Annotated Type Systems for Program Analysis

Kirsten Lackner Solberg
Computer Science Department,
Aarhus University, Denmark,
e-mail: kls@daimi.aau.dk

November 20, 1995

Acknowledgement

First of all I would like to thank my supervisor, Hanne Riis Nielson, for support and encouragement. I would also like to thank Flemming Nielson for valuable discussions.

Part of my research was carried out while visiting Cambridge University; the visit was supported by the Danish Research Academy. Thanks to Alan Mycroft for taking the time to explain lots of things to me. Thanks to Nick Benton for interesting discussions. Thanks to Sebastian Hunt for explaining further about PERs in abstract interpretation. I would like to thank Fritz Henglein for his hints and tips for solving the constraints in Chapter 5. Thanks also to LOMAPS (Esprit Basic Research) and DART (Danish Science Research Council) for partial funding. Thanks to Chris Hankin and Daniel Le Métayer for answering questions about their work.

I am grateful to the people at IMADA, especially Søren Larsen and Peter Kornerup for getting me started. Thanks to Odense Univerty for funding.

Thanks to all the people who have been reading draft versions of my work: Hanne Riis Nielson, Flemming Nielson, Torben Amtoft, Olivier Danvy, David Wright, John Immerkær.

And to all those people who never lost faith in me, celebrated my victories with me and helped me through the difficult times: my fiancé Jan Gasser, Per Waaben Hansen, Mette-Helene Beck, my father, Elisabeth, Sif and Thorlak, my mother, Anne and Ole, ...

Declaration

Part of the work in Chapter 2 is published as [SNN94] and co-authored by Hanne Riis Nielson and Flemming Nielson. The paper has been invited for publication in a special issue of the journal "Science of Computer Programming", devoted to SAS'94. The work in Chapter 3 and Chapter 4 will be published as [Sol95]. Parts of Chapter 5 is published as [SNN92] and co-authored by Hanne Riis Nielson and Flemming Nielson. A full version is reported in [Sol93]. The work in Chapter 6 is submitted for publication as [MS95] and is co-authored by Alan Mycroft.

Preface to the Revised Version

This report is a revised version of my Ph. D. thesis of the same title, which was accepted for the Ph. D. degree in Computer Science at Odense University, Denmark, in July 1995.

I would like to take this opportunity to thank the examiners, Joan Boyar, Department of Math. and Computer Science, Odense University, Denmark, Chris Hankin, Department of Computing, Imperial College, University of London, United Kingdom, and Bo Stig Hansen, Department of Computer Science, DTU, Denmark.

An expanded version of [SNN94] will appear in a special issue of the journal "Science of Computer Programming", devoted to SAS'94. A revised version of Chapter 6 appeared at PLILP'95 [MS95].

The Appendix with proofs and implementations is available from the author.

Contents

List of Tables

List of Figures

Danish Summary

Annoterede Typesystemer til Programanalyse

Det er velkendt at programanalyse er et brugbart vrktj nr programmeringssprog skal implementeres effektivt. Vi ser her p et par eksempler: For funktionelle sprog er *strictness-analyse* brugbart: en funktion er strict, hvis den anvendt p et ikke terminerende argument vil resultere i en beregning, der ikke terminerer. En strictness-analyse vil finde ud af om en funktion er strict. Analysen skal vre plidelig, dvs. hvis analysen siger at en funktion er strict, s er funktionen rent faktisk strict, men en funktion kan godt vre strict selvom analysen siger at den ikke er. For en strict funktion er det sikkert at beregne argumentet fr funktionskaldet, og derved optimere funktionskaldet.

Et andet eksempel er *totalitetsanalyse*. Her er mlet at finde ud af om en funktion er total, dvs. om funktionen anvendt p et terminerende argument vil terminere. Ogs for totale funktioner er det sikkert at beregne argumentet fr funktionskaldet.

Et sidste eksempel er *bindingstidsanalyse*. Analysen introducerer en skelnen mellem data tilgngelig p oversttelsestid eller p krselstid. Nr et argument til en funktion er kendt p oversttelsestidspunktet, s kan funktionen specialiseres med hensyn til dette argument p oversttelsestidspunktet, og derved reduceres krselstiden p bekostning at get oversttelsestid. Udfoldning af rekursion kan introducere ikke-terminerende beregninger p oversttelsestidspunktet. Dette er mske ikke nsket, selvom det oprindelig program ikke vil terminere. Her kan totalitetsanalysen hjlpe med information om hvornr det er sikkert at udfolde. Bindingstidsanalyse afviger fra strictness-analyse og totalitetsanalyse ved at ikke at interessere sig for en egenskab ved den vrdi, som programmet beregner, men for selve beregningen.

Forskellige teknikker til specifikation af programanalyser er udviklet: bl.a. *abstrakt fortolkning* [BHA86, Myc81] og *projektionsanalyse* [WH87, Lau91]. I abstrakt fortolkning gives en abstrakt vrdi til programmet i stil med denotationssemantik, hvor en konkret mening tildeles programmet.

I den seneste tid har flere forskere, deriblandt [NN88, KM89, Ben93, TJ92a, Wri91], anvendt typesystemer til at specificere programanalyser. Ideen er at annotere typerne med program analyse information. For et udtryk med standard typen $\mathtt{t}_1 \to \mathtt{t}_2$ kan vi annotere typekonstruktren, dvs. funktionspilen, med programanalyse informationen s som i $\mathtt{t}_1 \to^s \mathtt{t}_2$. Vi vil forst dette som "nr funktionen er beregnet, s vil den udvise opfrelsen s". En anden mde at annotere typer p er $(\mathtt{t}_1 \to \mathtt{t}_2)^s$ og vi vil forst det som "dette udtryk vil evaluere til en funktion med egenskaben s". For strictness-analyse er et muligt valg af annoteringer:

$$s ::= \bot \mid \top$$

Et program med typen $(\mathtt{t}_1 \to \mathtt{t}_2)^\bot$ vil evaluere til en strict funktion, der tager argumenter af typen $\mathtt{t}_1$ og giver et resultat af typen $\mathtt{t}_2$. Et program med typen $(\mathtt{t}_1 \to \mathtt{t}_2)^\top$ vil evaluere til en funktion, der tager argumenter af typen $\mathtt{t}_1$ og giver et resultat af typen $\mathtt{t}_2$ og vi ved ikke noget om dens opfrsel. Til totalitetsanalyse annoteringerne kan f.eks. vre:

$$s ::= \not\bot \mid \top$$

En total funktion fra $\mathtt{t}_1$ til $\mathtt{t}_2$ vil have typen $(\mathtt{t}_1 \to \mathtt{t}_2)^{\not\bot}$. Til bindingstids-analyse er et valg af annoteringer:

$$s ::= \mathbf{r} \mid \mathbf{c}$$

Et program med typen $\mathtt{t}_1 \to^{\mathbf{c}} \mathtt{t}_2$ vil f sit argument p oversttelsestidspunktet, hvorimod et program af types $\mathtt{t}_1 \to^{\mathbf{r}} \mathtt{t}_2$ vil f sit argument p krselstidspunktet.

I denne afhandling flges denne trend. I **Kapitel 1** ser vi p flere eksempler fra litteraturen af analyser specificeret ved typesystemer.

I **Kapitel 2** prsenterer vi en kombineret *strictness- og totalitetsanalyse*. Vi specificerer analysen som et annoteret typesystem. Typesystemet tillader konjunktion af annoterede types, men kun p verste niveau. Denne analyse er kraftigere end Kuo og Mishra's [KM89] strictness-analyse, da vi inkluderer totalitets egenskaber. Analysen vises sund med hensyn til en operations semantik. Det er ikke umiddelbart hvordan analysen kan udvides til "fuld" konjunktion.

Analysen i **Kapitel 3** er ogs en kombineret strictness- og totalitetsanalyse, men med "fuld" konjunktion. Sundhed af analysen er vist med hensyn til en denotations semantik. Analysen er kraftigere end strictness-analyserne af Jensen [Jen92a] og Benton [Ben93] — igen fordi vi ogs inkluderer totalitets egenskaber.

Indtil nu har vi kun set p *specifikation* af analyser, men for at de kan vre praktisk brugbare har vi brug for an algoritme, der kan rekonstruere de annoterede typer. I **Kapitel 4** konstruerer vi en algoritme for analysen i Kapitel 3 ved at anvende *lazy type* metoden af Hankin and Le Métayer [HM94a]. Grunden til at vi har valgt analysen i kapitel 3, er at metoden ikke kan anvendes p analysen fra Kapitel 2.

I **Kapitel 5** studerer vi en bindingstidsanalyse. Vi tager analysen specificeret af Nielson og Nielson [NN92] og vi konstruerer en mere effektiv algoritme end den foreslet i [NN92]. Algoritmen opsamler "constraints" ved strukturelt at g igennem udtrykket, ligesom standard type rekonstruktions algoritmen $\mathcal{T}$ [Dam85]. Bagefter beregnes den minimale lsning til mngden af "constraints".

Analysen i **Kapitel 6** er specificeret ved abstrakt fortolkning. Hunt [Hun91] har vist at projektions baseret analyse er inkluderet i PER (partiel kvivalens relation) baserede analyser i abstrakt fortolkning. De PERs, som Hunt bruger, er stricte, dvs. bundelementet er relateret til bundelementet. I Kapitel 6 fjerner vi denne restriktion ved at krve at PER'erne er *uniforme*, p den mde at de behandler alle tal ens. Ved at tillade ikke stricte PERs fr vi tre *uniforme egenskaber* p `Int`: $\{\perp, \mathbb{Z}, \mathbb{Z}_\perp\}$. De korresponderer til de tre annoteringer, **b**, **n** og $\top$, brugt i Kapitel 2 og 3.

Chapter 1

Introduction

It is well known that program analysis is a useful tool in the efficient implementation of programming languages. Let us give a few examples. For the lazy functional languages *strictness analysis* is profitable: a function is strict if its application to a looping argument results in a looping computation. A strictness analysis detects safely when functions are strict: whenever the analysis says a function is strict then the function is indeed strict, however the function *may be* strict even though the analysis says it is not. Function application can be optimised using strictness information because for strict functions it is safe to evaluate the argument before performing the function call and hence enforcing a call-by-value evaluation of function applications.

Another example is *totality analysis*. Here the goal is to determine whether a function is total, i.e. that the application of the function to any argument results in a terminating computation. For a total function it is also safe to evaluate the argument before performing the function call and thereby enforce call-by-value evaluation of function applications.

A final example is *binding time* analysis. The analysis introduces a distinction between data available at compile-time and at run-time. Whenever an argument to a function is already known at compile-time it is possible to specialise the function to that particular argument at compile-time and thereby reduce the evaluation time at run-time (at the cost of evaluation time at compile-time). Unfolding of recursion may introduce looping at compile-time which sometimes is not desirable though the original program would loop too. Here the totality analysis can help by providing information to tell when it is safe to do the unfolding. Binding time analysis differs from strictness analysis and totality analysis in that it is not concerned with a property of the value the program evaluates to but with the evaluation of the program.

A number of techniques have been developed for the specification of program analysis including *abstract interpretation* [BHA86, Myc81] and *projection analysis* [WH87, Lau91]. In abstract interpretation, there is associated an abstract value with each program in the spirit of denotational semantics where a denotation is associated with each program. In the projection based approach a projection is associated with each program. The analyses come in two flavours: forward analyses and backward analyses. Forward analysis amounts to: given a property for the argument to a function what is the property of the result of the application of the function. In backwards analysis we want to find the property of the argument given the property of the result of the function application.

More recently, the use of type systems for program analysis have been pursued by a number of researchers including [NN88, KM89, Ben93, TJ92a, Wri91]. The idea is to annotate the types with program analysis information. If an expression has the standard type $\mathtt{t}_1 \rightarrow \mathtt{t}_2$ we can annotate the type constructor, i.e. the function arrow, with the program analysis information s as in $\mathtt{t}_1 \rightarrow^s \mathtt{t}_2$. We will interpret this as "when the function is evaluated it will exhibit the behaviour s". Another way to annotate the type is $(\mathtt{t}_1 \rightarrow \mathtt{t}_2)^s$ and we can interpret this as "the expression will evaluate to a function with the property s". For a strictness analysis one possible choice of annotations is

$$s ::= \bot \mid \top$$

A program with the type $(\mathtt{t}_1 \rightarrow \mathtt{t}_2)^\bot$ evaluates to a strict function that takes programs of type $\mathtt{t}_1$ as argument and the result of the application has the type $\mathtt{t}_2$. A program of type $(\mathtt{t}_1 \rightarrow \mathtt{t}_2)^\top$ evaluates to any function that takes arguments of type $\mathtt{t}_1$ and the result of the application has the type $\mathtt{t}_2$, i.e. we do not know whether the function is strict or not. The annotations for a totality analysis can be

$$s ::= \not\bot \mid \top$$

A total function from $\mathtt{t}_1$ to $\mathtt{t}_2$ will then have the type $(\mathtt{t}_1 \rightarrow \mathtt{t}_2)^{\not\bot}$. For binding time analysis the annotations might be

$$s ::= \mathbf{r} \mid \mathbf{c}$$

A program of type $\mathtt{t}_1 \rightarrow^{\mathbf{c}} \mathtt{t}_2$ will be supplied with its argument at compile-time whereas a program of type $\mathtt{t}_1 \rightarrow^{\mathbf{r}} \mathtt{t}_2$ will be supplied with its argument at run-time.

In this thesis we shall follow this trend. The development is preformed for a simply typed lambda calculus. We will mainly work combined strictness and totality analysis and with binding time analysis.

In the remainder of this chapter we shall give an overview of different approaches found in the literature and show how the work of this thesis fits into the picture.

1.1 The Standard Type System

We consider the simply typed λ-calculus with constants. The standard types are either base-types or function types:

$$\mathtt{t} ::= \mathtt{B} \mid \mathtt{t} \rightarrow \mathtt{t}$$

and the base-types (the $\mathtt{B}$'s) may include `Int` and `Bool` and in some cases type variables. The terms are:

$$\mathtt{e} ::= \mathtt{x} \mid \lambda\mathtt{x}.\mathtt{e} \mid \mathtt{e}\ \mathtt{e} \mid \mathtt{c} \mid \mathtt{if\ e\ then\ e\ else\ e} \mid \mathtt{fix\ e}$$

The constants (the $\mathtt{c}$'s) include `true` and `false` of type `Bool` and all the integers of type `Int` in addition to `+`, `*`, ... of type $\mathtt{Int} \rightarrow \mathtt{Int} \rightarrow \mathtt{Int}$. We will only consider terms that are typeable according to the type inference rules defined in Figure 1.1 and for simplicity we shall require that the bound variables are different. The list A of assumptions gives types to free variables and for each constant $\mathtt{c}$ there is an type $\mathtt{t}_{\mathtt{c}}$, e.g.

$$\begin{aligned} \mathtt{t}_7 &= \mathtt{Int} \\ \mathtt{t}_{\mathtt{true}} &= \mathtt{Bool} \\ \mathtt{t}_{\mathtt{false}} &= \mathtt{Bool} \\ \mathtt{t}_+ &= \mathtt{Int} \rightarrow \mathtt{Int} \rightarrow \mathtt{Int} \end{aligned}$$

The set of free variables in the term $\mathtt{e}$ is written $\mathrm{FV}(\mathtt{e})$ and the usual substitution on terms is written $\mathtt{e}[\mathtt{e}_2/\mathtt{x}]$, where $\mathtt{e}[\mathtt{e}_2/\mathtt{x}]$ is the term $\mathtt{e}$ where all free occurrences of $\mathtt{x}$ are replaced by $\mathtt{e}_2$.

$$[\text{var}]\ \frac{}{\mathrm{A} \vdash \mathtt{x} : \mathtt{t}} \quad \text{if } \mathtt{x} : \mathtt{t} \in \mathrm{A}$$

$$[\text{abs}]\ \frac{\mathrm{A}, \mathtt{x} : \mathtt{t}_1 \vdash \mathtt{e} : \mathtt{t}_2}{\mathrm{A} \vdash \lambda\mathtt{x}.\mathtt{e} : \mathtt{t}_1 \to \mathtt{t}_2}$$

$$[\text{app}]\ \frac{\mathrm{A} \vdash \mathtt{e}_0 : \mathtt{t}_1 \to \mathtt{t}_2 \quad \mathrm{A} \vdash \mathtt{e}_1 : \mathtt{t}_1}{\mathrm{A} \vdash \mathtt{e}_0\ \mathtt{e}_1 : \mathtt{t}_2}$$

$$[\text{if}]\ \frac{\mathrm{A} \vdash \mathtt{e}_1 : \mathtt{Bool} \quad \mathrm{A} \vdash \mathtt{e}_2 : \mathtt{t} \quad \mathrm{A} \vdash \mathtt{e}_3 : \mathtt{t}}{\mathrm{A} \vdash \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 : \mathtt{t}}$$

$$[\text{fix}]\ \frac{\mathrm{A} \vdash \mathtt{e} : \mathtt{t} \to \mathtt{t}}{\mathrm{A} \vdash \mathtt{fix}\ \mathtt{e} : \mathtt{t}}$$

$$[\text{const}]\ \frac{}{\mathrm{A} \vdash \mathtt{c} : \mathtt{t}_c}$$

Figure 1.1: Type Inference

The idea of a type judgement, $\mathrm{A} \vdash \mathtt{e} : \mathtt{t}$, is that under the assumptions, A, for the free variables, the term, `e`, has the type `t`.

Example 1.1
With the rules in Figure 1.1 we can type

$$\emptyset \vdash \lambda\mathtt{x}.\mathtt{x} : \mathtt{Int} \to \mathtt{Int}$$

and

$$\emptyset \vdash \lambda\mathtt{x}.\mathtt{x} : \mathtt{Real} \to \mathtt{Real}$$

However we are not able to infer

$$\emptyset \vdash \mathtt{if\ e\ then\ 7\ else\ 7.2} : \mathtt{Real}$$

To type `if e then 7 else 7.2` we need to coerce `Int` to `Real` in order for the two branches, `7` and `7.2`, to have the same type. This is not supported by the type system of Figure 1.1. □

The standard type inference system must be sound with respect to a semantics. Suppose that `e` is closed. Soundness of the inference system amounts to: the semantics of `e` must be a member of the semantics of the type infer for `e`.

$$[\text{coer}]\ \frac{A \vdash \mathtt{e} : \mathtt{t}_1}{A \vdash \mathtt{e} : \mathtt{t}_2} \quad \text{if } \mathtt{t}_1 \subseteq \mathtt{t}_2$$

$$[\text{ref}]\ \frac{}{\mathtt{t} \subseteq \mathtt{t}}$$

$$[\text{trans}]\ \frac{\mathtt{t}_1 \subseteq \mathtt{t}_2 \quad \mathtt{t}_2 \subseteq \mathtt{t}_3}{\mathtt{t}_1 \subseteq \mathtt{t}_3}$$

$$[\text{arrow}]\ \frac{\mathtt{t}_3 \subseteq \mathtt{t}_1 \quad \mathtt{t}_2 \subseteq \mathtt{t}_4}{\mathtt{t}_1 \rightarrow \mathtt{t}_2 \subseteq \mathtt{t}_3 \rightarrow \mathtt{t}_4}$$

Figure 1.2: The Subtyping Rules

1.1.1 Subtyping

The type inference system can be extended with a set of rules for subtyping (Figure 1.2). We have a subtyping rule (or coercion rule):

$$\frac{A \vdash \mathtt{e} : \mathtt{t}_1}{A \vdash \mathtt{e} : \mathtt{t}_2} \quad \text{if } \mathtt{t}_1 \subseteq \mathtt{t}_2$$

where the relation between the types is reflexive, transitive, and anti-monotonic in contravariant position (see Figure 1.2). Assuming that `Int`, `Real`, and `Float` are base-types we may add:

$$\frac{}{\mathtt{Int} \subseteq \mathtt{Real}} \qquad \frac{}{\mathtt{Real} \subseteq \mathtt{Float}} \tag{1.1}$$

Terms of type `Int` also have type `Real`, which also have type `Float`. The reason is that the integer `7` can be viewed as the real `7.0` and the real `8.4` can be viewed as `8.40000000`. The idea is that whenever $\mathtt{t}_1 \subseteq \mathtt{t}_2$, then all terms of type $\mathtt{t}_1$ must also have the type $\mathtt{t}_2$.

Example 1.2
With the rules in Figure 1.1, Figure 1.2, and the rules (1.1) we can type

$$\emptyset \vdash \mathtt{if\ e\ then\ 7\ else\ 7.2} : \mathtt{Real}$$

so the problem in Example 1.1 is solved by the introduction of subtyping.
□

1.2 Annotated Type Systems

The type of the term tells something about the structure of the term, i.e. it is an integer or is is function. The type does not tell anything about the evaluation properties of the term. We will now attach to the types some kind of program analysis information, which can be strictness information, totality information, binding time information, etc. The general form of an annotated type is

$$\begin{array}{rcl} \mathtt{t} & ::= & \mathtt{B}^{s_1} \mid \mathtt{t}^{s_2} \mid \mathtt{ut}^{s_3} \mid \mathtt{t} \rightarrow^{s_4} \mathtt{t} \\ \mathtt{ut} & ::= & \mathtt{B} \mid \mathtt{ut}_1 \rightarrow \mathtt{ut}_2 \end{array}$$

where the annotations s_1, s_2, s_3, and s_4 belongs to four sets (maybe equal). Given an annotated type $\mathtt{t}$ (often just called type) we can speak about the *shape* or *underlying type* of the type $\mathtt{t}$ as the type obtained by removing all the annotations from the type; we will write $\varepsilon(\mathtt{t})$ for the underlying type of $\mathtt{t}$.

The annotations can be put on the base-types, on subtypes, or on type-constructors. A term with the annotated type $\mathtt{B}^{s_1}$ will have the type $\mathtt{B}$ and the property s_1. A term of the annotated type $\mathtt{t}^{s_2}$ will be a term of the annotated type $\mathtt{t}$ and it will furthermore have the property s_2. Another way to annotate subtypes is, $\mathtt{ut}^{s_3}$, where the subtype is only telling that this term is of the underlying type $\mathtt{ut}$ and has the property s_3. Functions with the annotated type $\mathtt{t}_1 \rightarrow^{s_4} \mathtt{t}_2$ will map terms of type $\mathtt{t}_1$ to terms of type $\mathtt{t}_2$ while exhibiting the property s_4.

For each of these possibilities we will look at some example analyses:

- In Section 1.2.1 the annotations are only allowed on the base-types.
- In Section 1.2.2 the annotations are on the subtypes. We are looking at three examples: the usage analysis by Wadler [Wad91],the strictness analysis by Kuo and Mishra [KM89] and the strictness analysis of Benton and Jensen [Ben93, Jen92a].
- In Section 1.2.3 the annotations are on the type-constructors only. We look at two examples: an usage analysis by Wright [Wri91] and a control flow analysis by Tang [Tan94].
- In Section 1.2.4 the annotations are allowed in more places. The binding time analysis [NN92, SNN92] allows to annotate base-types

and type-constructors.

The analyses considered are variation of the following analyses:

Strictness analysis In strictness analysis we want to decide whether a function is strict. A function is strict whenever the application of the function to a looping argument results in a looping computation.

Binding time analysis In binding time analysis it is the distinction between data available at compile-time and run-time that has to be determined.

Usage analysis In usage analysis we want to know how many times an expression is used. Whenever an expression is used zero times the compiler need not generate code for the expression at all. An expression that is used exactly one time can be garbage collected as soon as it is used.

Control flow analysis In control flow analysis we are interested in finding the functions possible called during the evaluation of an expression. A function is an lambda-abstraction with a unique name attached.

1.2.1 Annotating Base-types

In the first analysis we are only annotating the base-types with some kind of information. The annotated types will be

$$\mathtt{t} \ ::= \ \mathtt{B}^{s_1} \mid \mathtt{t} \rightarrow \mathtt{t}$$

Simple Strictness Analysis

An example of the set of annotions s_1 is

$$s_1 ::= \bot \mid \top$$

This analysis is a subset of the strictness analysis by Kuo and Mishra [KM89], Jensen [Jen91, Jen92b, Jen92a], and Benton [Ben93] in that it is only annotating the base-types. The idea is that a term with the type $\mathtt{B}^{\bot}$ is a term with the underlying type $\varepsilon(\mathtt{B}^{\bot}) = \mathtt{B}$ that does not evaluate to a HNF (Head Normal Form). This is opposed to a term with type $\mathtt{B}^{\top}$ which still has the underlying type $\varepsilon(\mathtt{B}^\top) = \mathtt{B}$ but *may* evaluate to a HNF but we really do not know. A term with type $\mathtt{t}_1 \to \mathtt{t}_2$ has the underlying type $\varepsilon(\mathtt{t}_1) \to \varepsilon(\mathtt{t}_2)$ and will when applied to a term of type $\mathtt{t}_1$ yield a term of type $\mathtt{t}_2$. The inference system is as in Figure 1.1 except for the rules for condition:

$$\frac{\mathrm{A} \vdash \mathtt{e}_1 : \mathtt{Bool}^\perp \quad \mathrm{A} \vdash \mathtt{e}_2 : \mathtt{t} \quad \mathrm{A} \vdash \mathtt{e}_3 : \mathtt{t}}{\mathrm{A} \vdash \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 : \mathtt{t}} \tag{1.2}$$

$$\frac{\mathrm{A} \vdash \mathtt{e}_1 : \mathtt{Bool}^\top \quad \mathrm{A} \vdash \mathtt{e}_2 : \mathtt{t} \quad \mathrm{A} \vdash \mathtt{e}_3 : \mathtt{t}}{\mathrm{A} \vdash \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 : \mathtt{t}} \tag{1.3}$$

For the constants we have a number of rules one for each possible type that the constant can have:

$$\begin{array}{rcl}
\mathrm{A} \vdash \mathtt{7} & : & \mathtt{Int}^\top \\
\mathrm{A} \vdash \mathtt{+} & : & \mathtt{Int}^\perp \to \mathtt{Int}^\top \to \mathtt{Int}^\perp \\
\mathrm{A} \vdash \mathtt{+} & : & \mathtt{Int}^\top \to \mathtt{Int}^\perp \to \mathtt{Int}^\perp \\
\mathrm{A} \vdash \mathtt{+} & : & \mathtt{Int}^\top \to \mathtt{Int}^\top \to \mathtt{Int}^\top
\end{array}$$

The coercion rules are as in Figure 1.2 together with:

$$\frac{}{\mathtt{B}^\perp \subseteq \mathtt{B}^\top} \tag{1.4}$$

expressing that terms of type B with no HNF are included in all term of type B.

Example 1.3

Using the rules in Figure 1.1 excluding the [if]-rule, the rules in Figure 1.2, the rules (1.2), (1.3) and (1.4) we can infer

$$\emptyset \vdash \lambda\mathtt{x.x} : \mathtt{Int}^\perp \to \mathtt{Int}^\perp$$

saying that the application of the identity function to a looping argument will loop and

$$\emptyset \vdash \lambda\mathtt{x.x} : \mathtt{Int}^\top \to \mathtt{Int}^\top$$

saying that the identity function will map any argument to any thing. However we are not able to infer

$$\emptyset \vdash \mathtt{if}\ (\mathtt{fix}\ \lambda\mathtt{x.x})\ \mathtt{then}\ \mathtt{7}\ \mathtt{else}\ \mathtt{8} : \mathtt{Int}^\perp$$

The reason that we cannot infer the above is that the annotated type of the conditional is the type of the two branches. The type of the branches is not affected by the fact that the test does not evaluate to a HNF. We need to make the first rule for conditional (1.2) more powerful. □

1.2.2 Annotating Subtypes

There is two different way of annotating subtypes:

- Annotate a subtype where the subtype is already an annotated type.
- Annotate a subtype where the subtype is not annotated, i.e. the subtype is a standard type.

Annotating Already annotated Subtypes

First consider the annotated types, where the subtypes are already annotated types:

$$\mathtt{t} \ ::= \ \mathtt{B}^{s_1} \mid \mathtt{t}^{s_2} \mid \mathtt{t} \rightarrow \mathtt{t}$$

The analysis in Wadler [Wad91] is one example of this kind of annotated type system. The annotates are:

$$\begin{aligned} s_1 &::= \mathit{0} \mid \mathit{1} \\ s_2 &::= \mathit{0} \mid \mathit{1} \end{aligned}$$

A linear type is a type of the form $(\mathtt{t})^{\mathit{0}}$ and a non-linear type is a type of the form $(\mathtt{t})^{\mathit{1}}$. A term of a linear type is used exactly once and a term of a non-linear type is used zero, once, or many times. The structure of the inference system does not match the ones described here: the inference system is much in the style of linear logic where the assumption list is manipulated very carefully.

Annotating Subtypes at the Top-level

We now consider the case where the subtypes are only annotated at the "top level". The annotated types are:

$$\mathtt{t} \ ::= \ \mathtt{B}^{s_1} \mid \mathtt{ut}^{s_3} \mid \mathtt{t} \rightarrow \mathtt{t}$$

$$\mathtt{ut} ::= \mathtt{B} \mid \mathtt{ut} \rightarrow \mathtt{ut}$$

where $\mathtt{ut}$ is a underlying type. As an example we will use the annotations:

$$\begin{aligned} s_1 &::= \perp \mid \top \\ s_3 &::= \perp \mid \top \end{aligned}$$

Again a term with the type $\mathtt{B}^\perp$ is a term with the underlying type $\varepsilon(\mathtt{B}^\perp) = \mathtt{B}$ that does not evaluate to a HNF. This is opposed to a term with type $\mathtt{B}^\top$ which still has the underlying type $\varepsilon(\mathtt{B}^\top) = \mathtt{B}$ but *may* evaluate to a HNF but we really do not know. A term with type $\mathtt{t}_1 \rightarrow \mathtt{t}_2$ has the underlying type $\varepsilon(\mathtt{t}_1) \rightarrow \varepsilon(\mathtt{t}_2)$ and will when applied to a term of type $\mathtt{t}_1$ yield a term of type $\mathtt{t}_2$. Finally a term with the type $\mathtt{ut}^\perp$ has the underlying type $\mathtt{ut}$ and does not evaluate to a HNF whereas a term with type $\mathtt{ut}^\top$ has the underlying type $\mathtt{ut}$ and nothing is known about its evaluation.

We can improve the first rule for condition (1.2):

$$\frac{\mathrm{A} \vdash \mathtt{e}_1 : \mathtt{Bool}^\perp \quad \mathrm{A} \vdash \mathtt{e}_2 : \mathtt{t} \quad \mathrm{A} \vdash \mathtt{e}_3 : \mathtt{t}}{\mathrm{A} \vdash \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 : \varepsilon(\mathtt{t})^\perp} \tag{1.5}$$

expressing that the conditional does not have a HNF whenever the test does not have a HNF.

Example 1.4
With this new rule (1.5) we can infer

$$\mathrm{A} \vdash \mathtt{if\ (fix\ }\lambda\mathtt{x.x)\ then\ 7\ else\ 8} : \mathtt{Int}^\perp$$

which is not possible using the rule (1.2). □

For the coercion-part we take:

$$\frac{}{\varepsilon(\mathtt{t})^\perp \subseteq \mathtt{t}} \tag{1.6}$$

$$\frac{}{\mathtt{ut}_1{}^\top \rightarrow \mathtt{ut}_2{}^\perp \subseteq (\mathtt{ut}_1 \rightarrow \mathtt{ut}_2)^\perp} \tag{1.7}$$

$$\frac{}{\mathtt{t} \subseteq \varepsilon(\mathtt{t})^\top} \tag{1.8}$$

$$\frac{}{(\mathtt{ut}_1 \rightarrow \mathtt{ut}_2)^\top \subseteq \mathtt{ut}_1{}^\top \rightarrow \mathtt{ut}_2{}^\top} \tag{1.9}$$

The rules (1.6) and (1.8) has (1.4) as a special case: the rule (1.6) expresses that a $\perp$-annotated type is less than any annotated type with the same underlying type, whereas (1.8) expresses that a $\top$-annotated type is greater than any annotated type with the same underlying type. For function types we can do even better: all functions mapping any term to a non-terminating term is included in the functions without a HNF (1.7) and all functions are included in the functions that maps anything to anything (1.9).

Example 1.5
Consider the set of rules in Figure 1.1 excluding the [if]-rule, the rules in Figure 1.2, and the rules (1.5), (1.3), (1.6), (1.7), (1.8), and (1.9).

Consider the term $\mathtt{e}$, which exchanges the arguments to $\mathtt{f}$, due to Kuo and Mishra [KM89]:

$$\begin{aligned}
\mathtt{e} &= \lambda\mathtt{f}.\lambda\mathtt{x}.\lambda\mathtt{y}.\lambda\mathtt{z}.\mathtt{if\ (=\ 0\ z)\ then\ +\ x\ y\ else\ f\ y\ x\ (-\ z\ 1)} \\
\mathtt{t}_1 &= \mathtt{Int}^{\perp} \rightarrow \mathtt{Int}^{\top} \rightarrow \mathtt{Int}^{\top} \rightarrow \mathtt{Int}^{\perp} \\
\mathtt{t}_2 &= \mathtt{Int}^{\top} \rightarrow \mathtt{Int}^{\perp} \rightarrow \mathtt{Int}^{\top} \rightarrow \mathtt{Int}^{\perp}
\end{aligned}$$

We would like to infer the type $\mathtt{t}_1$ or the type $\mathtt{t}_2$ for $\mathtt{fix\ e}$. This is not possible since we are only able to infer the following:

$$\emptyset \vdash \mathtt{e} : \mathtt{t}_2 \rightarrow \mathtt{t}_1$$

and

$$\emptyset \vdash \mathtt{e} : \mathtt{t}_1 \rightarrow \mathtt{t}_2$$

We will need conjunction in order to infer $\emptyset \vdash \mathtt{fix\ e} : \mathtt{t}_1$. □

The two ways of annotating subtypes are not equal expressive when we do not have conjunction. The reason is that $(\mathtt{B}^{\perp} \rightarrow \mathtt{B}^{\perp})^{\perp}$ can be viewed as the conjunction of $\mathtt{B}^{\perp} \rightarrow \mathtt{B}^{\perp}$ and $(\mathtt{B} \rightarrow \mathtt{B})^{\perp}$.

We will now examine two of the analyses from the literature, that are annotating subtypes at the top level:

- The strictness analysis by Kuo and Mishra [KM89], which is much like the analysis above but only allows to compare *matching* types.
- The strictness analysis with conjunctions by Benton [Ben93] and Jensen [Jen91, Jen92b, Jen92a], which is introducing conjunctions of annotated types.

Strictness Analysis

The strictness analysis of Kuo and Mishra [KM89] is for the un-typed λ-calculus. But their algorithm for inferring strictness types assumes the terms to be well-typed (i.e. the term has a Milner [Mil78] type). Here we will rewrite the analysis for a typed language. The analysis is as the one above, but they do not allow types which do not *match* to be related. Two types matches if they have the same underlying type and further they must have an annotation in the same place. The type types $\mathtt{ut}_1 \rightarrow^{s} \mathtt{ut}_2$ and ${\mathtt{ut}_1}^{s} \rightarrow {\mathtt{ut}_2}^{s'}$ do not match but the two types $\mathtt{ut}_1 \rightarrow^{s'} \mathtt{ut}_2$ and $\mathtt{ut}_1 \rightarrow^{s''} \mathtt{ut}_2$ do indeed match. Therefore we will have to exclude the rules (1.6), (1.7), (1.8), and (1.9) and include the following rule instead:

$$\frac{\forall 1 \leq i \leq n : \mathtt{t}_i \text{ matches } \mathtt{t}'_i}{\mathtt{t}_1 \rightarrow \ldots \rightarrow \mathtt{t}_n \rightarrow \mathtt{B}^{s_3} \subseteq \mathtt{t}'_1 \rightarrow \ldots \rightarrow \mathtt{t}'_n \rightarrow \mathtt{B}^{\top}} \quad (1.10)$$

Example 1.6
Using the rules in Figure 1.1 excluding the [if]-rule, the rules in Figure 1.2, the rules (1.2), (1.3), and (1.10) we can infer that `twice`:

$$\mathtt{twice} = \lambda \mathtt{f}.\lambda \mathtt{x}.\mathtt{f}\ (\mathtt{f}\ \mathtt{x})$$

has the annotated type

$$({\mathtt{ut}_1}^{\top} \rightarrow {\mathtt{ut}_2}^{\perp}) \rightarrow {\mathtt{ut}_1}^{\top} \rightarrow {\mathtt{ut}_2}^{\perp}$$

for all choices of $\mathtt{ut}_1$ and $\mathtt{ut}_2$ (not only for base-types). □

The strictness analysis of Leung and Mishra [LM91] is much in the line of the strictness analysis of Kuo and Mishra [KM89]. The main difference is that Leung and Mishra are not only distinguishing between terms that definitely has no HNF and all terms but also can tell if a term surely has no NF (Normal Form), i.e. the annotations are $\perp$, $\top$, and Ω. The meaning of the $\perp$ and $\top$ annotated types are as in Kuo and Mishra [KM89] and a term with a type annotated with Ω does not have a NF. Leung and Mishra also includes coercion rules that allows to compare strictness types which does not match.

Strictness Analysis with Conjunction

The strictness analyses of Benton [Ben93] and Jensen [Jen91, Jen92b, Jen92a] are for the simply typed λ-calculus opposed to the two strictness analysis by Kuo and Mishra [KM89] and Leung and Mishra [LM91]. The strictness types are:

$$
\begin{array}{rcl}
\mathtt{t} & ::= & \mathtt{B}^{s_1} \mid \mathtt{ut}^{s_3} \mid \mathtt{t} \rightarrow \mathtt{t} \mid \mathtt{t} \wedge \mathtt{t} \\
\mathtt{ut} & ::= & \mathtt{B} \mid \mathtt{ut} \rightarrow \mathtt{ut} \\
s_1 & ::= & \bot \mid \top \\
s_3 & ::= & \bot \mid \top
\end{array}
$$

where the bases types include $\mathtt{Int}$.[1]

Since a term only has one underlying type we must require that the two strictness types $\mathtt{t}_1$ and $\mathtt{t}_2$ must have the same underlying type in order to construct a conjunction of them. We do this by defining a well-formedness relation, $\vdash^W$, on the strictness types:

$$
\frac{}{\vdash^W \mathtt{B}^\top} \qquad \frac{}{\vdash^W \mathtt{B}^\bot}
$$

$$
\frac{}{\vdash^W \mathtt{ut}^\top} \qquad \frac{}{\vdash^W \mathtt{ut}^\bot}
$$

$$
\frac{\vdash^W \mathtt{t}_1 \quad \vdash^W \mathtt{t}_2}{\vdash^W \mathtt{t}_1 \rightarrow \mathtt{t}_2} \qquad \frac{\vdash^W \mathtt{t}_1 \quad \vdash^W \mathtt{t}_2}{\vdash^W \mathtt{t}_1 \wedge \mathtt{t}_2} \quad \text{if } \varepsilon(\mathtt{t}_1) = \varepsilon(\mathtt{t}_2)
$$

A term of the strictness type $\mathtt{ut}^\bot$ is a term of type $\mathtt{ut}$ without a HNF, whereas a term of strictness type $\mathtt{ut}^\top$ is a term of type $\mathtt{ut}$, but we know nothing about the evaluation of the term. The coercion rules are as for Leung and Mishra [LM91] including the following for conjunctions:

$$
\frac{}{\mathtt{t}_1 \wedge \mathtt{t}_2 \subseteq \mathtt{t}_1} \qquad \frac{}{\mathtt{t}_1 \wedge \mathtt{t}_2 \subseteq \mathtt{t}_2} \tag{1.11}
$$

$$
\frac{\mathtt{t}_3 \subseteq \mathtt{t}_1 \quad \mathtt{t}_3 \subseteq \mathtt{t}_2}{\mathtt{t}_3 \subseteq \mathtt{t}_1 \wedge \mathtt{t}_2} \tag{1.12}
$$

$$
\frac{}{(\mathtt{t}_1 \rightarrow \mathtt{t}_2) \wedge (\mathtt{t}_1 \rightarrow \mathtt{t}_3) \subseteq \mathtt{t}_1 \rightarrow (\mathtt{t}_2 \wedge \mathtt{t}_3)} \tag{1.13}
$$

The rules (1.11) express that whenever a term has the strictness type $\mathtt{t}_1 \wedge \mathtt{t}_2$ it also has the strictness type $\mathtt{t}_1$ and the strictness type $\mathtt{t}_2$, respectively. Whenever the terms of strictness type $\mathtt{t}_3$ is a subset of the

[1] We are writing $\bot$ for **f** and $\top$ for **t**.

terms of type $\mathtt{t}_1$ and a subset of the terms of type $\mathtt{t}_2$, then the terms of strictness type $\mathtt{t}_3$ is a subset of the terms of strictness type $\mathtt{t}_1 \wedge \mathtt{t}_2$, this fact is expressed by the rule (1.12). The rule (1.13) says that the functions that map terms of type $\mathtt{t}_1$ to terms of strictness type $\mathtt{t}_2$ and also map terms of type $\mathtt{t}_1$ to terms of type $\mathtt{t}_3$ must be a subset of the functions that map terms of type $\mathtt{t}_1$ to terms of strictness type $\mathtt{t}_2 \wedge \mathtt{t}_3$.

The new rule for conjunction in the analysis is:

$$\frac{\mathrm{A} \vdash \mathtt{e} : \mathtt{t}_1 \quad \mathrm{A} \vdash \mathtt{e} : \mathtt{t}_2}{\mathrm{A} \vdash \mathtt{e} : \mathtt{t}_1 \wedge \mathtt{t}_2} \tag{1.14}$$

Example 1.7
Using the rules in Figure 1.1 excluding the [if]-rule, the rules in Figure 1.2, the rules (1.5), (1.3), (1.6), (1.7), (1.8), (1.9), (1.11), (1.12), (1.13), and (1.14) we can infer

$$\emptyset \vdash \mathtt{fix}\ \mathtt{e} : \mathtt{t}_1$$
$$\emptyset \vdash \mathtt{fix}\ \mathtt{e} : \mathtt{t}_2$$

for the term $\mathtt{e}$ from Example 1.10; the reason is that we can make use of conjunction and infer

$$\emptyset \vdash \mathtt{e} : (\mathtt{t}_1 \wedge \mathtt{t}_2) \rightarrow (\mathtt{t}_1 \wedge \mathtt{t}_2)$$

So this strictness analysis solves the limitations of the strictness analysis by Kuo and Mishra [KM89].

□

1.2.3 Annotating Type-constructors

The third kind of annotated types will annotate the type-constructors:

$$\mathtt{t} ::= \mathtt{B} \mid \mathtt{t} \rightarrow^{s_4} \mathtt{t}$$

A term of type $\mathtt{t}_1 \rightarrow^{s_4} \mathtt{t}_2$ is a function from $\mathtt{t}_1$ to $\mathtt{t}_2$ with the behaviour s_4. The coercions rules have to take into account the new structure of the function types. The new rule for function arrow is:

$$\frac{\mathtt{t}_3 \subseteq \mathtt{t}_1 \quad \mathtt{t}_2 \subseteq \mathtt{t}_4}{\mathtt{t}_1 \rightarrow^{s_4{}'} \mathtt{t}_2 \subseteq \mathtt{t}_3 \rightarrow^{s_4{}''} \mathtt{t}_4} \quad \text{if } s_4{}' \preceq s_4{}'' \tag{1.15}$$

The coercions are inherited from the reflexive and transitive relation, $\preceq$, on the annotations. The rule is still anti-monotonic in contra-variant position.

The rules involving function types will look a bit different — we have to take the annotations on the function-arrow into account:

$$[\text{abs}'] \ \frac{\mathrm{A}, \mathtt{x} : \mathtt{t}_1 \vdash \mathtt{e} : \mathtt{t}_2}{\mathrm{A} \vdash \lambda\mathtt{x.e} : \mathtt{t}_1 \rightarrow^{s_4} \mathtt{t}_2}$$

$$[\text{app}'] \ \frac{\mathrm{A} \vdash \mathtt{e}_0 : \mathtt{t}_1 \rightarrow^{s_4} \mathtt{t}_2 \quad \mathrm{A} \vdash \mathtt{e}_1 : \mathtt{t}_1}{\mathrm{A} \vdash \mathtt{e}_0\ \mathtt{e}_1 : \mathtt{t}_2}$$

This analysis will not give much useful information, in fact it does not give more program analysis information than the standard type system. In order to get more information we must make a better guess on what annotation to put on the function arrow. We this by allowing one more component in the type judgement: the new judgements have the form

$$\mathrm{A} \vdash \mathtt{e} : \mathtt{t} : b$$

and says that under the assumptions A, for the free variables, the term $\mathtt{e}$ has the type $\mathtt{t}$ and the behaviour b. The assumption may not only include type information but also some sort of behaviour information. In the abstraction rule we will use this extra information to annotate the function arrow:

$$\frac{\mathrm{A}, \mathtt{x} : \mathtt{t}_1 \vdash \mathtt{e} : \mathtt{t}_2 : b}{\mathrm{A} \vdash \lambda\mathtt{x.e} : \mathtt{t}_1 \rightarrow^{s_4} \mathtt{t}_2 : b'}$$

where s_4 and b' depend on b. The rule for application is:

$$\frac{\mathrm{A} \vdash \mathtt{e}_0 : \mathtt{t}_1 \rightarrow^{s_4} \mathtt{t}_2 : b \quad \mathrm{A} \vdash \mathtt{e}_1 : \mathtt{t}_1 : b'}{\mathrm{A} \vdash \mathtt{e}_0\ \mathtt{e}_1 : \mathtt{t}_2 : b''}$$

where the behaviour b'' depends not only on b and b' but also on s_4. Here we can see the connection between the behaviours and the annotations.

We will now take a look at the usage analysis of Wright [Wri91] and the control flow analysis of Tang [Tan94].

Usage Analysis

One example of this class of annotated type systems is the usage analysis of Wright [Wri91, Amt93a, Amt94, Amt93b, Hen94]. The annotated types and the annotations are:

$$\begin{array}{rcl} \mathtt{t} & ::= & \mathtt{B} \mid \mathtt{t} \rightarrow^{s_4} \mathtt{t} \\ s_4 & ::= & 0 \mid 1 \mid \alpha \mid \neg s_4 \mid s_4 \wedge s_4 \mid s_4 \vee s_4 \end{array}$$

where α are *arrow variables* and the base-types includes *type variables*. The annotated type $\mathtt{t}_1 \rightarrow^0 \mathtt{t}_2$ denotes all constant functions from $\mathtt{t}_1$ to $\mathtt{t}_2$, i.e. the argument is used zero times. The type $\mathtt{t}_1 \rightarrow^1 \mathtt{t}_2$ denotes all strict functions from $\mathtt{t}_1$ to $\mathtt{t}_2$, i.e. the argument is used once or more. Note here that not all terms can be given a type. In all the other annotated type system there is a type denoting all terms with a given underlying type; there is no such annotated type here. Hence the term $\mathtt{if\ e\ then\ }\lambda\mathtt{x.x\ else\ }\lambda\mathtt{x.7}$ cannot be given a usage type in this system, since the two branches must have the same type.

The relation on annotations is defined by substitution on the annotations:

$$s_4{}' \preceq s_4{}'' \;\Leftrightarrow\; \exists S.S(s_4{}') = s_4{}'' \tag{1.16}$$

where S is a substitution from arrow variables to annotations. The coercions are inherited from the notion of *renaming instances* on the type variables:

$$\frac{\exists R \;.\; R\ (\mathtt{B}) = \mathtt{B}'}{\mathtt{B} \subseteq \mathtt{B}'} \tag{1.17}$$

where $\mathtt{B}$ and $\mathtt{B}'$ are base-types or type-variables and R is a substitution (i.e. a renaming of type variables) from type variables to type variables. The rule for function arrow is:

$$\frac{\mathtt{t}_1 \subseteq \mathtt{t}_1' \quad \mathtt{t}_2 \subseteq \mathtt{t}_2'}{\mathtt{t}_1 \rightarrow^{s_4{}'} \mathtt{t}_2 \subseteq \mathtt{t}_1' \rightarrow^{s_4{}''} \mathtt{t}_2'} \quad \text{if } s_4{}' \preceq s_4{}'' \tag{1.18}$$

Note that the function type constructor *is not* anti-monotonic in the first argument. The reason is that the relation is inherited from renaming of type variables.

Let the usage list, V, be a list of pairs of variables and annotations

$$\mathrm{V} \;::=\; (\mathtt{x}, s_4) : \mathrm{V} \mid \mathrm{nil}$$

We will use the usage list as the behaviours. Let V^0 be the usage list that assigns 0 to all the variables, i.e meaning that no variables are used. The rule for variables is:

$$\frac{}{\mathrm{A} \vdash \mathtt{x} : \mathtt{t} : (\mathtt{x},\ 1){:}\mathrm{V}^0} \quad \text{if } \mathtt{x} : \mathtt{t} \in \mathrm{A} \tag{1.19}$$

where the usage list, $(\mathtt{x},\ 1){:}\mathrm{V}^0$, says that all the variables except `x` is not used. The only place where the subtyping rule is allowed is after the rule for variables. Therefore we built it into the rule for variables (1.19):

$$\frac{}{\mathrm{A} \vdash \mathtt{x} : \mathtt{t}_2 : (\mathtt{x},\ 1){:}\mathrm{V}^0} \quad \text{if } \mathtt{x} : \mathtt{t}_1 \in \mathrm{A} \wedge \mathtt{t}_1 \subseteq \mathtt{t}_2 \tag{1.20}$$

The abstraction rule is:

$$\frac{\mathrm{A},\ \mathtt{x} : \mathtt{t}_1 \vdash \mathtt{e} : \mathtt{t}_2 : \mathrm{V}}{\mathrm{A} \vdash \lambda\mathtt{x.e} : \mathtt{t}_1 \to^{s_4} \mathtt{t}_2 : (\mathtt{x},\ 0){:}\mathrm{V}_{\mathtt{x}}} \quad \text{if } (\mathtt{x},\ s_4) \in \mathrm{V} \tag{1.21}$$

We record that the variable `x` is not used since `x` is no longer free in the term and the usage of `x` is recorded by the annotation on the function-arrow. The application rule is

$$\frac{\mathrm{A} \vdash \mathtt{e}_0 : \mathtt{t}_1 \to^{s_4} \mathtt{t}_2 : \mathrm{V}_0 \quad \mathrm{A} \vdash \mathtt{e}_1 : \mathtt{t}_1 : \mathrm{V}_1}{\mathrm{A} \vdash \mathtt{e}_0\ \mathtt{e}_1 : \mathtt{t}_2 : \mathrm{V}_0 \sqcup (s_4 \sqcap \mathrm{V}_1)} \tag{1.22}$$

where $\sqcap$ and $\sqcup$ is defined pointwise for each variable as the greatest lower bound and least upper bound of the usages defined by $\preceq$. The intuition of the new usage is that in $\mathtt{e}_0\ \mathtt{e}_1$ we are going to use the variables as in $\mathtt{e}_0$, i.e. as recorded by V_0, and the variables in $\mathtt{e}_1$ are used whenever $\mathtt{e}_1$ is used. The annotation s_4 on the function-arrow expresses the usage of $\mathtt{e}_1$. Hence the usage of the variables in $\mathtt{e}_1$ is expressed by $s_4 \sqcap \mathrm{V}_1$.

The presentation here has the conditional as a construct whereas in Wright [Wri91] it is a constant. There is only one rule for condition:

$$\frac{\begin{array}{c}\mathrm{A} \vdash \mathtt{e}_1 : \mathtt{Bool} : \mathrm{V}_1 \\ \mathrm{A} \vdash \mathtt{e}_2 : \mathtt{t} : \mathrm{V}_2 \\ \mathrm{A} \vdash \mathtt{e}_3 : \mathtt{t} : \mathrm{V}_3\end{array}}{\mathrm{A} \vdash \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 : \mathtt{t} : \mathrm{V}_1 \sqcap (\mathrm{V}_2 \sqcup \mathrm{V}_3)} \tag{1.23}$$

The idea behind the new usage is that for the conditional we are going to use the variables as in $\mathtt{e}_1$ and if a variable is used by either $\mathtt{e}_2$ or $\mathtt{e}_3$ then we might use it in the conditional.

We have a set of rules for the constants one for each type that that the constant can have:

$$\frac{}{\mathrm{A} \vdash \mathtt{c} : \mathtt{t}_{\mathtt{c}} : \mathrm{V}^0} \tag{1.24}$$

where the usage for all the variables are zero. Among others we have

$$\begin{aligned} \mathtt{t}_7 &= \mathtt{Int} \\ \mathtt{t}_{\mathtt{true}} &= \mathtt{Bool} \\ \mathtt{t}_+ &= \mathtt{Int} \rightarrow^1 \mathtt{Int} \rightarrow^1 \mathtt{Int} \end{aligned}$$

Example 1.8
Using the coercion rules in Figure 1.2 excluding the rule [arrow], and the rules, (1.16), (1.17), (1.18), (1.20), (1.21), (1.22), and (1.23) we can infer

$$\emptyset \vdash \lambda\mathtt{x}.\mathtt{x} : \mathtt{Int} \rightarrow^1 \mathtt{Int} : \mathrm{V}^0$$

saying that $\lambda\mathtt{x}.\mathtt{x}$ uses its argument and

$$\emptyset \vdash \lambda\mathtt{x}.7 : \mathtt{Int} \rightarrow^0 \mathtt{Int} : \mathrm{V}^0$$

saying that $\lambda\mathtt{x}.7$ does not use its argument. □

The usage analysis in [WBF93] extends the annotations to be any natural number, n, and uses $+$ as $\sqcup$ and $*$ as $\sqcap$.

Control Flow Analysis

Another example of annotating type-constructors are the effect systems [Tan94, TJ92a, TJ92b, TT94]. We will take a closer look at the control flow analysis in Tang [Tan94]. In control flow analysis we are interested in finding the functions possible called during the evaluation of an expression. An function is an lambda-abstraction with a unique name. So in the term that is going to be analysed all the lambdas has a label attached. In the analysis the annotations is sets of those labels:

$$s_4 \ ::= \ \emptyset \mid \{\mathbf{n}\} \mid s_4 \cup s_4$$

where the $\mathbf{n}$'s are such labels. The behaviours or effects of a term are the set of label that may be encountered during evaluation of the term. The type judgement $\mathrm{A} \vdash \mathtt{e} : \mathtt{t} : s_4$ says that under the assumptions A the term $\mathtt{e}$ has the type $\mathtt{t}$ and the abstractions possible encountered is s_4. The rule for variables is:

$$\frac{}{\mathrm{A} \vdash \mathtt{x} : \mathtt{t} : \emptyset} \quad \text{if } \mathtt{x} : \mathtt{t} \in \mathrm{A} \tag{1.25}$$

no functions are called when a variable is mentioned. For abstraction the rule is:

$$\frac{\mathrm{A}, \mathtt{x} : \mathtt{t}_1 \vdash \mathtt{e} : \mathtt{t}_2 : s_4}{\mathrm{A} \vdash \lambda\mathtt{x.e} : \mathtt{t}_1 \rightarrow^{s_4 \cup \{\mathbf{n}\}} \mathtt{t}_2 : \emptyset} \quad \text{if } \mathbf{n} \text{ is label of } \lambda \tag{1.26}$$

no functions are called by constructing a function but the latent effect of the function is all the program labels that are called by the body of the function including the label of the function itself. When the function is applied the latent effect of the function is part of the effect of the application together with the effects of $\mathtt{e}_0$ and $\mathtt{e}_1$:

$$\frac{\mathrm{A} \vdash \mathtt{e}_0 : \mathtt{t}_1 \rightarrow^{s_4''} \mathtt{t}_2 : s_4''' \quad \mathrm{A} \vdash \mathtt{e}_1 : \mathtt{t}_1 : s_4'}{\mathrm{A} \vdash \mathtt{e}_0\ \mathtt{e}_1 : \mathtt{t}_2 : s_4''' \cup s_4' \cup s_4''} \tag{1.27}$$

The relation of annotations is

$$s_4' \preceq s_4'' \Leftrightarrow (s_4' \text{ is a subset of } s_4'') \tag{1.28}$$

There are two different coercion rules: one that uses the subset-relation on the annotations to induce the relation on the types:

$$\frac{\mathrm{A} \vdash \mathtt{e} : \mathtt{t}_1 : s_4}{\mathrm{A} \vdash \mathtt{e} : \mathtt{t}_2 : s_4} \quad \text{if } \mathtt{t}_1 \subseteq \mathtt{t}_2 \tag{1.29}$$

and the other is sub-effecting:

$$\frac{\mathrm{A} \vdash \mathtt{e} : \mathtt{t} : s_4'}{\mathrm{A} \vdash \mathtt{e} : \mathtt{t} : s_4''} \quad \text{if } s_4' \preceq s_4'' \tag{1.30}$$

where only the effect is changed and not the type. The first coercion rule is the more powerful one, in the sense that more precise information can be gained.

Example 1.9
Using the coercion rules in Figure 1.2 excluding the rule [arrow], and the rules, (1.15), (1.25), (1.26), (1.27), and (1.28) we can infer

$$\emptyset \vdash (\lambda\mathtt{f.f}\ (\lambda\mathtt{a.a}))\ (\lambda\mathtt{g.g\ 1}) : \mathtt{Int} : \{\mathbf{n}_{\mathrm{f}}, \mathbf{n}_{\mathrm{a}}, \mathbf{n}_{\mathrm{g}}\}$$

Consider the term, $\mathtt{e}$:

$$\mathtt{e} \;=\; (\lambda\mathtt{f.+\ (f\ (}\lambda\mathtt{a.a)\ (f\ (}\lambda\mathtt{b.b))))(}\lambda\mathtt{g.g\ 1)}$$

Now we are not able to infer the following without one of the coercions rules:

$$\emptyset \vdash \texttt{e} : \texttt{Int} : \{\mathbf{n}_\mathrm{f}, \mathbf{n}_\mathrm{a}, \mathbf{n}_\mathrm{b}\ \mathbf{n}_\mathrm{g}\}$$

To see why, consider the type, $\texttt{t}_\mathrm{g}$ that must be inferred for $\lambda\texttt{g.g 1}$ which must be the same type as the type for the argument to the abstraction labelled $\mathbf{n}_\mathrm{f}$. This argument is applied to two different argument, i.e. $\lambda\texttt{a.a}$ and $\lambda\texttt{b.b}$, therefore the type $\texttt{t}_\mathrm{g}$ must be $\texttt{Int} \rightarrow^{\{\mathbf{n}_\mathrm{a},\mathbf{n}_\mathrm{b}\}} \texttt{Int}$ and hence both $\lambda\texttt{a.a}$ and $\lambda\texttt{b.b}$ must have the type $\texttt{t}_\mathrm{g}$. From the (1.25) we get

$$\texttt{a} : \texttt{Int} \vdash \texttt{a} \ : \ \texttt{Int} \ : \ \emptyset$$

and by applying the rule (1.26) we have

$$\emptyset \vdash \lambda\texttt{a.a} : \texttt{Int} \rightarrow^{\{\mathbf{n}_\mathrm{a}\}} \texttt{Int} : \emptyset$$

and no rules (other than the subtyping rule) can be applied to get the type $\texttt{t}_\mathrm{g}$. By applying the sub-typing (1.29) rule we get

$$\emptyset \vdash \lambda\texttt{a.a} : \texttt{Int} \rightarrow^{\{\mathbf{n}_\mathrm{a},\mathbf{n}_\mathrm{b}\}} \texttt{Int} : \emptyset$$

and we can construct the rest of the proof-tree straightforward. Now suppose we have the sub-effecting rule (1.30) instead. Again we will start by using the rule for variables (1.25):

$$\texttt{a} : \texttt{Int} \vdash \texttt{a} \ : \ \texttt{Int} \ : \ \emptyset$$

then we will apply the sub-effecting rule (1.30):

$$\texttt{a} : \texttt{Int} \vdash \texttt{a} \ : \ \texttt{Int} \ : \ \{\mathbf{n}_\mathrm{b}\}$$

and finally the abstraction rule (1.26) to get

$$\emptyset \vdash \lambda\texttt{a.a} : \texttt{Int} \rightarrow^{\{\mathbf{n}_\mathrm{a},\mathbf{n}_\mathrm{b}\}} \texttt{Int} : \emptyset$$

as required.

The difference between subtyping (1.29) and sub-effecting (1.30) is seen by looking at the type inferred for $\lambda\texttt{a.a}$: for subtyping we get the type $\texttt{Int} \rightarrow^{\{\mathbf{n}_\mathrm{a}\}} \texttt{Int}$ whereas for sub-effecting we get the less precise type $\texttt{Int} \rightarrow^{\{\mathbf{n}_\mathrm{a},\mathbf{n}_\mathrm{b}\}} \texttt{Int}$.

□

1.2.4 Annotating Base-types and Type-constructors

It is also possible to annotate both base-types and type constructors at the same time. An example where this is useful is binding time analysis.

Binding Time Analysis

The binding time analysis of Nielson and Nielson [NN92, SNN92] annotates both the base-types and the type constructors. The types are:

$$
\begin{aligned}
\mathtt{t} &::= \mathtt{B}^{s_1} \mid \mathtt{t} \rightarrow^{s_4} \mathtt{t} \\
s_1 &::= \mathbf{r} \mid \mathbf{c} \\
s_4 &::= \mathbf{r} \mid \mathbf{c}
\end{aligned}
$$

where the base-types include `Int` and `Bool`. The annotation **r** means run-time and the annotation **c** means compile-time. The analysis will introduce the distinction between data available at compile-time and at rune-time. Data available at compile-time can be manipulated by the compiler and whereby save execution time at run-time.

Not all types are well-formed — it is not allowed to mix run-time and compile-time annotated types except for run-time functions can be both of kind run-time and compile-time:

$$
\frac{}{\vdash^W \mathtt{B}^{\mathbf{r}} : \mathbf{r}} \qquad \frac{}{\vdash^W \mathtt{B}^{\mathbf{c}} : \mathbf{c}} \tag{1.31}
$$

$$
\frac{\vdash^W \mathtt{t}_1 : \mathbf{r} \quad \vdash^W \mathtt{t}_2 : \mathbf{r}}{\vdash^W \mathtt{t}_1 \rightarrow^{\mathbf{r}} \mathtt{t}_2 : \mathbf{r}} \tag{1.32}
$$

$$
\frac{\vdash^W \mathtt{t}_1 : \mathbf{c} \quad \vdash^W \mathtt{t}_2 : \mathbf{c}}{\vdash^W \mathtt{t}_1 \rightarrow^{\mathbf{c}} \mathtt{t}_2 : \mathbf{c}} \tag{1.33}
$$

$$
\frac{\vdash^W \mathtt{t}_1 : \mathbf{r} \quad \vdash^W \mathtt{t}_2 : \mathbf{r}}{\vdash^W \mathtt{t}_1 \rightarrow^{\mathbf{r}} \mathtt{t}_2 : \mathbf{c}} \tag{1.34}
$$

The analysis in [NN92, SNN92] is for the two level simply type λ-calculus, that means that also the terms are annotated with binding times — except the variables.

The list A of assumptions now contains both the type and kind of the variable: the list has the form $\mathtt{x}_1 : \mathtt{t}_1 : b_1, \ldots, \mathtt{x}_n : \mathtt{t}_n : b_n$ where the b_i's are either $\mathbf{r}$ or $\mathbf{c}$. In the rule for variables we take the binding time from the assumption for the variable as the overall kind of the term:

$$\frac{}{\mathrm{A} \vdash \mathtt{x} : \mathtt{t} : s_1} \quad \text{if } \mathtt{x} : \mathtt{t} : s_1 \in \mathrm{A} \wedge \vdash^W \mathtt{t} : s_1 \tag{1.35}$$

where we have to ensure that the type is well-formed. The rule for abstraction:

$$\frac{\mathrm{A}, \mathtt{x} : \mathtt{t}_1 : s_1 \vdash \mathtt{e} : \mathtt{t}_2 : s_1}{\mathrm{A} \vdash \lambda^{s_1}\mathtt{x.e} : \mathtt{t}_1 \to^{s_1} \mathtt{t}_2 : s_1} \quad \text{if } \vdash^W \mathtt{t}_1 : s_1 \tag{1.36}$$

the annotations on the type and on the term have to match with the kind. The rule for application:

$$\frac{\mathrm{A} \vdash \mathtt{e}_0 : \mathtt{t}_1 \to^{s_1} \mathtt{t}_2 : s_1 \quad \mathrm{A} \vdash \mathtt{e}_1 : \mathtt{t}_1 : s_1}{\mathrm{A} \vdash ({}^{s_1}\mathtt{e}_0\ \mathtt{e}_1) : \mathtt{t}_2 : s_1} \tag{1.37}$$

also here the annotations on the type and on the term has to match with the kind. There are two coercion rules:

$$\frac{\mathrm{A} \vdash \mathtt{e} : \mathtt{t}_1 \to^{\mathbf{r}} \mathtt{t}_2 : \mathbf{c}}{\mathrm{A} \vdash \mathtt{e} : \mathtt{t}_1 \to^{\mathbf{r}} \mathtt{t}_2 : \mathbf{r}} \tag{1.38}$$

$$\frac{\mathrm{A} \vdash \mathtt{e} : \mathtt{t}_1 \to^{\mathbf{r}} \mathtt{t}_2 : \mathbf{r}}{\mathrm{A} \vdash \mathtt{e} : \mathtt{t}_1 \to^{\mathbf{r}} \mathtt{t}_2 : \mathbf{c}} \quad \text{if } \forall(\mathtt{x}_i : \mathtt{t}_i : b_i) \in \mathrm{A} : b_i = \mathbf{c} \tag{1.39}$$

A run-time function is a piece of code which we can manipulate at compile-time, hence run-time functions can be of kind compile-time. However in oder to turn a run-time function of run-time kind into a run-time function of compile-time kind, the function is not allowed to refer to any run-time objects, i.e. none of the free variables must be of run-time kind. This is exactly that the side-condition in the rule (1.39) says.

Example 1.10
Using (1.31), (1.32), (1.33), (1.34), (1.35), (1.36), (1.37),(1.38), (1.39) we can infer

$$\emptyset \vdash \lambda^{\mathbf{r}}\mathtt{x.x} : \mathtt{Int}^{\mathbf{r}} \to^{\mathbf{r}} \mathtt{Int}^{\mathbf{r}} : \mathbf{r}$$

and

$$\emptyset \vdash \lambda^{\mathbf{c}}\mathtt{x.x} : \mathtt{Int}^{\mathbf{c}} \to^{\mathbf{c}} \mathtt{Int}^{\mathbf{c}} : \mathbf{c}$$

and using (1.39) we can infer

$$\emptyset \vdash \lambda^{\mathbf{r}}\mathtt{x.x} : \mathtt{Int}^{\mathbf{r}} \rightarrow^{\mathbf{r}} \mathtt{Int}^{\mathbf{r}} : \mathbf{c}$$

showing that a run-time function can be of compile-time kind. □

1.2.5 Summary

Section	Analysis	s_1	s_2	s_3	s_4	$\wedge$
1.2.1	Simple Strictness Analysis	•				
1.2.2	Strictness Analysis	•		•		
	Strictness Analysis with Conjunction	•		•		•
1.2.3	Usage Analysis				•	
	Control Flow Analysis				•	
1.2.4	Binding Time Analysis	•			•	

Table 1.1: Annotations in Chapter 1

In this Section we have been looking at different way of annotating types:

$$\begin{aligned} \mathtt{t} &::= \mathtt{B}^{s_1} \mid \mathtt{t}^{s_2} \mid \mathtt{ut}^{s_3} \mid \mathtt{t} \rightarrow^{s_4} \mathtt{t} \mid \mathtt{t} \wedge \mathtt{t} \\ \mathtt{ut} &::= \mathtt{B} \mid \mathtt{ut}_1 \rightarrow \mathtt{ut}_2 \end{aligned}$$

The annotations has been put on the base-types, on subtypes, which may or may not have annotations themselves, or on type-constructors. We have seen that there is a wide variety not only in the choice of annotations but also in the choice of how the annotations are attached to the types and how the types are put together. The different choices of annotations are summarised in Table 1.1.

The analyses we have seen are:

- A *simple strictness analysis* annotates only the base-types (s_1).
- The *strictness analysis* by Kuo and Mishra [KM89] where we annotate both the base-types and subtypes (s_1 and s_3).
- The *strictness analysis with conjunction* by Jensen [Jen91, Jen92b, Jen92a] and Benton [Ben93] where both base-types and subtypes are annotated (s_1 and s_3). Furthermore the analysis allows to construct conjunctions of annotated types.

- The *usage analysis* by Wright [Wri91] annotates the type constructors only (s_4).
- The *control flow analysis* of Tang [Tan94] only the type constructors are annotated (s_4).
- The *binding time analysis* by Nielson and Nielson [NN92] annotates both the base-types and the type-constructors (s_1 and s_4).

1.3 Overview of Thesis

Analysis	s_1	s_2	s_3	s_4	$\wedge$
Chapter 2	•		•		•[2]
Chapter 3	•		•		•
Chapter 4	Algorithm				
Chapter 5	•			•	
Chapter 6	Abstract Interpretation				

Table 1.2: Annotations in the Thesis

In **Chapter 2** we present a combined *strictness and totality analysis*. We are specifying the analysis as an annotated type system. The type system allows conjunctions of annotated types, but only at the top-level. The analysis is somewhat more powerful than the strictness analysis by Kuo and Mishra [KM89] due to the conjunctions and in that we also consider totality. The analysis is shown sound with respect to a natural-style operational semantics. The analysis is not immediately extendable to full conjunction.

The analysis of **Chapter 3** is also a combined strictness and totality analysis, however with "full" conjunction. Soundness of the analysis is shown with respect to a denotational semantics. The analysis is more powerful than the strictness analyses by Jensen [Jen92a] and Benton [Ben93] in that it in addition to strictness considers totality.

So far we have only specified the analyses, however in order for the analyses to be practically useful we need an algorithm for inferring the annotated types. In **Chapter 4** we construct an algorithm for the analysis of Chapter

[2] The conjunctions are only allow at the "top-level".

3 using the *lazy type* approach by Hankin and Le Métayer [HM94a]. The reason for choosing the analysis from Chapter 3 is that the approach not applicable to the analysis from Chapter 2.

In **Chapter 5** we study a binding time analysis. We take the analysis specified by Nielson and Nielson [NN92] and we construct an more efficient algorithm than the one proposed in [NN92]. The algorithm collects constraints in a structural manner as the algorithm $\mathcal{T}$ [Dam85]. Afterwards the minimal solution to the set of constraints is found.

The analysis in **Chapter 6** is specified by abstract interpretation. Hunt [Hun91] shows that projection based analyses are subsumed by PER (partial equivalence relation) based analyses using abstract interpretation. The PERs used by Hunt are strict, i.e. bottom is related to bottom. In Chapter 6 we lift this restriction by requiring the PERs to be *uniform*, in the sense that they treat all the integers equally. By allowing non-strict PERs we get the three properties on `Int`: $\{\bot, \mathbb{Z}, \mathbb{Z}_\bot\}$ corresponding to the three annotations, **b**, **n**, and $\top$ used in Chapter 2 and 3.

Chapter 7 contains concluding remarks.

Chapter 2

Strictness and Totality Analysis

Strictness analysis has proved useful in the implementation of lazy functional languages such as Miranda, Lazy ML and Haskell: when a function is strict it is safe to evaluate its argument before performing the function call. Totality analysis has not be adopted so widely: if the argument to a function is known to terminate then it is safe to evaluate it before performing the function call [Myc80].

In the literature there are several approaches to the specification of strictness analysis: abstract interpretation (e.g. [Myc81, BHA86]), projection analysis (e.g. [WH87]) and inference based methods (e.g. [Ben93, Jen91, Jen92b, KM89, Wri91]). Totality analysis has received much less attention and has primarily been specified using abstract interpretation [Myc81, Abr90]. Totality analysis can be regarded as an approximation to time complexity analysis; most literature performing such developments consider eager languages but [San90] considers lazy languages.

In this Chapter we present an inference system for performing strictness *and* totality analysis. Three annotations on underlying types, `ut`, are introduced:

- $\mathtt{ut}^{\mathbf{b}}$: the value has type `ut` and is definitely $\perp$,
- $\mathtt{ut}^{\mathbf{n}}$: the value has type `ut` and is definitely *not* $\perp$, and
- $\mathtt{ut}^{\top}$: the value has type `ut` and it can be any value.

Annotated types can be constructed using the function type constructor and (top-level) conjunction. As an example a function may have the annotated type $(\mathtt{Int}^{\mathbf{n}} \rightarrow \mathtt{Int}^{\mathbf{n}}) \wedge (\mathtt{Int}^{\mathbf{b}} \rightarrow \mathtt{Int}^{\mathbf{b}})$ which means that given a terminating argument the function will definitely terminate and given a non-terminating argument it will definitely not terminate. Thus we capture the strictness as well as the totality of the function. Strictness and totality information can also be combined as in

$$\begin{aligned}&(\mathtt{Int}^{\mathbf{n}} \to \mathtt{Int}^{\mathbf{n}} \to \mathtt{Int}^{\mathbf{n}}) \wedge (\mathtt{Int}^{\mathbf{b}} \to \mathtt{Int}^{\mathbf{n}} \to \mathtt{Int}^{\mathbf{n}})\\ &\quad \wedge(\mathtt{Int}^{\mathbf{n}} \to \mathtt{Int}^{\mathbf{b}} \to \mathtt{Int}^{\mathbf{n}}) \wedge (\mathtt{Int}^{\mathbf{b}} \to \mathtt{Int}^{\mathbf{b}} \to \mathtt{Int}^{\mathbf{b}})\end{aligned}$$

which will be the annotated type of McCarthy's ambiguity operator: if one of the two argument terminates so does the function call but if both argument diverges so does the function call.

The inference based approach allows to combine the two analyses. Mycroft [Myc81] presents both analyses using abstract interpretation but the semantic foundations are different: the strictness analysis is based on downward closed sets and the totality analysis on upward closed sets. We believe that the two analyses could be combined using the convex power-domains of [MN83] but this will be untractable for two reasons. One is that the mathematical foundations will be rather complicated and extensions to richer languages would not be easy. Another reason is that implementations based on abstract interpretation often are rather inefficient due to the local computation of fixpoints and we would like to explore the use of other approaches that seem to offer better performance.

The semantic foundations of our work is based on natural style operational semantics [Des86, Plo81]. We employ a lazy semantics so terms are evaluated to weak head normal form (WHNF). This means we capture the semantics of "real-life" lazy functional languages in contrast to most other papers on strictness analysis like [BHA86] where terms are evaluated to head normal forms. Since we are based on operational semantics fixpoint induction is not available for free and in the soundness proof for the analysis we shall use the trick of annotating the fixpoint operator with the number of unfoldings allowed.

2.0.1 Motivating Example

Example 2.1
Consider the CBN program

```
let  f = λg.λx.g (x)
     a = ...
in   f (λx.x) a
```

A naive CBV version of it may be

```
let  f = λg.λx.(g ()) (x)
     a = λ()....
in   f (λ().λx.x ()) a
```

However, an optimised CBV version is:

```
let  f = λg.λx.g (x)
     a = ...
in   f (λx.x) a
```

since `f` is strict in its first argument we need not thunkify the first argument to `f` and since the first argument to `f` is always a strict function we need not thunkify the second argument to `f`. This can be seen from the strictness type of `f`:

$$((\texttt{Int} \to \texttt{Int})^{\mathbf{b}} \to \texttt{Int}^{\mathbf{b}} \to \texttt{Int}^{\mathbf{b}}) \wedge ((\texttt{Int}^{\mathbf{b}} \to \texttt{Int}^{\mathbf{b}}) \to \texttt{Int}^{\mathbf{b}} \to \texttt{Int}^{\mathbf{b}})$$

Now consider the CBN program

```
let  f = λg.λx.g (x)
     h = ...
in   f h 1
```

A naive CBV version of it may be

```
let  f = λg.λx.(g ()) (x)
     h = λ()....
in   f h (λ().1)
```

However, an optimised CBV version is:

```
let  f = λg.λx.g (x)
     h = ...
in   f h 1
```

since, again, `f` is strict in its first argument we need not thunkify the first argument to `f` and since the argument to `g` (i.e. the second argument to `f`) is terminating we need not thunkify it. This information can be gained from the strictness type of `f`:

$$(\texttt{Int} \to \texttt{Int})^{\mathbf{b}} \to \texttt{Int}^{\mathbf{b}} \to \texttt{Int}^{\mathbf{b}}$$

and the totality type of `f`:

$$(\mathtt{Int}^{\mathbf{n}} \rightarrow \mathtt{Int}^{\mathbf{n}}) \rightarrow \mathtt{Int}^{\mathbf{n}} \rightarrow \mathtt{Int}^{\mathbf{n}}$$

Now let us combine the two examples into one:

```
let  f = λg.λx.g (x)
     a = ...
     h = ...
in   f (λx.x) a + f h 1
```

A naive CBV version of it may be

```
let  f = λg.λx.(g ()) (x)
     a = λ()....
     h = λ()....
in   f (λ().λx.x ()) a + f h (λ().1)
```

The strictness type of `f` is

$$((\mathtt{Int} \rightarrow \mathtt{Int})^{\mathbf{b}} \rightarrow \mathtt{Int}^{\mathbf{b}} \rightarrow \mathtt{Int}^{\mathbf{b}}) \wedge ((\mathtt{Int}^{\mathbf{b}} \rightarrow \mathtt{Int}^{\mathbf{b}}) \rightarrow \mathtt{Int}^{\mathbf{b}} \rightarrow \mathtt{Int}^{\mathbf{b}})$$

However, we cannot remove the thunkification of the second argument to `f` since in the second call to `f` the first argument is not a strict function. So what we get is

```
let  f = λg.λx.g (x)
     a = λ()....
     h = ...
in   f (λx.x) a + f h (λ().1)
```

The totality type of `f` is:

$$(\mathtt{Int}^{\mathbf{n}} \rightarrow \mathtt{Int}^{\mathbf{n}}) \rightarrow \mathtt{Int}^{\mathbf{n}} \rightarrow \mathtt{Int}^{\mathbf{n}}$$

We cannot use this information to remove the thunkification of the second argument to `f` since in the first call to `f` the second argument need not terminate.

But from the strictness and totality type of `f`:

$$((\mathtt{Int}^{\mathbf{n}} \rightarrow \mathtt{Int}^{\mathbf{n}}) \rightarrow \mathtt{Int}^{\mathbf{n}} \rightarrow \mathtt{Int}^{\mathbf{n}}) \wedge ((\mathtt{Int}^{\mathbf{b}} \rightarrow \mathtt{Int}^{\mathbf{b}}) \rightarrow \mathtt{Int}^{\mathbf{b}} \rightarrow \mathtt{Int}^{\mathbf{b}})$$

we can indeed remove the thunkification of the second argument to `f`.

This example shows clearly that we get more information by doing strictness and totality analysis at the same time, instead of do first strictness analysis and then totality analysis.

□

Overview In Section 2.1 we define the strictness and totality types and give rules for coercing between them; a notion of conjunction type is defined but only at "top-level"; finally the inference system is presented and examples of its use are given. In Section 2.2 we discuss the power of the fixpoint-rules; in Section 2.3 we then present a natural style operational semantics and finally in Section 2.4 the analysis is proven sound.

2.1 The Annotated Type System

2.1.1 Strictness and Totality Types

A strictness and totality type, $\mathtt{t}$, is either an annotated underlying type or a function type between strictness and totality types:

$$\begin{aligned} \mathtt{t} &::= \mathtt{ut}^{s} \mid \mathtt{t} \to \mathtt{t} \\ \mathtt{ut} &::= \mathtt{B} \mid \mathtt{ut} \to \mathtt{ut} \\ s &::= \top \mid \mathbf{n} \mid \mathbf{b} \end{aligned}$$

The annotations (the s's) can either be $\top$, $\mathbf{n}$, or $\mathbf{b}$. The idea is that a term with the strictness and totality type $\mathtt{ut}^{\mathbf{b}}$ has the underlying type $\mathtt{ut}$ and *does not* evaluate to a WHNF. A term with the strictness and totality type $\mathtt{ut}^{\mathbf{n}}$ has the underlying type $\mathtt{ut}$ and *does* evaluate to a WHNF. Finally a term with the strictness and totality type $\mathtt{ut}^{\top}$ has the underlying type $\mathtt{ut}$ but we do not know anything about the evaluation of the term. A term with the strictness and totality type $\mathtt{t}_1 \to \mathtt{t}_2$ will, when applied to a term with strictness and totality type $\mathtt{t}_1$, yield a term with strictness and totality type $\mathtt{t}_2$.

