]^{n+1}_{\eta +\!\!+ env(B)}\ (nms, \rho) \\
&= (\mathbf{let_N}\ d \leftarrow [\![t']\!]^{n+2}_{\eta +\!\!+ env(B)}\ \mathbf{in}\ \mathbf{unit_N}\ (\mathsf{ddefu} : d))\ (nms, \rho) \\
&= \mathbf{let_E}\ d \leftarrow [\![t']\!]^{n+2}_{\eta +\!\!+ env(B)}\ (nms, \rho)\ \mathbf{in}\ \mathbf{unit_E}\ (\mathsf{ddefu} : d) \\
&= (*)
\end{aligned}$$

Notice $(\Delta\,;\overline{\Delta'}) +\!\!+^{n+2} \Delta'' = \Delta\,;(\overline{\Delta'} +\!\!+^{n+1} \Delta'')$.
W.l.o.g. assume $dom(\Delta'') \cap dom(\overline{\Delta'} +\!\!+ \overline{\Delta_e}) = \emptyset$. Then by (f)

$$(\overline{\Delta_e}, \overline{\Gamma_e}) \text{ extends } (\overline{\Delta'} +\!\!+^{n+1} \Delta'', \theta\,\overline{\Gamma_r})$$

and by (e) and Lemma D.7

$$\eta \models_{((\overline{\Delta'} +\!\!+^{n+1} \Delta'', \theta\,\overline{\Gamma_r})} \theta\,\Gamma$$

Then by I.H. (ii) on (h)

$$[\![t']\!]^{n+2}_{\eta +\!\!+ env(B)}\ (nms, \rho) \in \mathbf{E}\ \mathcal{D}'_{wd}$$

where

$$\mathcal{D}'_{wd} = \left\{ d \in \mathcal{D} \;\middle|\; \begin{array}{l} termOf(d) \text{ well-defined}, \\ \forall i\ .\ vars(i - (n+1), termOf(d)) \subseteq dom(\overline{\Gamma_r}^{\,i}) \end{array} \right\}$$

Since $termOf(\mathsf{ddefu} : d) = \{?\ termOf(d)\ ?\}$ is well-defined and $\forall i\ .\ vars(i - n, \{?\ termOf(d)\ ?\}) = vars(i - (n+1), termOf(d))$, we have $(\mathsf{ddefu} : d) \in \mathcal{D}_{wd}$ and so $(*) \in \mathbf{E}\ \mathcal{D}_{wd}$ as required.

(iii) Furthermore, if $b = \mathbf{tt}$ then by I.H. (iii) on (h)

$$[\![t']\!]^{n+2}_{\eta +\!\!+ env(B)}\ (nms, \rho) \in \mathbf{E}\ \mathcal{D}'_{wt}$$

where

$$\mathcal{D}'_{wt} = \left\{ d' \in \mathcal{D} \;\middle|\; \begin{array}{l} termOf(d') \text{ well-defined}, \\ (\overline{\Delta'} +\!\!+ \overline{\Delta_e}) +\!\!+^{n+1} \Delta'' \mid \theta\,(\overline{C'}\,;C''\,;D) \mid (\theta\,\overline{\Gamma_r}) +\!\!+ \overline{\Gamma_e} \vdash^{n+1}_{\mathbf{tt}} \\ \quad termOf(d') : \theta\,\tau \end{array} \right\}$$

Hence $d \in \mathcal{D}'_{wt}$.
By (b) and (i)

$$\overline{\Delta'} +\!\!+^{n+1} \Delta'' \vdash^{n+1} \theta\, D \text{ contraint}$$

Then, if $n > 0$, by DEFERU1

$$\overline{\Delta'} +\!\!+ \overline{\Delta_e} \mid \theta\,(\overline{C'}\,;\,C'') \mid (\theta\,\overline{\Gamma_r}) +\!\!+ \overline{\Gamma_e} \vdash^{n}_{\mathtt{tt}} \{?\ termOf(d)\ ?\} : \{?\}$$

Or, if $n = 0$, then $\overline{C'} = \cdot$ and by DEFERT0

$$\overline{\Delta'} +\!\!+ \overline{\Delta_e} \mid \theta\, C'' \mid (\theta\,\overline{\Gamma_r}) +\!\!+ \overline{\Gamma_e} \vdash^{0} \{?\ termOf(d)\ ?\} : \{?\}$$

Since $\{?\ termOf(d)\ ?\} = termOf(\mathsf{ddefu} : d)$ and $\{?\} = \theta\ \{?\}$, $(*) \in \mathbf{E}\ \mathcal{D}_{wt}$ as required.

case RUNT0:

(i) Let (a) be

$$\Delta\,;\overline{\Delta'} \mid C \mid \Gamma\,;\overline{\Gamma'} \vdash^{0} \mathbf{run}\ t : \mathtt{IO}\ \tau \hookrightarrow \mathsf{run}\ T\ \mathsf{at}\ W$$

By RUNT0

$$\Delta\,;\overline{\Delta'} \mid C \mid \Gamma\,;\overline{\Gamma'} \vdash^{0} t : \{\!\{\ \tau\ \}\!\} \hookrightarrow T \tag{f}$$

$$(\Delta\,;\overline{\Delta'})^{0} \vdash^{0} \tau : \mathsf{Type} \tag{g}$$

$$C \vdash^{e} \mathtt{liftable}\ \tau \hookrightarrow W \tag{h}$$

Let $\overline{\Delta_e}$, $\overline{\Gamma_e}$ and *nms* be s.t.

$$(\overline{\Delta_e}, \overline{\Gamma_e})\ \mathsf{extends}\ (\overline{\Delta}, \theta\,\overline{\Gamma_r}) \wedge nms \in Names_{\backslash \overline{\Gamma_r} +\!\!+ \overline{\Gamma_e}} \tag{i}$$

By definition

$$
\begin{array}{l}
[\![\mathsf{run}\ T\ \mathsf{at}\ W]\!]^{0}_{\eta +\!\!+ env(B)}\ \rho \\
= (\ \mathbf{let_R}\ ev \leftarrow \mathbf{closure^E_R}\ [\![T]\!]^{0}_{\eta +\!\!+ env(B)} \\
\quad \mathbf{in\ unit_R}\ (\mathsf{cmd} :\ \mathbf{let_{MIO}}\ v \leftarrow \mathbf{lift^{MIO}_E}\ ev \\
\qquad\qquad\qquad \mathbf{in\ case}\ v\ \mathbf{of}\ \{ \\
\qquad\qquad\qquad\quad \mathsf{code} : md \rightarrow \\
\qquad\qquad\qquad\qquad \mathbf{let_{MIO}}\ d \leftarrow \mathbf{lift^{MIO}_M}\ md \\
\qquad\qquad\qquad\qquad \mathbf{in\ if}\ termOf(d)\ \text{well-defined} \\
\qquad\qquad\qquad\qquad\quad \mathbf{and}\ (\overline{\Delta_{init}} \mid \mathbf{true} \mid \overline{\Gamma_{init}} \vdash^{0} \\
\qquad\qquad\qquad\qquad\qquad\quad termOf(d) : typeOf([\![W]\!]_{\eta +\!\!+ env(B)}) \\
\qquad\qquad\qquad\qquad\qquad\quad \hookrightarrow T')\ \mathbf{then} \\
\qquad\qquad\qquad\qquad\qquad \mathbf{unit_{MIO}}\ (\mathbf{run_R}\ [\![T']\!]^{0}) \\
\qquad\qquad\qquad\qquad\quad \mathbf{else} \\
\qquad\qquad\qquad\qquad\qquad \mathbf{throw_{MIO}}; \\
\qquad\qquad\qquad\quad \mathbf{otherwise} \rightarrow \\
\qquad\qquad\qquad\qquad \mathbf{unit_{MIO}}\ (\mathbf{unit_E}\ (\mathsf{wrong} : *)) \\
\qquad\qquad\qquad \}))\ \rho
\end{array}
$$

$$
\begin{aligned}
= \mathbf{unit_E}\ (\mathsf{cmd} : \lambda nms\ .\ & \mathbf{let_{IO}}\ v \leftarrow \mathbf{lift_E^{IO}}\ ([\![T]\!]^0_{\eta +\!\!+ env(B)}\ \rho) \\
& \mathbf{in\ case}\ v\ \mathbf{of}\ \{ \\
& \quad \mathsf{code} : md \rightarrow \\
& \quad\quad \mathbf{let_{IO}}\ d \leftarrow \mathbf{lift_E^{IO}}\ (md\ nms) \\
& \quad\quad \mathbf{in\ if}\ termOf(d)\ \text{well-defined} \\
& \quad\quad\quad \mathbf{and}\ (\overline{\Delta_{init}} \mid \mathbf{true} \mid \overline{\Gamma_{init}} \vdash^0 \\
& \quad\quad\quad\quad\quad termOf(d) : typeOf([\![W]\!]_{\eta +\!\!+ env(B)}) \\
& \quad\quad\quad\quad\quad \hookrightarrow T')\ \mathbf{then} \\
& \quad\quad\quad\quad \mathbf{unit_{IO}}\ ([\![T']\!]^0_{\cdot}\ \emptyset) \\
& \quad\quad\quad \mathbf{else} \\
& \quad\quad\quad\quad \mathbf{throw_{IO}}; \\
& \quad \mathbf{otherwise} \rightarrow \\
& \quad\quad \mathbf{unit_{IO}}\ (\mathbf{unit_E}\ (\mathsf{wrong} : *)) \\
& \})
\end{aligned}
$$

$$= \mathbf{unit_E}\ (\mathsf{cmd} : \lambda nms\ .\ (**))$$

$$= (*)$$

By I.H. (i) on (f)

$$
\begin{aligned}
[\![T]\!]^0_{\eta +\!\!+ env(B)}\ \rho \in\ & [\![\theta\ \{\!\{\ \tau\ \}\!\}]\!]_{(\overline{\Delta'}, \theta\, \overline{\Gamma_r})} \\
=\ & [\![\{\!\{\ \theta\ \tau\ \}\!\}]\!]_{(\overline{\Delta'}, \theta\, \overline{\Gamma_r})} \\
=\ & \bigcap \left\{ S \mid (\overline{\Delta'_e}, \overline{\Gamma'_e})\ \text{extends}\ (\overline{\Delta'}, \theta\ \overline{\Gamma_r}) \right\}
\end{aligned}
$$

where

$$S = \mathbf{E} \left\{ \mathsf{code} : md' \;\middle|\; \begin{array}{c} md' \in \mathbf{M}\ \mathcal{D}, nms' \in Names_{\setminus \overline{\Gamma_r} +\!\!+ \overline{\Gamma'_e}} \\ \Longrightarrow md'\ nms' \in \mathbf{E}\ \mathcal{D}_{wt} \end{array} \right\}$$

and

$$\mathcal{D}_{wt} = \left\{ d' \in \mathcal{D} \;\middle|\; \begin{array}{l} termOf(d')\ \text{well-defined}, \\ \overline{\Delta'} +\!\!+ \overline{\Delta'_e} \mid \mathbf{true} \mid (\theta\ \overline{\Gamma_r}) +\!\!+ \overline{\Gamma'_e} \vdash^0 termOf(d') : \theta\ \tau \end{array} \right\}$$

By (b) and (g) $\overline{\Delta_{init}} \vdash^0 \theta\ \tau : \mathtt{Type}$. Hence from (c), (h) and Lemma D.2

$$\mathtt{true} \vdash^e \mathtt{liftable}\ \theta\ \tau \hookrightarrow W' \wedge [\![W']\!]_{\cdot} = [\![W]\!]_{env(B)} = [\![W]\!]_{\eta +\!\!+ env(B)}$$

and by Lemma 9.4

$$typeOf([\![W]\!]_{\eta +\!\!+ env(B)}) = \theta\ \tau$$

Thus v must be tagged by code, and $md\ nms \in \mathbf{E}\ \mathcal{D}'_{wt}$ where

$$\mathcal{D}'_{wt} = \mathbf{E} \left\{ d' \in \mathcal{D} \;\middle|\; \begin{array}{l} termOf(d')\ \text{well-defined}, \\ \overline{\Delta'} +\!\!+ \overline{\Delta_e} \mid \mathbf{true} \mid (\theta\ \overline{\Gamma_r}) +\!\!+ \overline{\Gamma_e} \vdash^0 termOf(d') : \theta\ \tau \end{array} \right\}$$