Example 2.2
All functions with the underlying type $\mathtt{ut}_1 \to \mathtt{ut}_2$ will also have the strictness and totality types $(\mathtt{ut}_1 \to \mathtt{ut}_2)^{\top}$ and $(\mathtt{ut}_1{}^{\top} \to \mathtt{ut}_2{}^{\top})$. A function with no WHNF has the strictness and totality type $(\mathtt{ut}_1 \to \mathtt{ut}_2)^{\mathbf{b}}$ and the function that applied to any term yields a term with no WHNF has the strictness and totality type $\mathtt{ut}_1{}^{\top} \to \mathtt{ut}_2{}^{\mathbf{b}}$. □

Later we shall need the predicate $\mathrm{BOT}_{\mathrm{ST}}(\mathtt{t})$ defined by

$$\begin{array}{ll} \mathrm{BOT}_{\mathrm{ST}}(\mathtt{ut}^{\mathbf{b}}) = \mathtt{tt} & \mathrm{BOT}_{\mathrm{ST}}(\mathtt{ut}^{\top}) = \mathtt{tt} \\ \mathrm{BOT}_{\mathrm{ST}}(\mathtt{ut}^{\mathbf{n}}) = \mathtt{ff} & \mathrm{BOT}_{\mathrm{ST}}(\mathtt{t}_1 \rightarrow \mathtt{t}_2) = \mathrm{BOT}_{\mathrm{ST}}(\mathtt{t}_2) \end{array}$$

The idea is, that it holds whenever the strictness and totality type will incorporate a term without WHNF.

$$[\text{ref}]\ \frac{}{\mathtt{t} \leq_{\mathrm{ST}} \mathtt{t}}$$

$$[\text{trans}]\ \frac{\mathtt{t}_1 \leq_{\mathrm{ST}} \mathtt{t}_2 \quad \mathtt{t}_2 \leq_{\mathrm{ST}} \mathtt{t}_3}{\mathtt{t}_1 \leq_{\mathrm{ST}} \mathtt{t}_3}$$

$$[\text{arrow}]\ \frac{\mathtt{t}_3 \leq_{\mathrm{ST}} \mathtt{t}_1 \quad \mathtt{t}_2 \leq_{\mathrm{ST}} \mathtt{t}_4}{\mathtt{t}_1 \rightarrow \mathtt{t}_2 \leq_{\mathrm{ST}} \mathtt{t}_3 \rightarrow \mathtt{t}_4}$$

$$[\text{top1}]\ \frac{}{\mathtt{t}\ \leq_{\mathrm{ST}}\ \varepsilon(\mathtt{t})^{\top}}$$

$$[\text{top2}]\ \frac{}{(\mathtt{ut}_1 \rightarrow \mathtt{ut}_2)^{\top}\ \leq_{\mathrm{ST}}\ \mathtt{ut}_1{}^{\top} \rightarrow \mathtt{ut}_2{}^{\top}}$$

$$[\text{bot}]\ \frac{}{(\mathtt{ut}_1 \rightarrow \mathtt{ut}_2)^{\mathbf{b}}\ \leq_{\mathrm{ST}}\ \mathtt{ut}_1{}^{\top} \rightarrow \mathtt{ut}_2{}^{\mathbf{b}}}$$

$$[\text{notbot}]\ \frac{}{\mathtt{ut}_1{}^{\mathbf{n}} \rightarrow \mathtt{ut}_2{}^{\mathbf{n}}\ \leq_{\mathrm{ST}}\ (\mathtt{ut}_1 \rightarrow \mathtt{ut}_2)^{\mathbf{n}}}$$

$$[\text{monotone}]\ \frac{}{\mathtt{t}_1 \rightarrow \mathtt{t}_2\ \leq_{\mathrm{ST}}\ \mathtt{t}'_1 \rightarrow \mathtt{t}'_2} \quad \text{if } \mathtt{t}'_1 = \downarrow\mathtt{t}_1 \text{ and } \mathtt{t}'_2 = \downarrow\mathtt{t}_2$$

Figure 2.1: Coercions Between Strictness and Totality Types

Coercions between strictness and totality types

Most terms have more than one strictness and totality type; as an example the strictness and totality types of $\lambda\mathtt{x}.7$ include $(\mathtt{Int} \rightarrow \mathtt{Int})^{\top}$, $(\mathtt{Int} \rightarrow \mathtt{Int})^{\mathbf{n}}$, and $\mathtt{Int}^{\top} \rightarrow \mathtt{Int}^{\mathbf{n}}$. Some of these are redundant and to express this we define coercions between them: $\mathtt{t}_1 \leq_{\mathrm{ST}} \mathtt{t}_2$ may only hold if all terms of strictness and totality type $\mathtt{t}_1$ also have strictness and totality type $\mathtt{t}_2$ (assuming the underlying types are the same).

The relation $\leq_{\mathrm{ST}}$ is defined by the rules in Figure 2.1: it is reflexive, transitive, and anti-monotonic in contravariant position. The three first rules are as the rules for the standard types (Figure 1.2). We write $\equiv$ for the equivalence induced by $\leq_{\mathrm{ST}}$, i.e. $\mathtt{t}_1 \equiv \mathtt{t}_2$ if and only if $\mathtt{t}_1 \leq_{\mathrm{ST}} \mathtt{t}_2$ and $\mathtt{t}_2 \leq_{\mathrm{ST}} \mathtt{t}_1$. The rule [top1] expresses that the strictness and totality type $\mathtt{ut}^{\top}$ is the greatest among the strictness and totality types with the underlying type $\mathtt{ut}$. One axiom derived from the rule [top1] is

$$\mathtt{ut}_1{}^{\top} \rightarrow \mathtt{ut}_2{}^{\top} \leq_{\mathrm{ST}} (\mathtt{ut}_1 \rightarrow \mathtt{ut}_2)^{\top} \qquad (2.1)$$

Axiom (2.1) then motivates rule [top2] because when combined they yield

$$(\mathtt{ut}_1 \to \mathtt{ut}_2)^\top \equiv \mathtt{ut}_1{}^\top \to \mathtt{ut}_2{}^\top$$

The left-hand side of the rule [bot] represents the functions without WHNF and the right-hand side represents all non-terminating functions; this also includes the functions without WHNF. The rule [notbot] says that functions that map terms with a WHNF to a term with WHNF are also included in the functions with a WHNF.

The rule [monotone] ensures that we live in a universe of monotone functions: if we know less about the argument to a function, then we should know less about the result as well. The formulation of this requires the function $\downarrow$ on strictness and totality types defined by

$$\begin{aligned}
\downarrow(\mathtt{ut}^{\mathbf{b}}) &= \mathtt{ut}^{\mathbf{b}} && (2.2)\\
\downarrow(\mathtt{ut}^{\top}) &= \mathtt{ut}^{\top} && (2.3)\\
\downarrow(\mathtt{ut}^{\mathbf{n}}) &= \mathtt{ut}^{\top} && (2.4)\\
\downarrow(\mathtt{t}_1 \to \mathtt{t}_2) &= \mathtt{t}_1 \to \downarrow\mathtt{t}_2 && (2.5)
\end{aligned}$$

The idea behind $\downarrow$ is that $\downarrow\mathtt{t}$ is the smallest type (in the sense of "containing" fewest elements) such that both $\mathtt{t} \leq_{\mathrm{ST}} \downarrow\mathtt{t}$ and $\mathrm{BOT}_{\mathrm{ST}}(\downarrow\mathtt{t})$ hold; this if formalised in Fact 2.27 in Section 2.4.1.

To see that the rule [monotone] is useful consider the term `twice`:

$$\lambda\mathtt{f}.\lambda\mathtt{x}.\mathtt{f}\ (\mathtt{f}\ \mathtt{x})$$

and the strictness and totality type

$$(\mathtt{Int}^{\mathbf{n}} \to \mathtt{Int}^{\mathbf{b}}) \to \mathtt{Int}^{\top} \to \mathtt{Int}^{\mathbf{b}}$$

In order to show that `twice` does indeed have that type we must be able to coerce

$$\mathtt{Int}^{\mathbf{n}} \to \mathtt{Int}^{\mathbf{b}} \leq_{\mathrm{ST}} \mathtt{Int}^{\top} \to \mathtt{Int}^{\mathbf{b}}$$

However we cannot do so without the [monotone]-rule. For more details see Example 2.7 below.

We shall later show that the relation $\leq_{\mathrm{ST}}$ is sound (Lemma 2.39). However it is not complete. To see this consider the two strictness and totality types

$\mathtt{Int}^{\mathbf{b}} \to \mathtt{Int}^{\mathbf{n}}$ and $\mathtt{Int}^{\top} \to \mathtt{Int}^{\mathbf{n}}$. It must be the case that every term with the first type also has the second type and vice versa since the terms are monotonic. However, although we can infer

$$\mathtt{Int}^{\top} \to \mathtt{Int}^{\mathbf{n}} \leq_{\mathrm{ST}} \mathtt{Int}^{\mathbf{b}} \to \mathtt{Int}^{\mathbf{n}}$$

it turns out that we cannot infer

$$\mathtt{Int}^{\mathbf{b}} \to \mathtt{Int}^{\mathbf{n}} \leq_{\mathrm{ST}} \mathtt{Int}^{\top} \to \mathtt{Int}^{\mathbf{n}}$$

using the coercions. This can be remedied by introducing the rule [monotone2] below: first we define the function $\uparrow$ on strictness and totality types as follows:

$$\begin{array}{ll} \uparrow(\mathtt{ut}^{\mathbf{b}}) = \mathtt{ut}^{\top} & \uparrow(\mathtt{ut}^{\top}) = \mathtt{ut}^{\top} \\ \uparrow(\mathtt{ut}^{\mathbf{n}}) = \mathtt{ut}^{\mathbf{n}} & \uparrow(\mathtt{t}_1 \to \mathtt{t}_2) = \mathtt{t}_1 \to \uparrow\mathtt{t}_2 \end{array}$$

The idea behind $\uparrow$ is that it is the smallest type such that both $\mathtt{t} \leq_{\mathrm{ST}} \uparrow\mathtt{t}$ and $\textsc{notBOT}_{\mathrm{ST}}(\uparrow\mathtt{t})$ hold where the predicate $\textsc{notBOT}_{\mathrm{ST}}(\mathtt{t})$ must hold whenever the strictness and totality type must incorporate a term with a WHNF. Now we can write the new coercion rule using $\uparrow$:

$$[\text{monotone2}]\ \frac{}{\mathtt{t}_1 \to \mathtt{t}_2 \ \leq_{\mathrm{ST}}\ \mathtt{t}_1' \to \mathtt{t}_2'} \quad \text{if } \mathtt{t}_1' = \uparrow\mathtt{t}_1 \text{ and } \mathtt{t}_2' = \uparrow\mathtt{t}_2$$

With this rule we can infer $\mathtt{Int}^{\mathbf{b}} \to \mathtt{Int}^{\mathbf{n}} \leq_{\mathrm{ST}} \mathtt{Int}^{\top} \to \mathtt{Int}^{\mathbf{n}}$. More work is needed to clarify if $\leq_{\mathrm{ST}}$ is complete with the new rule added.

Example 2.3

To see that the rule [monotone2] is useful consider the term `twice` defined by

$$\lambda\mathtt{f}.\lambda\mathtt{x}.\mathtt{f}\ (\mathtt{f}\ \mathtt{x})$$

and the strictness and totality type

$$(\mathtt{Int}^{\mathbf{b}} \to \mathtt{Int}^{\mathbf{n}}) \to \mathtt{Int}^{\top} \to \mathtt{Int}^{\mathbf{n}}$$

In order to show that `twice` does indeed have that type we must be able to coerce

$$\mathtt{Int}^{\mathbf{b}} \to \mathtt{Int}^{\mathbf{n}} \leq_{\mathrm{ST}} \mathtt{Int}^{\top} \to \mathtt{Int}^{\mathbf{n}}$$

However we cannot do so without the [monotone2]-rule. The details are analogous to Example 2.7 below.

□

While we conjecture that adding [monotone2] will be semantically sound the technical machinery needed for characterising the new auxiliary concepts, $\uparrow$ and NOTBOT$_{\mathrm{ST}}$, (corresponding to $\downarrow$ and BOT$_{\mathrm{ST}}$ for [monotone]) in order to formally prove our conjecture is sufficiently involved that we shall dispense with so doing.

2.1.2 Conjunction Types

Based on the strictness and totality types we now define the conjunction types. A conjunction type, $\mathtt{ct}$, is either a strictness and totality type or a conjunction of two conjunction types:

$$\begin{array}{rcl}\mathtt{ct} & ::= & \mathtt{t} \mid \mathtt{ct} \wedge \mathtt{ct} \\ \mathtt{t} & ::= & \mathtt{ut}^{s} \mid \mathtt{t} \to \mathtt{t} \\ \mathtt{ut} & ::= & \mathtt{B} \mid \mathtt{ut} \to \mathtt{ut} \\ s & ::= & \top \mid \mathbf{n} \mid \mathbf{b}\end{array}$$

Thus conjunction is only allowed at the top-level (just like type-schemes in ML are only allowed at the top-level [Mil78]). The introduction of conjunction types means that it is possible to have empty types like $\mathtt{Int}^{\mathbf{n}} \wedge \mathtt{Int}^{\mathbf{b}}$. Actually, the fine details of empty types are closely connected with the choice of semantic model: emptiness of the type

$$\begin{array}{l}(\mathtt{Int}^{\mathbf{b}} \to \mathtt{Int}^{\mathbf{n}} \to \mathtt{Int}^{\mathbf{n}}) \\ \quad \wedge (\mathtt{Int}^{\mathbf{n}} \to \mathtt{Int}^{\mathbf{b}} \to \mathtt{Int}^{\mathbf{n}}) \wedge (\mathtt{Int}^{\mathbf{b}} \to \mathtt{Int}^{\mathbf{b}} \to \mathtt{Int}^{\mathbf{b}})\end{array}$$

depends on whether the semantic model allows non-sequential behaviours of type $\mathtt{Int} \to \mathtt{Int} \to \mathtt{Int}$. This will normally be the case for denotational semantics but will not be the case for natural-style operational semantics when the order of evaluation is forced (as when specifying lazy reduction to WHNF). The restriction to top-level conjunctions allows us to avoid some of the problems introduced by empty types; we return to this later.

Since a term can only have one underlying type a well-formed conjunction type will not involve types with different underlying types. The well-formedness predicate is defined by:

$$\overline{\vdash^{W} \mathtt{t}}$$

a strictness and totality type is well-formed viewed as a conjunction type, and a conjunction of annotated types is well-formed whenever the two conjuncts are well-formed and they have the same underlying type:

$$\frac{\vdash^W \mathtt{ct}_1 \quad \vdash^W \mathtt{ct}_2}{\vdash^W \mathtt{ct}_1 \wedge \mathtt{ct}_2} \quad \text{if } \varepsilon(\mathtt{ct}_1) = \varepsilon(\mathtt{ct}_2)$$

This allows us to overload the function ε to also find the underlying type of a conjunction type: $\varepsilon(\mathtt{ct}_1 \wedge \mathtt{ct}_2) = \varepsilon(\mathtt{ct}_1)$. The predicate $\mathrm{BOT}_{\mathrm{ST}}$ is lifted to conjunction types:

$$\begin{aligned} \mathrm{BOT}_{\mathrm{CT}}(\mathtt{ct}_1 \wedge \mathtt{ct}_2) &= \mathrm{BOT}_{\mathrm{CT}}(\mathtt{ct}_1) \wedge \mathrm{BOT}_{\mathrm{CT}}(\mathtt{ct}_2) \\ \mathrm{BOT}_{\mathrm{CT}}(\mathtt{t}) &= \mathrm{BOT}_{\mathrm{ST}}(\mathtt{t}) \end{aligned}$$

The rules for coercing between conjunction types are given in Figure 2.2. The relation $\leq_{\mathrm{CT}}$ is reflexive and transitive, and for strictness and totality types the relation is inherited from the relation, $\leq_{\mathrm{ST}}$, on strictness and totality types; this is express by the rule [type] in Figure 2.2. For conjunctions we have the three rules as in Benton and Jensen [Ben93, Jen91, Jen92b, Jen92a]. However we do not have the rule

$$\frac{}{(\mathtt{t}_1 \rightarrow \mathtt{t}_2) \wedge (\mathtt{t}_1 \rightarrow \mathtt{t}_3) \leq_{\mathrm{CT}} \mathtt{t}_1 \rightarrow (\mathtt{t}_2 \wedge \mathtt{t}_3)}$$

the reason for this is that $(\mathtt{t}_1 \rightarrow (\mathtt{t}_2 \wedge \mathtt{t}_3))$ is not a well-formed conjunction type here.

2.1.3 The Conjunction Type System

We have now prepared the ground for presenting the conjunction type inference system of Figure 2.3. The list A of assumptions gives strictness and totality types to free variables. All the variables in the list are distinct. Only the lambda abstraction can extend the assumption list and since conjunction types only can appear at the top-level this means that assumption lists always will associate strictness and totality types, not conjunction types, with the variables. For each constant $\mathtt{c}$, we assume that a conjunction type $\mathtt{ct}_{\mathtt{c}}$ is specified; as an example $\mathtt{ct}_{\mathtt{succ}} = (\mathtt{Int}^{\mathbf{n}} \rightarrow \mathtt{Int}^{\mathbf{n}}) \wedge (\mathtt{Int}^{\mathbf{b}} \rightarrow \mathtt{Int}^{\mathbf{b}})$.

The rules [var], [abs], [app], and [const] are just as their standard type inference counterparts (see Figure 1.1). There are three rules for conditional

$$[\text{ref}]\ \frac{}{\mathtt{ct} \leq_{\text{CT}} \mathtt{ct}}$$

$$[\text{trans}]\ \frac{\mathtt{ct}_1 \leq_{\text{CT}} \mathtt{ct}_2 \quad \mathtt{ct}_2 \leq_{\text{CT}} \mathtt{ct}_3}{\mathtt{ct}_1 \leq_{\text{CT}} \mathtt{ct}_3}$$

$$[\wedge 1]\ \frac{}{\mathtt{ct}_1 \wedge \mathtt{ct}_2 \leq_{\text{CT}} \mathtt{ct}_1}$$

$$[\wedge 2]\ \frac{}{\mathtt{ct}_1 \wedge \mathtt{ct}_2 \leq_{\text{CT}} \mathtt{ct}_2}$$

$$[\wedge 3]\ \frac{\mathtt{ct} \leq_{\text{CT}} \mathtt{ct}_1 \quad \mathtt{ct} \leq_{\text{CT}} \mathtt{ct}_2}{\mathtt{ct} \leq_{\text{CT}} \mathtt{ct}_1 \wedge \mathtt{ct}_2}$$

$$[\text{type}]\ \frac{\mathtt{t}_1 \ \leq_{\text{ST}} \ \mathtt{t}_2}{\mathtt{t}_1 \leq_{\text{CT}} \mathtt{t}_2}$$

Figure 2.2: Coercions Between Conjunction Types

— depending on whether the test is of strictness and totality type $\mathtt{Bool}^{\mathbf{b}}$, $\mathtt{Bool}^{\mathbf{n}}$, or $\mathtt{Bool}^{\top}$.

The rule [coer] can be applied to change the strictness and totality type to a greater strictness and totality type. It is quite useful as a preparation for applying rule [if3]. The rule [conj] allows to construct conjunction types (as is the case also for rule [const]).

From rule [fix] we may derive rules

$$[\text{fix1}]\ \frac{\mathrm{A} \vdash_{\text{ST}} \mathtt{e} : \mathtt{t} \to \mathtt{t}}{\mathrm{A} \vdash_{\text{ST}} \mathtt{fix}\ \mathtt{e} : \mathtt{t}} \quad \text{if } \mathrm{BOT}_{\text{ST}}(\mathtt{t})$$

and

$$[\text{fix2}]\ \frac{\mathrm{A} \vdash_{\text{ST}} \mathtt{e} : \mathtt{t}_1 \to \mathtt{t}_2}{\mathrm{A} \vdash_{\text{ST}} \mathtt{fix}\ \mathtt{e} : \mathtt{t}_2} \quad \text{if } \mathrm{BOT}_{\text{ST}}(\mathtt{t}_1) \text{ and } \mathtt{t}_2 \ \leq_{\text{ST}} \ \mathtt{t}_1$$

that are simpler and more intuitive. Note that in rule [fix] we have to ensure that the type $\mathtt{t}_0$ can describe bottom in order to be able to calculate the fixpoint. After the first iteration, see Figure 2.4, the term has the strictness and totality type $\mathtt{t}_1$ and after the second the strictness and totality type $\mathtt{t}_2$, etc. When the term reaches the strictness and totality type $\mathtt{t}_q$ we can apply the rule [coer] because we have $\mathtt{t}_q \ \leq_{\text{ST}} \ \mathtt{t}_p$ and so the term has the strictness and totality type $\mathtt{t}_p$. In this way we can go on as long as

$$[\text{var}]\ \frac{}{\text{A} \vdash_{\text{ST}} \mathtt{x} : \mathtt{t}} \quad \text{if } \mathtt{x} : \mathtt{t} \in \text{A}$$

$$[\text{abs}]\ \frac{\text{A}, \mathtt{x} : \mathtt{t}_1 \vdash_{\text{ST}} \mathtt{e} : \mathtt{t}_2}{\text{A} \vdash_{\text{ST}} \lambda\mathtt{x}.\mathtt{e} : \mathtt{t}_1 \rightarrow \mathtt{t}_2}$$

$$[\text{abs2}]\ \frac{\text{A}, \mathtt{x} : \mathtt{t}_1 \vdash_{\text{ST}} \text{e} : \mathtt{t}_2}{\text{A} \vdash_{\text{ST}} \lambda\mathtt{x}.\mathtt{e} : \varepsilon(\mathtt{t}_1 \rightarrow \mathtt{t}_2)^{\mathbf{n}}}$$

$$[\text{app}]\ \frac{\text{A} \vdash_{\text{ST}} \mathtt{e}_1 : \mathtt{t}_1 \rightarrow \mathtt{t}_2 \quad \text{A} \vdash_{\text{ST}} \mathtt{e}_2 : \mathtt{t}_1}{\text{A} \vdash_{\text{ST}} \mathtt{e}_1\ \mathtt{e}_2 : \mathtt{t}_2}$$

$$[\text{if1}]\ \frac{\text{A} \vdash_{\text{ST}} \mathtt{e}_1 : \mathtt{Bool}^{\mathbf{b}} \quad \text{A} \vdash_{\text{ST}} \mathtt{e}_2 : \mathtt{ct} \quad \text{A} \vdash_{\text{ST}} \mathtt{e}_3 : \mathtt{ct}}{\text{A} \vdash_{\text{ST}} \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 : \varepsilon(\mathtt{ct})^{\mathbf{b}}}$$

$$[\text{if2}]\ \frac{\text{A} \vdash_{\text{ST}} \mathtt{e}_1 : \mathtt{Bool}^{\mathbf{n}} \quad \text{A} \vdash_{\text{ST}} \mathtt{e}_2 : \mathtt{ct} \quad \text{A} \vdash_{\text{ST}} \mathtt{e}_3 : \mathtt{ct}}{\text{A} \vdash_{\text{ST}} \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 : \mathtt{ct}}$$

$$[\text{if3}]\ \frac{\text{A} \vdash_{\text{ST}} \mathtt{e}_1 : \mathtt{Bool}^{\top} \quad \text{A} \vdash_{\text{ST}} \mathtt{e}_2 : \mathtt{ct} \quad \text{A} \vdash_{\text{ST}} \mathtt{e}_3 : \mathtt{ct}}{\text{A} \vdash_{\text{ST}} \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 : \mathtt{ct}} \quad \text{if } \text{BOT}_{\text{CT}}(\mathtt{ct})$$

$$[\text{fix}]\ \frac{\text{A} \vdash_{\text{ST}} \mathtt{e} : (\mathtt{t}_0 \rightarrow \mathtt{t}_1) \wedge (\mathtt{t}_1 \rightarrow \mathtt{t}_2) \wedge \ldots \wedge (\mathtt{t}_{n-1} \rightarrow \mathtt{t}_n)}{\text{A} \vdash_{\text{ST}} \mathtt{fix}\ \mathtt{e} : \mathtt{t}_n} \quad \text{if} \begin{cases} \text{BOT}_{\text{ST}}(\mathtt{t}_0), \\ \exists p, q : p < q \\ \wedge \mathtt{t}_q \leq_{\text{ST}} \mathtt{t}_p \end{cases}$$

$$[\text{const}]\ \frac{}{\text{A} \vdash_{\text{ST}} \mathtt{c} : \mathtt{ct}_{\mathtt{c}}}$$

$$[\text{coer}]\ \frac{\text{A} \vdash_{\text{ST}} \mathtt{e} : \mathtt{ct}_1}{\text{A} \vdash_{\text{ST}} \mathtt{e} : \mathtt{ct}_2} \quad \text{if } \mathtt{ct}_1 \leq_{\text{CT}} \mathtt{ct}_2$$

$$[\text{conj}]\ \frac{\text{A} \vdash_{\text{ST}} \mathtt{e} : \mathtt{ct}_1 \quad \text{A} \vdash_{\text{ST}} \mathtt{e} : \mathtt{ct}_2}{\text{A} \vdash_{\text{ST}} \mathtt{e} : \mathtt{ct}_1 \wedge \mathtt{ct}_2}$$

Figure 2.3: Conjunction Type Inference

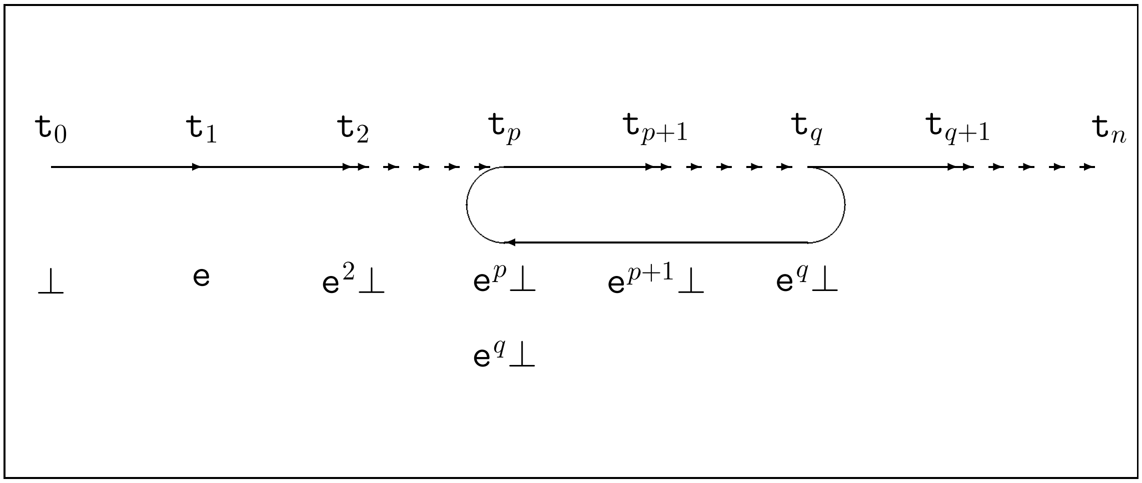

Figure 2.4: Picturing the [fix]-rule

necessary to evaluate the fixpoint. Finally we iterate $n - q$ more times to get the type $\mathtt{t}_n$ for the fixpoint.

The following observations are easily verified by induction on the shape of the inference tree:

Fact 2.4
If $\mathrm{A} \vdash_{\mathrm{ST}} \mathtt{e} : \mathtt{ct}$ then $\vdash^W \mathtt{ct}$ and $\varepsilon(\mathrm{A}) \vdash \mathtt{e} : \varepsilon(\mathtt{ct})$. □

Proof We assume $\mathrm{A} \vdash_{\mathrm{ST}} \mathtt{e} : \mathtt{ct}$ and then we prove by induction on the proof-tree for $\mathrm{A} \vdash_{\mathrm{ST}} \mathtt{e} : \mathtt{ct}$ that $\varepsilon(\mathrm{A}) \vdash \mathtt{e} : \varepsilon(\mathtt{ct})$ can be inferred.

For the full details see Appendix page 223. ■

We also have a form of completeness:

Fact 2.5
If $\mathrm{A} \vdash \mathtt{e} : \mathtt{ut}$ then $\mathtt{top}(\mathrm{A}) \vdash_{\mathrm{ST}} \mathtt{e} : \mathtt{ut}^\top$ where $\mathtt{top}(\mathtt{x} : \mathtt{ut}, \mathrm{A}) = (\mathtt{x} : \mathtt{ut}^\top)$, $\mathtt{top}(\mathrm{A})$. □

Example 2.6
In the inference system we can infer $\emptyset \vdash_{\mathrm{ST}} \mathtt{fix}\ (\lambda\mathtt{x}.\mathtt{x}) : \mathtt{Int}^{\mathbf{b}}$ which is more precise than the $\mathtt{Int}^\top$ obtained by [Wri91]. In the systems of [Ben93, Jen91, Jen92b] the best one can infer is the type $\mathtt{Int}^\top$ for the term $\mathtt{fix}\ (\lambda\mathtt{x}.7)$ whereas we can infer $\emptyset \vdash_{\mathrm{ST}} \mathtt{fix}\ (\lambda\mathtt{x}.7) : \mathtt{Int}^{\mathbf{n}}$ and so again are more precise. □

Example 2.7

The term twice is given by

$$\lambda\texttt{f}.\lambda\texttt{x}.\texttt{f (f x)}$$

and has the strictness and totality type

$$\texttt{t} \;=\; (\texttt{Int}^{\mathbf{n}} \rightarrow \texttt{Int}^{\mathbf{b}}) \rightarrow \texttt{Int}^{\top} \rightarrow \texttt{Int}^{\mathbf{b}}$$

In order to show this we need to apply the rule [monotone]. For this let

$$\mathrm{A} \;=\; \texttt{f} : \texttt{Int}^{\mathbf{n}} \rightarrow \texttt{Int}^{\mathbf{b}}, \texttt{x} : \texttt{Int}^{\top}$$

and let P_1 be

$$\frac{\texttt{Int}^{\mathbf{n}} \rightarrow \texttt{Int}^{\mathbf{b}} \leq_{\mathrm{ST}} \texttt{Int}^{\top} \rightarrow \texttt{Int}^{\mathbf{b}}}{\mathrm{A} \vdash_{\mathrm{ST}} \texttt{f} : \texttt{Int}^{\top} \rightarrow \texttt{Int}^{\mathbf{b}}}\ [\mathrm{var}] + [\mathrm{coer}] + [\mathrm{monotone}]$$

and let P_2 be

$$\frac{\dfrac{\texttt{Int}^{\mathbf{n}} \rightarrow \texttt{Int}^{\mathbf{b}} \leq_{\mathrm{ST}} \texttt{Int}^{\top} \rightarrow \texttt{Int}^{\top}}{\mathrm{A} \vdash_{\mathrm{ST}} \texttt{f} : \texttt{Int}^{\top} \rightarrow \texttt{Int}^{\top}}\ \begin{array}{l}[\mathrm{var}] + [\mathrm{coer}] + \\ [\mathrm{monotone}]\end{array} \qquad \mathrm{A} \vdash_{\mathrm{ST}} \texttt{x} : \texttt{Int}^{\top}}{\mathrm{A} \vdash_{\mathrm{ST}} \texttt{f x} : \texttt{Int}^{\top}}\ [\mathrm{app}]$$

Now we have

$$\frac{\dfrac{\dfrac{P_1 \qquad P_2}{\mathrm{A} \vdash_{\mathrm{ST}} \texttt{f (f x)} : \texttt{Int}^{\mathbf{b}}}\ [\mathrm{app}]}{\texttt{f} : \texttt{Int}^{\mathbf{n}} \rightarrow \texttt{Int}^{\mathbf{b}} \vdash_{\mathrm{ST}} \lambda\texttt{x}.\texttt{f (f x)} : \texttt{Int}^{\top} \rightarrow \texttt{Int}^{\mathbf{b}}}\ [\mathrm{abs}]}{\emptyset \vdash_{\mathrm{ST}} \lambda\texttt{f}.\lambda\texttt{x}.\texttt{f (f x)} : \texttt{t}}\ [\mathrm{abs}]$$

In this example we have used the rule [monotone] in an essential way. □

Example 2.8
Consider the term[1] e and types $\texttt{t}_1$ and $\texttt{t}_2$:

$$\begin{aligned}
\texttt{e} &= \lambda\text{f}.\lambda\text{x}.\lambda\text{y}.\lambda\text{z}.\texttt{if (= 0 z) then + x y else f y x (- z 1)} \\
\texttt{t}_1 &= \texttt{Int}^{\mathbf{b}} \rightarrow \texttt{Int}^{\top} \rightarrow \texttt{Int}^{\top} \rightarrow \texttt{Int}^{\mathbf{b}} \\
\texttt{t}_2 &= \texttt{Int}^{\top} \rightarrow \texttt{Int}^{\mathbf{b}} \rightarrow \texttt{Int}^{\top} \rightarrow \texttt{Int}^{\mathbf{b}}
\end{aligned}$$

We want to infer $\emptyset \vdash_{\mathrm{ST}} \texttt{fix e} : \texttt{t}_1$ but it is not possible to infer

$$\emptyset \vdash_{\mathrm{ST}} \texttt{e} : \texttt{t}_1 \rightarrow \texttt{t}_1$$

[1]This example is due to Kuo and Mishra [KM89].

However we can infer $\emptyset \vdash_{\mathrm{ST}} \mathtt{e} : \mathtt{t}_1 \to \mathtt{t}_2$ and $\emptyset \vdash_{\mathrm{ST}} \mathtt{e} : \mathtt{t}_2 \to \mathtt{t}_1$ and we can apply the [conj]-rule to get $\emptyset \vdash_{\mathrm{ST}} \mathtt{e} : (\mathtt{t}_1 \to \mathtt{t}_2) \wedge (\mathtt{t}_2 \to \mathtt{t}_1)$. Now we are able to apply the rule [fix] and thereby get $\emptyset \vdash_{\mathrm{ST}} \mathtt{fix}\,\mathtt{e} : \mathtt{t}_1$ as desired. This shows that even though we do not have "full" conjunction system of Jensen and Benton [Jen92a, Ben93] we *can* make good use of conjunction to type the "difficult" example of [KM89]. □

Example 2.9
Consider next the term[2] e given by

$$\begin{array}{rcl} \mathtt{e} & = & \mathtt{twice\ g} \\ \mathtt{twice} & = & \lambda\mathtt{f}.\lambda\mathtt{x}.\mathtt{f}\ (\mathtt{f}\ \mathtt{x}) \\ \mathtt{g} & = & \lambda\mathtt{y}.\lambda\mathtt{x}.+\ \mathtt{x}\ (\mathtt{y}\ (\mathtt{fix}\ \lambda\mathtt{x}.\mathtt{x})) \end{array}$$

It will have the strictness and totality type

$$(\mathtt{Int}^{\top} \to \mathtt{Int}^{\top}) \to \mathtt{Int}^{\top} \to \mathtt{Int}^{\mathbf{b}}$$

but we are *not* able to obtain it using our analysis because one needs full conjunction in order to construct the proof-tree. The reason is that we need to infer that `twice` has the type

$$((\mathtt{t}_1 \to \mathtt{t}_2) \wedge (\mathtt{t}_2 \to \mathtt{t}_3)) \to (\mathtt{t}_1 \to \mathtt{t}_3)$$

for any $\mathtt{t}_1$, $\mathtt{t}_2$, and $\mathtt{t}_3$ but this is not a well-formed conjunction type in the current system. □

2.2 The Power of the Fix-rules

Previous work on strictness analysis [Jen91, Jen92b, Ben93, KM89] contain only a simple fix-rule corresponding to our [fix1]-rule rather than our more general [fix]-rule. In this section we will investigate the extent to which this is essential.

Let $\mathcal{A}$ be a set of permissible annotations; so far we used $\mathcal{A} = \{\mathbf{n}, \mathbf{b}, \top\}$ but we shall consider also the restriction $\mathcal{A} = \{\mathbf{b}, \top\}$ that disallows $\mathbf{n}$ and that corresponds more closely to the aims of [Jen91, Jen92b, Ben93, KM89]. Note that the side-condition $\mathrm{BOT}_{\mathrm{ST}}(\mathtt{t})$ is trivially true when $\mathcal{A} = \{\mathbf{b}, \top\}$.

[2] Thanks to Nick Benton for pointing to this example.

Let $\vdash^{\mathcal{A}}_{\text{fix}}$ be the inference system of Figure 2.3 but with annotations in $\mathcal{A}$. Similarly let $\vdash^{\mathcal{A}}_{\text{fix1}}$ be the system where [fix] is replaced by [fix1] and let $\vdash^{\mathcal{A}}_{\text{fix2}}$ be the system where [fix] is replaced by [fix2]. Note that $\vdash^{\{\mathbf{b},\top\}}_{\text{fix1}}$ is the system of [KM89].

For any two inference systems $\vdash^{\mathcal{A}_1}_{\phi_1}$ and $\vdash^{\mathcal{A}_2}_{\phi_2}$ write

$$\vdash^{\mathcal{A}_1}_{\phi_1} \subseteq \vdash^{\mathcal{A}_2}_{\phi_2} \quad \text{for} \quad \mathrm{A} \vdash^{\mathcal{A}_1}_{\phi_1} \mathtt{e : t} \Rightarrow \mathrm{A} \vdash^{\mathcal{A}_2}_{\phi_2} \mathtt{e : t}$$

and

$$\begin{array}{lll} \vdash^{\mathcal{A}_1}_{\phi_1} = \vdash^{\mathcal{A}_2}_{\phi_2} & \text{for} & \vdash^{\mathcal{A}_1}_{\phi_1} \subseteq \vdash^{\mathcal{A}_2}_{\phi_2} \wedge \vdash^{\mathcal{A}_2}_{\phi_2} \subseteq \vdash^{\mathcal{A}_1}_{\phi_1} \\ \vdash^{\mathcal{A}_1}_{\phi_1} \subset \vdash^{\mathcal{A}_2}_{\phi_2} & \text{for} & \vdash^{\mathcal{A}_1}_{\phi_1} \subseteq \vdash^{\mathcal{A}_2}_{\phi_2} \wedge \neg(\vdash^{\mathcal{A}_2}_{\phi_2} \subseteq \vdash^{\mathcal{A}_1}_{\phi_1}) \end{array}$$

It is immediate that

$$\vdash^{\mathcal{A}}_{\text{fix1}} \subseteq \vdash^{\mathcal{A}}_{\text{fix2}} \subseteq \vdash^{\mathcal{A}}_{\text{fix}}$$

and that

$$\vdash^{\{\mathbf{b},\top\}}_{\phi} \subseteq \vdash^{\{\mathbf{n},\mathbf{b},\top\}}_{\phi}$$

for all $\mathcal{A}$ and $\phi \in \{\text{fix}, \text{fix1}, \text{fix2}\}$. We now consider the extent to which the inclusions are proper or are equalities; the results are summarised in Table 2.1.

annotations $\mathcal{A}$	fix-rules
$\{\mathbf{b}, \top\}$	$\vdash^{\mathcal{A}}_{\text{fix1}} = \vdash^{\mathcal{A}}_{\text{fix2}}$
	$\vdash^{\mathcal{A}}_{\text{fix2}} \subset \vdash^{\mathcal{A}}_{\text{fix}}$
$\{\mathbf{n}, \mathbf{b}, \top\}$	$\vdash^{\mathcal{A}}_{\text{fix1}} \subset \vdash^{\mathcal{A}}_{\text{fix2}}$
	$\vdash^{\mathcal{A}}_{\text{fix2}} \subset \vdash^{\mathcal{A}}_{\text{fix}}$

Table 2.1: Relation Between the Fix-rules

Claim $\vdash^{\{\mathbf{b},\top\}}_{\mathrm{fix1}} = \vdash^{\{\mathbf{b},\top\}}_{\mathrm{fix2}}$

In order to show that $\vdash^{\{\mathbf{b},\top\}}_{\mathrm{fix1}} = \vdash^{\{\mathbf{b},\top\}}_{\mathrm{fix2}}$ it suffices to show that the rule [fix2] can be derived from the rule [fix1]. For this assume

$$\begin{array}{l} \mathrm{A} \vdash^{\{\mathbf{b},\top\}}_{\mathrm{fix1}} \mathtt{e} : \mathtt{t}_1 \to \mathtt{t}_2 \\ \mathtt{t}_2 \leq_{\mathrm{ST}} \mathtt{t}_1 \\ \mathrm{BOT}_{\mathrm{ST}}(\mathtt{t}_1) \end{array}$$

so that

$$\mathrm{A} \vdash^{\{\mathbf{b},\top\}}_{\mathrm{fix1}} \mathtt{fix}\,\mathtt{e} : \mathtt{t}_2$$

can be inferred. Since none of the types involves the annotation **n** it must be the case that $\mathrm{BOT}_{\mathrm{ST}}(\mathtt{t}) = \mathtt{tt}$ for all types $\mathtt{t}$. We can now construct the proof-tree

$$\dfrac{\dfrac{\mathrm{A} \vdash^{\{\mathbf{b},\top\}}_{\mathrm{fix1}} \mathtt{e} : \mathtt{t}_1 \to \mathtt{t}_2 \qquad \dfrac{\mathtt{t}_2 \leq_{\mathrm{ST}} \mathtt{t}_1}{\mathtt{t}_1 \to \mathtt{t}_2 \leq_{\mathrm{ST}} \mathtt{t}_2 \to \mathtt{t}_2}}{\mathrm{A} \vdash^{\{\mathbf{b},\top\}}_{\mathrm{fix1}} \mathtt{e} : \mathtt{t}_2 \to \mathtt{t}_2} \qquad \mathrm{BOT}_{\mathrm{ST}}(\mathtt{t}_2)}{\mathrm{A} \vdash^{\{\mathbf{b},\top\}}_{\mathrm{fix1}} \mathtt{fix}\,\mathtt{e} : \mathtt{t}_2}\ [\mathrm{fix1}]$$

and this proves our claim.

Claim $\vdash^{\{\mathbf{b},\top\}}_{\mathrm{fix2}} \subset \vdash^{\{\mathbf{b},\top\}}_{\mathrm{fix}}$

To verify that $\vdash^{\{\mathbf{b},\top\}}_{\mathrm{fix2}} \subset \vdash^{\{\mathbf{b},\top\}}_{\mathrm{fix}}$ we must show that there exists a term $\mathtt{e}$ and a type $\mathtt{t}$ and an assumption list A, such that $\mathrm{A} \vdash^{\{\mathbf{b},\top\}}_{\mathrm{fix}} \mathtt{e} : \mathtt{t}$ can be inferred and we cannot infer $\mathrm{A} \vdash^{\{\mathbf{b},\top\}}_{\mathrm{fix2}} \mathtt{e} : \mathtt{t}$.

For this we take

$$\begin{array}{rcl} \mathtt{e} & = & \mathtt{fix}\,(\lambda \mathrm{f}.\lambda \mathrm{x}.\lambda \mathrm{y}.\lambda \mathrm{z}.\mathtt{if\ (=\ 0\ z)\ then\ +\ x\ y\ else\ f\ y\ x\ (-\ z\ 1)}) \\ \mathtt{t}_1 & = & \mathtt{Int}^{\mathbf{b}} \to \mathtt{Int}^{\top} \to \mathtt{Int}^{\top} \to \mathtt{Int}^{\mathbf{b}} \\ \mathtt{t}_2 & = & \mathtt{Int}^{\top} \to \mathtt{Int}^{\mathbf{b}} \to \mathtt{Int}^{\top} \to \mathtt{Int}^{\mathbf{b}} \end{array}$$

In Example 2.6 we have shown how to infer:

$$\emptyset \vdash^{\{\mathbf{b},\top\}}_{\mathrm{fix}} \mathtt{fix}\,\mathtt{e} : \mathtt{t}_1 \wedge \mathtt{t}_2$$

and we argued about the unlikeliness of being able to infer $\emptyset \vdash^{\{\mathbf{b},\top\}}_{\text{fix2}} \mathtt{fix\ e} : \mathtt{t}_1 \wedge \mathtt{t}_2$ (as is indeed stated also in [KM89]).

Claim $\vdash^{\{\mathbf{n},\mathbf{b},\top\}}_{\text{fix1}} \subset \vdash^{\{\mathbf{n},\mathbf{b},\top\}}_{\text{fix2}}$

When we go to the $\{\mathbf{n}, \mathbf{b}, \top\}$-part (both strictness and totality information on the types) the two rules [fix1] and [fix2] are no longer equivalent. Consider the term $\mathtt{fix}\ (\lambda\mathtt{x}.7)$ and the type $\mathtt{Int}^{\mathbf{n}}$. We can infer

$$\emptyset \vdash^{\{\mathbf{n},\mathbf{b},\top\}}_{\text{fix2}} \lambda\mathtt{x}.7 : \mathtt{Int}^{\mathbf{n}} \rightarrow \mathtt{Int}^{\mathbf{n}}$$

but this does not suffice for using the rule [fix1] to infer

$$\emptyset \vdash^{\{\mathbf{n},\mathbf{b},\top\}}_{\text{fix2}} \mathtt{fix}\ (\lambda\mathtt{x}.7) : \mathtt{Int}^{\mathbf{n}}$$

because $\text{BOT}_{\text{ST}}(\mathtt{Int}^{\mathbf{n}})$ fails. However we can infer

$$\emptyset \vdash_{\text{ST}} \lambda\mathtt{x}.7 : \mathtt{Int}^{\top} \rightarrow \mathtt{Int}^{\mathbf{n}}$$

and we can then apply the rule [fix2] to get the desired type. This argument shows

$$\neg(\text{A} \vdash^{\{\mathbf{n},\mathbf{b},\top\}}_{\text{fix2}} \mathtt{e} : \mathtt{t} \Rightarrow \text{A} \vdash^{\{\mathbf{n},\mathbf{b},\top\}}_{\text{fix1}} \mathtt{e} : \mathtt{t})$$

and thereby we have $\vdash^{\{\mathbf{n},\mathbf{b},\top\}}_{\text{fix1}} \subset \vdash^{\{\mathbf{n},\mathbf{b},\top\}}_{\text{fix2}}$.

Claim $\vdash^{\{\mathbf{n},\mathbf{b},\top\}}_{\text{fix2}} \subset \vdash^{\{\mathbf{n},\mathbf{b},\top\}}_{\text{fix}}$

To argue that $\vdash^{\{\mathbf{n},\mathbf{b},\top\}}_{\text{fix2}} \subset \vdash^{\{\mathbf{n},\mathbf{b},\top\}}_{\text{fix}}$ when we consider the full strictness and totality analysis we can use the same term and type as for showing $\vdash^{\{\mathbf{b},\top\}}_{\text{fix2}} \subset \vdash^{\{\mathbf{b},\top\}}_{\text{fix}}$.

x

2.3 Operational Semantics

The first step towards showing the analysis sound is to introduce the semantics. The semantics will be lazy except that all built-in functions will

$$[\text{app1}]\ \frac{\vdash \mathtt{e}_1 \Downarrow \lambda\mathtt{x}.\mathtt{e} \quad \vdash \mathtt{e}[\mathtt{e}_2/\mathtt{x}] \Downarrow \mathtt{v}}{\vdash \mathtt{e}_1\ \mathtt{e}_2 \Downarrow \mathtt{v}}$$

$$[\text{app2}]\ \frac{\vdash \mathtt{e}_1 \Downarrow \mathtt{c} \quad \vdash \mathtt{e}_2 \Downarrow \mathtt{v}}{\vdash \mathtt{e}_1\ \mathtt{e}_2 \Downarrow \mathtt{u}} \quad \text{if } (\mathtt{v}, \mathtt{u}) \in \delta(\mathtt{c})$$

$$[\text{fix}]\ \frac{\vdash \mathtt{e}\ (\mathtt{fix}\ \mathtt{e}) \Downarrow \mathtt{v}}{\vdash \mathtt{fix}\ \mathtt{e} \Downarrow \mathtt{v}}$$

$$[\text{abs}]\ \frac{}{\vdash \lambda\mathtt{x}.\mathtt{e} \Downarrow \lambda\mathtt{x}.\mathtt{e}} \qquad [\text{const}]\ \frac{}{\vdash \mathtt{c} \Downarrow \mathtt{c}}$$

$$[\text{condT}]\ \frac{\vdash \mathtt{e}_1 \Downarrow \mathtt{true} \quad \vdash \mathtt{e}_2 \Downarrow \mathtt{v}_2}{\vdash \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 \Downarrow \mathtt{v}_2}$$

$$[\text{condF}]\ \frac{\vdash \mathtt{e}_1 \Downarrow \mathtt{false} \quad \vdash \mathtt{e}_3 \Downarrow \mathtt{v}_3}{\vdash \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 \Downarrow \mathtt{v}_3}$$

Figure 2.5: Lazy Semantics for Closed Terms

be strict in each argument. Figure 2.5 defines a natural-style operational semantics [Plo77]. Terms are evaluated to WHNF, i.e. to constants or lambda-abstractions; we will let $\mathtt{u}$, $\mathtt{v}$, $\mathtt{c}$, and $\mathtt{f}$ be such WHNF's. The meaning of a constant $\mathtt{c}$ is given by a set $\delta(\mathtt{c})$ of pairs of constants; the idea is that if $(\mathtt{u}, \mathtt{v}) \in \delta(\mathtt{c})$ then $\mathtt{c}\ \mathtt{u} = \mathtt{v}$; e.g. $(\mathtt{2}, \mathtt{+}_2) \in \delta(\mathtt{+})$ and $(\mathtt{1}, \mathtt{3}) \in \delta(\mathtt{+}_2)$. As mentioned in the introduction to this Chapter the semantics is faithful to current lazy languages like Miranda [Tur85] and this is unlike other approaches (e.g. [BHA86]) where terms are evaluated to HNF rather than WHNF. As usual we shall regard α-equivalent terms to be equal.

Two closed terms are semantically equivalent, written $\mathtt{e}_1 \sim_{\mathtt{ut}} \mathtt{e}_2$, if they both evaluate to the same WHNF and have the same underlying type:

Definition 2.10
$(\mathtt{e}_1 \sim_{\mathtt{ut}} \mathtt{e}_2) \Leftrightarrow ((\vdash \mathtt{e}_1 \Downarrow \mathtt{v}) \Leftrightarrow (\vdash \mathtt{e}_2 \Downarrow \mathtt{v}))$
provided both $\emptyset \vdash \mathtt{e}_1 : \mathtt{ut}$ and $\emptyset \vdash \mathtt{e}_2 : \mathtt{ut}$ can be inferred. □

We shall assume throughout this Chapter that there are no empty types, i.e. for each underlying type there exists a *terminating* term with that type. Clearly, for each type there exists a non-terminating term of that type, for example $\mathtt{fix}\ (\lambda\mathtt{x}.\mathtt{x})$.

We shall write $\not\vdash \mathtt{e} \Downarrow$ to mean $\neg(\exists \mathtt{v} : \vdash \mathtt{e} \Downarrow \mathtt{v})$; this means that e does not terminate.

2.3.1 New Terms

For the proof of soundness of the conjunction inference system we find it helpful to introduce the terms $\mathtt{fix}_n\ \mathtt{e}$ where n is a number greater than or equal to 0. The idea is that n indicates how many times the fixpoint is allowed to be unfolded. So we need to expand the underlying type inference system and the semantics of the simply-typed λ-calculus. The underlying type of $\mathtt{fix}_n\ \mathtt{e}$ is the same as for fix e:

$$[\mathrm{fix}_n]\ \frac{\mathrm{A} \vdash \mathtt{e} : \mathtt{ut} \to \mathtt{ut}}{\mathrm{A} \vdash \mathtt{fix}_n\ \mathtt{e} : \mathtt{ut}}$$

and the semantics for $\mathtt{fix}_n\ \mathtt{e}$ is:

$$[\mathrm{fix}_n]\ \frac{\vdash \mathtt{e}\ (\mathtt{fix}_n\ \mathtt{e}) \Downarrow \mathtt{v}}{\vdash \mathtt{fix}_{n+1}\ \mathtt{e} \Downarrow \mathtt{v}}$$

There are no rules for $\mathtt{fix}_0\ \mathtt{e}$ and hence $\mathtt{fix}_0\ \mathtt{e}$ is stuck. We will allow the function ε to be applied to a term to remove all the annotations on fix. We do not allow the programmer to use $\mathtt{fix}_n$; hence there is no need for analysis of terms including $\mathtt{fix}_j$; it is merely a piece of syntax needed to facilitate the proof of the soundness theorem.

For proving the monotonicity-rule sound we need to construct a terminating term given any term e in such a way that the new term computes the same WHNF as e and terminates if e loops. However, this new term must also terminate when applied to a number of arguments. Consider the term $\lambda\mathtt{x.x}$ which evaluates to $\lambda\mathtt{x.x}$. Now we want that the new term associated with $\lambda\mathtt{x.x}$ applied to any argument terminates even if $\lambda\mathtt{x.x}$ applied to the same argument does not terminate. To achieve the we introduce the new terms $\mathcal{T}^n_{\mathtt{e}}$ where is e is a closed term without any $\mathtt{fix}_j$. The idea is that $\mathcal{T}^n_{\mathtt{e}}$ terminates when applied to $i \leq n$ arguments. The underlying type of $\mathcal{T}^n_{\mathtt{e}}$ is the same as for e:

$$[\mathcal{T}^n]\ \frac{\mathrm{A} \vdash \mathtt{e} : \mathtt{ut}}{\mathrm{A} \vdash \mathcal{T}^n_{\mathtt{e}} : \mathtt{ut}}$$

Let the arity of a standard type be 0 for base-types and for the function type, $\mathtt{ut}_1 \to \mathtt{ut}_2$, it is 1 plus the arity of $\mathtt{ut}_2$ and the *final result type* for a base-type $\mathtt{B}$ is $\mathtt{B}$ and for the function type, $\mathtt{ut}_1 \rightarrow \mathtt{ut}_2$, it is the final final result type of $\mathtt{ut}_2$. Now the semantics for $\mathcal{T}^n_{\mathtt{e}}$ is:

$$[\text{eval1}]\ \frac{\vdash \mathtt{e} \Downarrow \mathtt{v}}{\vdash \mathcal{T}^0_{\mathtt{e}} \Downarrow \mathtt{v}}$$

$$[\text{eval2}]\ \frac{\vdash \mathtt{e} \Downarrow \mathtt{v}}{\vdash \mathcal{T}^{n+1}_{\mathtt{e}} \Downarrow \mathcal{T}^{n+1}_{\mathtt{v}}}$$

$$[\text{eval3}]\ \frac{\not\vdash \mathtt{e} \Downarrow}{\vdash \mathcal{T}^n_{\mathtt{e}} \Downarrow \lambda \mathtt{x}_1 \ldots .\ \lambda \mathtt{x}_a . \mathtt{c}_{\mathtt{B}}} \quad \text{if} \begin{cases} \emptyset \vdash \mathtt{e} : \mathtt{ut} \\ a \text{ is the arity of } \mathtt{ut} \text{ and} \\ \mathtt{B} \text{ is the final result type of } \mathtt{ut} \end{cases}$$

$$[\mathcal{T}^n\text{app}]\ \frac{\vdash \mathtt{e}_1 \Downarrow \mathcal{T}^{n+1}_{\mathtt{v}} \quad \vdash \mathcal{T}^n_{\mathtt{v}\ \varepsilon(\mathtt{e}_2)} \Downarrow \mathtt{v}'}{\vdash \mathtt{e}_1\ \mathtt{e}_2 \Downarrow \mathtt{v}'}$$

where $\mathtt{c}_{\mathtt{B}}$ is a constant of type $\mathtt{B}$. Again the programmer is not allowed to use terms including $\mathcal{T}^n$; they are only introduced to be used in the soundness proof of the analysis.

The reason for not allowing terms to include $\mathtt{fix}_j$ inside the annotation on $\mathcal{T}^n$ is that otherwise monotonicity of evaluation will not be preserved. Consider Fact 2.15, below, and the term $\mathtt{fix}_6\ \mathtt{fac}\ \mathtt{7}$. We have

$$\not\vdash \mathtt{fix}_6\ \mathtt{fac}\ \mathtt{7} \Downarrow$$

and by the [eval3] rule we get

$$\vdash \mathcal{T}^0_{\mathtt{fix}_6\ \mathtt{fac}\ \mathtt{7}} \Downarrow \mathtt{c}_{\mathtt{Int}}$$

However we have

$$\vdash \varepsilon(\mathcal{T}^0_{\mathtt{fix}_6\ \mathtt{fac}\ \mathtt{7}}) \Downarrow \mathtt{5040}$$

2.3.2 Properties of the Semantics

Whenever $\mathtt{e}_1$ does not evaluate, then $\mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3$ cannot evaluate either:

Fact 2.11
$\not\vdash \mathtt{e}_1 \Downarrow \Rightarrow \not\vdash \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 \Downarrow$ □

Proof We assume $\nvdash \mathtt{e}_1 \Downarrow$. Assume that there exists a $\mathtt{v}'$ such that

$$\vdash \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 \Downarrow \mathtt{v}'$$

Then by the [condT]-rule we have $(\vdash \mathtt{e}_1 \Downarrow \mathtt{true})$ and $\vdash \mathtt{e}_2 \Downarrow \mathtt{v}'$ but this contradicts $\nvdash \mathtt{e}_1 \Downarrow$. Otherwise by the [condF]-rule we have $(\vdash \mathtt{e}_1 \Downarrow \mathtt{false})$ and $\vdash \mathtt{e}_3 \Downarrow \mathtt{v}'$ but this contradicts $\nvdash \mathtt{e}_1 \Downarrow$. So it must be the case that

$$\nvdash \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 \Downarrow$$

is true. ■

Whenever the function does not evaluate then the application cannot evaluate either:

Fact 2.12
$\nvdash \mathtt{e} \Downarrow \Rightarrow \nvdash \mathtt{e}\ \mathtt{e}' \Downarrow$ □

Proof We assume $\nvdash \mathtt{e} \Downarrow$ and we want to show $\nvdash \mathtt{e}\ \mathtt{e}' \Downarrow$. Now assume that it is not the case; that is assume $\exists\ \mathtt{v}' : \vdash \mathtt{e}\ \mathtt{e}' \Downarrow \mathtt{v}'$. Either [app1] or [app2] has been applied. From the [app1]-rule we get $\vdash \mathtt{e} \Downarrow \lambda\mathtt{x}.\mathtt{e}''$ and $\vdash \mathtt{e}''[\mathtt{e}'/\mathtt{x}] \Downarrow \mathtt{v}'$ but this contradicts the assumption $\nvdash \mathtt{e} \Downarrow$.

From the [app2]-rule we get $\vdash \mathtt{e} \Downarrow \mathtt{f}$, $\vdash \mathtt{e}' \Downarrow \mathtt{v}$, and $(\mathtt{v}, \mathtt{v}') \in \delta(\mathtt{f})$ but this also contradicts the assumption $\nvdash \mathtt{e} \Downarrow$. So it must be the case that $\nvdash \mathtt{e}\ \mathtt{e}' \Downarrow$. ■

Provided $\mathtt{e}_1$ and $\mathtt{e}_2$ are semantically equivalent, then $(\mathtt{e}_1\ \mathtt{e}')$ and $(\mathtt{e}_2\ \mathtt{e}')$ are semantically equivalent:

Fact 2.13
$(\mathtt{e}_1 \sim_{\mathtt{ut}_1 \to \mathtt{ut}_2} \mathtt{e}_2) \Rightarrow (\mathtt{e}_1\ \mathtt{e}' \sim_{\mathtt{ut}_2} \mathtt{e}_2\ \mathtt{e}')$ □

Proof First we show $(\emptyset \vdash \mathtt{e}_1\ \mathtt{e}' : \mathtt{ut}) \Leftrightarrow (\emptyset \vdash \mathtt{e}_2\ \mathtt{e}' : \mathtt{ut})$.

Next we show $(\vdash \mathtt{e}_1\ \mathtt{e}' \Downarrow \mathtt{v}') \Rightarrow (\vdash \mathtt{e}_2\ \mathtt{e}' \Downarrow \mathtt{v}')$. We do this by induction on the proof-tree of $\vdash \mathtt{e}_1\ \mathtt{e}' \Downarrow \mathtt{v}'$.

For the proof see Appendix page 226. ■

Fixpoints

The underlying types that can be inferred for a term $\mathtt{e}$ without any $\mathtt{fix}_n$'s can also be inferred for the term $\mathtt{e}'$ with $\mathtt{fix}_n$ replacing some occurrences of $\mathtt{fix}$ and vice versa:

Fact 2.14
$(\mathrm{A} \vdash \mathtt{e} : \mathtt{ut}) \Leftrightarrow (\mathrm{A} \vdash \varepsilon(\mathtt{e}) : \mathtt{ut})$ □

Proof First we assume that $(\mathrm{A} \vdash \mathtt{e} : \mathtt{ut})$ can be inferred and we show by induction on the proof-tree for $(\mathrm{A} \vdash \mathtt{e} : \mathtt{ut})$ that $(\mathrm{A} \vdash \varepsilon(\mathtt{e}) : \mathtt{ut})$ can be inferred.

The second part of bi-implication is analogous.

For the full details see Appendix page 229. ■

A fixpoint that can evaluate with n unfoldings can also evaluate if it is allowed to unfold an unlimited number of times:

Fact 2.15
$(\vdash \mathtt{e} \Downarrow \mathtt{v}) \Rightarrow (\vdash \varepsilon(\mathtt{e}) \Downarrow \varepsilon(\mathtt{v}))$ □

Proof We assume $\vdash \mathtt{e} \Downarrow \mathtt{v}$ and then we prove by induction on the proof-tree for $\vdash \mathtt{e} \Downarrow \mathtt{v}$ that $\vdash \varepsilon(\mathtt{e}) \Downarrow \varepsilon(\mathtt{v})$ can be inferred.

For the full details see Appendix page 232. ■

We now show that if $(\mathtt{fix}\ \mathtt{e})$ evaluates then there exists a number n such that $(\mathtt{fix}_n\ \mathtt{e})$ evaluates. In the proof of this result we need a way to modify some of the occurrences of $\mathtt{fix}$ in a term. For this we introduce the notion of tree-substitutions, π. They will tell us which occurrences of $\mathtt{fix}$ to replace with an occurrence of $\mathtt{fix}_j$.

Definition 2.16 *tree-substitution*
A tree-substitution π is a set of pairs of tree-addresses and a number. A tree-address is a list of 0, 1, 2.

For $n \in \{0, 1, 2\}$ let π^n be the part of the tree-substitution π where all the tree-addresses starts with an n but without this leading n, i.e.

$$\pi^n = \{(\mathrm{addr}, m) \mid (n : \mathrm{addr}, m) \in \pi\}$$

where "n : addr" denotes the list whose first element is n and whose tail is addr. Let $\pi + n$ be the tree-substitution

$$\pi + n = \{(\mathrm{addr}, m + n) \mid (\mathrm{addr}, m) \in \pi\}$$

Let $n\pi$ be the tree-substitution

$$n\pi = \{(n : \mathrm{addr}, m) \mid (\mathrm{addr}, m) \in \pi\}$$

and let $p\pi$ be the tree-substitution

$$\{(p \mathbin{++} \text{addr}, m) \mid (\text{addr}, m) \in \pi \wedge p \text{ is a tree-address}\}$$

where "++" denotes list concatenation.

The tree-substitution π applied to a term e is written $\texttt{[e]}^{\pi}$ and is defined inductively as follows:

$$\begin{array}{lcl}
\texttt{[x]}^{\pi} & = & \texttt{x} \\
\texttt{[c]}^{\pi} & = & \texttt{c} \\
\texttt{[if e}_1\texttt{ then e}_2\texttt{ else e}_3\texttt{]}^{\pi} & = & \texttt{if [e}_1\texttt{]}^{\pi^0}\texttt{ then [e}_2\texttt{]}^{\pi^1}\texttt{ else [e}_3\texttt{]}^{\pi^2} \\
\texttt{[e}_1\texttt{ e}_2\texttt{]}^{\pi} & = & \texttt{[e}_1\texttt{]}^{\pi^0}\ \texttt{[e}_2\texttt{]}^{\pi^1} \\
\texttt{[}\lambda\texttt{x.e]}^{\pi} & = & \lambda\texttt{x.[e]}^{\pi^0} \\
\texttt{[fix e]}^{\pi} & = & \begin{cases} \texttt{fix}_n\ \texttt{[e]}^{\pi^0}, & \text{if } ([\,], n) \in \pi \\ \texttt{fix [e]}^{\pi^0}, & \text{otherwise} \end{cases} \\
\texttt{[}\mathcal{T}^n_{\texttt{e}}\texttt{]}^{\pi} & = & \mathcal{T}^n_{\texttt{e}}
\end{array}$$

where [ ] denotes the empty list. □

Proportion 2.17
For e without any $\texttt{fix}_j$ we have

$$\vdash \texttt{e} \Downarrow \texttt{v} \;\Rightarrow\; \begin{cases} \forall\pi\exists m\exists\pi'\forall n \geq 0 : \\ \quad (\vdash \texttt{[e]}^{\pi+m+n} \Downarrow \texttt{[v]}^{\pi'+n}) \wedge \\ \quad ((\vdash \texttt{[e]}^{\pi} \Downarrow \texttt{v}') \Rightarrow m = 0) \end{cases} \qquad \square$$

The idea is that for any labelling of the fixpoints in a term, m is the minimal number to be added so that the term can evaluate. The number n indicates that whenever a labelling of the fixpoints will let the term evaluate, then increasing the labels it will still let the term evaluate; this is stated in Corollary 2.19 below.

Proof We assume $\vdash \texttt{e} \Downarrow \texttt{v}$ and that e is without any $\texttt{fix}_j$; then we prove by induction in the proof-tree for $\vdash \texttt{e} \Downarrow \texttt{v}$ that

$$\begin{array}{l} \forall\pi\exists m\exists\pi'\forall n \geq 0 : \\ \quad (\vdash \texttt{[e]}^{\pi+m+n} \Downarrow \texttt{[v]}^{\pi'+n}) \wedge ((\vdash \texttt{[e]}^{\pi} \Downarrow \texttt{v}') \Rightarrow m = 0) \end{array}$$

holds.

For the full details see Appendix page 235. ■

Corollary 2.18
$(\vdash \mathtt{fix}\ \mathtt{e} \Downarrow \mathtt{v}) \Rightarrow (\exists m \exists \mathtt{v}' : \vdash \mathtt{fix}_m\ \mathtt{e} \Downarrow \mathtt{v}')$ provided $\mathtt{e}$ is without any $\mathtt{fix}_j$.
□

Proof Use Proposition 2.17 with $\pi = \{([\,], 0)\}$ and $n = 0$. ■

A fixpoint that can evaluate with k unfoldings can also evaluate if it is allowed to unfold $k+1$ times:

Corollary 2.19
$(\vdash \mathtt{fix}_k\ \mathtt{e} \Downarrow \mathtt{v}) \Rightarrow (\exists \mathtt{v}' : \vdash \mathtt{fix}_{k+1}\ \mathtt{e} \Downarrow \mathtt{v}')$ provided $\mathtt{e}$ is without any $\mathtt{fix}_j$.
□

Proof Use Proposition 2.17 with $\pi = \{([\,], k)\}$ and $n = 1$ and observe that $m = 0$. ■

Terminating Terms

Suppose that a term $\mathtt{e}$ applied to some terms does indeed evaluate; we now consider to which term $\mathcal{T}^n_{\varepsilon(\mathtt{e})}$ evaluates when applied to the same terms.

Lemma 2.20
Given $1 \le i \le n \le a$ where a is the arity of $\mathtt{e}$:
$$(\vdash \mathtt{e}\ \mathtt{e}_1\ \ldots\ \mathtt{e}_i \Downarrow \mathtt{v}) \Rightarrow (\vdash \mathcal{T}^n_{\varepsilon(\mathtt{e})}\ \mathtt{e}_1\ \ldots\ \mathtt{e}_i \Downarrow \begin{cases} \varepsilon(\mathtt{v}), & \text{if } n = i \\ \mathcal{T}^{n-i}_{\varepsilon(\mathtt{v})}, & \text{otherwise} \end{cases})$$
□

Proof The lemma is shown by induction on i. ■

Next suppose that a term $\mathtt{e}$ does not evaluate when applied to certain terms; we now consider what happens for $\mathcal{T}^n_{\varepsilon(\mathtt{e})}$ when applied to the same terms.

Lemma 2.21
Given $1 \le i \le n \le a$ where a is the arity of $\mathtt{e}$:
$$(\not\vdash \mathtt{e}\ \mathtt{e}_1\ \ldots\ \mathtt{e}_i \Downarrow) \Rightarrow (\exists\ \mathtt{v}' : (\vdash \mathcal{T}^n_{\varepsilon(\mathtt{e})}\ \mathtt{e}_1\ \ldots\ \mathtt{e}_i \Downarrow \mathtt{v}'))$$
□

Proof We observe that either $\vdash \varepsilon(\mathtt{e}\ \mathtt{e}_1\ \ldots\ \mathtt{e}_i) \Downarrow \mathtt{v}'$ or $\not\vdash \varepsilon(\mathtt{e}\ \mathtt{e}_1\ \ldots\ \mathtt{e}_i) \Downarrow$ must be the case. In the first case we use Lemma 2.20. In the second case we use the rules [eval2] and [eval3]. ■

Finally, from the proof-tree for the term $(\mathtt{e}\ \mathtt{e}'\ \mathtt{e}_1\ \ldots\ \mathtt{e}_k)$ we can construct a proof-tree for the term $(\mathtt{e}\ \mathcal{T}^n_{\varepsilon(\mathtt{e}')}\ \mathtt{e}_1\ \ldots\ \mathtt{e}_k)$:

Lemma 2.22
$(\vdash \mathtt{e}\ \mathtt{e}'\ \mathtt{e}_1\ \ldots\ \mathtt{e}_k \Downarrow \mathtt{v}) \Rightarrow \exists\, \mathtt{v}' : \vdash \mathtt{e}\ \mathcal{T}^n_{\varepsilon(\mathtt{e}')}\ \mathtt{e}_1\ \ldots\ \mathtt{e}_k \Downarrow \mathtt{v}'$ □

Proof In this proof we regard a proof-tree as having its root at the bottom. For the proof we assume that $\vdash \mathtt{e}\ \mathtt{e}'\ \mathtt{e}_1\ \ldots\ \mathtt{e}_k \Downarrow \mathtt{v}$ and we prove that $\vdash \mathtt{e}\ \mathcal{T}^n_{\varepsilon(\mathtt{e}')}\ \mathtt{e}_1\ \ldots\ \mathtt{e}_k \Downarrow \mathtt{v}'$. We do this by first constructing a template for the given proof-tree and then later use this template to construct the desired proof-tree. For an example see Example 2.23 below.

To construct the template we first remove the parts of the proof-tree that are above certain nodes by traversing the given proof-tree in a left-most-top-first manner. Let $\mathtt{u}_0$ be $\mathtt{e}'$. Now remove the parts of the proof-tree that are above the nodes of the form:

- $\vdash \mathtt{u}_i \Downarrow \mathtt{u}_{i+1}$ with no nodes below that is $\mathtt{u}_i$ applied to a number of terms. For later use we let k_i be 0 in this case.
- $\vdash \mathtt{u}_i\ \mathtt{e}'_1\ \ldots\ \mathtt{e}'_{k_i} \Downarrow \mathtt{u}_{i+1}$ with no nodes below that is $\mathtt{u}_i$ applied to a greater number of terms.

We continue in this way until there are no more parts of the proof-tree that can be removed.

The template can be constructed by copying all nodes from the proof-tree resulting from the above process. However, in nodes involving any $\mathtt{u}_i$ we replace $\mathtt{u}_i$ with a pointer to the pair

$$(\mathtt{u}_i, \begin{cases} \varepsilon(\mathtt{u}_i), & \text{if } n \leq k_0 + \cdots + k_i \\ \mathcal{T}^{n-k_0-\cdots-k_i}_{\varepsilon(\mathtt{u}_i)}, & \text{otherwise} \end{cases})$$

Now note that the proof-tree for $\vdash \mathtt{e}\ \mathtt{e}'\ \mathtt{e}_1\ \ldots\ \mathtt{e}_k \Downarrow \mathtt{v}$ may be constructed from the template by extracting the first component of the pairs and then constructing the top parts of the tree. In a similar way the proof-tree for $\vdash \mathtt{e}\ \mathcal{T}^n_{\varepsilon(\mathtt{e}')}\ \mathtt{e}_1\ \ldots\ \mathtt{e}_k \Downarrow \mathtt{v}$ is constructed by extracting the second component of the pairs and using Lemma 2.20 to construct the top-parts of the tree. ■

Example 2.23
Consider the term $\texttt{e}\ \texttt{e}'$, where

$$\begin{array}{rcl} \texttt{e} & = & \lambda \texttt{x}.+ (+\ 1\ 2)\ (\texttt{x}\ 2) \\ \texttt{e}' & = & \lambda \texttt{y}.\times\ \texttt{y}\ 4 \end{array}$$

The full evaluation-tree is:

$$\dfrac{\vdash \texttt{e} \Downarrow \texttt{e} \qquad \dfrac{\dfrac{\dfrac{}{\vdash \texttt{+} \Downarrow \texttt{+}} \quad P_1 \quad \texttt{+}_3 \in \delta(\texttt{+}, \texttt{3})}{\vdash \texttt{+ (+ 1 2)} \Downarrow \texttt{+}_3} \qquad \dfrac{\dfrac{}{\vdash \texttt{e}' \Downarrow \texttt{e}'} \quad P_2}{\vdash \texttt{e}'\ 2 \Downarrow \texttt{8}} \qquad \texttt{11} \in \delta((\texttt{+}_3, 8))}{\vdash (+\ (+\ 1\ 2)\ (\texttt{x}\ 2))[\texttt{e}'/\texttt{x}] \Downarrow \texttt{11}}}{\vdash \texttt{e}\ \texttt{e}' \Downarrow \texttt{11}}$$

where P_1 is

$$\dfrac{\dfrac{\dfrac{}{\vdash \texttt{+} \Downarrow \texttt{+}} \quad \dfrac{}{\vdash \texttt{1} \Downarrow \texttt{1}} \quad \texttt{+}_1 \in \delta(\texttt{+}, \texttt{1})}{\vdash \texttt{+ 1} \Downarrow \texttt{+}_1} \qquad \dfrac{}{\vdash \texttt{2} \Downarrow \texttt{2}} \quad \texttt{3} \in \delta(\texttt{+}_1, \texttt{2})}{\vdash \texttt{+ 1 2} \Downarrow \texttt{3}}$$

and P_2 is

$$\dfrac{\dfrac{\dfrac{}{\vdash \times \Downarrow \times} \quad \dfrac{}{\vdash \texttt{2} \Downarrow \texttt{2}} \quad \times_2 \in \delta(\times, \texttt{2})}{\vdash \times\ \texttt{2} \Downarrow \times_2} \qquad \dfrac{}{\vdash \texttt{4} \Downarrow \texttt{4}} \quad \texttt{8} \in \delta(\times_2, \texttt{4})}{\vdash (\times\ \texttt{y}\ 4)[2/\texttt{y}] \Downarrow \texttt{8}}$$

First we remove the part of the tree that is above $\vdash \texttt{e}'\ 2 \Downarrow \texttt{8}$ and it looks like

$$\dfrac{\vdash \texttt{e} \Downarrow \texttt{e} \qquad \dfrac{\dfrac{\dfrac{}{\vdash \texttt{+} \Downarrow \texttt{+}} \quad P_1 \quad \texttt{+}_3 \in \delta(\texttt{+}, \texttt{3})}{\vdash \texttt{+ (+ 1 2)} \Downarrow \texttt{+}_3} \qquad \dfrac{}{\vdash \texttt{e}'\ 2 \Downarrow \texttt{8}} \qquad \texttt{11} \in \delta((\texttt{+}_3, 8))}{\vdash (+\ (+\ 1\ 2)\ (\texttt{x}\ 2))[\texttt{e}'/\texttt{x}] \Downarrow \texttt{11}}}{\vdash \texttt{e}\ \texttt{e}' \Downarrow \texttt{11}}$$

and we have

$$\begin{array}{rcl} u_0 & = & \texttt{e}' \\ u_1 & = & \texttt{8} \\ u_2 & = & \texttt{11} \end{array}$$

$$
\begin{aligned}
k_0 &= 1 \\
k_1 &= 0 \\
k_2 &= 0
\end{aligned}
$$

Now the template is:

$$
\dfrac{\vdash \mathtt{e} \Downarrow \mathtt{e} \qquad \dfrac{\dfrac{\dfrac{}{\vdash \mathtt{+} \Downarrow \mathtt{+}} \quad \mathrm{P}_1 \quad \mathtt{+}_3 \in \delta(\mathtt{+},\ \mathtt{3})}{\vdash \mathtt{+\ (+\ 1\ 2)} \Downarrow \mathtt{+}_3} \qquad \dfrac{}{\vdash p_0\ 2 \Downarrow p_1} \qquad p_2 \in \delta((\mathtt{+}_3,\ p_1))}{\vdash (+\ (+\ 1\ 2)\ (\mathtt{x}\ 2))[p_0/\mathtt{x}] \Downarrow p_2}}{\vdash \mathtt{e}\ p_0 \Downarrow p_2}
$$

where

$$
\begin{aligned}
p_0 &= (\mathtt{e}', \begin{cases} \mathtt{e}', & \text{if } n \leq 1 \\ \mathcal{T}^{n-1}_{\mathtt{e}'}, & \text{otherwise} \end{cases}) \\
p_1 &= (\mathtt{8}, \begin{cases} \mathtt{8}, & \text{if } n \leq 1 \\ \mathcal{T}^{n-1}_{\mathtt{8}} s, & \text{otherwise} \end{cases}) \\
p_2 &= (\mathtt{11}, \begin{cases} \mathtt{11}, & \text{if } n \leq 1 \\ \mathcal{T}^{n-1}_{\mathtt{11}}, & \text{otherwise} \end{cases})
\end{aligned}
$$

Whenever we want a proof-tree for $\mathtt{e}\ \mathtt{e}'$ we use the first component of the pairs and when we want a proof-tree for $\mathtt{e}\ \mathcal{T}^{n}_{\mathtt{e}'}$ we use the second component of the pairs.

□

2.4 Soundness

Our task is now to prove that the inference system of Figure 2.3 is sound with respect to the natural-style operational semantics of Figure 2.5. First we define a predicate $\models \mathtt{e} : \mathtt{ct}$ stating that the term $\mathtt{e}$ is valid of conjunction type $\mathtt{ct}$. Then we show some useful lemmas and finally we can prove the soundness result: if $\mathrm{A} \vdash_{\mathrm{ST}} \mathtt{e} : \mathtt{ct}$ then $\models \mathtt{e}[\overline{\mathtt{v}}/\overline{\mathtt{x}}] : \mathtt{ct}$ for all closed substitutions $[\overline{\mathtt{v}}/\overline{\mathtt{x}}]$ that are valid of the types in A.

The validity predicate is shown in Figure 2.6. The term $\mathtt{e}$ is valid of conjunction type $\mathtt{ct}_1 \wedge \mathtt{ct}_2$ if $\mathtt{e}$ is valid of type $\mathtt{ct}_1$ as well as $\mathtt{ct}_2$. That the

(I)	$(\models \mathtt{e} : \mathtt{ct}_1 \wedge \mathtt{ct}_2)$	$\Leftrightarrow$	$(\models \mathtt{e} : \mathtt{ct}_1) \wedge (\models \mathtt{e} : \mathtt{ct}_2)$
(II)	$(\models \mathtt{e} : \mathtt{ut}^{\mathbf{b}})$	$\Leftrightarrow$	$(\forall \mathrm{v}\colon \not\vdash \mathtt{e} \Downarrow) \wedge (\emptyset \vdash \mathtt{e} : \mathtt{ut})$
(III)	$(\models \mathtt{e} : \mathtt{ut}^{\mathbf{n}})$	$\Leftrightarrow$	$(\exists \mathrm{v}\colon \vdash \mathtt{e} \Downarrow \mathtt{v}) \wedge (\emptyset \vdash \mathtt{e} : \mathtt{ut})$
(IV)	$(\models \mathtt{e} : \mathtt{ut}^{\top})$	$\Leftrightarrow$	$(\emptyset \vdash \mathtt{e} : \mathtt{ut})$
(V)	$(\models \mathtt{e} : \mathtt{t}_1 \rightarrow \mathtt{t}_2)$	$\Leftrightarrow$	$(\forall \mathrm{e}'\colon (\models \mathtt{e}' : \mathtt{t}_1) \Rightarrow (\models \mathtt{e}\ \mathtt{e}' : \mathtt{t}_2))$ $\wedge\ (\emptyset \vdash \mathtt{e} : \varepsilon(\mathtt{t}_1) \rightarrow \varepsilon(\mathtt{t}_2))$

Figure 2.6: The definition of validity

term $\mathtt{e}$ has a WHNF and the underlying type $\mathtt{ut}$ amounts to $\models \mathtt{e} : \mathtt{ut}^{\mathbf{n}}$ being true; that $\mathtt{e}$ has no WHNF but has the underlying type $\mathtt{ut}$ amounts to $\models \mathtt{e} : \mathtt{ut}^{\mathbf{b}}$ being true (i.e. there exists no WHNF, $\mathtt{v}$, such that $\vdash \mathtt{e} \Downarrow \mathtt{v}$). A term with conjunction type $\mathtt{ut}^{\top}$ just has to be of the underlying type $\mathtt{ut}$, as we do not know anything about the evaluation of the term. A term $\mathtt{e}$ is valid of function type $\mathtt{t}_1 \rightarrow \mathtt{t}_2$ if for any other term $\mathtt{e}'$ that is valid of strictness and totality type $\mathtt{t}_1$, also $\mathtt{e}$ applied to $\mathtt{e}'$ will be valid of strictness and totality type $\mathtt{t}_2$.