Thus $d \in \mathcal{D}'_{wt}$, so $termOf(d)$ is well-defined, and

$$\begin{aligned}(**) = \;& \mathbf{let_{IO}}\ (\mathsf{code} : md) \leftarrow \mathbf{lift_E^{IO}}\ ([\![T]\!]^0_{\eta +\!\!+ env(B)}\ \rho) \\ & \mathbf{in\ let_{IO}}\ d \leftarrow \mathbf{lift_E^{IO}}\ (md\ nms) \\ & \mathbf{in\ if}\ (\overline{\Delta_{init}} \mid \mathsf{true} \mid \overline{\Gamma_{init}} \vdash^0 termOf(d) : \theta\ \tau \hookrightarrow T')\ \mathbf{then} \\ & \quad\quad \mathbf{unit_{IO}}\ ([\![T']\!]^0_{\cdot}\ \emptyset) \\ & \quad \mathbf{else} \\ & \quad\quad \mathbf{throw_{IO}}\end{aligned}$$

By I.H. (i) on the embedded judgement

$$\overline{\Delta_{init}} \mid \mathsf{true} \mid \overline{\Gamma_{init}} \vdash^0 termOf(d) : \theta\ \tau \hookrightarrow T'$$

(with identity substitution, empty value environment, empty renaming environment, and empty renamed context) we have

$$[\![T']\!]^0_{\cdot}\ \emptyset \in [\![\theta\ \tau]\!]_{(\overline{\Delta_{init}}, \overline{\Gamma_{init}})}$$

Thus $(**) \Downarrow_{\mathrm{IO}} ea$ implies

$$ea \in [\![\theta\ \tau]\!]_{(\overline{\Delta_{init}}, \overline{\Gamma_{init}})}$$

which by Lemma D.7 implies

$$ea \in [\![\theta\ \tau]\!]_{(\overline{\Delta'}, \theta \overline{\Gamma_r})}$$

Since the choice of nms, $\overline{\Delta_e}$ and $\overline{\Gamma_e}$ were arbitrary s.t. (i) holds, we have

$$\mathbf{unit_E}\ (\mathsf{cmd} : \lambda nms\ .\ (**)) \in [\![\mathtt{IO}\ (\theta\ \tau)]\!]_{(\overline{\Delta'}, \theta \overline{\Gamma_r})}$$

which in turn implies

$$(*) \in [\![\mathtt{IO}\ (\theta\ \tau)]\!]_{(\overline{\Delta'}, \theta \overline{\Gamma_r})} = [\![\theta\ (\mathtt{IO}\ \tau)]\!]_{(\overline{\Delta'}, \theta \overline{\Gamma_r})}$$

as required.

We may strengthen this result, though we only sketch the proof. Only the overall program environment may perform a command of type $\mathtt{IO}\ \tau$. Thus, the IO command $(**)$ will be performed only if **run** t is performed by the program environment. However, by the typing rules SPLICET1 and SPLICEU1, it is impossible for IO code to be performed underneath a splice. Thus, **run** t is well-typed with an empty $\overline{\Delta'}$ and $\overline{\Gamma'}$. Furthermore, assuming the initial environment $\eta_0 \models_{(\overline{\Delta_{init}}, \overline{\Gamma_{init}})} \Gamma_{init}$, then $\overline{\Gamma_r}$ may also be empty.

In this case, we see that $\overline{\Delta'} = \overline{\Delta_{init}}$, $\overline{\Gamma'} = \overline{\Gamma_r} = \overline{\Gamma_{init}}$, and $\rho = \emptyset$. Hence the inner typing judement succeeds, and command $(**)$ does not raise an exception.

case RUNT1:

(ii) Let (a) be

$$\Delta ; \overline{\Delta'} \mid C ; \overline{C'} ; C'' \mid \Gamma ; \overline{\Gamma'} \vdash^{n+1}_{b} \mathbf{run}\ t : \mathtt{IO}\ \tau \hookrightarrow \mathbf{run}\ t'$$

Then by RUNT1

$$\Delta\,;\overline{\Delta'}\mid C\,;\overline{C'}\,;\,C''\mid\Gamma\,;\overline{\Gamma'}\vdash^{n+1}_{b} t:\{\!\{\,\tau\,\}\!\}\hookrightarrow t' \qquad \text{(h)}$$

$$(\Delta\,;\overline{\Delta'})^{\leq n+1}\vdash^{n+1}\tau:\mathsf{Type} \qquad \text{(i)}$$

$$C''\vdash^{e}\mathtt{rttype}\;\tau \qquad \text{(j)}$$

By definition

$$\begin{aligned}&[\![\mathbf{run}\;t']\!]^{n+1}_{\eta+\!\!+env(B)}\;(nms,\rho)\\ &=(\mathbf{let_N}\;d\leftarrow[\![t]\!]^{n+1}_{\eta+\!\!+env(B)}\;\mathbf{in}\;\mathbf{unit_N}\;(\mathsf{drun}:d))\;(nms,\rho)\\ &=\mathbf{let_E}\;d\leftarrow[\![t]\!]^{n+1}_{\eta+\!\!+env(B)}\;(nms,\rho)\;\mathbf{in}\;\mathbf{unit_E}\;(\mathsf{drun}:d)\\ &=(*)\end{aligned}$$

By I.H. (ii) on (h)

$$[\![t']\!]^{n+1}_{\eta+\!\!+env(B)}\;(nms,\rho)\in\mathbf{E}\;\mathcal{D}_{wd}$$

and hence $d\in\mathcal{D}_{wd}$. Since $termOf(\mathsf{drun}:d)=\mathbf{run}\;termOf(d)$ and $vars(i-n,\mathbf{run}\;termOf(d))=vars(i-n,termOf(d))$, $(\mathsf{drun}:d)\in\mathcal{D}_{wd}$ and $(*)\in\mathbf{E}\;\mathcal{D}_{wd}$ as required.

(iii) Furthermore, if $b=\mathbf{tt}$ then by I.H. on (h)

$$[\![t']\!]^{n+1}_{\eta+\!\!+env(B)}\;(nms,\rho)\in\mathbf{E}\;\mathcal{D}'_{wt}$$

where if $n>0$ then

$$\mathcal{D}'_{wt}=\left\{d'\in\mathcal{D}\;\middle|\;\begin{array}{l}termOf(d')\text{ well-defined},\\ \overline{\Delta'}+\!\!+\overline{\Delta_e}\mid\theta\,(\overline{C'}\,;\,C'')\mid(\theta\,\overline{\Gamma_r})+\!\!+\overline{\Gamma_e}\vdash^{n}_{\mathbf{tt}}termOf(d'):\{\!\{\,\theta\,\tau\,\}\!\}\end{array}\right\}$$

otherwise

$$\mathcal{D}'_{wt}=\left\{d'\in\mathcal{D}\;\middle|\;\begin{array}{l}termOf(d')\text{ well-defined},\\ \overline{\Delta'}+\!\!+\overline{\Delta_e}\mid\theta\,C''\mid(\theta\,\overline{\Gamma_r})+\!\!+\overline{\Gamma_e}\vdash^{0}termOf(d'):\{\!\{\,\theta\,\tau\,\}\!\}\end{array}\right\}$$

Thus $d\in\mathcal{D}'_{wt}$.

By (b) and (i)

$$\overline{\Delta'}^{\leq n}\vdash^{n}\theta\,\tau:\mathsf{Type}$$

and thus

$$(\overline{\Delta'}+\!\!+\overline{\Delta_e})^{\leq n}\vdash^{n}\theta\,\tau:\mathsf{Type}$$

By (j) and Lemma D.3

$$\theta\;C''\vdash^{e}\mathtt{rttype}\;(\theta\,\tau)$$

Then if $n>0$, by RUNT1

$$\overline{\Delta'}+\!\!+\overline{\Delta_e}\mid\theta\,(\overline{C'}\,;\,C'')\mid(\theta\,\overline{\Gamma_r})+\!\!+\overline{\Gamma_e}\vdash^{n}_{\mathbf{tt}}\mathbf{run}\;termOf(d):\mathtt{IO}\;(\theta\,\tau)$$

Or, if $n=0$, by RUNT0

$$\overline{\Delta'}+\!\!+\overline{\Delta_e}\mid\theta\,(\overline{C'}\,;\,C'')\mid(\theta\,\overline{\Gamma_r})+\!\!+\overline{\Gamma_e}\vdash^{0}\mathbf{run}\;termOf(d):\mathtt{IO}\;(\theta\,\tau)$$

Since $termOf(\mathsf{drun} : d) = \mathtt{run}\ termOf(d)$ and $\mathtt{IO}\ (\theta\ \tau) = \theta\ (\mathtt{IO}\ \tau)$, then $(\mathsf{drun} : d) \in \mathcal{D}_{wt}$ and $(*) \in \mathbf{E}\ \mathcal{D}_{wt}$ as required.

case RUNU0:

(i) As for case RUNT0, but using $\mathcal{D}_{wd}$ instead of $\mathcal{D}_{wt}$. Hence there is no guarantee that the inner typing judement will succeed, and thus in this case run may throw an exception.

case RUNU1:

(ii)/(iii) As for case RUNT1.

case SPLICET1:

(ii) Let (a) be

$$\Delta\,;\overline{\Delta'} \mid C\,;C' \mid \Gamma\,;\overline{\Gamma'} \vdash^1_{\mathbf{tt}} \sim t : \tau \hookrightarrow \sim T$$

Then by SPLICET1

$$\Delta\,;\overline{\Delta'} \mid C \mid \Gamma\,;\overline{\Gamma'} \vdash^0 t : \{\!\{\ \tau\ \}\!\} \hookrightarrow T \tag{h}$$

By definition

$$\begin{aligned}
&[\![\sim T]\!]^1_{\eta +\!\!+ env(B)}\ (nms, \rho) \\
&= (\ \mathbf{let_N}\ v \leftarrow \mathbf{lift^N_R}\ [\![T]\!]^0_{\eta +\!\!+ env(B)} \\
&\quad\ \mathbf{in\ case}\ v\ \mathbf{of}\ \{ \\
&\quad\quad \mathsf{code} : md \to \mathbf{lift^N_M}\ md; \\
&\quad\quad \mathbf{otherwise} \to \mathbf{unit_N}\ (\mathsf{dwrong} : *) \\
&\quad\ \})\ (nms, \rho) \\
&= \ \mathbf{let_E}\ v \leftarrow [\![T]\!]^0_{\eta +\!\!+ env(B)}\ \rho \\
&\quad\ \mathbf{in\ case}\ v\ \mathbf{of}\ \{ \\
&\quad\quad \mathsf{code} : md \to md\ nms; \\
&\quad\quad \mathbf{otherwise} \to \mathbf{unit_E}\ (\mathsf{dwrong} : *) \\
&\quad\ \} \\
&= (*)
\end{aligned}$$

By I.H. (i) on (h)

$$\begin{aligned}
[\![T]\!]^0_{\eta +\!\!+ env(B)}\ \rho &\in [\![\theta\ \{\!\{\ \tau\ \}\!\}]\!]_{(\overline{\Delta'}, \theta \overline{\Gamma_r})} \\
&= [\![\{\!\{\ \theta\ \tau\ \}\!\}]\!]_{(\overline{\Delta'}, \theta \overline{\Gamma_r})} \\
&= \bigcap \{ S \mid (\overline{\Delta'_e}, \overline{\Gamma'_e})\ \mathsf{extends}\ (\overline{\Delta'}, \theta\ \overline{\Gamma_r}) \}
\end{aligned}$$

where

$$S = \mathbf{E}\ \left\{ \mathsf{code} : md \;\middle|\; \begin{array}{c} md \in \mathbf{M}\ \mathcal{D}, nms \in Names_{\backslash \overline{\Gamma_r} +\!\!+ \overline{\Gamma'_e}} \\ \Longrightarrow md\ nms \in \mathbf{E}\ \mathcal{D}'_{wt} \end{array} \right\}$$

and

$$\mathcal{D}'_{wt} = \left\{ d \in \mathcal{D} \;\middle|\; \begin{array}{l} termOf(d)\ \text{well-defined}, \\ \overline{\Delta'} +\!\!+ \overline{\Delta'_e} \mid \mathtt{true} \mid (\theta\ \overline{\Gamma_r}) +\!\!+ \overline{\Gamma'_e} \vdash^0 termOf(d) : \theta\ \tau \end{array} \right\}$$

Thus v is tagged by code and

$$(*) = \mathbf{let_E}\ (\mathsf{code} : md) \leftarrow [\![T]\!]^0_{\eta +\!\!+ env(B)}\ \rho\ \mathbf{in}\ md\ nms$$

Now, take $\overline{\Delta'_e} = \overline{\Delta_{init}}$ and $\overline{\Gamma'_e} = \overline{\Gamma_{init}}$. Then $md\ nms \in \mathbf{E}\ \mathcal{D}''_{wt}$ for

$$\mathcal{D}'_{wt} = \left\{ d \in \mathcal{D} \;\middle|\; \begin{array}{l} termOf(d) \text{ well-defined}, \\ \overline{\Delta'} \mid \mathtt{true} \mid (\theta\ \overline{\Gamma_r}) \vdash^0 termOf(d) : \theta\ \tau \end{array} \right\}$$

Then by Lemma D.4 $(*) \in \mathbf{E}\ \mathcal{D}_{wd}$ as required.

(iii) This time, take $\overline{\Delta'_e} = \overline{\Delta_e}$ and $\overline{\Gamma'_e} = \overline{\Gamma_e}$. Then $(*) \in \mathbf{E}\ \mathcal{D}_{wt}$ as required.

case SPLICET2:

(ii) Let (a) be

$$\Delta ; \overline{\Delta'} \mid C ; \overline{C'} ; C'' ; D \mid \Gamma ; \overline{\Gamma'} \vdash_b^{n+2} \sim t : \tau \hookrightarrow \sim t'$$

Then by SPLICET2

$$\Delta ; \overline{\Delta'} \mid C ; \overline{C'} ; C'' \mid \Gamma ; \overline{\Gamma'} \vdash_b^{n+1} t : \{\{\ \tau\ \}\} \hookrightarrow t' \tag{h}$$

By definition

$$\begin{aligned} &[\![\sim t']\!]^{n+2}_{\eta +\!\!+ env(B)}\ (nms, \rho) \\ &= (\mathbf{let_N}\ d \leftarrow [\![t']\!]^{n+1}_{\eta +\!\!+ env(B)}\ \mathbf{in}\ \mathbf{unit_N}\ (\mathsf{dsplice} : d))\ (nms, \rho) \\ &= \mathbf{let_E}\ d \leftarrow [\![t']\!]^{n+1}_{\eta +\!\!+ env(B)}\ (nms, \rho)\ \mathbf{in}\ \mathbf{unit_E}\ (\mathsf{dsplice} : d) \\ &= (*) \end{aligned}$$

By I.H. (ii) on (h)

$$[\![t']\!]^{n+1}_{\eta +\!\!+ env(B)}\ (nms, \rho) \in \mathbf{E}\ \mathcal{D}'_{wd}$$

where

$$\mathcal{D}'_{wd} = \left\{ d' \in \mathcal{D} \;\middle|\; \begin{array}{l} termOf(d') \text{ well-defined}, \\ \forall i\ .\ vars(i - n, termOf(d')) \subseteq dom(\overline{\Gamma_r}^i) \end{array} \right\}$$

Thus $d \in \mathcal{D}'_{wd}$. Since $termOf(\mathsf{dsplice} : d) = \sim termOf(d)$ is well-defined and $\forall i\ .\ vars(i - n, \sim termOf(d)) = vars(i - (n+1), \sim termOf(d))$, we have $(\mathsf{dsplice} : d) \in \mathcal{D}_{wd}$. Hence $(*) \in \mathbf{E}\ \mathcal{D}_{wd}$ as required.

(iii) Furthermore, if $b = \mathbf{tt}$ then by I.H. (iii) on (h)

$$[\![t']\!]^{n+1}_{\eta +\!\!+ env(B)}\ (nms, \rho) \in \mathbf{E}\ \mathcal{D}'_{wt}$$

where if $n > 0$ then

$$\mathcal{D}'_{wt} = \left\{ d' \in \mathcal{D} \;\middle|\; \begin{array}{l} termOf(d') \text{ well-defined}, \\ \overline{\Delta'} +\!\!+ \overline{\Delta_e} \mid \theta\ (\overline{C'} ; C'') \mid (\theta\ \overline{\Gamma_r}) +\!\!+ \overline{\Gamma_e} \vdash^n_{\mathbf{tt}} termOf(d') : \theta\ \{\{\ \tau\ \}\} \end{array} \right\}$$

otherwise

$$\mathcal{D}'_{wt} = \left\{ d' \in \mathcal{D} \;\middle|\; \begin{array}{l} termOf(d') \text{ well-defined}, \\ \overline{\Delta'} +\!\!+ \overline{\Delta_e} \mid \theta\ C'' \mid (\theta\ \overline{\Gamma_r}) +\!\!+ \overline{\Gamma_e} \vdash^0 termOf(d') : \theta\ \{\{\ \tau\ \}\} \end{array} \right\}$$

Notice $\theta\ \{\{\ \tau\ \}\} = \{\{\ \theta\ \tau\ \}\}$.

Then, if $n > 0$, by SPLICET2

$$\overline{\Delta'} +\!\!+ \overline{\Delta_e} \mid \theta\,(\overline{C'}\,;\,C''\,;\,D) \mid (\theta\,\overline{\Gamma_r}) +\!\!+ \overline{\Gamma_e} \vdash^{n+1}_{\mathbf{tt}} \sim termOf(d) : \theta\,\tau$$

Or, if $n = 0$, then $\overline{C'} = \cdot$ and by SPLICET1

$$\overline{\Delta'} +\!\!+ \overline{\Delta_e} \mid \theta\,(C\,;\,D) \mid (\theta\,\overline{\Gamma_r}) +\!\!+ \overline{\Gamma_e} \vdash^{1}_{\mathbf{tt}} \sim termOf(d) : \theta\,\tau$$

Since $termOf(\mathsf{dsplice} : d) = \sim termOf(d)$, then $(\mathsf{dsplice} : d) \in \mathcal{D}_{wt}$. Hence $(*) \in \mathbf{E}\,\mathcal{D}_{wt}$ as required.

case SPLICEU1:

(ii) Let (a) be

$$\Delta\,;\overline{\Delta'} \mid C\,;\,C' \mid \Gamma\,;\overline{\Gamma'} \vdash^{1}_{\mathbf{ff}} \sim t : \tau \hookrightarrow \sim T$$

Then by SPLICEU1

$$\Delta\,;\overline{\Delta'} \mid C \mid \Gamma\,;\overline{\Gamma'} \vdash^{0} t : \{?\} \hookrightarrow T \qquad \text{(h)}$$

$$\Delta\,;\overline{\Delta'} \vdash^{1} \tau : \mathsf{Type} \qquad \text{(i)}$$

By definition

$$\begin{aligned}
& [\![\sim T]\!]^{1}_{\eta +\!\!+ env(B)}\,(nms, \rho) \\
=\;& \mathbf{let_E}\ v \leftarrow [\![T]\!]^{0}_{\eta +\!\!+ env(B)}\,\rho \\
& \mathbf{in\ case}\ v\ \mathbf{of}\ \{ \\
& \quad \mathsf{code} : md \to md\ nms; \\
& \quad \mathbf{otherwise} \to \mathbf{unit_E}\ (\mathsf{dwrong} : *) \\
& \} \\
=\;& (*)
\end{aligned}$$

By I.H. (i) on (h)

$$\begin{aligned}
[\![T]\!]^{0}_{\eta +\!\!+ env(B)}\,\rho \in\;& [\![\theta\ \{?\}]\!]_{(\overline{\Delta'}, \theta\,\overline{\Gamma_r})} \\
=\;& [\![\{?\}]\!]_{(\overline{\Delta'}, \theta\,\overline{\Gamma_r})} \\
=\;& \bigcap \{ S \mid (\overline{\Delta'_e}, \overline{\Gamma'_e})\ \mathsf{extends}\ (\overline{\Delta'}, \theta\,\overline{\Gamma_r}) \}
\end{aligned}$$

where

$$S = \mathbf{E}\left\{ \mathsf{code} : md \;\middle|\; \begin{array}{l} md \in \mathbf{M}\,\mathcal{D}, nms \in Names_{\backslash \overline{\Gamma_r} +\!\!+ \overline{\Gamma'_e}} \\ \quad \Longrightarrow md\ nms \in \mathbf{E}\,\mathcal{D}'_{wd} \end{array} \right\}$$

and

$$\mathcal{D}'_{wd} = \left\{ d \in \mathcal{D} \;\middle|\; \begin{array}{l} termOf(d)\ \text{well-defined}, \\ \forall i\ .\ vars(i, termOf(d)) \subseteq dom(\overline{\Gamma_r}^{\,i}) \end{array} \right\}$$

Thus v is tagged by code and

$$(*) = \mathbf{let_E}\ (\mathsf{code} : md) \leftarrow [\![T]\!]^{0}_{\eta +\!\!+ env(B)}\,\rho\ \mathbf{in}\ md\ nms$$

where $md\ nms \in \mathbf{E}\,\mathcal{D}'_{wd}$.
Taking $\overline{\Delta'_e} = \overline{\Delta_e}$ and $\overline{\Gamma'_e} = \overline{\Gamma_e}$, we see $\mathcal{D}_{wd} = \mathcal{D}'_{wd}$. Thus $(*) \in \mathbf{E}\,\mathcal{D}_{wd}$ as required.

case SPLICEU2:

(ii) Let (a) be

$$\Delta ; \overline{\Delta'} \mid C ; \overline{C'} ; C'' ; D \mid \Gamma ; \overline{\Gamma'} \vdash^{n+2}_{\mathrm{ff}} \sim t : \tau \hookrightarrow \sim t'$$

Then by SPLICETU

$$\Delta ; \overline{\Delta'} \mid C ; \overline{C'} ; C'' \mid \Gamma ; \overline{\Gamma'} \vdash^{n+1}_{b} t : \{?\} \hookrightarrow t' \quad \text{(h)}$$

$$\Delta ; \overline{\Delta'} \vdash^{n+2} \tau : \mathsf{Type} \quad \text{(i)}$$

By definition

$$\begin{aligned} & [\![\sim t']\!]^{n+2}_{\eta + env(B)} (nms, \rho) \\ = {} & \mathbf{let_E}\ d \leftarrow [\![t']\!]^{n+1}_{\eta + env(B)} (nms, \rho)\ \mathbf{in}\ \mathbf{unit_E}\ (\mathsf{dsplice} : d) \\ = {} & (*) \end{aligned}$$

By I.H. (ii) on (h)

$$[\![t']\!]^{n+1}_{\eta + env(B)} (nms, \rho) \in \mathbf{E}\, \mathcal{D}'_{wd}$$

where

$$\mathcal{D}'_{wd} = \left\{ d' \in \mathcal{D} \,\middle|\, \begin{array}{l} termOf(d') \text{ well-defined}, \\ \forall i\ .\ vars(i-n, termOf(d')) \subseteq dom(\overline{\Gamma_r}^{i}) \end{array} \right\}$$

Thus $d \in \mathcal{D}'_{wd}$. Since $termOf(\mathsf{dsplice} : d) = \sim termOf(d)$ is well-defined and $\forall i$. $vars(i-n, \sim termOf(d)) = vars(i-(n+1), \sim termOf(d))$, we have $(\mathsf{dsplice} : d) \in \mathcal{D}_{wd}$. Hence $(*) \in \mathbf{E}\, \mathcal{D}_{wd}$ as required.

case LET0:

(i) Let (a) be

$$\Delta ; \overline{\Delta'} \mid C \mid \Gamma ; \overline{\Gamma'} \vdash^{0} \texttt{let}\ x = u\ \texttt{in}\ t : \tau \hookrightarrow \mathsf{let}\ x = (\mathsf{letw}\ B'\ \mathsf{in}\ \lambda names(D_2)\ .\ U)\ \mathsf{in}\ T$$

Then by LET0

$$(\Delta ; \overline{\Delta'}) +\!\!+^{0} \Delta'' \mid D_1 +\!\!+ D_2 \mid \Gamma ; \overline{\Gamma'} \vdash^{0} u : \upsilon \hookrightarrow U \quad \text{(f)}$$

$$\Delta ; \overline{\Delta'} \vdash^{0} D_1\ \mathsf{constraint} \quad \text{(g)}$$

$$(\Delta ; \overline{\Delta'}) +\!\!+^{0} \Delta'' \vdash^{0} D_2\ \mathsf{constraint} \quad \text{(h)}$$

$$inherit(D_1) \quad \text{(i)}$$

$$C \vdash^{e} D_1 \hookrightarrow B' \quad \text{(j)}$$

$$C \vdash^{e} \texttt{exists}\ \Delta''\ .\ D_2 \hookrightarrow \mathsf{True} \quad \text{(k)}$$

$$\Delta ; \overline{\Delta'} \mid C \mid (\Gamma ; \overline{\Gamma'}) +\!\!+^{0} x : \sigma \vdash^{0} t : \tau \hookrightarrow T \quad \text{(l)}$$

where $names(D_2) = (w_1, \ldots, w_m)$, $\sigma = \texttt{forall}\ \Delta''\ .\ anon(D_2)\ \texttt{=>}\ \upsilon$. and $\Delta'' = a_1 : \kappa_1, \ldots, a_o : \kappa_o$.