To prepare for the soundness of the conjunction type inference system we first need to bind all the free variables in the term. Let $\overline{\mathtt{x}}$ be the list of variables in A, let $\overline{\mathtt{t}}$ be the list of the strictness and totality types corresponding to the variables $\overline{\mathtt{x}}$, and let $\overline{\mathtt{v}}$ be a list of closed terms that are valid of the types $\overline{\mathtt{t}}$, i.e. $\models \overline{\mathtt{v}} : \overline{\mathtt{t}}$. We now define $\models \overline{\mathtt{v}} : \overline{\mathtt{t}}$ inductively by

$$\begin{aligned} \models (\mathtt{v}, \overline{\mathtt{v}}) : (\mathtt{t}, \overline{\mathtt{t}}) &= (\models \mathtt{v} : \mathtt{t}) \wedge (\models \overline{\mathtt{v}} : \overline{\mathtt{t}}) \\ \models [\,] : [\,] &= \mathtt{tt} \end{aligned}$$

The substitution $[[\overline{\mathtt{v}}/\overline{\mathtt{x}}]]$ is defined inductively by

$$\begin{aligned} \mathtt{e}[(\mathtt{v}, \overline{\mathtt{v}})/(\mathtt{x}, \overline{\mathtt{x}})] &= (\mathtt{e}[\mathtt{v}/\mathtt{x}])[[\overline{\mathtt{v}}/\overline{\mathtt{x}}]] \\ \mathtt{e}[[\,]/[\,]] &= \mathtt{e} \end{aligned}$$

Theorem 2.24 *Soundness*

For expressions $\mathtt{e}$ without any $\mathtt{fix}_n$ and $\mathcal{T}^n$ we have

$$(\overline{\mathtt{x}} : \overline{\mathtt{t}} \vdash_{\mathrm{ST}} \mathtt{e} : \mathtt{ct}) \Rightarrow (\forall\, \overline{\mathtt{v}}\colon (\models \overline{\mathtt{v}} : \overline{\mathtt{t}}) \Rightarrow (\models \mathtt{e}[\overline{\mathtt{v}}/\overline{\mathtt{x}}] : \mathtt{ct})). \quad \square$$

Before we prove the soundness theorem we need some facts and lemmas. They are divided into three groups: first we show one property of the underlying type system, then we show some properties of the analysis and finally we show some properties of the validity predicate.

2.4.1 Properties of the Standard Type System

For a free variable $\mathtt{x}$ in a term $\mathtt{e}$ we can substitute terms $\mathtt{e}'$ with the type indicated by the type environment A for $\mathtt{x}$. The terms $\mathtt{e}'$ do not have to be closed but may only use the same free variables as $\mathtt{e}$ except for $\mathtt{x}$.

Lemma 2.25
$((\mathrm{A} \vdash \mathtt{e} : \mathtt{ut}_2) \wedge (\mathrm{A}_{\mathtt{X}} \vdash \mathtt{e}_2 : \mathtt{ut}_1) \wedge (\mathtt{x} : \mathtt{ut}_1 \in \mathrm{A})) \Rightarrow (\mathrm{A}_{\mathtt{X}} \vdash \mathtt{e}[\mathtt{e}_2/\mathtt{x}] : \mathtt{ut}_2)$
$\square$

Proof We assume $\mathrm{A} \vdash \mathtt{e} : \mathtt{ut}_2$, $\mathrm{A}_{\mathtt{X}} \vdash \mathtt{e}_2 : \mathtt{ut}_1$, and that $\mathtt{x} : \mathtt{ut}_1$ is in A. Then we proof by induction in the proof-tree for $\mathrm{A} \vdash \mathtt{e} : \mathtt{ut}_2$ that $\mathrm{A}_{\mathtt{X}} \vdash \mathtt{e}[\mathtt{e}_2/\mathtt{x}] : \mathtt{ut}_2$ can be inferred.

For the full details see Appendix page 249. ■

2.4.2 Properties of the Conjunction Type System

Two conjunction types can only be compared if they have the same underlying type:

Fact 2.26
$(\mathtt{t}_1 \leq_{\mathrm{ST}} \mathtt{t}_2) \Rightarrow (\varepsilon(\mathtt{t}_1) = \varepsilon(\mathtt{t}_2))$

$(\mathtt{ct}_1 \leq_{\mathrm{CT}} \mathtt{ct}_2) \Rightarrow (\varepsilon(\mathtt{ct}_1) = \varepsilon(\mathtt{ct}_2))$ $\square$

Proof We assume $\mathtt{t}_1 \leq_{\mathrm{ST}} \mathtt{t}_2$ and then we prove by induction on the proof-tree for $\mathtt{t}_1 \leq_{\mathrm{ST}} \mathtt{t}_2$ that $\varepsilon(\mathtt{t}_1) = \varepsilon(\mathtt{t}_2)$ holds.

$(\mathtt{ct}_1 \leq_{\mathrm{CT}} \mathtt{ct}_2) \Rightarrow (\varepsilon(\mathtt{ct}_1) = \varepsilon(\mathtt{ct}_2))$

For the full details see Appendix page 254. ■

Some properties of the function $\downarrow$ are expressed by Fact 2.27:

Fact 2.27
The function $\downarrow$ has the following properties:

a) $\mathtt{t} \leq_{\mathrm{ST}} \downarrow\mathtt{t}$

b) $\downarrow(\downarrow\mathtt{t}) = \downarrow\mathtt{t}$

c) $(\mathtt{t}_1 \leq_{\mathrm{ST}} \mathtt{t}_2) \Rightarrow (\downarrow\mathtt{t}_1 \leq_{\mathrm{ST}} \downarrow\mathtt{t}_2)$

d) $\mathrm{BOT}_{\mathrm{ST}}(\downarrow\mathtt{t}) = \mathtt{tt}$

e) $((\mathtt{t} \leq_{\mathrm{ST}} \mathtt{t}') \wedge (\mathrm{BOT}_{\mathrm{ST}}(\mathtt{t}')) \Rightarrow (\downarrow\mathtt{t} \leq_{\mathrm{ST}} \mathtt{t}')$

□

Note that **e)** expresses that $\downarrow\mathtt{t}$ is the smallest type such that both $\mathtt{t} \leq_{\mathrm{ST}} \downarrow\mathtt{t}$ and $\mathrm{BOT}_{\mathrm{ST}}(\downarrow\mathtt{t})$ holds.

Proof

Part a) We show $\mathtt{t} \leq_{\mathrm{ST}} \downarrow\mathtt{t}$ by induction on $\mathtt{t}$.

Part b) We show $\downarrow\downarrow\mathtt{t} = \downarrow\mathtt{t}$ by induction on $\mathtt{t}$.

Part c) We assume $\mathtt{t}_1 \leq_{\mathrm{ST}} \mathtt{t}_2$ and show

$$\downarrow\mathtt{t}_1 \leq_{\mathrm{ST}} \downarrow\mathtt{t}_2$$

by induction on the proof-tree for $\mathtt{t}_1 \leq_{\mathrm{ST}} \mathtt{t}_2$.

Part d) We show $\mathrm{BOT}_{\mathrm{ST}}(\downarrow\mathtt{t})$ by induction on the strictness and totality type $\mathtt{t}$.

Part e) We assume $\mathtt{t}' \leq_{\mathrm{ST}} \mathtt{t}$ and that $\mathrm{BOT}_{\mathrm{ST}}(\mathtt{t}')$ is true, then we show by induction on $\mathtt{t}'$ that $\downarrow\mathtt{t} \leq_{\mathrm{ST}} \mathtt{t}'$ can be inferred.

For the full details see Appendix page 258. ■

Provided $\mathrm{BOT}_{\mathrm{CT}}(\mathtt{ct})$ is true, then the conjunction type $\mathtt{ct}$ is greater than $\varepsilon(\mathtt{ct})^{\mathbf{b}}$.

Lemma 2.28
$(\mathrm{BOT}_{\mathrm{CT}}(\mathtt{ct}) = \mathtt{tt}) \Leftrightarrow (\varepsilon(\mathtt{ct})^{\mathbf{b}} \leq_{\mathrm{CT}} \mathtt{ct})$ □

Proof First we assume $\mathrm{BOT}_{\mathrm{CT}}(\mathtt{ct}) = \mathtt{tt}$ and then we show by induction on the type $\mathtt{ct}$ that $\varepsilon(\mathtt{ct})^{\mathbf{b}} \leq_{\mathrm{CT}} \mathtt{ct}$ can be inferred. Second we assume $\varepsilon(\mathtt{ct})^{\mathbf{b}} \leq_{\mathrm{CT}} \mathtt{ct}$ and then we show by induction the type $\mathtt{ct}$ that $\mathrm{BOT}_{\mathrm{CT}}(\mathtt{ct})$ is true.

For the full details see Appendix page 265. ■

2.4.3 Properties of the Validity Predicate

The term $\mathcal{T}^0_{\mathtt{e}}$ always terminates:

Lemma 2.29
$(\models \mathtt{e} : \mathtt{ut}^{\top}) \Rightarrow (\models \mathcal{T}^0_{\varepsilon(\mathtt{e})} : \mathtt{ut}^{\mathbf{n}})$ □

Proof We assume $\models \mathtt{e} : \mathtt{ut}^{\top}$. There are now two possibilities: either $\vdash \mathtt{e} \Downarrow \mathtt{v}$ or $\not\vdash \mathtt{e} \Downarrow$. First assume $\vdash \mathtt{e} \Downarrow \mathtt{v}$. From Fact 2.15 we have $\vdash \varepsilon(\mathtt{e}) \Downarrow \varepsilon(\mathtt{v})$ and from the rule [eval1] we get $\vdash \mathcal{T}^0_{\varepsilon(\mathtt{e})} \Downarrow \varepsilon(\mathtt{v})$ hence we have $(\models \mathcal{T}^0_{\varepsilon(\mathtt{e})} : \mathtt{ut}^{\mathbf{n}})$.

Secondly assume $\not\vdash \mathtt{e} \Downarrow$. Now it must either be the case that $\vdash \varepsilon(\mathtt{e}) \Downarrow \mathtt{v}$ or $\not\vdash \varepsilon(\mathtt{e}) \Downarrow$. In the first case we do as above and we have $(\models \mathcal{T}^0_{\varepsilon(\mathtt{e})} : \mathtt{ut}^{\mathbf{n}})$. In the second case we apply the rule [eval3] to get $\vdash \mathcal{T}^0_{\varepsilon(\mathtt{e})} \Downarrow \lambda \mathtt{x}_1 \ldots \lambda \mathtt{x}_a . \mathtt{c}_{\mathsf{B}}$ where a is the arity of $\mathtt{ut}$ and B is the final result type of $\mathtt{ut}$. We now have $(\models \mathcal{T}^0_{\varepsilon(\mathtt{e})} : \mathtt{ut}^{\mathbf{n}})$ as required. ■

The term $\mathcal{T}^n_{\mathtt{e}}$ applied to n terms will always terminate:

Lemma 2.30
$(\models \mathtt{e} : \mathtt{t}_1 \to \ldots \mathtt{t}_n \to \mathtt{ut}^{\top}) \Rightarrow (\models \mathcal{T}^n_{\varepsilon(\mathtt{e})} : \mathtt{t}_1 \to \ldots \mathtt{t}_n \to \mathtt{ut}^{\mathbf{n}})$ □

Proof We assume $(\models \mathtt{e} : \mathtt{t}_1 \to \ldots \mathtt{t}_n \to \mathtt{ut}^{\top})$. We want to show

$$\models \mathcal{T}^n_{\varepsilon(\mathtt{e})} : \mathtt{t}_1 \to \ldots \mathtt{t}_n \to \mathtt{ut}^{\mathbf{n}}$$

which is equivalent to showing

$$\forall \mathtt{e}_1 \ldots \mathtt{e}_n : (\models \mathtt{e}_1 : \mathtt{t}_1 \ldots \models \mathtt{e}_n : \mathtt{t}_n) \Rightarrow (\models \mathcal{T}^n_{\varepsilon(\mathtt{e})}\ \mathtt{e}_1 \ldots \mathtt{e}_n : \mathtt{ut}^{\mathbf{n}})$$

We have $\models \mathtt{e} : \mathtt{t}_1 \to \ldots \to \mathtt{t}_n \to \mathtt{ut}^{\top}$. Now either $\vdash \mathtt{e}\ \mathtt{e}_1 \ldots \mathtt{e}_n \Downarrow \mathtt{v}$ or $\not\vdash \mathtt{e}\ \mathtt{e}_1 \ldots \mathtt{e}_n \Downarrow$ holds. In the first case we apply Lemma 2.20 and then have $\vdash \mathcal{T}^n_{\varepsilon(\mathtt{e})}\ \mathtt{e}_1 \ldots \mathtt{e}_n \Downarrow \mathtt{v}'$. In the second case we apply Lemma 2.21 and then have $\vdash \mathcal{T}^n_{\varepsilon(\mathtt{e})}\ \mathtt{e}_1 \ldots \mathtt{e}_n \Downarrow \mathtt{v}'$. In both cases we have

$$\vdash \mathcal{T}^n_{\varepsilon(\mathtt{e})}\ \mathtt{e}_1 \ldots \mathtt{e}_n \Downarrow \mathtt{v}'$$

so it must be the case that $\models \mathcal{T}^n_{\varepsilon(\mathtt{e})}\ \mathtt{e}_1 \ldots \mathtt{e}_n : \mathtt{ut}^{\mathbf{n}}$ holds. ■

Semantic equivalence is lifted to conjunction types:

Lemma 2.31
$((\models \mathtt{e}_1 : \mathtt{ct}) \wedge (\mathtt{e}_1 \sim_{\varepsilon(\mathtt{ct})} \mathtt{e}_2)) \Rightarrow (\models \mathtt{e}_2 : \mathtt{ct})$ □

Proof We assume $\models \mathtt{e}_1 : \mathtt{ct}$ and $\mathtt{e}_1 \sim_{\varepsilon(\mathtt{ct})} \mathtt{e}_2$ then we show by induction in the type $\mathtt{ct}$ that $\models \mathtt{e}_2 : \mathtt{ct}$ is true. In all the case we know that both $\emptyset \vdash \mathtt{e}_1 : \varepsilon(\mathtt{ct})$ and $\emptyset \vdash \mathtt{e}_2 : \varepsilon(\mathtt{ct})$ can be inferred.

For the full details see Appendix page 268. ■

The Facts 2.32, 2.33, 2.34, 2.35 are applications of Lemma 2.31. They are all proven by showing that the two terms are semantically equivalent and then applying Lemma 2.31.

Fact 2.32
$(\models (\lambda \mathtt{x}.\mathtt{e})\ \mathtt{e}' : \mathtt{ct}) \Leftrightarrow (\models \mathtt{e}[\mathtt{e}'/\mathtt{x}] : \mathtt{ct})$ □

Proof For the proof see Appendix page 270. ■

Fact 2.33
We have

$$(\models \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ (\mathtt{e}_2\ \mathtt{e}')\ \mathtt{else}\ (\mathtt{e}_3\ \mathtt{e}') : \mathtt{ct})$$
$$\Updownarrow$$
$$(\models (\mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3)\ \mathtt{e}' : \mathtt{ct})$$

□

Proof For the proof see Appendix page 272. ■

Unfolding $\mathtt{fix}_n$ or $\mathtt{fix}$ does not change validity:

Fact 2.34
$(\models \mathtt{e}\ (\mathtt{fix}_n\ \mathtt{e}) : \mathtt{ct}) \Leftrightarrow (\models \mathtt{fix}_{n+1}\ \mathtt{e} : \mathtt{ct})$ □

Proof For the proof see Appendix page 275. ■

Fact 2.35
$(\models \mathtt{e}\ (\mathtt{fix}\ \mathtt{e}) : \mathtt{ct}) \Leftrightarrow (\models \mathtt{fix}\ \mathtt{e} : \mathtt{ct})$ □

Proof The proof is analogous the proof of Fact 2.34. ■

Provided $\mathtt{e}_1$ has the type $\mathtt{Bool}^{\mathbf{n}}$ and both $\mathtt{e}_2$ and $\mathtt{e}_3$ are valid of the conjunction type $\mathtt{ct}$, then the conditional is valid of the type $\mathtt{ct}$:

Fact 2.36
$((\models \mathtt{e}_1 : \mathtt{Bool}^{\mathbf{n}}) \wedge (\models \mathtt{e}_2 : \mathtt{ct}) \wedge (\models \mathtt{e}_3 : \mathtt{ct})) \Leftrightarrow$
$(\models \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 : \mathtt{ct})$ □

Proof We assume $(\models \mathtt{e}_1 : \mathtt{Bool}^{\mathbf{n}})$, $(\models \mathtt{e}_2 : \mathtt{ct})$, and $(\models \mathtt{e}_3 : \mathtt{ct})$, then we show by induction on the conjunction type $\mathtt{ct}$ that

$$(\models \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 : \mathtt{ct})$$

is true. In all the cases we are using that $\mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3$ has the underlying type $\varepsilon(\mathtt{ct})$.

For the full details see Appendix page 278. ■

Provided $\mathtt{e}_1$ has the type $\mathtt{Bool}^{\top}$ and both $\mathtt{e}_2$ and $\mathtt{e}_3$ are valid of the conjunction type $\mathtt{ct}$, and $\mathtt{ct}$ can describe bottom, then the conditional is valid of the type $\mathtt{ct}$:

Fact 2.37
$((\models \mathtt{e}_1 : \mathtt{Bool}^{\top}) \wedge (\models \mathtt{e}_2 : \mathtt{ct}) \wedge (\models \mathtt{e}_3 : \mathtt{ct}) \wedge \mathrm{BOT}_{\mathrm{CT}}(\mathtt{ct})) \Rightarrow$
$(\models \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 : \mathtt{ct})$ □

Proof We assume $(\models \mathtt{e}_1 : \mathtt{Bool}^{\top})$, $(\models \mathtt{e}_2 : \mathtt{ct})$, $(\models \mathtt{e}_3 : \mathtt{ct})$, and that $\mathrm{BOT}_{\mathrm{CT}}(\mathtt{ct})$ is true, then we show by induction on the type $\mathtt{ct}$ that

$$(\models \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 : \mathtt{ct})$$

is true. In all the cases we use that $\mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3$ has the underlying type $\varepsilon(\mathtt{ct})$.

For the full details see Appendix page 281. ■

Next we show that our rules for $\leq_{\mathrm{ST}}$ and $\leq_{\mathrm{CT}}$ are sound:

Lemma 2.38 *Soundness of* $\leq_{\mathrm{ST}}$
$((\models \mathtt{e} : \mathtt{t}_1) \wedge (\mathtt{t}_1 \leq_{\mathrm{ST}} \mathtt{t}_2)) \Rightarrow (\models \mathtt{e} : \mathtt{t}_2)$ □

Proof We assume that $\models \mathtt{e} : \mathtt{t}_1$ is true and that $\mathtt{t}_1 \leq_{\mathrm{ST}} \mathtt{t}_2$ can be inferred, then we show by induction in the proof-tree of $\mathtt{t}_1 \leq_{\mathrm{ST}} \mathtt{t}_2$ that

$\models \texttt{e} : \texttt{t}_2$ is true. Throughout the proof we use that $\emptyset \vdash \texttt{e} : \varepsilon(\texttt{t}_2)$ can be inferred because

$$\models \texttt{e} : \texttt{t}_1$$

$\Downarrow$

$$\emptyset \vdash \texttt{e} : \varepsilon(\texttt{t}_1)$$

$\Updownarrow$ $\varepsilon(\texttt{t}_1) = \varepsilon(\texttt{t}_2)$ from Fact 2.26

$$\emptyset \vdash \texttt{e} : \varepsilon(\texttt{t}_2)$$

For the full details see Appendix page 284. ■

Lemma 2.39 *Soundness of* $\leq_{\mathrm{CT}}$
$((\models \texttt{e} : \texttt{ct}_1) \wedge (\texttt{ct}_1 \leq_{\mathrm{CT}} \texttt{ct}_2)) \Rightarrow (\models \texttt{e} : \texttt{ct}_2)$ □

Proof We assume that $\models \texttt{e} : \texttt{ct}_1$ is true and that $\texttt{ct}_1 \leq_{\mathrm{CT}} \texttt{ct}_2$ can be inferred, then we show by induction in the proof-tree of $\texttt{ct}_1 \leq_{\mathrm{CT}} \texttt{ct}_2$ that $\models \texttt{e} : \texttt{ct}_2$ is true.

For the full details see Appendix page 296. ■

We know from the semantics that $(\texttt{fix}_0\ \texttt{e})$ cannot evaluate hence it is valid of any type that can describe non-termination:

Lemma 2.40
$(\mathrm{BOT}_{\mathrm{ST}}(\texttt{t}_1) \wedge \varepsilon(\texttt{t}_1) = \varepsilon(\texttt{t}_2) \wedge \models \texttt{e} : \texttt{t}_1 \rightarrow \texttt{t}_2) \Rightarrow (\models \texttt{fix}_0\ \texttt{e} : \texttt{t}_1)$ □

Proof It is easy to show that $\models \texttt{fix}_0\ \texttt{e} : \varepsilon(\texttt{t}_1)^{\mathbf{b}}$ holds. Since we have shown that $\mathrm{BOT}_{\mathrm{ST}}(\texttt{t}_1)$ implies $\varepsilon(\texttt{t}_1)^{\mathbf{b}} \leq_{\mathrm{ST}} \texttt{t}_1$ (Lemma 2.28) we obtain the result using Lemma 2.39.

For the full details see Appendix page 298. ■

The relationship between $\texttt{fix}_j$ and $\texttt{fix}$ is clarified by:

Lemma 2.41
$(\exists j_0, j_1 : \forall k \geq 0 : (\models \texttt{fix}_{j_0+j_1\times k}\ \texttt{e} : \texttt{t})) \Rightarrow (\models \texttt{fix}\ \texttt{e} : \texttt{t})$ provided $\texttt{e}$ is without any $\texttt{fix}_j$ □

Proof We assume $\exists j_0, j_1 : \forall k \geq 0 : (\models \texttt{fix}_{j_0+j_1\times k}\ \texttt{e} : \texttt{t})$ and then we prove by induction on the strictness and totality type $\texttt{t}$ that $\models \texttt{fix}\ \texttt{e} : \texttt{t}$ is true.

For the full details see Appendix page 298. ■

2.4.4 The Soundness Proof

Finally we can prove Theorem 2.24:

Theorem 2.24 Soundness
For expressions $\mathtt{e}$ without any $\mathtt{fix}_n$ and $\mathcal{T}^n$ we have

$$(\overline{\mathtt{x}} : \overline{\mathtt{t}} \vdash_{\mathrm{ST}} \mathtt{e} : \mathtt{ct}) \Rightarrow (\forall\, \overline{\mathtt{v}}\colon (\models \overline{\mathtt{v}} : \overline{\mathtt{t}}) \Rightarrow (\models \mathtt{e}[\overline{\mathtt{v}}/\overline{\mathtt{x}}] : \mathtt{ct})). \qquad \square$$

Proof We assume that $A \vdash_{\mathrm{ST}} \mathtt{e} : \mathtt{ct}$ and that $(\models \overline{\mathtt{v}} : \overline{\mathtt{t}})$ is true, then we prove by induction in the proof-tree for $A \vdash_{\mathrm{ST}} \mathtt{e} : \mathtt{ct}$ that $\models \mathtt{e}[\overline{\mathtt{v}}/\overline{\mathtt{x}}] : \mathtt{ct}$ is true.

The case [abs]: We assume $A \vdash_{\mathrm{ST}} \lambda\mathtt{x}.\mathtt{e} : \mathtt{t}_1 \to \mathtt{t}_2$ and that $\models \overline{\mathtt{v}} : \overline{\mathtt{t}}$ is true. From the [abs]-rule we get $A, \mathtt{x} : \mathtt{t}_1 \vdash_{\mathrm{ST}} \mathtt{e} : \mathtt{t}_2$. Applying the induction hypothesis to this we get

$$((\models \overline{\mathtt{v}} : \overline{\mathtt{t}} \text{ for } A \wedge \models \mathtt{v} : \mathtt{t}_1) \Rightarrow (\models \mathtt{e}[\overline{\mathtt{v}}/\overline{\mathtt{x}}]\ [\mathtt{v}/\mathtt{x}] : \mathtt{t}_2)) \qquad (2.6)$$

We know that $\mathtt{x} : \mathtt{t}' \notin A$ since all the bound variables are distinct, that is $\mathtt{x} \notin \overline{\mathtt{x}}$. We want to show that

$$\models (\lambda\mathtt{x}.\mathtt{e})[\overline{\mathtt{v}}/\overline{\mathtt{x}}] : \mathtt{t}_1 \to \mathtt{t}_2$$

which is equivalent to show

$$(\forall \mathtt{e}' : (\models \mathtt{e}' : \mathtt{t}_1) \Rightarrow (\models (\lambda\mathtt{x}.\mathtt{e})[\overline{\mathtt{v}}/\overline{\mathtt{x}}]\ \mathtt{e}' : \mathtt{t}_2))$$

because $\mathtt{x} \notin \overline{\mathtt{x}}$ we have

$$(\forall \mathtt{e}' : (\models \mathtt{e}' : \mathtt{t}_1) \Rightarrow (\models \lambda\mathtt{x}.\mathtt{e}[\overline{\mathtt{v}}/\overline{\mathtt{x}}]\ \mathtt{e}' : \mathtt{t}_2))$$

By using (2.6) with $\models \overline{\mathtt{v}} : \overline{\mathtt{t}}$ and $\models \mathtt{e}' : \mathtt{t}_1$ we get

$$\models \mathtt{e}[\overline{\mathtt{v}}/\overline{\mathtt{x}}][\mathtt{e}'/\mathtt{x}] : \mathtt{t}_2$$

from Fact 2.32 we have

$$\models (\lambda\mathtt{x}.\mathtt{e}[\overline{\mathtt{v}}/\overline{\mathtt{x}}])\ \mathtt{e}' : \mathtt{t}_2$$

because $\mathtt{x} \notin \overline{\mathtt{x}}$ we get

$$\models (\lambda\mathtt{x}.\mathtt{e})[\overline{\mathtt{v}}/\overline{\mathtt{x}}]\ \mathtt{e}' : \mathtt{t}_2$$

as required.

The case [if2]: We assume $A \vdash_{ST} \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : \texttt{ct}$ and that $\models \overline{v} : \overline{t}$ is true. From the [if2]-rule we get

$$A \vdash_{ST} e_1 : \texttt{Bool}^{\mathbf{n}}$$
$$A \vdash_{ST} e_2 : \texttt{ct}$$
$$A \vdash_{ST} e_3 : \texttt{ct}$$

by applying the induction hypothesis to all three we get

$$\models e_1[\overline{v}/\overline{x}] : \texttt{Bool}^{\mathbf{n}}$$
$$\models e_2[\overline{v}/\overline{x}] : \texttt{ct}$$
$$\models e_3[\overline{v}/\overline{x}] : \texttt{ct}$$

By applying Fact 2.36 we have

$$\models \texttt{if } (e_1[\overline{v}/\overline{x}]) \texttt{ then } (e_2[\overline{v}/\overline{x}]) \texttt{ else } (e_3[\overline{v}/\overline{x}]) : \texttt{ct}$$

which is equivalent to

$$\models (\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3)[\overline{v}/\overline{x}] : \texttt{ct}$$

as required.

The case [fix]: We assume

$$A \vdash_{ST} \texttt{fix } e : t_n$$

$\mathrm{BOT}_{ST}(t_1)$, $t_q \leq_{ST} t_p$, $p < q$, and that $\models \overline{v} : \overline{t}$ is true. From the [fix]-rule we get

$$A \vdash_{ST} e : t_1 \to t_2 \wedge t_2 \to t_3 \wedge \ldots \wedge t_{n-1} \to t_n$$

By applying the induction hypothesis we get

$$\models e[\overline{v}/\overline{x}] : t_1 \to t_2 \wedge t_2 \to t_3 \wedge \ldots \wedge t_{n-1} \to t_n$$

which is equivalent to

$$\models e[\overline{v}/\overline{x}] : t_1 \to t_2$$
$$\models e[\overline{v}/\overline{x}] : t_2 \to t_3$$
$$\vdots$$
$$\models e[\overline{v}/\overline{x}] : t_{n-1} \to t_n$$

From Lemma 2.40 we have

$$\models \texttt{fix}_0\ \texttt{e}[\overline{\texttt{v}}/\overline{\texttt{x}}] : \texttt{t}_1$$

By applying $\models \texttt{e}[\overline{\texttt{v}}/\overline{\texttt{x}}] : \texttt{t}_1 \to \texttt{t}_2$ we get

$$\models \texttt{e}\ (\texttt{fix}_0\ \texttt{e}[\overline{\texttt{v}}/\overline{\texttt{x}}]) : \texttt{t}_2$$

and Fact 2.34 gives

$$\models \texttt{fix}_1\ \texttt{e}[\overline{\texttt{v}}/\overline{\texttt{x}}] : \texttt{t}_2$$

By applying $\models \texttt{e}[\overline{\texttt{v}}/\overline{\texttt{x}}] : \texttt{t}_2 \to \texttt{t}_3$ we get

$$\models \texttt{e}\ (\texttt{fix}_1\ \texttt{e}[\overline{\texttt{v}}/\overline{\texttt{x}}]) : \texttt{t}_3$$

and Fact 2.34 gives

$$\models \texttt{fix}_2\ \texttt{e}[\overline{\texttt{v}}/\overline{\texttt{x}}] : \texttt{t}_3$$

We arrive at

$$\models \texttt{fix}_{q-1}\ \texttt{e}[\overline{\texttt{v}}/\overline{\texttt{x}}] : \texttt{t}_q$$

Now because we have $\texttt{t}_q \ \leq_{\text{ST}}\ \texttt{t}_p$ we can apply Lemma 2.38 to get

$$\models \texttt{fix}_{q-1}\ \texttt{e}[\overline{\texttt{v}}/\overline{\texttt{x}}] : \texttt{t}_p$$

By applying $\models \texttt{e}[\overline{\texttt{v}}/\overline{\texttt{x}}] : \texttt{t}_p \to \texttt{t}_{p+1}$

$$\models \texttt{e}\ (\texttt{fix}_{q-1}\ \texttt{e}[\overline{\texttt{v}}/\overline{\texttt{x}}]) : \texttt{t}_{p+1}$$

using Fact 2.34 we get

$$\models \texttt{fix}_{q-1+1}\ \texttt{e}[\overline{\texttt{v}}/\overline{\texttt{x}}] : \texttt{t}_{p+1}$$

We have

$$\forall k \geq 0 : \models \texttt{fix}_{q-1+(q-p)k}\ \texttt{e}[\overline{\texttt{v}}/\overline{\texttt{x}}] : \texttt{t}_q$$

and using Lemma 2.41 gives

$$\models \texttt{fix}\,\texttt{e}[\overline{\texttt{v}}/\overline{\texttt{x}}] : \texttt{t}_q$$

By applying $\models \mathtt{e}[\overline{\mathtt{v}}/\overline{\mathtt{x}}] : \mathtt{t}_q \to \mathtt{t}_{q+1}$

$$\models \mathtt{e}[\overline{\mathtt{v}}/\overline{\mathtt{x}}]\ (\mathtt{fix}\ \mathtt{e}[\overline{\mathtt{v}}/\overline{\mathtt{x}}]) : \mathtt{t}_{q+1}$$

Now Fact 2.35 gives

$$\models \mathtt{fix}\ \mathtt{e}[\overline{\mathtt{v}}/\overline{\mathtt{x}}] : \mathtt{t}_{q+1}$$

We arrive at

$$\models \mathtt{fix}\ \mathtt{e}[\overline{\mathtt{v}}/\overline{\mathtt{x}}] : \mathtt{t}_n$$

that is

$$\models (\mathtt{fix}\ \mathtt{e})[\overline{\mathtt{v}}/\overline{\mathtt{x}}] : \mathtt{t}_n$$

as required.

The case [coer]: We assume

$$\begin{array}{l} \mathrm{A} \vdash_{\mathrm{ST}} \mathtt{e} : \mathtt{ct}_2 \\ \mathtt{ct}_1 \leq_{\mathrm{CT}} \mathtt{ct}_2 \end{array}$$

and that $\models \overline{\mathtt{v}} : \overline{\mathtt{t}}$ is true. From the [coer]-rule we have

$$\mathrm{A} \vdash_{\mathrm{ST}} \mathtt{e} : \mathtt{ct}_1$$

By applying the induction hypothesis we get

$$\models \mathtt{e}[\overline{\mathtt{v}}/\overline{\mathtt{x}}] : \mathtt{ct}_1$$

Using Lemma 2.38 we get

$$\models \mathtt{e}[\overline{\mathtt{v}}/\overline{\mathtt{x}}] : \mathtt{ct}_2$$

as required.

For the remaining cases see Appendix page 305. ■

Soundness of the Standard Type Inference System

Now that we have a semantics we can state the soundness of the standard type inference system (Figure 1.1): first we extend the notion of validity to standard types:

$$\begin{array}{lcl}
\models \mathtt{e} : \mathtt{Bool} & \Leftrightarrow & (\vdash \mathtt{e} \Downarrow \mathtt{true}) \vee (\vdash \mathtt{e} \Downarrow \mathtt{false}) \\
\models \mathtt{e} : \mathtt{Int} & \Leftrightarrow & (\vdash \mathtt{e} \Downarrow \mathtt{c}) \quad \mathtt{c} \text{ is an integer} \\
\models \mathtt{e} : \mathtt{ut}_1 \to \mathtt{ut}_2 & \Leftrightarrow & (\forall \mathtt{e}' : (\models \mathtt{e}' : \mathtt{ut}_1) \Rightarrow (\models \mathtt{e}\ \mathtt{e}' : \mathtt{ut}_2))
\end{array}$$

Now soundness of the standard type system can be formalised as:

$$\begin{array}{l}
A \vdash \mathtt{e} : \mathtt{ut} \Rightarrow \\
\quad (\forall \overline{\mathtt{v}} : ((\models \overline{\mathtt{v}} : \overline{\mathtt{ut}}) \wedge (\vdash \mathtt{e}[\overline{\mathtt{v}}/\overline{\mathtt{x}}] \Downarrow \mathtt{v})) \Rightarrow (\models \mathtt{e}[\overline{\mathtt{v}}/\overline{\mathtt{x}}] : \mathtt{ut}))
\end{array}$$

The soundness proof for the analysis can easily be adapted to a proof of soundness for the standard type system.

2.5 Summary

We have described an inference system for combining strictness and totality analysis and we have proved the analysis sound with respect to a natural-style operational semantics.

We have briefly compared the results obtained by our analysis to those obtained by e.g. [Jen91, Ben93, Jen92b, KM89, Wri91]. In some cases we get more precise results, in others they do. One may note that the type systems of Jensen [Jen91] and Benton [Ben93] allows general conjunction types. The reason that Jensen has no problems with unrestricted conjunctions is that it is not possible to construct empty types: the type system only includes the $\{\mathbf{b}, \top\}$ annotated part of our system.

An open problem is the meaningful integration of lists and other datatypes. For the strictness part one may be inspired by [Wad87]. Consider the type `B list` where `B` is a base-type. The strictness and totality type $(\mathtt{B}^{\mathbf{n}})\mathtt{list}$ might then describe the finite lists with no bottom elements, the type $(\mathtt{B}^{\mathbf{b}})\mathtt{list}$ might describe the infinite lists or lists with bottom elements, and the strictness and totality type $(\mathtt{B}^{\top})\mathtt{list}$ might describe all list. A strictness and totality type of the `map` function would then be

$$(\mathtt{B}^{\mathbf{n}} \to \mathtt{B}'^{\mathbf{n}}) \to (\mathtt{B}^{\mathbf{n}})\mathtt{list} \to (\mathtt{B}'^{\mathbf{n}})\mathtt{list}$$

Similarly, `foldl` and `foldr` will have strictness and totality types

$$(\mathtt{B}^{\mathbf{n}} \to \mathtt{B}'^{\mathbf{n}} \to \mathtt{B}^{\mathbf{n}}) \to \mathtt{B}^{\mathbf{n}} \to (\mathtt{B}'^{\mathbf{n}})\mathtt{list} \to \mathtt{B}^{\mathbf{n}}$$

and

$$(\mathtt{B}^{\mathbf{n}} \to \mathtt{B}'^{\mathbf{n}} \to \mathtt{B}'^{\mathbf{n}}) \to \mathtt{B}'^{\mathbf{n}} \to (\mathtt{B}^{\mathbf{n}})\mathtt{list} \to \mathtt{B}'^{\mathbf{n}}$$

respectively. However, to get this information from the analysis we need to analyse fixpoints in a better way, e.g. as suggested in [NN95]. Consider the factorial function:

$$\mathtt{fac} ::= \mathtt{fix}\ (\lambda\mathtt{f}.\lambda\mathtt{x}.\mathtt{if\ =\ x\ 1\ then\ 1\ else\ *(f\ (-\ x\ 1))\ x})$$

We can infer the strictness and totality type for the factorial function

$$(\mathtt{Int}^{\mathbf{n}} \to \mathtt{Int}^{\top}) \wedge (\mathtt{Int}^{\mathbf{b}} \to \mathtt{Int}^{\mathbf{b}})$$

but not the type

$$\mathtt{Int}^{\mathbf{n}} \to \mathtt{Int}^{\mathbf{n}}$$

In order to do that we have to define a well-founded ordering as done in [NN95].

In Chapter 3 we will lift the restriction on the placement of conjunction; this results in a somewhat more powerful system.

Chapter 3

Strictness and Totality Analysis with Conjunction

The type system in Chapter 2 only allows conjunctions at the top-level. Therefore we are not able to write the strictness and totality type

$$((\mathtt{t}_1 \wedge \mathtt{t}_2) \rightarrow (\mathtt{t}_2 \wedge \mathtt{t}_3)) \rightarrow \mathtt{t}_1 \rightarrow \mathtt{t}_3$$

which is exactly what we needed in Example 2.9. In this Chapter we will lift this restriction:

$$\mathtt{t} ::= \mathtt{ut}^s \mid \mathtt{t} \rightarrow \mathtt{t} \mid \mathtt{t} \wedge \mathtt{t}$$

However it is not immediate to expand the development of Chapter 2 to the new annotated types. Recall that $\downarrow$ was introduced as an operation on types that allowed us to express downwards-closure on types. This is needed in order to state the monotonicity rule. We will need to extend the $\downarrow$-operation to conjunction types and a first attempt may be to define

$$\downarrow(\mathtt{t}_1 \wedge \mathtt{t}_2) \;\; = \;\; \downarrow\mathtt{t}_1 \wedge \downarrow\mathtt{t}_2$$

However this is not sound because types may be empty: The type

$$\downarrow(\mathtt{Int}^{\mathbf{n}} \wedge \mathtt{Int}^{\mathbf{b}})$$

should be empty because $(\mathtt{Int}^{\mathbf{n}} \wedge \mathtt{Int}^{\mathbf{b}})$ is, but

$$\downarrow\mathtt{Int}^{\mathbf{n}} \wedge \downarrow\mathtt{Int}^{\mathbf{b}} \;\; = \;\; \mathtt{Int}^{\top} \wedge \mathtt{Int}^{\mathbf{b}} \equiv \mathtt{Int}^{\mathbf{b}}$$

clearly is not empty.

To overcome this problem we shall introduce $\downarrow$ as a syntactic construct. So the annotated types will be given by

$$\mathtt{t} ::= \mathtt{ut}^s \mid \mathtt{t} \rightarrow \mathtt{t} \mid \mathtt{t} \wedge \mathtt{t} \mid \downarrow\mathtt{t}$$

The ordering on types will be such that $\downarrow(\mathtt{t}_1 \wedge \mathtt{t}_2) \leq_{\mathrm{D}} \downarrow\mathtt{t}_1 \wedge \downarrow\mathtt{t}_2$, but as we shall see we will not have $\downarrow(\mathtt{t}_1 \wedge \mathtt{t}_2) \geq_{\mathrm{D}} \downarrow\mathtt{t}_1 \wedge \downarrow\mathtt{t}_2$.

Overview In Section 3.1 we define the strictness and totality types with conjunction and give rules for coercing between them; and the inference system is presented and examples of its use are given. In Section 3.2 we discuss the power of the fixpoint-rules; in Section 3.3 we then present a denotational semantics and finally in Section 3.4 the analysis is proven correct.

3.1 The Annotated Type System

3.1.1 The Strictness and Totality Types

A strictness and totality type with conjunction, $\mathtt{t}$, is either an annotated underlying type, a function type between strictness and totality types, a conjunction of two strictness and totality types, or a $\downarrow$-type:

$$\begin{array}{rcl} \mathtt{t} & ::= & \mathtt{ut}^s \mid \mathtt{t} \rightarrow \mathtt{t} \mid \mathtt{t} \wedge \mathtt{t} \mid \downarrow\mathtt{t} \\ \mathtt{ut} & ::= & \mathtt{B} \mid \mathtt{ut} \rightarrow \mathtt{ut} \\ s & ::= & \top \mid \mathbf{n} \mid \mathbf{b} \end{array}$$

As in Chapter 2 we will not allow the two conjuncts of $\mathtt{t}_1 \wedge \mathtt{t}_2$ to have different underlying types, so again we will define a well-formedness predicate for strictness and totality types. An annotated underlying type is a well-formed strictness and totality type:

$$\frac{}{\vdash^W \mathtt{ut}^s} \tag{3.1}$$

A function type is well-formed, whenever the two subtypes are well-formed:

$$\frac{\vdash^W \mathtt{t}_1 \quad \vdash^W \mathtt{t}_2}{\vdash^W \mathtt{t}_1 \rightarrow \mathtt{t}_2} \tag{3.2}$$

A conjunction is well-formed whenever the two conjuncts are well-formed and they have the same underlying type:

$$\frac{\vdash^W \mathtt{t}_1 \quad \vdash^W \mathtt{t}_2}{\vdash^W \mathtt{t}_1 \wedge \mathtt{t}_2} \quad \text{if } \varepsilon(\mathtt{t}_1) = \varepsilon(\mathtt{t}_2) \tag{3.3}$$

and the strictness and totality type $\downarrow\mathtt{t}$ is well-formed provided $\mathtt{t}$ is well-formed:

$$\frac{\vdash^W \mathtt{t}}{\vdash^W \downarrow\mathtt{t}} \tag{3.4}$$

The predicate, $\mathrm{BOT}_{\mathrm{ST}}$, is true for the strictness and totality strictness and totality type, $\mathtt{t}$, whenever bottom can be described by the type $\mathtt{t}$. It is defined by:

$$\begin{aligned}
\mathrm{BOT}_{\mathrm{ST}}(\mathtt{ut}^{\mathbf{n}}) &= \mathtt{ff} \\
\mathrm{BOT}_{\mathrm{ST}}(\mathtt{ut}^{\top}) &= \mathtt{tt} \\
\mathrm{BOT}_{\mathrm{ST}}(\mathtt{ut}^{\mathbf{b}}) &= \mathtt{tt} \\
\mathrm{BOT}_{\mathrm{ST}}(\downarrow\mathtt{t}) &= \mathrm{BOT}_{\mathrm{ST}}(\mathtt{t}) \\
\mathrm{BOT}_{\mathrm{ST}}(\mathtt{t}_1 \to \mathtt{t}_2) &= \mathrm{BOT}_{\mathrm{ST}}(\mathtt{t}_2) \\
\mathrm{BOT}_{\mathrm{ST}}(\mathtt{t}_1 \wedge \mathtt{t}_2) &= \mathrm{BOT}_{\mathrm{ST}}(\mathtt{t}_1) \wedge \mathrm{BOT}_{\mathrm{ST}}(\mathtt{t}_2)
\end{aligned}$$

The reason for *not* taking $\mathrm{BOT}_{\mathrm{ST}}(\downarrow\mathtt{t})$ to be $\mathtt{tt}$ is $\mathtt{t}$ may be empty, e.g. if $\mathtt{t} = \mathtt{ut}^{\mathbf{n}} \wedge \mathtt{ut}^{\mathbf{b}}$. Therefore $\mathrm{BOT}_{\mathrm{ST}}(\downarrow(\mathtt{ut}^{\mathbf{n}} \wedge \mathtt{ut}^{\mathbf{b}}))$ cannot be true. However, the definition of $\mathrm{BOT}_{\mathrm{ST}}(\downarrow\mathtt{t})$ that we have adopted is not as precise as one would wish. An example of which is when $\mathtt{t} = \mathtt{ut}^{\mathbf{n}}$ where we get

$$\mathrm{BOT}_{\mathrm{ST}}(\downarrow\mathtt{ut}^{\mathbf{n}}) = \mathrm{BOT}_{\mathrm{ST}}(\mathtt{ut}^{\mathbf{n}}) = \mathtt{ff}$$

however

$$\mathrm{BOT}_{\mathrm{ST}}(\downarrow\mathtt{ut}^{\mathbf{n}}) = \mathtt{tt}$$

is more precise. In Chapter 2 we did not have this problem with the predicate for the simple reason that we did not have to define the $\mathrm{BOT}_{\mathrm{ST}}$ on $\downarrow$-types.

The coercion relation $\leq_{\mathrm{D}}$ is defined by the rules of Figure 3.1. Most of the rules are the ones from Figure 2.1, which define the relation $\leq_{\mathrm{ST}}$ and from Figure 2.2, which define the relation $\leq_{\mathrm{CT}}$. We will write $\equiv_{\mathrm{D}}$ for the

$$[\text{ref}]\ \frac{}{\mathtt{t}\ \le_{\mathrm{D}}\ \mathtt{t}}$$

$$[\text{trans}]\ \frac{\mathtt{t}_1\ \le_{\mathrm{D}}\ \mathtt{t}_2 \quad \mathtt{t}_2\ \le_{\mathrm{D}}\ \mathtt{t}_3}{\mathtt{t}_1\ \le_{\mathrm{D}}\ \mathtt{t}_3}$$

$$[\text{arrow}]\ \frac{\mathtt{t}_3\ \le_{\mathrm{D}}\ \mathtt{t}_1 \quad \mathtt{t}_2\ \le_{\mathrm{D}}\ \mathtt{t}_4}{\mathtt{t}_1 \to \mathtt{t}_2\ \le_{\mathrm{D}}\ \mathtt{t}_3 \to \mathtt{t}_4}$$

$$[\text{top1}]\ \frac{}{\mathtt{t}\ \le_{\mathrm{D}}\ \varepsilon(\mathtt{t})^{\top}}$$

$$[\text{top2}]\ \frac{}{(\mathtt{ut}_1 \to \mathtt{ut}_2)^{\top}\ \le_{\mathrm{D}}\ \mathtt{ut}_1{}^{\top} \to \mathtt{ut}_2{}^{\top}}$$

$$[\text{bot}]\ \frac{}{(\mathtt{ut}_1 \to \mathtt{ut}_2)^{\mathbf{b}}\ \le_{\mathrm{D}}\ \mathtt{ut}_1{}^{\top} \to \mathtt{ut}_2{}^{\mathbf{b}}}$$

$$[\text{notbot}]\ \frac{}{\mathtt{ut}_1{}^{\mathbf{n}} \to \mathtt{ut}_2{}^{\mathbf{n}}\ \le_{\mathrm{D}}\ (\mathtt{ut}_1 \to \mathtt{ut}_2)^{\mathbf{n}}}$$

$$[\wedge 1]\ \frac{}{\mathtt{t}_1 \wedge \mathtt{t}_2\ \le_{\mathrm{D}}\ \mathtt{t}_1} \qquad [\wedge 2]\ \frac{}{\mathtt{t}_1 \wedge \mathtt{t}_2\ \le_{\mathrm{D}}\ \mathtt{t}_2}$$

$$[\wedge 3]\ \frac{\mathtt{t}\ \le_{\mathrm{D}}\ \mathtt{t}_1 \quad \mathtt{t}\ \le_{\mathrm{D}}\ \mathtt{t}_2}{\mathtt{t}\ \le_{\mathrm{D}}\ \mathtt{t}_1 \wedge \mathtt{t}_2}$$

$$[\downarrow 1]\ \frac{}{\mathtt{t}\ \le_{\mathrm{D}}\ \downarrow\mathtt{t}} \qquad [\downarrow 2]\ \frac{\mathtt{t}_1\ \le_{\mathrm{D}}\ \mathtt{t}_2}{\downarrow\mathtt{t}_1\ \le_{\mathrm{D}}\ \downarrow\mathtt{t}_2}$$

$$[\downarrow 3]\ \frac{}{\mathtt{ut}^{\top}\ \le_{\mathrm{D}}\ \downarrow\mathtt{ut}^{\mathbf{n}}} \qquad [\downarrow 4]\ \frac{}{\downarrow\mathtt{ut}^{\mathbf{b}}\ \le_{\mathrm{D}}\ \mathtt{ut}^{\mathbf{b}}}$$

$$[\downarrow 5]\ \frac{}{\downarrow(\downarrow\mathtt{t})\ \le_{\mathrm{D}}\ \downarrow\mathtt{t}} \qquad [\downarrow 6]\ \frac{}{\downarrow(\mathtt{t}_1 \wedge \mathtt{t}_2)\ \le_{\mathrm{D}}\ \downarrow\mathtt{t}_1 \wedge \downarrow\mathtt{t}_2}$$

$$[\downarrow 7]\ \frac{}{\downarrow(\mathtt{t}_1 \to \mathtt{t}_2) \equiv_{\mathrm{D}} \mathtt{t}_1 \to \downarrow\mathtt{t}_2}$$

$$[\text{monotone}]\ \frac{}{\mathtt{t}_1 \to \mathtt{t}_2\ \le_{\mathrm{D}}\ \downarrow\mathtt{t}_1 \to \downarrow\mathtt{t}_2}$$

$$[\to \wedge]\ \frac{}{(\mathtt{t}_1 \to \mathtt{t}_2) \wedge (\mathtt{t}_1 \to \mathtt{t}_3)\ \le_{\mathrm{D}}\ \mathtt{t}_1 \to (\mathtt{t}_2 \wedge \mathtt{t}_3)}$$

Figure 3.1: Coercions Between Strictness and Totality Types

equivalence relation induced by $\leq_D$, i.e. $\mathtt{t}_1 \equiv_D \mathtt{t}_2$ if and only if $\mathtt{t}_1 \leq_D \mathtt{t}_2$ and $\mathtt{t}_2 \leq_D \mathtt{t}_1$. Compared with Chapter 2 the rule $[\wedge \rightarrow]$ is new, and all the rules involving $\downarrow$.

The rule $[\wedge \rightarrow]$ is the one that Jensen and Benton [Jen92a, Ben93] have but we had to discard in Chapter 2 due to un-well-formedness of the types involved. The rule $[\downarrow 1]$ expresses that all terms of type $\mathtt{t}$ is included in the set of terms of type $\downarrow\mathtt{t}$. Note that this is expressed by Fact 2.27 part a). Whenever two types, $\mathtt{t}_1$ and $\mathtt{t}_2$, are related then so are $\downarrow\mathtt{t}_1$ and $\downarrow\mathtt{t}_2$. This is expressed by the rule $[\downarrow 2]$, which is comparable with Fact 2.27 part c). The rule $[\downarrow 5]$ which is comparable with Fact 2.27 part b), says that applying the $\downarrow$-construct twice makes no difference as to applying the $\downarrow$-construct once.

Terms of type $\mathtt{ut}^{\top}$ is included in, in fact is equal to, the terms of type $\downarrow\mathtt{ut}^{\mathbf{n}}$ is expressed by the rule $[\downarrow 3]$. From [top1] we have $\downarrow\mathtt{ut}^{\mathbf{n}} \leq_D \mathtt{ut}^{\top}$ and thereby we have $\downarrow\mathtt{ut}^{\mathbf{n}} \equiv_D \mathtt{ut}^{\top}$. This is exactly what the definition, of the $\downarrow$-operation (2.3) in Chapter 2, says. The rule $[\downarrow 4]$ says that terms of type $\downarrow\mathtt{ut}^{\mathbf{b}}$ are include in the terms of type $\mathtt{ut}^{\mathbf{b}}$. From the rule $[\downarrow 1]$ we have $\mathtt{ut}^{\mathbf{b}} \leq_D \downarrow\mathtt{ut}^{\mathbf{b}}$, so we have $\mathtt{ut}^{\mathbf{b}} \equiv_D \downarrow\mathtt{ut}^{\mathbf{b}}$. This corresponds to (2.2) of the definition of the $\downarrow$-operation. The rule $[\downarrow 7]$ correspond directly to (2.5). The rule $[\downarrow 6]$ says that the terms of type $\downarrow(\mathtt{t}_1 \wedge \mathtt{t}_2)$ is included in the terms of type $(\downarrow\mathtt{t}_1 \wedge \downarrow\mathtt{t}_2)$. There is no counterpart to this rule in Chapter 2 since the $\downarrow$-operation is not defined on conjunction types. There is no special rule corresponding to (2.4) of the definition of the $\downarrow$-operation. However from the rules $[\downarrow 1]$ and [top1] we have $\mathtt{ut}^{\top} \leq_D \downarrow\mathtt{ut}^{\top}$ and $\downarrow\mathtt{ut}^{\top} \leq_D \mathtt{ut}^{\top}$, respectively.

To summarise: we have using the coercion rules:

$$\begin{array}{ll} \downarrow\mathtt{ut}^{\mathbf{n}} \equiv_D \mathtt{ut}^{\top} & \qquad \downarrow\mathtt{ut}^{\top} \equiv_D \mathtt{ut}^{\top} \\ \downarrow\mathtt{ut}^{\mathbf{b}} \equiv_D \mathtt{ut}^{\mathbf{b}} & \qquad \downarrow(\mathtt{t}_1 \rightarrow \mathtt{t}_2) \equiv_D \mathtt{t}_1 \rightarrow \downarrow\mathtt{t}_2 \end{array}$$

which is directly comparable with the definition of the $\downarrow$-operation, and

$$\downarrow(\mathtt{t}_1 \wedge \mathtt{t}_2) \leq_D \downarrow\mathtt{t}_1 \wedge \downarrow\mathtt{t}_2$$

So we do not have an equivalence between the $\downarrow$-types and the types without the $\downarrow$-construct. The advantage of having an equivalence would be that the programmer would not have to think about the $\downarrow$-types, however the programmer can choose not to use the $\downarrow$-types.

The relation $\leq_{\mathrm{D}}$ is sound but not complete. The soundness result is formalised in Lemma 3.9 below. The lack of completeness is seen by the same example (see page 34) as for the lack of completeness for $\leq_{\mathrm{ST}}$ in Chapter 2: We are not able to infer $\mathtt{Int}^{\mathbf{b}} \to \mathtt{Int}^{\mathbf{n}} \leq_{\mathrm{D}} \mathtt{Int}^{\top} \to \mathtt{Int}^{\mathbf{n}}$ even though all terms of type $\mathtt{Int}^{\mathbf{b}} \to \mathtt{Int}^{\mathbf{n}}$ is included in the terms of type $\mathtt{Int}^{\top} \to \mathtt{Int}^{\mathbf{n}}$.

3.1.2 The Analysis

Now we can define the analysis: The list A of assumptions gives strictness and totality types with "full" conjunction to the free variables. For each constant $\mathtt{c}$, we assume that a strictness and totality type is specified. The inference rules for the analysis is Figure 3.2.

Note that the analysis is as in Figure 3.2 except that we do not distinguish between $\mathtt{ct}$ and $\mathtt{t}$ — all the types are strictness and totality types with "full" conjunction.

Example 3.1
For the term $\mathtt{e}$ from Example 2.9:

$$\begin{aligned} \mathtt{e} &= \mathtt{twice\ g} \\ \mathtt{twice} &= \lambda\mathtt{f}.\lambda\mathtt{x}.\mathtt{f}\ (\mathtt{f}\ \mathtt{x}) \\ \mathtt{g} &= \lambda\mathtt{y}.\lambda\mathtt{x}.+\ \mathtt{x}\ (\mathtt{y}\ (\mathtt{fix}\ \lambda\mathtt{x}.\mathtt{x})) \end{aligned}$$

we can infer the type

$$(\mathtt{Int}^{\top} \to \mathtt{Int}^{\top}) \to \mathtt{Int}^{\top} \to \mathtt{Int}^{\mathbf{b}}$$

which is not possible using the analysis in Chapter 2. □

3.2 The Power of the Fix-rules

Also for this Chapter we will investigate the power of the fix-rule as we did in Section 2.2 for the analysis in Chapter 2.

Recall the two rules [fix1] and [fix2] from Chapter 2:

$$[\mathrm{fix1}]\ \frac{\mathrm{A} \vdash \mathtt{e} : \mathtt{t} \to \mathtt{t}}{\mathrm{A} \vdash \mathtt{fix}\ \mathtt{e} : \mathtt{t}} \quad \text{if } \mathrm{BOT}_{\mathrm{ST}}(\mathtt{t})$$

$$[\text{var}]\ \frac{}{\mathrm{A} \vdash \mathtt{x} : \mathtt{t}} \quad \text{if } \mathtt{x} : \mathtt{t} \in \mathrm{A}$$

$$[\text{abs}]\ \frac{\mathrm{A}, \mathtt{x} : \mathtt{t}_1 \vdash \mathtt{e} : \mathtt{t}_2}{\mathrm{A} \vdash \lambda\mathtt{x}.\mathtt{e} : \mathtt{t}_1 \to \mathtt{t}_2}$$

$$[\text{abs2}]\ \frac{\mathrm{A}, \mathtt{x} : \mathtt{t}_1 \vdash \mathrm{e} : \mathtt{t}_2}{\mathrm{A} \vdash \lambda\mathtt{x}.\mathtt{e} : \varepsilon(\mathtt{t}_1 \to \mathtt{t}_2)^{\mathbf{n}}}$$

$$[\text{app}]\ \frac{\mathrm{A} \vdash \mathtt{e}_1 : \mathtt{t}_1 \to \mathtt{t}_2 \quad \mathrm{A} \vdash \mathtt{e}_2 : \mathtt{t}_1}{\mathrm{A} \vdash \mathtt{e}_1\ \mathtt{e}_2 : \mathtt{t}_2}$$

$$[\text{if1}]\ \frac{\mathrm{A} \vdash \mathtt{e}_1 : \mathtt{Bool}^{\mathbf{b}} \quad \mathrm{A} \vdash \mathtt{e}_2 : \mathtt{t} \quad \mathrm{A} \vdash \mathtt{e}_3 : \mathtt{t}}{\mathrm{A} \vdash \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 : \varepsilon(\mathtt{t})^{\mathbf{b}}}$$

$$[\text{if2}]\ \frac{\mathrm{A} \vdash \mathtt{e}_1 : \mathtt{Bool}^{\mathbf{n}} \quad \mathrm{A} \vdash \mathtt{e}_2 : \mathtt{t} \quad \mathrm{A} \vdash \mathtt{e}_3 : \mathtt{t}}{\mathrm{A} \vdash \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 : \mathtt{t}}$$

$$[\text{if3}]\ \frac{\mathrm{A} \vdash \mathtt{e}_1 : \mathtt{Bool}^{\top} \quad \mathrm{A} \vdash \mathtt{e}_2 : \mathtt{t} \quad \mathrm{A} \vdash \mathtt{e}_3 : \mathtt{t}}{\mathrm{A} \vdash \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 : \mathtt{t}} \quad \text{if } \mathrm{BOT}_{\mathrm{ST}}(\mathtt{t})$$

$$[\text{fix}]\ \frac{\mathrm{A} \vdash \mathtt{e} : \mathtt{t}_1 \to \mathtt{t}_2 \wedge \mathtt{t}_2 \to \mathtt{t}_3 \wedge \ldots \wedge \mathtt{t}_{n-1} \to \mathtt{t}_n}{\mathrm{A} \vdash \mathtt{fix}\ \mathtt{e} : \mathtt{t}_n} \quad \text{if } \begin{cases} \mathrm{BOT}_{\mathrm{ST}}(\mathtt{t}_1), \\ \exists p, q : p < q \\ \wedge \mathtt{t}_q \leq_{\mathrm{D}} \mathtt{t}_p \end{cases}$$

$$[\text{const}]\ \frac{}{\mathrm{A} \vdash \mathtt{c} : \mathtt{t}_{\mathtt{c}}}$$

$$[\text{coer}]\ \frac{\mathrm{A} \vdash \mathtt{e} : \mathtt{t}_1}{\mathrm{A} \vdash \mathtt{e} : \mathtt{t}_2} \quad \text{if } \mathtt{t}_1 \leq_{\mathrm{D}} \mathtt{t}_2$$

$$[\text{conj}]\ \frac{\mathrm{A} \vdash \mathtt{e} : \mathtt{t}_1 \quad \mathrm{A} \vdash \mathtt{e} : \mathtt{t}_2}{\mathrm{A} \vdash \mathtt{e} : \mathtt{t}_1 \wedge \mathtt{t}_2}$$

Figure 3.2: Strictness and Totality Type Inference

and

$$[\text{fix2}]\ \frac{\mathrm{A} \vdash \mathtt{e} : \mathtt{t}_1 \to \mathtt{t}_2}{\mathrm{A} \vdash \mathtt{fix}\ \mathtt{e} : \mathtt{t}_2} \quad \text{if } \mathrm{BOT}_{\mathrm{ST}}(\mathtt{t}_1) \text{ and } \mathtt{t}_2 \ \leq_{\mathrm{ST}}\ \mathtt{t}_1$$

Let $\vdash^{\mathcal{A}}_{\text{fix}}$ be the inference system of Figure 3.2 but with annotations in $\mathcal{A}$. Similarly let $\vdash^{\mathcal{A}}_{\text{fix1}}$ be the system where [fix] is replaced by [fix1] and let $\vdash^{\mathcal{A}}_{\text{fix2}}$ be the system where [fix] is replaced by [fix2]. Note that in the inference systems $\vdash^{\{\mathbf{b},\top\}}_{\text{fix1}}$, $\vdash^{\{\mathbf{b},\top\}}_{\text{fix2}}$, and $\vdash^{\{\mathbf{b},\top\}}_{\text{fix}}$ is makes no difference whether we allow $\downarrow$-types or not. The reason is that we cannot construct empty types using only $\mathbf{b}$ and $\top$ annotates. Thereby we have

$$\downarrow(\mathtt{t}_1 \wedge \mathtt{t}_2) \equiv_{\mathrm{D}} \downarrow\mathtt{t}_1 \wedge \downarrow\mathtt{t}_2$$

Also note that $\vdash^{\{\mathbf{b},\top\}}_{\text{fix1}}$ is the strictness analysis of Jensen and Benton [Jen91, Jen92b, Ben93].

It is immediate that

$$\vdash^{\mathcal{A}}_{\text{fix1}} \subseteq \vdash^{\mathcal{A}}_{\text{fix2}} \subseteq \vdash^{\mathcal{A}}_{\text{fix}}$$

and that

$$\vdash^{\{\mathbf{b},\top\}}_{\phi} \subseteq \vdash^{\{\mathbf{n},\mathbf{b},\top\}}_{\phi}$$

for all $\mathcal{A}$ and $\phi \in \{\text{fix, fix1, fix2}\}$. We now consider the extent to which the inclusions are proper or are equalities; the results are summarised in Table 3.1.

Claim $\vdash^{\{\mathbf{b},\top\}}_{\text{fix1}} = \vdash^{\{\mathbf{b},\top\}}_{\text{fix2}}$

In order to show that $\vdash^{\{\mathbf{b},\top\}}_{\text{fix1}} = \vdash^{\{\mathbf{b},\top\}}_{\text{fix2}}$ it suffices to show that the rule [fix2] can be derived from the rule [fix1]. For this assume

$$\begin{array}{l} \mathrm{A} \vdash^{\{\mathbf{b},\top\}}_{\text{fix1}} \mathtt{e} : \mathtt{t}_1 \to \mathtt{t}_2 \\ \mathtt{t}_2 \leq_{\mathrm{ST}} \mathtt{t}_1 \\ \mathrm{BOT}_{\mathrm{ST}}(\mathtt{t}_1) \end{array}$$

so that

$$\mathrm{A} \vdash^{\{\mathbf{b},\top\}}_{\text{fix1}} \mathtt{fix}\ \mathtt{e} : \mathtt{t}_2$$

annotations $\mathcal{A}$	fix-rules in Chapter 2	fix-rules in Chapter 3
$\{\mathbf{b}, \top\}$	$\vdash^{\mathcal{A}}_{\text{fix1}} = \vdash^{\mathcal{A}}_{\text{fix2}}$	$\vdash^{\mathcal{A}}_{\text{fix1}} = \vdash^{\mathcal{A}}_{\text{fix2}}$
	$\vdash^{\mathcal{A}}_{\text{fix2}} \subset \vdash^{\mathcal{A}}_{\text{fix}}$	$\vdash^{\mathcal{A}}_{\text{fix2}} = \vdash^{\mathcal{A}}_{\text{fix}}$
$\{\mathbf{n}, \mathbf{b}, \top\}$	$\vdash^{\mathcal{A}}_{\text{fix1}} \subset \vdash^{\mathcal{A}}_{\text{fix2}}$	$\vdash^{\mathcal{A}}_{\text{fix1}} \subset \vdash^{\mathcal{A}}_{\text{fix2}}$
	$\vdash^{\mathcal{A}}_{\text{fix2}} \subset \vdash^{\mathcal{A}}_{\text{fix}}$	$\vdash^{\mathcal{A}}_{\text{fix2}} \subset \vdash^{\mathcal{A}}_{\text{fix}}$

Table 3.1: Relation Between the Fix-rules

can be inferred. Since none of the types involves the annotation $\mathbf{n}$ it must be the case that $\text{BOT}_{\text{ST}}(\mathtt{t}) = \mathtt{tt}$ for all types $\mathtt{t}$. We can now construct the proof-tree

$$\dfrac{\dfrac{\text{A} \vdash^{\{\mathbf{b},\top\}}_{\text{fix1}} \mathtt{e} : \mathtt{t}_1 \to \mathtt{t}_2 \qquad \dfrac{\mathtt{t}_2 \leq_{\text{ST}} \mathtt{t}_1}{\mathtt{t}_1 \to \mathtt{t}_2 \leq_{\text{ST}} \mathtt{t}_2 \to \mathtt{t}_2}}{\text{A} \vdash^{\{\mathbf{b},\top\}}_{\text{fix1}} \mathtt{e} : \mathtt{t}_2 \to \mathtt{t}_2} \qquad \text{BOT}_{\text{ST}}(\mathtt{t}_2)}{\text{A} \vdash^{\{\mathbf{b},\top\}}_{\text{fix1}} \mathtt{fix}\ \mathtt{e} : \mathtt{t}_2}\ [\text{fix1}]$$

and this proves our claim.

Claim $\vdash^{\{\mathbf{b},\top\}}_{\text{fix2}} = \vdash^{\{\mathbf{b},\top\}}_{\text{fix}}$

To verify that $\vdash^{\{\mathbf{b},\top\}}_{\text{fix2}} = \vdash^{\{\mathbf{b},\top\}}_{\text{fix}}$ it suffices to show that the rule [fix] can be derived from the rule [fix2]. For this assume

$$\text{A} \vdash^{\{\mathbf{b},\top\}}_{\text{fix2}} \mathtt{e} : \mathtt{t}_1 \to \mathtt{t}_2 \wedge \mathtt{t}_2 \to \mathtt{t}_3 \wedge \ldots\ \mathtt{t}_{n-1} \to \mathtt{t}_n$$

that $\text{BOT}_{\text{ST}}(\mathtt{t}_1)$ is true, and that exists $p < q$ such that $\mathtt{t}_q \leq_{\text{D}} \mathtt{t}_p$ and we want to show that $\text{A} \vdash^{\{\mathbf{b},\top\}}_{\text{fix2}} \mathtt{fix}\ \mathtt{e} : \mathtt{t}_n$ can be inferred. We have

$$\cfrac{\cfrac{\cfrac{A \vdash^{\{\mathbf{b},\top\}}_{\text{fix2}} \texttt{e} : \texttt{t}_1 \rightarrow \texttt{t}_2 \wedge \texttt{t}_2 \rightarrow \texttt{t}_3 \wedge \ldots \texttt{t}_{n-1} \rightarrow \texttt{t}_n}{A \vdash^{\{\mathbf{b},\top\}}_{\text{fix2}} \texttt{e} : (\texttt{t}_p \wedge \ldots \wedge \texttt{t}_n) \rightarrow (\texttt{t}_p \wedge \ldots \wedge \texttt{t}_n)}\,[\text{coer}]}{A \vdash^{\{\mathbf{b},\top\}}_{\text{fix2}} \texttt{fix e} : (\texttt{t}_p \wedge \ldots \wedge \texttt{t}_n)}\,[\text{fix2}]}{A \vdash^{\{\mathbf{b},\top\}}_{\text{fix2}} \texttt{fix e} : \texttt{t}_n}\,[\text{coer}]$$

since we have

$$\begin{array}{ll}
\multicolumn{2}{l}{\texttt{t}_1 \rightarrow \texttt{t}_2 \wedge \texttt{t}_2 \rightarrow \texttt{t}_3 \wedge \ldots \texttt{t}_{n-1} \rightarrow \texttt{t}_n} \\
\leq_{\text{D}} & \texttt{t}_p \rightarrow \texttt{t}_{p+1} \wedge \ldots \texttt{t}_{n-1} \rightarrow \texttt{t}_n \\
\leq_{\text{D}} & (\texttt{t}_p \wedge \ldots \wedge \texttt{t}_n) \rightarrow \texttt{t}_{p+1} \wedge \ldots (\texttt{t}_p \wedge \ldots \wedge \texttt{t}_n) \rightarrow \texttt{t}_n \\
\leq_{\text{D}} & (\texttt{t}_p \wedge \ldots \wedge \texttt{t}_n) \rightarrow \texttt{t}_{p+1} \wedge \ldots (\texttt{t}_p \wedge \ldots \wedge \texttt{t}_n) \rightarrow \texttt{t}_n \wedge (\texttt{t}_{q-1} \rightarrow \texttt{t}_q) \\
\leq_{\text{D}} & (\texttt{t}_p \wedge \ldots \wedge \texttt{t}_n) \rightarrow \texttt{t}_{p+1} \wedge \ldots (\texttt{t}_p \wedge \ldots \wedge \texttt{t}_n) \rightarrow \texttt{t}_n \wedge (\texttt{t}_{q-1} \rightarrow \texttt{t}_p) \\
\leq_{\text{D}} & (\texttt{t}_p \wedge \ldots \wedge \texttt{t}_n) \rightarrow \texttt{t}_p \wedge \ldots (\texttt{t}_p \wedge \ldots \wedge \texttt{t}_n) \rightarrow \texttt{t}_n \\
\leq_{\text{D}} & (\texttt{t}_p \wedge \ldots \wedge \texttt{t}_n) \rightarrow (\texttt{t}_p \wedge \ldots \wedge \texttt{t}_n)
\end{array}$$

Note that we could have used the rule [fix1] in the proof-tree instead of [fix2] and thereby we can show that $\vdash^{\{\mathbf{b},\top\}}_{\text{fix1}} = \vdash^{\{\mathbf{b},\top\}}_{\text{fix}}$ as well.

Claim $\vdash^{\{\mathbf{n},\mathbf{b},\top\}}_{\text{fix1}} \subset \vdash^{\{\mathbf{n},\mathbf{b},\top\}}_{\text{fix2}}$

When we go to the $\{\mathbf{n}, \mathbf{b}, \top\}$-part (both strictness and totality information on the types) the two rules [fix1] and [fix2] are no longer equivalent. Consider the term $\texttt{fix}\ (\lambda\texttt{x}.7)$ and the type $\texttt{Int}^{\mathbf{n}}$. We can infer

$$\emptyset \vdash^{\{\mathbf{n},\mathbf{b},\top\}}_{\text{fix2}} \lambda\texttt{x}.7 : \texttt{Int}^{\mathbf{n}} \rightarrow \texttt{Int}^{\mathbf{n}}$$

but this does not suffice for using the rule [fix1] to infer

$$\emptyset \vdash^{\{\mathbf{n},\mathbf{b},\top\}}_{\text{fix2}} \texttt{fix}\ (\lambda\texttt{x}.7) : \texttt{Int}^{\mathbf{n}}$$

because $\text{BOT}_{\text{ST}}(\texttt{Int}^{\mathbf{n}})$ fails. However we can infer

$$\emptyset \vdash_{\text{ST}} \lambda\texttt{x}.7 : \texttt{Int}^{\top} \rightarrow \texttt{Int}^{\mathbf{n}}$$

and we can then apply the rule [fix2] to get the desired type. This argument shows

$$\neg(A \vdash^{\{\mathbf{n},\mathbf{b},\top\}}_{\text{fix2}} \texttt{e} : \texttt{t} \Rightarrow A \vdash^{\{\mathbf{n},\mathbf{b},\top\}}_{\text{fix1}} \texttt{e} : \texttt{t})$$

and thereby we have $\vdash^{\{\mathbf{n},\mathbf{b},\top\}}_{\text{fix1}} \subset \vdash^{\{\mathbf{n},\mathbf{b},\top\}}_{\text{fix2}}$.

Claim $\vdash^{\{\mathbf{n},\mathbf{b},\top\}}_{\text{fix2}} \subset \vdash^{\{\mathbf{n},\mathbf{b},\top\}}_{\text{fix}}$

To argue that $\vdash^{\{\mathbf{n},\mathbf{b},\top\}}_{\text{fix2}} \subset \vdash^{\{\mathbf{n},\mathbf{b},\top\}}_{\text{fix}}$ when we consider the full strictness and totality analysis we define

$$\begin{aligned}
\texttt{pair} &= \lambda\texttt{x}.\lambda\texttt{y}.\lambda\texttt{z}.\texttt{z x y} \\
\texttt{fst} &= \lambda\texttt{p}.\texttt{p}\ (\lambda\texttt{x}.\lambda\texttt{y}.\texttt{x}) \\
\texttt{snd} &= \lambda\texttt{p}.\texttt{p}\ (\lambda\texttt{x}.\lambda\texttt{y}.\texttt{y}) \\
\texttt{t} &= (\texttt{Int}^{\mathbf{n}} \to \texttt{Int}^{\mathbf{n}} \to \texttt{Int}^{\mathbf{n}}) \to \texttt{Int}^{\mathbf{n}}
\end{aligned}$$

and

$$\texttt{e} = \lambda\texttt{p}.\texttt{pair}\ (\texttt{snd p})\ 3 \tag{3.5}$$

Now we would like to infer

$$\emptyset \vdash \texttt{fix e} : \texttt{t}$$

we can show that $\texttt{e}$ has type $\texttt{t} \to \texttt{t}$:

$$\emptyset \vdash \texttt{e} : \texttt{t} \to \texttt{t}$$

However we are not able to apply the rule [fix2] (or [fix1]), since

$$\text{BOT}_{\text{ST}}(\texttt{t}) = \texttt{ff}$$

Let

$$\begin{aligned}
\texttt{t}_1 &= (\texttt{Int}^{\mathbf{b}} \to \texttt{Int}^{\mathbf{b}} \to \texttt{Int}^{\mathbf{b}}) \to \texttt{Int}^{\mathbf{b}} \\
\texttt{t}_2 &= (\texttt{Int}^{\mathbf{b}} \to \texttt{Int}^{\mathbf{n}} \to \texttt{Int}^{\mathbf{n}}) \to \texttt{Int}^{\mathbf{n}}
\end{aligned}$$

It is *not* possible to infer

$$\emptyset \vdash \texttt{e} : \texttt{t}_1 \to \texttt{t}$$

so that we can apply the rule [fix2]. However we *can* infer

$$\begin{aligned}
\emptyset &\vdash \texttt{e} : \texttt{t}_1 \to \texttt{t}_2 \\
\emptyset &\vdash \texttt{e} : \texttt{t}_2 \to \texttt{t}
\end{aligned}$$

Using the rule [conj] we get

$$\emptyset \vdash \texttt{e} : (\texttt{t}_1 \to \texttt{t}_2) \wedge (\texttt{t}_2 \to \texttt{t}) \wedge (\texttt{t} \to \texttt{t})$$

and we can apply the rule [fix] to get the desired type.

3.3 Denotational Semantics

In Chapter 2 the analysis is proven sound with respect to a natural style operational semantics by defining a validity predicate: $\models \mathtt{e} : \mathtt{ct}$ for both strictness and totality types and for the conjunctions types. Here we need to extend the predicate to $\downarrow$-types. However it is not quite clear how to do this. Therefore we will define the semantics as a denotational semantics:

We have a type-indexed family of domains:

$$\begin{aligned} \mathrm{D}_{\mathtt{B}} &= \mathtt{B}_\perp \\ \mathrm{D}_{\mathtt{ut}_1 \to \mathtt{ut}_2} &= (\mathrm{D}_{\mathtt{ut}_1} \to_{\text{cont}} \mathrm{D}_{\mathtt{ut}_2})_\perp \end{aligned}$$

For the base-types the domain $\mathtt{B}_\perp$ is the lifted domain used for the base-type $\mathtt{B}$, e.g. we have

$$\begin{aligned} \mathrm{D}_{\mathtt{Int}} &= \mathbb{Z}_\perp \\ &= \{\ldots, -2, -1, 0, 1, 2, \ldots\} \cup \{\perp_{\mathtt{Int}}\} \\ \mathrm{D}_{\mathtt{Bool}} &= \{\mathit{true}, \mathit{false}\}_\perp \\ &= \{\mathit{true}, \mathit{false}, \perp_{\mathtt{Bool}}\} \end{aligned}$$

where we have labelled the different bottom elements with the domain that they belong to. For the function space, $\mathrm{D}_{\mathtt{ut}_1 \to \mathtt{ut}_2}$, we use the continuous functions from $\mathrm{D}_{\mathtt{ut}_1}$ to $\mathrm{D}_{\mathtt{ut}_2}$ and we lift this, e.g. we include a bottom-element.

We need the two functions `up` and `dn` to get from a domain to the lifted domain and back again:

$$\begin{aligned} \mathtt{up} &:: \mathrm{D} \to \mathrm{D}_\perp \\ \mathtt{up}(d) &= d \end{aligned} \qquad\qquad \begin{aligned} \mathtt{dn} &:: \mathrm{D}_\perp \to \mathrm{D} \\ \mathtt{dn}(d) &= \begin{cases} \perp_{\mathrm{D}}, & \text{if } \mathrm{d} = \perp_{\mathrm{D}_\perp} \\ d, & \text{otherwise} \end{cases} \end{aligned}$$

We also need an environment ρ that assigns denotations to variables. This is a partial function from variables to the disjoint union of the domains.

Now the semantics assigns denotations to terms, meaning that if we have $\emptyset \vdash \mathtt{e} : \mathtt{ut}$, then $[\![\mathtt{e}]\!]$ is a partial function from environments to $\mathrm{D}_{\mathtt{ut}}$ (Figure 3.3). For each constant, $\mathtt{c}$, there is a unique predefined denotation c, e.g.

$$[\![\mathtt{true}]\!] = \mathit{true}$$

$$
\begin{aligned}
[\![\mathtt{x}]\!]\ \rho &= \rho\ (\mathtt{x}) \\
[\![\lambda\mathtt{x.e}]\!]\ \rho &= \mathtt{up}(\lambda d.[\![\mathtt{e}]\!]\ \rho[d/\mathtt{x}]) \\
[\![\mathtt{e}_1\ \mathtt{e}_2]\!]\ \rho &= \mathtt{dn}([\![\mathtt{e}_1]\!]\ \rho)([\![\mathtt{e}_2]\!]\ \rho) \\
[\![\mathtt{fix\ e}]\!]\ \rho &= \sqcup_n d_n \text{where} \\
& \quad d_0 = \bot \\
& \quad d_{n+1} = \mathtt{dn}([\![\mathtt{e}]\!]\ \rho) d_n \\
[\![\mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3]\!]\ \rho &= \begin{cases} \bot, & \text{if} [\![\mathtt{e}_1]\!]\ \rho = \bot_{\mathrm{D}_{\mathtt{Bool}}} \\ [\![\mathtt{e}_2]\!]\ \rho, & \text{if} [\![\mathtt{e}_1]\!]\ \rho = \mathit{true} \\ [\![\mathtt{e}_3]\!]\ \rho, & \text{if} [\![\mathtt{e}_1]\!]\ \rho = \mathit{false} \end{cases} \\
[\![\mathtt{c}]\!]\ \rho &= c
\end{aligned}
$$

Figure 3.3: Denotational Semantics for the λ-calculus

$$
\begin{aligned}
[\![\mathtt{false}]\!] &= \mathit{false} \\
[\![7]\!] &= 7 \\
[\![+]\!] &= \lambda d_1.\lambda d_2. + d_1 d_2
\end{aligned}
$$

3.3.1 Relation Between the Semantics

We will now sketch how to relate the above denotational semantics to the natural style operational semantics in Chapter 2. More details can be found in [Ben93, Plo77].

The natural-style operational semantics in Chapter 2 shows how the terms evaluate to WHNF's whereas the denotational semantics gives denotations to terms. We can easily related the WHNF `7` with the denotation 7 and the WHNF `true` to the denotation *true*. How can we relate the WHNF λx.`e` to a function f? The only thing we can do with a function is to apply it. The one thing that can be observed from a term is whether it evaluated to a WHNF of a base-type or not. This motivates the definition of *operational meaning*:

$$
\mathcal{O}\ [\![\mathtt{e}]\!] = \begin{cases} \mathtt{v}, & \text{if} \vdash \mathtt{e} \Downarrow \mathtt{v} \\ \bot, & \text{otherwise} \end{cases}
$$

Whenever $\mathtt{e}$ is closed we will write $[\![\mathtt{e}]\!]$ for the denotational semantics of $\mathtt{e}$, i.e. we do not write the empty environment. Now for all closed terms of base-types we want the operational meaning and the denotational semantics to agree:

Proportion 3.2
Given a term, $\mathtt{e}$, of a base-type

$$\mathcal{O}\ [\![\mathtt{e}]\!] = [\![\mathtt{e}]\!]$$

□

We will show this by first proving that $\mathcal{O}\ [\![\mathtt{e}]\!] \leq [\![\mathtt{e}]\!]$ and then $\mathcal{O}\ [\![\mathtt{e}]\!] \geq [\![\mathtt{e}]\!]$.

Lemma 3.3
$(\vdash \mathtt{e} \Downarrow \mathtt{v}) \Rightarrow ([\![\mathtt{e}]\!] = \mathtt{v})$ □

Proof We show by induction of the proof tree of $\vdash \mathtt{e} \Downarrow \mathtt{v}$ that $[\![\mathtt{e}]\!] = [\![\mathtt{v}]\!]$ and hence since $[\![\mathtt{v}]\!] = \mathtt{v}$ that $[\![\mathtt{e}]\!] = \mathtt{v}$. ■

For the second part we need more than just induction on the terms or types: let $\{\mathcal{R}^{\mathtt{t}}\}$ be a type-indexed family of relations which relate denotations and terms. The relation is defined by

$$\begin{aligned} (d)\ \mathcal{R}^{\mathtt{B}}\ (\mathtt{e}) &\Leftrightarrow d \leq \mathcal{O}\ [\![\mathtt{e}]\!] \\ (d)\ \mathcal{R}^{\mathtt{t}_1 \to \mathtt{t}_2}\ (\mathtt{e}) &\Leftrightarrow \forall d_1, \mathtt{e}_1 : (d_1)\ \mathcal{R}^{\mathtt{t}_1}\ (\mathtt{e}_1) \Rightarrow (d\ d_1)\ \mathcal{R}^{\mathtt{t}_2}\ (\mathtt{e}\ \mathtt{e}_1) \end{aligned}$$

Lemma 3.4
We have

1. For all closed terms, $\mathtt{e}$, of a base-type we have

$$(\bot_{\mathrm{D}_{\mathtt{B}}})\ \mathcal{R}^{\mathtt{B}}\ (\mathtt{e})$$

2. For all closed terms, $\mathtt{e}$, of a base-type and whenever $\{d_n\}$ is a chain in $\mathrm{D}_{\mathtt{t}}$ and for all n we have $(d_n)\ \mathcal{R}^{\mathtt{t}}\ (\mathtt{e})$ then

$$(\sqcup d_n)\ \mathcal{R}^{\mathtt{t}}\ (\mathtt{e})$$

3. For closed terms, $\mathtt{e}_1$ and $\mathtt{e}_2$, and denotations, $d \in \mathrm{D}_{\mathtt{t}}$, then

$$\begin{gathered} \vdash \mathtt{e}_1 \Downarrow \mathtt{v} \Rightarrow \vdash \mathtt{e}_2 \Downarrow \mathtt{v} \\ \Downarrow \\ (d)\ \mathcal{R}^{\mathtt{t}}\ (\mathtt{e}_1) \Rightarrow (d)\ \mathcal{R}^{\mathtt{t}}\ (\mathtt{e}_2) \end{gathered}$$

4. Assume that A and ρ satisfies that for each $\mathtt{x}_i \in \mathtt{dom}(\rho)$ there is a closed term $\mathtt{e}_i$ such that $(\rho(\mathtt{x}_i))\ \mathcal{R}^{\mathcal{A}(\mathtt{x}_i)}\ (\mathtt{e}_i)$. For any term, $\mathtt{e}$, we have

$$\begin{array}{c} \mathrm{FV}(\mathtt{e}) \subseteq \mathtt{dom}(\rho) \wedge \mathrm{A} \vdash \mathtt{e} : \mathtt{t} \\ \Downarrow \\ ([\![\mathtt{e}]\!]\ \rho)\ \mathcal{R}^{\mathtt{t}}\ (\mathtt{e}[\mathtt{e}_i/\mathtt{x}_i]) \end{array}$$

□

Proof All parts are proven by induction on the type $\mathtt{t}$. ■

From Part 4 follows

Lemma 3.5
$\mathcal{O}\ [\![\mathtt{e}]\!] \geq [\![\mathtt{e}]\!]$ □

and we have proved Proposition 3.2.

3.4 Soundness

In this section we will prove the analysis in Figure 3.2 sound with respect to the denotational semantics in Figure 3.3.

To each strictness and totality type $\mathtt{t}$ we will relate a subset of $\mathrm{D}_{\varepsilon(\mathtt{t})}$.

(Figure 3.4). The set $[\![\mathtt{ut}^{\mathbf{n}}]\!]$ is the set of denotations in $\mathrm{D}_{\mathtt{ut}}$ which are not bottom whereas the set $[\![\mathtt{ut}^{\mathbf{b}}]\!]$ is only the denotation bottom from the domain $\mathrm{D}_{\mathtt{ut}}$. The set $[\![\mathtt{ut}^{\top}]\!]$ is just all the denotations in $\mathrm{D}_{\mathtt{ut}^{\top}}$. The functions in $[\![\mathtt{t}_1 \rightarrow \mathtt{t}_2]\!]$ must map elements of $[\![\mathtt{t}_1]\!]$ to $[\![\mathtt{t}_2]\!]$. For conjunctions we take the intersection of the two sets. And for the $\downarrow$-types, $\downarrow\mathtt{t}$, we take the downwards closure of the set for type itself, $\mathtt{t}$. We define the downwards closure of a subset of a domain:

$$\mathtt{dc}(X) \ = \ \{d' \mid \exists d \in X : d' \leq d\}$$

Definition 3.6
A subset X of a domain $\mathrm{D}_{\mathtt{ut}}$ is *limit closed* if whenever $d_0 \sqsubseteq d_1 \sqsubseteq \cdots$ is a chain in $\mathrm{D}_{\mathtt{ut}}$ and $\forall i : d_i \in X$ then $\sqcup_i d_i \in X$ and it is *convex* if whenever $d_1 \sqsubseteq d_2 \sqsubseteq d_3 \in \mathrm{D}_{\mathtt{ut}}$ and $d_1 \in X$ and $d_3 \in \mathrm{X}$ then $d_2 \in X$. □

$$
\begin{array}{rcl}
[\![\mathtt{ut}^{\mathbf{n}}]\!] & = & \mathrm{D}_{\mathtt{ut}} \backslash \{\perp_{\mathrm{D}_{\mathtt{ut}}}\} \\
[\![\mathtt{ut}^{\mathbf{b}}]\!] & = & \{\perp_{\mathrm{D}_{\mathtt{ut}}}\} \\
[\![\mathtt{ut}^{\top}]\!] & = & \mathrm{D}_{\mathtt{ut}} \\
[\![\mathtt{t}_1 \rightarrow \mathtt{t}_2]\!] & = & \{f \in \mathrm{D}_{\varepsilon(\mathtt{t}_1 \rightarrow \mathtt{t}_2)} \mid \mathtt{dn}(f)[\![\mathtt{t}_1]\!] \subseteq [\![\mathtt{t}_2]\!]\} \\
[\![\mathtt{t}_1 \wedge \mathtt{t}_2]\!] & = & [\![\mathtt{t}_1]\!] \cap [\![\mathtt{t}_2]\!] \\
[\![\downarrow\mathtt{t}]\!] & = & \mathtt{dc}([\![\mathtt{t}]\!])
\end{array}
$$

Figure 3.4: The Meaning of the Strictness and Totality Types

First we prove that each $[\![\mathtt{t}]\!]$ is a limit-closed subset of $\mathrm{D}_{\varepsilon(\mathtt{t})}$ and convex:

Proportion 3.7 *Limit Closed subsets*
The set $[\![\mathtt{t}]\!]$ is a limit closed and convex subset of $\mathrm{D}_{\varepsilon(\mathtt{t})}$. □

Proof We assume that $d_0 \sqsubseteq d_1 \sqsubseteq \cdots$ is a chain in $\mathrm{D}_{\varepsilon(\mathtt{t})}$ such that for all i we have $d_i \in [\![\mathtt{t}]\!]$; then we show by induction on $\mathtt{t}$ that $\sqcup_i d_i \in [\![\mathtt{t}]\!]$ holds. Next we assume that $d_1 \sqsubseteq d_2 \sqsubseteq d_3$ and both d_1 and d_3 are in $[\![\mathtt{t}]\!]$; then we show by induction on $\mathtt{t}$ that d_2 is in $[\![\mathtt{t}]\!]$.

For the full proof see Appendix page 309. ■

The predicate $\mathrm{BOT}_{\mathrm{ST}}$ is sound but it is not complete. Whenever the predicate is true, then bottom is a member of the denotation of the type:

Lemma 3.8
$(\mathrm{BOT}_{\mathrm{ST}}(\mathtt{t}) = \mathtt{tt}) \Rightarrow (\perp_{\mathrm{D}_{\varepsilon(\mathtt{t})}} \in [\![\mathtt{t}]\!])$ □

Proof We assume that $\mathrm{BOT}_{\mathrm{ST}}(\mathtt{t})$ is true and then we prove by induction on $\mathtt{t}$ that $\perp_{\mathrm{D}_{\varepsilon(\mathtt{t})}} \in [\![\mathtt{t}]\!]$ holds.

For the full proof see Appendix page 311. ■

Next we want to prove that the coercion rules are sound:

Lemma 3.9 *Soundness of coercions*
If $\mathtt{t}_1 \leq_{\mathrm{D}} \mathtt{t}_2$ then $[\![\mathtt{t}_1]\!] \subseteq [\![\mathtt{t}_2]\!]$. □

Proof We assume $\mathtt{t}_1 \leq_{\mathrm{D}} \mathtt{t}_2$ and then we prove by induction on the proof-tree for $\mathtt{t}_1 \leq_{\mathrm{D}} \mathtt{t}_2$ that $[\![\mathtt{t}_1]\!] \subseteq [\![\mathtt{t}_2]\!]$ holds.

For the full proof see Appendix page 312. ■

The validity predicate $\models$ is defined for denotations and properties and extended to environments:

Definition 3.10 *Validity*
$d \models \mathtt{t} \Leftrightarrow (d \in [\![\mathtt{t}]\!])$
$\rho \models \mathrm{A} \Leftrightarrow (\mathrm{dom}(\mathrm{A}) = \mathrm{dom}(\rho) \wedge \forall\, \mathtt{x} \in \mathrm{dom}(\rho) : \rho\ (\mathtt{x}) \models \mathrm{A}(\mathtt{x}))$ □

Now soundness is:

Proportion 3.11 *Soundness*
$\mathrm{A} \vdash \mathtt{e} : \mathtt{t} \Rightarrow (\forall\ \rho : \rho \models \mathrm{A} \Rightarrow [\![\mathtt{e}]\!]\ \rho \models \mathtt{t})$ □

Proof We assume $\mathrm{A} \vdash \mathtt{e} : \mathtt{t}$ and $\rho \models \mathrm{A}$; then we show by induction on the proof-tree for $\mathrm{A} \vdash \mathtt{e} : \mathtt{t}$ that $[\![\mathtt{e}]\!]\ \rho \models \mathtt{t}$ holds.

The case [abs]: We assume $\mathrm{A} \vdash \lambda\mathtt{x}.\mathtt{e} : \mathtt{t}_1 \rightarrow \mathtt{t}_2$. From the [abs]-rule we get

$$\mathrm{A}, \mathtt{x} : \mathtt{t}_1 \vdash \mathtt{e} : \mathtt{t}_2$$

by applying the induction hypothesis we get

$$(\forall \rho' : \rho' \models \mathrm{A}, \mathtt{x} : \mathtt{t}_1) \Rightarrow ([\![\mathtt{e}]\!]\ \rho' \models \mathtt{t}_2) \tag{3.6}$$

We want to show

$$[\![\lambda\mathtt{x}.\mathtt{e}]\!]\ \rho \models \mathtt{t}_1 \rightarrow \mathtt{t}_2$$

which is equivalent to show

$$[\![\lambda\mathtt{x}.\mathtt{e}]\!]\ \rho \in [\![\mathtt{t}_1 \rightarrow \mathtt{t}_2]\!]$$

which is equivalent to

$$[\![\lambda\mathtt{x}.\mathtt{e}]\!]\ \rho \in \{f \in \mathrm{D}_{\varepsilon(\mathtt{t}_1 \rightarrow \mathtt{t}_2)} \mid \mathtt{dn}(f)[\![\mathtt{t}_1]\!] \subseteq [\![\mathtt{t}_2]\!]\}$$

which is equivalent to

$$\mathtt{up}(\lambda d.[\![\mathtt{e}]\!]\ \rho[d/\mathtt{x}]) \in \{f \in \mathrm{D}_{\varepsilon(\mathtt{t}_1 \rightarrow \mathtt{t}_2)} \mid \mathtt{dn}(f)[\![\mathtt{t}_1]\!] \subseteq [\![\mathtt{t}_2]\!]\}$$

which is equivalent to

$$\forall d \in [\![\texttt{t}_1]\!] : ([\![\texttt{e}]\!]\ \rho[d/\texttt{x}]) \in [\![\texttt{t}_2]\!]$$

Now let $\rho' = \rho[d/\texttt{x}]$ in (3.6) and we get

$$[\![\texttt{e}]\!]\ \rho[d/\texttt{x}] \models \texttt{t}_2$$

as required.

The case [if2]: We assume $\mathrm{A} \vdash \texttt{if e}_1 \texttt{ then e}_2 \texttt{ else e}_3 : \texttt{t}$. From the [if2]-rule we get

$$\begin{aligned} &\mathrm{A} \vdash \texttt{e}_1 : \texttt{Bool}^{\mathbf{n}} \\ &\mathrm{A} \vdash \texttt{e}_2 : \texttt{t} \\ &\mathrm{A} \vdash \texttt{e}_3 : \texttt{t} \end{aligned}$$

by applying the induction hypothesis we get

$$\begin{aligned} &[\![\texttt{e}_1]\!]\ \rho \models \texttt{Bool}^{\mathbf{n}} \\ &[\![\texttt{e}_2]\!]\ \rho \models \texttt{t} \\ &[\![\texttt{e}_3]\!]\ \rho \models \texttt{t} \end{aligned}$$

now we have

$$\begin{aligned} &[\![\texttt{if e}_1 \texttt{ then e}_2 \texttt{ else e}_3]\!]\ \rho \\ &= \begin{cases} [\![\texttt{e}_2]\!]\ \rho, & \text{if} [\![\texttt{e}_1]\!]\ \rho = \mathit{true} \\ [\![\texttt{e}_3]\!]\ \rho, & \text{if} [\![\texttt{e}_1]\!]\ \rho = \mathit{false} \end{cases} \in [\![\texttt{t}]\!] \end{aligned}$$

as required.

The case [fix]: We assume $\mathrm{A} \vdash \texttt{fix e : t}_n$ and $\mathrm{BOT}_{\mathrm{ST}}(\texttt{t}_1)$ and

$$\exists p, q : p < q \wedge \texttt{t}_q \leq_{\mathrm{D}} \texttt{t}_p$$

From the [fix]-rule we get

$$\mathrm{A} \vdash \texttt{e} : \texttt{t}_1 \rightarrow \texttt{t}_2 \wedge \texttt{t}_2 \rightarrow \texttt{t}_3 \wedge \cdots \wedge \texttt{t}_{n-1} \rightarrow \texttt{t}_n$$

by applying the induction hypothesis we get

$$[\![\texttt{e}]\!]\ \rho \models \texttt{t}_1 \rightarrow \texttt{t}_2 \wedge \texttt{t}_2 \rightarrow \texttt{t}_3 \wedge \cdots \wedge \texttt{t}_{n-1} \rightarrow \texttt{t}_n$$

We have

$$\begin{aligned}
&[\![\mathtt{e}]\!]\ \rho \models \mathtt{t}_1 \to \mathtt{t}_2\\
&[\![\mathtt{e}]\!]\ \rho \models \mathtt{t}_2 \to \mathtt{t}_3\\
&\vdots\\
&[\![\mathtt{e}]\!]\ \rho \models \mathtt{t}_{n-1} \to \mathtt{t}_n
\end{aligned}$$

We have from Proposition 3.8 we have

$$d_0 = \bot_{\mathrm{D}_{\varepsilon(\mathtt{t}_1)}} \in [\![\mathtt{t}_1]\!]$$

and we have

$$\begin{aligned}
d_1 &= \mathtt{dn}([\![\mathtt{e}]\!]\ \rho)\bot_{\mathrm{D}_{\varepsilon(\mathtt{t}_1)}} \in [\![\mathtt{t}_2]\!]\\
d_2 &= (\mathtt{dn}([\![\mathtt{e}]\!]\ \rho))^2\bot_{\mathrm{D}_{\varepsilon(\mathtt{t}_1)}} \in [\![\mathtt{t}_3]\!]\\
&\vdots\\
d_{q-1} &= (\mathtt{dn}([\![\mathtt{e}]\!]\ \rho))^{q-1}\bot_{\mathrm{D}_{\varepsilon(\mathtt{t}_1)}} \in [\![\mathtt{t}_q]\!]
\end{aligned}$$

since $\mathtt{t}_q \ \leq_{\mathrm{D}}\ \mathtt{t}_p$ we have from Proposition 3.9

$$d_{q-1} = (\mathtt{dn}([\![\mathtt{e}]\!]\ \rho))^{q-1}\bot_{\mathrm{D}_{\varepsilon(\mathtt{t}_1)}} \in [\![\mathtt{t}_p]\!]$$

We arrive at

$$\forall k \geq 0 : d_{q-1+(q-p)k} = (\mathtt{dn}([\![\mathtt{e}]\!]\ \rho))^{q-1+(q-p)k}\bot_{\mathrm{D}_{\varepsilon(\mathtt{t}_1)}} \in [\![\mathtt{t}_q]\!]$$

This is a chain in $[\![\mathtt{t}_q]\!]$ since $[\![\mathtt{e}]\!]\ \rho$ is monotonic. From Proposition 3.7 we have $\sqcup_i d_n = [\![\mathtt{fix\ e}]\!]\ \rho$ is in $[\![\mathtt{t}_q]\!]$. Now

$$(\mathtt{dn}([\![\mathtt{e}]\!]\ \rho))([\![\mathtt{fix\ e}]\!]\ \rho) \in [\![\mathtt{t}_{q+1}]\!]$$

and we have

$$[\![\mathtt{fix\ e}]\!]\ \rho \in [\![\mathtt{t}_{q+1}]\!]$$

We may now arrive at

$$[\![\mathtt{fix\ e}]\!]\ \rho \in [\![\mathtt{t}_n]\!]$$

as required.

The case [coer]: We assume $\mathrm{A} \vdash \mathtt{e} : \mathtt{t}_2$ and $\mathtt{t}_1 \leq_{\mathrm{D}} \mathtt{t}_2$. From the [coer]-rule we get

$$\mathrm{A} \vdash \mathtt{e} : \mathtt{t}_1$$

by applying the induction hypothesis we get

$$[\![\mathtt{e}]\!]\ \rho \models \mathtt{t}_1$$

since $\mathtt{t}_1 \leq_{\mathrm{D}} \mathtt{t}_2$ we get from Proposition 3.9

$$[\![\mathtt{e}]\!]\ \rho \models \mathtt{t}_2$$

as required.

For the remaining cases see Appendix page 318. ■

Semantic soundness of the underlying type inference system follows from Lemma 2.5 and soundness of the strictness and totality analysis:

Corollary 3.12
$\mathrm{A} \vdash \mathtt{e} : \mathtt{ut} \Rightarrow (\forall \rho : (\forall \mathtt{x} \in \mathrm{dom}(\mathrm{A}) : \rho\mathtt{x} \in \mathrm{dom}(\mathrm{A})) \Rightarrow [\![\mathtt{e}]\!]\ \rho \in \mathrm{D}_{\mathtt{ut}}\ )$ □

3.5 Summary

In this Chapter:

- We have removed the restriction that conjunctions may only occur at the top-level; i.e. we have "full" conjunction.
- In order to define the monotonicity-rule we need the ↓-operation also on conjunction type. However it is not clear how to define this, therefore we let ↓ be part of the syntax as a type constructor.
- Now since ↓ is part of the types it is not clear how to define validity for ↓-types using the natural style operational semantics defined in Chapter 2. Hence we define a denotational semantics.
- We show the analysis sound with respect to the denotational semantics.

The soundness proof in this Chapter is much smaller and elegant than the one in Chapter 2. One reason is that in order to prove soundness of the fixpoint rule in Chapter 2 we need to introduce special terms (i.e. $\mathtt{fix}_n$) and show how they relate to the "normal" terms (i.e. `fix`). Here we can use that the sets $[\![\mathtt{t}]\!]$ are limit closed (Proposition 3.7).

In Chapter 2 in the proof of soundness of the coercions in the case [monotone] we had to construct an terminating term from an arbitrary term. Here we use that the sets $[\![\mathtt{t}]\!]$ are convex (Proposition 3.7).

Chapter 4

Inference Algorithms

So far we have described the information that we want: In Chapter 1 we presented an inference system that tells how terms and standard types are related. In Chapter 2 and 3 we have defined inference systems for how strictness and totality types relate to terms.

We can now ask two different questions:

- given a term, `e`, does there exists a type, `t`, such that $\mathrm{A} \vdash \mathtt{e} : \mathtt{t}$ can be inferred?
- given a term, `e`, *and* a type, `t`, is it true that $\vdash \mathtt{e} : \mathtt{t}$ can be inferred?

The first question can be answered by a *type inference algorithm*, i.e. the algorithm computes the type provided it does exists. The second question can be answered by a *type checking algorithm*, i.e. the algorithm checks that a proof-tree can be constructed.

For the standard types there are two algorithms in the literature: the algorithm $\mathcal{W}$ by Milner [Mil78] and the algorithm $\mathcal{T}$ by Damas [Dam85].

A complete type inference algorithm is also a type checking algorithm, since we can use the type inference algorithm to infer a type and then check the inferred type against the given type. Whenever the set of types is finite we can use the type checking algorithm to define a type inference algorithm, by checking each type. Hence we will refer to both kinds of algorithms as type inference algorithms.

For the strictness and totality inference system in Chapter 2 and 3 it is trivial to construct a sound type inference algorithm: use a standard inference algorithm to compute the standard type, `ut`, of the term, now the term will have the strictness and totality type $\mathtt{ut}^{\top}$. This is not what we are interested in: we want to infer a more informative type. The type that the inference algorithm may compute is the *most general type* [HM94a], i.e. the conjunction of all the types that can be infer for the term. We cannot just choose to compute one of the conjuncts. To see this consider the most general type of $\lambda\mathtt{x}.\mathtt{x}$:

$$\begin{aligned}&(\mathtt{Int}^{\mathbf{n}} \rightarrow \mathtt{Int}^{\top}) \wedge (\mathtt{Int}^{\mathbf{b}} \rightarrow \mathtt{Int}^{\top}) \wedge (\mathtt{Int}^{\mathbf{n}} \rightarrow \mathtt{Int}^{\mathbf{n}})\\ &\quad \wedge(\mathtt{Int}^{\mathbf{b}} \rightarrow \mathtt{Int}^{\mathbf{b}}) \wedge (\mathtt{Int}^{\top} \rightarrow \mathtt{Int}^{\top}) \wedge (\mathtt{Int} \rightarrow \mathtt{Int})^{\mathbf{n}} \wedge (\mathtt{Int} \rightarrow \mathtt{Int})^{\top}\end{aligned}$$

Note that some of these conjuncts are incomparable, so there is no least type.

The most general type approach can be used to construct an type inference algorithm for both the analysis in Chapter 2 and the analysis in Chapter3. However, for the analysis in Chapter 3 we can do even better: we can construct a type checking algorithm using the *lazy type* approach by Hankin and Le Métayer [HM94a].

Overview In Section 4.1 we will review the standard type inference algorithm $\mathcal{T}$ by Damas [Dam85]. In Section 2 we present a strictness and totality inference algorithm following the lazy type approach by [HM94a]. Syntactic soundness of the algorithm is proven in Section 4.3 and completeness of the algorithm is discussed.

4.1 Standard Type Inference Algorithms

We extend the standard types with type variables:

$$\mathtt{t} ::= \mathtt{B} \mid \mathtt{t} \rightarrow \mathtt{t} \mid \alpha$$

where α range over type variables. Now we can infer the types in a bottom-up manner: Consider the term $\lambda\mathtt{x}.\mathtt{x}$: we will give it the type $\alpha \rightarrow \alpha$. Then, when we discover that $\lambda\mathtt{x}.\mathtt{x}$ is applied to a term of type $\mathtt{Bool}$ we are going to *unify* the type variable, α, with the type $\mathtt{Bool}$.

First we will review the unification algorithm by Robinson [Rob65].

$$
\begin{array}{lcl}
\mathcal{U}(\alpha_1, \alpha_2) & = & [\alpha_1/\alpha_2] \\
\mathcal{U}(\alpha, \mathtt{t}) & = & [\mathtt{t}/\alpha], \text{ if } \ \alpha \notin \mathrm{FV}(\mathtt{t}) \\
\mathcal{U}(\mathtt{t}, \alpha) & = & \mathcal{U}(\alpha, \mathtt{t}) \\
\mathcal{U}(\mathtt{t}_1 \rightarrow \mathtt{t}_2, \mathtt{t}_3 \rightarrow \mathtt{t}_4) & = & \ \text{let} \quad S_1 = \mathcal{U}(\mathtt{t}_1, \mathtt{t}_3) \\
 & & \qquad\quad S_2 = \mathcal{U}(S_1\mathtt{t}_2, S_1\mathtt{t}_4) \\
 & & \ \text{in} \quad S_2 \circ S_1 \\
\mathcal{U}(\mathtt{t}_1, \mathtt{t}_2) & = & \mathtt{FAIL}, \text{ otherwise}
\end{array}
$$

Figure 4.1: The Algorithm $\mathcal{U}$

Definition 4.1
A substitution S is a partial function from type variables to types. A substitution S applied to a type will replace all type variables α with $S\alpha$. A *ground* substitution is a substitution where the type variables are mapped to closed types, i.e. there are no type variables left in the types.
□

We will write $[\mathtt{t}/\alpha]$ for the substitution that maps α to $\mathtt{t}$ and any other type variable is mapped to itself.

Now the algorithm $\mathcal{U}$ displayed in Figure 4.1 will given two types, $\mathtt{t}_1$ and $\mathtt{t}_2$, return a substitution, S, such that $S\mathtt{t}_1 = S\mathtt{t}_2$, or it will `FAIL`. When two type variables are going to be unified, we map one type variable to the other type variable. A type variable and a type can be unified whenever the type variable does not occur in the type. Two function types, $\mathtt{t}_1 \rightarrow \mathtt{t}_2$ and $\mathtt{t}_3 \rightarrow \mathtt{t}_4$, can be unified provided $\mathtt{t}_1$ and $\mathtt{t}_3$ can be unified and $\mathtt{t}_2$ and $\mathtt{t}_4$ can be unified. The result of the first unification is applied to $\mathtt{t}_2$ and $\mathtt{t}_4$ before unifying them. In all other cases we are not able to unify the two types, e.g. `Int` and `Bool`is not unifiable.

Theorem 4.2
If $\mathcal{U}(\mathtt{t}_1, \mathtt{t}_2) = S$ then

- $S\mathtt{t}_1 = S\mathtt{t}_2$
- whenever a substitution, R, unifies $\mathtt{t}_1$ and $\mathtt{t}_2$, then for some substitution S': $R = S' \circ S$
- dom $(S) \subseteq \mathrm{FV}(\mathtt{t}_1) \cup \mathrm{FV}(\mathtt{t}_2)$

□

Proof See [Rob65]. ■

Example 4.3
Consider the two types $\alpha_1 \to \alpha_2$ and $\alpha_2 \to \alpha_3$. We have

$$\begin{array}{rcll} S = \mathcal{U}(\alpha_1 \to \alpha_2,\, \alpha_2 \to \alpha_3) & = & \text{let} & S_1 = \mathcal{U}(\alpha_1,\, \alpha_2) \\ & & & S_2 = \mathcal{U}(S_1\alpha_2,\, S_1\alpha_3) \\ & & \text{in} & S_2 \circ S_1 \\ & = & \text{let} & S_1 = [\alpha_1/\alpha_2] \\ & & & S_2 = \mathcal{U}(S_1\alpha_2,\, S_1\alpha_3) \\ & & \text{in} & S_2 \circ S_1 \\ & = & \multicolumn{2}{l}{\mathcal{U}(\alpha_1,\, \alpha_3) \circ [\alpha_1/\alpha_2]} \\ & = & \multicolumn{2}{l}{[\alpha_1/\alpha_3] \circ [\alpha_1/\alpha_2]} \end{array}$$

and we have

$$S(\alpha_1 \to \alpha_2) = \alpha_1 \to \alpha_1 = S(\alpha_2 \to \alpha_3)$$

as required. □

4.1.1 The Algorithm $\mathcal{T}$

The algorithm $\mathcal{T}$ was first described by Damas [Dam85]. The algorithm $\mathcal{T}$ will when applied to a term, return a list of assumptions and a type. The assumption list is the assumption made typing the term. For our language the algorithm is given in Figure 4.2.

For variables we assign a fresh type variable to `x` in the list of assumptions and this type variable is the type of the term.

When we have analysed the body of an abstraction we have a list of assumptions and the type of the body. Whenever `x` is in the assumption list we use the type recorded by the assumptions to define the type of the abstraction. However, when `x` is not in the list of assumptions we use a fresh type variable to define the type of the abstraction.

For application we first analyse $\mathtt{e}_1$ and $\mathtt{e}_2$. Next we must ensure that the type of $\mathtt{e}_1$ is a function type and that it can be applied to the argument, $\mathtt{e}_2$, i.e. we must ensure that the type $\mathtt{t}_1$ is of the form $\mathtt{t}_2 \to \alpha$. We do this by unifying $\mathtt{t}_1$ with the type $\mathtt{t}_2 \to \alpha$. Further we must ensure that the two

$$
\begin{array}{lcll}
\mathcal{T}(\mathtt{x}) & = & \text{let} & \alpha \text{ be a fresh type variable} \\
 & & \text{in} & (\mathtt{x} : \alpha,\ \alpha) \\
\mathcal{T}(\lambda\mathtt{x}.\mathtt{e}) & = & \text{let} & \alpha \text{ be a fresh type variable} \\
 & & & (\mathrm{A},\ \mathtt{t}) = \mathcal{T}(\mathtt{e}) \\
 & & \text{in} & \ \text{if} \quad \mathtt{x} \in \mathrm{dom}(\mathrm{A}) \text{ then} \\
 & & & \qquad\quad (\mathrm{A} \setminus \mathtt{x},\ \mathrm{A}(\mathtt{x}) \to \mathtt{t}) \\
 & & & \ \text{else} \\
 & & & \qquad\quad ((\mathrm{A},\ \alpha \to \mathtt{t}) \\
\mathcal{T}(\mathtt{e}_1\ \mathtt{e}_2) & = & \text{let} & (\mathrm{A}_1,\ \mathtt{t}_1) = \mathcal{T}(\mathtt{e}_1) \\
 & & & (\mathrm{A}_2,\ \mathtt{t}_2) = \mathcal{T}(\mathtt{e}_2) \\
 & & & \alpha \text{ be a fresh type variable} \\
 & & & S_1 = \mathcal{U}(\mathtt{t}_1,\ \mathtt{t}_2 \to \alpha) \\
 & & & S_2 = \mathcal{U}(S_1\mathrm{A}_1,\ S_1\mathrm{A}_2) \\
 & & & S = S_2 \circ S_1 \\
 & & \text{in} & (S\mathrm{A}_1 \cup S\mathrm{A}_2,\ S\alpha) \\
\multicolumn{4}{l}{\mathcal{T}(\mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3)} \\
 & = & \text{let} & (\mathrm{A}_1,\ \mathtt{t}_1) = \mathcal{T}(\mathtt{e}_1) \\
 & & & (\mathrm{A}_2,\ \mathtt{t}_2) = \mathcal{T}(\mathtt{e}_2) \\
 & & & (\mathrm{A}_3,\ \mathtt{t}_3) = \mathcal{T}(\mathtt{e}_3) \\
 & & & S_1 = \mathcal{U}(\mathtt{t}_1,\ \mathtt{Bool}) \\
 & & & S_2 = \mathcal{U}(S_1\mathtt{t}_2,\ S_1\mathtt{t}_3) \\
 & & & S_3 = \mathcal{U}((S_2 \circ S_1)\mathrm{A}_1,\ (S_2 \circ S_1)\mathrm{A}_2,\ (S_2 \circ S_1)\mathrm{A}_3) \\
 & & & S = S_3 \circ S_2 \circ S_1 \\
 & & \text{in} & (S\mathrm{A}_1 \cup S\mathrm{A}_2 \cup S\mathrm{A}_3,\ S\mathtt{t}_2) \\
\mathcal{T}(\mathtt{fix}\ \mathtt{e}) & = & \text{let} & \alpha \text{ be a fresh type variable} \\
 & & & (\mathrm{A},\ \mathtt{t}) = \mathcal{T}(\mathtt{e}) \\
 & & & S = \mathcal{U}(\mathtt{t},\ \alpha \to \alpha) \\
 & & \text{in} & (S\mathrm{A},\ S\alpha) \\
\mathcal{T}(\mathtt{c}) & = & \multicolumn{2}{l}{([\,],\ \mathtt{t}_{\mathtt{c}})}
\end{array}
$$

Figure 4.2: The Algorithm $\mathcal{T}$

lists of assumptions agree on the assumptions for the same variable. We can do this by unifying the types for the same variable. Before we unify the types we must apply the substitution from the first unification to the assumption lists, thereby updating the assumptions made so far with the information gained by the unification. The assumptions for application is the two substitutions applied to the union of the two list of assumptions and the type is the two substitutions applied to α.

For conditional we first analyse $\mathtt{e}_1$ and the two branches. Next we unify the type, $\mathtt{t}_1$, with the type `Bool`. The two branches must have the same type, so we unify $\mathtt{t}_2$ and $\mathtt{t}_3$. Again the we must unify the different assumptions for each variables in the three lists of assumptions. Each time we have gained more information, by unification, we apply the substitution before doing anything else. The assumptions for the conditional is the union of the three assumption lists and the type is the unified type of the branches.

For fixpoints we first analyse the body. Next we have to ensure that the type is a function type and that if has the form $\alpha \rightarrow \alpha$. We apply the substitution to the assumptions and the type variable, α, as the result.

For constants we do not construct any assumptions and the type is the predefined type for the constant.

The type inference algorithm, $\mathcal{T}$, is sound:

Theorem 4.4 *Soundness of $\mathcal{T}$*
If $\mathcal{T}(\mathtt{e}) = (\mathrm{A}, \mathtt{t})$ then for all ground substitutions S_1 we can infer

$$S_1\mathrm{A} \vdash \mathtt{e} : S_1\mathtt{t}$$

□

Proof By structural induction on `e`. For the details see [Dam85] ■

and complete:

Theorem 4.5 *Completeness of $\mathcal{T}$*
If $\mathrm{A} \vdash \mathtt{e} : \mathtt{t}$ then there exists a substitution, S, and a subset, A'', of A such that

$$\mathcal{T}(\mathtt{e}) = (\mathrm{A}', \mathtt{t}')$$

and

$$\mathtt{t} = S\mathtt{t}'$$

and

$$\mathrm{A}'' = S\mathrm{A}'$$

□

Proof By structural induction on the proof-tree for $\mathrm{A} \vdash \mathtt{e} : \mathtt{t}$. For the details see [Dam85]. ■

Example 4.6
Consider the term $\lambda\mathtt{f}.\lambda\mathtt{x}.\mathtt{f}\ (\mathtt{f}\ \mathtt{x})$. We calculate:

$$\begin{array}{ll} \mathcal{T}(\lambda\mathtt{f}.\lambda\mathtt{x}.\mathtt{f}\ (\mathtt{f}\ \mathtt{x})) & \\ \quad = & \text{let} \quad \alpha_1,\ \alpha_2,\ \alpha_3,\ \alpha_4,\ \alpha_5,\ \alpha_6,\ \alpha_7 \text{ be fresh type variables} \\ & \text{in} \quad ((\mathtt{f} : \alpha_3 \to \alpha_3,\ \mathtt{x} : \alpha_3,\ (\alpha_3 \to \alpha_3) \to \alpha_3 \to \alpha_3) \end{array}$$

hence we can for all types, $\mathtt{t}$, infer $\emptyset \vdash \lambda\mathtt{f}.\lambda\mathtt{x}.\mathtt{f}\ (\mathtt{f}\ \mathtt{x}) : (\mathtt{t} \to \mathtt{t}) \to \mathtt{t} \to \mathtt{t}$.
□

Another well known standard type inference algorithm is the algorithm $\mathcal{W}$ by Milner [Mil78]. The difference between the algorithm $\mathcal{W}$ and the algorithm $\mathcal{T}$ is that the algorithm $\mathcal{W}$ computes both a type and a substitution given a term and a list of assumptions, whereas the algorithm $\mathcal{T}$ computes the type and the list of assumption given a term. The substitution in the algorithm $\mathcal{W}$ is to correct/update the assumption made so far. In the algorithm $\mathcal{T}$ the assumptions themselves are corrected/updated, hence no need for returning the substitution. The algorithm $\mathcal{W}$, presented in Figure 4.3, is also sound and complete [Mil78, Dam85].

The above algorithms does not deal with sub-typing. For algorithms that do take sub-typing into account see Mitchell [Mit91] and Fuh and Mishra [FM88, FM89, FM90].

4.2 Strictness and Totality Inference Algorithm

One way to construct an type inference algorithm for strictness and totality analysis — for both the analysis in Chapter 2 and the analysis in Chapter 3

$$
\begin{array}{lcll}
\mathcal{W}(\mathrm{A},\, \mathtt{x}) & = & ([\,],\, \mathrm{A}(\mathtt{x})) & \\
\mathcal{W}(\mathrm{A},\, \lambda\mathtt{x.e}) & = & \text{let} & \alpha \text{ be a fresh type variable} \\
 & & & (S,\, \mathtt{t}) = \mathcal{W}(\mathrm{A},\, \mathtt{x} : \alpha,\, \mathtt{e}) \\
 & & \text{in} & (S,\, S\alpha \to \mathtt{t}) \\
\mathcal{W}(\mathrm{A},\, \mathtt{e}_1\ \mathtt{e}_2) & = & \text{let} & (S_1,\, \mathtt{t}_1) = \mathcal{W}(\mathrm{A},\, \mathtt{e}_1) \\
 & & & (S_2,\, \mathtt{t}_2) = \mathcal{W}(S_1\mathrm{A},\, \mathtt{e}_2) \\
 & & & \alpha \text{ be a fresh type variable} \\
 & & & S_3 = \mathcal{U}(S_2\mathtt{t}_1,\, \mathtt{t}_2 \to \alpha) \\
 & & \text{in} & (S_3 \circ S_2 \circ S_1, S_3\alpha) \\
\multicolumn{4}{l}{\mathcal{W}(\mathrm{A},\, \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3)} \\
 & = & \text{let} & (S_1,\, \mathtt{t}_1) = \mathcal{W}(\mathrm{A},\, \mathtt{e}_1) \\
 & & & (S_2,\, \mathtt{t}_2) = \mathcal{W}(S_1\mathrm{A},\, \mathtt{e}_2) \\
 & & & (S_3,\, \mathtt{t}_3) = \mathcal{W}((S_2 \circ S_1)\mathrm{A},\, \mathtt{e}_3) \\
 & & & S_4 = \mathcal{U}((S_3 \circ S_2)\mathtt{t}_1,\, \mathtt{Bool}) \\
 & & & S_5 = \mathcal{U}((S_4 \circ S_3)\mathtt{t}_2,\, S_4\mathtt{t}_3) \\
 & & \text{in} & (S_5 \circ S_4 \circ S_3 \circ S_2 \circ S_1, (S_5 \circ S_4)\mathtt{t}_3) \\
\mathcal{W}(\mathrm{A},\, \mathtt{fix}\ \mathtt{e}) & = & \text{let} & \alpha \text{ be a fresh type variable} \\
 & & & (S_1,\, \mathtt{t}) = \mathcal{W}(\mathrm{A},\, \mathtt{e}) \\
 & & & S_2 = \mathcal{U}(\mathtt{t},\, \alpha \to \alpha) \\
 & & \text{in} & (S_2 \circ S_1, S_2\alpha) \\
\mathcal{W}(\mathrm{A},\, \mathtt{c}) & = & ([\,],\, \mathtt{t}_{\mathtt{c}}) &
\end{array}
$$

Figure 4.3: The Algorithm $\mathcal{W}$

is:

$$
\begin{array}{lcll}
\mathcal{T}'(\mathtt{e}) & = & \text{let} & (\mathrm{A}, \mathtt{t}) = \mathcal{T}(\mathtt{e}) \\
 & & & S \text{ is a ground substitution} \\
 & & \text{in} & S\mathtt{t}^{\top}
\end{array}
$$

where the algorithm $\mathcal{T}$ computes the standard type of $\mathtt{e}$. A strictness and totality type for $\mathtt{e}$ is now $\mathtt{t}^{\top}$. However this type does not give more strictness and totality information than the standard type. A second way to construct an inference algorithm for the analyses in Chapter 2 and 3 is to extend the standard inference algorithm to compute the *most general type* by following the *most general type* approach by Hankin and Le Métayer [HM94a]. For a given term, the algorithm will find *all* the strictness and totality types that can be inferred for the term. Often we are only interested in knowing if a term possesses one particular strictness and totality type and not all of them, so this approach seems like using a sledge hammer to crack a nut. We follow the *lazy type* approach of Hankin and Le Métayer [HM94a] where only the information necessary to answer *one* question is calculated.

The algorithm is constructed as follows:

- Make the inference system structural in both term and strictness and totality type. This is achieved by integrating the rule [coer] into all the appropriate rules and axioms.
- Introduce the lazy strictness and totality types.
- Extract an algorithm from the lazy strictness and totality type inference system.

4.2.1 The Structural Strictness and Totality Inference System

The coercion-rule:

$$\frac{\mathrm{A} \vdash_{\mathrm{S}} \mathtt{e} : \mathtt{t}_1}{\mathrm{A} \vdash_{\mathrm{S}} \mathtt{e} : \mathtt{t}_2} \quad \text{if } \mathtt{t}_1 \ \leq_{\mathrm{D}} \ \mathtt{t}_2$$

can be applied anywhere in the proof-tree. Our goal is to construct an inference system without the coercion rule, whereby we can construct proof-trees without being concerned with the when to apply the coercion rule. The new inference system is presented in Figure 4.4. The name of the inference system, "Structural Strictness and Totality Type Inference" refers to that the rules are structural in the term and type.

The new rules are constructed by applying the rule [coer] after each of the old rules:

$$\frac{\dfrac{}{\mathrm{A} \vdash \mathtt{e} : \mathtt{t}_1}\,[\text{old rule}] \qquad \mathtt{t}_1 \ \leq_{\mathrm{D}} \ \mathtt{t}_2}{\mathrm{A} \vdash \mathtt{e} : \mathtt{t}_2}\,[\text{coer}]$$

The new rule for variables is:

$$\frac{}{\mathrm{A} \vdash_{\mathrm{S}} \mathtt{x} : \mathtt{t}_2} \quad \text{if } \mathtt{x} : \mathtt{t}_1 \in \mathrm{A},\ \mathtt{t}_1 \ \leq_{\mathrm{D}} \ \mathtt{t}_2$$

$$[\mathrm{var_S}]\ \frac{}{\mathrm{A} \vdash_{\mathrm{S}} \mathtt{x} : \mathtt{t}_2} \quad \text{if } \mathtt{x} : \mathtt{t}_1 \in \mathrm{A} \wedge \mathtt{t}_1 \leq_{\mathrm{D}} \mathtt{t}_2$$

$$[\mathrm{abs1_S}]\ \frac{\mathrm{A}, \mathtt{x} : \mathtt{t}_1 \vdash_{\mathrm{S}} \mathtt{e} : \mathtt{t}_2}{\mathrm{A} \vdash_{\mathrm{S}} \lambda\mathtt{x.e} : \mathtt{t}_1 \rightarrow \mathtt{t}_2}$$

$$[\mathrm{abs2_S}]\ \frac{\varepsilon(\mathrm{A}) \vdash \lambda\mathtt{x.e} : \mathtt{ut}_1 \rightarrow \mathtt{ut}_2}{\mathrm{A} \vdash_{\mathrm{S}} \lambda\mathtt{x.e} : (\mathtt{ut}_1 \rightarrow \mathtt{ut}_2)^{\mathbf{n}}}$$

$$[\mathrm{abs3_S}]\ \frac{\varepsilon(\mathrm{A}) \vdash \lambda\mathtt{x.e} : \mathtt{ut}_1 \rightarrow \mathtt{ut}_2}{\mathrm{A} \vdash_{\mathrm{S}} \lambda\mathtt{x.e} : (\mathtt{ut}_1 \rightarrow \mathtt{ut}_2)^{\top}}$$

$$[\mathrm{abs4_S}]\ \frac{\mathrm{A}, \mathtt{x} : \mathtt{t}_1 \vdash_{\mathrm{S}} \mathtt{e} : \mathtt{t}_2}{\mathrm{A} \vdash_{\mathrm{S}} \lambda\mathtt{x.e} : \downarrow\mathtt{t}_1 \rightarrow \downarrow\mathtt{t}_2}$$

$$[\mathrm{app_S}]\ \frac{\mathrm{A} \vdash_{\mathrm{S}} \mathtt{e}_1 : \mathtt{t}_1 \rightarrow \mathtt{t}_2 \quad \mathrm{A} \vdash_{\mathrm{S}} \mathtt{e}_2 : \mathtt{t}_1}{\mathrm{A} \vdash_{\mathrm{S}} \mathtt{e}_1\ \mathtt{e}_2 : \mathtt{t}_2}$$

$$[\mathrm{if1_S}] \frac{\mathrm{A} \vdash_{\mathrm{S}} \mathtt{e}_1 : \mathtt{Bool}^{\mathbf{b}} \quad \mathrm{A} \vdash_{\mathrm{S}} \mathtt{e}_2 : \mathtt{t}' \quad \mathrm{A} \vdash_{\mathrm{S}} \mathtt{e}_3 : \mathtt{t}' \quad \varepsilon(\mathtt{t}')^{\mathbf{b}} \leq_{\mathrm{D}} \mathtt{t}}{\mathrm{A} \vdash_{\mathrm{S}} \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 : \mathtt{t}}$$

$$[\mathrm{if2_S}]\ \frac{\mathrm{A} \vdash_{\mathrm{S}} \mathtt{e}_1 : \mathtt{Bool}^{\mathbf{n}} \quad \mathrm{A} \vdash_{\mathrm{S}} \mathtt{e}_2 : \mathtt{t} \quad \mathrm{A} \vdash_{\mathrm{S}} \mathtt{e}_3 : \mathtt{t}}{\mathrm{A} \vdash_{\mathrm{S}} \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 : \mathtt{t}}$$

$$[\mathrm{if3_S}]\ \frac{\mathrm{A} \vdash_{\mathrm{S}} \mathtt{e}_1 : \mathtt{Bool}^{\top} \quad \mathrm{A} \vdash_{\mathrm{S}} \mathtt{e}_2 : \mathtt{t} \quad \mathrm{A} \vdash_{\mathrm{S}} \mathtt{e}_3 : \mathtt{t}}{\mathrm{A} \vdash_{\mathrm{S}} \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 : \mathtt{t}} \quad \text{if } \mathrm{BOT_{ST}}(\mathtt{t})$$

$$[\mathrm{fix_S}]\ \frac{\mathrm{A} \vdash_{\mathrm{S}} \mathtt{e} : (\mathtt{t}_1 \rightarrow \mathtt{t}_2) \wedge (\mathtt{t}_2 \rightarrow \mathtt{t}_3) \wedge \cdots \wedge (\mathtt{t}_{n-1} \rightarrow \mathtt{t}_n)}{\mathrm{A} \vdash_{\mathrm{S}} \mathtt{fix}\ \mathtt{e} : \mathtt{t}_n}$$
$$\text{if} \begin{cases} \mathrm{BOT_{ST}}(\mathtt{t}_1), \\ \exists p, q : p < q \wedge \mathtt{t}_q \leq_{\mathrm{D}} \mathtt{t}_p \end{cases}$$

$$[\mathrm{const_S}]\ \frac{\mathtt{t_c} \leq_{\mathrm{D}} \mathtt{t}}{\mathrm{A} \vdash_{\mathrm{S}} \mathtt{c} : \mathtt{t}}$$

$$[\mathrm{conj_S}]\ \frac{\mathrm{A} \vdash_{\mathrm{S}} \mathtt{e} : \mathtt{t}_1 \quad \mathrm{A} \vdash_{\mathrm{S}} \mathtt{e} : \mathtt{t}_2}{\mathrm{A} \vdash_{\mathrm{S}} \mathtt{e} : \mathtt{t}_1 \wedge \mathtt{t}_2}$$

$$[\mathrm{down_S}]\ \frac{\mathrm{A} \vdash_{\mathrm{S}} \mathtt{e} : \mathtt{down}'(\mathtt{t})}{\mathrm{A} \vdash_{\mathrm{S}} \mathtt{e} : \downarrow\mathtt{t}}$$

Figure 4.4: Structural Strictness and Totality Type Inference

The new rules for constants, and the first rule for conditional, is constructed in the same way. The coercion rule is not needed after the rules [if2], [if3], [app], and [fix] since we can “push” the use of the coercion rule upwards in the proof-tree. To see how, consider the proof-tree

$$\dfrac{\dfrac{A \vdash \mathtt{e}_1 : \mathtt{Bool}^{\mathbf{n}} \quad A \vdash \mathtt{e}_2 : \mathtt{t}_1 \quad A \vdash \mathtt{e}_3 : \mathtt{t}_1}{A \vdash \mathtt{if\ e1\ then\ e}_2\ \mathtt{else\ e}_3 : \mathtt{t}_1}\,[\mathrm{if2}] \qquad \mathtt{t}_1 \leq_{\mathrm{D}} \mathtt{t}_2}{A \vdash \mathtt{if\ e1\ then\ e}_2\ \mathtt{else\ e}_3 : \mathtt{t}_2}\,[\mathrm{coer}]$$

which can be transformed into

$$\dfrac{A \vdash \mathtt{e}_1 : \mathtt{Bool}^{\mathbf{n}} \quad \dfrac{A \vdash \mathtt{e}_2 : \mathtt{t}_1 \quad \mathtt{t}_1 \leq_{\mathrm{D}} \mathtt{t}_2}{A \vdash \mathtt{e}_2 : \mathtt{t}_2} \quad \dfrac{A \vdash \mathtt{e}_3 : \mathtt{t}_1 \quad \mathtt{t}_1 \leq_{\mathrm{D}} \mathtt{t}_2}{A \vdash \mathtt{e}_3 : \mathtt{t}_2}}{A \vdash \mathtt{if\ e1\ then\ e}_2\ \mathtt{else\ e}_3 : \mathtt{t}_2}\,[\mathrm{if2}]$$

where the use of the rule [coer] is moved upwards in the proof-tree.

For abstraction we will at least have the two rules corresponding to the rules [abs1] and [abs2]. Note that we have changed the presentation of the rule [abs2] slightly. The reason is that now we are interesting in doing as little as possible to check that a term has a given type assuming that it is “cheaper” to construct a standard type proof-tree than a strictness and totality type proof-tree.

The coercion rule is need after the rules [abs1], [abs2], and [conj]; e.g. we may construct the following proof-tree:

$$\dfrac{\dfrac{A \vdash \lambda \mathtt{x.e} : \mathtt{ut}_1 \rightarrow \mathtt{ut}_2}{A \vdash \lambda \mathtt{x.e} : (\mathtt{ut}_1 \rightarrow \mathtt{ut}_2)^{\mathbf{n}}}\,[\mathrm{abs2}]}{A \vdash \lambda \mathtt{x.e} : (\mathtt{ut}_1 \rightarrow \mathtt{ut}_2)^{\top}}\,[\mathrm{coer}]$$

This motivates the new rule:

$$[\mathrm{abs3}_{\mathrm{S}}]\ \dfrac{\varepsilon(A) \vdash \lambda \mathtt{x.e} : \mathtt{ut}_1 \rightarrow \mathtt{ut}_2}{A \vdash_{\mathrm{S}} \lambda \mathtt{x.e} : (\mathtt{ut}_1 \rightarrow \mathtt{ut}_2)^{\top}}$$

thereby allowing us not to think about using the coercion rule. We can also construct the proof-tree:

$$\dfrac{\dfrac{A, \mathtt{x} : \mathtt{t}_1 \vdash \mathtt{e} : \mathtt{t}_2}{A \vdash \lambda \mathtt{x.e} : \mathtt{t}_1 \rightarrow \mathtt{t}_2}\,[\mathrm{abs1}]}{A \vdash \lambda \mathtt{x.e} : \downarrow\mathtt{t}_1 \rightarrow \downarrow\mathtt{t}_2}\,[\mathrm{coer}]$$

This motivates the new rule:

$$[\mathrm{abs4_S}]\ \frac{\mathrm{A},\ \mathtt{x} : \mathtt{t}_1 \vdash_{\mathrm{S}} \mathtt{e} : \mathtt{t}_2}{\mathrm{A} \vdash_{\mathrm{S}} \lambda\mathtt{x}.\mathtt{e} : \downarrow\mathtt{t}_1 \rightarrow \downarrow\mathtt{t}_2}$$

We can construct the following rule by applying the coercion rule to the result of applying either [abs1] or [abs2]:

$$\frac{\mathrm{A} \vdash_{\mathrm{S}} \lambda\mathtt{x}.\mathtt{e} : \mathtt{t}}{\mathrm{A} \vdash_{\mathrm{S}} \lambda\mathtt{x}.\mathtt{e} : \downarrow\mathtt{t}}$$

This rule can be stated more generally as

$$[\mathrm{down_S}]\ \frac{\mathrm{A} \vdash_{\mathrm{S}} \mathtt{e} : \mathtt{down}'(\mathtt{t})}{\mathrm{A} \vdash_{\mathrm{S}} \mathtt{e} : \downarrow\mathtt{t}}$$

where the function $\mathtt{down}'$ moves the $\downarrow$-construct inwards one level using the coercion rules such that

$$\mathtt{t} \leq_{\mathrm{D}} \mathtt{down}'(\mathtt{t}) \leq_{\mathrm{D}} \downarrow\mathtt{t}$$

We also define the function $\mathtt{down}$ which also moves the $\downarrow$-construct inwards one level using the coercion rules, however this time we want

$$\downarrow\mathtt{t} \leq_{\mathrm{D}} \mathtt{down}(\mathtt{t})$$

The functions are defined by:

$$\begin{aligned}
\mathtt{down}'(\mathtt{t}_1 \wedge \mathtt{t}_2) &= \mathtt{t}_1 \wedge \mathtt{t}_2 \\
\mathtt{down}'(\mathtt{t}) &= \mathtt{down}(\mathtt{t})
\end{aligned}$$

$$\begin{aligned}
\mathtt{down}(\mathtt{ut}^{\mathbf{n}}) &= \mathtt{ut}^{\top} \\
\mathtt{down}(\mathtt{ut}^{\mathbf{b}}) &= \mathtt{ut}^{\mathbf{b}} \\
\mathtt{down}(\mathtt{ut}^{\top}) &= \mathtt{ut}^{\top} \\
\mathtt{down}(\mathtt{t}_1 \rightarrow \mathtt{t}_2) &= \mathtt{t}_1 \rightarrow \downarrow\mathtt{t}_2 \\
\mathtt{down}(\mathtt{t}_1 \wedge \mathtt{t}_2) &= \downarrow\mathtt{t}_1 \wedge \downarrow\mathtt{t}_2 \\
\mathtt{down}(\downarrow\mathtt{t}) &= \downarrow\mathtt{t}
\end{aligned}$$

Note that, whenever $\mathtt{t}$ is not a conjunction type, we have

$$\downarrow\mathtt{t} \equiv_{\mathrm{D}} \mathtt{down}'(\mathtt{t}) \equiv_{\mathrm{D}} \mathtt{down}(\mathtt{t})$$

Fact 4.7
$\mathtt{t} \le_{\mathrm{D}} \mathtt{down}'(\mathtt{t}) \le_{\mathrm{D}} \downarrow\mathtt{t} \le_{\mathrm{D}} \mathtt{down}(\mathtt{t})$ □

Proof We will show

$$\mathtt{t} \le_{\mathrm{D}} \mathtt{down}'(\mathtt{t}) \le_{\mathrm{D}} \downarrow\mathtt{t} \le_{\mathrm{D}} \mathtt{down}(\mathtt{t})$$

by induction on the strictness and totality type $\mathtt{t}$.

For the full proof see Appendix page 323. ■

The question is now "How many rules are we going to have for abstraction?" We can verify that no more rules are needed by induction of the type of the abstraction:

$\mathtt{ut}^{\mathbf{n}}$: Whenever the abstraction has the type $\mathtt{ut}^{\mathbf{n}}$ we can use the rule [abs2_{S}].

$\mathtt{ut}^{\mathbf{b}}$: This is not a possible type for an abstraction since it always has a WHNF.

$\mathtt{ut}^{\top}$: Whenever the abstraction has the type $\mathtt{ut}^{\top}$ we can use the rule [abs3_{S}].

$\mathtt{t}_1 \to \mathtt{t}_2$: We use either the rule [abs$_{\mathrm{S}}$1] or the rule [abs4_{S}].

$\mathtt{t}_1 \wedge \mathtt{t}_2$: It must be the case that we can infer both the type $\mathtt{t}_1$ and the type $\mathtt{t}_2$ for the abstraction, hence we can use the rule [conj]$_{\mathrm{S}}$.

$\downarrow\mathtt{t}$: We can use the rule [down$_{\mathrm{S}}$].

For conjunction we have the old [conj]-rule and the following new rule:

$$\frac{A \vdash_{\mathrm{S}} \mathtt{e} : \mathtt{t}_1 \wedge \mathtt{t}_2}{A \vdash_{\mathrm{S}} \mathtt{e} : \downarrow(\mathtt{t}_1 \wedge \mathtt{t}_2)}$$

However we can use the general rule [down$_{\mathrm{S}}$] instead of the above rule. We do not need to generate more rules for conjunction since the rest of the cases can be push up the proof-tree.

The new structural strictness and totality type inference system defined by Figure 4.4 is sound with respect to the strictness and totality inference system:

Lemma 4.8
$A \vdash_{\mathrm{S}} \mathtt{e} : \mathtt{t} \Rightarrow A \vdash \mathtt{e} : \mathtt{t}$ □

Proof We show that all rules in the structural inference system can be derived in the non-structural inference system.

For the proof see Appendix page 325. ■

The new structural inference system is *not* complete with respect to the strictness and totality inference system in Figure 3.2. Consider the term $\lambda\mathtt{x}.\bot_{\mathtt{Int}}$ and the strictness and totality type

$$\mathtt{t} = \downarrow((\mathtt{Int}^{\mathbf{n}} \to \mathtt{Int}^{\mathbf{n}}) \wedge (\mathtt{Int}^{\mathbf{n}} \to \mathtt{Int}^{\mathbf{n}}))$$

Using the non-structural inference system we can infer that $\lambda\mathtt{x}.\bot_{\mathtt{Int}}$ has the type $\mathtt{t}$:

$$\dfrac{\dfrac{\mathtt{x} : \mathtt{Int}^{\mathbf{n}} \vdash \bot_{\mathtt{Int}} : \mathtt{Int}^{\top}}{\emptyset \vdash \lambda\mathtt{x}.\bot_{\mathtt{Int}} : \mathtt{Int}^{\mathbf{n}} \to \mathtt{Int}^{\top}}\,[\text{abs}] \qquad \mathtt{Int}^{\mathbf{n}} \to \mathtt{Int}^{\top} \leq_{\text{ST}} \mathtt{t}}{\emptyset \vdash \lambda\mathtt{x}.\bot_{\mathtt{Int}} : \mathtt{t}}\,[\text{coer}]$$

In the structural inference system it is not possible to construct a proof-tree for $\emptyset \vdash_{\text{S}} \lambda\mathtt{x}.\bot_{\mathtt{Int}} : \mathtt{t}$. To see this observe that the last rule used to construct a proof-tree for $\emptyset \vdash_{\text{S}} \lambda\mathtt{x}.\bot_{\mathtt{Int}} : \mathtt{t}$ cannot have been one of the abstractions rules — it could only have been the rule [down_{S}]. Now we have to construct a proof-tree for

$$\emptyset \vdash_{\text{S}} \lambda\mathtt{x}.\bot_{\mathtt{Int}} : (\mathtt{Int}^{\mathbf{n}} \to \mathtt{Int}^{\mathbf{n}}) \wedge (\mathtt{Int}^{\mathbf{n}} \to \mathtt{Int}^{\mathbf{n}})$$

The last rule used here must have been the rule [conj_{S}] and we are left with the construction of a proof-tree for

$$\emptyset \vdash_{\text{S}} \lambda\mathtt{x}.\bot_{\mathtt{Int}} : \mathtt{Int}^{\mathbf{n}} \to \mathtt{Int}^{\mathbf{n}}$$

The last rule must have been the rule [abs1_{S}]:

$$\mathtt{x} : \mathtt{Int}^{\mathbf{n}} \vdash_{\text{S}} \bot_{\mathtt{Int}} : \mathtt{Int}^{\mathbf{n}}$$

which cannot be the case. Hence we cannot construct a proof-tree for $\emptyset \vdash_{\text{S}} \lambda\mathtt{x}.\bot_{\mathtt{Int}} : \mathtt{t}$.

4.2.2 Lazy Strictness and Totality Type Inference System

Following the lazy type approach by Hankin and Le Métayer [HM94a] we are now ready to introduce the *lazy strictness and totality types*: we will allow terms and assumption list pairs to be part of the types. In this way we can delay the construction of a part of the proof-tree.

The lazy strictness and totality types are now:

$$\begin{array}{rcl}
\mathtt{t} & ::= & \mathtt{ut}^s \mid \mathtt{t} \rightarrow \mathtt{t} \mid \mathtt{t} \wedge \mathtt{t} \mid \downarrow\!\mathtt{t} \mid (\mathrm{A}, \mathtt{e}) \\
\mathtt{ut} & ::= & \mathtt{B} \mid \mathtt{ut} \rightarrow \mathtt{ut} \\
s & ::= & \top \mid \mathbf{n} \mid \mathbf{b}
\end{array}$$

The new type constructor, $(\mathrm{A}, \mathtt{e})$, is a shorthand for the conjunction of all the strictness and totality types, that can be inferred for $\mathtt{e}$ using the assumption list A. It can be thought of as the delayed construction of the proof-tree for $\mathtt{e}$. The function **`expand`** maps lazy strictness and totality types to strictness and totality types:

$$\begin{array}{rcl}
\mathtt{expand}(\mathtt{ut}^{\mathbf{n}}) & = & \mathtt{ut}^{\mathbf{n}} \\
\mathtt{expand}(\mathtt{ut}^{\mathbf{b}}) & = & \mathtt{ut}^{\mathbf{b}} \\
\mathtt{expand}(\mathtt{ut}^{\top}) & = & \mathtt{ut}^{\top} \\
\mathtt{expand}(\downarrow\!\mathtt{t}) & = & \downarrow\!\mathtt{expand}(\mathtt{t}) \\
\mathtt{expand}(\mathtt{t}_1 \rightarrow \mathtt{t}_2) & = & \mathtt{expand}(\mathtt{t}_1) \rightarrow \mathtt{expand}(\mathtt{t}_2) \\
\mathtt{expand}(\mathtt{t}_1 \wedge \mathtt{t}_2) & = & \mathtt{expand}(\mathtt{t}_1) \wedge \mathtt{expand}(\mathtt{t}_2) \\
\mathtt{expand}((\mathrm{A}, \mathtt{e})) & = & \bigwedge\{\mathtt{t} \mid \mathtt{expand}(\mathrm{A}) \vdash_{\mathrm{L}} \mathtt{e} : \mathtt{t}\}
\end{array}$$

and the function is extended to environments in a component-wise manner. We now want to construct an inference system that makes use of the lazy types. The new inference system is presented in Figure 4.5.

The well-formedness predicate on lazy strictness and totality types is an extension of the well-formedness predicate on strictness and totality types (see (3.1), (3.2), (3.3) and (3.4)) in that the lazy types are always well-formed:

$$\overline{\vdash^{W} (\mathrm{A}, \mathtt{e})}$$

$$[\mathrm{var_S}]\ \frac{}{\mathrm{A} \vdash_{\mathrm{L}} \mathtt{x} : \mathtt{t}_2} \quad \text{if } \mathtt{x} : \mathtt{t}_1 \in \mathrm{A} \wedge \mathtt{t}_1 \leq_{\mathrm{D}} \mathtt{t}_2$$

$$[\mathrm{abs1_S}]\ \frac{\mathrm{A}, \mathtt{x} : \mathtt{t}_1 \vdash_{\mathrm{L}} \mathtt{e} : \mathtt{t}_2}{\mathrm{A} \vdash_{\mathrm{L}} \lambda\mathtt{x.e} : \mathtt{t}_1 \to \mathtt{t}_2}$$

$$[\mathrm{abs2_S}]\ \frac{\varepsilon(\mathrm{A}) \vdash \lambda\mathtt{x.e} : \mathtt{ut}_1 \to \mathtt{ut}_2}{\mathrm{A} \vdash_{\mathrm{L}} \lambda\mathtt{x.e} : (\mathtt{ut}_1 \to \mathtt{ut}_2)^{\mathbf{n}}}$$

$$[\mathrm{abs3_S}]\ \frac{\varepsilon(\mathrm{A}) \vdash \lambda\mathtt{x.e} : \mathtt{ut}_1 \to \mathtt{ut}_2}{\mathrm{A} \vdash_{\mathrm{L}} \lambda\mathtt{x.e} : (\mathtt{ut}_1 \to \mathtt{ut}_2)^{\top}}$$

$$[\mathrm{abs4_S}]\ \frac{\mathrm{A}, \mathtt{x} : \mathtt{t}_1 \vdash_{\mathrm{L}} \mathtt{e} : \mathtt{t}_2}{\mathrm{A} \vdash_{\mathrm{L}} \lambda\mathtt{x.e} : \downarrow\mathtt{t}_1 \to \downarrow\mathtt{t}_2}$$

$$[\mathrm{app_L}]\ \frac{\mathrm{A} \vdash_{\mathrm{L}} \mathtt{e}_1 : (\mathrm{A}, \mathtt{e}_2) \to \mathtt{t}}{\mathrm{A} \vdash_{\mathrm{L}} \mathtt{e}_1\ \mathtt{e}_2 : \mathtt{t}}$$

$$[\mathrm{if1_L}]\ \frac{\mathrm{A} \vdash_{\mathrm{L}} \mathtt{e}_1 : \mathtt{Bool}^{\mathbf{b}}}{\mathrm{A} \vdash_{\mathrm{L}} \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 : \mathtt{t}} \quad \text{if} \begin{cases} \varepsilon(\mathrm{A}) \vdash \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 : \mathtt{ut} \\ \wedge\ \mathtt{ut}^{\mathbf{b}} \leq_{\mathrm{D}} \mathtt{t} \end{cases}$$

$$[\mathrm{if2_S}]\ \frac{\mathrm{A} \vdash_{\mathrm{L}} \mathtt{e}_1 : \mathtt{Bool}^{\mathbf{n}} \quad \mathrm{A} \vdash_{\mathrm{L}} \mathtt{e}_2 : \mathtt{t} \quad \mathrm{A} \vdash_{\mathrm{L}} \mathtt{e}_3 : \mathtt{t}}{\mathrm{A} \vdash_{\mathrm{L}} \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 : \mathtt{t}}$$

$$[\mathrm{if3_S}]\ \frac{\mathrm{A} \vdash_{\mathrm{L}} \mathtt{e}_1 : \mathtt{Bool}^{\top} \quad \mathrm{A} \vdash_{\mathrm{L}} \mathtt{e}_2 : \mathtt{t} \quad \mathrm{A} \vdash_{\mathrm{L}} \mathtt{e}_3 : \mathtt{t}}{\mathrm{A} \vdash_{\mathrm{L}} \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 : \mathtt{t}} \quad \text{if } \mathrm{BOT_{ST}}(\mathtt{t})$$

$$[\mathrm{fix_S}]\ \frac{\mathrm{A} \vdash_{\mathrm{L}} \mathtt{e} : (\mathtt{t}_1 \to \mathtt{t}_2) \wedge (\mathtt{t}_2 \to \mathtt{t}_3) \wedge \cdots \wedge (\mathtt{t}_{n-1} \to \mathtt{t}_n)}{\mathrm{A} \vdash_{\mathrm{L}} \mathtt{fix}\ \mathtt{e} : \mathtt{t}_n} \quad \text{if} \begin{cases} \mathrm{BOT_{ST}}(\mathtt{t}_1), \\ \exists p, q : p < q \wedge \mathtt{t}_q \leq_{\mathrm{D}} \mathtt{t}_p \end{cases}$$

$$[\mathrm{const_S}]\ \frac{\mathtt{t}_{\mathtt{c}} \leq_{\mathrm{D}} \mathtt{t}}{\mathrm{A} \vdash_{\mathrm{L}} \mathtt{c} : \mathtt{t}}$$

$$[\mathrm{conj_S}]\ \frac{\mathrm{A} \vdash_{\mathrm{L}} \mathtt{e} : \mathtt{t}_1 \quad \mathrm{A} \vdash_{\mathrm{L}} \mathtt{e} : \mathtt{t}_2}{\mathrm{A} \vdash_{\mathrm{L}} \mathtt{e} : \mathtt{t}_1 \wedge \mathtt{t}_2} \qquad [\mathrm{down_S}]\ \frac{\mathrm{A} \vdash_{\mathrm{L}} \mathtt{e} : \mathtt{down}'(\mathtt{t})}{\mathrm{A} \vdash_{\mathrm{L}} \mathtt{e} : \downarrow\mathtt{t}}$$

Figure 4.5: Lazy Strictness and Totality Type Inference

The predicate $\mathrm{BOT_{ST}}$ is also extended to lazy strictness and totality types:

$$\mathrm{BOT_{ST}}((\mathrm{A}, \mathtt{e})) = \mathrm{BOT_{ST}}(\mathtt{expand}((\mathrm{A}, \mathtt{e})))$$

hence we have

Fact 4.9
For all lazy strictness and totality types $\mathtt{t}$ we have

$$\mathrm{BOT_{ST}}(\mathtt{t}) \Leftrightarrow \mathrm{BOT_{ST}}(\mathtt{expand}(\mathtt{t}))$$

□

The lazy strictness and totality types are useful in the application rule:

$$[\mathrm{app_L}]\ \frac{\mathrm{A} \vdash_{\mathrm{L}} \mathtt{e}_1 : (\mathrm{A}, \mathtt{e}_2) \rightarrow \mathtt{t}}{\mathrm{A} \vdash_{\mathrm{L}} \mathtt{e}_1\ \mathtt{e}_2 : \mathtt{t}}$$

where the construction of the proof-tree for $\mathtt{e}_2$ is delayed until it is necessary to construct.

In the [if1]-rule we no longer construct proof-trees for $\mathtt{e}_2$ and $\mathtt{e}_3$:

$$[\mathrm{if1_L}]\ \frac{\mathrm{A} \vdash_{\mathrm{L}} \mathtt{e}_1 : \mathtt{Bool}^{\mathbf{b}}}{\mathrm{A} \vdash_{\mathrm{L}} \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 : \mathtt{t}}$$
$$\text{if } \varepsilon(\mathrm{A}) \vdash \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 : \mathtt{ut} \wedge \mathtt{ut}^{\mathbf{b}} \leq_{\mathrm{D}} \mathtt{t}$$

The reason for this is, that this rule is derivable from the rule $[\mathrm{if1_S}]$: we assume

$$\mathrm{A} \vdash_{\mathrm{S}} \mathtt{e}_1 : \mathtt{Bool}^{\mathbf{b}}$$
$$\varepsilon(\mathrm{A}) \vdash \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 : \mathtt{ut}$$
$$\mathtt{ut}^{\mathbf{b}} \leq_{\mathrm{D}} \mathtt{t}$$

From the standard inference system we have

$$\varepsilon(\mathrm{A}) \vdash \mathtt{e}_2 : \mathtt{ut}$$
$$\varepsilon(\mathrm{A}) \vdash \mathtt{e}_3 : \mathtt{ut}$$

Therefore we also have

$$\mathrm{A} \vdash_{\mathrm{S}} \mathtt{e}_2 : \mathtt{ut}^{\top}$$
$$\mathrm{A} \vdash_{\mathrm{S}} \mathtt{e}_3 : \mathtt{ut}^{\top}$$

Now we can apply the rule [$\mathrm{if1_S}$] to get

$$\mathrm{A} \vdash_{\mathrm{S}} \texttt{if } \mathtt{e}_1 \texttt{ then } \mathtt{e}_2 \texttt{ else } \mathtt{e}_3 : \mathtt{t}$$

as required. Furthermore we have that the rule [$\mathrm{if1_S}$] is derivable from [$\mathrm{if1_L}$]: We assume

$$\begin{array}{l} \mathrm{A} \vdash_{\mathrm{L}} \mathtt{e}_1 : \mathtt{Bool}^{\mathbf{b}} \\ \mathrm{A} \vdash_{\mathrm{L}} \mathtt{e}_2 : \mathtt{t}' \\ \mathrm{A} \vdash_{\mathrm{L}} \mathtt{e}_3 : \mathtt{t}' \\ \mathtt{ut}^{\mathbf{b}} \leq_{\mathrm{D}} \mathtt{t} \end{array}$$

In the standard inference system we have

$$\begin{array}{l} \varepsilon(\mathrm{A}) \vdash \mathtt{e}_1 : \mathtt{Bool} \\ \varepsilon(\mathrm{A}) \vdash \mathtt{e}_2 : \varepsilon(\mathtt{t}') \\ \varepsilon(\mathrm{A}) \vdash \mathtt{e}_3 : \varepsilon(\mathtt{t}') \end{array}$$

and hence we have

$$\varepsilon(\mathrm{A}) \vdash \texttt{if } \mathtt{e}_1 \texttt{ then } \mathtt{e}_2 \texttt{ else } \mathtt{e}_3 : \varepsilon(\mathtt{t}')$$

and we can now apply the rule [$\mathrm{if1_L}$] to get

$$\mathrm{A} \vdash_{\mathrm{L}} \texttt{if } \mathtt{e}_1 \texttt{ then } \mathtt{e}_2 \texttt{ else } \mathtt{e}_3 : \mathtt{t}$$

as required. All the other rules remain the same.

We need to extend the subtyping relation to the lazy strictness and totality type. We have two rules for relating the new lazy strictness and totality types:

$$[\mathrm{env_L}]\ \frac{\mathrm{A} \vdash_{\mathrm{L}} \mathtt{e} : \mathtt{t}}{(\mathrm{A}, \mathtt{e}) \ \leq_{\mathrm{L}} \ \mathtt{t}} \quad \text{if } \mathtt{t} \neq (\mathrm{A}', \mathtt{e}')$$

$$[\mathrm{env_R}]\ \frac{\forall \mathtt{t}' : \mathrm{A} \vdash_{\mathrm{L}} \mathtt{e} : \mathtt{t}' \Rightarrow (\mathtt{t} \leq_{\mathrm{L}} \mathtt{t}')}{\mathtt{t} \ \leq_{\mathrm{L}} \ (\mathrm{A}, \mathtt{e})}$$

The first rule says that if we can infer the strictness and totality type, $\mathtt{t}$, for $\mathtt{e}$ using the assumptions, A, then the lazy type, $(\mathrm{A}, \mathtt{e})$ is less than $\mathtt{t}$; i.e. $\mathtt{t}$ is included in the conjunction of the strictness and totality types that $(\mathrm{A}, \mathtt{e})$ represents. Therefore the type $\mathtt{t}$ must be a strictness and totality type and not a lazy strictness and totality type. The second rule says that whenever all the types that can be infer for $\mathtt{e}$, using the assumptions, A, are greater than the type $\mathtt{t}$, then $\mathtt{t} \leq_{\mathrm{L}} (\mathrm{A}, \mathtt{e})$. Also here the type $\mathtt{t}'$ ranges over strictness and totality type and not over lazy strictness and totality types. The first rule is comparable with the rules [$\wedge$1] and [$\wedge$2], whereas the the second rule is comparable with the rule [$\wedge$3] in Figure 3.1.

The subtyping relation $\leq_{\mathrm{L}}$ is defined by the rules in Figure 3.1 and the two rules [$\mathrm{env}_{\mathrm{L}}$] and [$\mathrm{env}_{\mathrm{R}}$].

The lazy strictness and totality type inference system is sound with respect to the structural strictness and totality type inference system:

Lemma 4.10 *Soundness of* $\leq_{\mathrm{L}}$ *and* $\vdash_{\mathrm{L}}$
We have

$$
\begin{aligned}
&(\mathtt{t}_1 \leq_{\mathrm{L}} \mathtt{t}_2 \wedge \mathrm{A} \vdash_{\mathrm{L}} \mathtt{e} : \mathtt{t}) \\
&\quad \Rightarrow \quad (\mathtt{expand}(\mathtt{t}_1) \leq_{\mathrm{D}} \mathtt{expand}(\mathtt{t}_2) \wedge \mathtt{expand}(\mathrm{A}) \vdash_{\mathrm{S}} \mathtt{e} : \mathtt{expand}(\mathtt{t}))
\end{aligned}
$$

□

Proof We will assume $\mathtt{t}_1 \leq_{\mathrm{L}} \mathtt{t}_2$ and $\mathrm{A} \vdash_{\mathrm{L}} \mathtt{e} : \mathtt{t}$, we will then show

$$
\begin{aligned}
&\mathtt{expand}(\mathtt{t}_1) \leq_{\mathrm{D}} \mathtt{expand}(\mathtt{t}_2) \wedge \\
&\mathtt{expand}(\mathrm{A}) \vdash_{\mathrm{S}} \mathtt{e} : \mathtt{expand}(\mathtt{t})
\end{aligned}
$$

by simultaneous induction on the proof-tree for $\leq_{\mathrm{L}}$ and $\vdash_{\mathrm{L}}$.

The reason that we have to do simultaneous induction on $\leq_{\mathrm{L}}$ and $\vdash_{\mathrm{L}}$ is that $\leq_{\mathrm{L}}$ depends on $\vdash_{\mathrm{L}}$ and not only $\vdash_{\mathrm{L}}$ depending on $\leq_{\mathrm{L}}$ as in the other inference systems (e.g. in Chapter 2 and 3).