By definition

$$
\begin{aligned}
& [\![\mathsf{let}\ x = (\mathsf{letw}\ B'\ \mathsf{in}\ \lambda names(D)\ .\ U)\ \mathsf{in}\ T]\!]^0_{\eta +\!\!+ env(B)}\ \rho \\
= & (\ \mathbf{let_R}\ ev \leftarrow \mathbf{closure^E_R}\ [\![\lambda names(D)\ .\ U]\!]^0_{\eta +\!\!+ env(B', env(B))} \\
& \quad \mathbf{in}\ [\![T]\!]^0_{\eta +\!\!+ env(B), x \mapsto ev})\ \rho \\
= & (\ \mathbf{let_R}\ ev \leftarrow \mathbf{closure^E_R}\ (\\
& \qquad \mathbf{let_R}\ f \leftarrow \mathbf{closurefun^E_R}(\lambda(y_1, \ldots, y_m)\ .\ [\![U]\!]^0_{\eta +\!\!+ env(B', env(B)), w_1 \mapsto y_1, \ldots, w_n \mapsto y_m}) \\
& \qquad \mathbf{in\ unit_R}\ (\mathsf{tfunc}_m : f)) \\
& \quad \mathbf{in}\ [\![T]\!]^0_{\eta +\!\!+ env(B), x \mapsto ev})\ \rho \\
= & \ \mathbf{let_E}\ ev \leftarrow \mathbf{unit_E}\ (\mathbf{unit_E} \\
& \qquad (\mathsf{tfunc}_m : \lambda(y_1, \ldots, y_m)\ .\ [\![U]\!]^0_{\eta +\!\!+ env(B', env(B)), w_1 \mapsto y_1, \ldots, w_n \mapsto y_m}\ \rho)) \\
& \quad \mathbf{in}\ [\![T]\!]^0_{\eta +\!\!+ env(B), x \mapsto ev}\ \rho \\
= & [\![T]\!]^0_{\eta +\!\!+ env(B), x \mapsto ev}\ \rho \\
& \text{where } ev = \mathbf{unit_E}\ (\mathsf{tfunc} : \lambda(y_1, \ldots, y_m)\ .\ [\![U]\!]^0_{\eta +\!\!+ env(B', env(B)), w_1 \mapsto y_1, \ldots, w_n \mapsto y_m}\ \rho) \\
= & (*)
\end{aligned}
$$

Notice $(\Delta\ ; \overline{\Delta'}) +\!\!+^0 \Delta'' = (\Delta +\!\!+ \Delta'')\ ; \overline{\Delta'}$. Since $dom(\Delta'') \cap dom(\Delta) = \emptyset$, we have $dom(\Delta'') \cap dom(\theta) = \emptyset$.
Then by (b) and (h)

$$\Delta''\ ; \overline{\Delta'} \vdash^0 \theta\ D_2\ \mathsf{constraint} \tag{m}$$

By (b), (k) and Lemma D.2

$$\mathtt{true} \vdash \theta\ \mathtt{exists}\ \Delta''\ .\ D_2 \hookrightarrow \mathsf{True} \iff \mathtt{true} \vdash \mathtt{exists}\ \Delta''\ .\ (\theta\ D_2) \hookrightarrow \mathsf{True}$$

Then by (m) and Lemma 9.4 there exists types $\overline{v'}$ s.t. $\overline{\Delta_{init}} \vdash^0 \overline{v' : \kappa}$ and

$$\mathtt{true} \vdash (\theta\ D_2)[\overline{a \mapsto v'}] \hookrightarrow B_2 \tag{n}$$

From (j) and Lemma D.2

$$\mathtt{true} \vdash \theta\ D_1 \hookrightarrow B_1 \wedge env(B_1) = env(B', env(B))_{\restriction names(D_1)} \tag{o}$$

Let $\theta' = [a_1 \mapsto v'_1, \ldots, a_o \mapsto v'_o] \circ \theta$. Then $\Delta +\!\!+ \Delta'' \vdash \theta'$ gsubst and $\theta'\ \overline{\Gamma_r} = \theta\ \overline{\Gamma_r}$.
Then from (n), (o) and definition of entailment

$$\mathtt{true} \vdash^e \theta'\ (D_1 +\!\!+ D_2) \hookrightarrow B_1 +\!\!+ B_2$$

Then by I.H. (i) on (f)

$$[\![U]\!]^0_{\eta +\!\!+ env(B', env(B)) +\!\!+ env(B_2)}\ \rho \in [\![\theta'\ v]\!]_{(\overline{\Delta'}, \theta'\ \overline{\Gamma_r})} = [\![\theta'\ v]\!]_{(\overline{\Delta'}, \theta\ \overline{\Gamma_r})}$$

Since this holds for any choice of $\overline{v'}$ s.t. (n) holds, we have

$$ev \in \bigcap \left\{ S \;\middle|\; \begin{array}{l} \overline{\Delta_{init}} \vdash^0 \overline{v'' : \kappa}, \\ \mathtt{true} \vdash^e (\theta\ D_2)[\overline{a \mapsto v''}] \hookrightarrow B'' \end{array} \right\}$$

where

$$S = \mathbf{E}\left\{ \mathsf{tfunc}_m : f \;\middle|\; \begin{array}{c} f \in (\prod_{1\leq i\leq m} \mathcal{T}) \to \mathbf{E}\,\mathcal{V}, \\ f\ (\llbracket w_1 \rrbracket_{env(B'')}, \dots, \llbracket w_m \rrbracket_{env(B'')}) \\ \in \llbracket (\theta\ v)[a \mapsto v''] \rrbracket_{(\overline{\Delta'}, \theta\,\overline{\Gamma_r})} \end{array} \right\}$$

Hence

$$ev \in \llbracket \mathtt{forall}\ \Delta''\ .\ (\theta\ D_2)\ \mathtt{=>}\ (\theta\ v) \rrbracket_{(\overline{\Delta'}, \theta\,\overline{\Gamma_r})} = \llbracket \theta\ \sigma \rrbracket_{(\overline{\Delta'}, \theta\,\overline{\Gamma_r})}$$

Notice $(\Gamma\,;\overline{\Gamma'}) +\!\!+^0 x : \sigma = (\Gamma +\!\!+ x : \sigma)\,;\overline{\Gamma'}$. Let $\eta' = \eta, x \mapsto ev$. Then $\eta' \models_{(\overline{\Delta'}, \theta\,\overline{\Gamma_r})} \theta\,(\Gamma +\!\!+ x : \sigma)$.
So by I.H. (i) on (l)

$$\llbracket T \rrbracket^0_{\eta' +\!\!+ env(B)}\ \rho = (*) \in \llbracket \theta\ \tau \rrbracket_{(\overline{\Delta'}, \theta\,\overline{\Gamma_r})}$$

as required.

case LET1:

(ii) Let (a) be

$$\Delta\,;\overline{\Delta'} \mid C\,;\overline{C'}\,;C'' \mid \Gamma\,;\overline{\Gamma'} \vdash^{n+1}_b \mathtt{let}\ x\ \mathtt{=}\ u\ \mathtt{in}\ t : \tau \hookrightarrow \mathtt{let}\ x\ \mathtt{=}\ u'\ \mathtt{in}\ t'$$

Then by LET1

$$(\Delta\,;\overline{\Delta'}) +\!\!+^{n+1} \Delta'' \mid C\,;\overline{C'}\,;D_1 +\!\!+ D_2 \mid \Gamma\,;\overline{\Gamma'} \vdash^{n+1}_b u : v \hookrightarrow u' \quad \text{(h)}$$

$$\Delta\,;\overline{\Delta'} \vdash^{n+1} D_1\ \mathsf{constraint} \quad \text{(i)}$$

$$(\Delta\,;\overline{\Delta'}) +\!\!+^{n+1} \Delta'' \vdash^{n+1} D_2\ \mathsf{constraint} \quad \text{(j)}$$

$$inherit(D_1) \quad \text{(k)}$$

$$C'' \vdash^e D_1 \quad \text{(l)}$$

$$C'' \vdash^e \mathtt{exists}\ \Delta''\ .\ D_2 \quad \text{(m)}$$

$$\Delta\,;\overline{\Delta'} \mid C\,;\overline{C'}\,;C'' \mid (\Gamma\,;\overline{\Gamma'}) +\!\!+^{n+1} x : \sigma \vdash^{n+1}_b t : \tau \hookrightarrow t' \quad \text{(n)}$$

where $\sigma = (\mathtt{forall}\ \Delta''\ .\ anon(D_2)\ \mathtt{=>}\ v)$.
Notice $(\Delta;\overline{\Delta'}) +\!\!+^{n+1} \Delta'' = \Delta;(\overline{\Delta'} +\!\!+^n \Delta'')$, and $(\Gamma;\overline{\Gamma'}) +\!\!+^{n+1} x : \sigma = \Gamma;(\overline{\Gamma'} +\!\!+^n x : \sigma)$.
W.l.o.g. assume $dom(\Delta'') \cap dom(\overline{\Delta'} +\!\!+ \overline{\Delta_e}) = \emptyset$. Then

$$(\overline{\Delta_e}, \overline{\Gamma_e})\ \mathsf{extends}\ (\overline{\Delta'} +\!\!+^n \Delta'', \theta\,\overline{\Gamma'_r})$$

From (e) and Lemma D.7

$$\eta \models_{(\overline{\Delta'} +\!\!+^n \Delta'', \theta\,\overline{\Gamma_r})} \theta\,\Gamma$$

Hence by I.H. (ii) on (h)

$$\llbracket u' \rrbracket^{n+1}_{\eta +\!\!+ env(B)}\ (nms, \rho) \in \mathbf{E}\,\mathcal{D}'_{wd} \quad \text{(o)}$$

where

$$\mathcal{D}'_{wd} = \left\{ d'' \in \mathcal{D} \;\middle|\; \begin{array}{l} termOf(d'')\ \text{well-defined}, \\ \forall i\ .\ vars(i - n, termOf(d'')) \subseteq dom(\overline{\Gamma_r}^i) \end{array} \right\}$$

W.l.o.g. assume nms = "y" : nms', where by (f) $y \notin dom(\Gamma_r +\!\!+ \Gamma_e)$. Let $\overline{\Gamma'_r} = \overline{\Gamma_r} +\!\!+^n y : \sigma$ and $\rho' = \rho[x \mapsto y]$. (This may override an existing binding for x in ρ.)