For the full proof see Appendix page 326. ■

For a discussion of completeness of the lazy strictness and totality type inference system with respect to the structural strictness and totality type inference system see Section 4.3.1.

4.2.3 The Lazy Strictness and Totality Type Inference Algorithm

The last step is to construct the lazy strictness and totality type checking algorithm $\mathcal{ST}$ from the lazy strictness and totality type inference system.

The algorithm $\mathcal{ST}$ takes as argument a list of assumptions, A, a term, $\mathtt{e}$, and a lazy strictness and totality type, $\mathtt{t}$, and the result is either $\mathtt{tt}$ or $\mathtt{ff}$. The idea is that whenever the algorithm returns $\mathtt{tt}$ then $\mathrm{A} \vdash_{\mathrm{L}} \mathtt{e} : \mathtt{t}$ can be inferred. The algorithm assumes that the term is well-formed with respect to the standard type system, i.e. that $\varepsilon(\mathrm{A}) \vdash \mathtt{e} : \varepsilon(\mathtt{t})$ can be inferred. The algorithm $\mathcal{ST}$ is presented in Figure 4.6.

For variables we lookup the type in the assumption list and use the algorithm $\mathcal{I}$ to test whether the type in the assumption list is less than the desired type, $\mathtt{t}$. The algorithm $\mathcal{I}$ takes as argument two lazy strictness and totality types, $\mathtt{t}_1$ and $\mathtt{t}_2$ and returns either $\mathtt{tt}$ or $\mathtt{ff}$. The idea is that the algorithm return $\mathtt{tt}$ whenever $\mathtt{t}_1 \leq_{\mathrm{L}} \mathtt{t}_2$ can be inferred. Note, the correspondence between the clause for variables in the definition of $\mathcal{ST}$ and the rule [$\mathrm{var_L}$] in Figure 4.5.

There are five clauses involving abstraction: The first three clauses correspond directly to the rules [$\mathrm{abs1_L}$], [$\mathrm{abs2_L}$], and [$\mathrm{abs3_L}$], respectively. In the first we extend the assumption list with an assumption about $\mathtt{x}$ and test the body. The two next clauses just return $\mathtt{tt}$, since there are no hypothesis in the rule [$\mathrm{abs2_L}$] and [$\mathrm{abs3_L}$] that has to be fulfilled. In order for the abstraction to have the type $\downarrow\mathtt{t}_1 \rightarrow \downarrow\mathtt{t}_2$ we could either use the rule [$\mathrm{abs1_L}$] or the rule [$\mathrm{abs4_L}$]. Hence we test both possibilities. The last clause for abstraction is for the type $(\mathtt{ut}_1 \rightarrow \mathtt{ut}_2)^{\mathbf{b}}$: an abstraction can never have this type so we return $\mathtt{ff}$.

The clause for application makes use of the lazy construct and is directly comparable with the rule [$\mathrm{app_L}$]. For conditional three possible rules could have been used: [$\mathrm{if1_L}$], [$\mathrm{if2_L}$], or [$\mathrm{if3_L}$]. In the case of the rule [$\mathrm{if1_L}$] we make a recursive call of $\mathcal{ST}$ to test whether $\mathtt{e}_1$ has the type $\mathtt{Bool}^{\mathbf{b}}$, furthermore the type, $\mathtt{t}$, for the whole term must be greater than $\varepsilon(\mathtt{t})^{\mathbf{b}}$; we call $\mathcal{I}$ with the arguments $\varepsilon(\mathtt{t})^{\mathbf{b}}$ and $\mathtt{t}$. Secondly, assuming that the applied rule is [$\mathrm{if2_L}$] we must ensure that both branches has the type $\mathtt{t}$ in order for the whole term to have the lazy strictness and totality type $\mathtt{t}$ and the test, $\mathtt{e}_1$, must have the type $\mathtt{Bool}^{\mathbf{n}}$. The last possibility is that the rule [$\mathrm{if3_L}$]

$$
\begin{array}{lcl}
\mathcal{ST}(\mathrm{A},\, \mathtt{x},\, \mathtt{t}) & = & \mathcal{I}(\mathrm{A}(\mathtt{x}),\, \mathtt{t}) \\
\mathcal{ST}(\mathrm{A},\, \lambda\mathtt{x.e},\, \mathtt{t}_1 \to \mathtt{t}_2) & = & \mathcal{ST}((\mathtt{x} : \mathtt{t}_1)\!: \mathrm{A},\, \mathtt{e},\, \mathtt{t}_2) \\
\mathcal{ST}(\mathrm{A},\, \lambda\mathtt{x.e},\, (\mathtt{ut}_1 \to \mathtt{ut}_2)^{\mathbf{n}}) & = & \mathtt{tt} \\
\mathcal{ST}(\mathrm{A},\, \lambda\mathtt{x.e},\, (\mathtt{ut}_1 \to \mathtt{ut}_2)^{\top}) & = & \mathtt{tt} \\
\mathcal{ST}(\mathrm{A},\, \lambda\mathtt{x.e},\, \downarrow\!\mathtt{t}_1 \to \downarrow\!\mathtt{t}_2) & = & \mathcal{ST}((\mathtt{x} : \mathtt{t}_1)\!: \mathrm{A},\, \mathtt{e},\, \mathtt{t}_2) \vee \\
 & & \mathcal{ST}((\mathtt{x} : \downarrow\!\mathtt{t}_1)\!: \mathrm{A},\, \mathtt{e},\, \downarrow\!\mathtt{t}_2) \\
\mathcal{ST}(\mathrm{A},\, \lambda\mathtt{x.e},\, (\mathtt{ut}_1 \to \mathtt{ut}_2)^{\mathbf{b}}) & = & \mathtt{ff} \\
\mathcal{ST}(\mathrm{A},\, \mathtt{e}_1\ \mathtt{e}_2,\, \mathtt{t}) & = & \mathcal{ST}(\mathrm{A},\, \mathtt{e}_1,\, (\mathrm{A},\, \mathtt{e}_2) \to \mathtt{t}) \\
\multicolumn{3}{l}{\mathcal{ST}(\mathrm{A},\, \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3,\, \mathtt{t}) =} \\
\multicolumn{3}{r}{(\mathcal{ST}(\mathrm{A},\, \mathtt{e}_1,\, \mathtt{Bool}^{\mathbf{b}}) \wedge \mathcal{I}(\varepsilon(\mathtt{t})^{\mathbf{b}},\, \mathtt{t})) \vee} \\
\multicolumn{3}{r}{(\mathcal{ST}(\mathrm{A},\, \mathtt{e}_2,\, \mathtt{t}) \wedge \mathcal{ST}(\mathrm{A},\, \mathtt{e}_3,\, \mathtt{t})) \wedge} \\
\multicolumn{3}{r}{(\mathcal{ST}(\mathrm{A},\, \mathtt{e}_1,\, \mathtt{Bool}^{\mathbf{n}}) \vee (\mathcal{ST}(\mathrm{A},\, \mathtt{e}_1,\, \mathtt{Bool}^{\top}) \wedge \mathrm{BOT}_{\mathrm{ST}}(\mathtt{t})))} \\
\mathcal{ST}(\mathrm{A},\, \mathtt{fix\ e,\ t}) & = & \mathcal{FIX}(\mathrm{A},\, \mathtt{e},\, \mathtt{t}) \\
\mathcal{ST}(\mathrm{A},\, \mathtt{c},\, \mathtt{t}) & = & \mathcal{I}(\mathtt{t}_{\mathtt{c}},\, \mathtt{t}) \\
\mathcal{ST}(\mathrm{A},\, \mathtt{e},\, \downarrow\!\mathtt{t}) & = & \mathcal{ST}(\mathrm{A},\, \mathtt{e},\, \mathtt{down}'(\mathtt{t})) \\
\mathcal{ST}(\mathrm{A},\, \mathtt{e},\, \mathtt{t}_1 \wedge \mathtt{t}_2) & = & \mathcal{ST}(\mathrm{A},\, \mathtt{e},\, \mathtt{t}_1) \wedge \mathcal{ST}(\mathrm{A},\, \mathtt{e},\, \mathtt{t}_2)
\end{array}
$$

$$
\begin{array}{lcll}
\mathcal{FIX}(\mathrm{A},\, \mathtt{e},\, \mathtt{t}) & = & \text{let} & l1 = \mathcal{ST}(\mathrm{A},\, \mathtt{e},\, \mathcal{ALL}(\varepsilon(\mathtt{t}))) \\
 & & & l2 = \mathcal{CHAIN}(\mathtt{t},\, l1) \\
 & & \text{in} & l2 \neq [\,]
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{CHAIN}(\mathtt{t},\, [\,]) & = & [\,] \\
\mathcal{CHAIN}(\mathtt{t},\, \mathtt{t}' : l) & = & \begin{cases} \mathtt{t}' : \mathcal{CHAIN}(\mathtt{t},\, l), & \text{if } \mathrm{P}(\mathtt{t},\, \mathtt{t}') \\ \mathcal{CHAIN}(\mathtt{t},\, l), & \text{otherwise} \end{cases}
\end{array}
$$

$$
\begin{array}{lcl}
\mathrm{P}(\mathtt{t},\mathtt{t}') & = & (\mathtt{t}' = (\mathtt{t}_1 \to \mathtt{t}_2) \wedge (\mathtt{t}_2 \to \mathtt{t}_3) \wedge \ldots \wedge (\mathtt{t}_n \to \mathtt{t})) \\
 & & \wedge \mathrm{BOT}_{\mathrm{ST}}(\mathtt{t}_1) \\
 & & \wedge \exists p, q : p < q \wedge \mathtt{t}_q \leq_{\mathrm{ST}} \mathtt{t}_p
\end{array}
$$

Figure 4.6: The Algorithm $\mathcal{ST}$

has been used: again we must ensure that both branches has the type $\mathtt{t}$ in order for the whole term to have the lazy strictness and totality type $\mathtt{t}$, the test, i.e. $\mathtt{e}_1$, must have the type $\mathtt{Bool}^\top$, and the predicate $\mathrm{BOT}_{\mathrm{ST}}$ must be true on the type $\mathtt{t}$.

The clause for fixpoints just makes a call to the function $\mathcal{FIX}$. The function $\mathcal{FIX}$ takes three arguments, an assumption list, A, a term, $\mathtt{e}$, and a type, $\mathtt{t}$, the result is either $\mathtt{tt}$ or $\mathtt{ff}$. The idea is that the result is $\mathtt{tt}$, when $\mathrm{A} \vdash_{\mathrm{L}} \mathtt{fix}\ \mathtt{e} : \mathtt{t}$ can be inferred. The function $\mathcal{FIX}$ uses the function $\mathcal{ALL}$ which given a standard type returns the list of all the strictness and totality types with that standard type as its underlying type. We overload the function $\mathcal{ST}$ and apply it to a list of types. The result is the list of all the types, $\mathtt{t}$, from the list, where $\mathcal{ST}(\mathrm{A}, \mathtt{e}, \mathtt{t}) = \mathtt{tt}$. This is used in the definition of $\mathcal{FIX}$ to compute the list, $l1$, of all the types that $\mathtt{e}$ can have. Now we have to find the list, $l2$, of all the types, $\mathtt{t}'$, such that:

$$\begin{array}{l}
\mathtt{t}' = (\mathtt{t}'_1 \to \mathtt{t}'_2) \wedge (\mathtt{t}'_2 \to \mathtt{t}'_3) \wedge \ldots \wedge (\mathtt{t}'_{n-1} \to \mathtt{t}'_n) \\
\forall i : \mathtt{t}'_i \in l1 \\
\mathtt{t}'_n = \mathtt{t} \\
\mathrm{BOT}_{\mathrm{ST}}(\mathtt{t}'_1) \\
\exists p, q : p < q \wedge \mathtt{t}'_q \leq_{\mathrm{L}} \mathtt{t}'_p
\end{array}$$

This is done by the function $\mathcal{CHAIN}$.

The clause for constants is analogous to the clause for variables just as the the rule $[\mathrm{const}_{\mathrm{L}}]$ is analogous to the rule $[\mathrm{var}_{\mathrm{L}}]$.

The clause

$$\mathcal{ST}(\mathrm{A}, \mathtt{e}, \downarrow\mathtt{t}) \;=\; \mathcal{ST}(\mathrm{A}, \mathtt{e}, \mathtt{down}'(\mathtt{t}))$$

correspond to the rule $[\mathrm{down}_{\mathrm{L}}]$.

In the clause for conjunction we make two recursive calls using the two conjuncts of the lazy strictness and totality type. This correspond to the rule $[\mathrm{conj}_{\mathrm{L}}]$.

The algorithm, $\mathcal{I}$, for checking the coercions is in displayed Figure 4.7 and Figure 4.8. The definition is in most cases straightforward: The first clause in Figure 4.7 says that $\mathtt{t}$ cannot be less than $\mathtt{t}'$ whenever they do not have the same underlying type.

$$
\begin{array}{lcl}
\mathcal{I}(\mathtt{t},\, \mathtt{t}') & = & \mathtt{ff}\text{, if } \varepsilon(\mathtt{t}) \neq \varepsilon(\mathtt{t}') \\
\mathcal{I}(\mathtt{t},\, \mathtt{t}) & = & \mathtt{tt} \\
\mathcal{I}(\mathtt{t},\, \mathtt{ut}^{\top}) & = & \mathtt{tt} \\
\mathcal{I}(\mathtt{ut}^{\mathbf{n}},\, \mathtt{ut}^{\mathbf{b}}) & = & \mathtt{ff} \\
\mathcal{I}(\mathtt{ut}^{\mathbf{b}},\, \mathtt{ut}^{\mathbf{n}}) & = & \mathtt{ff} \\
\mathcal{I}(\mathtt{ut}^{\top},\, \mathtt{ut}^{\mathbf{n}}) & = & \mathtt{ff} \\
\mathcal{I}(\mathtt{ut}^{\top},\, \mathtt{ut}^{\mathbf{b}}) & = & \mathtt{ff} \\
\mathcal{I}(\downarrow(\mathtt{ut}^{\mathbf{b}}),\, \mathtt{ut}^{\mathbf{b}}) & = & \mathtt{tt} \\
\mathcal{I}(\mathtt{t}_1 \rightarrow \mathtt{t}_2,\, (\mathtt{ut}_1 \rightarrow \mathtt{ut}_2)^{\mathbf{b}}) & = & \mathtt{ff} \\
\mathcal{I}(\downarrow\mathtt{t},\, \mathtt{ut}^{\mathbf{n}}) & = & \mathtt{ff} \\
\mathcal{I}(\mathtt{t}',\, \mathtt{t}_1 \wedge \mathtt{t}_2) & = & \mathcal{I}(\mathtt{t}',\, \mathtt{t}_1) \wedge \mathcal{I}(\mathtt{t}',\, \mathtt{t}_2) \\
\mathcal{I}(\mathtt{t}_1 \rightarrow \mathtt{t}_2,\, \mathtt{t}_1' \rightarrow \mathtt{t}_2') & = & (\mathcal{I}(\mathtt{t}_1',\, \mathtt{t}_1) \wedge \mathcal{I}(\mathtt{t}_2,\, \mathtt{t}_2')) \vee \\
 & & (\mathcal{I}(\mathtt{t}_1',\, \downarrow\mathtt{t}_1) \wedge \mathcal{I}(\downarrow\mathtt{t}_2,\, \mathtt{t}_2')) \vee \\
 & & \mathcal{I}(\varepsilon(\mathtt{t}_2)^{\top},\, \mathtt{t}_2') \\
\mathcal{I}(\downarrow\mathtt{t}_1,\, \downarrow\mathtt{t}_2) & = & \mathcal{I}(\mathtt{t}_1,\, \mathtt{t}_2) \vee \\
 & & \mathcal{I}(\mathtt{down}(\mathtt{t}_1),\, \downarrow\mathtt{t}_2) \vee \\
 & & \mathcal{I}(\downarrow\mathtt{t}_1,\, \mathtt{down}'(\mathtt{t}_2)) \vee \\
 & & \mathcal{I}(\mathtt{down}(\mathtt{t}_1),\, \mathtt{down}'(\mathtt{t}_2)) \\
\mathcal{I}(\mathtt{t}_1 \rightarrow \mathtt{t}_2,\, (\mathtt{ut}_1 \rightarrow \mathtt{ut}_2)^{\mathbf{n}}) & = & \mathcal{I}(\mathtt{ut}_1{}^{\mathbf{n}},\, \mathtt{t}_1) \wedge \mathcal{I}(\mathtt{t}_2,\, \mathtt{ut}_2{}^{\mathbf{n}}) \\
\mathcal{I}(\mathtt{t}_1 \wedge \mathtt{t}_2,\, \mathtt{ut}^{\mathbf{n}}) & = & \mathcal{I}(\mathtt{t}_1,\, \mathtt{ut}^{\mathbf{n}}) \vee \mathcal{I}(\mathtt{t}_2,\, \mathtt{ut}^{\mathbf{n}}) \\
\mathcal{I}(\mathtt{t}_1 \wedge \mathtt{t}_2,\, \mathtt{ut}^{\mathbf{b}}) & = & \mathcal{I}(\mathtt{t}_1,\, \mathtt{ut}^{\mathbf{b}}) \vee \mathcal{I}(\mathtt{t}_2,\, \mathtt{ut}^{\mathbf{b}}) \\
\mathcal{I}((\mathtt{ut}_1 \rightarrow \mathtt{ut}_2)^{\mathbf{n}},\, \mathtt{t}_1 \rightarrow \mathtt{t}_2) & = & \mathcal{I}(\mathtt{ut}_2{}^{\top},\, \mathtt{t}_2) \\
\mathcal{I}((\mathtt{ut}_1 \rightarrow \mathtt{ut}_2)^{\mathbf{b}},\, \mathtt{t}_1 \rightarrow \mathtt{t}_2) & = & \mathcal{I}(\mathtt{ut}_2{}^{\top},\, \mathtt{t}_2) \vee \mathcal{I}(\mathtt{ut}_2{}^{\mathbf{b}},\, \mathtt{t}_2) \\
\mathcal{I}((\mathtt{ut}_1 \rightarrow \mathtt{ut}_2)^{\top},\, \mathtt{t}_1 \rightarrow \mathtt{t}_2) & = & \mathcal{I}(\mathtt{ut}_2{}^{\top},\, \mathtt{t}_2)
\end{array}
$$

Figure 4.7: The Algorithm $\mathcal{I}$ (Part 1)

The second clause corresponds to the rule [ref] in Figure 3.1 and the third clause to the rule [top1].

The next four clauses return $\mathtt{ff}$ due to that we cannot construct proof-tree for them. The next clause correspond to the rule [$\downarrow$4]. The clause for $\mathtt{t}_1 \rightarrow \mathtt{t}_2$ and $(\mathtt{ut}_1 \rightarrow \mathtt{ut}_2)^{\mathbf{b}}$ returns $\mathtt{ff}$, the reason is that we are not able to infer $\mathtt{t}_1 \rightarrow \mathtt{t}_2 \leq_{\mathrm{L}} (\mathtt{ut}_1 \rightarrow \mathtt{ut}_2)^{\mathbf{b}}$, since the most we can say about the functions in $\mathtt{t}_1 \rightarrow \mathtt{t}_2$ is that when applied, then the result is described by $\mathtt{t}_2$, hence they are not included in the functions without a WHNF.

The terms of type $\downarrow\mathtt{t}$ may include $\perp$, however the term of type $\mathtt{ut}^{\mathbf{n}}$ does

$$
\begin{array}{lcl}
\mathcal{I}(\mathtt{t}_1 \wedge \mathtt{t}_2,\, \mathtt{t}'_1 \rightarrow \mathtt{t}'_2) & = & \mathcal{I}(\mathtt{t}_1,\, \mathtt{t}'_1 \rightarrow \mathtt{t}'_2) \vee \mathcal{I}(\mathtt{t}_2,\, \mathtt{t}'_1 \rightarrow \mathtt{t}'_2) \\
 & & \vee\, \mathcal{I}(\varepsilon(\mathtt{t}_1)^{\top},\, \mathtt{t}'_2) \vee \\
 & & ((\mathtt{t}_1 = \mathtt{t}''_1 \rightarrow \mathtt{t}'_2) \wedge \\
 & & (\mathtt{t}_2 = \mathtt{t}''_1 \rightarrow \mathtt{t}''_3) \wedge (\mathtt{t}'_1 = \mathtt{t}''_1) \wedge \\
 & & (\mathtt{t}'_2 = \mathtt{t}''_2 \wedge \mathtt{t}''_3)) \\
\mathcal{I}(\downarrow\mathtt{t},\, \mathtt{t}'_1 \rightarrow \mathtt{t}'_2) & = & \mathcal{I}(\varepsilon(\mathtt{t}'_2)^{\top},\, \mathtt{t}'_2) \vee \\
 & & (\mathtt{t} = (\mathtt{ut}_1 \rightarrow \mathtt{ut}_2)^{\mathbf{b}} \wedge \mathcal{I}(\varepsilon(\mathtt{t}'_2)^{\mathbf{b}},\, \mathtt{t}'_2)) \vee \\
 & & (\mathtt{t} = (\mathtt{t}''_1 \rightarrow \mathtt{t}''_2) \wedge \\
 & & \quad \mathcal{I}(\mathtt{t}''_1 \rightarrow \downarrow\mathtt{t}''_2,\, \mathtt{t}'_1 \rightarrow \mathtt{t}'_2)) \vee \\
 & & (\mathtt{t} = (\mathtt{t}''_1 \wedge \mathtt{t}''_2) \wedge \\
 & & \quad \mathcal{I}(\downarrow\mathtt{t}''_1 \wedge \downarrow\mathtt{t}''_2,\, \mathtt{t}'_1 \rightarrow \mathtt{t}'_2)) \\
\mathcal{I}(\mathtt{ut}^{\mathbf{n}},\, \downarrow\mathtt{t}) & = & \mathcal{I}(\mathtt{ut}^{\mathbf{n}},\, \mathtt{t}) \\
\mathcal{I}(\mathtt{ut}^{\mathbf{b}},\, \downarrow\mathtt{t}) & = & \mathcal{I}(\mathtt{ut}^{\mathbf{b}},\, \mathtt{t}) \vee (\mathtt{t} = \mathtt{ut}^{\mathbf{n}}) \\
\mathcal{I}(\mathtt{ut}^{\top},\, \downarrow\mathtt{t}) & = & \mathcal{I}(\mathtt{ut}^{\top},\, \mathtt{t}) \vee (\mathtt{t} = \mathtt{ut}^{\mathbf{n}}) \\
\mathcal{I}(\mathtt{t}_1 \rightarrow \mathtt{t}_2,\, \downarrow\mathtt{t}') & = & \mathcal{I}(\mathtt{t}_1 \rightarrow \mathtt{t}_2,\, \mathtt{t}') \\
\mathcal{I}(\mathtt{t}_1 \wedge \mathtt{t}_2,\, \downarrow\mathtt{t}') & = & \mathcal{I}(\mathtt{t}_1 \wedge \mathtt{t}_2,\, \mathtt{t}') \vee \mathcal{I}(\mathtt{t}_1,\, \downarrow\mathtt{t}') \vee \mathcal{I}(\mathtt{t}_2,\, \downarrow\mathtt{t}') \\
\mathcal{I}((\mathrm{A},\, \mathtt{e}),\, \mathtt{t}) & = & \mathcal{ST}(\mathrm{A},\, \mathtt{e},\, \mathtt{t}) \\
\mathcal{I}(\mathtt{t},\, (\mathrm{A},\, \mathtt{e})) & = & \mathcal{C}(\mathcal{ALL}(\varepsilon(\mathtt{t})),\, \mathrm{A},\, \mathtt{e},\, \mathtt{t})
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{C}([\,],\, \mathrm{A},\, \mathtt{e},\, \mathtt{t}) & = & \mathtt{tt} \\
\mathcal{C}(\mathtt{t}'{:}l,\, \mathrm{A},\, \mathtt{e},\, \mathtt{t}) & = & \begin{cases} \mathcal{I}(\mathtt{t},\, \mathtt{t}') \wedge \mathcal{C}(l,\, \mathrm{A},\, \mathtt{e},\, \mathtt{t}), & \text{if } \mathcal{ST}(\mathrm{A},\, \mathtt{e},\, \mathtt{t}') \\ \mathcal{C}(l,\, \mathrm{A},\, \mathtt{e},\, \mathtt{t}), & \text{otherwise} \end{cases}
\end{array}
$$

Figure 4.8: The Algorithm $\mathcal{I}$ (Part 2)

not include $\perp$, therefore we let

$$\mathcal{I}(\downarrow\mathtt{t}, \mathtt{ut}^{\mathbf{n}}) \;=\; \mathtt{ff}$$

The clause for $\mathtt{t}'$ and $\mathtt{t}_1 \wedge \mathtt{t}_2$ correspond to the rule [$\wedge$3], where we must ensure both $\mathtt{t}' \ \leq_{\mathrm{L}}\ \mathtt{t}_1$ and $\mathtt{t}' \ \leq_{\mathrm{L}}\ \mathtt{t}_2$.

To construct the proof-tree for $\mathtt{t}_1 \rightarrow \mathtt{t}_2 \ \leq_{\mathrm{L}} \ \mathtt{t}'_1 \rightarrow \mathtt{t}'_2$ there are three possibilities:

- use the rule [$\rightarrow$]
- use the rule [monotone] followed by [trans]
- use the rule [top1] and [top2] followed by [trans]

this is exactly what is expressed by the clause in the definition of $\mathcal{I}$.

The clause for $\downarrow\mathtt{t}_1$ and $\downarrow\mathtt{t}_2$ is a combination of trying the different possibilities for moving the $\downarrow$-construct inwards using the two function `down` and `down`$'$ and the rule [$\downarrow$2].

The clause $\mathtt{t}_1 \rightarrow \mathtt{t}_2$ and $(\mathtt{ut}_1 \rightarrow \mathtt{ut}_2)^{\mathbf{n}}$ correspond to first applying the rule [trans] and then [notbot].

All the clauses with $\mathtt{t}_1 \wedge \mathtt{t}_2$ on the left-hand side correspond to the rule [$\wedge$1] or [$\wedge$2]. In the first clause in Figure 4.8 it is also possible to apply the rule [$\wedge$4] or the [top1] and [top2]-rules. In the clause for $\mathtt{t}_1 \wedge \mathtt{t}_2$ and $\downarrow\mathtt{t}$ it is also possible to apply the rule [$\downarrow$6] and [trans].

The three last clauses in Figure 4.7 correspond to applying the rule [trans], [top1], and [top2]. In the second last clause it is also possible to apply the rule [bot].

In all the clauses involving $\downarrow$-types some of the [$\downarrow i$]-rules has been applied.

The last two clauses in Figure 4.8 for $\mathcal{I}$ is for the lazy strictness and totality type. When the lazy-construct is on the left side we use the algorithm $\mathcal{ST}$; this corresponding to the rule [$\mathrm{env_L}$]. The algorithm $\mathcal{C}$ is used in the last clause to make the test correspond to the rule [$\mathrm{env_R}$]. The function $\mathcal{C}$ takes a list of strictness and totality type, the assumption list, the term, and a type and check the hypothesis of the rule [$\mathrm{env_R}$].

4.3 Soundness

The algorithms are sound with respect to the *lazy* strictness and totality type inference system:

Lemma 4.11 *Soundness of $\mathcal{ST}$ and $\mathcal{I}$*
We have

$$
\begin{array}{ll}
& (\mathcal{ST}(\mathrm{A}, \mathtt{e}, \mathtt{t}) = \mathtt{true} \wedge \mathcal{I}(\mathtt{t}_1, \mathtt{t}_2) = \mathtt{true}) \\
\Downarrow & \\
& (\mathrm{A} \vdash_{\mathrm{L}} \mathtt{e} : \mathtt{t} \wedge \mathtt{t}_1 \leq_{\mathrm{L}} \mathtt{t}_2)
\end{array}
$$

□

Proof We will assume that both $\mathcal{ST}(\mathrm{A}, \mathtt{e}, \mathtt{t})$ and $\mathcal{I}(\mathtt{t}_1, \mathtt{t}_2)$ are true and then we will prove $\mathrm{A} \vdash_{\mathrm{L}} \mathtt{e} : \mathtt{t}$ and $\mathtt{t}_1 \leq_{\mathrm{L}} \mathtt{t}_2$ by induction on $\mathtt{e}$, $\mathtt{t}$, and $\mathtt{t}_1$, $\mathtt{t}_2$.

For the full proof see Appendix page 329. ■

Finally we have that the inference algorithm is sound with respect to the strictness and totality type inference system:

Theorem 4.12
$\mathcal{ST}(\mathrm{A}, \mathtt{e}, \mathtt{t}) \Rightarrow \mathtt{expand}(\mathrm{A}) \vdash \mathtt{e} : \mathtt{expand}(\mathtt{t})$ □

Proof First we assume $\mathcal{ST}(\mathrm{A}, \mathtt{e}, \mathtt{t})$ and by Lemma 4.11 we get

$$\mathrm{A} \vdash_{\mathrm{L}} \mathtt{e} : \mathtt{t}$$

By applying Lemma 4.10 we get

$$\mathtt{expand}(\mathrm{A}) \vdash_{\mathrm{S}} \mathtt{e} : \mathtt{expand}(\mathtt{t})$$

and by Lemma 4.8 we have

$$\mathtt{expand}(\mathrm{A}) \vdash \mathtt{e} : \mathtt{expand}(\mathtt{t})$$

as required. ■

4.3.1 Discussion of Completeness of the Algorithm

We conjecture that the algorithm is complete with respect to the *lazy* strictness and totality type inference system, and hence (using Lemma 4.14 and Conjecture 4.15 below) with respect to the structural strictness and totality inference system Figure 4.4.

Conjecture 4.13 *Completeness of $\mathcal{ST}$ and $\mathcal{I}$*
We have

$$(\mathrm{A} \vdash_{\mathrm{L}} \mathtt{e} : \mathtt{t}) \wedge (\mathtt{t}_1 \leq_{\mathrm{L}} \mathtt{t}_2)$$
$$\Downarrow$$
$$\mathcal{ST}(\mathrm{A}, \mathtt{e}, \mathtt{t}) = \mathtt{true} \wedge \mathcal{I}(\mathtt{t}_1, \mathtt{t}_2) = \mathtt{true}$$

□

Sketch of proof For the proof we will assume $\mathrm{A} \vdash_{\mathrm{L}} \mathtt{e} : \mathtt{t}$ and $\mathtt{t}_1 \leq_{\mathrm{L}} \mathtt{t}_2$ and then we will show $\mathcal{ST}(\mathrm{A}, \mathtt{e}, \mathtt{t}) = \mathtt{tt}$ and $\mathcal{I}(\mathtt{t}_1, \mathtt{t}_2) = \mathtt{tt}$ by induction on the proof-trees for $\vdash_{\mathrm{L}}$ and $\leq_{\mathrm{L}}$. Most cases are easy — the only non-trivial case is to show that the algorithm $\mathcal{I}$ is transitive, i.e. that $\mathcal{I}(\mathtt{t}_1, \mathtt{t}_2) = \mathtt{tt}$ and $\mathcal{I}(\mathtt{t}_2, \mathtt{t}_3) = \mathtt{tt}$ implies that $\mathcal{I}(\mathtt{t}_1, \mathtt{t}_3) = \mathtt{tt}$. ■

We can show that the subtyping relation defined by $\leq_{\mathrm{L}}$ is complete with respect to the subtyping relation defined by $\leq_{\mathrm{D}}$:

Lemma 4.14 *Completeness of $\leq_{\mathrm{L}}$*
$\mathtt{expand}(\mathtt{t}_1) \leq_{\mathrm{D}} \mathtt{expand}(\mathtt{t}_2) \Rightarrow \mathtt{t}_1 \leq_{\mathrm{L}} \mathtt{t}_2$ □

Proof We will assume $\mathtt{expand}(\mathtt{t}_1) \leq_{\mathrm{D}} \mathtt{expand}(\mathtt{t}_2)$ and then we will show $\mathtt{t}_1 \leq_{\mathrm{L}} \mathtt{t}_2$ by induction on the prooftree for $\mathtt{expand}(\mathtt{t}_1) \leq_{\mathrm{D}} \mathtt{expand}(\mathtt{t}_2)$.

For the full proof see Appendix page 344. ■

We conjecture that the lazy strictness and totality inference system is complete with respect to the structural strictness and totality inference system:

Conjecture 4.15 *Completeness of $\vdash_{\mathrm{L}}$*
$\mathtt{expand}(\mathrm{A}) \vdash_{\mathrm{S}} \mathtt{e} : \mathtt{expand}(\mathtt{t}) \Rightarrow \mathrm{A} \vdash_{\mathrm{L}} \mathtt{e} : \mathtt{t}$ □

Sketch of proof We will assume $\mathtt{expand}(\mathrm{A}) \vdash_{\mathrm{S}} \mathtt{e} : \mathtt{expand}(\mathtt{t})$ and show $\mathrm{A} \vdash_{\mathrm{L}} \mathtt{e} : \mathtt{t}$ by induction on the proof-tree for

$$\mathtt{expand}(\mathrm{A}) \vdash_{\mathrm{S}} \mathtt{e} : \mathtt{expand}(\mathtt{t})$$

The proof is mostly straightforward but we will need the following property of the structural strictness and totality inference system:

$$\mathrm{A} \vdash_{\mathrm{S}} \mathtt{e} : \mathtt{t}_1 \wedge (\mathtt{t}_1 \leq_{\mathrm{ST}} \mathtt{t}_2)$$
$$\Downarrow$$
$$\mathrm{A} \vdash_{\mathrm{S}} \mathtt{e} : \mathtt{t}_2$$

This property will be a consequence of completeness of the structural inference system with respect to the original inference system in Chapter 3. It is not clear whether this property can be show directly as the inference systems are now. ■

The algorithm is however *not* complete with respect to the strictness and totality inference system in Chapter 3. The reason is that the structural strictness and totality inference system is not complete with respect to the analysis in Chapter 3. The problem arises from the fact that

$$\downarrow(\mathtt{t}_1 \wedge \mathtt{t}_2) \not\equiv_{\mathrm{D}} \downarrow\mathtt{t}_1 \wedge \downarrow\mathtt{t}_2$$

We only have

$$\downarrow(\mathtt{t}_1 \wedge \mathtt{t}_2) \leq_{\mathrm{ST}} \downarrow\mathtt{t}_1 \wedge \downarrow\mathtt{t}_2$$

since $\mathtt{t}_1 \wedge \mathtt{t}_2$ can be empty. The rule [down$_{\mathrm{S}}$] says

$$\begin{array}{ll} & \mathrm{A} \vdash_{\mathrm{S}} \mathtt{e} \ : \mathtt{t}_1 \wedge \mathtt{t}_2 \\ \Downarrow & \\ & \mathrm{A} \vdash_{\mathrm{S}} \mathtt{e} \ : \downarrow(\mathtt{t}_1 \wedge \mathtt{t}_2) \end{array}$$

but we do not have that

$$\mathrm{A} \vdash_{\mathrm{S}} \mathtt{e} \ : \downarrow(\mathtt{t}_1 \wedge \mathtt{t}_2)$$

implies

$$\mathrm{A} \vdash_{\mathrm{S}} \mathtt{e} \ : \mathtt{t}_1 \wedge \mathtt{t}_2$$

To gain completeness of the structural strictness and totality inference system with respect to the inference system in Chapter 3, we must find a strictness and totality type, $\mathtt{t}$, that is simpler than $\downarrow(\mathtt{t}_1 \wedge \mathtt{t}_2)$, such that

$$\begin{array}{ll} & \mathrm{A} \vdash_{\mathrm{S}} \mathtt{e} \ : \downarrow(\mathtt{t}_1 \wedge \mathtt{t}_2) \\ \Updownarrow & \\ & \mathrm{A} \vdash_{\mathrm{S}} \mathtt{e} \ : \mathtt{t} \end{array}$$

One idea is to try to let $\mathtt{t}$ be $\downarrow\mathtt{t}_1 \wedge \downarrow\mathtt{t}_2$ in the cases where $\mathtt{t}_1 \wedge \mathtt{t}_2$ is not empty. We will now sketch a solution in the case where neither $\mathtt{t}_1$ nor $\mathtt{t}_2$ is a conjunction and they have the same structure (e.g. they are both an annotated underlying type or they are both function types). For the base cases we have:

$$\begin{array}{rcl}
\downarrow(\mathtt{ut}^{\mathbf{n}} \wedge \mathtt{ut}^{\mathbf{n}}) & = & \downarrow\mathtt{ut}^{\mathbf{n}} = \mathtt{ut}^{\top} \\
\downarrow(\mathtt{ut}^{\mathbf{b}} \wedge \mathtt{ut}^{\mathbf{n}}) & = & \mathtt{ut}^{\mathbf{n}} \wedge \mathtt{ut}^{\mathbf{b}} = \downarrow(\mathtt{ut}^{\mathbf{n}} \wedge \mathtt{ut}^{\mathbf{b}}) \\
\downarrow(\mathtt{ut}^{\top} \wedge \mathtt{ut}^{\mathbf{n}}) & = & \downarrow\mathtt{ut}^{\top} \wedge \downarrow\mathtt{ut}^{\mathbf{n}} = \mathtt{ut}^{\top} = \downarrow(\mathtt{ut}^{\mathbf{n}} \wedge \mathtt{ut}^{\top}) \\
\downarrow(\mathtt{ut}^{\mathbf{b}} \wedge \mathtt{ut}^{\mathbf{b}}) & = & \downarrow\mathtt{ut}^{\mathbf{b}} = \mathtt{ut}^{\mathbf{b}} \\
\downarrow(\mathtt{ut}^{\top} \wedge \mathtt{ut}^{\mathbf{b}}) & = & \downarrow\mathtt{ut}^{\top} \wedge \downarrow\mathtt{ut}^{\mathbf{b}} = \mathtt{ut}^{\mathbf{b}} = \downarrow(\mathtt{ut}^{\mathbf{b}} \wedge \mathtt{ut}^{\top}) \\
\downarrow(\mathtt{ut}^{\top} \wedge \mathtt{ut}^{\top}) & = & \downarrow\mathtt{ut}^{\top} = \mathtt{ut}^{\top}
\end{array}$$

In the case of function types we will first expand the type with all possible conjunctions:

$$\begin{array}{l}
(\mathtt{t}_1 \rightarrow \mathtt{t}_2) \wedge (\mathtt{t}_3 \rightarrow \mathtt{t}_4) \\
\quad \equiv_{\mathrm{D}} \quad (\mathtt{t}_1 \rightarrow \mathtt{t}_2) \wedge (\mathtt{t}_3 \rightarrow \mathtt{t}_4) \wedge ((\mathtt{t}_1 \wedge \mathtt{t}_3) \rightarrow (\mathtt{t}_2 \wedge \mathtt{t}_4))
\end{array}$$

It is straightforward to show that this rule is sound with respect to the semantics.

Lemma 4.16
We have

$$\begin{array}{l}
[\![(\mathtt{t}_1 \rightarrow \mathtt{t}_2) \wedge (\mathtt{t}_3 \rightarrow \mathtt{t}_4)]\!] \\
\quad = \quad [\![(\mathtt{t}_1 \rightarrow \mathtt{t}_2) \wedge (\mathtt{t}_3 \rightarrow \mathtt{t}_4) \wedge ((\mathtt{t}_1 \wedge \mathtt{t}_3) \rightarrow (\mathtt{t}_2 \wedge \mathtt{t}_4))]\!]
\end{array}$$

$\square$

Proof From the coercion rules we can infer

$$\begin{array}{l}
(\mathtt{t}_1 \rightarrow \mathtt{t}_2) \wedge (\mathtt{t}_3 \rightarrow \mathtt{t}_4) \wedge ((\mathtt{t}_1 \wedge \mathtt{t}_3) \rightarrow (\mathtt{t}_2 \wedge \mathtt{t}_4)) \\
\quad \leq_{\mathrm{ST}} \quad (\mathtt{t}_1 \rightarrow \mathtt{t}_2) \wedge (\mathtt{t}_3 \rightarrow \mathtt{t}_4)
\end{array}$$

so we only have to show

$$\begin{array}{l}
[\![(\mathtt{t}_1 \rightarrow \mathtt{t}_2) \wedge (\mathtt{t}_3 \rightarrow \mathtt{t}_4)]\!] \\
\quad \subseteq \quad [\![(\mathtt{t}_1 \rightarrow \mathtt{t}_2) \wedge (\mathtt{t}_3 \rightarrow \mathtt{t}_4) \wedge ((\mathtt{t}_1 \wedge \mathtt{t}_3) \rightarrow (\mathtt{t}_2 \wedge \mathtt{t}_4))]\!]
\end{array}$$

Assume

$$\begin{array}{rcl}
f & \in & [\![(\mathtt{t}_1 \rightarrow \mathtt{t}_2) \wedge (\mathtt{t}_3 \rightarrow \mathtt{t}_4)]\!] \\
 & = & \{f \mid f[\![\mathtt{t}_1]\!] \subseteq [\![\mathtt{t}_2]\!]\} \cap \{f \mid f[\![\mathtt{t}_3]\!] \subseteq [\![\mathtt{t}_4]\!]\}
\end{array}$$

We have

$$\begin{array}{rcl} f[\![\mathtt{t}_1 \wedge \mathtt{t}_3]\!] & = & f([\![\mathtt{t}_1]\!] \cap [\![\mathtt{t}_3]\!]) \\ & \subseteq & f[\![\mathtt{t}_1]\!] \cap f[\![\mathtt{t}_3]\!] \\ & \subseteq & [\![\mathtt{t}_2]\!] \cap [\![\mathtt{t}_4]\!] \\ & = & [\![\mathtt{t}_2 \wedge \mathtt{t}_4]\!] \end{array}$$

and hence

$$f \in [\![(\mathtt{t}_1 \wedge \mathtt{t}_3) \rightarrow (\mathtt{t}_2 \wedge \mathtt{t}_4)]\!]$$

but we also had that

$$f \;\in\; [\![(\mathtt{t}_1 \rightarrow \mathtt{t}_2) \wedge (\mathtt{t}_3 \rightarrow \mathtt{t}_4)]\!]$$

so it must be the case that

$$\begin{array}{rcl} f & \in & [\![(\mathtt{t}_1 \rightarrow \mathtt{t}_2) \wedge (\mathtt{t}_3 \rightarrow \mathtt{t}_4)]\!] \cap [\![(\mathtt{t}_1 \wedge \mathtt{t}_3) \rightarrow (\mathtt{t}_2 \wedge \mathtt{t}_4)]\!] \\ & = & [\![(\mathtt{t}_1 \rightarrow \mathtt{t}_2) \wedge (\mathtt{t}_3 \rightarrow \mathtt{t}_4) \wedge ((\mathtt{t}_1 \wedge \mathtt{t}_3) \rightarrow (\mathtt{t}_2 \wedge \mathtt{t}_4))]\!] \end{array}$$

as required. ∎

The next step is to move the $\downarrow$ inward:

$$\begin{array}{l} \downarrow((\mathtt{t}_1 \rightarrow \mathtt{t}_2) \wedge (\mathtt{t}_3 \rightarrow \mathtt{t}_4) \wedge ((\mathtt{t}_1 \wedge \mathtt{t}_3) \rightarrow (\mathtt{t}_2 \wedge \mathtt{t}_4))) \\ \quad \equiv_{\mathrm{D}} \quad (\mathtt{t}_1 \rightarrow \downarrow\mathtt{t}_2) \wedge (\mathtt{t}_3 \rightarrow \downarrow\mathtt{t}_4) \wedge ((\mathtt{t}_1 \wedge \mathtt{t}_3) \rightarrow \downarrow(\mathtt{t}_2 \wedge \mathtt{t}_4)) \end{array}$$

To see that this construct is sensible consider the empty type:

$$(\mathtt{Int}^{\mathbf{n}} \rightarrow \mathtt{Int}^{\mathbf{n}}) \wedge (\mathtt{Int}^{\mathbf{n}} \rightarrow \mathtt{Int}^{\mathbf{b}})$$

We expand the type to:

$$\begin{array}{l} (\mathtt{Int}^{\mathbf{n}} \rightarrow \mathtt{Int}^{\mathbf{n}}) \wedge (\mathtt{Int}^{\mathbf{n}} \rightarrow \mathtt{Int}^{\mathbf{b}}) \wedge \\ \qquad\qquad\qquad ((\mathtt{Int}^{\mathbf{n}} \wedge \mathtt{Int}^{\mathbf{n}}) \rightarrow (\mathtt{Int}^{\mathbf{n}} \wedge \mathtt{Int}^{\mathbf{b}})) \end{array}$$

We have

$$\begin{array}{rl} \multicolumn{2}{l}{\downarrow((\mathtt{Int}^{\mathbf{n}} \rightarrow \mathtt{Int}^{\mathbf{n}}) \wedge (\mathtt{Int}^{\mathbf{n}} \rightarrow \mathtt{Int}^{\mathbf{b}}) \wedge ((\mathtt{Int}^{\mathbf{n}} \wedge \mathtt{Int}^{\mathbf{n}}) \rightarrow (\mathtt{Int}^{\mathbf{n}} \wedge \mathtt{Int}^{\mathbf{b}})))} \\ = & (\mathtt{Int}^{\mathbf{n}} \rightarrow \downarrow\mathtt{Int}^{\mathbf{n}}) \wedge (\mathtt{Int}^{\mathbf{n}} \rightarrow \downarrow\mathtt{Int}^{\mathbf{b}}) \wedge \\ & ((\mathtt{Int}^{\mathbf{n}} \wedge \mathtt{Int}^{\mathbf{n}}) \rightarrow \downarrow(\mathtt{Int}^{\mathbf{n}} \wedge \mathtt{Int}^{\mathbf{b}})) \\ = & (\mathtt{Int}^{\mathbf{n}} \rightarrow \mathtt{Int}^{\top}) \wedge (\mathtt{Int}^{\mathbf{n}} \rightarrow \mathtt{Int}^{\mathbf{b}}) \\ & \wedge((\mathtt{Int}^{\mathbf{n}} \wedge \mathtt{Int}^{\mathbf{n}}) \rightarrow (\mathtt{Int}^{\mathbf{n}} \wedge \mathtt{Int}^{\mathbf{b}})) \end{array}$$

which is still an empty strictness and totality type as required. One part of the semantically soundness of the last rule is easy: from the coercion rules we get

$$\begin{aligned} &\downarrow((\mathtt{t}_1 \to \mathtt{t}_2) \wedge (\mathtt{t}_3 \to \mathtt{t}_4) \wedge ((\mathtt{t}_1 \wedge \mathtt{t}_3) \to (\mathtt{t}_2 \wedge \mathtt{t}_4))) \\ &\quad \leq_{\mathrm{ST}} \quad (\mathtt{t}_1 \to \downarrow\mathtt{t}_2) \wedge (\mathtt{t}_3 \to \downarrow\mathtt{t}_4) \wedge ((\mathtt{t}_1 \wedge \mathtt{t}_3) \to \downarrow(\mathtt{t}_2 \wedge \mathtt{t}_4)) \end{aligned}$$

The second part is the non-trivial part.

Conjecture 4.17
We have

$$\begin{aligned} &[\![(\mathtt{t}_1 \to \downarrow\mathtt{t}_2) \wedge (\mathtt{t}_3 \to \downarrow\mathtt{t}_4) \wedge ((\mathtt{t}_1 \wedge \mathtt{t}_3) \to \downarrow(\mathtt{t}_2 \wedge \mathtt{t}_4))]\!] \\ &\quad \subseteq \ [\![\downarrow((\mathtt{t}_1 \to \mathtt{t}_2) \wedge (\mathtt{t}_3 \to \mathtt{t}_4) \wedge ((\mathtt{t}_1 \wedge \mathtt{t}_3) \to (\mathtt{t}_2 \wedge \mathtt{t}_4)))]\!] \end{aligned}$$

□

Sketch of proof The idea of the proof is for any f in

$$[\![(\mathtt{t}_1 \to \downarrow\mathtt{t}_2) \wedge (\mathtt{t}_3 \to \downarrow\mathtt{t}_4) \wedge ((\mathtt{t}_1 \wedge \mathtt{t}_3) \to \downarrow(\mathtt{t}_2 \wedge \mathtt{t}_4))]\!]$$

to construct a function g such that

$$g \in [\![(\mathtt{t}_1 \to \mathtt{t}_2) \wedge (\mathtt{t}_3 \to \mathtt{t}_4) \wedge ((\mathtt{t}_1 \wedge \mathtt{t}_3) \to (\mathtt{t}_2 \wedge \mathtt{t}_4))]\!]$$

and

$$f \leq g$$

then we will have

$$f \in [\![\downarrow((\mathtt{t}_1 \to \mathtt{t}_2) \wedge (\mathtt{t}_3 \to \mathtt{t}_4) \wedge ((\mathtt{t}_1 \wedge \mathtt{t}_3) \to (\mathtt{t}_2 \wedge \mathtt{t}_4)))]\!]$$

as required.

One way to construct g is to define it as f but whenever $f\ x$ is too "small" to define $g\ x$ to be bigger. This will ensure $f \leq g$. But is g, defined in this way, a monotonic and continuous function?

In order to make the proof of g being a monotonic and continuous function a bit easier we will identify some of the members of the domains to be the "good" ones:

$$\begin{aligned} \mathrm{D}^{\mathrm{G}}_{\mathtt{Int}} &= \{\bot_{\mathtt{Int}}, 0\} \\ \mathrm{D}^{\mathrm{G}}_{\mathtt{Bool}} &= \{\bot_{\mathtt{Bool}}, \mathit{true}\} \\ \mathrm{D}^{\mathrm{G}}_{\mathtt{ut}_2 \to \mathtt{ut}_1} &= (\mathrm{D}_{\mathtt{ut}_2} \to_{\mathrm{cont}} \mathrm{D}^{\mathrm{G}}_{\mathtt{ut}_1})_\bot \end{aligned}$$

In this way all the members of a domain are laying on a chain. Now given any element of a domain we would like to approximate it with a good one. We define

$$
\begin{array}{rcl}
\mathcal{G}_{\mathtt{Int}}(\bot_{\mathtt{Int}}) & = & \bot_{\mathtt{Int}} \\
\mathcal{G}_{\mathtt{Int}}(x) & = & 0 \\
\mathcal{G}_{\mathtt{Bool}}(\bot_{\mathtt{Int}}) & = & \bot_{\mathtt{Bool}} \\
\mathcal{G}_{\mathtt{Bool}}(\mathit{true}) & = & \mathit{true} \\
\mathcal{G}_{\mathtt{Bool}}(\mathit{false}) & = & \mathit{true} \\
\mathcal{G}_{\mathtt{ut}_1\to\mathtt{ut}_2}(\bot_{\mathtt{ut}_1\to\mathtt{ut}_2}) & = & \bot_{\mathtt{ut}_1\to\mathtt{ut}_2} \\
\mathcal{G}_{\mathtt{ut}_1\to\mathtt{ut}_2}(f) & = & \lambda x.\mathcal{G}_{\mathtt{ut}_2}(fx)
\end{array}
$$

Now instead of testing if d belong to $[\![\mathtt{t}]\!]$ it suffices to test if $\mathcal{G}_{\varepsilon(\mathtt{t})}(d)$ belong to $[\![\mathtt{t}]\!]$:

Lemma 4.18
We have

$$\mathcal{G}_{\varepsilon(\mathtt{t})}(d) \in [\![\mathtt{t}]\!] \Leftrightarrow d \in [\![\mathtt{t}]\!]$$

□

Proof The lemma is easily shown by induction on the strictness and totality type $\mathtt{t}$. ■

As a consequence of Lemma 4.18 we can assume that f is an good element. We define g to be:

$$
g \;=\; \lambda x. \begin{cases}
\sqcup\{a \mid a \in [\![\mathtt{t}_2 \wedge \mathtt{t}_4]\!], a \in \mathrm{D}^{\mathrm{G}}_{\varepsilon(\mathtt{t}_2)}, fx \leq a\}, & \text{if } x \in [\![\mathtt{t}_1 \wedge \mathtt{t}_3]\!] \\
\sqcup\{b \mid b \in [\![\mathtt{t}_2]\!], b \in \mathrm{D}^{\mathrm{G}}_{\varepsilon(\mathtt{t}_2)}, fx \leq b\}, & \text{if } x \in [\![\mathtt{t}_1]\!] \backslash [\![\mathtt{t}_3]\!] \\
\sqcup\{c \mid c \in [\![\mathtt{t}_4]\!], c \in \mathrm{D}^{\mathrm{G}}_{\varepsilon(\mathtt{t}_4)}, fx \leq c\} & \text{if } x \in [\![\mathtt{t}_3]\!] \backslash [\![\mathtt{t}_1]\!]
\end{cases}
$$

Now we must ensure that g satisfies

- g is a monotonic and continuous function
- $g \in [\![\downarrow((\mathtt{t}_1 \to \mathtt{t}_2) \wedge (\mathtt{t}_3 \to \mathtt{t}_4) \wedge ((\mathtt{t}_1 \wedge \mathtt{t}_3) \to (\mathtt{t}_2 \wedge \mathtt{t}_4)))]\!]$
- $f \leq g$

Firstly we have to ensure that the least upper bounds exists:

Conjecture 4.19
Let d_1 and d_2 be members of $\mathrm{D}^{\mathrm{G}}_{\mathtt{ut}}$, then

$$d_1 \sqcup d_2 \in \mathrm{D}^{\mathrm{G}}_{\mathtt{ut}}$$

□

In order to show that g is monotonic we must consider all possible ways of choosing x_1 and x_2 in the three clauses of the definition of g. For example, assume

$$x_1 \leq x_2$$

and

$$\begin{array}{rcl} x_1 & \in & [\![\mathtt{t}_1 \wedge \mathtt{t}_3]\!] \\ x_2 & \in & [\![\mathtt{t}_1]\!] \backslash [\![\mathtt{t}_3]\!] \end{array}$$

we must show

$$\begin{array}{c} \sqcup\{a \mid a \in [\![\mathtt{t}_2 \wedge \mathtt{t}_4]\!], a \in \mathrm{D}^{\mathrm{G}}_{\varepsilon(\mathtt{t}_2)}, fx \leq a\} \\ \leq \{b \mid b \in [\![\mathtt{t}_2]\!], b \in \mathrm{D}^{\mathrm{G}}_{\varepsilon(\mathtt{t}_2)}, fx \leq b\} \end{array}$$

We will do it by showing

$$\begin{array}{c} \sqcup\{a \mid a \in [\![\mathtt{t}_2 \wedge \mathtt{t}_4]\!], a \in \mathrm{D}^{\mathrm{G}}_{\varepsilon(\mathtt{t}_2)}, fx \leq a\} \\ \leq \sqcup\{a \mid a \in [\![\mathtt{t}_2]\!], a \in \mathrm{D}^{\mathrm{G}}_{\varepsilon(\mathtt{t}_2)}, fx \leq a\} \end{array}$$

and

$$\begin{array}{c} \sqcup\{a \mid a \in [\![\mathtt{t}_2]\!], a \in \mathrm{D}^{\mathrm{G}}_{\varepsilon(\mathtt{t}_2)}, fx \leq a\} \\ \leq \ \{b \mid b \in [\![\mathtt{t}_2]\!], b \in \mathrm{D}^{\mathrm{G}}_{\varepsilon(\mathtt{t}_2)}, fx \leq b\} \end{array}$$

The first one is obvious since

$$\begin{array}{c} \{a \mid a \in [\![\mathtt{t}_2 \wedge \mathtt{t}_4]\!], a \in \mathrm{D}^{\mathrm{G}}_{\varepsilon(\mathtt{t}_2)}, fx \leq a\} \\ \subseteq \{a \mid a \in [\![\mathtt{t}_2]\!], a \in \mathrm{D}^{\mathrm{G}}_{\varepsilon(\mathtt{t}_2)}, fx \leq a\} \end{array}$$

We show the second one by constructing an element

$$b = a \sqcup fx_2$$

We know that $f\ x_2 \in [\![\downarrow\mathtt{t}_2]\!]$ so we can apply

Conjecture 4.20
Let d_1 and d_2 be members of $\mathrm{D}^{\mathrm{G}}_{\mathtt{ut}}$ and $d_1 \in [\![\mathtt{t}]\!]$ and $d_2 \in [\![\downarrow\mathtt{t}]\!]$, then

$$d_1 \sqcup d_2 \in [\![\mathtt{t}]\!]$$

$\square$

to get the desired result. Etcetera! $\blacksquare$

The next step in getting completeness is to extend the construction of $\downarrow(\mathtt{t}_1 \wedge \mathtt{t}_2)$ to allow $\mathtt{t}_1$ and $\mathtt{t}_2$ to be of different structure.

4.4 Summary

We have constructed an algorithm for inferring the strictness and totality types by following the lazy types approach of [HM94a]. The algorithm is sound with respect to the strictness and totality analysis but not complete.

An implementation in Miranda of the algorithm is presented in the Appendix page 347.

The type checking algorithm does indeed terminate: in all clauses a finite set of recursive calls on either a sub-term or a subtype is done. Since there is a finite number of strictness and totality types with a given underlying type, then algorithm algorithm $\mathcal{ALL}$ terminates. Hence the list given to $\mathcal{CHAIN}$ is finite, and since that algorithm just steps trough the list, it will terminate. Finally the algorithm $\mathcal{I}$ will terminate since all the recursive calls are on a subtypes and the number of recursive calls is finite. Again the algorithm $\mathcal{C}$ steps trough the list provided by $\mathcal{ALL}$.

Chapter 5

Binding Time Analysis

We consider the problem of introducing a distinction between binding times (e.g. compile-time and run-time) into functional languages. It is well-known that such a distinction is important for the efficient implementation of imperative languages [ASU86] and more recent results show that the performance of functional languages may be improved by using binding time information (e.g. [NN89, Jør92]).

There are several approaches to the specification of binding time analysis. Some approaches are based on variants of abstract interpretation (e.g. [Bon90, Con90, HS91]), others are based on projection analysis (e.g. [Lau91]) and yet others (e.g. [NN92, NN88, HM94b]) use non-standard type systems and develop corresponding type inference algorithms. In this Chapter we shall take a logical approach and aim at constructing an algorithm for generating a set of constraints to be solved. In this way we will be able to make full use of substitutions as in ordinary type inference [Mil78] — this is contrary to other algorithms (e.g. [NN92]) where extra recursive calls have to be performed.

Overview The starting point for our work is the inference system for binding times of the simply typed λ-calculus as specified in [NN92]. This is reviewed in Section 5.1. However, we shall reformulate it in a style motivated by the inference systems in Chapter 1. We will annotate both the base-types and the type constructors with the annotations:

$$\begin{aligned} s_1 &::= \mathbf{r} \mid \mathbf{c} \mid b \\ s_4 &::= \mathbf{r} \mid \mathbf{c} \mid b \end{aligned}$$

where b is a binding time variable. Furthermore we will let constraints between the binding times be part of the inference system. This is described in Section 5.2. We construct an algorithm for binding time analysis from this inference system. This algorithm is $\mathcal{O}(n^4)$ where n is the size of the given term where the algorithm of [NN92] is exponential in the size of the term. We proceed in a couple of stages. First we get rid of the two rules [up] and [down] to get a simpler inference system. This is done in Section 5.3. In Section 5.4 we present an algorithm for finding the constraints that has to be fulfilled in order to turn a 1-level term into a term in the 2-level λ-calculus and in Section 5.5 we solve the constraints.

Note that this Chapter differs from the preceeding chapters, that the analysis described here is not a new analysis but the purpose of this work is to construct a more efficient algorithm than the one constructed in [NN92].

5.1 Review of Binding Time Analysis

In this section we review the binding time analysis of Nielson and Nielson [NN92].

In a 2-level λ-calculus the binding times are explicitly marked on each construction. For us a type, $\mathtt{t} \in \mathrm{T2}$, of the 2-level λ-calculus is either a base type or a function type:

$$\mathtt{t} ::= \mathtt{B} \mid \underline{\mathtt{B}} \mid \mathtt{t} \rightarrow \mathtt{t} \mid \mathtt{t}\underline{\rightarrow}\mathtt{t}$$

where the $\mathtt{B}$ are the base types including `Int` and `Bool`. The underlined constructions are those of run-time kind and the non-underlined are those of compile-time kind. A term of compile-time kind can be evaluated at compile-time, whereas a term of run-time kind must be evaluated at run-time.

Alternatively we can present the annotated types as:

$$\begin{aligned} \mathtt{t} &::= \mathtt{B}^s \mid \mathtt{t} \rightarrow^s \mathtt{t} \\ s &::= \mathbf{r} \mid \mathbf{c} \end{aligned}$$

which is more in the line of the work here. The **r**-annotations correspond to underlining and the **c**-annotations correspond to no underlining. The

$$[\underline{\mathtt{B}}]\ \frac{}{\vdash_0 \underline{\mathtt{B}} : \mathbf{r}} \qquad\qquad [\mathtt{B}]\ \frac{}{\vdash_0 \mathtt{B} : \mathbf{c}}$$

$$[\underline{\rightarrow}]\ \frac{\vdash_0 \mathtt{t}_1 : \mathbf{r} \quad \vdash_0 \mathtt{t}_2 : \mathbf{r}}{\vdash_0 \mathtt{t}_1 \underline{\rightarrow} \mathtt{t}_2 : \mathbf{r}} \qquad\qquad [\rightarrow]\ \frac{\vdash_0 \mathtt{t}_1 : \mathbf{c} \quad \vdash_0 \mathtt{t}_2 : \mathbf{c}}{\vdash_0 \mathtt{t}_1 \rightarrow \mathtt{t}_2 : \mathbf{c}}$$

$$[\mathrm{up}]\ \frac{\vdash_0 \mathtt{t}_1 \underline{\rightarrow} \mathtt{t}_2 : \mathbf{r}}{\vdash_0 \mathtt{t}_1 \underline{\rightarrow} \mathtt{t}_2 : \mathbf{c}}$$

Figure 5.1: Well-formedness of the 2-level Types

2-level terms, $\mathtt{e} \in$ E2, are

$$\begin{aligned} \mathtt{e} \ ::= \ & \mathtt{x} \mid \lambda\mathtt{x.e} \mid \underline{\lambda}\mathtt{x.e} \mid \mathtt{e}\ (\mathtt{e}) \mid \mathtt{e}\ \underline{(}\mathtt{e}\underline{)} \mid \\ & \mathtt{if\ e\ then\ e\ else\ e} \mid \underline{\mathtt{if}}\ \mathtt{e}\ \underline{\mathtt{then}}\ \mathtt{e}\ \underline{\mathtt{else}}\ \mathtt{e} \mid \\ & \mathtt{fix\ e} \mid \underline{\mathtt{fix}}\ \mathtt{e} \end{aligned}$$

Again we have an alternative presentation of the terms:

$$\begin{aligned} \mathtt{e} \ ::= \ & \mathtt{x} \mid \lambda^s\mathtt{x.e} \mid \mathtt{e}\ (\mathtt{e})^s \mid \\ & \mathtt{if}^s\ \mathtt{e\ then\ e\ else\ e} \mid \mathtt{fix}^s\ \mathtt{e} \\ s \ ::= \ & \mathbf{r} \mid \mathbf{c} \end{aligned}$$

Notice that there is only one sort of variable, $\mathtt{x}$. The overall binding time of a variable is determined by the λ-binding of it.

5.1.1 Well-formedness of Types

We first introduce rules for annotating types. First we say that a type $\mathtt{t}$ is well-formed of binding time $\mathtt{b}$ where $\mathtt{b}$ is either $\mathbf{r}$ or $\mathbf{c}$, if $\vdash_0 \mathtt{t} : \mathtt{b}$. This well-formedness relation is given in Figure 5.1. A run-time function type can be thought of as a piece of code. The compiler, which generates code, can manipulate this piece of code. Therefore a run-time function type can be both of run-time kind and compile-time kind. This fact is expressed by the rule [up], which allows us to turn a run-time function type of kind run-time into a run-time function type of kind compile-time. Only the [up] rule allows us to transform a run-time type into a compile-time type and furthermore this is only possible for function types.

Example 5.1
An example of using Figure 5.1 is to show that the type

$$((\underline{\mathtt{B} \rightarrow \mathtt{B}}) \underline{\rightarrow} (\underline{\mathtt{B} \rightarrow \mathtt{B}})) \rightarrow (\underline{\mathtt{B} \rightarrow \mathtt{B}}) \rightarrow (\underline{\mathtt{B} \rightarrow \mathtt{B}})$$

is well-formed of compile-time kind for some base type $\underline{\mathtt{B}}$. First we have that $\vdash_0 \underline{\mathtt{B}} : \mathbf{r}$ from $[\underline{\mathtt{B}}]$. From $[\underline{\rightarrow}]$ we get

$$\frac{\vdash_0 \underline{\mathtt{B}} : \mathbf{r} \quad \vdash_0 \underline{\mathtt{B}} : \mathbf{r}}{\vdash_0 \underline{\mathtt{B} \rightarrow \mathtt{B}} : \mathbf{r}} \tag{5.1}$$

Applying the rule $[\underline{\rightarrow}]$ to two copies of (5.1) we get

$$\vdash_0 (\underline{\mathtt{B} \rightarrow \mathtt{B}}) \underline{\rightarrow} (\underline{\mathtt{B} \rightarrow \mathtt{B}}) : \mathbf{r}$$

Now we apply the rule [up] to get the binding time $\mathbf{c}$

$$\vdash_0 (\underline{\mathtt{B} \rightarrow \mathtt{B}}) \underline{\rightarrow} (\underline{\mathtt{B} \rightarrow \mathtt{B}}) : \mathbf{c} \tag{5.2}$$

From (5.1) we get by applying [up]

$$\vdash_0 \underline{\mathtt{B} \rightarrow \mathtt{B}} : \mathbf{c} \tag{5.3}$$

Now we can apply the rule $[\rightarrow]$ to two copies of (5.3) to get

$$\vdash_0 (\underline{\mathtt{B} \rightarrow \mathtt{B}}) \rightarrow (\underline{\mathtt{B} \rightarrow \mathtt{B}}) : \mathbf{c} \tag{5.4}$$

The result is now obtained by using $[\rightarrow]$ to combine (5.2) and (5.4)

$$\vdash_0 ((\underline{\mathtt{B} \rightarrow \mathtt{B}}) \underline{\rightarrow} (\underline{\mathtt{B} \rightarrow \mathtt{B}})) \rightarrow (\underline{\mathtt{B} \rightarrow \mathtt{B}}) \rightarrow (\underline{\mathtt{B} \rightarrow \mathtt{B}}) : \mathbf{c}$$

as desired. □

5.1.2 Well-formedness of Expressions

Next we say that the term `e` has type `t` and binding time `b` under the assumptions *tenv*[1] if

$$\mathit{tenv} \vdash_0 \mathtt{e} : \mathtt{t} : \mathtt{b}$$

[1] in [NN92] *tenv* is used to denote the assumption list, or as called in [NN92] the type environment

where the type environment, *tenv*, is a function from variables to 2-level types and binding times. That is

$$tenv\ \mathtt{x} = (\mathtt{t}, \mathtt{b})$$

where $\mathtt{t}$ is the type of the variable $\mathtt{x}$ and $\mathtt{b}$ is the binding time of $\mathtt{x}$. Given *tenv* then the function $tenv[(\mathtt{t},\ \mathtt{b})/\mathrm{x}]$ is defined by

$$(tenv[(\mathtt{t}, \mathtt{b})/\mathtt{x}])\ \mathtt{y} = \begin{cases} (\mathtt{t}, \mathtt{b}), & \text{if } \mathtt{x} = \mathtt{y} \\ tenv\ \mathtt{y}, & \text{otherwise} \end{cases}$$

The well-formedness relation for 2-level terms is given in Figure 5.2. Basically we have two copies of the traditional inference system for typing the λ-calculus, one for the run-time level and one for the compile-time level. Furthermore we have the two rules [up] and [down] allowing the two binding times to mix. The idea behind [up] is that in order to turn a compile-time term into a run-time term (i.e. to allow it to be evaluated at run-time) it has to express some computation, i.e. it must have a run-time function type. In order to turn a run-time term into a compile-time term (i.e. to talk about its evaluation at compile-time) its type must not only be a run-time function type, but the term is also not allowed to reference "free" run-time objects.

Example 5.2
As an example of using Figure 5.2 we show that the term

$$\lambda \mathtt{x}.\lambda \mathtt{y}.\mathtt{x}\ \underline{(}\mathtt{y}\underline{)}$$

has the type

$$((\underline{\mathtt{B} \rightarrow \mathtt{B}}) \underline{\rightarrow} (\underline{\mathtt{B} \rightarrow \mathtt{B}})) \rightarrow (\underline{\mathtt{B} \rightarrow \mathtt{B}}) \rightarrow (\underline{\mathtt{B} \rightarrow \mathtt{B}})$$

and is well-formed of compile-time kind for some base-type $\underline{\mathtt{B}}$.

Let *tenv* be given by

$$\begin{aligned} tenv\ \mathtt{x} &= ((\underline{\mathtt{B} \rightarrow \mathtt{B}}) \underline{\rightarrow} (\underline{\mathtt{B} \rightarrow \mathtt{B}}), \mathbf{c}) \\ tenv\ \mathtt{y} &= (\underline{\mathtt{B} \rightarrow \mathtt{B}}, \mathbf{c}) \\ tenv\ \mathtt{z} &= \text{undefined, if } \mathtt{z} \neq \mathtt{x} \text{ and } \mathtt{z} \neq \mathtt{y} \end{aligned}$$

$$[\text{var}]\ \frac{}{\mathit{tenv} \vdash_0 \mathtt{x} : \mathtt{t} : \mathtt{b}} \quad \text{if } \mathit{tenv}\ \mathtt{x} = (\mathtt{t}, \mathtt{b}) \wedge \vdash_0 \mathtt{t} : \mathtt{b}$$

$$[\underline{\text{abs}}]\ \frac{\mathit{tenv}[(\mathtt{t}_1, \mathbf{r})/\mathtt{x}] \vdash_0 \mathtt{e} : \mathtt{t}_2 : \mathbf{r}}{\mathit{tenv} \vdash_0 \underline{\lambda}\mathtt{x}.\mathtt{e} : \mathtt{t}_1 \underline{\rightarrow} \mathtt{t}_2 : \mathbf{r}} \quad \text{if } \vdash_0 \mathtt{t}_2 : \mathbf{r}$$

$$[\text{abs}]\ \frac{\mathit{tenv}[(\mathtt{t}_1, \mathbf{c})/\mathtt{x}] \vdash_0 \mathtt{e} : \mathtt{t}_2 : \mathbf{c}}{\mathit{tenv} \vdash_0 \lambda\mathtt{x}.\mathtt{e} : \mathtt{t}_1 \rightarrow \mathtt{t}_2 : \mathbf{c}} \quad \text{if } \vdash_0 \mathtt{t}_2 : \mathbf{c}$$

$$[\underline{\text{app}}]\ \frac{\mathit{tenv} \vdash_0 \mathtt{e}_1 : \mathtt{t}_1 \underline{\rightarrow} \mathtt{t}_2 : \mathbf{r} \quad \mathit{tenv} \vdash_0 \mathtt{e}_2 : \mathtt{t}_1 : \mathbf{r}}{\mathit{tenv} \vdash_0 \mathtt{e}_1\ \underline{(}\mathtt{e}_2\underline{)} : \mathtt{t}_2 : \mathbf{r}}$$

$$[\text{app}]\ \frac{\mathit{tenv} \vdash_0 \mathtt{e}_1 : \mathtt{t}_1 \rightarrow \mathtt{t}_2 : \mathbf{c} \quad \mathit{tenv} \vdash_0 \mathtt{e}_2 : \mathtt{t}_1 : \mathbf{c}}{\mathit{tenv} \vdash_0 \mathtt{e}_1\ (\mathtt{e}_2) : \mathtt{t}_2 : \mathbf{c}}$$

$$[\underline{\text{if}}]\ \frac{\mathit{tenv} \vdash_0 \mathtt{e}_1 : \underline{\mathtt{Bool}} : \mathbf{r} \quad \mathit{tenv} \vdash_0 \mathtt{e}_2 : \mathtt{t} : \mathbf{r} \quad \mathit{tenv} \vdash_0 \mathtt{e}_3 : \mathtt{t} : \mathbf{r}}{\mathit{tenv} \vdash_0 \underline{\mathtt{if}}\ \mathtt{e}_1\ \underline{\mathtt{then}}\ \mathtt{e}_2\ \underline{\mathtt{else}}\ \mathtt{e}_3 : \mathtt{t} : \mathbf{r}}$$

$$[\text{if}]\ \frac{\mathit{tenv} \vdash_0 \mathtt{e}_1 : \mathtt{Bool} : \mathbf{c} \quad \mathit{tenv} \vdash_0 \mathtt{e}_2 : \mathtt{t} : \mathbf{c} \quad \mathit{tenv} \vdash_0 \mathtt{e}_3 : \mathtt{t} : \mathbf{c}}{\mathit{tenv} \vdash_0 \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 : \mathtt{t} : \mathbf{c}}$$

$$[\underline{\text{fix}}]\ \frac{\mathit{tenv} \vdash_0 \mathtt{e} : \mathtt{t} \underline{\rightarrow} \mathtt{t} : \mathbf{r}}{\mathit{tenv} \vdash_0 \underline{\mathtt{fix}}\ \mathtt{e} : \mathtt{t} : \mathbf{r}}$$

$$[\text{fix}]\ \frac{\mathit{tenv} \vdash_0 \mathtt{e} : \mathtt{t} \rightarrow \mathtt{t} : \mathbf{c}}{\mathit{tenv} \vdash_0 \mathtt{fix}\ \mathtt{e} : \mathtt{t} : \mathbf{c}}$$

$$[\underline{\text{const}}]\ \frac{}{\mathit{tenv} \vdash_0 \underline{\mathtt{c}} : \mathtt{t} : \mathbf{r}}$$

$$[\text{const}]\ \frac{}{\mathit{tenv} \vdash_0 \mathtt{c} : \mathtt{t} : \mathbf{c}}$$

$$[\text{down}]\ \frac{\mathit{tenv} \vdash_0 \mathtt{e} : \mathtt{t} : \mathbf{c}}{\mathit{tenv} \vdash_0 \mathtt{e} : \mathtt{t} : \mathbf{r}} \quad \text{if } \vdash_0 \mathtt{t} : \mathbf{r}$$

$$[\text{up}]\ \frac{\mathit{tenv} \vdash_0 \mathtt{e} : \mathtt{t} : \mathbf{r}}{\mathit{tenv} \vdash_0 \mathtt{e} : \mathtt{t} : \mathbf{c}} \quad \text{if } \vdash_0 \mathtt{t} : \mathbf{c} \wedge \forall \mathtt{x} \in \mathit{tenv} : \mathtt{x} = (\mathtt{t}, \mathbf{c})$$

Figure 5.2: Well-formedness of the 2-level λ-calculus

From [var] we get

$$tenv \vdash_0 \mathtt{x} : ((\underline{\mathtt{B} \rightarrow \mathtt{B}}) \underline{\rightarrow} (\underline{\mathtt{B} \rightarrow \mathtt{B}})) : \mathbf{c}$$

and

$$tenv \vdash_0 \mathtt{y} : \underline{\mathtt{B} \rightarrow \mathtt{B}} : \mathbf{c}$$

Applying [down] on both of them gives

$$tenv \vdash_0 \mathtt{x} : ((\underline{\mathtt{B} \rightarrow \mathtt{B}}) \underline{\rightarrow} (\underline{\mathtt{B} \rightarrow \mathtt{B}})) : \mathbf{r}$$

and

$$tenv \vdash_0 \mathtt{y} : \underline{\mathtt{B} \rightarrow \mathtt{B}} : \mathbf{r}$$

Now we can apply $[\underline{()}]$ to get

$$tenv \vdash_0 \mathtt{x}\ \underline{(}\mathtt{y}\underline{)} : \underline{\mathtt{B} \rightarrow \mathtt{B}} : \mathbf{r}$$

Since *tenv* only contains variables of compile-time kind and the type of $\mathtt{x}\ \underline{(}\mathtt{y}\underline{)}$ is a run-time function type of run-time kind it is possible to apply [up] to obtain

$$tenv \vdash_0 \mathtt{x}\ \underline{(}\mathtt{y}\underline{)} : \underline{\mathtt{B} \rightarrow \mathtt{B}} : \mathbf{c}$$

After applying [λ] two times we obtain the desired result

$$\frac{tenv' \vdash_0 \lambda\mathtt{y}.\mathtt{x}\ \underline{(}\mathtt{y}\underline{)} : (\underline{\mathtt{B} \rightarrow \mathtt{B}}) \rightarrow (\underline{\mathtt{B} \rightarrow \mathtt{B}}) : \mathbf{c}}{tenv'' \vdash_0 \lambda\mathtt{x}.\lambda\mathtt{y}.\mathtt{x}\ \underline{(}\mathtt{y}\underline{)} : \mathtt{t} : \mathbf{c}}$$

where

$$\mathtt{t} \;=\; ((\underline{\mathtt{B} \rightarrow \mathtt{B}}) \underline{\rightarrow} (\underline{\mathtt{B} \rightarrow \mathtt{B}})) \rightarrow (\underline{\mathtt{B} \rightarrow \mathtt{B}}) \rightarrow (\underline{\mathtt{B} \rightarrow \mathtt{B}})$$

and $tenv'$ and $tenv''$ are given by

$$\begin{aligned} tenv'\ \mathtt{x} &= (((\underline{\mathtt{B} \rightarrow \mathtt{B}}) \underline{\rightarrow} (\underline{\mathtt{B} \rightarrow \mathtt{B}})), \mathbf{c}) \\ tenv'\ \mathtt{z} &= \text{undefined, if } \mathtt{z} \neq \mathtt{x} \end{aligned}$$

and

$$tenv''\ \mathtt{z} \;=\; \text{undefined for all variables}$$

$\square$

This inference system is part of the one used in [NN92] to construct a binding time analysis.

5.1.3 Algorithms for Binding Time Analysis

In [NN92] the algorithm for binding time analysis is in two parts, one for types and one for terms.

The algorithm $\mathcal{T}_{\mathrm{BTA}}$ for binding time analysis of types presented in [NN92] calculates an annotated type $\mathtt{t}$ and its overall binding time $\mathtt{b}$ ($\mathbf{r}$ or $\mathbf{c}$) given a type $\mathtt{t}'$ and the overall binding time $\mathtt{b}'$ of the type. The calculated type is the type with as few underlined constructions as possible and it is well-formed of kind $\mathtt{b}$ (i.e. $\vdash_0 \mathtt{t} : \mathtt{b}$ can be inferred from Figure 5.1). This annotation expresses that as many computations as possible are performed at compile-time.

The purpose of this Chapter is to construct a new and more efficient algorithm for binding time analysis. We do this four in steps:

- First we reformulate the analysis in Figure 5.2 to allow binding time variables, wherefore we will introduce constraints between the binding times.
- Next we will transform the new inference system to be syntax directed.
- Thirdly we construct an algorithm for computing the constraints, that has to be satisfied.
- Finally we solve the constraints.

The algorithm $\mathcal{E}_{\mathrm{BTA}}$ for binding time analysis of terms presented in [NN92] calculates an annotated term $\mathtt{e}$, its type $\mathtt{t}$, and its overall binding time $\mathtt{b}$ given a term $\mathtt{e}'$, a type $\mathtt{t}'$ and an overall binding time $\mathtt{b}'$. The annotated term is the term with as few underlined constructions as possible and it is well-formed of type $\mathtt{t}$ and binding time $\mathtt{b}$ (i.e. $\vdash_0 \mathtt{e} : \mathtt{t} : \mathtt{b}$ can be inferred from Figure 5.2).

5.2 A Constraint based Binding Time Analysis

We are now going to use the alternative way of writing types and term and then to construct constraints between the annotations. In this way we can write the rule for e.g. $[\underline{\text{abs}}]$ and [abs] as one rule. The new system we get in this section corresponds in a one-to-one manner to the analysis of Section 5.2.

5.2.1 Types and Their Well-formedness

We will allow the annotations to include binding time variables here — the type system is now:

$$
\begin{array}{rcl}
\mathtt{t} & ::= & \mathtt{B}^s \mid \mathtt{t} \rightarrow^s \mathtt{t} \\
s & ::= & \mathbf{r} \mid \mathbf{c} \mid b
\end{array}
$$

where b is a binding time variable.