Since *nms* contains only distinct variable names

$$nms' \in Names_{\setminus \overline{\Gamma'_r} + \overline{\Gamma_e}}$$

and

$$(\overline{\Delta_e}, \overline{\Gamma_e}) \text{ extends } (\overline{\Delta'}, \theta\,\overline{\Gamma'_r})$$

By (d)

$$\rho'\,(\overline{\Gamma'} +\!\!+^n x : \sigma) \subseteq \Gamma'_r$$

By (e) and Lemma D.7

$$\eta \models_{(\overline{\Delta'}, \theta\,\overline{\Gamma'_r})} \theta\,\Gamma$$

Then by I.H. (ii) on (n)

$$[\![t']\!]^{n+1}_{\eta +\!\!+ env(B)}\,(nms', \rho') \in \mathbf{E}\,\mathcal{D}''_{wd} \tag{p}$$

where

$$\mathcal{D}''_{wd} = \left\{ d''' \in \mathcal{D} \;\middle|\; \begin{array}{l} termOf(d''') \text{ well-defined}, \\ \forall i\,.\,vars(i - n, termOf(d''')) \subseteq dom(\overline{\Gamma'_r}^{\,i}) \end{array} \right\}$$

By definition

$$\begin{aligned}
&[\![\texttt{let } x = u' \texttt{ in } t']\!]^{n+1}_{\eta +\!\!+ env(B)}\,(nms, \rho) \\
= (\;&\mathbf{let_N}\ d \leftarrow [\![u']\!]^{n+1}_{\eta +\!\!+ env(B)} \\
&\mathbf{in\ let_N}\ (nm, d') \leftarrow \mathbf{rename_N}\ \text{“}x\text{”}\ [\![t']\!]^{n+1}_{\eta +\!\!+ env(B)} \\
&\mathbf{in\ unit_N}\ (\mathsf{dlet} : (nm, d, d')))\ (nms, \rho) \\
= \;&\mathbf{let_E}\ d \leftarrow [\![u']\!]^{n+1}_{\eta +\!\!+ env(B)}\,(nms, \rho) \\
&\mathbf{in\ let_E}\ d' \leftarrow [\![t']\!]^{n+1}_{\eta +\!\!+ env(B)}\,(nms', \rho') \\
&\mathbf{in\ unit_E}\ (\mathsf{dlet} : (\text{“}y\text{”}, d, d')) \\
= \;&(*)
\end{aligned}$$

Then by (o) $d \in \mathcal{D}'_{wd}$ and by (p) $d' \in \mathcal{D}''_{wd}$. Hence

$$termOf(\mathsf{dlet} : (\text{“}y\text{”}, d, d')) = \texttt{let } y = termOf(d) \texttt{ in } termOf(d')$$

is well-defined, and

$$\begin{aligned}
&vars(0, \texttt{let } y = termOf(d) \texttt{ in } termOf(d')) \\
&= vars(0, termOf(d)) \cup (vars(0, termOf(d')) \setminus \{y\}) \\
&\subseteq dom(\overline{\Gamma_r}^{\,n})
\end{aligned}$$

Thus $(*) \in \mathbf{E}\,\mathcal{D}_{wd}$ as required.
(iii) Furthermore, if $b = \mathbf{tt}$ then by I.H. (iii) on (h)

$$[\![u']\!]^{n+1}_{\eta +\!\!+ env(B)}\,(nms, \rho) \in \mathbf{E}\,\mathcal{D}'_{wt}$$

where if $n > 0$ then

$$\mathcal{D}'_{wt} = \left\{ d'' \in \mathcal{D} \;\middle|\; \begin{array}{l} termOf(d'') \text{ well-defined}, \\ (\overline{\Delta'} +\!\!+ \overline{\Delta_e}) +\!\!+^n \Delta'' \mid \theta\,(\overline{C'}\,;\, D_1 +\!\!+ D_2) \mid (\theta\,\overline{\Gamma_r}) +\!\!+ \overline{\Gamma_e} \vdash^n_{\mathtt{tt}} \\ \quad termOf(d'') : \theta\, v \end{array} \right\}$$

otherwise

$$\mathcal{D}'_{wt} = \left\{ d'' \in \mathcal{D} \;\middle|\; \begin{array}{l} termOf(d'') \text{ well-defined}, \\ (\overline{\Delta'} +\!\!+ \overline{\Delta_e}) +\!\!+^n \Delta'' \mid \theta\,(D_1 +\!\!+ D_2) \mid (\theta\,\overline{\Gamma_r}) +\!\!+ \overline{\Gamma_e} \vdash^0 \\ \quad termOf(d'') : \theta\, v \end{array} \right\}$$

Also, by I.H. (iii) on (n)

$$[\![t']\!]^{n+1}_{\eta +\!\!+ env(B)}\,(nms', \rho') \in \mathbf{E}\,\mathcal{D}''_{wt}$$

where if $n > 0$

$$\mathcal{D}''_{wt} = \left\{ d''' \in \mathcal{D} \;\middle|\; \begin{array}{l} termOf(d''') \text{ well-defined}, \\ \overline{\Delta'} +\!\!+ \overline{\Delta_e} \mid \theta\,(\overline{C'}\,;\, C'') \mid (\theta\,(\overline{\Gamma_r} +\!\!+^n y : \sigma)) +\!\!+ \overline{\Gamma_e} \vdash^n_{\mathtt{tt}} \\ \quad termOf(d''') : \theta\, \tau \end{array} \right\}$$

otherwise

$$\mathcal{D}''_{wt} = \left\{ d''' \in \mathcal{D} \;\middle|\; \begin{array}{l} termOf(d''') \text{ well-defined}, \\ \overline{\Delta'} +\!\!+ \overline{\Delta_e} \mid \theta\, C'' \mid (\theta\,(\overline{\Gamma_r} +\!\!+^n y : \sigma)) +\!\!+ \overline{\Gamma_e} \vdash^0 \\ \quad termOf(d''') : \theta\, \tau \end{array} \right\}$$

So now $d \in \mathcal{D}'_{wt}$ and $d' \in \mathcal{D}''_{wt}$.
By (b), (i) and (j)

$$\overline{\Delta'} \vdash^n \theta\, D_1 \text{ constraint}$$

$$\overline{\Delta'} +\!\!+^n \Delta'' \vdash^n \theta\, D_2 \text{ constraint}$$

By definition of *inherit* and (k)

$$inherit(\theta\, D_1)$$

By (b), Lemma D.3 and since $dom(\theta) \cap dom(\Delta'') = \emptyset$

$$\theta\, C'' \vdash^e \theta\, D_1$$

$$\theta\, C'' \vdash^e \mathtt{exists}\ \Delta''\ .\ \theta\, D_2$$

Also

$$\theta\, \sigma = \mathtt{forall}\ \Delta''\ .\ (\theta\, anon(D_2)) \Rightarrow \theta\, v$$

Then if $n > 0$, by LET1

$$\overline{\Delta'} +\!\!+ \overline{\Delta_e} \mid \theta\,(\overline{C'}\,;\, C'') \mid (\theta\,\overline{\Gamma_r}) +\!\!+ \overline{\Gamma_e} \vdash^{n+1}_{\mathtt{tt}} \mathtt{let}\ y = termOf(d)\ \mathtt{in}\ termOf(d') : \theta\, \tau$$

Or, if $n = 0$ then $\overline{C'} = \cdot$ and by LET0

$$\overline{\Delta'} +\!\!+ \overline{\Delta_e} \mid \theta\, C'' \mid (\theta\,\overline{\Gamma_r}) +\!\!+ \overline{\Gamma_e} \vdash^0 \mathtt{let}\ y = termOf(d)\ \mathtt{in}\ termOf(d') : \theta\, \tau$$

Thus $(*) \in \mathbf{E}\,\mathcal{D}_{wt}$ as required.

case LIFT0:

(i) Let (a) be

$$\Delta\,;\overline{\Delta'} \mid C \mid \Gamma\,;\overline{\Gamma'} \vdash^0 \texttt{lift}\ t : \{\!\{\ \tau\ \}\!\} \hookrightarrow \mathsf{lift}\ T\ \mathsf{using}\ W$$

Then by LIFT0

$$\Delta\,;\overline{\Delta'} \mid C \mid \Gamma\,;\overline{\Gamma} \vdash^0 t : \tau \hookrightarrow T \tag{f}$$

$$(\Delta\,;\overline{\Delta'})^0 \vdash^0 \tau : \texttt{Type} \tag{g}$$

$$C \vdash^e \texttt{liftable}\ \tau \hookrightarrow W \tag{h}$$

By definition

$$\begin{aligned}
&[\![\mathsf{lift}\ T\ \mathsf{using}\ W]\!]^0_{\eta +\!\!+ env(B)}\ \rho \\
&= (\ \mathbf{let_R}\ v \leftarrow [\![T]\!]^0_{\eta +\!\!+ env(B)} \\
&\quad\ \mathbf{in\ case}\ (v, [\![W]\!]_{\eta +\!\!+ env(B)})\ \mathbf{of}\ \{ \\
&\qquad (\mathsf{int} : i, \mathsf{tint} : *) \rightarrow \mathbf{unit_R}\ (\mathsf{code} : \mathbf{unit_M}\ (\mathsf{dconst} : i)); \\
&\qquad \mathbf{otherwise} \rightarrow \mathbf{unit_R}\ (\mathsf{wrong} : *) \\
&\quad\ \})\ \rho \\
&=\ \mathbf{let_E}\ v \leftarrow [\![T]\!]^0_{\eta +\!\!+ env(B)}\ \rho \\
&\quad\ \mathbf{in\ case}\ (v, [\![W]\!]_{\eta +\!\!+ env(B)})\ \mathbf{of}\ \{ \\
&\qquad (\mathsf{int} : i, \mathsf{tint} : *) \rightarrow \mathbf{unit_E}\ (\mathsf{code} : \lambda nms\ .\ \mathbf{unit_E}\ (\mathsf{dconst} : i)); \\
&\qquad \mathbf{otherwise} \rightarrow \mathbf{unit_E}\ (\mathsf{wrong} : *) \\
&\quad\ \} \\
&= (*)
\end{aligned}$$

By I.H. (i) on (f)

$$[\![T]\!]_{\eta +\!\!+ env(B)}\ \rho \in [\![\theta\ \tau]\!]_{(\overline{\Delta'}, \theta\,\overline{\Gamma_r})} \tag{i}$$

By (b), (c), (g) and Lemma D.2

$$\mathbf{true} \vdash^e \texttt{rttype}\ (\theta\ \tau) \hookrightarrow W' \wedge [\![W']\!]_{.} = [\![W]\!]_{env(B)} = [\![W]\!]_{\eta +\!\!+ env(B)}$$

Then by Lemma 9.4

$$typeOf([\![W]\!]_{\eta +\!\!+ env(B)}) = \theta\ \tau \in \{\mathbf{Int}\}$$

We proceed by (trivial!) case analysis on $\theta\ \tau$:

case Int: By (i)

$$[\![T]\!]_{\eta +\!\!+ env(B', env(B))}\ \rho \in [\![\mathbf{Int}]\!]_{(\overline{\Delta'}, \theta\,\overline{\Gamma_r})} = \mathbf{E}\ \{\mathsf{int} : i \mid i \in \mathcal{Z}\}$$

Then v is tagged by int, $[\![w]\!]_{env(B', env(B))}$ is $\mathsf{tint} : *$ and

$$\begin{aligned}
(*) =\ & \mathbf{let_E}\ (\mathsf{int} : i) \leftarrow [\![T]\!]^0_{\eta +\!\!+ env(B)}\ \rho \\
& \mathbf{in\ unit_E}\ (\mathsf{code} : \lambda nms\ .\ \mathbf{unit_E}\ (\mathsf{dconst} : i))
\end{aligned}$$

where $i \in \mathcal{Z}$.
Let $\overline{\Delta_e}$, $\overline{\Gamma_e}$ and *nms* be s.t.

$$(\overline{\Delta_e}, \overline{\Gamma_e}) \text{ extends } (\overline{\Delta'}, \theta\,\overline{\Gamma_r}) \wedge nms \in Names_{\setminus\overline{\Gamma_r} +\!\!+ \overline{\Gamma_e}} \tag{j}$$

Then

$$\mathbf{unit_E}\ (\mathsf{dconst} : i) \in \mathbf{E}\ \mathcal{D}_{wt}$$

where

$$\mathcal{D}_{wt} = \left\{ d \in \mathcal{D} \;\middle|\; \begin{array}{l} termOf(d) \text{ well-defined}, \\ \overline{\Delta'} +\!\!+ \overline{\Delta_e} \mid \mathbf{true} \mid (\theta\,\overline{\Gamma_r}) +\!\!+ \overline{\Gamma_e} \vdash^0 termOf(d) : \mathtt{Int} \end{array} \right\}$$

Thus

$$(*) \in \mathbf{E} \left\{ \mathsf{code} : md \;\middle|\; \begin{array}{c} md \in \mathbf{M}\ \mathcal{D}, nms \in Names_{\setminus\overline{\Gamma_r} +\!\!+ \overline{\Gamma_e}} \\ \Longrightarrow md\ nms \in \mathbf{E}\ \mathcal{D}_{wt} \end{array} \right\}$$

Since this holds for any choice of $\overline{\Delta_e}$, $\overline{\Gamma_e}$ and *nms* s.t. (j) holds, we have

$$(*) \in [\![\{\{\ \mathtt{Int}\ \}\}]\!]_{(\overline{\Gamma'}, \theta\,\overline{\Gamma_r})} = [\![\theta\ \{\{\ \mathtt{Int}\ \}\}]\!]_{(\overline{\Gamma'}, \theta\,\overline{\Gamma_r})}$$

as required.

(If `liftable` were extended to other types, the cases would proceed analogously.)

case LETM0:

(i) Let (a) be

$$\Delta\,;\overline{\Delta'} \mid C \mid \Gamma\,;\overline{\Gamma'} \vdash^0 \mathtt{let}\ x\ \mathtt{<-}\ u\ \mathtt{in}\ t : \mathtt{IO}\ \tau \hookrightarrow \mathsf{let}\ x \leftarrow U\ \mathsf{in}\ T$$

Then by LETM0

$$\Delta\,;\overline{\Delta'} \mid C \mid \Gamma\,;\overline{\Gamma'} \vdash^0 u : \mathtt{IO}\ \upsilon \hookrightarrow U \tag{f}$$

$$\Delta\,;\overline{\Delta'} \mid C \mid (\Gamma\,;\overline{\Gamma'}) +\!\!+^0 x : \upsilon \vdash^0 t : \mathtt{IO}\ \tau \hookrightarrow T \tag{g}$$

By definition

$$\begin{array}{l}
[\![\mathsf{let}\ x \leftarrow U\ \mathsf{in}\ T]\!]^0_{\eta +\!\!+ env(B)} \\
= (\ \mathbf{let_R}\ ev \leftarrow \mathbf{closure^E_R}\ [\![U]\!]^0_{\eta +\!\!+ env(B)} \\
\quad \mathbf{in}\ \mathbf{let_R}\ f \leftarrow \mathbf{closurefun^E_R}\ (\lambda ev' \,.\, [\![T]\!]^0_{\eta +\!\!+ env(B), x \mapsto ev'}) \\
\quad \mathbf{in}\ \mathbf{unit_R}\ (\mathsf{cmd} :\ \mathbf{let_{MIO}}\ v \leftarrow \mathbf{lift^{MIO}_E}\ ev \\
\qquad\qquad\qquad \mathbf{in\ case}\ v\ \mathbf{of}\ \{ \\
\qquad\qquad\qquad\quad \mathsf{cmd} : ioev \rightarrow \\
\qquad\qquad\qquad\qquad \mathbf{let_{MIO}}\ ev' \leftarrow ioev \\
\qquad\qquad\qquad\qquad \mathbf{in}\ \mathbf{let_{MIO}}\ v' \leftarrow \mathbf{lift^{MIO}_E}\ (f\ ev') \\
\qquad\qquad\qquad\qquad \mathbf{in\ case}\ v'\ \mathbf{of}\ \{ \\
\qquad\qquad\qquad\qquad\quad \mathsf{cmd} : ioev' \rightarrow ioev'; \\
\qquad\qquad\qquad\qquad\quad \mathbf{otherwise} \rightarrow \mathbf{unit_{MIO}}\ (\mathbf{unit_E}\ (\mathsf{wrong} : *)) \\
\qquad\qquad\qquad\qquad \}; \\
\qquad\qquad\qquad\quad \mathbf{otherwise} \rightarrow \mathbf{unit_{MIO}}\ (\mathbf{unit_E}\ (\mathsf{wrong} : *)) \\
\qquad\qquad\qquad \}))\ \rho
\end{array}$$

$$\begin{array}{l}
= \ \mathbf{let_E}\ ev \leftarrow \mathbf{unit_E}\ ([\![U]\!]^0_{\eta + \!\!+ env(B)}\ \rho) \\
\quad \mathbf{in\ let_E}\ f \leftarrow \mathbf{unit_E}\ (\lambda ev' .\ [\![T]\!]^0_{\eta + \!\!+ env(B), x \mapsto ev'}\ \rho) \\
\quad \mathbf{in\ unit_E}\ (\mathsf{cmd} : \lambda nms .\ \ \mathbf{let_{IO}}\ v \leftarrow \mathbf{lift^{IO}_E}\ ev \\
\qquad\qquad\qquad\qquad \mathbf{in\ case}\ v\ \mathbf{of}\ \{ \\
\qquad\qquad\qquad\qquad\quad \mathsf{cmd} : ioev \rightarrow \\
\qquad\qquad\qquad\qquad\qquad \mathbf{let_{IO}}\ ev' \leftarrow ioev\ nms \\
\qquad\qquad\qquad\qquad\qquad \mathbf{in\ let_{IO}}\ v' \leftarrow \mathbf{lift^{IO}_E}\ (f\ ev') \\
\qquad\qquad\qquad\qquad\qquad \mathbf{in\ case}\ v'\ \mathbf{of}\ \{ \\
\qquad\qquad\qquad\qquad\qquad\quad \mathsf{cmd} : ioev' \rightarrow ioev'\ nms; \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{otherwise} \rightarrow \mathbf{unit_{IO}}\ (\mathbf{unit_E}\ (\mathsf{wrong} : *)) \\
\qquad\qquad\qquad\qquad\qquad \}; \\
\qquad\qquad\qquad\qquad\quad \mathbf{otherwise} \rightarrow \mathbf{unit_{IO}}\ (\mathbf{unit_E}\ (\mathsf{wrong} : *)) \\
\qquad\qquad\qquad\qquad \}) \\
= \mathbf{unit_E}\ (\mathsf{cmd} : \lambda nms .\ \ \mathbf{let_{IO}}\ v \leftarrow \mathbf{lift^{IO}_E}\ ([\![U]\!]^0_{\eta + \!\!+ env(B)}\ \rho) \\
\qquad\qquad\qquad \mathbf{in\ case}\ v\ \mathbf{of}\ \{ \\
\qquad\qquad\qquad\quad \mathsf{cmd} : ioev \rightarrow \\
\qquad\qquad\qquad\qquad \mathbf{let_{IO}}\ ev' \leftarrow ioev\ nms \\
\qquad\qquad\qquad\qquad \mathbf{in\ let_{IO}}\ v' \leftarrow \mathbf{lift^{IO}_E}\ ([\![T]\!]^0_{\eta + \!\!+ env(B), x \mapsto ev'}\ \rho) \\
\qquad\qquad\qquad\qquad \mathbf{in\ case}\ v'\ \mathbf{of}\ \{ \\
\qquad\qquad\qquad\qquad\quad \mathsf{cmd} : ioev' \rightarrow ioev'\ nms; \\
\qquad\qquad\qquad\qquad\quad \mathbf{otherwise} \rightarrow \mathbf{unit_{IO}}\ (\mathbf{unit_E}\ (\mathsf{wrong} : *)) \\
\qquad\qquad\qquad\qquad \}; \\
\qquad\qquad\qquad\quad \mathbf{otherwise} \rightarrow \mathbf{unit_{IO}}\ (\mathbf{unit_E}\ (\mathsf{wrong} : *)) \\
\qquad\qquad\qquad \}) \\
= \mathbf{unit_E}\ (\mathsf{cmd} : \lambda nms\ .\ (**)) \\
= (*)
\end{array}$$

Let $\overline{\Delta_e}$, $\overline{\Gamma_e}$ and nms be s.t.

$$(\overline{\Delta_e}, \overline{\Gamma_e})\ \mathsf{extends}\ (\overline{\Delta'}, \theta\ \overline{\Gamma_r}) \wedge nms \in Names_{\backslash \overline{\Gamma_r} + \!\!+ \overline{\Gamma_e}} \tag{h}$$

By I.H. (i) on (f)

$$\begin{aligned}
[\![U]\!]^0_{\eta + \!\!+ env(B)}\ \rho &\in [\![\theta\ \mathtt{IO}\ v]\!]_{(\overline{\Delta'}, \theta \overline{\Gamma_r})} \\
&\in [\![\mathtt{IO}\ (\theta\ v)]\!]_{(\overline{\Delta'}, \theta \overline{\Gamma_r})} \\
&= \bigcap \{ S \mid (\overline{\Delta'_e}, \overline{\Gamma'_e})\ \mathsf{extends}\ (\overline{\Delta}, \theta\ \overline{\Gamma_r}) \}
\end{aligned}$$

where

$$S = \mathbf{E} \left\{ \mathsf{cmd} : io \;\middle|\; \begin{array}{l} io \in \mathbf{MIO}\ (\mathbf{E}\ \mathcal{V}), \\ nms \in Names_{\backslash \overline{\Gamma_r} + \!\!+ \overline{\Gamma'_e}} \wedge (io\ nms) \Downarrow_{\mathbf{IO}} ea \\ \quad \Longrightarrow ea \in [\![\theta\ v]\!]_{(\overline{\Delta'}, \theta\ \overline{\Gamma_r})} \end{array} \right\}$$

Hence v is tagged by cmd and

$$ev' \in [\![\theta\ v]\!]_{(\overline{\Delta'}, \theta \overline{\Gamma_r})}$$

Notice $(\Gamma\,;\overline{\Gamma'}) +\!\!+^0 x : v = (\Gamma +\!\!+ x : v)\,;\overline{\Gamma'}$. Let $\eta' = \eta, x \mapsto ev'$. Then

$$\eta' \models_{(\overline{\Delta'},\theta\,\overline{\Gamma_r})} (\theta\,\Gamma) +\!\!+ x : \theta\, v \iff \eta' \models_{(\overline{\Delta'},\theta\,\overline{\Gamma_r})} \theta\,(\Gamma +\!\!+ x : v)$$

Then by I.H. (i) on (g)

$$\begin{aligned}[\![T]\!]^0_{\eta' +\!\!+ env(B)}\,\rho &= [\![T]\!]^0_{\eta +\!\!+ env(B), x\mapsto ev'}\,\rho \\ &\in [\![\theta \;\texttt{IO}\;\tau]\!]_{(\overline{\Delta'},\theta\,\overline{\Gamma_r})} \\ &\in [\![\texttt{IO}\;(\theta\;\tau)]\!]_{(\overline{\Delta'},\theta\,\overline{\Gamma_r})} \\ &= \bigcap \{ S' \mid (\overline{\Delta'_e}, \overline{\Gamma'_e}) \text{ extends } (\overline{\Delta}, \theta\,\overline{\Gamma_r}) \}\end{aligned}$$

where

$$S' = \mathbf{E}\left\{ \mathsf{cmd} : io \;\middle|\; \begin{array}{l} io \in \mathbf{MIO}\;(\mathbf{E}\;\mathcal{V}), \\ nms \in Names_{\backslash \overline{\Gamma_r} +\!\!+ \overline{\Gamma'_e}} \wedge (io\;nms) \Downarrow_{\mathbf{IO}} ea \\ \quad \Longrightarrow ea \in [\![\theta\;\tau]\!]_{(\overline{\Delta'},\theta\,\overline{\Gamma_r})} \end{array} \right\}$$

Hence v' is tagged by cmd. Thus

$$\begin{aligned}(**) = \;& \mathbf{let_{IO}}\;(\mathsf{cmd} : ioev) \leftarrow \mathbf{lift^{IO}_E}\;([\![U]\!]^0_{\eta +\!\!+ env(B)}\,\rho) \\ & \mathbf{in\;let_{IO}}\;ev' \leftarrow ioev\;nms \\ & \mathbf{in\;let_{IO}}\;(\mathsf{cmd} : ioev') \leftarrow \mathbf{lift^{IO}_E}\;([\![T]\!]^0_{\eta +\!\!+ env(B), x\mapsto ev'}\,\rho) \\ & \mathbf{in}\;ioev'\;nms\end{aligned}$$

and

$$(**) \Downarrow_{\mathbf{IO}} ea \Longrightarrow ea \in [\![\theta\;\tau]\!]_{(\overline{\Delta'},\theta\,\overline{\Gamma_r})}$$

Since the choice of $\overline{\Delta_e}$, $\overline{\Gamma_e}$ and nms is arbitrary s.t. (h) holds, we have

$$(*) \in [\![\texttt{IO}\;(\theta\;\tau)]\!]_{(\overline{\Delta'},\theta\,\overline{\Gamma_r})} = [\![\theta\;\texttt{IO}\;\tau]\!]_{(\overline{\Delta'},\theta\,\overline{\Gamma_r})}$$

as required.

case UNIT0:
(i) Straightforward.

case LETREC0:
(i) Similar to ABS0.

case LETREC1, UNITM1, LETM1, LIFT1:
(ii) and (iii): These cases all proceed as for case ABS1, RUNT1 and LET1.

case VAR0 with constant k:
(i) Straightforward.

case VAR1 with constant k:
(ii) and (iii): Constants are rebuilt as themselves and have the same type in every stage.

□

Bibliography

(Please note that all of the URLs mentioned in this bibliography were correct as of February 2001.)