The types still has to be well-formed. We do this by means of constraints on the values a binding time variable can take. The constraints are a list of inequalities between binding times of the forms

$$
\begin{array}{rcl}
p & = & b \\
p & < & b \\
p & \leq & b
\end{array}
$$

later we shall also allow constraints of other forms. The constraints can be solved if there exists a mapping from all the binding time variables to $\{\mathbf{r}, \mathbf{c}\}$ such that all the inequalities are satisfied. From this follows that the constraint set is unsolvable if its transitive closure contains inequalities of the forms

$$
\begin{array}{rcl}
\mathbf{c} & \leq & \mathbf{r} \\
\mathbf{c} & = & \mathbf{r} \\
\mathbf{r} & = & \mathbf{c} \\
\mathbf{r} & < & \mathbf{r} \\
\mathbf{c} & < & \mathbf{c} \\
\mathbf{c} & < & \mathbf{r}
\end{array}
$$

The function $\mathcal{W}$, defined in Figure 5.3, is used to determine constraints so that the type $\mathtt{t}$ with overall binding time p is well-formed. A base type $\mathtt{B}^b$ is well-formed of kind p if $b = p$. A function type, $\mathtt{t}_1 \rightarrow^b \mathtt{t}_2$, is well-formed of kind p provided $\mathtt{t}_1$ and $\mathtt{t}_2$ are well-formed of kind b and $b = p$

$$
\begin{array}{lcl}
\mathcal{W}(\mathtt{B}^b,\, p) & = & [b = p] \\
\mathcal{W}(\mathtt{t}_1 \rightarrow^b \mathtt{t}_2,\, p) & = & [\mathcal{W}(\mathtt{t}_1,\, b),\, \mathcal{W}(\mathtt{t}_2,\, b),\, b \leq p]
\end{array}
$$

Figure 5.3: Constraints for Well-formedness for Types

furthermore a run-time function type can be of both run-time kind and compile-time kind, hence the constraint $b \leq p$. The relation between the function $\mathcal{W}$ in Figure 5.3 and the well-formedness relation for types $\vdash_0$ in Figure 5.1 is given by Lemma 5.4 and 5.3:

Lemma 5.3 *Soundness of $\mathcal{W}$*
If $\mathcal{W}(\mathtt{t},\, b)$ is solvable by M, then $\vdash_0 M\mathtt{t} \,:\, Mb$ can be derived. □

Proof We assume that $\mathcal{W}(\mathtt{t},\, b)$ is solvable by the mapping M and we prove that $\vdash_0 M\mathtt{t} \,:\, Mb$ can be inferred by induction on the type $\mathtt{t}$.

For the details see Appendix page 373 ■

Lemma 5.4 *Completeness of $\mathcal{W}$*
If $\vdash_0 \mathtt{t} \,:\, b$, then $\mathcal{W}(\mathtt{t},\, b)$ is solvable. □

Proof We assume that

$$\vdash_0 \mathtt{t} \,:\, b$$

can be inferred and we prove that $\mathcal{W}(\mathtt{t},\, b)$ is solvable by induction on the shape of the proof for $\vdash_0 \mathtt{t} \,:\, b$. More precisely we show that all mappings from binding time variables to $\{\mathtt{r},\, \mathtt{c}\}$ will satisfy the constraints of $\mathcal{W}(\mathtt{t},\, b)$.

For the details see Appendix page 375. ■

Example 5.5
We will in this example see how one calculation of $\mathcal{W}$ captures all the well-formed annotations of a type that can be constructed with several proof-trees using the well-formedness relation defined in Figure 5.1. We will find the constraints for

$$
\begin{array}{l}
\mathtt{t} = ((\mathtt{B}^{b_4} \rightarrow^{b_3} \mathtt{B}^{b_5}) \rightarrow^{b_2} (\mathtt{B}^{b_6} \rightarrow^{b_8} \mathtt{B}^{b_7})) \rightarrow^{b_1} \\
\qquad (\mathtt{B}^{b_{11}} \rightarrow^{b_{10}} \mathtt{B}^{b_{12}}) \rightarrow^{b_9} (\mathtt{B}^{b_{14}} \rightarrow^{b_{13}} \mathtt{B}^{b_{15}})
\end{array}
$$

to be well-formed of binding time p. We calculate

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathcal{W}(\mathtt{t},\, p)} \\
= & [\mathcal{W}((\mathtt{B}^{b_4} \rightarrow^{b_3} \mathtt{B}^{b_5}) \rightarrow^{b_2} (\mathtt{B}^{b_6} \rightarrow^{b_8} \mathtt{B}^{b_7}),\, b_1), \\
& \mathcal{W}((\mathtt{B}^{b_{11}} \rightarrow^{b_{10}} \mathtt{B}^{b_{12}}) \rightarrow^{b_9} (\mathtt{B}^{b_{14}} \rightarrow^{b_{13}} \mathtt{B}^{b_{15}}),\, b_1), \\
& b_1 \leq p] \\
= & [\mathcal{W}(\mathtt{B}^{b_4} \rightarrow^{b_3} \mathtt{B}^{b_5},\, b_2), \mathcal{W}(\mathtt{B}^{b_6} \rightarrow^{b_8} \mathtt{B}^{b_7},\, b_2), b_2 \leq b_1, \\
& \mathcal{W}(\mathtt{B}^{b_{11}} \rightarrow^{b_{10}} \mathtt{B}^{b_{12}},\, b_9), \mathcal{W}(\mathtt{B}^{b_{14}} \rightarrow^{b_{13}} \mathtt{B}^{b_{15}},\, b_9), b_9 \leq b_1, \\
& b_1 \leq p] \\
= & [\mathcal{W}(\mathtt{B}^{b_4},\, b_3), \mathcal{W}(\mathtt{B}^{b_5},\, b_3), b_3 \leq b_2, \\
& \mathcal{W}(\mathtt{B}^{b_6},\, b_8), \mathcal{W}(\mathtt{B}^{b_7},\, b_8), b_8 \leq b_2, b_2 \leq b_1, \\
& \mathcal{W}(\mathtt{B}^{b_{11}},\, b_{10}), \mathcal{W}(\mathtt{B}^{b_{12}},\, b_{10}), b_{10} \leq b_9, \\
& \mathcal{W}(\mathtt{B}^{b_{14}},\, b_{13}), \mathcal{W}(\mathtt{B}^{b_{15}},\, b_{13}), b_{13} \leq b_9, b_9 \leq b_1, \\
& b_1 \leq p] \\
= & [b_4 = b_3, b_5 = b_3, b_3 \leq b_2, \\
& b_6 = b_8, b_7 = b_8, b_8 \leq b_2, b_2 \leq b_1, \\
& b_{11} = b_{10}, b_{12} = b_{10}, b_{10} \leq b_9, \\
& b_{14} = b_{13}, b_{15} = b_{13}, b_{13} \leq b_9, b_9 \leq b_1, \\
& b_1 \leq p] \\
= & [b_3 = b_4 = b_5, b_3 \leq b_2, \\
& b_6 = b_7 = b_8, b_8 \leq b_2, b_2 \leq b_1, \\
& b_{10} = b_{11} = b_{12}, b_{10} \leq b_9, \\
& b_{13} = b_{14} = b_{15}, b_{13} \leq b_9, b_9 \leq b_1, \\
& b_1 \leq p]
\end{array}
$$

This means that the type, $\mathtt{t}$, must have the form

$$
\begin{array}{l}
((\mathtt{B}^{b_3} \rightarrow^{b_3} \mathtt{B}^{b_3}) \rightarrow^{b_2} (\mathtt{B}^{b_6} \rightarrow^{b_6} \mathtt{B}^{b_6})) \rightarrow^{b_1} \\
\quad (\mathtt{B}^{b_{10}} \rightarrow^{b_{10}} \mathtt{B}^{b_{10}}) \rightarrow^{b_9} (\mathtt{B}^{b_{13}} \rightarrow^{b_{13}} \mathtt{B}^{b_{13}})
\end{array}
$$

with binding time p and constraints

$$[b_3 \leq b_2, b_8 \leq b_2, b_2 \leq b_1, b_{10} \leq b_9, b_{13} \leq b_9, b_9 \leq b_1, b_1 \leq p]$$

b_1	b_2	b_3	b_6	b_9	b_{10}	b_{13}	p
r	**r**	**r**	**r**	**r**	**r**	**r**	**r**
r	**r**	**r**	**r**	**r**	**r**	**r**	**c**
c	**r**	**r**	**r**	**r**	**r**	**r**	**c**
c	**r**	**r**	**r**	**c**	**r**	**r**	**c**
c	**r**	**r**	**r**	**c**	**r**	**c**	**c**
c	**r**	**r**	**r**	**c**	**c**	**r**	**c**
c	**r**	**r**	**r**	**c**	**c**	**c**	**c**
c	**c**	**r**	**r**	**r**	**r**	**r**	**c**
c	**c**	**r**	**r**	**c**	**r**	**r**	**c**
c	**c**	**r**	**r**	**c**	**r**	**c**	**c**
c	**c**	**r**	**r**	**c**	**c**	**r**	**c**
c	**c**	**r**	**r**	**c**	**c**	**c**	**c**
c	**c**	**r**	**c**	**r**	**r**	**r**	**c**
c	**c**	**r**	**c**	**c**	**r**	**r**	**c**

b_1	b_2	b_3	b_6	b_9	b_{10}	b_{13}	p
c	**c**	**r**	**c**	**c**	**r**	**c**	**c**
c	**c**	**r**	**c**	**c**	**c**	**r**	**c**
c	**c**	**r**	**c**	**c**	**c**	**c**	**c**
c	**c**	**c**	**r**	**r**	**r**	**r**	**c**
c	**c**	**c**	**r**	**c**	**r**	**r**	**c**
c	**c**	**c**	**r**	**c**	**r**	**c**	**c**
c	**c**	**c**	**r**	**c**	**c**	**r**	**c**
c	**c**	**c**	**r**	**c**	**c**	**c**	**c**
c	**c**	**c**	**c**	**r**	**r**	**r**	**c**
c	**c**	**c**	**c**	**c**	**r**	**r**	**c**
c	**c**	**c**	**c**	**c**	**r**	**c**	**c**
c	**c**	**c**	**c**	**c**	**c**	**r**	**c**
c	**c**	**c**	**c**	**c**	**c**	**c**	**c**

Table 5.1: Solutions to the Constraints in Example 5.5

All the solutions to the constraints are listed in Table 5.1. Notice that the constraints have more than one solution. This means that if we want to find all the well-formed annotations of a given type, then we first assign binding time variables to the type. Now we can find the constraints using $\mathcal{W}$. That we have found all the possible annotations of the type follows from Lemma 5.3 and 5.4. $\square$

5.2.2 Expressions and Their Well-formedness

An assumption list has the form

$$\mathtt{x}_1 : \mathtt{t}_1 : b_1, \ldots, \mathtt{x}_n : \mathtt{t}_n : b_n$$

where the $\mathtt{x}_i$'s are variables, the $\mathtt{t}_i$'s are the type of the i'th variable, and the b_i's are the binding time of the i'th variable. We shall *assume* throughout that all the $\mathtt{x}_i$'s are distinct.

When the assumption list is written in the form A:I, then I is the list of all the binding times of all the variables, and A is the list of all variables

$$[\text{var}]\ \frac{}{\text{A:I} \vdash_1 \mathtt{x} : \mathtt{t} : b\ [\mathcal{W}(\mathtt{t},\ b)]} \quad \text{if } \mathtt{x} : \mathtt{t} : b \in \text{A:I}$$

$$[\text{abs}]\ \frac{\text{A:I},\ \mathtt{x} : \mathtt{t}_1 : b_1 \vdash_1 \mathtt{e} : \mathtt{t}_2 : b_2\ [\text{C}]}{\text{A:I} \vdash_1 \lambda^{b_1}\mathtt{x.e} : \mathtt{t}_1 \to^{b_1} \mathtt{t}_2 : b_1\ [\text{C},\ b_1 = b_2,\ \mathcal{W}(\mathtt{t}_1,\ b_1)]}$$

$$[\text{app}]\ \frac{\text{A:I} \vdash_1 \mathtt{e}_1 : \mathtt{t}_1 \to^{b} \mathtt{t}_2 : b_1\ [\text{C}] \quad \text{A:I} \vdash_1 \mathtt{e}_2 : \mathtt{t}_1 : b_2\ [\text{D}]}{\text{A:I} \vdash_1 \mathtt{e}_1\ (\mathtt{e}_2)^{b} : \mathtt{t}_2 : b\ [\text{C},\ \text{D},\ b = b_1 = b_2]}$$

$$[\text{if}]\ \frac{\begin{array}{c}\text{A:I} \vdash_1 \mathtt{e}_1 : \mathtt{Bool}^{b} : b_1\ [\text{C}] \\ \text{A:I} \vdash_1 \mathtt{e}_2 : \mathtt{t} : b_2\ [\text{D}] \\ \text{A:I} \vdash_1 \mathtt{e}_3 : \mathtt{t} : b_3\ [\text{E}]\end{array}}{\text{A:I} \vdash_1 \mathtt{if}^{b}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 : \mathtt{t} : b \left[\begin{array}{l}\text{C, D, E,} \\ b = b_1 = b_2 = b_3\end{array}\right]}$$

$$[\text{const}]\ \frac{}{\text{A:I} \vdash_1 \mathtt{c}^{b} : \mathtt{t}_{\mathtt{c}^{b}} : b\ [\ ]}$$

$$[\text{fix}]\ \frac{\text{A:I} \vdash_1 \mathtt{e} : \mathtt{t} \to^{b} \mathtt{t} : b_1\ [\text{C}]}{\text{A:I} \vdash_1 \mathtt{fix}^{b}\ \mathtt{e} : \mathtt{t} : b\ [\text{C},\ b = b_1]}$$

$$[\text{down}]\ \frac{\text{A:I} \vdash_1 \mathtt{e} : \mathtt{t} : b_1\ [\text{C}]}{\text{A:I} \vdash_1 \mathtt{e} : \mathtt{t} : b_2\ [\text{C},\ \mathcal{D}(\mathtt{t},\ b_1,\ b_2)]}$$

$$[\text{up}]\ \frac{\text{A:I} \vdash_1 \mathtt{e} : \mathtt{t} : b_1\ [\text{C}]}{\text{A:I} \vdash_1 \mathtt{e} : \mathtt{t} : b_2\ [\text{C},\ \mathcal{U}(\mathtt{t},\ b_1,\ \text{I},\ b_2)]}$$

$$\begin{array}{ll}\mathcal{D}(\mathtt{B}^{b},\ b_1,\ b_2) & = [\mathbf{r} = \mathbf{c}] \\ \mathcal{D}(\mathtt{t}_1 \to^{b} \mathtt{t}_2,\ b_1,\ b_2) & = [b_2 < b_1,\ b \leq b_2] \\ & \\ \mathcal{U}(\mathtt{B}^{b},\ b_1,\ \text{I},\ b_2) & = [\mathbf{r} = \mathbf{c}] \\ \mathcal{U}(\mathtt{t}_1 \to^{b} \mathtt{t}_2,\ b_1,\ \text{I},\ b_2) & = [b_1 < b_2,\ b \leq b_1,\ b_2 \leq \text{I}]\end{array}$$

Figure 5.4: The Well-formedness Relation for the 2-level λ-calculus

and their types.

The well-formedness of Expressions

Now the well-formedness relation has the form

$$\text{A:I} \vdash_1 \mathtt{e} : \mathtt{t} : b \; [\text{C}]$$

and says that the term $\mathtt{e}$ has type $\mathtt{t}$ and binding time b under the assumptions A:I, and provided that the constraint set C can be solved.

The [var]-rule of Figure 5.4 says that with the assumption that the variable $\mathtt{x}$ has type $\mathtt{t}$ and binding time b, then $\mathtt{x}$ has type $\mathtt{t}$ and kind b if $\mathtt{t}$ is well-formed of kind b. This rule is much the same as the rule [var] in Figure 5.2.

The [abs]-rule says that if with the assumption list A:I, $\mathtt{x} : \mathtt{t}_1 : b_1$ the term $\mathtt{e}$ has type $\mathtt{t}_2$ and binding time b_2, and constraints C, then with the assumption list A:I the term $\lambda^{b_1}\mathtt{x.e}$ has type $\mathtt{t}_1 \rightarrow^{b_1} \mathtt{t}_2$ with binding time b_1 and constraints C and $[b_1 = b_2]$. By comparing this rule with [$\underline{\text{abs}}$] and [abs] in Figure 5.2 the rules say that the variable and the body of the abstraction must have exactly the same binding time as the λ-abstraction itself.

The rule [app] says that if with the assumption list A:I a term $\mathtt{e}_1$ has type $\mathtt{t}_1 \rightarrow^{b} \mathtt{t}_2$ and binding time b_1 and constraints C, and with the assumption list A:I a term $\mathtt{e}_2$ has type $\mathtt{t}_1$ and binding time b_2 and constraints D, then with the assumption list A:I the term $\mathtt{e}_1\ (\mathtt{e}_2)^b$ has type $\mathtt{t}_2$ and binding time b and constraints C and D and $[b = b_1 = b_2]$. By comparing this rule with the rules [$\underline{\text{app}}$] and [app] in Figure 5.2 the rules say that if with the same type environment the two terms have the same binding time, then the new term has this binding time.

The rules [if], [fix], and [const] can be explained and compared with Figure 5.2 in much the same way.

The [down]-rule is used to transform a term of run-time function type of compile-time kind into a term of run-time function type of run-time kind. The function $\mathcal{D}$ is used to generate the constraints to ensure the correct use of the [down]-rule. To explain the definition of the function consider the [down]-rule of Figure 5.2. We have to change the binding time from **c** to **r**, this is achieved by the constraint $[b_1 < b_2]$; the only solution to this is $b_2 = \mathbf{r}$ and $b_1 = \mathbf{c}$. It is required that the rule is only applied on run-time function types and not to compile-time function types; that is we must ensure that b (the annotation on the function arrow) is **r** is the only possible solution for b. This is achieved by the constraint $[b \leq b_2]$ $(= [b \leq \mathbf{r}])$. The side condition of [down] in Figure 5.2 says that the type has to be well-formed of the new binding time b_2 $(= \mathbf{r})$; this can be ensured by the constraints generated by $\mathcal{W}(\mathtt{t}_1 \rightarrow^b \mathtt{t}_2, b_2)$. But we know that $\mathtt{t}_1 \rightarrow^b \mathtt{t}_2$ is well-formed of binding time b_1 $(= \mathbf{c})$ and that the type is a run-time function type. Then we also know that $\mathtt{t}_1 \rightarrow^b \mathtt{t}_2$ is well-formed of binding time $\mathbf{r}$. So we can omit to generate the constraints $\mathcal{W}(\mathtt{t}_1 \rightarrow^b \mathtt{t}_2, b_2)$. It should only be possible to apply [down] in the case where the term has a function type and therefore $\mathcal{D}$ will generate the unsolvable constraints $[\mathbf{r} = \mathbf{c}]$ in all other cases.

The [up]-rule is used to transform a term of run-time function type of run-time kind into a term of run-time function type of compile-time kind. The function $\mathcal{U}$ is used to generate constraints to ensure the correct use of the [up]-rule. This time we have to ensure that the binding time is changed from $\mathbf{r}$ to $\mathbf{c}$, this is done by the constraint $[b_1 < b_2]$, which has one solution, $b_1 = \mathbf{r}$ and $b_2 = \mathbf{c}$. Furthermore in the assumption list all the binding times have to be compile-time and this is ensured by the constraints $[b_2 \leq$ I$]$ because $b_2 = \mathbf{c}$. The operation is point-wise on I and can be written as

$$\begin{aligned} [b \leq ()] &= () \\ [b \leq (b_1, I)] &= [b \leq b_1, b \leq I] \end{aligned}$$

where we write (b_1, I) for the list of bindings times with the first element b_1 and the rest is I. But we will write it as $[b \leq$ I$]$ for simplicity. We have to ensure that the rule is only applied on run-time function types; that is we must ensure that b is $\mathbf{r}$: here the constraint $[b \leq b_1]$ $(= [b \leq \mathbf{r}])$ will do. We have to ensure that the type is well-formed of the new binding time b_2 $(= \mathbf{c})$; this can be ensured by the constraints generated by $\mathcal{W}(\mathtt{t}_1 \rightarrow^b \mathtt{t}_2, b_2)$. Again we use the fact that $\mathtt{t}_1 \rightarrow^b \mathtt{t}_2$ is well-formed of binding time b_1 $(= \mathbf{r})$ and that it is a run-time function type and therefore the type is also well-formed of binding time $\mathbf{c}$. Finally, in the cases where the term does not have a function type we let $\mathcal{U}$ generate an unsolvable constraint $[\mathbf{r} = \mathbf{c}]$.

Properties of the well-formedness relation

All the types constructed in $\vdash_1$ are well-formed. This property is ensured by Lemma 5.6:

Lemma 5.6
We have

$$\begin{array}{ll} & \text{A:I} \vdash_1 \texttt{e} : \texttt{t} : b \text{ [C] and the constraints C are solvable by } M \\ \Downarrow & \\ & \mathcal{W}(\texttt{t}, b) \text{ is solvable by } M \end{array}$$

□

Proof We will assume A:I $\vdash_1$ $\texttt{e} : \texttt{t} : b$ [C], and that C is solvable by M, We will show that $\mathcal{W}(\texttt{t}, b)$ is solvable by M by induction on the proof-tree for A:I $\vdash_1$ $\texttt{e} : \texttt{t} : b$ [C].

For the full details see Appendix page 377. ■

The well-formedness relation in Figure 5.4 is sound and complete with respect to the one defined in Figure 5.2:

Proportion 5.7 *Soundness of* $\vdash_1$
We have

$$\begin{array}{ll} & \text{A:I} \vdash_1 \texttt{e} : \texttt{t} : b \text{ [C]} \\ & \wedge \text{ C is solvable by } M \\ & \wedge\ tenv\ \texttt{x} = (M\texttt{t}', Mp), \text{if } \texttt{x} : \texttt{t}' : p \in A : I \\ \Downarrow & \\ & tenv \vdash_0 M\texttt{e} : M\texttt{t} : Mb \end{array}$$

□

Proof We assume A:I $\vdash_1$ $\texttt{e} : \texttt{t} : b$ [C], that the constraints C are solvable by M,

$$tenv\ \texttt{x} = (M\texttt{t}', Mp), \text{if } \texttt{x} : \texttt{t}' : p \in A : I$$

and we show by induction on the proof-tree for A:I $\vdash_1$ $\texttt{e} : \texttt{t} : b$ [C] that $tenv \vdash_0 M\texttt{e} : M\texttt{t} : Mb$ can be inferred.

For the details see Appendix page 378. ■

Proportion 5.8 *Completeness of* $\vdash_1$
We have

$$tenv \vdash_0 \texttt{e} : \texttt{t} : b$$

$$\Downarrow$$

$$\begin{array}{l} \exists \text{ C solvable} \\ \mathtt{x} : \mathtt{t}_1 : p \in \text{A} : \text{I}, \text{if } \textit{tenv}\ \mathtt{x} = (\mathtt{t}_1, p) \\ \text{A:I} \vdash_1 \mathtt{e} : \mathtt{t} : b\ [\text{C}] \end{array}$$

$\square$

Proof We will assume $\textit{tenv} \vdash_0 \mathtt{e} : \mathtt{t} : b$ and then we will show that there exists constraints C such that C is solvable by M, A:I $\vdash_1 \mathtt{e} : \mathtt{t} : b$ [C] can be inferred (where $\mathtt{x} : \mathtt{t}_1 : p \in$ A:I, if $\textit{tenv}\ \mathtt{x} = (\mathtt{t}_1, p)$) by induction on the proof-tree of $\textit{tenv} \vdash_0 \mathtt{e} : \mathtt{t} : b$.

For the details see Appendix page 381. $\blacksquare$

Example 5.9
As an example of using the system in Figure 5.4 we use the same term as in Example 5.5. We will see that we with the construction of one proof-tree can capture the construction of several proof-tree in the analysis in Figure 5.2. Our main goal is to show that the term

$$\lambda^{\mathbf{c}}\mathtt{x}.\lambda^{\mathbf{c}}\mathtt{y}.\mathtt{x}\ (\mathtt{y})^{\mathbf{r}}$$

has type

$$((\mathtt{B}^{\mathbf{r}} \rightarrow^{\mathbf{r}} \mathtt{B}^{\mathbf{r}}) \rightarrow^{\mathbf{c}} (\mathtt{B}^{\mathbf{r}} \rightarrow^{\mathbf{r}} \mathtt{B}^{\mathbf{r}})) \rightarrow^{\mathbf{c}} (\mathtt{B}^{\mathbf{r}} \rightarrow^{\mathbf{r}} \mathtt{B}^{\mathbf{r}}) \rightarrow^{\mathbf{r}} (\mathtt{B}^{\mathbf{r}} \rightarrow^{\mathbf{r}} \mathtt{B}^{\mathbf{r}})$$

and binding time $\mathbf{c}$ for some base type $\mathtt{B}$. In doing this we first place binding time variables everywhere we can to make the proof more general. The question is now how can we annotate the term

$$\lambda^{b_1}\mathtt{x}.\lambda^{b_2}\mathtt{y}.\mathtt{x}\ (\mathtt{y})^{b_3}$$

given that it has binding time p? We start by defining the list of assumptions A:I to be

$$\text{A} : \text{I} \;=\; \mathtt{x} : \mathtt{t}_{\mathtt{x}} : p_1, \mathtt{y} : \mathtt{t}_{\mathtt{y}} : p_2$$

where

$$\begin{aligned} \mathtt{t}_{\mathtt{y}} &= \mathtt{B}^{b_3} \rightarrow^{b_2} \mathtt{B}^{b_4} \\ \mathtt{t}_1 &= \mathtt{B}^{b_6} \rightarrow^{b_5} \mathtt{B}^{b_7} \\ \mathtt{t}_{\mathtt{x}} &= \mathtt{t}_{\mathtt{y}} \rightarrow^{b_1} \mathtt{t}_1 \end{aligned}$$

Using [var] twice and we get

$$\text{A:I} \vdash_1 \mathtt{x} : \mathtt{t_x} : p_1 \left[\mathcal{W}(\mathtt{t_x}, p_1)\right]$$

and

$$\text{A:I} \vdash_1 \mathtt{y} : \mathtt{t_y} : p_2 \left[\mathcal{W}(\mathtt{t_y}, p_2)\right]$$

Applying [app] we get

$$\text{A:I} \vdash_1 \mathtt{x}\ (\mathtt{y})^{b_1} : \mathtt{t}_1 : b_1 \left[\mathcal{W}(\mathtt{t_x}, p_1), \mathcal{W}(\mathtt{t_y}, p_2), p_1 = p_2 = b_1\right]$$

Applying [abs] we get

$$\mathtt{x} : \mathtt{t_x} : p_1 \vdash_1 \lambda^{p_2}\mathtt{y.x}\ (\mathtt{y})^{b_1} : \mathtt{t_y} \rightarrow^{p_2} \mathtt{t}_1 : p_2\ [\text{C}]$$

where

$$\text{C} \;=\; \mathcal{W}(\mathtt{t_x}, p_1), \mathcal{W}(\mathtt{t_y}, p_2), p_1 = p_2 = b_1, p_2 = b_1, \mathcal{W}(\mathtt{t_y}, p_2)$$

Applying [abs] once more we get to

$$\emptyset \vdash_1 \lambda^{p_1}\mathtt{x}.\lambda^{p_2}\mathtt{y.x}\ (\mathtt{y})^{b_1} : \mathtt{t_x} \rightarrow^{p_1} \mathtt{t_y} \rightarrow^{p_2} \mathtt{t}_1 : p_1\ [\text{D}]$$

where

$$\text{D} \;=\; \text{C}, p_1 = p_2, \mathcal{W}(\mathtt{t_x}, p_1)$$

All the constraints are

$$\begin{aligned} \text{D} \;=\; & [b_3 = b_2, b_4 = b_2, b_2 \leq b_1, b_6 = b_5, b_7 = b_5, b_5 \leq b_1, b_1 \leq p_1, b_3 = b_2, \\ & b_4 = b_2, b_2 \leq p_2, p_1 = p_2 = b_1, p_2 = b_1, p_1 = p_2]. \end{aligned}$$

The solutions to the constraints are displayed in Table 5.2 and the term is

$$\lambda^{b_1}\mathtt{x}.\lambda^{b_1}\mathtt{y.x}\ (\mathtt{y})^{b_1}$$

with the type

$$((\mathtt{B}^{b_2} \rightarrow^{b_2} \mathtt{B}^{b_2}) \rightarrow^{b_1} (\mathtt{B}^{b_5} \rightarrow^{b_5} \mathtt{B}^{b_5})) \rightarrow^{b_1} (\mathtt{B}^{b_2} \rightarrow^{b_2} \mathtt{B}^{b_2}) \rightarrow^{b_1} (\mathtt{B}^{b_5} \rightarrow^{b_5} \mathtt{B}^{b_5})$$

and binding time b_1.

$b_1 = p_1 = p_2$	$b_2 = b_3 = b_4$	$b_5 = b_6 = b_7$
r	**r**	**r**
c	**c**	**r**
c	**c**	**c**

Table 5.2: Solution to the constraints in Example 5.9

None of the solutions are the one we are looking for. In the proof we did not apply the rule [up] and [down] as we did in the proof in Example 5.2. Now we try to copy what we did in Example 5.2. Again we start by using the rule [var] twice as above, but before we apply [app] we apply [down] on the results from [var]. Then we get

$$\mathrm{A{:}I} \vdash_1 \mathtt{x} : \mathtt{t_x} : p_3 \ [\mathcal{W}(\mathtt{t_x}, p_1), \mathcal{D}(\mathtt{t_x}, p_1, p_3)]$$

and

$$\mathrm{A{:}I} \vdash_1 \mathtt{y} : \mathtt{t_y} : p_4 \left[\mathcal{W}(\mathtt{t_y}, p_2), \mathcal{D}(\mathtt{t_y}, p_2, p_4)\right]$$

Now we apply [app] to get

$$\mathrm{A{:}I} \vdash_1 \mathtt{x}\ (\mathtt{y})^{b_1} : \mathtt{t}_1 : b_1 \ [\mathrm{C}']$$

where

$$\mathrm{C}' = \mathcal{W}(\mathtt{t_x}, p_1), \mathcal{D}(\mathtt{t_x}, p_1, p_3), \mathcal{W}(\mathtt{t_y}, p_2), \mathcal{D}(\mathtt{t_y}, p_2, p_4), p_3 = p_4 = b_1$$

Now we apply [up] and get

$$\mathrm{A{:}I} \vdash_1 \mathtt{x}\ (\mathtt{y})^{b_1} : \mathtt{t}_1 : p_5 \ [\mathrm{C}', \mathcal{U}(\mathtt{t}_1, b_1, (p_1, p_2), p_5)]$$

Finally we apply [abs] twice to get

$$\emptyset \vdash_1 \lambda^{p_1}\mathtt{x}.\lambda^{p_3}\mathtt{y}.\mathtt{x}\ (\mathtt{y})^{b_1} : \mathtt{t_x} \rightarrow^{p_1} \mathtt{t_y} \rightarrow^{p_2} \mathtt{t}_1 : p_1 \ [\mathrm{C}'']$$

where

$$\mathrm{C}'' = \mathrm{C}', \mathcal{U}(\mathtt{t}_1, b_1, (p_1, p_2), p_5), p_2 = p_5, \mathcal{W}(\mathtt{t_y}, p_2), p_1 = p_2, \mathcal{W}(\mathtt{t_x}, p_1)$$

There is one solution to the constraints C″, which is the one we are looking for

$b_2 = b_3 = b_4$	$b_6 = b_5 = b_7$	$p_3 = p_4 = b_1$	$p_1 = p_2 = p_5$
r	**r**	**r**	**c**

□

$$
\begin{array}{rcl}
\mathcal{D}'(\mathtt{B}^b,\, b_1,\, b_2) & = & [\mathbf{r} = \mathbf{c}] \\
\mathcal{D}'(\mathtt{t}_1 \rightarrow^b \mathtt{t}_2,\, b_1,\, b_2) & = & [b_2 \leq b_1, b \leq b_2] \\
\\
\mathcal{U}'(\mathtt{B}^b,\, b_1,\, \mathrm{I},\, b_2) & = & [\mathbf{r} = \mathbf{c}] \\
\mathcal{U}'(\mathtt{t}_1 \rightarrow^b \mathtt{t}_2,\, b_1,\, \mathrm{I},\, b_2) & = & [b_1 \leq b_2, b \leq b_1, b_2 \leq b_1 \sqcup \mathrm{I}]
\end{array}
$$

Figure 5.5: [up] and [down] on Function Types

5.3 Incorporating [up] and [down]

Now we want to build the two rules [up] and [down] into all the other rules and the axiom [var]. This makes it easier to make a proof in the inference system, since we do not have to think explicitly about using [up] and [down] as we had to do in Example 5.9. This is an advantage when we construct the algorithm, since it implies that when making a proof, we just apply the rules to get all the solutions with one proof instead of two or even more proofs as in Example 5.9. To do this we proceed in four stages:

Stage 1: Modify the definition of $\mathcal{U}$ and $\mathcal{D}$ such that [up] and [down] may leave the binding time unchanged (in the case of function types).

Stage 2: Modify the definition of $\mathcal{U}$ and $\mathcal{D}$ such that [up] and [down] may succeed on base types and product types as well as function types.

Stage 3: Combine the [up] and [down] rule into one rule called [up-down].

Stage 4: Integrate the [up-down]-rule with all the other rules.

Again the new system we get in this section corresponds in a one-to-one manner to the system in Section 5.3 and therefore also to the analysis in Section 5.2.

5.3.1 [up] and [down] on Function Types

In **Stage 1** we shall modify the definitions of $\mathcal{U}$ and $\mathcal{D}$ such that [up] and [down] may leave the binding time unchanged (in the case of function types). This is done by defining two new functions $\mathcal{U}'$ and $\mathcal{D}'$ with this property and use them together with the rules in Figure 5.4 instead of $\mathcal{U}$ and $\mathcal{D}$. This new well-formed relation is called $\vdash_2$. The two functions $\mathcal{D}'$ and $\mathcal{U}'$ are defined in Figure 5.5. The idea is to allow $b_2 \le b_1$ instead of $b_2 < b_1$ in $\mathcal{D}'$ and $b_1 \le b_2$ instead of $b_1 < b_2$ in $\mathcal{U}'$. This works fine for $\mathcal{D}'$ but not for $\mathcal{U}'$. To see this consider the case where the type is a function type, $\mathtt{t}_1 \rightarrow^b \mathtt{t}_2$ and both b_1 and b_2 are $\mathbf{c}$, that means no change in binding time. Then the constraints generated by $\mathcal{U}'$ would be

$$\begin{aligned} \mathcal{U}'(\mathtt{t}_1 \rightarrow^b \mathtt{t}_2,\, b_1,\, \mathrm{I},\, b_2) &= \mathcal{U}'(\mathtt{t}_1 \rightarrow^b \mathtt{t}_2,\, \mathbf{c},\, \mathrm{I},\, \mathbf{c}) \\ &= [\mathbf{c} \le \mathbf{c}, b \le \mathbf{c}, \mathbf{c} \le I] \end{aligned}$$

Thus the assumption list is restricted to contain only compile-time variables. To avoid this we allow the constraints to also contain inequalities of the form

$$b_1 \le b_2 \sqcup \mathrm{I}$$

which means that the least upper bound of b_2 and every binding time of I has to be greater than or equal to b_1. It is an abbreviation for

$$\begin{aligned} [b_1 \le b_2 \sqcup (b, \mathrm{I})] &= [b_1 \le b_2 \sqcup b, b_1 \le b_2 \sqcup \mathrm{I}] \\ [b_1 \le b_2 \sqcup ()] &= () \end{aligned}$$

Instead of using $[b_2 \le \mathrm{I}]$ in the definition of $\mathcal{U}'$ we use $[b_2 \le b_1 \sqcup \mathrm{I}]$. This solves the problem since $\mathbf{c} \sqcup b$ is $\mathbf{c}$ for all b.

If the [up]-rule is going to change the binding time, then b_1 is $\mathbf{r}$ (and b_2 is $\mathbf{c}$) and $\mathbf{r} \sqcup b$ is b for all b and hence $[b_2 \le \mathbf{r} \sqcup \mathrm{I}]$ is equivalent to $[b_2 \le \mathrm{I}]$.

In the case of no change in binding time for a **compile-time** function type the constraints that ensure that the type is a run-time function type, that is $[b \le b_1]$ in $\mathcal{U}'$ and $[b \le b_2]$ in $\mathcal{D}'$, are both solvable because both b_1 and b_2 have to be $\mathbf{c}$ since otherwise the type is not well-formed. If both b_1 and b_2 are $\mathbf{r}$, then the type is not well-formed and the constraints are unsolvable because $[b \le b_1] = [\mathbf{c} \le \mathbf{r}]$ and $[b \le b_2] = [\mathbf{c} \le \mathbf{r}]$ are unsolvable.

The new inference system using $\mathcal{U}'$ and $\mathcal{D}'$ instead of $\mathcal{U}$ and $\mathcal{D}$ is sound and complete with respect to the inference system (Figure 5.4) using $\mathcal{U}$ and $\mathcal{D}$:

Lemma 5.10 *Soundness of* $\vdash_2$
We have

$$\begin{array}{ll} & \text{A:I} \vdash_2 \texttt{e} : \texttt{t} : b\ [\text{C}] \wedge M \text{ solves C} \\ \Downarrow & \\ & \exists \text{C}' : \text{A:I} \vdash_1 \texttt{e} : \texttt{t} : b\ [\text{C}'] \wedge M \text{ solves C}' \end{array}$$

□

Proof We assume A:I $\vdash_2$ $\texttt{e} : \texttt{t} : b$ [C] and that M solves C, then we prove that there exists constraints C′ such that A:I $\vdash_1$ $\texttt{e} : \texttt{t} : b$ [C′] and that M solves C′ by induction on the proof-tree for A:I $\vdash_2$ $\texttt{e} : \texttt{t} : b$ [C].

For the details see Appendix page 383. ■

Lemma 5.11 *Completeness of* $\vdash_2$
We have

$$\begin{array}{ll} & \text{A:I} \vdash_1 \texttt{e} : \texttt{t} : b\ [\text{C}] \wedge M \text{ solves C} \\ \Downarrow & \\ & \exists \text{C}' : \text{A:I} \vdash_2 \texttt{e} : \texttt{t} : b\ [\text{C}'] \wedge M \text{ solves C}' \end{array}$$

□

Proof We assume A:I $\vdash_1$ $\texttt{e} : \texttt{t} : b$ [C] and that M solves C, then we prove that there exists constraints C′ such that A:I $\vdash_2$ $\texttt{e} : \texttt{t} : b$ [C′] and that M solves C′ by induction on the proof-tree for A:I $\vdash_1$ $\texttt{e} : \texttt{t} : b$ [C].

For the details see Appendix page 386. ■

5.3.2 [up] and [down] on Non-function Types

In **Stage 2** we modifying the [up] and [down] rules to succeed on base type as well as function types. The two new functions $\mathcal{U}''$ and $\mathcal{D}''$ are defined in Figure 5.6 and are used in Figure 5.4 instead of $\mathcal{U}''$ and $\mathcal{D}''$. The new well-formed relation is now called $\vdash_3$. The constraints generated for a base-type must ensure that the binding time is not changed ($[b_1 = b_2]$).

$$
\begin{aligned}
\mathcal{D}''(\mathtt{B}^b,\, b_1,\, b_2) &= [b_1 = b_2] \\
\mathcal{D}''(\mathtt{t}_1 \rightarrow^b \mathtt{t}_2,\, b_1,\, b_2) &= [b_2 \leq b_1, b \leq b_2] \\
\\
\mathcal{U}''(\mathtt{B}^b,\, b_1,\, \mathrm{I},\, b_2) &= [b_1 = b_2] \\
\mathcal{U}''(\mathtt{t}_1 \rightarrow^b \mathtt{t}_2,\, b_1,\, \mathrm{I},\, b_2) &= [b_1 \leq b_2, b \leq b_1, b_2 \leq b_1 \sqcup \mathrm{I}]
\end{aligned}
$$

Figure 5.6: [up] and [down] on Non-function Types

The new inference system using $\mathcal{U}''$ and $\mathcal{D}''$ instead of $\mathcal{U}'$ and $\mathcal{D}'$ is sound and complete with respect to the inference system (Figure 5.4) using $\mathcal{U}'$ and $\mathcal{D}'$:

Lemma 5.12 *Soundness of* $\vdash_3$
We have

$$
\begin{array}{ll}
& \mathrm{A{:}I} \vdash_3 \mathtt{e} : \mathtt{t} : b\ [\mathrm{C}] \wedge M \text{ solves C} \\
\Downarrow & \\
& \exists \mathrm{C}' : \mathrm{A{:}I} \vdash_2 \mathtt{e} : \mathtt{t} : b\ [\mathrm{C}'] \wedge M \text{ solves } \mathrm{C}'
\end{array}
$$

□

Proof We assume A:I $\vdash_3$ $\mathtt{e} : \mathtt{t} : b$ [C] and that M solves C, then we prove that there exists constraints C′ such that A:I $\vdash_2$ $\mathtt{e} : \mathtt{t} : b$ [C′] and that M solves C′ by induction on the proof-tree for A:I $\vdash_3$ $\mathtt{e} : \mathtt{t} : b$ [C].

For the details see Appendix page 387. ■

$$[\text{up-down}]\ \frac{\text{A:I} \vdash_4 \texttt{e} : \texttt{t} : b_1\ [\text{C}]}{\text{A:I} \vdash_4 \texttt{e} : \texttt{t} : b_2\ [\text{C}, \mathcal{UD}(\texttt{t}, b_1, \text{I}, b_2)]}$$

$$\begin{array}{rcl} \mathcal{UD}(\texttt{B}^b, b_1, \text{I}, b_2) & = & [b_1 = b_2] \\ \mathcal{UD}(\texttt{t}_1 \rightarrow^b \texttt{t}_2, b_1, \text{I}, b_2) & = & [b \leq b_1, b \leq b_2, b_2 \leq b_1 \sqcup \text{I}] \end{array}$$

Figure 5.7: The [up-down]-rule

Lemma 5.13 *Completeness of* $\vdash_3$
We have

$$\begin{array}{ll} & \text{A:I} \vdash_2 \texttt{e} : \texttt{t} : b\ [\text{C}] \wedge M \text{ solves C} \\ \Downarrow & \\ & \exists \text{C}' : \text{A:I} \vdash_3 \texttt{e} : \texttt{t} : b\ [\text{C}'] \wedge M \text{ solves C}' \end{array}$$

□

Proof We assume A:I $\vdash_2$ $\texttt{e} : \texttt{t} : b$ [C] and that M solves C, then we prove that there exists constraints C′ such that A:I $\vdash_3$ $\texttt{e} : \texttt{t} : b$ [C′] and that M solves C′ by induction on the proof-tree for A:I $\vdash_2$ $\texttt{e} : \texttt{t} : b$ [C].

For the details see Appendix page 388. ■

5.3.3 The [up-down]-rule

In **Stage 3** we combine [up] and [down] into one rule [up-down], this can be achieved by having one rule combining [up] and [down] and then generate constraints with a new function $\mathcal{UD}$. The new well-formed relation $\vdash_4$ is defined as in Figure 5.4 but instead of using the two rules [up] and [down] we use the rule [up-down] defined in Figure 5.7.

For base types there are no change from $\mathcal{D}''$ and $\mathcal{U}''$: we still generate the constraint $[b_1 = b_2]$. We want the constraints to be as follows on function types:

b	b_1	b_2	$\mathcal{UD}(\mathtt{t}_1 \to^b \mathtt{t}_2, b_1, \mathrm{I}, b_2)$
r	**r**	**r**	*id*
r	**r**	**c**	*up*
r	**c**	**r**	*down*
r	**c**	**c**	*id*
c	**r**	**r**	*unsolvable*
c	**r**	**c**	*unsolvable*
c	**c**	**r**	*unsolvable*
c	**c**	**c**	*id*

Here *id* means that the generated constraints have to be solvable and do not change the binding time. The annotations *up* and *down* mean that the constraints have to be solvable and they change the binding time according to the application of respectively the rule [up] or [down]. The three rows marked with *unsolvable* correspond to the fact that a compile-time function type cannot be of run-time kind. Both $\mathcal{D}''$ and $\mathcal{U}''$ behaves just like this but for one case each: $\mathcal{D}''$ cannot cope with the case *up* and $\mathcal{U}''$ not with *down*. The problem comes from the bond between b_1 and b_2. Clearly we cannot include both $b_1 \le b_2$ and $b_2 \le b_1$ as then $b_1 = b_2$ would follow and rule [up-down] would always act as the identity, contrary to what we are aiming for. In the following table the constraints from $\mathcal{D}''$ and $\mathcal{U}''$ are summarised. In the last column there is a `OK` if all the constraints are solvable and `FAIL` if they are unsolvable. If the solvability of the constraints depends on I then this is showed by the constraints involving I. This only happens in the case of *up*.

b	b_1	b_2	$[b \le b_2]$	$[b \le b_1]$	$[b_2 \le b_1 \sqcup \mathrm{I}]$	solvability
r	**r**	**r**	$[\mathbf{r} \le \mathbf{r}]$	$[\mathbf{r} \le \mathbf{r}]$	$[\mathbf{r} \le \mathbf{r} \sqcup \mathrm{I}]$	`OK`
r	**r**	**c**	$[\mathbf{r} \le \mathbf{c}]$	$[\mathbf{r} \le \mathbf{r}]$	$[\mathbf{c} \le \mathbf{r} \sqcup \mathrm{I}]$	$[\mathbf{c} \le \mathbf{r} \sqcup \mathrm{I}]$
r	**c**	**r**	$[\mathbf{r} \le \mathbf{r}]$	$[\mathbf{r} \le \mathbf{c}]$	$[\mathbf{r} \le \mathbf{c} \sqcup \mathrm{I}]$	`OK`
r	**c**	**c**	$[\mathbf{r} \le \mathbf{c}]$	$[\mathbf{r} \le \mathbf{c}]$	$[\mathbf{c} \le \mathbf{c} \sqcup \mathrm{I}]$	`OK`
c	**r**	**r**	$[\mathbf{c} \le \mathbf{r}]$	$[\mathbf{c} \le \mathbf{r}]$	$[\mathbf{r} \le \mathbf{r} \sqcup \mathrm{I}]$	`FAIL`
c	**r**	**c**	$[\mathbf{c} \le \mathbf{c}]$	$[\mathbf{c} \le \mathbf{r}]$	$[\mathbf{c} \le \mathbf{r} \sqcup \mathrm{I}]$	`FAIL`
c	**c**	**r**	$[\mathbf{c} \le \mathbf{r}]$	$[\mathbf{c} \le \mathbf{c}]$	$[\mathbf{r} \le \mathbf{c} \sqcup \mathrm{I}]$	`FAIL`
c	**c**	**c**	$[\mathbf{c} \le \mathbf{c}]$	$[\mathbf{c} \le \mathbf{c}]$	$[\mathbf{c} \le \mathbf{c} \sqcup \mathrm{I}]$	`OK`

Notice that the constraints of $\mathcal{UD}(\mathtt{t}_1 \to^b \mathtt{t}_2, b_1, \mathrm{I}, b_2)$ (see Figure 5.7) are those of $\mathcal{D}''(\mathtt{t}_1 \to^b \mathtt{t}_2, b_1, b_2)$ and $\mathcal{U}''(\mathtt{t}_1 \to^b \mathtt{t}_2, b_1, \mathrm{I}, b_2)$ except that we have not included $b_2 \le b_1$ from $\mathcal{D}''$ and $b_1 \le b_2$ from $\mathcal{U}''$.

The only case where it is necessary to look at I is when b_1 is **r** and b_2 is **c**. This knowledge can be used when solving the constraints, so we collect the constraints in the form $[b_2 \leq b_1 \sqcup \mathrm{I}]$ rather than writing it out.

The relation between the inference systems of Figure 5.6 and 5.7 is given by the next two lemmas:

Lemma 5.14 *Soundness of* $\vdash_4$
We have

$$\begin{array}{ll} & \mathrm{A{:}I} \vdash_4 \mathtt{e} : \mathtt{t} : b\ [\mathrm{C}] \wedge M \text{ solves C} \\ \Downarrow & \\ & \exists \mathrm{C}' : \mathrm{A{:}I} \vdash_3 \mathtt{e} : \mathtt{t} : b\ [\mathrm{C}'] \wedge M \text{ solves } \mathrm{C}' \end{array}$$

□

Proof We assume A:I $\vdash_4$ $\mathtt{e} : \mathtt{t} : b$ [C] and that M solves C, then we prove that there exists constraints C′ such that A:I $\vdash_3$ $\mathtt{e} : \mathtt{t} : b$ [C′] and that M solves C′ by induction on the proof-tree for A:I $\vdash_4$ $\mathtt{e} : \mathtt{t} : b$ [C].

For the details see Appendix page 389. ■

Lemma 5.15 *Completeness of* $\vdash_4$
We have

$$\begin{array}{ll} & \mathrm{A{:}I} \vdash_3 \mathtt{e} : \mathtt{t} : b\ [\mathrm{C}] \wedge M \text{ solves C} \\ \Downarrow & \\ & \exists \mathrm{C}' : \mathrm{A{:}I} \vdash_4 \mathtt{e} : \mathtt{t} : b\ [\mathrm{C}'] \wedge M \text{ solves } \mathrm{C}' \end{array}$$

□

Proof We assume A:I $\vdash_3$ $\mathtt{e} : \mathtt{t} : b$ [C] and that M solves C, then we prove that there exists constraints C′ such that A:I $\vdash_4$ $\mathtt{e} : \mathtt{t} : b$ [C′] and that M solves C′ by induction on the proof-tree for A:I $\vdash_3$ $\mathtt{e} : \mathtt{t} : b$ [C].

For the details see Appendix page 391. ■

5.3.4 Making the [up-down]-rule Implicit

The new rules can now be formed like

$$\dfrac{\dfrac{}{\text{A:I} \vdash_5 \texttt{e} : \texttt{t} : b_1 \; [\text{C}]}\;[\text{old rule}]}{\text{A:I} \vdash_5 \texttt{e} : \texttt{t} : b_2 \; [\text{C}, \mathcal{UD}(\texttt{t},\, b_1,\, \text{I},\, b_2)]}\;[\text{up-down}]$$

for every rule in Figure 5.4. This is illustrated for the rule [abs]:

The new rule [abs] is obtained by

$$\dfrac{\dfrac{\text{A:I}, \texttt{x} : \texttt{t}_1 : b_1 \vdash_5 \texttt{e} : \texttt{t}_2 : b_2 \; [\text{C}]}{\text{A:I} \vdash_5 \lambda^{b_1}\texttt{x.e} : \texttt{t}_1 \to^{b_1} \texttt{t}_2 : b_1 \; [\text{C},\, b_1 = b_2,\, \mathcal{W}(\texttt{t}_1,\, b_1)]}\;[\text{old abs}]}{\text{A:I} \vdash_5 \texttt{e} : \texttt{t} : b_3 \; [\text{D}]}\;[\text{up-down}]$$

where

$$\text{D} \;=\; \text{C}, b_1 = b_2, \mathcal{W}(\texttt{t}_1,\, b_1), \mathcal{UD}(\texttt{t}_1 \to^{b_1} \texttt{t}_2,\, b_1,\, \text{I},\, b_3)$$

Figure 5.8 defines the well-formed relation with the [up-down]-rule integrated in all the logical rules, and therefore no explicit [up-down] rule. There is one exception in the rule [const] whenever we know that the type of a constant is a base-type it makes no sense to apply the [up-down]-rule.

The new inference system $\vdash_5$ is sound and complete with respect to the inference system $\vdash_4$:

Lemma 5.16 *Soundness of* $\vdash_5$
We have

$$\begin{array}{l} \qquad \text{A:I} \vdash_5 \texttt{e} : \texttt{t} : b \; [\text{C}] \wedge M \text{ solves C} \\ \Downarrow \\ \qquad \exists \text{C}' : \text{A:I} \vdash_4 \texttt{e} : \texttt{t} : b \; [\text{C}'] \wedge M \text{ solves C}' \end{array}$$

□

Proof The [up-down] rule is constructed such that it can always be applied and it can leave the binding time unchanged. So from the proof-tree of

$$\text{A:I} \vdash_4 \texttt{e} : \texttt{t} : b \; [\text{C}]$$

$$[\text{var}]\ \frac{}{\text{A:I} \vdash_5 \mathtt{x} : \mathtt{t} : b_1 \left[\begin{array}{l}\mathcal{W}(\mathtt{t},\, b),\\ \mathcal{UD}(\mathtt{t},\, b,\, \text{I},\, b_1)\end{array}\right]} \quad \text{if } \mathtt{x} : \mathtt{t} : b \in \text{A:I}$$

$$[\text{abs}]\ \frac{\text{A:I},\, \mathtt{x} : \mathtt{t}_1 : b_1 \vdash_5 \mathtt{e} : \mathtt{t}_2 : b_2\ [\text{C}]}{\text{A:I} \vdash_5 \lambda^{b_1}\mathtt{x.e} : \mathtt{t}_1 \to^{b_1} \mathtt{t}_2 : b_3 \left[\begin{array}{l}\text{C}, b_1 = b_2, \mathcal{W}(\mathtt{t}_1,\, b_1),\\ \mathcal{UD}(\mathtt{t}_1 \to^{b_1} \mathtt{t}_2,\, b_1,\, \text{I},\, b_3)\end{array}\right]}$$

$$[\text{app}]\ \frac{\text{A:I} \vdash_5 \mathtt{e}_1 : \mathtt{t}_1 \to^{b} \mathtt{t}_2 : b_1\ [\text{C}] \quad \text{A:I} \vdash_5 \mathtt{e}_2 : \mathtt{t}_2 : b_2\ [\text{D}]}{\text{A:I} \vdash_5 \mathtt{e}_1\ (\mathtt{e}_2)^{b} : \mathtt{t}_2 : b_3 \left[\begin{array}{l}\text{C}, \text{D}, b = b_1 = b_2,\\ \mathcal{UD}(\mathtt{t}_2,\, b,\, \text{I},\, b_3)\end{array}\right]}$$

$$[\text{if}]\ \frac{\begin{array}{c}\text{A:I} \vdash_5 \mathtt{e}_1 : \mathtt{Bool}^{b} : b_1\ [\text{C}]\\ \text{A:I} \vdash_5 \mathtt{e}_2 : \mathtt{t} : b_2\ [\text{D}]\\ \text{A:I} \vdash_5 \mathtt{e}_3 : \mathtt{t} : b_3\ [\text{E}]\end{array}}{\text{A:I} \vdash_5 \mathtt{if}^{b}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 : \mathtt{t} : b_4 \left[\begin{array}{l}\text{C}, \text{D}, \text{E},\\ b = b_1 = b_2 = b_3,\\ \mathcal{UD}(\mathtt{t},\, b,\, \text{I},\, b_4)\end{array}\right]}$$

$$[\text{fix}]\ \frac{\text{A:I} \vdash_5 \mathtt{e} : \mathtt{t} \to^{b} \mathtt{t} : b_1\ [\text{C}]}{\text{A:I} \vdash_5 \mathtt{fix}^{b}\ \mathtt{e} : \mathtt{t} : b_2\ [\text{C},\, b = b_1,\, \mathcal{UD}(\mathtt{t},\, b,\, \text{I},\, b_2)]}$$

$$[\text{const}]\ \frac{}{\text{A:I} \vdash_1 \mathtt{c}^{b} : \mathtt{t}_{\mathtt{c}^{b}} : b_1 \left[\mathcal{UD}(\mathtt{t}_{\mathtt{c}^{b}},\, b,\, \text{I},\, b_1)\right]}$$

$$\begin{array}{lcl}
\mathcal{W}(\mathtt{B}^{b},\, p) & = & [b = p]\\
\mathcal{W}(\mathtt{t}_1 \to^{b} \mathtt{t}_2,\, p) & = & [\mathcal{W}(\mathtt{t}_1,\, b),\, \mathcal{W}(\mathtt{t}_2,\, b),\, b \le p]\\
\\
\mathcal{UD}(\mathtt{B}^{b},\, b_1,\, \text{I},\, b_2) & = & [b_1 = b_2]\\
\mathcal{UD}(\mathtt{t}_1 \to^{b} \mathtt{t}_2,\, b_1,\, \text{I},\, b_2) & = & [b \le b_1,\, b \le b_2,\, b_2 \le b_1 \sqcup \text{I}]
\end{array}$$

Figure 5.8: The Well-formedness Relation for the 2-level λ-calculus Without [up] and [down]

we can construct a proof-tree for

$$\mathrm{A{:}I} \vdash_5 \mathtt{e} : \mathtt{t} : b\ [\mathrm{C}']$$

by applying the rule [up-down] after each rule — but not after the rule [up-down]. Now each pair of rules correspond to a rule in $\vdash_5$. ■

Lemma 5.17 *Completeness of* $\vdash_5$
We have

$$\begin{array}{ll} & \mathrm{A{:}I} \vdash_4 \mathtt{e} : \mathtt{t} : b\ [\mathrm{C}] \wedge M \text{ solves C} \\ \Downarrow & \\ & \exists \mathrm{C}' : \mathrm{A{:}I} \vdash_5 \mathtt{e} : \mathtt{t} : b\ [\mathrm{C}'] \wedge M \text{ solves } \mathrm{C}' \end{array}$$

□

Proof Since all the rules in $\vdash_5$ has the form

$$\dfrac{\dfrac{\Pi}{\mathrm{A{:}I} \vdash_5 \mathtt{e} : \mathtt{t}\!: b_1\ [\mathrm{C}]}\ [\text{old rule}]}{\mathrm{A{:}I} \vdash_5 \mathtt{e} : \mathtt{t} : b_2\ [\mathrm{C}, \mathcal{UD}(\mathtt{t}, b_1, \mathrm{I}, b_2)]}\ [\text{up-down}]$$

we can infer

$$\mathrm{A{:}I} \vdash_4 \mathtt{e} : \mathtt{t} : b\ [\mathrm{C}]$$

using the same proof-tree as for

$$\mathrm{A{:}I} \vdash_5 \mathtt{e} : \mathtt{t} : b\ [\mathrm{C}]$$

■

From the Proportions 5.7 and 5.8 and the Lemmas 5.10, 5.11, 5.12, 5.13, 5.14, 5.15, 5.16, and 5.17 follows:

Theorem 5.18 *Soundness and completeness of* $\vdash_5$ *with respect to* $\vdash_0$
We have

$$\begin{array}{ll} & \mathrm{A{:}I} \vdash_5 \mathtt{e} : \mathtt{t} : b\ [\mathrm{C}] \\ & \wedge\ \mathrm{C} \text{ is solvable by } M \\ & \wedge\ \mathit{tenv}\ \mathtt{x} = (M\mathtt{t}', Mp), \text{if } \mathtt{x} : \mathtt{t}' : p \in \mathrm{A} : \mathrm{I} \\ \Downarrow & \\ & \mathit{tenv} \vdash_0 M\mathtt{e} : M\mathtt{t} : Mb \end{array}$$

and

$$\begin{array}{ll} & tenv \vdash_0 \texttt{e} : \texttt{t} : b \\ \Downarrow & \\ & \exists \text{ C solvable} \\ & \texttt{x} : \texttt{t}_1 : p \in \text{A} : \text{I}, \text{if } tenv \ \texttt{x} = (\texttt{t}_1, p) \\ & \text{A:I} \vdash_5 \texttt{e} : \texttt{t} : b \ [\text{C}] \end{array}$$

□

Example 5.19
We will in this example see how just need to construct *one* proof-tree in order to capture all the proof-trees that can be constructed for a given term in the analysis given in Figure 5.2. We use the same term as for Example 5.9. We want to show that the term

$$\lambda^{\mathbf{c}}\texttt{x}.\lambda^{\mathbf{c}}\texttt{y}.\texttt{x}\ (\texttt{y})^{\mathbf{r}}$$

has type

$$((\texttt{B}^{\mathbf{r}} \rightarrow^{\mathbf{r}} \texttt{B}^{\mathbf{r}}) \rightarrow^{\mathbf{c}} (\texttt{B}^{\mathbf{r}} \rightarrow^{\mathbf{r}} \texttt{B}^{\mathbf{r}})) \rightarrow^{\mathbf{c}} (\texttt{B}^{\mathbf{r}} \rightarrow^{\mathbf{r}} \texttt{B}^{\mathbf{r}}) \rightarrow^{\mathbf{r}} (\texttt{B}^{\mathbf{r}} \rightarrow^{\mathbf{r}} \texttt{B}^{\mathbf{r}})$$

and binding time **c** is well-formed for some base type B. We start with using [var] twice and we have

$$\text{A:I} \vdash_1 \texttt{x} : \texttt{t}_{\texttt{x}} : p_3 \ [\mathcal{W}(\texttt{t}_{\texttt{x}}, p_1), \mathcal{UD}(\texttt{t}_{\texttt{x}}, p_1, \text{I}, p_3)]$$

and

$$\text{A:I} \vdash_1 \texttt{y} : \texttt{t}_{\texttt{y}} : p_4 \ \left[\mathcal{W}(\texttt{t}_{\texttt{y}}, p_2), \mathcal{UD}(\texttt{t}_{\texttt{y}}, p_2, \text{I}, p_4)\right]$$

where A:I is as in Example 5.9:

$$\text{A} : \text{I} \ = \ \texttt{x} : \texttt{t}_{\texttt{x}} : p_1, \texttt{y} : \texttt{t}_{\texttt{y}} : p_2$$

Applying [app] we get

$$\text{A:I} \vdash_1 \texttt{x}\ (\texttt{y})^{b_1} : \texttt{t}_1 : p_5 \left[\begin{array}{l} \mathcal{W}(\texttt{t}_{\texttt{x}}, p_1), \mathcal{UD}(\texttt{t}_{\texttt{x}}, p_1, \text{I}, p_3), \\ \mathcal{W}(\texttt{t}_{\texttt{y}}, p_2), \mathcal{UD}(\texttt{t}_{\texttt{y}}, p_2, \text{I}, p_4), \\ p_3 = p_4 = b_1, \mathcal{UD}(\texttt{t}_1, b_1, \text{I}, p_5) \end{array}\right]$$

Applying [abs] we get

$$\texttt{x} : \texttt{t}_{\texttt{x}} : p_1 \vdash_1 \lambda^{p_2}\texttt{y}.\texttt{x}\ (\texttt{y})^{b_1} : \texttt{t}_{\texttt{y}} \rightarrow^{p_2} \texttt{t}_1 : p_6 \ [\text{C}]$$

row	$b_1 = p_2 = p_4$	$b_2 = b_3 = b_4$	$b_5 = b_6 = b_7$	$p_1 = p_6$	$p_3 = p_5$	p_7
1	**r**	**r**	**r**	**r**	**r**	**r**
2	**r**	**r**	**r**	**r**	**r**	**c**
3	**r**	**r**	**r**	**c**	**r**	**c**
4	**r**	**r**	**r**	**c**	**c**	**c**
5	**c**	**r**	**r**	**c**	**c**	**c**
6	**c**	**r**	**c**	**c**	**c**	**c**
7	**c**	**c**	**r**	**c**	**c**	**c**
8	**c**	**c**	**c**	**c**	**c**	**c**

Table 5.3: Solutions to the Constraints of Example 5.19

where

$$\begin{aligned} \mathrm{C} \;=\; & \mathcal{W}(\mathtt{t_x},\, p_1), \mathcal{UD}(\mathtt{t_x},\, p_1,\, \mathrm{I},\, p_3), \mathcal{W}(\mathtt{t_y},\, p_2), \mathcal{UD}(\mathtt{t_y},\, p_2,\, \mathrm{I},\, p_4), \\ & p_3 = p_4 = b_1, \mathcal{UD}(\mathtt{t}_1,\, b_1,\, \mathrm{I},\, p_5), p_2 = p_5, \mathcal{W}(\mathtt{t_y} \rightarrow^{p_2} \mathtt{t}_1,\, p_2), \\ & \mathcal{UD}(\mathtt{t_y} \rightarrow^{p_2} \mathtt{t}_1,\, p_2,\, (p_1),\, p_6) \end{aligned}$$

Applying [abs] once more we get to

$$\emptyset \vdash_1 \lambda^{p_1}\mathtt{x}.\lambda^{p_2}\mathtt{y}.\mathtt{x}\ (\mathtt{y})^{b_1} : \mathtt{t_x} \rightarrow^{p_1} \mathtt{t_y} \rightarrow^{p_2} \mathtt{t}_1 : p_7\ [\mathrm{D}]$$

where

$$\begin{aligned} \mathrm{D} \;=\; & \mathcal{W}(\mathtt{t_x},\, p_1), \mathcal{UD}(\mathtt{t_x},\, p_1,\, \mathrm{I},\, p_3), \mathcal{W}(\mathtt{t_y},\, p_2), \mathcal{UD}(\mathtt{t_y},\, p_2,\, \mathrm{I},\, p_4), \\ & p_3 = p_4 = b_1, \mathcal{UD}(\mathtt{t}_1,\, b_1,\, \mathrm{I},\, p_5), p_2 = p_5, \mathcal{W}(\mathtt{t_y} \rightarrow^{p_2} \mathtt{t}_1,\, p_2), \\ & \mathcal{UD}(\mathtt{t_y} \rightarrow^{p_2} \mathtt{t}_1,\, p_2,\, (p_1),\, p_6) p_1 = p_6, \mathcal{W}(\mathtt{t_x},\, p_1), \\ & \mathcal{UD}(\mathtt{t_x},\, p_1,\, (),\, p_7) \end{aligned}$$

Now the term $\lambda^{p_1}\mathtt{x}.\lambda^{p_2}\mathtt{y}.\mathtt{x}\ (\mathtt{y})^{b_1}$ has type

$$\mathtt{t_x} \rightarrow^{p_1} \mathtt{t_y} \rightarrow^{p_2} \mathtt{t}_1$$

and binding time p_7.

If we look at the solutions in Table 5.3 with $p_1 = p_3 = b_1 = p_7$, then the solutions in Table 5.3 (rows one, seven, and eight) are as those in Example 5.9 plus two rows more (five and six). Row four is the solution found in the second half of Example 5.9. The two last rows (rows two and three) correspond to none of the solutions found before but they can be found. □

5.4 Generating the Constraint Set

Now we can construct an algorithm for finding the constraint set that expresses the well-formed annotations of a term. The algorithm is a variation of the algorithm $\mathcal{T}$ (Section 4.1.1) but in addition to the assumption list and the type it also returns the annotated term, the binding time of the term, and some constraints. Furthermore, the algorithm will return two lists of pending arguments to $\mathcal{UD}$ and $\mathcal{W}$, respectively. The reason for this is that in order to infer the types of terms we need to introduce *type variables*. Because of the type variables we cannot immediately find the constraints coming from $\mathcal{W}$ and $\mathcal{UD}$ when the types involves type variables; instead we collect the arguments separately in order to expose them to substitutions.

A type is now either a 2-level type or a 2-level type variable. A type variable is written X^b, where b is the binding time of the type and X is the name of the type variable.

$$
\begin{array}{lcl@{\qquad\qquad}lcl}
\mathcal{K}\ (\mathtt{B}^b) & = & b & \mathcal{P}\ (\mathtt{B}^b) & = & \mathtt{B} \\
\mathcal{K}\ (\mathtt{t}_1 \to^b \mathtt{t}_2) & = & b & \mathcal{P}\ (\mathtt{t}_1 \to^b \mathtt{t}_2) & = & \mathtt{t}_1 \to \mathtt{t}_2 \\
\mathcal{K}\ (\mathrm{X}^b) & = & b & \mathcal{P}\ (\mathrm{X}^b) & = & \mathrm{X}
\end{array}
$$

Figure 5.9: Auxiliary Functions $\mathcal{K}$ and $\mathcal{P}$

The *pseudo types*, `pt`, are as the 2-level types but without the top level binding time:

$$
\begin{array}{rcl}
\mathtt{pt} & ::= & \mathtt{B} \mid \mathtt{t} \to \mathtt{t} \\
\mathtt{t} & ::= & \mathtt{B}^s \mid \mathtt{t} \to^s \mathtt{t} \\
s & ::= & \mathbf{r} \mid \mathbf{c} \mid b
\end{array}
$$

The auxiliary function $\mathcal{K}$ (Figure 5.9) is an easy way to get only the top-level binding time of a type and the function $\mathcal{P}$ (Figure 5.9) gets only the pseudo type of the type.

Definition 5.20
Here a substitution is a mapping from type variables and binding time variables to *pseudo types* and binding times. A substitution S applied to a type is defined by:

$$\begin{aligned} S\mathrm{X}^b &= (S\mathrm{X})^{Sb} \\ S\mathtt{B}^b &= \mathtt{B}^{Sb} \\ S(\mathtt{t}_1 \to^b \mathtt{t}_2) &= S\mathtt{t}_1 \to^{Sb} S\mathtt{t}_2 \end{aligned}$$

and on an assumption list

$$S(\mathtt{x}_1 : \mathtt{t}_1 : b_1, \ldots, \mathtt{x}_n : \mathtt{t}_n : b_n) = (\mathtt{x}_1 : S\mathtt{t}_1 : Sb_1, \ldots, \mathtt{x}_n : S\mathtt{t}_n : Sb_n)$$

and on the first pending list

$$\begin{aligned} &S[(\mathtt{t}_1, b_{11}, \mathrm{I}_1, b_{21}), \ldots, (\mathtt{t}_n, b_{1n}, \mathrm{I}_n, b_{2n})] \\ &\quad = [(S\mathtt{t}_1, Sb_{11}, S\mathrm{I}_1, Sb_{21}), \ldots, (S\mathtt{t}_n, Sb_{1n}, S\mathrm{I}_n, Sb_{2n})] \end{aligned}$$

and on the second pending list

$$S[(\mathtt{t}_1, b_1), \ldots, (\mathtt{t}_n, b_n)] = [(S\mathtt{t}_1, Sb_1), \ldots, (S\mathtt{t}_n, Sb_n)]$$

and on the list of binding times

$$S(b_1, \ldots, b_n) = (Sb_1, \ldots, Sb_n)$$

and on a binding time

$$\begin{aligned} S\mathbf{r} &= \mathbf{r} \\ S\mathbf{c} &= \mathbf{c} \\ Sb &= Sb \end{aligned}$$

and on a term

$$\begin{aligned} S\mathtt{x} &= \mathtt{x} \\ S(\lambda^b\mathtt{x.e}) &= \lambda^{Sb}\mathtt{x}.S\mathtt{e} \\ S(\mathtt{e}_1\ (\mathtt{e}_2)^b) &= S\mathtt{e}_1\ (S\mathtt{e}_2)^{Sb} \\ S(\mathtt{if}^b\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3) &= \mathtt{if}^{Sb}\ S\mathtt{e}_1\ \mathtt{then}\ S\mathtt{e}_2\ \mathtt{else}\ S\mathtt{e}_3 \\ S(\mathtt{fix}^b\ \mathtt{e}) &= \mathtt{fix}^{Sb}\ S\mathtt{e} \\ S\mathtt{c}^b &= \mathtt{c}^{Sb} \end{aligned}$$

□

$$
\begin{array}{lcl}
\mathcal{U}\ (\mathrm{E}) & = & \mathcal{U}_{List}\ (\mathrm{E},\ id) \\
 & & \\
\mathcal{U}_{List}\ ([\,],\ \mathrm{S}) & = & S \\
\mathcal{U}_{List}\ ([(\mathtt{t}_1,\ b_1) = (\mathtt{t}_2,\ b_2),\ \mathrm{E}],\ S) & & \\
 & = & \text{let}\quad S' = \mathcal{U}_{Bt}\ (Sb_1,\ Sb_2) \\
 & & \qquad\ \ S'' = \mathcal{U}_{Type}\ ((S' \circ S)\mathtt{t}_1,\ (S' \circ S)\mathtt{t}_2) \\
 & & \text{in}\quad \mathcal{U}_{List}\ (\mathrm{E},\ S'' \circ S' \circ S)
\end{array}
$$

Figure 5.10: Auxiliary Functions $\mathcal{U}$ and $\mathcal{U}_{List}$

$$
\begin{array}{lcl@{\qquad\qquad}lcl}
\mathcal{U}_{Bt}\ (\mathtt{r},\ \mathtt{r}) & = & id & \mathcal{U}_{Bt}\ (b,\ \mathtt{r}) & = & [\mathtt{r}/b] \\
\mathcal{U}_{Bt}\ (\mathtt{c},\ \mathtt{c}) & = & id & \mathcal{U}_{Bt}\ (b,\ \mathtt{c}) & = & [\mathtt{c}/b] \\
\mathcal{U}_{Bt}\ (\mathtt{r},\ \mathtt{c}) & = & \mathtt{FAIL} & \mathcal{U}_{Bt}\ (\mathtt{r},\ b) & = & [\mathtt{r}/b] \\
\mathcal{U}_{Bt}\ (\mathtt{c},\ \mathtt{r}) & = & \mathtt{FAIL} & \mathcal{U}_{Bt}\ (\mathtt{c},\ b) & = & [\mathtt{c}/b] \\
 & & & \mathcal{U}_{Bt}\ (b_1,\ b_2) & = & [b_2/b_1]
\end{array}
$$

Figure 5.11: Auxiliary Function $\mathcal{U}_{Bt}$

Definition 5.21
A substitution, S, is a *ground* substitution if $S\mathrm{X}^b = \mathtt{t}$ implies that $\mathtt{t}$ has no type variables. □

The unification algorithm here is an extension of the unification algorithm presented in Chapter 4 in Figure 4.1 in that it takes the annotations into account.

Unification of types and binding times is done by the function $\mathcal{U}$ defined in Figure 5.10. The function $\mathcal{U}_{Bt}$, defined in Figure 5.11, unifies two binding times. The function $\mathcal{U}_{Type}$, presented in Figure 5.12, unifies two types. It uses $\mathcal{U}_{Bt}$ to unify the binding times. Now the function $\mathcal{U}_{List}$, defined in Figure 5.10, unifies a list of type and binding time pairs.

If a substitution is the the result of unifying two binding times, two types, or two lists of pairs of type and binding time, then the substitution unifies the two objects:

Lemma 5.22

$$
\begin{array}{lcl}
\mathcal{U}_{Type}\ (\mathrm{X}^{b},\ \mathtt{t}) & = & \text{let}\quad S' = \mathcal{U}_{Bt}\ (b_1,\ \mathcal{K}\ (\mathtt{t})) \\
& & \text{in}\quad \text{if}\quad \mathrm{X} \text{ does not occur in } \mathtt{t}, \text{ then} \\
& & \qquad\qquad [\mathcal{P}\ (S'\mathtt{t})/\mathrm{X}] \circ S' \\
& & \qquad\ \ \text{else} \\
& & \qquad\qquad \mathtt{FAIL} \\
\mathcal{U}_{Type}\ (\mathtt{t},\ \mathrm{X}^{b}) & = & \mathcal{U}_{Type}\ (\mathrm{X}^{b},\ \mathtt{t}) \\
\mathcal{U}_{Type}\ (\mathtt{B}^{b_1},\ \mathtt{B}^{b_2}) & = & \mathcal{U}_{Bt}\ (b_1,\ b_2) \\
\mathcal{U}_{Type}\ (\mathtt{t}_1 \rightarrow^{b_1} \mathtt{t}_1',\ \mathtt{t}_2 \rightarrow^{b_2} \mathtt{t}_2') & & \\
& = & \text{let}\quad S' = \mathcal{U}_{Bt}\ (b_1,\ b_2) \\
& & \qquad\ \ S'' = \mathcal{U}_{Type}\ (S'\mathtt{t}_1,\ S'\mathtt{t}_2) \\
& & \qquad\ \ S''' = \mathcal{U}_{Type}\ (S''\mathtt{t}_1',\ S''\mathtt{t}_2') \\
& & \text{in}\quad S''' \circ S'' \circ S' \\
\mathcal{U}_{Type}\ (\mathtt{t}_1,\ \mathtt{t}_2) & = & \mathtt{FAIL}
\end{array}
$$

Figure 5.12: Auxiliary Function $\mathcal{U}_{Type}$

We have if $\mathcal{U}_{Bt}(b_1,\ b_2) = S$ then

- $Sb_1 = Sb_2$
- whenever a substitution, R, unifies b_1 and b_2, then for some substitution S': $R = S' \circ S$
- dom $(S) \subseteq \mathrm{FV}(b_1) \cup \mathrm{FV}(b_2)$

and if $\mathcal{U}_{Type}(\mathtt{t}_1,\ \mathtt{t}_2) = S$ then

- $S\mathtt{t}_1 = S\mathtt{t}_2$
- whenever a substitution, R, unifies $\mathtt{t}_1$ and $\mathtt{t}_2$, then for some substitution S': $R = S' \circ S$
- dom $(S) \subseteq \mathrm{FV}(\mathtt{t}_1) \cup \mathrm{FV}(\mathtt{t}_2)$

and if $S = \mathcal{U}_{List}\ ([(\mathtt{t}_1, b_1) = (\mathtt{t}_1', b_1'), \ldots, (\mathtt{t}_n, b_n) = (\mathtt{t}_n', b_n')], S')$, then

- for all $1 \leq i \leq n : S\mathtt{t}_i = S\mathtt{t}_i'$ and $Sb_i = Sb_i'$
- whenever a substitution, R, unifies

  $$([(\mathtt{t}_1, b_1) = (\mathtt{t}_1', b_1'), \ldots, (\mathtt{t}_n, b_n) = (\mathtt{t}_n', b_n')], S')$$

  then for some substitution S'': $R = S'' \circ S$

- dom $(S) \subseteq \cup\,(\mathrm{FV}(\mathtt{t}_i) \cup \mathrm{FV}(\mathtt{t}'_i) \cup \mathrm{FV}(b_i) \cup \mathrm{FV}(b'_i))$

□

Proof For Part 1 we assume $S = \mathcal{U}_{Bt}\ (b_1, b_2)$ and that $S \neq \mathtt{FAIL}$ and we show $Sb_1 = Sb_2$ by case analysis on b_1 and b_2.

For Part 2 we assume $S = \mathcal{U}_{Type}\ (\mathtt{t}_1, \mathtt{t}_2)$ and that $S \neq \mathtt{FAIL}$ and we show $S\mathtt{t}_1 = S\mathtt{t}_2$ by induction on the type $\mathtt{t}_1$.

Part 3 is shown by induction on the length of the list.

For the details see Appendix page 393. ■

We will construct a well-formed annotation of a 1-level term $\mathtt{e}$ by first using the algorithm $\mathcal{L}$ to get A:I, $\mathtt{e}'$, $\mathtt{t}$, b, C, P and P_2. Then by using a ground substitution S_0, and $\mathcal{UD}$ and $\mathcal{W}$ on $S_0\mathrm{P}$ and $S_0\mathrm{P}_2$, respectively, all the constraints C′ are found. Now $S_0\mathrm{A{:}I} \vdash_5 S_0\mathtt{e}' : S_0\mathtt{t} : S_0 b\ [\mathrm{C}']$ can be inferred (see Proposition 5.23).

To explain the algorithm $\mathcal{L}$ (Figure 5.13 and 5.13) we have to look at the rules in Figure 5.8. The rule [var] corresponds to $\mathcal{L}(\mathtt{x})$. We give $\mathtt{x}$ the type X^{b_1}, which is a new type variable, and the binding time b_3. The assumption list must contain an assumption about $\mathtt{x}$. Note that the binding time in the assumption list is different from the overall binding time of the term $\mathtt{x}$, this is due to the application of $\mathcal{UD}$. The type has to be well-formed of kind b_2 and because the type is a type variable we put the arguments to $\mathcal{W}$ in the second pending list. We also have to apply $\mathcal{UD}$ to the type and again we put the arguments to $\mathcal{UD}$ into the first pending list.

The second rule of Figure 5.8, [abs], corresponds to $\mathcal{L}(\lambda\mathtt{x}.\mathtt{e})$. First we do a recursive call on the body of the abstraction. Whenever there is an assumption about $\mathtt{x}$ in the list of assumption then we use that to construct the type of the abstraction. Otherwise we use a fresh type variable and a fresh binding time variable. Note here that it **is** possible to apply $\mathcal{UD}$ to the type regardless of the type is containing type variable or not. The function $\mathcal{UD}$ is only looking at the top-level binding time provided the type is a function type. Here we know that the type is a function type opposed to the clause $\mathcal{L}(\mathtt{x})$ where the type is only a type variable. However it is not possible to apply $\mathcal{W}$ to a type variable, hence the arguments to $\mathcal{W}$ is put in the second pending list.