[1] ABADI, M., CARDELLI, L., PIERCE, B., AND PLOTKIN, G. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems 13*, 2 (Apr. 1991), 237–268.

[2] ABADI, M., CARDELLI, L., PIERCE, B., AND RÉMY, D. Dynamic typing in polymorphic languages. *Journal of Functional Programming 5*, 1 (Jan. 1995), 111–130.

[3] ADLER, S., BERGLUND, A., ET AL. *Extensible Stylesheet Language (XSL) Version 1.0.* W3C Candidate Recommendation, Nov. 2000. Available at `http://www.w3.org/TR/2000/CR-xsl-20001121`.

[4] AIKEN, A. Introduction to set constraint-based program analysis. *Science of Computer Programming 35* (1999), 79–111.

[5] AIKEN, A., AND WIMMERS, E. L. Type inclusion constraints and type inference. In *Proceedings of the ACM SIGPLAN Conference on Functional Programming Languages and Computer Architecture (FPCA'93), Copenhagen, Denmark* (June 1993), ACM Press, pp. 31–41.

[6] BAILEY, D. H., BORWEIN, P. B., AND PLOUFFE, S. On the rapid computation of various polylogarithmic constants. *Mathematics of Computation 66*, 218 (Apr. 1997), 903–913.

[7] BARBANERA, F., DEZANI-CIANCAGLINI, M., AND DE'LIGUORO, U. Intersection and union types: Syntax and semantics. *Information and Computation 119*, 2 (June 1995), 202–230.

[8] BARENDREGT, H. P. *The Lambda Calculus: Its Syntax and Semantics.* North Holland, 1984.

[9] BARENDREGT, H. P. Lambda calculi with types. In *Handbook of Logic in Computer Science*, S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, Eds., vol. 2. Oxford Science Publishers, 1992, ch. 2, pp. 117–309.

[10] BENAISSA, Z. E.-A., MOGGI, E., TAHA, W., AND SHEARD, T. A categorical analysis of multi-level languages (extended abstract). Tech. Rep. CSE-98-018, Department of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology, Dec. 1998.

[11] BENTON, N., HUGHES, J., AND MOGGI, E. Monads and effects. In *International Summer School On Applied Semantics (APPSEM'2000), Caminha, Minho, Portugal* (Sept. 2000). Available at http://www.disi.unige.it/person/MoggiE/APPSEM00/BHM.ps.

[12] BRAY, T., PAOLI, J., SPERBERG-MCQUEEN, C. M., AND MALER, E. *Extensible Markup Language (XML) 1.0*, 2nd ed. W3C Recommendation, Oct. 2000. Available at http://www.w3.org/TR/2000/REC-xml-20001006.

[13] BRÜGGEMANN-KLEIN, A., AND WOOD, D. Unambiguous regular expressions and SGML document grammars. Technical Report 337, Computer Science Department, University of Western Ontario, London, Ontario, Canada, Nov. 1992.

[14] BUNEMAN, P., AND PIERCE, B. Union types for semistructured data. In *Proceedings of the International Database Programming Languages Workshop (DBPL-7), Kinloch Rannoch, Scotland* (Sept. 1999), R. Connor and A. O. Mendelzon, Eds., LNCS 1949, Springer-Verlag. Also available as University of Pennsylvania Dept. of CIS technical report MS-CIS-99-09.

[15] BURNS, J., LAUZON, M., AND HEIN, R. A. *eXtensible Programming Language (XPL) Specification*, July 2000. Draft available at http://www.vbxml.com/xpl/spec_draft.asp.

[16] CARDELLI, L., AND MITCHELL, J. Operations on records. *Mathematical Structures in Computer Science 1*, 1 (Mar. 1991), 3–48.

[17] CHAMPARNAUD, J.-M., ZIADI, D., AND PONTY, J.-L. Determinization of Glushkov automata. In *Automata Implementation: Third International Workshop on Implementing Automata, (WIA'98), Rouen, France* (Sept. 1998), J.-M. Champarnaud, D. Maurel, and D. Ziadi, Eds., LNCS 1660, Springer-Verlag, pp. 57–68.

[18] CLARK, J. *XSL Transformations (XSLT).* W3C Recommendation, Nov. 1999. Available at http://www.w3.org/TR/1999/REC-xslt-19991116.

[19] DAMAS, L., AND MILNER, R. Principle type schemes for functional programs. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albequerque, New Mexico* (Jan. 1982), ACM Press, pp. 207–212.

[20] DAVIES, R. A temporal logic approach to binding-time analysis. In *Proceedings of the Eleventh Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey* (July 1996), E. Clarke, Ed., IEEE Computer Society Press, pp. 184–195.

[21] DAVIES, R., AND PFENNING, F. A modal analysis of staged computation. In *Proceedings of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida* (Jan. 1996), ACM Press, pp. 258–270.

[22] DECISIONSOFT. *XML Script and its uses*, 2000. Available at http://www.DecisionSoft.com/TechnicalDescription.html.

[23] ELLIOTT, C., AND HUDAK, P. Functional reactive animation. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP'97), Amsterdam, The Netherlands* (June 1997), ACM Press, pp. 263–273.

[24] FALLSIDE, D. C. *XML Schema Part 0: Primer.* W3C Candidate Recommendation, Oct. 2000. Available at `http://www.w3.org/TR/xmlschema-0`.

[25] FANKHAUSER, P., FERNÁNDEZ, M., MALHOTRA, A., RYS, M., SIMÉON, J., AND WADLER, P. *The XML Query Algebra.* W3C Working Draft, Dec. 2000. Available at `http://www.w3.org/TR/2000/WD-query-algebra-20001204/`.

[26] FERNÁNDEZ, M., AND ROBI, J. *XML Query Data Model.* W3C Working Draft, May 2000. Available at `http://www.w3.org/TR/2000/WD-query-datamodel-20000511/`.

[27] FINNE, S., AND PEYTON JONES, S. L. Composing the user interface with Haggis. In *Advanced Functional Programming: Second International School, Olympia, Washington* (Aug. 1996), J. Launchbury, E. Meijer, and T. Sheard, Eds., LNCS 1129, pp. 1–37.

[28] FIORE, M., PLOTKIN, G., AND TURI, D. Abstract syntax and variable binding. In *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science (LICS'99), Trento, Italy* (July 1999), G. Longo, Ed., IEEE Computer Society Press, pp. 193–202.

[29] FLANAGAN, D. *Javascript: The Definitive Guide.* O'Reilly and Associates, June 1998.

[30] GABBAY, M., AND PITTS, A. A new approach to abstract syntax involving binders. In *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science (LICS'99), Trento, Italy* (July 1999), G. Longo, Ed., IEEE Computer Society Press, pp. 214–224.

[31] GASTER, B. R., AND JONES, M. P. A polymorphic type system for extensible records and variants. Tech. Rep. NOTTCS-TR-96-3, Department of Computer Science, University of Nottingham, Nov. 1996.

[32] GIRARD, J.-Y. The system F of variable types, Fifteen years later. In *Logical Foundations of Functional Programming*, G. Huet, Ed. Addison-Wesley, 1990, ch. 6, pp. 87–126.

[33] GOMARD, C. K., AND JONES, N. D. A partial evaluator for the untyped lambda calculus. *Journal of Functional Programming 1*, 1 (Jan. 1991), 21–69.

[34] GOSLING, J., JOY, B., AND STEELE, G. *The Java Language Specification.* Addison-Wesley, 1996.

[35] HARPER, R., AND PIERCE, B. A record calculus based on symmetric concatenation. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages (POPL'91), Orlando, Florida* (Jan. 1991), ACM Press, pp. 131–142.

[36] HARRISON, W. L., AND KAMIN, S. N. Metacomputation-based compiler architecture. In *Fifth International Conference of Mathematics of Program Construction (MPC 2000), Ponte de Lima, Portugal* (July 2000), J. N. Oliveira and R. C. Backhouse, Eds., LNCS 1837, Springer-Verlag.

[37] HINZE, R. A new approach to generic functional programming. In *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'00), Boston, Massachusetts* (Jan. 2000), ACM Press, pp. 119–132.

[38] HOSOYA, H., AND PIERCE, B. C. XDuce: A typed XML processing language. In *Proceedings of Third International Workshop on the Web and Databases (WebDB2000), Dallas, Texas* (May 2000). Available at `http://www.cis.upenn.edu/ hahosoya/papers/xduce-prelim.ps`.

[39] HOSOYA, H., AND PIERCE, B. C. Regular expression matching for XML. In *Proceedings of the 28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01), London, England* (Jan. 2001), ACM Press, pp. 67–80.

[40] HOSOYA, H., VOUILLON, J., AND PIERCE, B. C. Regular expression types for XML. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00), Montreal, Canada* (Sept. 2000), ACM Press, pp. 11–22.

[41] HUDAK, P. Modular domain specific languages and tools. In *Fifth International Conference on Software Reuse (ICSR'98), Victoria, B.C., Canada* (1998), P. Devanbu and J. Poulin, Eds., IEEE Computer Society Press, pp. 134–142.

[42] HUDAK, P., MAKUCEVICH, T., GADDE, S., AND WHONG, B. Haskore music notation–an algebra of music. *Journal of Functional Programming 6*, 3 (May 1996), 465–483.

[43] HUGHES, J. Why functional programming matters. *The Computer Journal 32*, 2 (Feb. 1989), 98–107.

[44] HUGHES, J. The design of a pretty-printing library. In *Advanced Functional Programming* (1995), J. Jeuring and E. Meijer, Eds., LNCS 925, Springer-Verlag, pp. 53–96.

[45] ISO 8879: Standard generalized markup language (SGML), 1986.

[46] JONES, M., AND PEYTON JONES, S. L. Lightweight extensible records for Haskell. In *Proceedings of the 1999 Haskell Workshop, Paris, France* (Oct. 1999). Available as Technical Report UU-CS-1999-28, Department of Computer Science, University of Utrecht.

[47] JONES, M. P. *Qualified Types: Theory and Practice.* Distinguished Dissertations in Computer Science. Cambridge University Press, 1994.

[48] JONES, M. P. Simplifying and improving Qualified Types. Tech. Rep. YALEU/DCS/RR-1040, Computer Science Department, Yale University, New Haven, Connecticut, June 1994. Shorter version appears in FPCA'95, 160–169.

[49] JONES, M. P. First-class polymorphism with type inference. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris, France* (Jan. 1997), ACM Press, pp. 483–496.

[50] JONES, N. D., GOMARD, C. K., AND SESTOFT, P. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International, 1993.

[51] JONES, N. D., SESTOFT, P., AND SØNDERGAARD, H. An experiment in partial evaluation: the generation of a compiler generator. *ACM SIGPLAN Notices 20*, 8 (Aug. 1985), 82–87.

[52] KAHL, W. Beyond pretty-printing: Galley concepts in document formatting combinators. In *First International Workshop on Practical Aspects of Declarative Languages (PADL'99), San Antonio, Texas* (Jan. 1999), LNCS 1551, Springer-Verlag, pp. 76–90.

[53] LANDIN, P. J. The next 700 programming languages. *Communications of the ACM 9*, 3 (Mar. 1966), 157–164.

[54] LEE, P., AND LEONE, M. Optimizing ML with run-time code generation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'96), Philadelphia, Pennsylvania* (May 1996), ACM Press, pp. 137–148.

[55] LEIJEN, D., AND MEIJER, E. Domain specific embedded compilers. In *Proceedings of the Second USENIX Conference on Domain-Specific Languages (DSL'99), Austin, Texas* (Oct. 1999), USENIX Association, pp. 109–122. Also appears in *ACM SIGPLAN Notices 35*, 1, (Jan. 2000).

[56] LEROY, X., AND MAUNY, M. Dynamics in ML. *Journal of Functional Programming 3*, 4 (1993), 431–463.

[57] LEWIS, J., SHIELDS, M., MEIJER, E., AND LAUNCHBURY, J. Implicit parameters: Dynamic scoping with static types. In *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'00), Boston, Massachusetts* (Jan. 2000), ACM Press, pp. 108–118.

[58] LIE, H. W., AND BOS, B. *Cascading Style Sheets (CSS), level 1*, revised ed. W3C Recommendation, Jan. 1999. Available at `http://www.w3.org/TR/1999/REC-CSS1-19990111`.

[59] MACQUEEN, D., PLOTKIN, G., AND SETHI, R. An ideal model for recursive polymorphic types. In *Proceedings of the Eleventh Annual ACM Symposium on Principles of Programming Languages (POPL'84), Salt Lake City, Utah* (1984), ACM Press, pp. 165–174.