The third rule of Figure 5.8 is [app] and corresponds to $\mathcal{L}(\mathtt{e}_1\ (\mathtt{e}_2))$. Again

$$
\begin{array}{lll}
\mathcal{L}(\mathtt{x}) & = & \text{let } \ \mathrm{X},\, b_1,\, b_2,\, b_3 \text{ be fresh} \\
 & & \text{in } \ (\mathtt{x} : \mathrm{X}^{b_1} : b_2,\, \mathtt{x},\, \mathrm{X}^{b_1},\, b_3 ,\, [\,],\\
 & & \qquad [(\mathrm{X}^{b_1},\, b_2,\, (b_2),\, b_3)],\, [(\mathrm{X}^{b_1},\, b_2)])
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{L}(\lambda\mathtt{x.e}) & = & \text{let } \ b \text{ be fresh} \\
 & & \qquad (\mathrm{A{:}I},\, \mathtt{e}',\, \mathtt{t}_2,\, b_2,\, \mathrm{C},\, \mathrm{P},\, \mathrm{P}_2) = \mathcal{L}(\mathtt{e}) \\
 & & \qquad (\mathtt{t}_1,\, b_1) = \ \text{if} \quad \mathtt{x} : \mathtt{t}_1 : b_1 \in \mathrm{A{:}I} \text{ then} \\
 & & \qquad\qquad\qquad\qquad (\mathtt{t}_1,\, b_1) \\
 & & \qquad\qquad\qquad \text{else} \\
 & & \qquad\qquad\qquad\qquad \text{let X, } b_1 \text{ be fresh in } (\mathrm{X},\, b_1) \\
 & & \qquad \mathrm{B{:}J} = \ \text{if} \quad \mathtt{x} : \mathtt{t}_1 : b_1 \in \mathrm{A{:}I} \text{ then} \\
 & & \qquad\qquad\qquad \mathrm{A{:}I} \setminus \mathtt{x} : \mathtt{t}_1 : b_1 \\
 & & \qquad\qquad \text{else} \\
 & & \qquad\qquad\qquad \text{let A:I} \\
 & & \text{in } \ (\mathrm{B{:}J},\, \lambda^{b_1}\mathtt{x.e}',\, \mathtt{t}_1 \rightarrow^{b_1} \mathtt{t}_2,\, b, \\
 & & \qquad [\mathrm{C},\, b_1 = b_2,\, \mathcal{UD}(\mathtt{t}_1 \rightarrow^{b_1} \mathtt{t}_2,\, b_1,\, \mathrm{J},\, b)],\, \mathrm{P}, \\
 & & \qquad [\mathrm{P}_2,\, (\mathtt{t}_1,\, b_1)])
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{L}(\mathtt{e}_1\ (\mathtt{e}_2)) & = & \text{let } \ b_3,\, b_4,\, b_5,\, \mathrm{X} \text{ be fresh} \\
 & & \qquad (\mathrm{A{:}I},\, \mathtt{e}_1',\, \mathtt{t}_1,\, b_1,\, \mathrm{C},\, \mathrm{P},\, \mathrm{P}_2) = \mathcal{L}(\mathtt{e}_1) \\
 & & \qquad (\mathrm{B{:}J},\, \mathtt{e}_2',\, \mathtt{t}_2,\, b_2,\, \mathrm{D},\, \mathrm{Q},\, \mathrm{Q}_2) = \mathcal{L}(\mathtt{e}_2) \\
 & & \qquad S = \ \mathcal{U}([(\mathtt{t}_1,\, \mathcal{K}\ (\mathtt{t}_1)) = (\mathtt{t}_2 \rightarrow^{b_4} \mathrm{X}^{b_5},\, b_4)] \cup \\
 & & \qquad\qquad [(\mathtt{t}_1',\, b_1') = (\mathtt{t}_2',\, b_2') \mid \mathtt{x} : \mathtt{t}_1' : b_1' \in \mathrm{A{:}I}, \\
 & & \qquad\qquad\qquad\qquad \mathtt{x} : \mathtt{t}_2' : b_2' \in \mathrm{B{:}J}]) \\
 & & \text{in } \ (S(\mathrm{A{:}I},\, \mathrm{B{:}J}),\, S(\mathtt{e}_1'\ (\mathtt{e}_2')^{b_4}),\, S\mathrm{X}^{b_5},\, b_3, \\
 & & \qquad [S\mathrm{C},\, S\mathrm{D},\, Sb_1 = Sb_2 = Sb_4], \\
 & & \qquad [S\mathrm{P},\, S\mathrm{Q},\, (S\mathrm{X}^{b_5},\, Sb_4,\, S(\mathrm{I},\, \mathrm{J}),\, b_3)],\, [S\mathrm{P}_2,\, S\mathrm{Q}_2])
\end{array}
$$

Figure 5.13: Algorithm $\mathcal{L}$ for Collecting Constraints (Part 1)

we first do recursive calls on $\mathtt{e}_1$ and $\mathtt{e}_2$. Next we have to make sure that $\mathtt{e}_1$ has a function type and that $\mathtt{e}_2$ has the right type such that $\mathtt{e}_1$ can be applied to $\mathtt{e}_2$. This checking is done by unifying the type inferred for $\mathtt{e}_1$ with the type $\mathtt{t}_2 \rightarrow^{b_4} \mathrm{X}^{b_5}$ where $\mathtt{t}_2$ is the type inferred for $\mathtt{e}_2$ and X^{b_5} is a new type variable. This is done at the same time as the two assumption

$$
\begin{array}{lcll}
\mathcal{L}(\texttt{if e}_1\ \texttt{then e}_2\ \texttt{else e}_3) & & & \\
 & = & \text{let} & b_1,\, b_5 \text{ be fresh} \\
 & & & (\mathrm{A}_1{:}\mathrm{I}_1,\, \texttt{e}'_1,\, \texttt{t}_1,\, b_2,\, \mathrm{C},\, \mathrm{P},\, \mathrm{P}_2) = \mathcal{L}(\texttt{e}_1) \\
 & & & (\mathrm{A}_2{:}\mathrm{I}_2,\, \texttt{e}'_2,\, \texttt{t}_2,\, b_3,\, \mathrm{D},\, \mathrm{Q},\, \mathrm{Q}_2) = \mathcal{L}(\texttt{e}_2) \\
 & & & (\mathrm{A}_3{:}\mathrm{I}_3,\, \texttt{e}'_3,\, \texttt{t}_3,\, b_4,\, \mathrm{E},\, \mathrm{R},\, \mathrm{R}_2) = \mathcal{L}(\texttt{e}_3) \\
 & & & S = \ \mathcal{U}([(\texttt{Bool}^{b_1},\, b_1) = (\texttt{t}_1,\, \mathcal{K}\ (\texttt{t}_1), \\
 & & & \qquad (\texttt{t}_2,\, \mathcal{K}\ (\texttt{t}_2)) = (\texttt{t}_3,\, \mathcal{K}\ (\texttt{t}_3))] \cup \\
 & & & \qquad [(\texttt{t}'_1,\, b'_1) = (\texttt{t}'_2,\, b'_2),\, (\texttt{t}'_1,\, b'_1) = (\texttt{t}'_3,\, b'_3) \\
 & & & \qquad \mid \mathrm{x} : \texttt{t}'_1 : b'_1 \in \mathrm{A{:}I},\, \mathrm{x} : \texttt{t}'_2 : b'_2 \in \mathrm{B{:}J}, \\
 & & & \qquad\quad \mathrm{x} : \texttt{t}'_3 : b_3 \in \mathrm{F{:}K}]) \\
 & & \text{in} & (S(\mathrm{A}_1{:}\mathrm{I}_1,\, \mathrm{A}_2{:}\mathrm{I}_2,\, \mathrm{A}_3{:}\mathrm{I}_3), \\
 & & & S(\texttt{if}^{b_1}\ \texttt{e}'_1\ \texttt{then e}'_2\ \texttt{else e}'_3)\ S\texttt{t}_2,\, b_5, \\
 & & & [S\mathrm{C},\, S\mathrm{D},\, S\mathrm{E},\, Sb_1 = Sb_2 = Sb_3 = Sb_4\ ], \\
 & & & [S\mathrm{P},\, S\mathrm{Q},\, S\mathrm{R},\, (S\texttt{t}_2,\, Sb_1,\, S(\mathrm{I}_1,\, \mathrm{I}_2,\, \mathrm{I}_3),\, b_5)], \\
 & & & [S\mathrm{P}_2,\, S\mathrm{Q}_2,\, S\mathrm{R}_2]) \\
 & & & \\
\mathcal{L}(\texttt{fix e}) & = & \text{let} & b_1,\, b_2,\, b_3,\, \mathrm{X} \text{ be fresh} \\
 & & & (\mathrm{A{:}I},\, \texttt{e}',\, \texttt{t},\, b_4,\, \mathrm{C},\, \mathrm{P},\, \mathrm{P}_2) = \mathcal{L}(\texttt{e}) \\
 & & & \mathrm{S} = \mathcal{U}\ ([(\texttt{t},\, \mathcal{K}\ (\texttt{t})) = (\mathrm{X}^{b_3} \rightarrow^{b_1} \mathrm{X}^{b_3},\, b_1)]) \\
 & & \text{in} & (S(\mathrm{A{:}I}),\, S\texttt{fix}^{b_1}\ \texttt{e}',\, S\mathrm{X}^{b_3},\, b_2,\, [S\mathrm{C},\, Sb_1 = Sb_4], \\
 & & & \ [S\mathrm{P},\, (S\mathrm{X}^{b_3},\, Sb_1,\, S\mathrm{I},\, b_2)],\, S\mathrm{P}_2) \\
 & & & \\
\mathcal{L}(\texttt{c}) & = & \text{let} & b_2,\, b_3 \text{ be fresh} \\
 & & \text{in} & ((\ ),\, \texttt{c}^{b_2},\, \texttt{tc}_{b_2},\, b_3,\, [\ ], \\
 & & & [(\texttt{tc}_{b_2},\, b_2,\, (\ ),\, b_3)],\, [(\texttt{tc}_{b_2},\, b_2)])
\end{array}
$$

Figure 5.14: Algorithm $\mathcal{L}$ for Collecting Constraints (Part 2)

lists are compared. They have to agree on the type and binding time of the variables. The substitution obtained by the unification has to be applied to all the types and binding times. It is needless to apply the substitution to the binding time variable b_3 because it is a new binding time variable and it has not taken part in the unification.

The remaining three rules in Figure 5.8 corresponds to the last three cases in the definition of $\mathcal{L}$. The are obtained in the same way as the first three clauses.

The algorithm $\mathcal{L}$ is sound with respect to the inference system and complete:

Proportion 5.23 *Soundness of $\mathcal{L}$*
Whenever

$$\mathcal{L}(\mathtt{e}) = (\mathrm{A{:}I},\ \mathtt{e}',\ \mathtt{t},\ b,\ \mathrm{C},\ \mathrm{P},\ \mathrm{P}_2)$$

and S_0 is a ground substitution, then there exists constraints C″ such that

$$S_0(\mathrm{A{:}I}) \vdash_5 S_0\mathtt{e}' : S_0\mathtt{t} : S_0 b\ [\mathrm{C}'']$$

where

$$\mathrm{C}' = [S_0\mathrm{C}, \mathcal{UD}(S_0\mathrm{P}), \mathcal{W}(S_0\mathrm{P}_2)]$$

and C′ is solvable by M implies that C″ is solvable by M.

□

Proof We show the proposition by induction on the term e.

For the details see Appendix page 398. ■

Proportion 5.24 *Completeness of $\mathcal{L}$*
If $\mathrm{A{:}I} \vdash_5 \mathtt{e} : \mathtt{t} : b\ [\mathrm{C}]$ then there exists a ground substitution, S, and a subset, A″, of A such that

$$\mathcal{L}(\mathtt{e}) = (\mathrm{A}' : \mathrm{I}', \mathtt{e}', \mathtt{t}', b', \mathrm{C}', \mathrm{P}, \mathrm{P}_2)$$

and

$$\begin{aligned} \mathtt{e} &= S\mathtt{e}' \\ \mathtt{t} &= S\mathtt{t}' \\ b &= Sb' \\ \mathrm{A}'' : \mathrm{I}'' &= S(\mathrm{A}' : \mathrm{I}') \\ \mathrm{C}'' &= [S\mathrm{C}, \mathcal{UD}(S\mathrm{P}), \mathcal{W}(S\mathrm{P}_2)] \end{aligned}$$

and

$$M \text{ solves C} \;\Rightarrow\; M \text{ solves C}''$$

$\square$

Proof We will assume that $\text{A:I} \vdash_5 \texttt{e} : \texttt{t} : b$ [C] can be inferred and that M solves C. We will show the Proposition by induction on the proof-tree for $\text{A:I} \vdash_5 \texttt{e} : \texttt{t} : b$ [C].

For the details see Appendix page 400. $\blacksquare$

Example 5.25
Now we can let the algorithm compute the set of constraints in stead of doing the ourselves as we did in Example 5.19. As an example we calculate $\mathcal{L}(\lambda\texttt{x}.\lambda\texttt{y}.\texttt{x y})$. First we calculate $\mathcal{L}(\text{x})$ and $\mathcal{L}\text{y}$:

$$\begin{array}{rcll}
\mathcal{L}(\texttt{x}) & = & \text{let} & p_1,\, b_1,\, b_6,\, \text{X}_1 \text{ be fresh}\\
& & \text{in} & (\texttt{x} : \text{X}_1{}^{p_1} : b_1,\, \texttt{x},\, \text{X}_1{}^{p_1},\, [\,],\, [(\text{X}_1{}^{p_1},\, b_1,\, (b_1),\, b_6)],\\
& & & [(\text{X}_1{}^{p_1},\, b_1)])
\end{array}$$

$$\begin{array}{rcll}
\mathcal{L}(\texttt{y}) & = & \text{let} & p_2,\, b_2,\, b_7,\, \text{X}_2 \text{ be fresh}\\
& & \text{in} & (\texttt{y} : \text{X}_2{}^{p_2} : b_2,\, \texttt{y},\, \text{X}_2{}^{p_2},\, b_7,\, [\,],\, [(\text{X}_2{}^{p_2},\, b_2,\, (b_2),\, b_7)],\\
& & & [(\text{X}_2{}^{p_2},\, b_2)])
\end{array}$$

Now we can calculate $\mathcal{L}(\texttt{x y})$:

$$\begin{array}{rcll}
\mathcal{L}(\texttt{x y}) & = & \text{let} & b_8,\, b_3,\, b_9,\, \text{X}_3,\, p_1,\, b_1,\, b_6,\, \text{X}_1,\, p_2,\, b_2,\, b_7,\, \text{X}_2 \text{ be fresh}\\
& & & S_3 = \mathcal{U}\ ([(\text{X}_1{}^{p_1},\, p_1) = (\text{X}_2{}^{p_2} \rightarrow^{b_3} \text{X}_3{}^{b_9},\, b_3)])\\
& & \text{in} & (S_3\ (\texttt{x} : \text{X}_1{}^{p_1} : b_1,\, \texttt{y} : \text{X}_2{}^{p_2} : b_2),\, S_3(\texttt{x}\ (\texttt{y})^{b_3}),\, S_3\text{X}_3{}^{b_9},\\
& & & b_8,\, [S_3 b_6 = S_3 b_7 = S_3 b_3],\\
& & & [(S_3\text{X}_1{}^{p_1},\, S_3 b_1,\, (S_3 b_1),\, S_3 b_6),\\
& & & (S_3\text{X}_2{}^{p_2},\, S_3 b_2,\, (S_3 b_2),\, S_3 b_7),\\
& & & (S_3\text{X}_3{}^{b_9},\, S_3 b_3,\, (S_3 b_1,\, S_3 b_2),\, S_3 b_8)],\\
& & & [(S_3\text{X}_1{}^{p_1},\, S_3 b_1),\, (S_3\text{X}_2{}^{p_2},\, S_3 b_2)])
\end{array}$$

And $\mathcal{L}(\lambda\texttt{y}.\texttt{x y})$:

$$
\begin{array}{rcll}
\mathcal{L}(\lambda \texttt{y}.\texttt{x y}) & = & \text{let} & b_8,\, b_3,\, b_9,\, \mathrm{X}_3,\, p_1,\, b_1,\, b_6,\, \mathrm{X}_1,\, p_2,\, b_2, \\
& & & b_7,\, \mathrm{X}_2,\, b_4 \text{ be fresh} \\
& & & S_3 = \mathcal{U}\ ([(\mathrm{X}_1{}^{p_1},\, p_1) = (\mathrm{X}_2{}^{p_2} \rightarrow^{b_3} \mathrm{X}_3{}^{b_9},\, b_3)]) \\
& & \text{in} & (S_3\ (\texttt{x} : \mathrm{X}_1{}^{p_1} : b_1),\, \lambda^{S_3 b_2}\texttt{y}.S_3(\texttt{x}\ (\texttt{y})^{b_3}), \\
& & & S_3\mathrm{X}_2{}^{p_2} \rightarrow^{S_3 b_2} S_3\mathrm{X}_3{}^{b_9},\, b_4, \\
& & & [S_3 b_6 = S_3 b_7 = S_3 b_3, \\
& & & \mathcal{UD}(S_3\mathrm{X}_2{}^{p_2} \rightarrow^{S_3 b_2} S_3\mathrm{X}_3{}^{b_9},\, S_3 b_2,\, (S_3 b_1),\, b_4), \\
& & & S_3 b_2 = S_3 b_8], \\
& & & [(S_3\mathrm{X}_1{}^{p_1},\, S_3 b_1,\, (S_3 b_1),\, S_3 b_6), \\
& & & (S_3\mathrm{X}_2{}^{p_2},\, S_3 b_2,\, (S_3 b_2),\, S_3 b_7), \\
& & & (S_3\mathrm{X}_3{}^{b_9},\, S_3 b_3,\, (S_3 b_1,\, S_3 b_2),\, S_3 b_8)], \\
& & & [(S_3\mathrm{X}_1{}^{p_1},\, S_3 b_1),\, (S_3\mathrm{X}_2{}^{p_2},\, S_3 b_2),\, (S_3\mathrm{X}_2{}^{p_2},\, S_3 b_2)])
\end{array}
$$

Finally we calculate $\mathcal{L}(\lambda \texttt{x}.\lambda \texttt{y}.\texttt{x y})$:

$$
\begin{array}{rcll}
\mathcal{L}(\lambda \texttt{x}.\lambda \texttt{y}.\texttt{x y}) & = & \text{let} & b_8,\, b_3,\, b_9,\, \mathrm{X}_3,\, p_1,\, b_1,\, b_6,\, \mathrm{X}_1,\, p_2,\, b_2, \\
& & & b_7,\, \mathrm{X}_2,\, b_4,\, b_5 \text{ be fresh} \\
& & & S_3 = \mathcal{U}\ ([(\mathrm{X}_1{}^{p_1},\, p_1) = (\mathrm{X}_2{}^{p_2} \rightarrow^{b_3} \mathrm{X}_3{}^{b_9},\, b_3)]) \\
& & \text{in} & ((\ ),\, \lambda^{S_3 b_1}\texttt{x}.\lambda^{S_3 b_2}\texttt{y}.S_3(\texttt{x}\ (\texttt{y})^{b_3}), \\
& & & S_3\mathrm{X}_1{}^{p_1} \rightarrow^{S_3 b_1} S_3\mathrm{X}_2{}^{p_2} \rightarrow^{S_3 b_2} S_3\mathrm{X}_3{}^{b_9},\, b_5, \\
& & & [S_3 b_6 = S_3 b_7 = S_3 b_3, \\
& & & \mathcal{UD}(S_3\mathrm{X}_2{}^{p_2} \rightarrow^{S_3 b_2} S_3\mathrm{X}_3{}^{b_9},\, S_3 b_2,\, (S_3 b_1),\, b_4), \\
& & & S_3 b_2 = S_3 b_8, \\
& & & \mathcal{UD}(S_3\mathrm{X}_1{}^{p_1} \rightarrow^{S_3 b_1} S_3\mathrm{X}_2{}^{p_2} \rightarrow^{S_3 b_2} S_3\mathrm{X}_3{}^{b_9}, \\
& & & S_3 b_1,\, (),\, b_5), \\
& & & S_3 b_1 = b_4], \\
& & & [(S_3\mathrm{X}_1{}^{p_1},\, S_3 b_1,\, (S_3 b_1),\, S_3 b_6), \\
& & & (S_3\mathrm{X}_2{}^{p_2},\, S_3 b_2,\, (S_3 b_2),\, S_3 b_7), \\
& & & (S_3\mathrm{X}_3{}^{b_9},\, S_3 b_3,\, (S_3 b_1,\, S_3 b_2),\, S_3 b_8)], \\
& & & [(S_3\mathrm{X}_1{}^{p_1},\, S_3 b_1),\, (S_3\mathrm{X}_2{}^{p_2},\, S_3 b_2),\, (S_3\mathrm{X}_2{}^{p_2},\, S_3 b_2), \\
& & & (S_3\mathrm{X}_1{}^{p_1},\, S_3 b_1)])
\end{array}
$$

The substitution S_3 is

$$
\begin{array}{rcl}
S_3 & = & \mathcal{U}([(\mathrm{X}_1{}^{p_1}, p_1) = (\mathrm{X}_2{}^{p_2} \rightarrow^{b_3} \mathrm{X}_3{}^{b_9}, b_3)]) \\
& = & \mathcal{U}_{List}\ (([(\mathrm{X}_1{}^{p_1}, p_1) = (\mathrm{X}_2{}^{p_2} \rightarrow^{b_3} \mathrm{X}_3{}^{b_9}, b_3)], id) \\
& = & \mathcal{U}_{List}\ ([\ ], S'' \circ S' \circ id) \\
& = & S'' \circ S' \circ id
\end{array}
$$

where

$$\begin{aligned} S' &= \mathcal{U}_{Bt}(idp_1, idb_3) \\ S'' &= \mathcal{U}_{Type}((S' \circ id)\mathrm{X_1}^{p_1}, (S' \circ id)(\mathrm{X_2}^{p_2} \rightarrow^{b_3} \mathrm{X_3}^{b_9})) \end{aligned}$$

We have

$$\begin{aligned} S' &= \mathcal{U}_{Bt}(idp_1, idb_3) \\ &= \mathcal{U}_{Bt}(p_1, b_3) \\ &= [b_3/p_1] \end{aligned}$$

and

$$\begin{aligned} S'' &= \mathcal{U}_{Type}((S' \circ id)\mathrm{X_1}^{p_1}, (S' \circ id)(\mathrm{X_2}^{p_2} \rightarrow^{b_3} \mathrm{X_3}^{b_9})) \\ &= \mathcal{U}_{Type}(S'\mathrm{X_1}^{p_1}, S'(\mathrm{X_2}^{p_2} \rightarrow^{b_3} \mathrm{X_3}^{b_9})) \\ &= \mathcal{U}_{Type}(\mathrm{X_1}^{b_3}, (\mathrm{X_2}^{p_2} \rightarrow^{b_3} \mathrm{X_3}^{b_9})) \\ &= [\mathcal{P}(S_2'(\mathrm{X_2}^{p_2} \rightarrow^{b_3} \mathrm{X_3}^{b_9}))\mathrm{X_1}] \circ S_2' \\ &= [(S_2'(\mathrm{X_2}^{p_2} \rightarrow \mathrm{X_3}^{b_9}))\mathrm{X_1}] \circ S_2' \end{aligned}$$

where

$$\begin{aligned} S_2' &= \mathcal{U}_{Bt}(b_3, b_3) \\ &= [b_3/b_3] \\ &= id \end{aligned}$$

Now we have

$$\begin{aligned} S_3 &= S'' \circ S' \circ id \\ &= S'' \circ [b_3/p_1] \\ &= [(S_2'(\mathrm{X_2}^{p_2} \rightarrow \mathrm{X_3}^{b_9}))/\mathrm{X_1}] \circ S_2' \circ [b_3/p_1] \\ &= [(S_2'(\mathrm{X_2}^{p_2} \rightarrow \mathrm{X_3}^{b_9}))/\mathrm{X_1}] \circ id \circ [b_3/p_1] \\ &= [(S_2'(\mathrm{X_2}^{p_2} \rightarrow \mathrm{X_3}^{b_9}))/\mathrm{X_1}] \circ [b_3/p_1] \end{aligned}$$

The term

$$\lambda^{S_3 b_1}\mathtt{x}.\lambda^{S_3 b_2}\mathtt{y}.S_3(\mathtt{x}\ (\mathtt{y})^{b_3})$$

equals

$$\lambda^{b_1}\mathtt{x}.\lambda^{b_2}\mathtt{y}.\mathtt{x}\ (\mathtt{y})^{b_3}$$

and has the type

$$(\mathrm{X_2}^{p_2} \rightarrow^{b_3} \mathrm{X_3}^{b_9}) \rightarrow^{b_1} \mathrm{X_2}^{p_2} \rightarrow^{b_2} \mathrm{X_3}^{b_9}$$

and binding time b_5. We now want to find the constraints such that the term has the same type as in Example 5.19, i.e. the type is

$$\mathtt{t}'_\mathtt{x} \rightarrow^{p'_1} \mathtt{t}'_\mathtt{y} \rightarrow^{p'_2} \mathtt{t}'_1$$

where

$$\begin{aligned} \mathtt{t}'_\mathtt{y} &= \mathtt{B}^{b'_2} \rightarrow^{b'_2} \mathtt{B}^{b'_2} \\ \mathtt{t}'_1 &= \mathtt{B}^{b'_5} \rightarrow^{b'_5} \mathtt{B}^{b'_5} \\ \mathtt{t}'_\mathtt{x} &= \mathtt{t}'_\mathtt{y} \rightarrow^{b'_1} \mathtt{t}'_1 \end{aligned}$$

We will use the ground substitution S_0:

$$\begin{aligned} S_0 \;=\; & [\mathcal{P}(\mathtt{t}'_\mathtt{y})/\mathrm{X}_2] \circ [\mathcal{P}(\mathtt{t}'_1)/\mathrm{X}_3] \circ \\ & [b'_5/b_9] \circ [b'_2/p_2] \circ [p'_3/b_2] \circ [p'_1/b_1] \circ [b'_1/b_3] \end{aligned}$$

and the constraints:

$$\begin{aligned} &[S_0b_6 = S_0b_7 = S_0b_3, \mathcal{UD}(S_0(\mathrm{X_2}^{p_2} \rightarrow^{b_2} \mathrm{X_3}^{b_9}),\ S_0b_2,\ (S_0b_1),\ S_0b_4), \\ &S_0b_2 = S_0b_8, \\ &\mathcal{UD}(S_0((\mathrm{X_2}^{p_2} \rightarrow^{b_2} \mathrm{X_3}^{b_9}) \rightarrow^{b_1} \mathrm{X_2}^{p_2} \rightarrow^{b_2} \mathrm{X_3}^{b_9}),\ S_0b_1,\ (),\ S_0b_5), \\ &S_0b_1 = S_0b_4, \mathcal{UD}(S_0(\mathrm{X_2}^{p_2} \rightarrow^{b_2} \mathrm{X_3}^{b_9}),\ S_0b_1,\ (S_0b_1),\ S_0b_6), \\ &\mathcal{UD}(S_0\mathrm{X_2}^{p_2},\ S_0b_2,\ (S_0b_2),\ S_0b_7), \mathcal{UD}(S_0\mathrm{X_3}^{b_9},\ S_0b_3,\ (S_0b_1,\ S_0b_2),\ S_0b_8), \\ &\mathcal{W}(S_0(\mathrm{X_2}^{p_2} \rightarrow^{b_2} \mathrm{X_3}^{b_9}),\ S_0b_1), \mathcal{W}(S_0\mathrm{X_2}^{S_0p_2},\ S_0b_2), \\ &\mathcal{W}(S_0\mathrm{X_2}^{S_0p_2},\ S_0b_2), \mathcal{W}(S_0(\mathrm{X_2}^{p_2} \rightarrow^{b_2} \mathrm{X_3}^{b_9}),\ S_0b_1)] \end{aligned}$$

By carefully examination of the constraints we can observe that these constraints are comparable to the constraints found in Example 5.19.

The next step is to solve the constraints. □

5.5 Solving the Constraint Set

Here as in [NN92] we will only find the solution with as many **c**'s as possible. This is called the minimal solution because *most* of the work is done at compile-time and a *minimum* at run-time. The minimal solution to the various forms of constraints are listed in Table 5.4.

First we assume that the solution to the constraints C are the one that maps all the binding time variables to **c**. Next we have to find the constraints that forces some of the binding time variables to be mapped to **r**. This can then affect other constraints. So we have to find the constraints that are affected by this. The algorithm is as follows:

1. Divide the constraints into the three groups of Table 5.5:
   - The constraints that *are not* affected by some binding time variable being mapped to **r**.
   - The constraints that *forces* the solution to map some binding time variable to **r** .
   - The constraints that *are* affected by some binding time variable being mapped to **r**.
2. Find the binding time variables that have to be mapped to **r**.
3. Apply the intermediate solution to the rest of the constraints. That is only the constraints in the last group — those that are affected by the forced binding time variables.

   We will not consider the constraints that are not affected by the forcing binding time variables, again. Hence there is no need to apply the substitution to them. The constraints that force the binding time to be **r** will not give any more contribution to the solution of the set of constraints.
4. Repeat step 1 to 3 on the set of constraints that are affected by intermediate solution until either
   - no constraints force any binding time variables to be mapped to **r**
   - no constraints are affected by the intermediate solution

constraint	minimal solution
$b = \mathbf{r}$	$M\ b = \mathbf{r}$
$b = \mathbf{c}$	$M\ b = \mathbf{c}$
$\mathbf{r} = b$	$M\ b = \mathbf{r}$
$\mathbf{c} = b$	$M\ b = \mathbf{c}$
$b \leq \mathbf{r}$	$M\ b = \mathbf{r}$
$\mathbf{c} \leq b$	$M\ b = \mathbf{c}$
$\mathbf{c} \leq \mathbf{r} \sqcup \mathrm{I}$	$\begin{cases} M\ b = \mathbf{c} \text{ for all } b \text{ in I, if I contains no } \mathbf{r} \\ \text{unsolvable, otherwise} \end{cases}$
$\mathbf{r} = \mathbf{r}$	
$\mathbf{c} = \mathbf{c}$	
$b_1 = b_2$	$M\ b_1 = \mathbf{c}$ and $M\ b_2 = \mathbf{c}$
$\mathbf{r} \leq \mathbf{r}$	
$\mathbf{r} \leq \mathbf{c}$	
$\mathbf{c} \leq \mathbf{c}$	
$\mathbf{r} \leq b$	$M\ b = \mathbf{c}$
$b \leq \mathbf{c}$	$M\ b = \mathbf{c}$
$b_1 \leq b_2$	$M\ b_1 = \mathbf{c}$ and $M\ b_2 = \mathbf{c}$
$\mathbf{r} \leq \mathbf{c} \sqcup \mathrm{I}$	$M\ b_i = \mathbf{c}$, for all b_i in I
$\mathbf{c} \leq \mathbf{c} \sqcup$	$M\ b_i = \mathbf{c}$, for all b_i in I
$b \leq \mathbf{c} \sqcup \mathrm{I}$	$M\ b = \mathbf{c}$ and $M\ b_i = \mathbf{c}$, for all b_i in I
$\mathbf{r} \leq \mathbf{r} \sqcup \mathrm{I}$	$M\ b_i = \mathbf{c}$, for all b_i in I
$b \leq \mathbf{r} \sqcup \mathrm{I}$	$\begin{cases} M\ b = \mathbf{c}, \text{ if I contains no } \mathbf{r} \\ M\ b = \mathbf{r}, \text{ otherwise} \end{cases}$
	and $M\ b_i = \mathbf{c}$ for all b_i in I
$\mathbf{r} \leq b \sqcup \mathrm{I}$	$M\ b = \mathbf{c}$ and $M\ b_i = \mathbf{c}$ for all b_i in I
$\mathbf{c} \leq b \sqcup \mathrm{I}$	$M\ b = \mathbf{c}$ and $M\ b_i = \mathbf{c}$ for all b_i in I
$b_2 \leq b_1 \sqcup \mathrm{I}$	$M\ b_1 = \mathbf{c}$, $M\ b_2 = \mathbf{c}$, and $M\ b_i = \mathbf{c}$ for all b_i in I

Table 5.4: Minimal Solutions to the Constraints

not affected (`N`)	forces (`F`)	affected (`A`)
$\mathbf{r} = \mathbf{c}$	$b = \mathbf{r}$	$b = \mathbf{c}$
$\mathbf{c} = \mathbf{r}$	$\mathbf{r} = b$	$\mathbf{c} = b$
$\mathbf{c} \leq \mathbf{r}$	$b \leq \mathbf{r}$	$\mathbf{c} \leq b$
$\mathbf{r} = \mathbf{r}$		$b_1 = b_2$
$\mathbf{c} = \mathbf{c}$		$b_1 \leq b_2$
$\mathbf{r} \leq \mathbf{r}$		$\mathbf{c} \leq b \sqcup \mathrm{I}$
$\mathbf{r} \leq \mathbf{c}$		$b_2 \leq b_1 \sqcup \mathrm{I}$
$\mathbf{c} \leq \mathbf{c}$		
$\mathbf{r} \leq b$		
$b \leq \mathbf{c}$		
$\mathbf{r} \leq \mathbf{c} \sqcup \mathrm{I}$		
$\mathbf{c} \leq \mathbf{c} \sqcup \mathrm{I}$		
$b \leq \mathbf{c} \sqcup \mathrm{I}$		
$\mathbf{r} \leq \mathbf{r} \sqcup \mathrm{I}$		
$\mathbf{r} \leq b \sqcup \mathrm{I}$		

Table 5.5: The Three Groups of Constraints

$$
\begin{array}{lcl}
\textsc{Exp}\ ([\mathbf{r},\ \mathrm{I}],\ b) & = & [b \leq \mathbf{r},\ \textsc{Exp}\ (\mathrm{I},\ b)] \\
\textsc{Exp}\ ([\mathbf{c},\ \mathrm{I}],\ b) & = & [b \leq \mathbf{c},\ \textsc{Exp}\ (\mathrm{I},\ b)] \\
\textsc{Exp}\ ([b_1,\ \mathrm{I}],\ b) & = & [b \leq b_1,\ \textsc{Exp}\ (\mathrm{I},\ b)] \\
\textsc{Exp}\ ([\ ],\ b) & = & [\ ] \\
\\
\textsc{Exp}\ ([\mathbf{r},\ \mathrm{I}],\ \mathbf{c}) & = & \texttt{FAIL} \\
\textsc{Exp}\ ([\mathbf{c},\ \mathrm{I}],\ \mathbf{c}) & = & [\mathbf{c} \leq \mathbf{c},\ \textsc{Exp}\ (\mathrm{I},\ \mathbf{c})] \\
\textsc{Exp}\ ([b,\ \mathrm{I}],\ \mathbf{c}) & = & [\mathbf{c} \leq b,\ \textsc{Exp}\ (\mathrm{I},\ \mathbf{c})] \\
\textsc{Exp}\ ([\ ],\ \mathbf{c}) & = & [\ ]
\end{array}
$$

Figure 5.15: The Function Exp

If an unsolvable constraint ($\mathbf{r} = \mathbf{c}$, $\mathbf{c} = \mathbf{r}$, $\mathbf{c} \leq \mathbf{r}$) is encountered while solving a set of constraints then we know that the set of constraints is unsolvable so we can stop searching for a solution.

The constraints $b \leq \mathbf{r} \sqcup \mathrm{I}$ and $\mathbf{c} \leq \mathbf{r} \sqcup \mathrm{I}$ have to be expanded into the constraints $b \leq \mathrm{I}$ and $\mathbf{c} \leq \mathrm{I}$ respectively, and then they can be distributed into the three groups. This part of the algorithm is done by the function EXP (Figure 5.15).

Fact 5.26 expresses that it is safe to expand the constraints $[b \leq \mathbf{r} \sqcup \mathrm{I}]$ into $[b \leq \mathrm{I}]$ and $[\mathbf{c} \leq \mathrm{I}]$. This is exactly what the function EXP does.

Fact 5.26
M solves $[b \leq \mathbf{r} \sqcup \mathrm{I}] \Leftrightarrow M$ solves $[b \leq \mathrm{I}]$ □

Proof Since we have $\mathbf{r} \sqcup b = b$ for all b (i.e. b is $\mathbf{r}$, $\mathbf{c}$, or a binding time variable) it must be the case that whenever $[b_2 \leq \mathbf{r} \sqcup b]$ is solvable by M, then so is $[b_2 \leq b]$ and visa versa. ■

The first part of the algorithm (the devision of the constraints into the three groups) is done by the function DIV (Figure 5.16). Let $M_{\mathbf{c}}$ and $M_{\mathbf{r}}$ be the mappings that maps all binding time variables to $\mathbf{c}$ and $\mathbf{r}$, respectively. Fact 5.27 expresses that the function DIV divides the constraints into the three groups described in Table 5.5:

Fact 5.27
If (`N`, `F`, `A`) = DIV (C, [ ], [ ], [ ]) then

- `N` is solvable by both $M_{\mathbf{c}}$ and $M_{\mathbf{r}}$.
- `F` is solvable by $M_{\mathbf{r}}$ and *not* by $M_{\mathbf{c}}$.
- `A` is solvable by $M_{\mathbf{c}}$.

□

Proof It is easily seen by inspection of Table 5.5 that all constraints in the first column are solvable by both $M_{\mathbf{c}}$ and $M_{\mathbf{r}}$, and that all the constraints in the the second column are solvable by $M_{\mathbf{r}}$ only. Finally, all the constraints in the last column are solvable by $M_{\mathbf{c}}$ and maybe by $M_{\mathbf{r}}$.

Now by inspection of the algorithm DIV we can see that the constraints are put into the right groups. Hence the Fact holds. ■

Next we have to find the variables that have to be mapped to $\mathbf{r}$. This part of the algorithm is done by the function FORCER (Figure 5.17). The function is also defined for the constraints $\mathbf{r} = \mathbf{r}$ and $\mathbf{r} \leq \mathbf{r}$ because the intermediate solution is applied to the rest of the constraints as soon as it is found.

$$
\begin{array}{lcl}
\textsc{Div}\ ([\,], \mathtt{N}, \mathtt{F}, \mathtt{A}) & = & (\mathtt{N}, \mathtt{F}, \mathtt{A}) \\
\textsc{Div}\ ([\mathbf{c} = \mathbf{c}, \mathrm{C}], \mathtt{N}, \mathtt{F}, \mathtt{A}) & = & \textsc{Div}\ (\mathrm{C}, [\mathbf{c} = \mathbf{c}, \mathtt{N}], \mathtt{F}, \mathtt{A}) \\
\textsc{Div}\ ([\mathbf{c} = \mathbf{r}, \mathrm{C}], \mathtt{N}, \mathtt{F}, \mathtt{A}) & = & \mathtt{FAIL} \\
\textsc{Div}\ ([\mathbf{r} = \mathbf{c}, \mathrm{C}], \mathtt{N}, \mathtt{F}, \mathtt{A}) & = & \mathtt{FAIL} \\
\textsc{Div}\ ([\mathbf{r} = \mathbf{r}, \mathrm{C}], \mathtt{N}, \mathtt{F}, \mathtt{A}) & = & \textsc{Div}\ (\mathrm{C}, [\mathbf{r} = \mathbf{r}, \mathtt{N}], \mathtt{F}, \mathtt{A}) \\
\textsc{Div}\ ([\mathbf{r} \leq \mathbf{r}, \mathrm{C}], \mathtt{N}, \mathtt{F}, \mathtt{A}) & = & \textsc{Div}\ (\mathrm{C}, [\mathbf{r} \leq \mathbf{r}, \mathtt{N}], \mathtt{F}, \mathtt{A}) \\
\textsc{Div}\ ([\mathbf{c} \leq \mathbf{r}, \mathrm{C}], \mathtt{N}, \mathtt{F}, \mathtt{A}) & = & \mathtt{FAIL} \\
\textsc{Div}\ ([\mathbf{r} \leq \mathbf{c}, \mathrm{C}], \mathtt{N}, \mathtt{F}, \mathtt{A}) & = & \textsc{Div}\ (\mathrm{C}, [\mathbf{r} \leq \mathbf{c}, \mathtt{N}], \mathtt{F}, \mathtt{A}) \\
\textsc{Div}\ ([\mathbf{c} \leq \mathbf{c}, \mathrm{C}], \mathtt{N}, \mathtt{F}, \mathtt{A}) & = & \textsc{Div}\ (\mathrm{C}, [\mathbf{c} \leq \mathbf{c}, \mathtt{N}], \mathtt{F}, \mathtt{A}) \\
\textsc{Div}\ ([\mathbf{r} \leq b, \mathrm{C}], \mathtt{N}, \mathtt{F}, \mathtt{A}) & = & \textsc{Div}\ (\mathrm{C}, [\mathbf{r} \leq b, \mathtt{N}], \mathtt{F}, \mathtt{A}) \\
\textsc{Div}\ ([b \leq \mathbf{c}, \mathrm{C}], \mathtt{N}, \mathtt{F}, \mathtt{A}) & = & \textsc{Div}\ (\mathrm{C}, [b \leq \mathbf{c}, \mathtt{N}], \mathtt{F}, \mathtt{A}) \\
\textsc{Div}\ ([\mathbf{r} \leq \mathbf{c} \sqcup \mathrm{I}, \mathrm{C}], \mathtt{N}, \mathtt{F}, \mathtt{A}) & = & \textsc{Div}\ (\mathrm{C}, [\mathbf{c} \sqcup \mathrm{I} \geq \mathbf{r}, \mathtt{N}], \mathtt{F}, \mathtt{A}) \\
\textsc{Div}\ ([\mathbf{c} \leq \mathbf{c} \sqcup \mathrm{I}, \mathrm{C}], \mathtt{N}, \mathtt{F}, \mathtt{A}) & = & \textsc{Div}\ (\mathrm{C}, [\mathbf{c} \sqcup \mathrm{I} \geq \mathbf{c}, \mathtt{N}], \mathtt{F}, \mathtt{A}) \\
\textsc{Div}\ ([\mathbf{r} \leq \mathbf{r} \sqcup \mathrm{I}, \mathrm{C}], \mathtt{N}, \mathtt{F}, \mathtt{A}) & = & \textsc{Div}\ (\mathrm{C}, [\mathbf{r} \sqcup \mathrm{I} \geq \mathbf{r}, \mathtt{N}], \mathtt{F}, \mathtt{A}) \\
\textsc{Div}\ ([b \leq \mathbf{c} \sqcup \mathrm{I}, \mathrm{C}], \mathtt{N}, \mathtt{F}, \mathtt{A}) & = & \textsc{Div}\ (\mathrm{C}, [\mathbf{c} \sqcup \mathrm{I} \geq b, \mathtt{N}], \mathtt{F}, \mathtt{A}) \\
\textsc{Div}\ ([\mathbf{r} \leq b \sqcup \mathrm{I}, \mathrm{C}], \mathtt{N}, \mathtt{F}, \mathtt{A}) & = & \textsc{Div}\ (\mathrm{C}, [b \sqcup \mathrm{I} \geq \mathbf{r}, \mathtt{N}], \mathtt{F}, \mathtt{A}) \\
\textsc{Div}\ ([b \leq \mathbf{r} \sqcup \mathrm{I}, \mathrm{C}], \mathtt{N}, \mathtt{F}, \mathtt{A}) & = & \textsc{Div}\ ([\textsc{Exp}\ (\mathrm{I}, b), \mathrm{C}]), \mathtt{N}, \mathtt{F}, \mathtt{A}) \\
\textsc{Div}\ ([b = \mathbf{r}, \mathrm{C}], \mathtt{N}, \mathtt{F}, \mathtt{A}) & = & \textsc{Div}\ (\mathrm{C}, \mathtt{N}, [b = \mathbf{r}, \mathtt{F}], \mathtt{A}) \\
\textsc{Div}\ ([\mathbf{r} = b, \mathrm{C}], \mathtt{N}, \mathtt{F}, \mathtt{A}) & = & \textsc{Div}\ (\mathrm{C}, \mathtt{N}, [\mathbf{r} = b, \mathtt{F}], \mathtt{A}) \\
\textsc{Div}\ ([b \leq \mathbf{r}, \mathrm{C}], \mathtt{N}, \mathtt{F}, \mathtt{A}) & = & \textsc{Div}\ (\mathrm{C}, \mathtt{N}, [b \leq \mathbf{r}, \mathtt{F}], \mathtt{A}) \\
\textsc{Div}\ ([b = \mathbf{c}, \mathrm{C}], \mathtt{N}, \mathtt{F}, \mathtt{A}) & = & \textsc{Div}\ (\mathrm{C}, \mathtt{N}, \mathtt{F}, [b = \mathbf{c}, \mathtt{A}]) \\
\textsc{Div}\ ([\mathbf{c} = b, \mathrm{C}], \mathtt{N}, \mathtt{F}, \mathtt{A}) & = & \textsc{Div}\ (\mathrm{C}, \mathtt{N}, \mathtt{F}, [\mathbf{c} = b, \mathtt{A}]) \\
\textsc{Div}\ ([b_1 = b_2, \mathrm{C}], \mathtt{N}, \mathtt{F}, \mathtt{A}) & = & \textsc{Div}\ (\mathrm{C}, \mathtt{N}, \mathtt{F}, [b_1 = b_2, \mathtt{A}])) \\
\textsc{Div}\ ([\mathbf{c} \leq b, \mathrm{C}], \mathtt{N}, \mathtt{F}, \mathtt{A}) & = & \textsc{Div}\ (\mathrm{C}, \mathtt{N}, \mathtt{F}, [\mathbf{c} \leq b, \mathtt{A}]) \\
\textsc{Div}\ ([b_1 \leq b_2, \mathrm{C}], \mathtt{N}, \mathtt{F}, \mathtt{A}) & = & \textsc{Div}\ (\mathrm{C}, \mathtt{N}, \mathtt{F}, [b_1 \leq b_2, \mathtt{A}]) \\
\textsc{Div}\ ([\mathbf{c} \leq \mathbf{r} \sqcup \mathrm{I}, \mathrm{C}], \mathtt{N}, \mathtt{F}, \mathtt{A}) & = & \textsc{Div}\ ([\textsc{Exp}\ (\mathrm{I}, \mathbf{c}), \mathrm{C}], \mathtt{N}, \mathtt{F}, \mathtt{A}) \\
\textsc{Div}\ ([\mathbf{c} \leq b \sqcup \mathrm{I}, \mathrm{C}], \mathtt{N}, \mathtt{F}, \mathtt{A}) & = & \textsc{Div}\ (\mathrm{C}, \mathtt{N}, \mathtt{F}, [\mathbf{c} \leq b \sqcup \mathrm{I}, \mathtt{A}]) \\
\textsc{Div}\ ([b_2 \leq b_1 \sqcup \mathrm{I}, \mathrm{C}], \mathtt{N}, \mathtt{F}, \mathtt{A}) & = & \textsc{Div}\ (\mathrm{C}, \mathtt{N}, \mathtt{F}, [b_2 \leq b_1 \sqcup \mathrm{I}, \mathtt{A}])
\end{array}
$$

Figure 5.16: The function Div

$$
\begin{array}{lcl}
\textsc{Force}\mathrm{R}\ ([\,],\ M) & = & M \\
\textsc{Force}\mathrm{R}\ ([b = \mathbf{r},\ \mathrm{C}],\ M) & = & \textsc{Force}\mathrm{R}\ ([b/\mathbf{r}]\mathrm{C},\ [b/\mathbf{r}] \circ M) \\
\textsc{Force}\mathrm{R}\ ([\mathbf{r} = b,\ \mathrm{C}],\ M) & = & \textsc{Force}\mathrm{R}\ ([b/\mathbf{r}]\mathrm{C},\ [b/\mathbf{r}] \circ M) \\
\textsc{Force}\mathrm{R}\ ([b \leq \mathbf{r},\ \mathrm{C}],\ M) & = & \textsc{Force}\mathrm{R}\ ([b/\mathbf{r}]\mathrm{C},\ [b/\mathbf{r}] \circ M) \\
\textsc{Force}\mathrm{R}\ ([\mathbf{r} = \mathbf{r},\ \mathrm{C}],\ M) & = & \textsc{Force}\mathrm{R}\ (\mathrm{C},\ M) \\
\textsc{Force}\mathrm{R}\ ([\mathbf{r} \leq \mathbf{r},\ \mathrm{C}],\ M) & = & \textsc{Force}\mathrm{R}\ (\mathrm{C},\ M)
\end{array}
$$

Figure 5.17: The Function FORCER

The next lemma expresses that the function FORCER extends the solution M which solves C to also solve a set of constraints C′. The constraints C′ all belongs to the group that forces some binding time variable to be mapped to **r** (see Table 5.5).

Lemma 5.28
If M solves C, and C and `F` has no variables in common, and

$$M' = \textsc{Force}\mathrm{R}(\mathtt{F}, M)$$

then M' solves [C, `F`]. □

Proof We assume that M solves C and that C and `F` has no variables in common. Then we prove by induction on the size of the set of constraints `F` that $M' = \textsc{Force}\mathrm{R}\ (\mathtt{F},\ M)$ solves [C, `F`].

For the details see Appendix page 406. ■

Now all the pieces are put together by the function SOLVE′ (Figure 5.18). It first divides the constraints into the three groups `N`, `F`, and `A` by applying the function DIV to the constraints. Next we apply FORCER to the constraints in `F` to the for the present solution M'. Finally we apply SOLVE′ to the constraints $M'\mathtt{A}$ and M'. The operation "+" in Figure 5.18 is defined as follows

$$(M_1 \text{“}+\text{''}\, M_2)b \;=\; \begin{cases} M_2 b, \text{if } M_2 b \text{ is defined} \\ M_1 b, \text{otherwise} \end{cases}$$

The result is that all the binding time variable not forced to be mapped to **r** are mapped to **c**.

$$
\begin{array}{lcll}
\textsc{Solve}'\ ([\,],\ M) & = & M & \\
\textsc{Solve}'\ (\mathrm{C},\ M) & = & \text{let} & (\mathtt{N},\ \mathtt{F},\ \mathtt{A}) = \textsc{Div}\ (\mathrm{C},\ [\,],\ [\,],\ [\,]) \\
 & & & M' = \textsc{ForceR}\ (\mathtt{F},\ M) \\
 & & \text{in} & \text{if } \mathtt{F} = [\,] \text{ then} \\
 & & & \quad M' \\
 & & & \text{else} \\
 & & & \quad \textsc{Solve}'\ (M'\mathtt{A},\ M') \\
 & & & \\
\textsc{Solve}\ \mathrm{C} & = & M_{\mathbf{C}}\ \text{“+”}\ \textsc{Solve}'\ (\mathrm{C},\ \mathit{undef}) &
\end{array}
$$

Figure 5.18: The Functions SOLVE and SOLVE′

Fact 5.29 expresses that the functions FORCER and SOLVE′ extend the mapping M to also solve the constraints C and it does not forget the solutions so far:

Fact 5.29
Suppose that $M\mathrm{C} = \mathrm{C}$, i.e. M is not defined for any of the binding time variables in C.

Whenever $M' = \textsc{ForceR}\ (\mathrm{C}, M)$, then $(M\ b_1 = b_2) \Rightarrow (M'\ b_1 = b_2)$ and if $M' = \textsc{Solve}'\ (\mathrm{C}, M)$, then $(M\ b_1 = b_2) \Rightarrow (M'\ b_1 = b_2)$. □

Proof Assume that M is not defined for any of the binding time variables in C. In all clauses of FORCER and SOLVE′ either M itself is returned or $M' \circ M$ where M' does not involve binding time variables in M. Therefore it must be the case that $(M\ b_1 = b_2) \Rightarrow (M''\ b_1 = b_2)$ where M'' is either FORCER (C, M) or SOLVE′ (C, M). ■

Lemma 5.30
If M solves C and

$$M' = \textsc{Solve}'(C', M)$$

and C and C′ has no variables in common, then $M_{\mathbf{C}}$ “+” M' solves [C, C′]. □

Proof We assume that M solves C and that C and C′ has no variables in common. We prove by induction on the size of C′ that $M_{\mathbf{C}}$ “+” M' solves

[C, C′], where

$$M' = \textsc{Solve}'(C', M)$$

For the details see Appendix page 407. ■

The solution to the constraints C are found by Solve defined in Figure 5.18.

We can always map the rest of the binding time variables to **c**:

Fact 5.31
If M solves C then $M_{\mathbf{c}}$ "+" M solves C. □

Proof Assume that M solves C, hence for all binding time variables, b, in C we have that Mb is defined and hence

$$M_{\mathbf{c}}\text{“}+\text{''}\, Mb \;=\; Mb$$

and therefore $M_{\mathbf{c}}$ "+" M will also solve C as required. ■

We have Theorem 5.32 which follows directly from Lemma 5.30:

Theorem 5.32
If $M = \textsc{Solve}$ C then M solves C. □

Now we can put is all together:

Theorem 5.33 *Soundness of the Algorithm with respect to* $\vdash_0$
Assume that S_0 is a ground substitution and

$$\begin{array}{rcl} \mathcal{L}(\mathtt{e}) & = & (\mathrm{A} : \mathrm{I}, \mathtt{e}', \mathtt{t}, b, \mathrm{C}, \mathrm{P}, \mathrm{P}_2) \\ \mathrm{C}' & = & [S_0\mathrm{C}, \mathcal{UD}(S_0\mathrm{P}), \mathcal{W}(S_0\mathrm{P}_2)] \\ \textsc{Solve}(\mathrm{C}') & = & M \end{array}$$

then

$$\mathit{tenv} \vdash_0 (M \circ S_0)\mathtt{e}' \,:\, (M \circ S_0)\mathtt{t} \,:\, (M \circ S_0)b$$

where

$$\mathit{tenv}\ \mathtt{x} \;=\; ((M \circ S_0)\mathtt{t}', (M \circ S_0)p), \text{if } \mathtt{x} : \mathtt{t}' : p \in \mathrm{A} : \mathrm{I}$$

□

Proof Follows from Theorem 5.32, 5.23 and 5.18. ■

To get completeness of the algorithm with respect to $\vdash_0$ we need completeness of the algorithm solving the constraints:

Conjecture 5.34
Whenever C is solvable, then

$$\textsc{Solve}(\mathrm{C}) \neq \mathtt{FAIL}$$

□

Now completeness is:

Conjecture 5.35 *Completeness of the Algorithm with respect to* $\vdash_0$
Assume

$$tenv \vdash_0 \mathtt{e} : \mathtt{t} : b$$

and

$$\begin{array}{rcl} \mathcal{L}(\mathtt{e}) & = & (\mathrm{A}' : \mathrm{I}', \mathtt{e}', \mathtt{t}', b', \mathrm{C}', \mathrm{P}, \mathrm{P}_2) \\ \mathrm{C}'' & = & [S\mathrm{C}, \mathcal{UD}(S\mathrm{P}), \mathcal{W}(S\mathrm{P}_2)] \end{array}$$

then

$$\textsc{Solve}(\mathrm{C}'') \neq \mathtt{FAIL}$$

□

Proof Follows from Conjecture 5.34 and Theorem 5.24 and 5.18. ■

5.6 Summary

In this Chapter we have taken an inference system for binding time analysis (Figure 5.1 and Figure 5.2) and reformulated it (Figure 5.4) as an annotated type system. Then the rules [up] and [down] are eliminated so we end up with a purely logical inference system (Figure 5.8) — all the rules are structural in the term. From this system an algorithm for binding time analysis (Figure 5.13, 5.14 and 5.18) is constructed. One of the salient features of the algorithm presented here (Figure 5.13 and 5.14) is the use of substitution to infer the type in case of function application; this is contrary to [NN92] where extra recursive call are necessary.

Clearly the algorithm $\mathcal{L}$ for collecting the constraints terminates since we go through the term in a structural way. The algorithms EXP, FORCER, and DIV terminates since they just step through a list. In the algorithm SOLVE′ if the list `F` is empty the algorithm terminates, otherwise we do a recursive call on the list `A` which is smaller than the original list, hence the algorithm terminates.

The complexity of going though the term finding the constraints is linear in the size of the term, however the number of constraints for a given term is $\mathcal{O}(n^2)$ where n is the size of the term, hence the complexity of finding the constraints must be $\mathcal{O}(n^2)$. The complexity for solving the constraints is quadratic in the number of constraints — so the whole binding time analysis is $\mathcal{O}(n^4)$ where n is the size of the term. The algorithm described here for binding time analysis is faster than the one presented in [NN92]. We conjecture that it possible to find an algorithm to solve the constraints in $\mathcal{O}(n \log n)$. So that the binding time analysis becomes an $\mathcal{O}(n^2 \log n)$ algorithm.

The work of this Chapter is inspired by the work of Wadler [Wad91] where an inference system for linear types using "use" types is presented. The types is annotated with "uses". Then the rules [dereliction] and [promotion] are eliminated and an algorithm for finding constraints is constructed from the resulting system.

A somewhat related approach is that of Henglein [Hen91]. Here binding time analysis is also performed via constraints and there is a discussion of efficient algorithms for their solution. A type of [Hen91] is either the type constant B, denoting "static" (compile-time) base values, the type constant Λ, representing all unevaluated (run-time) terms, a function type $\tau_1 \rightarrow \tau_2$, where τ_1 and τ_2 are types, represents a higher-order value, or a type variable α. There is no structure on the dynamic (compile-time) values, there is therefore only one kind of function type.

Another difference between the inference system in Figure 5.2 and that of [Hen91] is that when the [down]-rule is applied it is explicitly marked in the term. There are no analogous to the [up]-rule. In the inference system of [Hen91] only base values can be made dynamic (run-time object) by application of the lift-operator, and *not* terms of function type.

The constraints of [Hen91] are between types, whereas the constraints in this paper are between binding times (binding time values (`r`, `c`) and binding time variables). To every λ-bound variable `x` in the term `e` there is associated a type variable $\alpha_{\mathtt{x}}$, and to every term $\mathtt{e}'$ occurring in `e` two type variables $\alpha_{\mathtt{e}'}$ and $\overline{\alpha_{\mathtt{e}'}}$. The idea of $\overline{\alpha_{\mathtt{e}'}}$ is the type (binding time) of $\mathtt{e}'$ if it is lifted. Now constraints are made between all these type variables to express the same as the inference system. The constraints are solved by first normalising them and then finding a minimal solution.

An implementation in Miranda of our algorithm is presented in the Appendix page 409.

Chapter 6

Uniform PERs and Comportment Analysis

The analysis in this Chapter is specified by abstract interpretations using names of uniform PERs. This approach differs from the one used in the previous chapters. The analysis will capture both strictness and totality properties in additions to constancy. The semantically foundation is slightly different from the other chapters.

6.1 Introduction

Strictness properties are properties of functions between domains; in principle they are intended to capture the notion of how the function reacts to changes in the definedness of its arguments rather than changes between incomparable values of its arguments. A comparison to the notion of partial differentiation may prove fruitful. Since many properties conforming to this notion (e.g. totality) exclude $\perp$, we use the word *comportment*[1] *property* as coined by Cousot and Cousot [CC94] to avoid abusing the epithet "strictness property" by allowing it to encompass totality.

For many years there were two forms of strictness property, the ideal-based [Myc80, BHA86, EM91, CC94] form and the projection-based form [WH87, Hun91]. Given domains D and E and a continuous function $f : D \to E$ (the denotational semantics of a program function) these properties can be summarised as follows.

[1]The Collins Dictionary: **comport:** *vb.* **1.** (tr.) to conduct or bear (oneself) in a specified way. **2.** (*intr.*; foll. by with) to agree (with). **comportment** *n*

Let I range over the *ideals* of D (non-empty Scott-closed sets) and J over those of E. The ideal-based strictness properties are those of the form

$$\models^{\text{ideal}} f : W_{I,J} \Leftrightarrow f(I) \subseteq J \tag{6.1}$$

Similarly, let α range over projections on D (continuous, idempotent functions such that $\alpha(x) \sqsubseteq x$) and β over those on E. The projection-based strictness properties are those of the form

$$\models^{\text{proj}} f : W_{\alpha,\beta} \Leftrightarrow \beta \circ f = \beta \circ f \circ \alpha \tag{6.2}$$

or equivalently of the form

$$\models^{\text{proj}} f : W_{\alpha,\beta} \Leftrightarrow \beta \circ f \sqsupseteq f \circ \alpha$$

Hunt [Hun91] observed that PER-based properties generalise both the above forms. (Actually Hunt only considered strict and inductive PER-properties which suffice for his generalisation but we relax this so as to be able to encompass as many comportment properties as possible.)

A PER P on D is a relation on D which is symmetric and transitive. It is inductive if it contains limits of chains when seen as a subset of $D \times D$. Such a PER P defines a property W_P on D given by its diagonal

$$\models^{\text{PER}} d : W_P \Leftrightarrow d \in |P|$$

where

$$|P| = \{d \mid dPd\}$$

Hunt essentially defines the PER-properties of functions in two stages, first defining the *basic* PER-properties and then the PER-properties as the conjunctive completion of these. Letting P range over (a class of, see below) PERs of D and Q over (a class of) PERs of E, the basic PER-properties appear as

$$\models^{\text{basic PER}} f : W_{P,Q} \Leftrightarrow (\forall x, y \in D)(xPy \Rightarrow (fx)Q(fy))$$

Conjunctions (intersections) of these give PER-properties.

The reader familiar with the typical UK-Danish presentations of abstract interpretations will here note an absence of specifying *how* the properties at higher-order are related to those at lower-order (or first-order). Indeed, Cousot and Cousot [CC94] argue that the framework of abstract interpretation should remain neutral on this and applications should select representations for program properties at each type independently, all that is required is that function properties, ranged over by C, should be in Galois correspondence $\gamma : C \to \mathcal{P}(D \to E)$ where $\mathcal{P}(\cdot)$ is the power-*set* construct. This yields properties

$$\models^{\text{Galois}} f : W_C \Leftrightarrow f \in \gamma(C)$$

We take the opposite view in this paper, that it is fundamental to define a set (or range of sets) of properties for each type rather than allowing parasitic applications in which the *space of* properties is not even decidable. After all, restrictions of media have historically spawned great art.

One can, perhaps over-simplistically, observe that the French (or at least [CC94]) approach is platonic in that it prescribes a framework and then searches for special cases which explain or yield various analyses. Similarly the UK-Danish approach has generally been constructivist: abstract spaces have been crafted which explain exactly the range of phenomena at hand; abstract spaces for higher types are expected to be derivable from those at lower types (cf. intuitionistic implication). The old arguments transfer beautifully: the Platonist argues that the constructivist is doomed to extend his constructions each time the world demands (and never models it perfectly); the constructivist chides the generality of the Platonist for allowing parasitic solutions which have no application or physical reality.[2]

The present construction of *uniform* PERs, both at ground type and hereditarily at higher types, is designed to capture as many comportment properties as possible (hopefully all) in a constructive manner. It was developed concurrently with [CC94] and attempts to capture comportments from the constructive, as opposed to platonic, viewpoint. Interestingly the two approaches differ (ours yields four extra properties at type $\texttt{Int} \to \texttt{Int}$) but the reasons for this are not yet clear.

The range of comportment properties expressible by uniform PERs include

[2]There is more programming analogy here: the Platonist is the *program specifier* who constructs the framework containing A and B; the constructivist is the *program writer* who needs to consider whether obtaining A from B is possible in practice. There is an amusing analogy (thanks to Thomas Jensen for this) which links back to abstract interpretation in that Platonist/specification essentially forms the greatest fixpoint (ban only what requirements forbid) whereas constructivism/programming forms the least fixpoint (allow only what the requirements necessitate).

not only strictness properties but also totality properties. These latter are also captured by the annotated type system of Chapter 2 and 3.

Overview In Section 6.2 we recall the definition of PERs and we define both the subset ordering and the Egli-Milner ordering on PERs.

In Section 6.3 we define the notion of uniform PER on the integers. These PERs are uniform in the sense that they treat all the integers identically (as in [EM91]). We observe that some of the uniform PERs on the integers are *not* strict. Far from being a problem these non-strict PERs can describe the property of being a non-bottom value (or at higher types being a total function) in contrast to most work on ideal- or projection-based program analysis (subject to the unsurprising need to use the Egli-Milner ordering on PERs which we define). The empty PER also appears as a uniform PER but is not proscribed because of its possible use to represent some sort of dead code. Next we form following Hunt the PERs on the function space and again we observe that some of the PERs are not strict.

In Section 6.4 we take a closer look at the uniform PERs on $\mathbb{Z}_\perp \to \mathbb{Z}_\perp$. It appears that we are able to express all the properties that are expressible with ideal-based [EM91, Myc81], projection-based [WH87], and non-standard [Ben93, SNN94] program analysis. Some new properties also emerge, e.g. the property of being constant on the integers, the property of being non-bottom on the integers, the property of being constant and non-bottom on the integers, the property of being non-bottom on the integer *or* the bottom-function.

For each property there exists an optimisation that can be applied to the code implementing an expression with that property. So again we can see the advantage of also considering the non-strict PERs. Moreover for each property there exists a function which has that property as the best/most exact description of the expression.

The next step is to define the language in Section 6.5, its standard semantics, and a non-standard semantics for program analysis. We define the semantics and the program analysis as two different interpretations of the language (as in [Hun91]). Following Hunt's proof of correctness of the analysis we show that the standard and non-standard semantics of terms satisfy a ternary logical relation which relates pairs of standard values with an abstract value.

The final step is to define the abstract-denotations of the constants, e.g. *if*, *fix*, $+$, such that their standard-denotation is related to the abstract-denotation. The problems here is mostly for *fix*. Not all the PERs we are dealing with are strict therefore we cannot just follow Hunt using a union operator to form the fixpoint. (We cannot just start with the least PER in the subset ordering which is the empty PER and apply the functional to it. The result would be the empty PER and hence the fixpoint would be the empty PER.) Our solution is to start with the least strict PER and a form of the Egli-Milner-ordering which we define on PERs.

6.2 Formalism

We start by defining the notion of a PER on a set and then consider the possible orderings on PERs when the sets has order-structure:

6.2.1 PERs on Domains

Recall that a PER on a set S is a relation on S that is *symmetric* and *transitive*. Both the domain and range of the PER is equal to the diagonal-part of P, defined by $|P| = \{x \mid (x, x) \in P\}$. For a given set A of PERs, the *properties* associated with A are the set of diagonals of members of A. PERs can be ordered in at least two different ways: the subset ordering and the Egli-Milner ordering.

Definition 6.1 *The subset ordering*
The PER P is less than or equal to the PER Q, written $P \leq Q$, if $P \subseteq Q$
$\square$

6.2.2 The Egli-Milner Ordering

The subset ordering does not take into account the structure of the domain on which the PERs are built. The least PER in the subset ordering is the empty PER. The result of applying a functional to the empty PER is the empty PER, therefore the fixpoint of a functional is the empty PER. What we want to do is to start with the least, strict, PER. But starting there may not yield a chain under the subset ordering. The obvious choice is to use the Egli-Milner ordering.

Let D be a cpo. We define the Egli-Milner ordering on the space of PERs over D by treating the PERs as subsets of $D \times D$.

Definition 6.2 *The Egli-Milner ordering on PERs over D*
Let P and Q be PERs over D. We define $P \sqsubseteq^{\mathrm{PER}}_{\mathrm{EM}} Q$ if $P \sqsubseteq Q$ when considered as subsets of $D \times D$, i.e.

$$(\forall p \in P \exists q \in Q : p \sqsubseteq_{D\times D} q) \wedge (\forall q \in Q \exists p \in P : p \sqsubseteq_{D\times D} q)$$

□

We also define a PER P being strict and downwards closed by inheritance from $D \times D$:

Definition 6.3 *strict and downwards closed* PERs
A PER P on D is strict if $(\perp_D, \perp_D) \in P$; it is downwards closed whenever P seen as a subset of $D \times D$ is downwards closed, i.e.

$$((d_1, d_2) \sqsubseteq (d_3, d_4) \wedge (d_3, d_4) \in P) \Rightarrow (d_1, d_2) \in P$$

□

For strict PERs (i.e. Hunt's work) we want the subset ordering to coincide with the Egli-Milner ordering:

Lemma 6.4
For all *strict* PERs P on D and for all *downwards closed* PERs Q on D:

$$P \sqsubseteq^{\mathrm{PER}}_{\mathrm{EM}} Q \Leftrightarrow P \leq Q$$

□

Proof We assume P is a strict PER on D and that Q is a downwards closed PER on D.

First we assume $P \sqsubseteq^{\mathrm{PER}}_{\mathrm{EM}} Q$ and show $P \leq Q$. We have $(x, y) \in P$ then from Definition 6.2 part one we get that there exits $(x', y') \in Q$ such that $(x, y) \leq (x', y')$. Since Q is downwards closed it can be the case that $x' = x$ and $y' = y$. Hence we have that P is a subset of Q.

Next assume $P \leq Q$ and show $P \sqsubseteq^{\mathrm{PER}}_{\mathrm{EM}} Q$. Since P is a subset of Q then for all $(x, y) \in P$ we have $(x, y) \in Q$ such that $(x, y) \leq (x, y)$ which is the first part of Definition 6.2.

We have $(\bot_D, \bot_D) \in P$ since P is strict. Therefore for all $(x, y) \in Q$ we have $(\bot_D, \bot_D) \leq (x, y)$, which is the second part of Definition 6.2. ■

6.3 Uniform PERs on Types

Although the intuition is to define a class of uniform PERs associated with a *domain* it turns out to be more natural to define the classes of uniform PERs associated with (the standard interpretation of) a *type*. (For the purposes of abstract interpretation representations of these PERs can be used as abstract values and it is too restrictive to insist that (accidentally) isomorphic domains described by different types should have identical sets of abstract values). We will continue to refer to (e.g.) "the set of uniform PERs on $\mathbb{Z}_\bot$" when this reads better.

We start with a set standard types:

$$\mathtt{t} ::= \mathtt{Int} \mid \mathtt{t} \rightarrow \mathtt{t} \mid \mathtt{t} \times \mathtt{t}$$

For each type $\mathtt{t}$ there is a standard domain $D^{\mathrm{S}}_{\mathtt{t}}$:

$$\begin{aligned} D^{\mathrm{S}}_{\mathtt{Int}} &= \mathbb{Z}_\bot \\ D^{\mathrm{S}}_{\mathtt{t}_1 \rightarrow \mathtt{t}_2} &= D^{\mathrm{S}}_{\mathtt{t}_1} \rightarrow D^{\mathrm{S}}_{\mathtt{t}_2} \\ D^{\mathrm{S}}_{\mathtt{t}_1 \times \mathtt{t}_2} &= D^{\mathrm{S}}_{\mathtt{t}_1} \times D^{\mathrm{S}}_{\mathtt{t}_2} \end{aligned}$$

Note here that the function space is *not* lifted.

A PER on a type is:

Definition 6.5
Let $\mathtt{t}$ be a type, then P is a PER on $\mathtt{t}$ if P is a PER on $D^{\mathrm{S}}_{\mathtt{t}}$. □

For each type $\mathtt{t}$ we now define a finite set of *uniform PERs*, $\mathcal{U}(\mathtt{t})$, consisting of PERs on $\mathtt{t}$. A uniform PER on the integers is a PER on the standard domain of the integers which treats all the integers in the same way. The reason for only looking at uniform PERs is that in a comportment analysis we are typically interested in knowing whether an expression evaluates to an integer or undefined value—not about it being any particular integer. The uniform PER on the integers are:

Definition 6.6
A PER P on `Int` is *uniform* if, whenever π is a permutation on $\mathbb{Z}_\perp$ $(= D^{\mathrm{S}}_{\mathtt{Int}})$ leaving $\perp$ unchanged, then

$$\forall x, y \in \mathbb{Z}_\perp : (x, y) \in P \Leftrightarrow (\pi x, \pi y) \in P$$

□

The set of uniform PERs on the integers, which we call $\mathcal{U}(\mathtt{Int})$, thus contains the following 7 elements:

- $1\mathrm{A} = \emptyset$
- $2\mathrm{A} = \{(x, x) \mid x \in \mathbb{Z}\}$
- $3\mathrm{A} = \mathbb{Z} \times \mathbb{Z}$
- $1\mathrm{B} = \{(\perp, \perp)\}$
- $2\mathrm{B} = \{(x, x) \mid x \in \mathbb{Z}_\perp\}$
- $3\mathrm{B} = \mathbb{Z} \times \mathbb{Z} \cup \{(\perp, \perp)\}$
- $4 = \mathbb{Z}_\perp \times \mathbb{Z}_\perp$

and they are related by the subset ordering as in Figure 6.1 and the Egli-Milner ordering as in Figure 6.2. For the strict PERs {1B, 2B, 3B, 4} we observe that the Egli-Milner ordering coincides with the subset ordering.

Note that $\mathcal{U}(\mathtt{Int})$ is intersection and union closed: intersection closedness is desirable for abstract interpretation as it ensures each (standard) value has a best abstract approximation. Union closedness ensures that no more information than necessary is lost on a merge resulting from *if then else* .

We define the *uniform properties* $\mathcal{P}(\mathtt{t})$ associated with `t` in the obvious way:

Definition 6.7
To the set $\mathcal{U}(\mathtt{t})$ of uniform PERs there is associated the set $\mathcal{P}(\mathtt{t})$ of properties:

$$\mathcal{P}(\mathtt{t}) = \{|P| \mid P \in \mathcal{U}(\mathtt{t})\}$$

□

The uniform properties of `Int` are as one might expect:

$$\mathcal{P}(\mathtt{Int}) = \{\emptyset, \{\perp\}, \mathbb{Z}, \mathbb{Z}_\perp\}$$

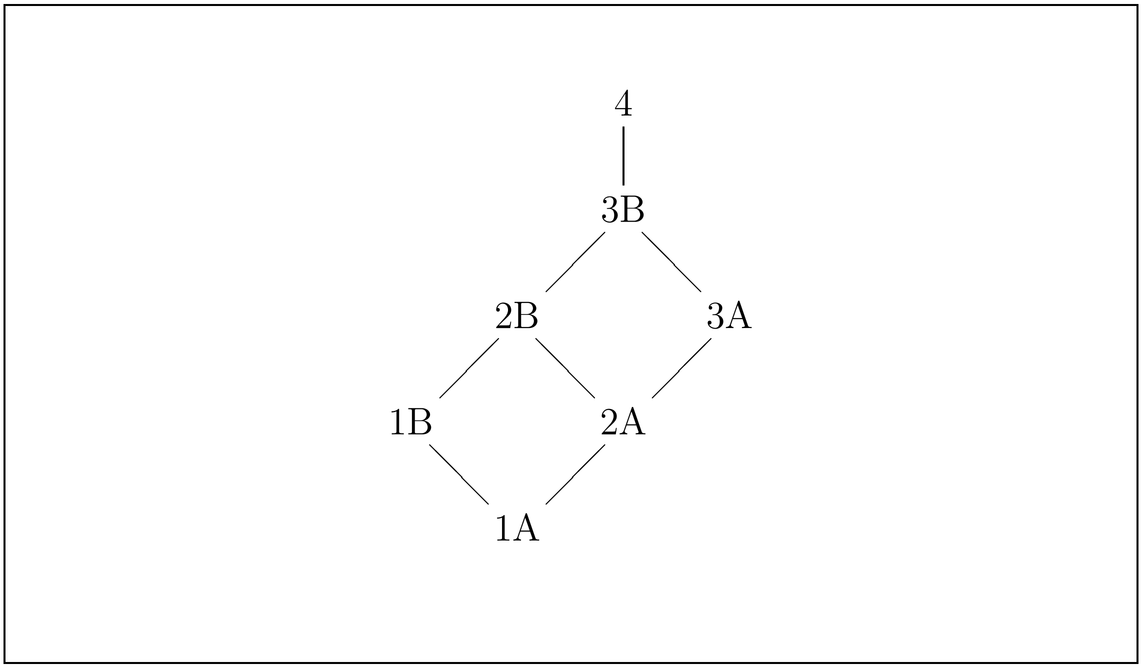

Figure 6.1: The Subset Ordering on `Int`

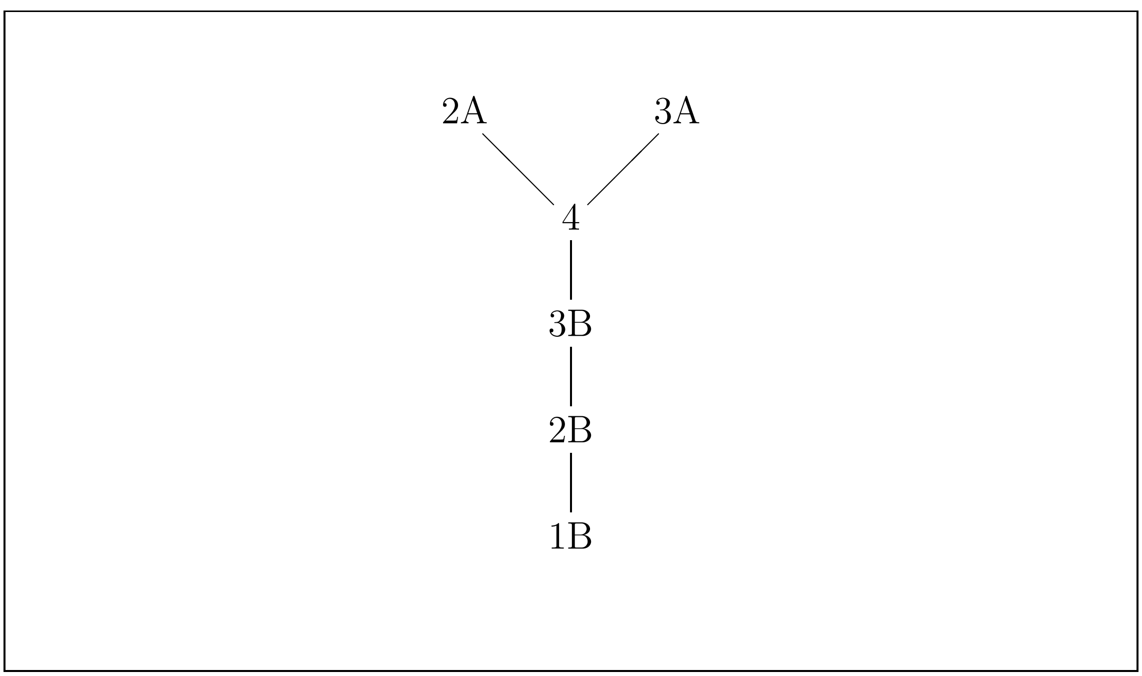

Figure 6.2: The Egli-Milner Ordering on `Int`

Starting from $\mathcal{U}(\mathtt{Int})$ we will define uniform PERs compound types. First we recall constructions which derive PERs at compound types from PERs of their components:

Definition 6.8
Given a PER P on $\mathtt{t}_1$ and a PER Q on $\mathtt{t}_2$ we can construct a PER $P \boxed{\rightarrow} Q$ on $\mathtt{t}_1 \rightarrow \mathtt{t}_2$ as:

$$P \boxed{\rightarrow} Q = \{(f, g) \in D^{\mathrm{S}}_{\mathtt{t}_1 \rightarrow \mathtt{t}_2} \times D^{\mathrm{S}}_{\mathtt{t}_1 \rightarrow \mathtt{t}_2} \mid \forall (a, b) \in P \Rightarrow (fa, gb) \in Q\}$$

and a PER $P \boxed{\times} Q$ on $\mathtt{t}_1 \times \mathtt{t}_2$ as:

$$P \boxed{\times} Q = \{((a, c), (b, d)) \in D^{\mathrm{S}}_{\mathtt{t}_1 \times \mathtt{t}_2} \times D^{\mathrm{S}}_{\mathtt{t}_1 \times \mathtt{t}_2} \mid (a, b) \in P, (c, d) \in Q\}$$

□

Now we define the set of *uniform* PERs on compound types inductively. Doing so is facilitated by defining them mutually with a set of *basic* PERs.