[60] MARAIS, H. *Compaq's Web Language: A Programming Language for the Web*, 1999. Available at `http://www.research.compaq.com/SRC/WebL/WebL.pdf`.

[61] MARTI, J. B., HEARN, A. C., GRISS, M. L., AND GRISS, C. Standard Lisp report. *ACM SIGPLAN Notices 14*, 10 (Oct. 1979), 48–68.

[62] MASSALIN, H. *Synthesis: An Efficient Implementation of Fundamental Operating System Services.* PhD thesis, Columbia University, 1992.

[63] MATTHEWS, J., LAUNCHBURY, J., AND COOK, B. Microprocessor specification in Hawk. In *International Conference on Computer Languages (ICCL'98), Chicago, Illinois* (May 1998), IEEE Computer Society Press, pp. 90–101.

[64] MEIJER, E. Server side Web scripting in Haskell. *Journal of Functional Programming 10*, 1 (Jan. 2000), 1–18.

[65] MEIJER, E., AND SHIELDS, M. XMλ: A functional language for constructing and manipulating XML documents. Unpublished draft. Available at `http://www.cse.ogi.edu/~mbs/pub/xmlambda/xmlambda.ps`), 1999.

[66] MILNER, R. A theory of type polymorphism in programming. *Journal of Computer and System Sciences 17* (1978), 348–375.

[67] MILNER, R., TOFTE, M., HARPER, R., AND MACQUEEN, D. *The Definition of Standard ML (Revised).* The MIT Press, Oct. 1997.

[68] MILO, T., SUCIU, D., AND VIANU, V. Typechecking for XML transformers. In *Proceedings of the Nineteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (SIGMOD/PODS 2000), Dallas, Texas* (May 2000), ACM Press, pp. 11–22.

[69] MITCHELL, J. C. *Foundations of Programming Languages.* The MIT Press, 1996.

[70] MOGGI, E. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science* (June 1989), IEEE Computer Society Press, pp. 14–23.

[71] MOGGI, E. A categorical account of two-level languages. In *Proceedings of the Thirteenth Annual Conference on Mathematical Foundations of Programming Semantics* (1997), ENTCS 6, Elsevier Science Publishers.

[72] MOGGI, E. Metalanguages and applications. In *Semantics and Logics of Computation*, A. M. Pitts and P. Dybjer, Eds., Newton Institute Publications. Cambridge University Press, 1997.

[73] MOGGI, E. Functor categories and two-level languages. In *Foundations of Software Science and Computation Structure, First International Conference, (FoSSaCS'98)* (1998), M. Nivat, Ed., LNCS 1378, Springer-Verlag, pp. 211–225.

[74] NICOL, G. T. *XEXPR - A Scripting Language for XML.* W3C Note, Nov. 2000. Available at `http://www.w3.org/TR/2000/NOTE-xexpr-20001121/`.

[75] NIELSON, F. *Abstract Interpretation using Domain Theory.* PhD thesis, Department of Computer Science, University of Edinburgh, Oct. 1984.

[76] NIELSON, F., AND NIELSON, H. R. Two-level semantics and code generation. *Journal of Theoretical Computer Science 56*, 1 (Jan. 1988), 59–133.

[77] NIELSON, F., AND NIELSON, H. R. *Two-level Functional Languages.* Cambridge University Press, 1992.

[78] NIELSON, F., AND NIELSON, R. H. Automatic binding time analysis for a typed λ-calculus. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages (POPL'88), San Diego, Calafornia* (1988), ACM Press, pp. 98–106.

[79] ODERSKY, M., SULZMANN, M., AND WEHR, M. Type inference with constrained types. *Theory and Practice of Object Systems 5*, 1 (1999), 35–55. An earlier version appears in FOOL 4, 1997.

[80] OHORI, A. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems 17*, 6 (Nov. 1995), 844–895. An earlier version appears in POPL'93, pp. 99–112.

[81] OLES, F. J. Type algebras, functor categories, and block structure. In *Algebraic Methods in Semantics*, M. Nivat and J. C. Reynolds, Eds. Cambridge University Press, 1985, pp. 543–573.

[82] PELEGRÍ-LLOPART, E. JavaServer pages specification (version 1.2). Tech. Rep. JSR-000053, Sun Microsystems, Inc., Palo Alto, California, Oct. 2000. Available at `http://java.sun.com/aboutJava/communityprocess/first/jsr053/jsp12.pdf`.

[83] PETERSON, J., HUDAK, P., AND ELLIOTT, C. Lambda in motion: Controlling robots with Haskell. In *First International Workshop on Practical Aspects of Declarative Languages (PADL'99), San Antonio, Texas* (Jan. 1999), LNCS 1551, Springer-Verlag, pp. 91–105.

[84] PEYTON JONES, S. L., EBER, J.-M., AND SEWARD, J. Composing contracts: an adventure in financial engineering. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00), Montreal, Canada* (Sept. 2000), ACM Press, pp. 280–292.

[85] PEYTON JONES, S. L., HUGHES, J., ET AL. *Haskell 98: A Non-strict, Purely Functional Language*, Feb. 1999. Available at `http://www.haskell.org/onlinereport/`.

[86] PEYTON JONES, S. L., REID, A., HOARE, T., MARLOW, S., AND HENDERSON, F. A semantics for imprecise exceptions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99), Atlanta, Georgia* (May 1999), ACM Press, pp. 25–36. Also appears in *ACM SIGPLAN Notices 34*, 5 (May 1999).

[87] PEYTON JONES, S. L., AND WADLER, P. Imperative functional programming. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages (POPL'93), Charleston, South Carolina* (Jan. 1993), ACM Press, pp. 71–84.

[88] PFENNING, F., AND DAVIES, R. A modal analysis of staged computation. Tech. Rep. CMU-CS-99-153, School of Computer Science, Carnegie Mellon University, Aug. 1999. An earlier version appears in POPL'96, pp. 258–270.

[89] PIERCE, B. C. Intersection types and bounded polymorphism. *Mathematical Structures in Computer Science 7*, 2 (Apr. 1997), 129–193.

[90] PLOTKIN, G. D. A powerdomain construction. *SIAM Journal of Computing 5*, 3 (1976), 452–487.

[91] RAGGETT, D., LE HORS, A., AND JACOBS, I. *HTML 4.01 Specification.* W3C Recommendation, Dec. 1999. Available at `http://www.w3.org/TR/1999/REC-html401-19991224/`.

[92] REID, A., PETERSON, J., HAGER, G., AND HUDAK, P. Prototyping real-time vision systems: An experiment in DSL design. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99), Los Angeles, California* (May 1999), ACM Press, pp. 484–493.

[93] RÉMY, D. Typing record concatenation for free. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages (POPL'92), Albuquerque, New Mexico* (Jan. 1992), ACM Press, pp. 166–176.

[94] RÉMY, D. Type inference for records in a natural extension of ML. In *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*, C. A. Gunter and J. C. Mitchell, Eds. The MIT Press, 1993. An earlier verision appears in POPL'89, pp. 77–87.

[95] REYNOLDS, J. C. Design of the programming language FORSYTHE. In *ALGOL-like Languages*, P. W. O'Hearn and R. D. Tennent, Eds. Birkhäuser, 1997, pp. 173–233.

[96] SANDHOLM, A., AND SCHWARTZBACH, M. I. A type system for dynamic Web documents. In *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'00), Boston, Massachusetts* (Jan. 2000), ACM Press, pp. 290–301.

[97] SHEARD, T., BENAISSA, Z., AND MARTEL, M. *Introduction to Multistage Programming Using MetaML*, 2nd ed. Pacific Software Research Center, Oregon Graduate Institute, 2000. Available at `http://cse.ogi.edu/~sheard/papers/manual.ps`.

[98] SHIELDS, M., AND MEIJER, E. Type-indexed rows. In *Proceedings of the 28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01), London, England* (Jan. 2001), ACM Press, pp. 261–275.

[99] SHIELDS, M., SHEARD, T., AND PEYTON JONES, S. L. Dynamic typing by staged type inference. In *Proceedings of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California* (Jan. 1998), ACM Press, pp. 289–302.

[100] STOLZENBURG, F. An algorithm for general set unification and its complexity. *Journal of Automated Reasoning 22*, 1 (Jan. 1999), 45–63.

[101] SULZMANN, M. A general type inference framework for Hindley/Milner style systems. Tech. Rep. TR2000/15, Department of Computer Science, The University of Melbourne, July 2000.

[102] SWIERSTRA, S. D., AND ALCOCER, P. R. A. Fast, error correcting parser combinators: A short tutorial. In *Theory and Practice of Informatics, 26th Seminar on Current Trends in Theory and Practice of Informatics (SOFSEM'99)* (Nov. 1999), J. Pavelka, G. Tel, and M. Bartosek, Eds., LNCS 1725, Springer-Verlag, pp. 111–129.

[103] SWIERSTRA, S. D., ALCOCER, P. R. A., AND SARAIAVA, J. Designing and implementing combinator languages. In *Advanced Functional Programming, Third International School, (AFP'98)* (1999), LNCS 1608, Springer-Verlag, pp. 150–206.

[104] TAHA, W. *Multi-Stage Programming: Its Theory and Applications.* PhD thesis, Department of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology, 1999.

[105] TAHA, W., BENAISSA, Z., AND SHEARD, T. Multi-stage programming: Axiomatization and type-safety. In *Proceedings 25'th International Colloquium on Automata, Languages, and Programming (ICALP '98), Aalborg, Denmark* (July 1998), LNCS 1443, Springer-Verlag, pp. 918–929.

[106] TAHA, W., MOGGI, E., BENAISSA, Z., AND SHEARD, T. An idealized MetaML: Simpler, and more expressive. In *Proceedings of the European Symposium On Programming (ESOP'99)* (1999), LNCS 1576, Springer-Verlag, pp. 193–207.

[107] TAHA, W., AND SHEARD, T. Multi-stage programming with explicit annotations. In *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM'97), Amsterdam, The Netherlands* (1997), ACM Press, pp. 203–217.

[108] WADLER, P. Monads for functional programming. In *Advanced Functional Programming: First International Spring School, Bastad, Sweden* (1995), J. Jeuring and E. Meijer, Eds., LNCS 925, Springer-Verlag, pp. 24–52.

[109] WADLER, P., AND BLOTT, S. How to make *ad-hoc* polymorphism less *ad hoc.* In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages (POPL'89), Austin, Texas* (Jan. 1989), ACM Press, pp. 60–76.

[110] WALL, L., CHRISTIANSEN, T., AND ORWANT, J. *Programming Perl*, 3rd ed. O'Reilly and Associates, July 2000.

[111] WALLACE, M., AND RANCIMAN, C. Haskell and XML: Generic combinators or type-based translation? In *Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming (ICFP'99), Paris, France* (Sept. 1999), ACM Press, pp. 148–159.

[112] WAND, M. Complete type inference for simple objects. In *Proceedings of the Second Annual IEEE Symposium on Logic in Computer Science, (LICS'87), Ithaca, New York* (June 1987), IEEE Computer Society Press, pp. 37–44. Corrigendum in LICS'88, p. 132.

[113] WAND, M. Type inference for record concatenation and multiple inheritance. *Information and Computation 93*, 1 (July 1991), 1–15.

[114] WELLS, J. B. Typability and type-checking in the second-order λ-calculus are equivalent and undecidable. In *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science (LICS'94), Paris, France* (1994), IEEE Computer Society Press, pp. 176–185.

[115] WRIGHT, A. K., AND FELLEISEN, M. A syntactic approach to type soundness. *Information and Computation 115*, 1 (Nov. 1994), 38–94.

[116] XML FOR ALL INC. *XFA Reference Manual*, 1999. Available at http://www.xmlforall.com/cgi-bin/xfa?doc:doc40.

Biographical Sketch

Mark Shields was born on the 13th of April, 1969 in Melbourne, Australia. He was awarded a Bachelors of Science, majoring in Computer Science, from Monash University in 1991, and a Bachelors of Science (Honours) in Computer Science from The University of Melbourne in 1996. He worked as a software developer between 1990 and 1995. He began his PhD in 1996 at the University of Technology, Sydney, transferred in 1997 to the University of Glasgow, and transferred again in 1998 to the Oregon Graduate Institute of Science and Technology. His research centers on exploiting type systems to increase the utility, expressiveness and verifiability of programming languages.