Definition 6.9 *The uniform* PERs *on compound types*
Given types $\mathtt{t}_1$ and $\mathtt{t}_2$ and their associated sets $\mathcal{U}(\mathtt{t}_1)$ and $\mathcal{U}(\mathtt{t}_2)$ of uniform PERs, we define, at type $\mathtt{t}_1 \rightarrow \mathtt{t}_2$, the set of basic PERs, $\mathcal{B}(\mathtt{t}_1 \rightarrow \mathtt{t}_2)$ as:

$$\mathcal{B}(\mathtt{t}_1 \rightarrow \mathtt{t}_2) = \{P \boxed{\rightarrow} Q \mid P \in \mathcal{U}(\mathtt{t}_1), Q \in \mathcal{U}(\mathtt{t}_2)\}$$

and the set of uniform PERs, $\mathcal{U}(\mathtt{t}_1 \rightarrow \mathtt{t}_2)$ as:

$$\mathcal{U}(\mathtt{t}_1 \rightarrow \mathtt{t}_2) = \{\cap S \mid S \subseteq \mathcal{B}(\mathtt{t}_1 \rightarrow \mathtt{t}_2)\}.$$

Similarly we define, at type $\mathtt{t}_1 \times \mathtt{t}_2$, the set of basic PERs, $\mathcal{B}(\mathtt{t}_1 \times \mathtt{t}_2)$, as:

$$\mathcal{B}(\mathtt{t}_1 \times \mathtt{t}_2) = \{P \boxed{\times} Q \mid P \in \mathcal{U}(\mathtt{t}_1), Q \in \mathcal{U}(\mathtt{t}_2)\}$$

and the set of uniform PERs, $\mathcal{U}(\mathtt{t}_1 \times \mathtt{t}_2)$, as:

$$\mathcal{U}(\mathtt{t}_1 \times \mathtt{t}_2) = \{\cup S \mid S \subseteq \mathcal{B}(\mathtt{t}_1 \times \mathtt{t}_2)\}.$$

□

In general $\mathcal{B}(\mathtt{t}_1 \rightarrow \mathtt{t}_2)$ is not intersection closed; hence the definition of $\mathcal{U}(\mathtt{t}_1 \rightarrow \mathtt{t}_2)$ as its intersection-closure. Note that $\mathcal{U}(\mathtt{t}_1 \rightarrow \mathtt{t}_2)$ is not union closed but we resist the temptation to form the union-closure because the given definition preserves the 'abstract function spaces are function spaces on abstract values' property found in [BHA86, EM91, Hun91].

Fact 6.10
Let $\mathtt{t}$ be a type and A and B be members of $\mathcal{U}(\mathtt{t})$ then: $A \cap B \in \mathcal{U}(\mathtt{t})$ but in general $A \cup B \notin \mathcal{U}(\mathtt{t})$. □

Section 6.4 considers the uniform PERs and properties on `Int` $\rightarrow$ `Int` and the need for intersection closure in some detail.

For the Observation 6.12 below we need the following lemma:

Lemma 6.11 *Intersections and unions*
For PERs P_1, P_2, ... P_n on $\mathtt{t}_1$ and one PER Q on $\mathtt{t}_2$:

$$\begin{aligned} &(P_1 \cup P_2 \cup \cdots \cup P_n) \boxed{\rightarrow} Q \\ &= \;(P_1 \boxed{\rightarrow} Q) \cap (P_2 \boxed{\rightarrow} Q) \cap \cdots \cap (P_n \boxed{\rightarrow} Q) \end{aligned}$$

□

Proof Let

$$P_{\mathrm{L}} = (P_1 \cup P_2 \cup \cdots \cup P_n) \boxed{\rightarrow} Q$$

and

$$P_{\mathrm{R}} = (P_1 \boxed{\rightarrow} Q) \cap (P_2 \boxed{\rightarrow} Q) \cap \cdots \cap (P_n \boxed{\rightarrow} Q)$$

We show $P_{\mathrm{L}} = P_{\mathrm{R}}$ by first showing $P_{\mathrm{L}} \subseteq P_{\mathrm{R}}$ and then $P_{\mathrm{L}} \supseteq P_{\mathrm{R}}$.

Assume $(f, f') \in P_{\mathrm{L}}$ and further assume $(d, d') \in P_1 \cup P_2 \cup \cdots \cup P_n$. From the definition of $\boxed{\rightarrow}$ we get $(fd, f'd') \in$ Q. That is

$$\begin{aligned} (d, d') \in P_i &\Rightarrow ((fd, f'd) \in Q) \\ (d, d') \notin P_i &\Rightarrow ((fd, f'd) \in Q) \end{aligned}$$

and hence

$$(f, f') \in P_i \boxed{\rightarrow} Q$$

so

$$(f, f') \in \cap(P_i \boxed{\rightarrow} Q)$$

Next we assume that for all $1 \leq i \leq n$ we have $(f, f') \in \cap\ (P_i \boxed{\rightarrow}$ Q). That is

$$(d, d') \in P_i \Rightarrow ((fd, f'd) \in Q)$$

and hence

$$(d, d') \in \cup P_i \Rightarrow ((fd, f'd) \in Q)$$

so

$$(f, f') \in \cup P_i \boxed{\rightarrow} Q$$

as required. ■

Observation 6.12
We have

$$\begin{aligned}
&\mathcal{U}(\mathtt{t}_1 \times \mathtt{t}_2 \rightarrow \mathtt{t}_3) \\
&\quad = \{\cap S \mid S \subseteq \mathcal{B}(\mathtt{t}_1 \times \mathtt{t}_2 \rightarrow \mathtt{t}_3)\} \\
&\quad = \{\cap S \mid S \subseteq \{P \boxed{\rightarrow} Q \mid P \in \mathcal{U}(\mathtt{t}_1 \times \mathtt{t}_2), Q \in \mathcal{U}(\mathtt{t}_3)\}\} \\
&\quad = \{\cap S \mid S \subseteq \{P \boxed{\rightarrow} Q \mid P \in \{\cup S_1 \mid S_1 \in \mathcal{B}(\mathtt{t}_1 \times \mathtt{t}_2)\}, Q \in \mathcal{U}(\mathtt{t}_3)\}\}
\end{aligned}$$

Consider the uniform PERs on $\mathtt{t}_1 \times \mathtt{t}_2 \rightarrow \mathtt{t}_3$ but with the restriction that the product PERs involved are basic PERs. That is look at:

$$P_{\mathrm{B}} = \{\cap S \mid S \subseteq \{P \boxed{\rightarrow} Q \mid P \in \mathcal{B}(\mathtt{t}_1 \times \mathtt{t}_2), Q \in \mathcal{U}(\mathtt{t}_3)\}\}$$

Now since we have

$$\mathcal{B}(\mathtt{t}_1) \subseteq \mathcal{U}(\mathtt{t}_1)$$

we arrive at

$$\mathcal{U}(\mathtt{t}_1 \times \mathtt{t}_2 \rightarrow \mathtt{t}_3) \supseteq P_{\mathrm{B}}$$

Next we want to show that $\mathcal{U}(\mathtt{t}_1 \times \mathtt{t}_2 \rightarrow \mathtt{t}_3) \subseteq P_{\mathrm{B}}$ so that we have

$$\mathcal{U}(\mathtt{t}_1 \times \mathtt{t}_2 \rightarrow \mathtt{t}_3) = P_{\mathrm{B}}$$

Consider a PER in $\mathcal{U}(\mathtt{t}_1 \times \mathtt{t}_2 \rightarrow \mathtt{t}_3)$:

$$(P_{11} \cdots \cup P_{1n} \boxed{\rightarrow} Q_1) \cap \cdots \cap (P_{k1} \cup \cdots \cup P_{kn} \boxed{\rightarrow} Q_k)$$

where $P_{ij} \in \mathcal{B}(\mathtt{t}_1 \times \mathtt{t}_2)$ and $Q_i \in \mathcal{U}(\mathtt{t}_3)$. By applying Lemma 6.11 we get

$$\begin{aligned}
&(P_{11} \boxed{\rightarrow} Q_1) \cap \cdots \cap (P_{1n} \boxed{\rightarrow} Q_1) \\
&\qquad \cap \cdots \cap (P_{k1} \boxed{\rightarrow} Q_k) \cap \cdots \cap P_{kn} \boxed{\rightarrow} Q_k)
\end{aligned}$$

So we have that the PER is in P_{B} and therefore $\mathcal{U}(\mathtt{t}_1 \times \mathtt{t}_2 \rightarrow \mathtt{t}_3) \subseteq P_{\mathrm{B}}$.

□

Therefore in this Chapter we will restrict the types to the form

$$\texttt{t} ::= \texttt{Int} \mid \texttt{t} \times \cdots \times \texttt{t} \rightarrow \texttt{t}$$

This has the sole effect of forbidding products in function results. Indeed, we could have achieved much the same effect by treating such restricted use of product types as shorthand for curried functions. However the present formalism is more natural and enables us to discuss properties of products. Note that because of Observation 6.12, it suffices to consider $\mathcal{B}(\texttt{t}_1 \times \texttt{t}_2)$ instead of $\mathcal{U}(\texttt{t}_1 \times \texttt{t}_2)$; i.e. $\mathcal{U}(\texttt{t}_1 \times \texttt{t}_2) = \mathcal{B}(\texttt{t}_1 \times \texttt{t}_2)$.

6.4 Examples in $\texttt{Int} \rightarrow \texttt{Int}$

As an example let us take a look at the basic and uniform on $\texttt{Int} \rightarrow \texttt{Int}$. The PERs on $\texttt{Int} \rightarrow \texttt{Int}$ is a subset of the standard domain of

$$(\texttt{Int} \rightarrow \texttt{Int}) \times (\texttt{Int} \rightarrow \texttt{Int})$$

Given two uniform PERs P and Q on $\texttt{Int}$ a *basic* PER on the function space $\texttt{Int} \rightarrow \texttt{Int}$ can be constructed as:

$$\begin{aligned} & P \boxed{\rightarrow} Q \\ & \quad = \{(f, g) \in (\mathbb{Z}_\perp \rightarrow \mathbb{Z}_\perp) \times (\mathbb{Z}_\perp \rightarrow \mathbb{Z}_\perp) \mid \forall (a, b) \in P \Rightarrow (fa, gb) \in Q\} \end{aligned}$$

The set of all basic PERs on the function space $\texttt{Int} \rightarrow \texttt{Int}$ is:

$$\mathcal{B}(\texttt{Int} \rightarrow \texttt{Int}) = \{P \boxed{\rightarrow} Q \mid P, Q \in \mathcal{U}(\texttt{Int})\}$$

The set of all uniform PERs on the function space $\mathbb{Z}_\perp \rightarrow \mathbb{Z}_\perp$ is:

$$\mathcal{U}(\texttt{Int} \rightarrow \texttt{Int}) = \{\cap S \mid S \subseteq \mathcal{B}(\texttt{Int} \rightarrow \texttt{Int})\}$$

The reason why it is not sufficient to look at the basic PERs only is that not every expression has a least and best PER. Consider the function $\lambda x.\textit{if } x = \perp \textit{ then } \perp \textit{ else } 42$; we have that

$$(\lambda x.\textit{if } x = \perp \textit{ then } \perp \textit{ else } 42) \in |1\mathrm{B} \boxed{\rightarrow} 1\mathrm{B}|$$

and

$$(\lambda x.\textit{if } x = \perp \textit{ then } \perp \textit{ else } 42) \in |3\mathrm{A} \boxed{\rightarrow} 2\mathrm{A}|.$$

But the greatest lower bound in the subset ordering of PERs of the form $P \boxed{\rightarrow} Q$ of the two PERs $1B \boxed{\rightarrow} 1B$ and $3A \boxed{\rightarrow} 2A$ is the PER $3B \boxed{\rightarrow} 1A$ which is empty. One of the missing PER *is* the greatest lower bound in the subset ordering of $1B \boxed{\rightarrow} 1B$ and $3A \boxed{\rightarrow} 2A$.

The properties, $\mathcal{P}(\texttt{Int} \rightarrow \texttt{Int})$, on $\texttt{Int} \rightarrow \texttt{Int}$ are:

- `empty` $= \emptyset$
- `cgt` $= \{f \in \mathbb{Z}_\perp \rightarrow \mathbb{Z}_\perp \mid \forall a, b \in \mathbb{Z}_\perp : fa = fb \in \mathbb{Z}\}$
- `strcon`*† $= \{f \in \mathbb{Z}_\perp \rightarrow \mathbb{Z}_\perp \mid (\forall a, b \in \mathbb{Z} : fa = fb \in \mathbb{Z}) \wedge (f\perp = \perp)\}$
- `totcon`* $= \{f \in \mathbb{Z}_\perp \rightarrow \mathbb{Z}_\perp \mid \forall a, b \in \mathbb{Z} : fa = fb \in \mathbb{Z}\}$
- `ide`† $= \{f \in \mathbb{Z}_\perp \rightarrow \mathbb{Z}_\perp \mid (\forall a \in \mathbb{Z} : fa \in \mathbb{Z}) \wedge (f\perp = \perp)\}$
- `tot` $= \{f \in \mathbb{Z}_\perp \rightarrow \mathbb{Z}_\perp \mid \forall a \in \mathbb{Z} : fa \in \mathbb{Z}\}$
- `div` $= \{f \in \mathbb{Z}_\perp \rightarrow \mathbb{Z}_\perp \mid \forall a \in \mathbb{Z}_\perp : fa = \perp\}$
- `cgt|div` $= \{f \in \mathbb{Z}_\perp \rightarrow \mathbb{Z}_\perp \mid \forall a, b \in \mathbb{Z}_\perp : fa = fb \in \mathbb{Z}_\perp\}$
- `strcon|div`*† $= \{f \in \mathbb{Z}_\perp \rightarrow \mathbb{Z}_\perp \mid (\forall a, b \in \mathbb{Z} : fa = fb \in \mathbb{Z}_\perp) \wedge (f\perp = \perp)\}$
- `totcon|div`* $= \{f \in \mathbb{Z}_\perp \rightarrow \mathbb{Z}_\perp \mid \forall a, b \in \mathbb{Z} : fa = fb \in \mathbb{Z}_\perp\}$
- `ide|div`† $= \{f \in \mathbb{Z}_\perp \rightarrow \mathbb{Z}_\perp \mid (\forall a \in \mathbb{Z} : fa \in \mathbb{Z}) \wedge f\perp = \perp\} \cup \texttt{div}$
- `tot|div` $= \{f \in \mathbb{Z}_\perp \rightarrow \mathbb{Z}_\perp \mid \forall a \in \mathbb{Z} : fa \in \mathbb{Z}\} \cup \texttt{div}$
- `strict` $= \{f \in \mathbb{Z}_\perp \rightarrow \mathbb{Z}_\perp \mid f\perp = \perp\}$
- `all` $= \{f \in \mathbb{Z}_\perp \rightarrow \mathbb{Z}_\perp\}$

and they are related by the subset ordering as in Figure 6.3 and by the Egli-Milner ordering as in Figure 6.4. In the above, names have been chosen to match [CC94] except that '`cgt`' is used for 'convergent' leaving '`con`' to indicate 'constant'. Annotations: † indicates a PER added by conjunctive completion; * indicates a PER not in [CC94]. For PERs in [CC94] the '`|`' character in the name indicates it is added by disjunctive completion. Note that [CC94] has an error in that `ide|div` $\subseteq$ `tot|div` is omitted from the Hasse diagrams.

We see that subset ordering coincides with the Egli-Milner ordering for the strict PERs (i.e. `div`, `cgt|div`, `strcon|div`, `totcon|div`, `ide|div`, `tot|div`, `strict`, and `all`). Also note that some of the PERs are Egli-Milner equivalent, i.e.

$$\begin{aligned} \texttt{totcon|div} &\equiv_{\mathrm{EM}} \texttt{cgt|div} \\ \texttt{strict} &\equiv_{\mathrm{EM}} \texttt{ide|div} \\ \texttt{all} &\equiv_{\mathrm{EM}} \texttt{tot|div} \end{aligned}$$

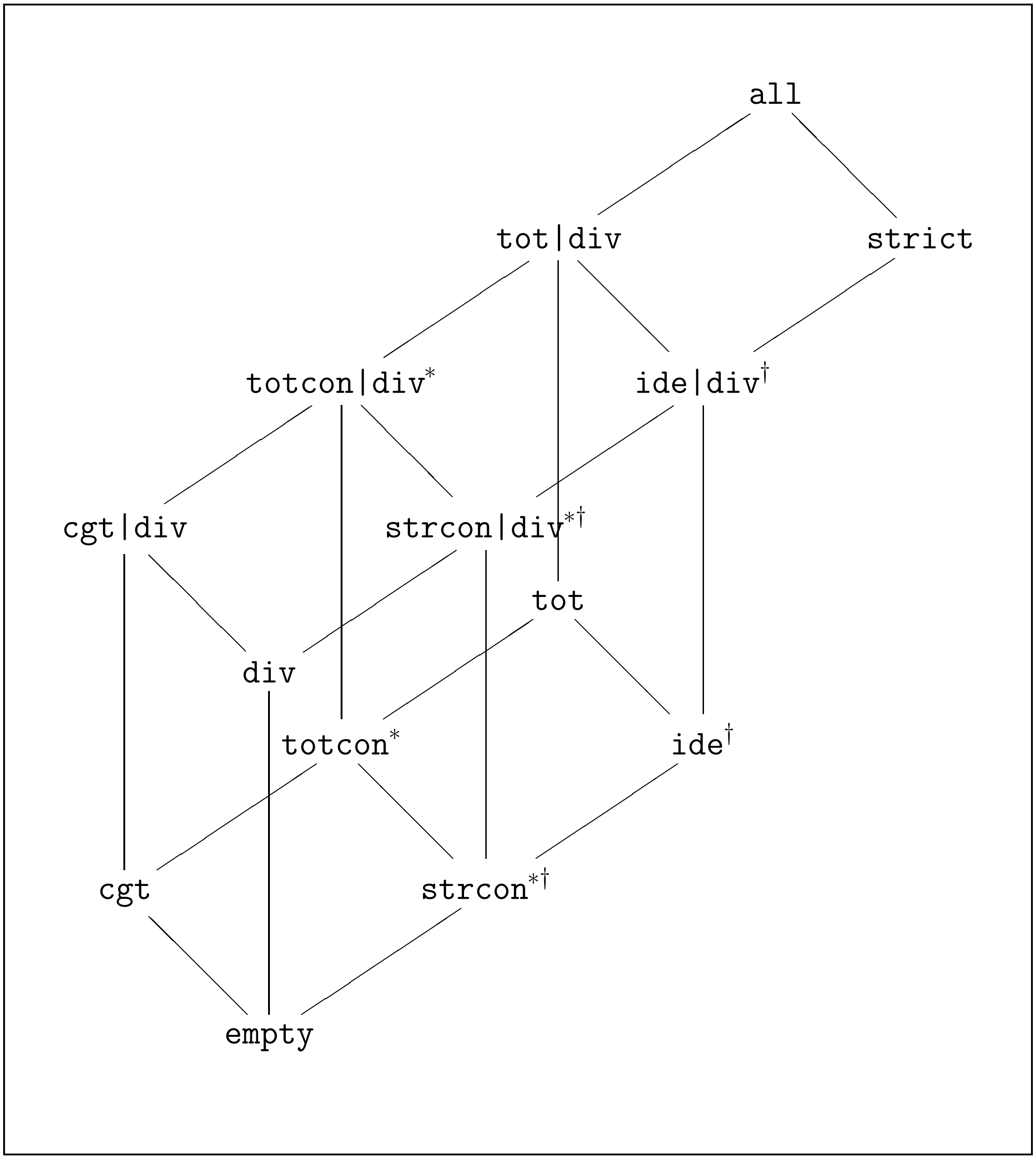

Figure 6.3: The Subset Ordering on Int → Int

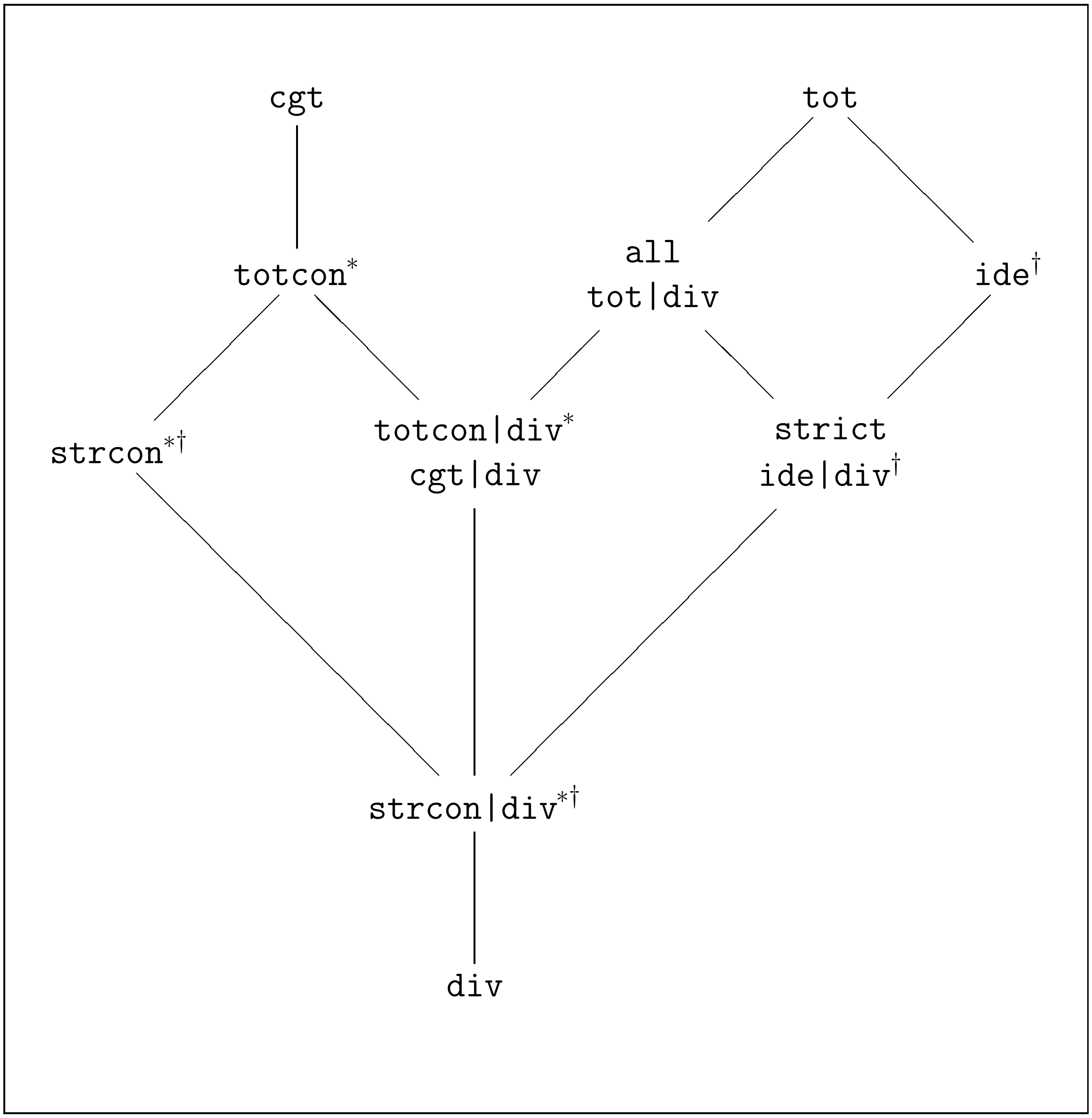

Figure 6.4: The Egli-Milner Ordering on `Int` → `Int`

It is however not clear how this affects the power of the framework.

One question to be asked is: "Is it useful with all those properties?" The advantage that we can get from knowing that an expression has a certain property is that we can optimise the code implementing the expression. Let f be an function from `Int` to `Int` with property:

`empty` — falsity property—which no function possesses.

`cgt` — evaluate $f\bot$ at compile-time and replace all calls to f with that result which is a terminating value

`strcon` — if the argument, a, is known to be an integer then $f0$ can be evaluated at compile-time and all these calls to f can be replaced by the result and if the argument is known not to terminate, then we can replace the calls with the result $\bot$

`totcon` — if the argument, a, is know to be an integer then $f0$ can be evaluated at compile-time and all these calls to f can be replaced by the result

`ide` — if the argument is known not to terminate, then we can replace the calls with the result $\bot$ and in the case where we know that the argument, a, is an integer then fa can be evaluated at compile-time

`tot` — if the argument is known to be an integer then we can evaluate the call at compile-time

`div` — replace all calls to f be the result $\bot$

`cgt|div` — replace all calls to f by $f\bot$; Egli-Milner equivalent to (and a subset of) `totcon|div`

`strcon|div` — if the argument is known not to terminate, then we can replace the calls with the result $\bot$ and if the argument, a, is know to be an integer then all these calls can be replaced by calls of e.g. $f0$

`totcon|div` — if the argument, a, is know to be an integer then all these calls can be replaced by calls of e.g. $f0$

`ide|div` — Egli-Milner equivalent to its convex closure `strict` compared to which there do not seem to be additional optimisations `tot|div` There appear no optimisations for this property (which is perhaps unsurprising given that it is Egli-Milner equivalent to `all`)

`strict` transform call-by-need and call-by-name to call-by-value

`all` truth property—which all functions possess

The next question to ask is: "Does there exist functions with all these properties?" The function $\lambda x.4$ has the property `cgt` and it does not possess a property less than `cgt`. The property `cgt` is the best description of that function. Similarly $\lambda x.\bot$ has the property `div` which is the best. Temporarily suppose e' is the term *true*$\oplus$*false* where $\oplus$ denotes non-deterministic choice; then we can construct a term with the best property `cgt|div` viz. `if` e' `then` $\lambda x.4$ `else` $\lambda x.\bot$. This argument is more delicate in the absence of such an operator since any such e' must reduce to *true*, *false* or *bot*. However this fact is not discernible uniformly (for any analysis method there are undetectable tautologies) and hence for any analysis method there is such a term with best property `cgt|div`. The term *fix* $(\lambda f.\lambda x.$`if x=0 then 1 else f(x-1)`$)$ has the property `strcon` as the best one. Now we are able to construct terms for the remainder as we did for the property `cgt|div`.

6.5 Comportment Analysis

We assume a simple typed functional language, whose types coincide with the meta-language types above. Its syntax is

$$\mathtt{e} = \mathtt{x} \mid \lambda\mathtt{x.e} \mid \mathtt{e\ e} \mid \mathtt{c} \mid \mathtt{if\ e\ then\ e\ else\ e} \mid \mathtt{fix\ e}$$

where `c` ranges over constants including `true` and `false` of type `Bool`, all the integers of type `Int`, and `pair`, `fst`, and `snd` for building and destroying pairs. Its semantics is given in terms of type-indexed semantic functions $\mathcal{E}^I_{\mathtt{t}}$ by

$$\begin{aligned}
&\mathcal{E}^I_{\mathtt{t}}\ [\![\mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3]\!]\rho \\
&\qquad\qquad = \mathtt{cond}^{\mathrm{I}}_{\mathtt{t}}\ (\mathcal{E}^I_{\mathtt{Bool}}\ [\![\mathtt{e}_1]\!]\rho, \mathcal{E}^I_{\mathtt{t}}\ [\![\mathtt{e}_2]\!]\rho, \mathcal{E}^I_{\mathtt{t}}\ [\![\mathtt{e}_3]\!]\rho) \\
&\mathcal{E}^I_{\mathtt{t}}\ [\![\mathtt{x}]\!]\rho = \rho\mathtt{x} \\
&\mathcal{E}^I_{\mathtt{t}_1\to\mathtt{t}_2}\ [\![\lambda\mathtt{x.e}]\!]\rho = \mathtt{lam}^I_{\mathtt{t}_1\to\mathtt{t}_2}(\lambda d.\mathcal{E}^I_{\mathtt{t}_2}\ [\![\mathtt{e}]\!]\rho[d/\mathtt{x}])
\end{aligned}$$

$$
\begin{aligned}
\mathcal{E}^{I}_{\mathtt{t}_2}\,[\![\mathtt{e}_1\mathtt{e}_2]\!]\rho &= \mathtt{app}^{I}_{\mathtt{t}_1\rightarrow\mathtt{t}_2}(\mathcal{E}^{I}_{\mathtt{t}_1\rightarrow\mathtt{t}_2}\,[\![\mathtt{e}_1]\!]\rho)(\mathcal{E}^{I}_{\mathtt{t}_1}\,[\![\mathtt{e}_2]\!]\rho)\\
\mathcal{E}^{I}_{\mathtt{t}}\,[\![\mathtt{c}]\!]\rho &= \mathtt{c}^{\mathrm{I}}\\
\mathcal{E}^{I}_{\mathtt{t}}\,[\![\mathtt{fix}\ \mathtt{e}]\!]\rho &= \mathtt{fix}^{\mathrm{I}}_{\mathtt{t}}\ (\mathcal{E}^{I}_{\mathtt{t}\rightarrow\mathtt{t}}\,[\![\mathtt{e}]\!]\rho)
\end{aligned}
$$

where $\mathtt{c}^{\mathrm{I}}$ (including $\mathtt{lam}^{I}_{\mathtt{t}_1}$ and $\mathtt{app}^{I}_{\mathtt{t}_1}$) are given by an interpretation which also specifies the interpretation of types as below.

For each type $\mathtt{t}$ we have beside the standard domains $D^{\mathrm{S}}_{\mathtt{t}}$ an abstract domain $D^{\mathrm{A}}_{\mathtt{t}}$. We will take $D^{\mathrm{A}}_{\mathtt{t}}$ to be the set of names of uniform PERs on $\mathtt{t}$; e.g. $D^{\mathrm{A}}_{\mathtt{t}} = \mathcal{N}(\mathtt{t})$. The names of the uniform PERs are defined as follows:

$$
\begin{aligned}
\mathcal{N}(\mathtt{Int}) &= \{1\mathrm{A}, 2\mathrm{A}, 3\mathrm{A}, 1\mathrm{B}, 2\mathrm{B}, 3\mathrm{B}, 4\}\\
\mathcal{N}(\mathtt{Bool}) &= \{1\mathrm{A}^{\mathtt{Bool}}, 2\mathrm{A}^{\mathtt{Bool}}, 3\mathrm{A}^{\mathtt{Bool}}, 1\mathrm{B}^{\mathtt{Bool}}, 2\mathrm{B}^{\mathtt{Bool}}, 3\mathrm{B}^{\mathtt{Bool}}, 4^{\mathtt{Bool}}\}\\
\mathcal{N}(\mathtt{t}_1 \rightarrow \mathtt{t}_2) &= \{[n_1, m_1; \ldots ; n_k, m_k] \mid n_i \in \mathcal{N}(\mathtt{t}_1), m_i \in \mathcal{N}(\mathtt{t}_2),\\
&\qquad k = |\mathcal{N}(\mathtt{t}_1)|\}\text{in one name all the } n_i\text{'s are distinct}\\
\mathcal{N}(\mathtt{t}_1 \times \mathtt{t}_2) &= \{(n, m) \mid n \in \mathcal{N}(\mathtt{t}_1), m \in \mathcal{N}(\mathtt{t}_2)\}
\end{aligned}
$$

Names on the function space $\mathtt{t}_1 \rightarrow \mathtt{t}_2$ can be seen as a graph of a function from names on $\mathtt{t}_1$ to names on $\mathtt{t}_2$. The reason that we can take the names for products to be pairs of names is that the uniform PERs on products is just the basic PERs; not unions of them.

For each type there is a function $\gamma_{\mathtt{t}_1}$, *the logical concretisation map*, from the set of names of PERs on the type, $\mathcal{N}(\mathtt{t})$, to the set of uniform PERs on the type, $\mathcal{U}(\mathtt{t})$:

$$
\gamma_{\mathtt{t}_1} :: \mathcal{N}(\mathtt{t}_1) \rightarrow \mathcal{U}(\mathtt{t}_1)
$$

$$
\begin{aligned}
\gamma_{\mathtt{Int}}(1\mathrm{A}) &= \emptyset\\
\gamma_{\mathtt{Int}}(2\mathrm{A}) &= \{(x, x) \mid x \in \mathbf{Z}\}\\
\gamma_{\mathtt{Int}}(3\mathrm{A}) &= \mathbf{Z} \times \mathbf{Z}\\
\gamma_{\mathtt{Int}}(1\mathrm{B}) &= \{(\perp, \perp)\}\\
\gamma_{\mathtt{Int}}(2\mathrm{B}) &= \{(x, x) \mid x \in \mathbf{Z}_{\perp}\}\\
\gamma_{\mathtt{Int}}(3\mathrm{B}) &= \mathbf{Z} \times \mathbf{Z} \cup \{(\perp, \perp)\}\\
\gamma_{\mathtt{Int}}(4) &= \mathbf{Z}_{\perp} \times \mathbf{Z}_{\perp}\\
\gamma_{\mathtt{Bool}}(1\mathrm{A}^{\mathtt{Bool}}) &= \emptyset\\
\gamma_{\mathtt{Bool}}(2\mathrm{A}^{\mathtt{Bool}}) &= \{(x, x) \mid x \in \mathtt{Bool}\}\\
\gamma_{\mathtt{Bool}}(3\mathrm{A}^{\mathtt{Bool}}) &= \mathtt{Bool} \times \mathtt{Bool}
\end{aligned}
$$

$$
\begin{array}{rcl}
\gamma_{\texttt{Bool}}(1\mathrm{B}^{\texttt{Bool}}) & = & \{(\bot,\bot)\} \\
\gamma_{\texttt{Bool}}(2\mathrm{B}^{\texttt{Bool}}) & = & \{(x,x) \mid x \in \texttt{Bool}_\bot\} \\
\gamma_{\texttt{Bool}}(3\mathrm{B}^{\texttt{Bool}}) & = & \texttt{Bool} \times \texttt{Bool} \cup \{(\bot,\bot)\} \\
\gamma_{\texttt{Bool}}(4^{\texttt{Bool}}) & = & \texttt{Bool}_\bot \times \texttt{Bool}_\bot \\
\gamma_{\texttt{t}_1 \to \texttt{t}_2}([n_1, m_1; \ldots; n_k, m_k]) & = & (\gamma_{\texttt{t}_1}(n_1) \boxed{\to} \gamma_{\texttt{t}_2}(m_1)) \cap \ldots \\
 & & \ldots \cap (\gamma_{\texttt{t}_1}(n_k) \boxed{\to} \gamma_{\texttt{t}_2}(m_k)) \\
\gamma_{\texttt{t}_1 \times \texttt{t}_2}((n,m)) & = & \gamma_{\texttt{t}_1}(n) \boxed{\times} \gamma_{\texttt{t}_2}(m)
\end{array}
$$

When using this representation (of functions by graphs) it is important to recall Hunt's observation that the γ's are not injective — two (extensionally) different functions between properties may describe the same PER on the function space.

The Egli-Milner ordering on names of uniform PERs is inherited from the Egli-Milner ordering on the uniform PERs:

Definition 6.13
Let n and m be in $\mathcal{N}(\texttt{t}_1)$ then $n \sqsubseteq^{\mathcal{N}}_{\mathrm{EM}} m$ if $\gamma_{\texttt{t}_1}(n) \sqsubseteq^{\mathrm{PER}}_{\mathrm{EM}} \gamma_{\texttt{t}_1}(m)$ □

Some other helpful functions are displayed in Figure 6.5 and 6.6.

$$
\begin{array}{lcl@{\qquad}lcl}
\pi_1^{\mathrm{S}} & :: & D^{\mathrm{S}}_{\texttt{t}_1 \times \texttt{t}_2} \to D^{\mathrm{S}}_{\texttt{t}_1} & \pi_1^{\mathrm{A}} & :: & D^{\mathrm{A}}_{\texttt{t}_1 \times \texttt{t}_2} \to D^{\mathrm{A}}_{\texttt{t}_1} \\
\pi_1^{\mathrm{S}}((n,m)) & = & n & \pi_1^{\mathrm{A}}((n,m)) & = & n \\
\pi_2^{\mathrm{S}} & :: & D^{\mathrm{S}}_{\texttt{t}_1 \times \texttt{t}_2} \to D^{\mathrm{S}}_{\texttt{t}_2} & \pi_2^{\mathrm{A}} & :: & D^{\mathrm{A}}_{\texttt{t}_1 \times \texttt{t}_2} \to D^{\mathrm{A}}_{\texttt{t}_2} \\
\pi_2^{\mathrm{S}}((n,m)) & = & m & \pi_2^{\mathrm{A}}((n,m)) & = & m
\end{array}
$$

Figure 6.5: The Projection Functions

The application and projection functions are monotonic and continues:

Lemma 6.14
The two families of functions $\texttt{app}^{\mathrm{S}}_{\texttt{t}_1 \to \texttt{t}_2}$ and $\texttt{app}^{\mathrm{A}}_{\texttt{t}_1 \to \texttt{t}_2}$ are monotonic.

□

Proof From domain theory we know that $\texttt{app}^{\mathrm{S}}_{\texttt{t}}$ is monotonic. We prove that $\texttt{app}^{\mathrm{A}}_{\texttt{t}}$ is monotonic by induction on the type $\texttt{t}$.

For the details see Appendix page 433. ■

$$
\begin{array}{lcl}
\mathtt{app}^{\mathrm{S}}_{\mathtt{t}_1\to\mathtt{t}_2} & :: & D^{\mathrm{S}}_{\mathtt{t}_1\to\mathtt{t}_2} \to D^{\mathrm{S}}_{\mathtt{t}_1} \to D^{\mathrm{S}}_{\mathtt{t}_2} \\
\mathtt{app}^{\mathrm{S}}_{\mathtt{t}_1\to\mathtt{t}_2}\ fx & = & fx \\
\\
\mathtt{app}^{\mathrm{A}}_{\mathtt{t}_1\to\mathtt{t}_2} & :: & D^{\mathrm{A}}_{\mathtt{t}_1\to\mathtt{t}_2} \to D^{\mathrm{A}}_{\mathtt{t}_1} \to D^{\mathrm{A}}_{\mathtt{t}_2} \\
\multicolumn{3}{l}{\mathtt{app}^{\mathrm{A}}_{\mathtt{t}_1\to\mathtt{t}_2}\ [n_1, m_1; \ldots; n_k, m_k] n_i = m_i} \\
\\
\mathtt{lam}^{\mathrm{S}}_{\mathtt{t}_1\to\mathtt{t}_2} & :: & (D^{\mathrm{S}}_{\mathtt{t}_1} \to D^{\mathrm{S}}_{\mathtt{t}_2}) \to D^{\mathrm{S}}_{\mathtt{t}_1\to\mathtt{t}_2} \\
\mathtt{lam}^{\mathrm{S}}_{\mathtt{t}_1\to\mathtt{t}_2}\ f & = & f \\
\\
\mathtt{lam}^{\mathrm{A}}_{\mathtt{t}_1\to\mathtt{t}_2} & :: & (D^{\mathrm{A}}_{\mathtt{t}_1} \to D^{\mathrm{A}}_{\mathtt{t}_2}) \to D^{\mathrm{A}}_{\mathtt{t}_1\to\mathtt{t}_2} \\
\mathtt{lam}^{\mathrm{A}}_{\mathtt{t}_1\to\mathtt{t}_2}\ f & = & [n_1, fn_1; \ldots; n_k, fn_k], \\
 & & \ \text{where}\ \ (n_i \in D^{\mathrm{A}}_{\mathtt{t}_1}) \wedge \\
 & & (k = |D^{\mathrm{A}}_{\mathtt{t}_1}|) \\
\\
\mathtt{fix}^{\mathrm{S}}_{\mathtt{t}_1} & :: & D^{\mathrm{S}}_{\mathtt{t}_1\to\mathtt{t}_1} \to D^{\mathrm{S}}_{\mathtt{t}_1} \\
\mathtt{fix}^{\mathrm{S}}_{\mathtt{t}_1}\ f & = & \sqcup\ \{d_i\} \\
 & & d_0 = \bot_{D^{\mathrm{S}}_{\mathtt{t}_1}} \\
 & & d_{i+1} = \mathtt{app}^{\mathrm{S}}_{\mathtt{t}_1\to\mathtt{t}_1}\ fd_i \\
\\
\mathtt{fix}^{\mathrm{A}}_{\mathtt{t}_1} & :: & D^{\mathrm{A}}_{\mathtt{t}_1\to\mathtt{t}_1} \to D^{\mathrm{A}}_{\mathtt{t}_1} \\
\mathtt{fix}^{\mathrm{A}}_{\mathtt{t}_1}\ n & = & \sqcup^{\mathcal{N}}_{\mathrm{EM}}\ \{d_i\} \\
 & & d_0 = \bot^{\mathrm{EM}\mathcal{N}}_{D^{\mathrm{A}}_{\mathtt{t}_1}} \\
 & & d_{i+1} = \mathtt{app}^{\mathrm{A}}_{\mathtt{t}_1\to\mathtt{t}_1}\ nd_i \\
\\
\mathtt{cond}^{\mathrm{S}}_{\mathtt{t}} & :: & D^{\mathrm{S}}_{\mathtt{Bool}} \to D^{\mathrm{S}}_{\mathtt{t}} \to D^{\mathrm{S}}_{\mathtt{t}} \to D^{\mathrm{S}}_{\mathtt{t}} \\
\mathtt{cond}^{\mathrm{S}}_{\mathtt{t}}\ (d_1, d_2, d_3) & = & \begin{cases} d_2, & \text{if } d_1 = \mathit{true} \\ d_3, & \text{if } d_1 = \mathit{false} \\ \bot_{\mathtt{t}}, & \text{if } d_1 = \bot_{\mathtt{Bool}} \end{cases} \\
\\
\mathtt{cond}^{\mathrm{A}}_{\mathtt{t}} & :: & D^{\mathrm{A}}_{\mathtt{Bool}} \to D^{\mathrm{A}}_{\mathtt{t}} \to D^{\mathrm{A}}_{\mathtt{t}} \to D^{\mathrm{A}}_{\mathtt{t}} \\
\mathtt{cond}^{\mathrm{A}}_{\mathtt{t}}\ (d_1, d_2, d_3) & = & \begin{cases} d_2 \sqcup^{\mathcal{N}}_{\mathrm{Subset}} d_3, & \text{if } d_1 \text{ is strict} \\ \bot^{\mathrm{EM}\mathcal{N}}_{D^{\mathrm{A}}_{\mathtt{t}}} \sqcup^{\mathcal{N}}_{\mathrm{Subset}} d_2 \sqcup^{\mathcal{N}}_{\mathrm{Subset}} d_3, & \text{otherwise} \end{cases}
\end{array}
$$

Figure 6.6: Auxiliary Functions

Lemma 6.15
The functions π_1^{S}, π_1^{A}, π_2^{S}, and π_2^{A} are monotonic. □

Proof From domain theory we know that π_1^{S} and π_2^{S} are monotonic. We prove that π_1^{A} and π_2^{A} are monotonic by induction on the type $\mathtt{t}_1 \times \mathtt{t}_2$.

For the details see Appendix page 434. ■

Lemma 6.16
The functions $\mathtt{app}_{\mathtt{t}}^{\mathrm{S}}$, $\mathtt{app}_{\mathtt{t}}^{\mathrm{A}}$, π_1^{S}, π_1^{A}, π_2^{S}, and π_2^{A} are continues. □

Proof It is a consequence of Lemma 6.14 and 6.15 and by recalling that monotonic functions from a space of finite height are continuous. ■

6.5.1 Correctness of the Analysis

Now we can define the relations by:

$$(d, d') \in \gamma_{\mathtt{t}_1}(a) \Leftrightarrow (d, d')\ \mathcal{R}_{\mathtt{t}_1}\ a$$

Definition 6.17 *Ternary Logical Relations*
A family $\mathcal{R}$ of type-indexed relations $(\mathcal{R}_{\mathtt{t}})$ is logical if for all $\mathtt{t}_1$ and $\mathtt{t}_2$:

$$\begin{array}{l} (f, f')\ \mathcal{R}_{\mathtt{t}_1 \to \mathtt{t}_2}\ h \\ \quad \Leftrightarrow\ (\forall d, d' \in D_{\mathtt{t}_1}^{\mathrm{S}}, \forall a \in D_{\mathtt{t}_1}^{\mathrm{A}} : (d, d')\ \mathcal{R}_{\mathtt{t}_1}\ a \Leftrightarrow \\ \quad\quad (\mathtt{app}_{\mathtt{t}_1 \to \mathtt{t}_2}^{\mathrm{S}}\ f d, \mathtt{app}_{\mathtt{t}_1 \to \mathtt{t}_2}^{\mathrm{S}}\ f' d')\ \mathcal{R}_{\mathtt{t}_2}\ (\mathtt{app}_{\mathtt{t}_1 \to \mathtt{t}_2}^{\mathrm{A}}\ h a)) \end{array}$$

and

$$\begin{array}{rcl} (p_1, p_2)\ \mathcal{R}_{\mathtt{t}_1 \times \mathtt{t}_2}\ p & \Leftrightarrow & (\pi_1^{\mathrm{S}}(p_1), \pi_1^{\mathrm{S}}(p_2))\ \mathcal{R}_{\mathtt{t}_1}\ \pi_1^{\mathrm{A}}(p) \wedge \\ & & (\pi_2^{\mathrm{S}}(p_1), \pi_2^{\mathrm{S}}(p_2))\ \mathcal{R}_{\mathtt{t}_2}\ \pi_2^{\mathrm{A}}(p) \end{array}$$

□

Proportion 6.18
The relation $\mathcal{R}$ is logical. □

Proof We prove that $\mathcal{R}$ is a logical relation by induction on the type.

For the details see Appendix page 435. ■

Proportion 6.19
The relation $\mathcal{R}$ is inductive. □

Proof

We prove that the relation $\mathcal{R}$ is inductive by induction on the type.

For the details see Appendix page 437. ■

The standard fixpoint and the abstract fixpoint of related values are related:

Lemma 6.20
Let $(f, f)\ \mathcal{R}_{\mathtt{t}\to\mathtt{t}}\ h$ then $(\mathtt{fix}^{\mathrm{S}}_{\mathtt{t}} f, \mathtt{fix}^{\mathrm{S}}_{\mathtt{t}} f)\ \mathcal{R}_{\mathtt{t}}\ (\mathtt{fix}^{\mathrm{A}}_{\mathtt{t}} h)$ □

Proof We assume $(f, f)\ \mathcal{R}_{\mathtt{t}\to\mathtt{t}}\ h$. First we show that $(d^{\mathrm{S}}_i, d^{\mathrm{S}}_i)\ \mathcal{R}_{\mathtt{t}}\ d^{\mathrm{A}}_i$ holds for all i by induction on i. Next from Lemma 6.14 we know that $\mathtt{app}^{\mathrm{S}}$ and $\mathtt{app}^{\mathrm{A}}$ are monotonic therefore are $\{d^{\mathrm{S}}_i\}$ and $\{d^{\mathrm{A}}_i\}$ chains and since $\mathcal{R}$ is inductive (Proposition 6.19) we have

$$(\sqcup_i\{d^{\mathrm{S}}_i\}, \sqcup_i\{d^{\mathrm{S}}_i\})\ \mathcal{R}_{\mathtt{t}}\ (\sqcup^{\mathcal{N}}_{\mathrm{EM}}\{d^{\mathrm{A}}_i\})$$

as required.

The case $i = 0$: We want to show $(\perp_{D^{\mathrm{S}}_{\mathtt{t}}}, \perp_{D^{\mathrm{S}}_{\mathtt{t}}})\ \mathcal{R}_{\mathtt{t}_1}\ \perp^{\mathrm{EM}\mathcal{N}}_{D^{\mathrm{A}}_{\mathtt{t}}}$ that is

$$(\perp_{D^{\mathrm{S}}_{\mathtt{t}}}, \perp_{D^{\mathrm{S}}_{\mathtt{t}}}) \in \gamma_{\mathtt{t}}(\perp^{\mathrm{EM}\mathcal{N}}_{D^{\mathrm{A}}_{\mathtt{t}}})$$

This is true since $\gamma_{\mathtt{t}}(\perp^{\mathrm{EM}\mathcal{N}}_{D^{\mathrm{A}}_{\mathtt{t}}})$ is strict.

The case $i + 1$: Now assume $(d^{\mathrm{S}}_i, d^{\mathrm{S}}_i)\ \mathcal{R}_{\mathtt{t}}\ d^{\mathrm{A}}_i$ and we will show

$$(d^{\mathrm{S}}_{i+1}, d^{\mathrm{S}}_{i+1})\ \mathcal{R}_{\mathtt{t}}\ d^{\mathrm{A}}_{i+1}$$

We have

$$\begin{aligned} d^{\mathrm{S}}_{i+1} &= \mathtt{app}^{\mathrm{S}}_{\mathtt{t}}\ f d^{\mathrm{S}}_i \\ d^{\mathrm{A}}_{i+1} &= \mathtt{app}^{\mathrm{A}}_{\mathtt{t}}\ h d^{\mathrm{A}}_i \end{aligned}$$

since $\mathcal{R}$ is logical (Proposition 6.18) we have

$$(\mathtt{app}^{\mathrm{S}}_{\mathtt{t}}\ f d^{\mathrm{S}}_i, \mathtt{app}^{\mathrm{S}}_{\mathtt{t}}\ f d^{\mathrm{S}}_i)\ \mathcal{R}_{\mathtt{t}}\ \mathtt{app}^{\mathrm{A}}_{\mathtt{t}}\ h d^{\mathrm{A}}_i$$

as required.

■

Lemma 6.21
Let $(f, f)\ \mathcal{R}_{\mathtt{t}_1 \to \mathtt{t}_2}\ h$ and $(d, d)\ \mathcal{R}_{\mathtt{t}_1}\ a$ then

$$(\mathtt{app}^{\mathrm{S}}_{\mathtt{t}_1 \to \mathtt{t}_2}\ fd, \mathtt{app}^{\mathrm{S}}_{\mathtt{t}_1 \to \mathtt{t}_2}\ fd)\ \mathcal{R}_{\mathtt{t}_2}\ (\mathtt{app}^{\mathrm{A}}_{\mathtt{t}_1 \to \mathtt{t}_2}\ ha)$$

□

Proof We have

$$\begin{aligned} (f, f)\ \mathcal{R}_{\mathtt{t}_1 \to \mathtt{t}_2}\ h \\ (d, d)\ \mathcal{R}_{\mathtt{t}_1}\ a \end{aligned}$$

from since $\mathcal{R}$ is a logical relation (Proposition 6.18) we get

$$(\mathtt{app}^{\mathrm{S}}_{\mathtt{t}_1 \to \mathtt{t}_2}\ fd, \mathtt{app}^{\mathrm{S}}_{\mathtt{t}_1 \to \mathtt{t}_2}\ fd)\ \mathcal{R}_{\mathtt{t}_2}\ (\mathtt{app}^{\mathrm{A}}_{\mathtt{t}_1 \to \mathtt{t}_2}\ ha)$$

as required. ■

Lemma 6.22
Let $(d_1, d_1)\ \mathcal{R}_{\mathtt{Bool}}\ a_1$, $(d_2, d_2)\ \mathcal{R}_{\mathtt{t}}\ a_2$, and $(d_3, d_3)\ \mathcal{R}_{\mathtt{t}}\ a_3$ then

$$(\mathtt{cond}^{\mathrm{S}}_{\mathtt{t}}\ (d_1, d_2, d_3), \mathtt{cond}^{\mathrm{S}}_{\mathtt{t}}\ (d_1, d_2, d_3))\ \mathcal{R}_{\mathtt{t}}\ \mathtt{cond}^{\mathrm{A}}_{\mathtt{t}}\ (a_1, a_2, a_3)$$

□

Proof We assume $(d_1, d_1)\ \mathcal{R}_{\mathtt{Bool}}\ a_1$, $(d_2, d_2)\ \mathcal{R}_{\mathtt{t}}\ a_2$, and $(d_3, d_3)\ \mathcal{R}_{\mathtt{t}}\ a_3$ and show by induction on d_1 that

$$(\mathtt{cond}^{\mathrm{S}}_{\mathtt{t}}\ (d_1, d_2, d_3), \mathtt{cond}^{\mathrm{S}}_{\mathtt{t}}\ (d_1, d_2, d_3))\ \mathcal{R}_{\mathtt{t}}\ \mathtt{cond}^{\mathrm{A}}_{\mathtt{t}}\ (a_1, a_2, a_3)$$

holds. We have

$$\begin{aligned}
\mathtt{cond}^{\mathrm{S}}_{\mathtt{t}}\ (d_1, d_2, d_3) &= \begin{cases} d_2, & \text{if } d_1 = \mathit{true} \\ d_3, & \text{if } d_1 = \mathit{false} \\ \bot_{\mathtt{t}}, & \text{if } d_1 = \bot_{\mathtt{Bool}} \end{cases} \\
\mathtt{cond}^{\mathrm{A}}_{\mathtt{t}}\ (d_1, d_2, d_3) &= \begin{cases} \bot^{\mathrm{EM}\mathcal{N}}_{D^{\mathrm{A}}_{\mathtt{t}}} \sqcup^{\mathcal{N}}_{\mathrm{Subset}} d_2 \sqcup^{\mathcal{N}}_{\mathrm{Subset}} d_3, & \text{if } d_1 \text{ is not strict} \\ d_2 \sqcup^{\mathcal{N}}_{\mathrm{Subset}} d_3, & \text{if } d_1 \text{ is strict} \end{cases}
\end{aligned}$$

The case $d_1 = \bot_{\mathtt{Bool}}$: We have

$$\mathtt{cond}^{\mathrm{S}}_{\mathtt{t}}\ (d_1, d_2, d_3) \ = \ \bot_{\mathtt{t}}$$

since $(d_1, d_1)\ \mathcal{R}_{\mathtt{Bool}}\ a_1$ is must be the case that a_1 is strict and hence

$$\mathtt{cond}^{\mathrm{A}}_{\mathtt{t}}\ (d_1, d_2, d_3) \;=\; \bot^{\mathrm{EM}\mathcal{N}}_{D^{\mathrm{A}}_{\mathtt{t}}} \sqcup^{\mathcal{N}}_{\mathrm{Subset}} d_2 \sqcup^{\mathcal{N}}_{\mathrm{Subset}} d_3$$

and clearly

$$(\bot_{\mathtt{t}}, \bot_{\mathtt{t}})\ \mathcal{R}_{\mathtt{t}}\ \bot^{\mathrm{EM}\mathcal{N}}_{D^{\mathrm{A}}_{\mathtt{t}}} \sqcup^{\mathcal{N}}_{\mathrm{EM}} d_2 \sqcup^{\mathcal{N}}_{\mathrm{EM}} d_3$$

as required.

The case $d_1 = \mathit{true}$: We have

$$\mathtt{cond}^{\mathrm{S}}_{\mathtt{t}}\ (d_1, d_2, d_3) \;=\; d_2$$

For the assumption we have

$$(d_2, d_2)\ \mathcal{R}_{\mathtt{t}}\ a_2$$

hence we have

$$(d_2, d_2)\ \mathcal{R}_{\mathtt{t}}\ a_2 \sqcup^{\mathcal{N}}_{\mathrm{Subset}} d_3$$

and

$$(d_2, d_2)\ \mathcal{R}_{\mathtt{t}}\ \bot^{\mathrm{EM}\mathcal{N}}_{D^{\mathrm{A}}_{\mathtt{t}}} \sqcup^{\mathcal{N}}_{\mathrm{EM}} a_2 \sqcup^{\mathcal{N}}_{\mathrm{Subset}} d_3$$

and therefore

$$(d_2, d_2)\ \mathcal{R}_{\mathtt{t}}\ \mathtt{cond}^{\mathrm{A}}_{\mathtt{t}}\ (a_1, a_2, a_3)$$

as required.

The case $d_1 = \mathit{false}$: Analogous to the case above.

■

Now the soundness and completeness of the analysis is:

Theorem 6.23
Let ρ be a standard environment and let δ be an abstract environment. Suppose for all constants $\mathtt{c}$ of type $\mathtt{t}'$ we have $(\mathtt{c}^{\mathrm{S}}, \mathtt{c}^{\mathrm{S}})\ \mathcal{R}_{\mathtt{t}'}\ \mathtt{c}^{\mathrm{A}}$ then for all $\mathtt{t}$ and $\mathtt{e}$ we have

$$\begin{array}{l} (\rho, \rho)\ \mathcal{R}\ \delta \\ \Downarrow \\ (\mathcal{E}^{S}_{\mathtt{t}}\ [\![\mathtt{e}]\!]\rho, \mathcal{E}^{S}_{\mathtt{t}}\ [\![\mathtt{e}]\!]\rho)\ \mathcal{R}_{\mathtt{t}}\ (\mathcal{E}^{A}_{\mathtt{t}}\ [\![\mathtt{e}]\!]\delta) \end{array}$$

□

Proof We prove the Theorem is by induction on the term.

For the details see Appendix page 442. ■

Example 6.24
First we will show that the analysis can discover that $\lambda\mathtt{x}.4$ is a total function and that the fixpoint of $\lambda\mathtt{x}.4$ is a integer. Next we show that the function $\lambda\mathtt{x}.\bot_{\mathtt{Int}}$ is divergent and the fixpoint of it is bottom. We calculate

$$\begin{aligned}
&\mathcal{E}^{A}_{\mathtt{Int}\to\mathtt{Int}}\ [\![\lambda\mathtt{x}.4]\!]\delta \\
&\quad= \mathtt{lam}^{\mathrm{A}}_{\mathtt{Int}\to\mathtt{Int}}\ (\lambda d.\mathcal{E}^{A}_{\mathtt{Int}}\ [\![4]\!]\delta[\mathtt{d}/\mathtt{x}]) \\
&\quad= \mathtt{lam}^{\mathrm{A}}_{\mathtt{Int}\to\mathtt{Int}}\ (\lambda d.4^{\mathrm{A}}) \\
&\quad= \mathtt{lam}^{\mathrm{A}}_{\mathtt{Int}\to\mathtt{Int}}\ (\lambda d.2\mathrm{A}) \\
&\quad= [1\mathrm{A},2\mathrm{A};2\mathrm{A},2\mathrm{A};3\mathrm{A},2\mathrm{A};1\mathrm{B},2\mathrm{A};2\mathrm{B},2\mathrm{A};3\mathrm{B},2\mathrm{A};4,2\mathrm{A}]
\end{aligned}$$

Now we have

$$\begin{aligned}
\mathcal{E}^{S}_{\mathtt{Int}\to\mathtt{Int}}\ [\![\lambda\mathtt{x}.4]\!]\rho &\in \mathcal{P}(\gamma_{\mathtt{Int}\to\mathtt{Int}}(\mathcal{E}^{A}_{\mathtt{Int}\to\mathtt{Int}}\ [\![\lambda\mathtt{x}.4]\!]\delta)) \\
&= \mathtt{cgt}
\end{aligned}$$

Recall that cgt is be best description of the function.

We also have

$$\mathcal{E}^{A}_{\mathtt{Int}}\ [\![\mathtt{fix}\ \lambda\mathtt{x}.4]\!]\delta \ = \ \mathtt{fix}^{\mathrm{A}}_{\mathtt{Int}\to\mathtt{Int}}\ (\mathcal{E}^{A}_{\mathtt{Int}\to\mathtt{Int}}\ [\![\lambda\mathtt{x}.4]\!]\delta)$$

and

$$\begin{aligned}
d_0 &= 1\mathrm{B} \\
d_1 &= \mathtt{app}^{\mathrm{A}}_{\mathtt{Int}\to\mathtt{Int}}\ (\mathcal{E}^{A}_{\mathtt{Int}\to\mathtt{Int}}\ [\![\lambda\mathtt{x}.4]\!]\delta)(1\mathrm{B}) \\
&= 2\mathrm{A} \\
d_2 &= \mathtt{app}^{\mathrm{A}}_{\mathtt{Int}\to\mathtt{Int}}\ (\mathcal{E}^{A}_{\mathtt{Int}\to\mathtt{Int}}\ [\![\lambda\mathtt{x}.4]\!]\delta)(1\mathrm{B}) \\
&= 2\mathrm{A}
\end{aligned}$$

and hence

$$\mathcal{E}^{A}_{\mathtt{Int}}\ [\![\mathtt{fix}\ \lambda\mathtt{x}.4]\!]\delta \ = \ 2\mathrm{A}$$

which is be best description of *4*.

We calculate

$$\begin{array}{l}\mathcal{E}^{A}_{\mathtt{Int}\to\mathtt{Int}}\,[\![\lambda\mathtt{x}.\bot_{\mathtt{Int}}]\!]\delta \\ \quad = \mathtt{lam}^{\mathrm{A}}_{\mathtt{Int}\to\mathtt{Int}}\,(\lambda d.\mathcal{E}^{A}_{\mathtt{Int}}\,[\![\bot_{\mathtt{Int}}]\!]\delta[d/x]) \\ \quad = \mathtt{lam}^{\mathrm{A}}_{\mathtt{Int}\to\mathtt{Int}}\,(\lambda d.{\bot_{\mathtt{Int}}}^{\mathrm{A}}) \\ \quad = \mathtt{lam}^{\mathrm{A}}_{\mathtt{Int}\to\mathtt{Int}}\,(\lambda d.1\mathrm{B}) \\ \quad = [1\mathrm{A}, 1\mathrm{B}; 2\mathrm{A}, 1\mathrm{B}; 3\mathrm{A}, 1\mathrm{B}; 1\mathrm{B}, 1\mathrm{B}; 2\mathrm{B}, 1\mathrm{B}; 3\mathrm{B}, 1\mathrm{B}; 4, 1\mathrm{B}]\end{array}$$

and hence we have

$$\begin{array}{rcl}\mathcal{E}^{S}_{\mathtt{Int}\to\mathtt{Int}}\,[\![\lambda\mathtt{x}.\bot_{\mathtt{Int}}]\!]\rho & \in & \mathcal{P}(\gamma_{\mathtt{Int}\to\mathtt{Int}}(\mathcal{E}^{A}_{\mathtt{Int}\to\mathtt{Int}}\,[\![\lambda\mathtt{x}.\bot_{\mathtt{Int}}]\!]\delta)) \\ & = & \mathtt{div}\end{array}$$

which is the best description of this function.

We also have

$$\mathcal{E}^{A}_{\mathtt{Int}}\,[\![\mathtt{fix}\;\lambda\mathtt{x}.\bot_{\mathtt{Int}}]\!]\delta \;=\; \mathtt{fix}^{\mathrm{A}}_{\mathtt{Int}\to\mathtt{Int}}\,(\mathcal{E}^{A}_{\mathtt{Int}\to\mathtt{Int}}\,[\![\lambda\mathtt{x}.\bot_{\mathtt{Int}}]\!]\delta)$$

and

$$\begin{array}{rcl}d_0 & = & 1\mathrm{B} \\ d_1 & = & \mathtt{app}^{\mathrm{A}}_{\mathtt{Int}\to\mathtt{Int}}\,(\mathcal{E}^{A}_{\mathtt{Int}}\,[\![\mathtt{fix}\;\lambda\mathtt{x}.\bot_{\mathtt{Int}}]\!])1\mathrm{B} \\ & = & 1\mathrm{B} \\ d_2 & = & \mathtt{app}^{\mathrm{A}}_{\mathtt{Int}\to\mathtt{Int}}\,(\mathcal{E}^{A}_{\mathtt{Int}}\,[\![\mathtt{fix}\;\lambda\mathtt{x}.\bot_{\mathtt{Int}}]\!])1\mathrm{B} \\ & = & 1\mathrm{B}\end{array}$$

and hence

$$\mathcal{E}^{A}_{\mathtt{Int}}\,[\![\mathtt{fix}\;\lambda\mathtt{x}.\bot_{\mathtt{Int}}]\!]\delta \;=\; 1\mathrm{B}$$

which is the best description of $\bot_{\mathtt{Int}}$. □

6.6 Summary

We have defined the notion of uniform PERs on the integers and by following the framework of Hunt [Hun91] we have lifted the uniform PERs to higher types. In the work of Hunt all the PERs are strict. Here we have encountered PERs which are not strict; since they are useful too, we had to handle the fixpoint iteration in another way. We defined the Egli-Milner ordering on PERs and used it for the fixpoint iteration.

More work is need to clarify why our approach yields four extra properties at type $\mathtt{Int} \to \mathtt{Int}$ compared to the approach in [CC94]. The significance of some of the properties to be Egli-Milner equivalent also need to be worked out.

This framework is for the un-lifted function space; future work will try to lift the framework to lifted function space — to be more in the line with the analyses in Chapter 2 and 3.

BHA-properties

Here we show that the strictness properties of Burn, Hankin and Abramsky [BHA86] naturally embed in the uniform PER properties presented here.

To be precise, let $\mathcal{U}'(\mathtt{Int})$ be given by

$$\{1\mathrm{B} = \{(\bot, \bot)\}, 2\mathrm{B} = \{(x, x) \mid x \in \mathbb{Z}_\bot\}\}$$

and $\mathcal{U}'(\mathtt{t}_1 \to \mathtt{t}_2)$ be defined inductively in the same manner as $\mathcal{U}(\mathtt{t}_1 \to \mathtt{t}_2)$. Then for all types $\mathtt{t}$ we have $\mathcal{U}'(\mathtt{t}) \subseteq \mathcal{U}(\mathtt{t})$ and moreover $\mathcal{U}'(\mathtt{t})$ is isomorphic to $\mathit{BHA}\ (\mathtt{t})$. Here $\mathit{BHA}\ (\mathtt{t})$ are the set of strictness properties defined by Burn, Hankin and Abramsky [BHA86]:

$$\begin{aligned} \mathit{BHA}\ (\mathtt{Int}) &= 2 = \{0, 1\} \\ \mathit{BHA}\ (\mathtt{t}_1 \to \mathtt{t}_2) &= \mathit{BHA}\ (\mathtt{t}_1) \to \mathit{BHA}\ (\mathtt{t}_2) \end{aligned}$$

Note that this embedding can also be seen as selecting uniform PERs representatives of the uniform ideals of [EM91].

Proportion 6.25
The set $\mathcal{U}'(\mathtt{t})$ of uniform PERs is isomorphic to $\mathit{BHA}\ (\mathtt{t})$. □

Proof The prove that $\mathcal{U}'(\mathtt{t})$ and $\mathit{BHA}\ (\mathtt{t})$ are isomorphic by induction on the type.

The case `Int`: We have

$$\begin{aligned} \mathcal{U}'(\mathtt{Int}) &= \{1\mathrm{B}, 2\mathrm{B}\} \\ \mathit{BHA}\ (\mathtt{Int}) &= \{0, 1\} \end{aligned}$$

and they are clearly isomorphic.

The case $\mathtt{t}_1 \to \mathtt{t}_2$: By applying the induction hypothesis to $\mathtt{t}_1$ and $\mathtt{t}_2$ we get that $\mathcal{U}'(\mathtt{t}_1)$ and $\mathit{BHA}\ (\mathtt{t}_1)$ are isomorphic and that $\mathcal{U}'(\mathtt{t}_2)$ and $\mathit{BHA}\ (\mathtt{t}_2)$ are isomorphic. We have

$$\mathit{BHA}\ (\mathtt{t}_1 \to \mathtt{t}_2) \;=\; \mathit{BHA}\ (\mathtt{t}_1) \to \mathit{BHA}\ (\mathtt{t}_2)$$

and

$$\mathcal{U}'(\mathtt{t}_1 \to \mathtt{t}_2) \;=\; \{\cap S \mid S \subseteq \{P \boxed{\to} Q \mid P \in \mathcal{U}'(\mathtt{t}_1), Q \in \mathcal{U}'(\mathtt{t}_2)\}\}$$

Let f_1 be an isomorphism between $\mathit{BHA}\ (\mathtt{t}_1)$ and $\mathcal{U}'(\mathtt{t}_1)$ and let f_2 be an isomorphism between $\mathit{BHA}\ (\mathtt{t}_2)$ and $\mathcal{U}'(\mathtt{t}_2)$. Now for any function $f \in \mathit{BHA}\ (\mathtt{t}_1 \to \mathtt{t}_2)$ we define the PER

$$P_f \;=\; \bigcap\{f_1 x \boxed{\to} \mathtt{f}_2(fx) \mid x \in \mathit{BHA}\ (\mathtt{t}_1)\}$$

which is an isomorphism between $\mathit{BHA}\ (\mathtt{t}_1 \to \mathtt{t}_2)$ and $\mathcal{U}'(\mathtt{t}_1 \to \mathtt{t}_2)$ as required.

■

Strictness and Totality Types

All the properties of Chapter 2 and 3 can be modelled by the uniform PERs except for two: the strictness and totality type, $((\mathtt{ut}_1 \to \mathtt{ut}_2)^{\mathbf{b}})$, expresses the property of knowing that a function does not have a WHNF and the property $((\mathtt{ut}_1 \to \mathtt{ut}_2)^{\mathbf{n}})$ of knowing that a function does have a WHNF. This is not a lack of the uniform PER notion, but due to the fact that we have used non-lifted function spaces in which $\lambda x.\bot = \bot$ and hence WHNF properties are inexpressible.

Let the ideal-based properties be as in (6.1), and the projection based properties are as in (6.2). For $\mathtt{Int} \to \mathtt{Int}$ we have summarised the approaches in Table 6.1.

PER	ideal (6.1)	projections (6.2)	Chapter 3
`empty`			
`cgt`			$\mathtt{Int}^{\mathbf{b}} \to \mathtt{Int}^{\mathbf{n}}$, $\mathtt{Int}^{\top} \to \mathtt{Int}^{\mathbf{n}}$
`strcon`			
`totcon`			
`ide`			
`tot`			$\mathtt{Int}^{\mathbf{n}} \to \mathtt{Int}^{\mathbf{n}}$
`div`	$W_{\mathbf{Z}_\perp,\{\perp\}}$		$\mathtt{Int}^{\top} \to \mathtt{Int}^{\mathbf{b}}$, $\mathtt{Int}^{\mathbf{n}} \to \mathtt{Int}^{\mathbf{b}}$
`cgt|div`		$W_{\lambda x.x,\lambda x.\perp}$	
`strcon|div`			
`totcon|div`			
`ide`			
`tot|div`			
`strict`	$W_{\{\perp\},\{\perp\}}$		$\mathtt{Int}^{\mathbf{b}} \to \mathtt{Int}^{\mathbf{b}}$
`all`	$W_{\mathbf{Z}_\perp,\mathbf{Z}_\perp}$, $W_{\{\perp\},\mathbf{Z}_\perp}$	$W_{\lambda x.x,\lambda x.x}$, $W_{\lambda x.\perp,\lambda x.\perp}$, $W_{\lambda x.\perp,\lambda x.x}$	$\mathtt{Int}^{\top} \to \mathtt{Int}^{\top}$, $\mathtt{Int}^{\mathbf{n}} \to \mathtt{Int}^{\top}$, $\mathtt{Int}^{\mathbf{b}} \to \mathtt{Int}^{\top}$, $(\mathtt{Int} \to \mathtt{Int})^{\top}$
			$(\mathtt{Int} \to \mathtt{Int})^{\mathbf{b}}$
			$(\mathtt{Int} \to \mathtt{Int})^{\mathbf{n}}$

Table 6.1: Properties on $\mathtt{Int} \to \mathtt{Int}$

Chapter 7

Conclusion

We conclude by summarise what we have done in this Thesis and discuss some future work.

7.1 Summary

Chapter 1 defines a general annotated type system. The strictness and totality analysis of Chapter 2 is an instance of this general annotated type system. In Chapter 3 we have extended this analysis to "full" conjunction. In Chapter 4 we then construct an type checking algorithm for the analysis of Chapter 3. The analysis of Chapter 5 is also an instance of the general annotated type system in Chapter 1. The analysis of Chapter 6 is specified using abstract interpretations, and the program analysis information includes strictness, totality, and constancy.

We will do two summaries: one that draws lines between the analyses constructed, and one that focus of the different techniques used.

7.1.1 Summary of Analyses

In **Chapter 1** we have reviewed some of the program analyses found in the literature. We have focussed on the analyses specified by an annotated type system. The analyses we saw was strictness analysis, usage analysis, control flow analysis, and binding time analysis. They can all be seen as instances of a general annotated type system.

In **Chapter 2** we defined a combined strictness and totality analysis. The functions that we consider are monotonic. We wanted that to be reflected by the type system. In order to state the monotonicity rule we had to define the $\downarrow$-operation on the strictness and totality types. However, it was not clear how to define the $\downarrow$-operation on conjunction types, therefore we only allowed conjunctions of annotated types at the top-level. We showed the analysis sound with respect to a natural style operational semantics.

In **Chapter 3** the analysis from Chapter 2 was extended to allow full conjunction. By letting $\downarrow$ be part of the syntax as a type constructor, we avoided the problem of defining the $\downarrow$-operation on conjunctions. The reason for not doing this already in Chapter 2 was that we cannot define validity for types of the form $\downarrow\mathtt{t}$ using the operational semantics, however this is easily done in the denotational semantics. We showed the analysis sound with respect to the denotational semantics.

In **Chapter 4** we then constructed an algorithm for checking the strictness and totality types from Chapter 3. The algorithm was constructed using the lazy type approach by Hankin and Le Métayer [HM94a]. The algorithm will given a term and an annotated type tell whether the term has the type. Our algorithm is only sound with respect to the analysis. The reason that our algorithm is not complete is found in the fact that we do *not* have the equivalence:

$$\downarrow(\mathtt{t}_1 \wedge \mathtt{t}_2) \equiv_{\mathrm{D}} \downarrow\mathtt{t}_1 \wedge \downarrow\mathtt{t}_2$$

Which is the same problem we had in Chapter 2 in defining the $\downarrow$-operation on conjunctions.

In **Chapter 5** we recalled the binding time analysis of Nielson and Nielson [NN92] and constructed an more efficient algorithm for inferring the annotated types. This algorithm differs from the one constructed in Chapter 4 in that it finds an annotated type for the term and a set of constraints that has to be fulfilled in order for the term to have the inferred type. The constraints are then solved and the solution applied to the annotated type; i.e. the algorithm of Chapter 4 is a type *checking* algorithm, whereas the algorithm here is a type *inference* algorithm. The complexity of the algorithm is $\mathcal{O}(n^4)$ where n is the size of the term.

In **Chapter 6** we have constructed an comportment analysis, i.e. a combined strictness, totality, and constancy analysis. We are using the names of the uniform PERs for the abstract interpretation. We have extend the framework of Hunt [Hun91] to allow non-strict PERs, thereby defined the Egli-Milner ordering on PERs.

7.1.2 Summary of Techniques

We will compare the different approaches to specifying the analyses, to proving the analyses sound, and to implementing the analyses.

Specification of Program Analyses

In this thesis we have seen two different ways of specifying a program analysis: in Chapter 2, 3 and 5 we specified the analyses by annotated type systems and in Chapter 6 we specified the analysis by abstract interpretation. In the annotated type system approach we define annotated types:

$$\begin{aligned} \mathtt{t} &::= \mathtt{B}^{S_1} \mid \mathtt{t}^{S_2} \mid \mathtt{ut}^{S_3} \mid \mathtt{t} \to^{S_4} \mathtt{t} \mid \mathtt{t} \wedge \mathtt{t} \\ \mathtt{ut} &::= \mathtt{B} \mid \mathtt{ut}_1 \to \mathtt{ut}_2 \end{aligned}$$

We construct an inference system with judgements of the form

$$\mathrm{A} \vdash \mathtt{e} : \mathtt{t}$$

saying that the term $\mathtt{e}$ has the type $\mathtt{t}$ under the assumptions A. The assumptions give annotated types to the free variables in the term. The analysis in Chapter 5 does not quite match this description: the judgement is of the form:

$$\mathrm{A} \vdash \mathtt{e} : \mathtt{t} : b$$

saying that the term $\mathtt{e}$ has the type $\mathtt{t}$ under the assumptions A for the free variables in $\mathtt{e}$ *and* the overall binding time b. However we can write it is as

$$\mathrm{A} \vdash \mathtt{e} : \mathtt{t}^b$$

and we are back in line.

The analysis in Chapter 6 is specified by abstract interpretation. We define a function that gives abstract values to the terms in the style of denotational semantics:

$$\mathrm{D}_{\mathtt{ut}} = \{\text{names of uniform PERs on } \mathtt{ut}\}$$

and we define a function $\mathcal{E}^{I}_{\mathtt{ut}}\ [\![.]\!]$ that gives abstract values to the terms:

$$\mathcal{E}^{I}_{\mathtt{ut}}\ [\![.]\!]\ ::\ Exp \rightarrow (Var \times \mathrm{D})^n \rightarrow \mathrm{D}_{\mathtt{ut}}$$

where Exp is the set of terms, Var is the set of variables, and n is the number of free variables in the term.

Soundness of the Analyses

Chapter	Specification of Analysis	Specification of Semantics	Proof-technique
2	inference system	inference system	induction on $\vdash$
3	inference system	denotational	induction on $\vdash$
6	denotational	denotational	induction on $\mathtt{t}$

Table 7.1: Proof Techniques

The specifications of the analyses must be proved sound with respect to the semantics of the language. We have seen two different ways of specifying the semantics: in Chapter 2 we defined a natural-style operational semantics (a big-step semantics) and in Chapter 3 and 6 we defined a denotational semantics. The proof-techniques used in the three chapters are different: in Chapter 2 both the analysis and the semantics are specified by inference systems, and we prove soundness by structural induction on the proof-tree for the analysis. In Chapter 3 the analysis is specified by an inference system and we have specified a denotational semantics. The proof is still done by induction on the proof-tree for the analysis. In Chapter 6 both the analysis and the semantics is specified as an interpretation of the language; an abstract interpretation and a standard interpretation. The proof is done by induction on the standard type of the term. This is summarised in Table 7.1.

One might expect that the proofs in Chapter 2 and Chapter 6 are the most easy ones since the approach of specifying the analysis and the semantics match. This is true for the proof in Chapter 6. However the proof of Chapter 2 is much more involved than that of Chapter 3. The reason is that where the denotational semantics gives an straightforward way of reasoning about fixpoints, we need in Chapter 2 to introduce special terms telling how many times it is allowed to unfold the fixpoint.

Algorithms for Program Analyses

A specification of an analysis is not practically useful before we have an algorithm implementing it. One advantage of specifying the analysis by abstract interpretation is that it suggests an algorithm at the same time: Just go ahead an implement the specification. But the resulting algorithm will in most cases be very inefficient due to the fixpoint computation.

Now the advantage of specifying the analysis by an inference system is that it does not suggest an algorithm, so we are free to choose whatever approach to construct the algorithm that we like.

In this thesis we have seen two approaches to standard type inference algorithms, i.e. the algorithm $\mathcal{T}$ by Damas [Dam85] and the algorithm $\mathcal{W}$ by [Mil78]. These algorithms will find the standard type that can be inferred for the given term. The algorithm for binding time analysis constructed in Chapter 5 is a variation of the algorithm $\mathcal{T}$. It does not only find the type of the term but also a set of constraints that has to be satisfied in order for the term to have the calculated type. Next an algorithm for solving the set of constraints is constructed. The solution that we are interested in is the one that is minimal in the sense that as few things as possible is going to be of run-time kind; i.e. as much as possible is done at compile-time. The algorithm constructed in Chapter 4 is different: given both the term and the type it will decide whether the term can have this type or not.

7.2 Future Work

The developments here was mostly done for the lambda calculus with constants and fixpoints. Real programming languages like Miranda [Tur85], Lazy ML and Haskell also includes lists, pairs, algebraic data-types, polymorphism, in addition to the lambda calculus.

The binding time analysis of Chapter 5 is already extended to products and lists in [Sol93]. The presentation of the analysis in Chapter 6 is with products, however we need to redo the development for the lifted function space. The work of Hunt [Hun91] is extended to sum types and recursive types.

Extending the analysis of Chapter 3 to lists can be inspired by the four-point domain of Wadler [Wad87]: we will have

$$\begin{aligned} \mathtt{ut} &::= \cdots \mid \mathtt{ut\ list} \\ \mathtt{t} &::= \cdots \mid \mathtt{t\ list} \end{aligned}$$

We may interpret the type $(\mathtt{ut\ list})^{\mathbf{n}}$ as any list of standard type `ut list` except the bottom list, e.g. the lists [1,3], [1, . . ., and [1, ⊥] are list of this type, $(\mathtt{ut\ list})^{\mathbf{b}}$ as the bottom list, ($\mathtt{Int}^{\mathbf{b}}$ `list`) as the infinite lists or lists with bottom elements of standard type `Int list`, e.g. [1, . . . and [⊥, ⊥], and ($\mathtt{Int}^{\mathbf{n}}$ `list`) as the finite list with no bottom elements, e.g. [1,3] and [ ].

Both Jensen [Jen92a] and Benton [Ben93] have extended their analyses to algebraic data-types.

Polymorphism can be include at two different levels: in the underlying type system and in the annotations. We might have

$$\begin{aligned} \mathtt{ut} &::= \cdots \mid \alpha \\ \mathtt{t} &::= \cdots \mid \alpha^{\beta} \mid \mathtt{ut}^{\beta} \end{aligned}$$

where α is a standard type variable, β is a annotation variable. Now we can give the following strictness and totality types to the identify function (λ`x.x`):

$$\begin{aligned} &\forall\alpha.\alpha \to \alpha \\ &\forall\beta.\mathtt{ut}^{\beta} \to \mathtt{ut}^{\beta} \\ &\forall\alpha,\beta.\alpha^{\beta} \to \alpha^{\beta} \\ &\forall\alpha.\alpha^{\mathbf{n}} \to \alpha^{\mathbf{n}} \end{aligned}$$

among others.

7.2.1 Multi-paradigmatic Languages

Multi-paradigmatic languages like CML [Rep91] and Facile [PGM90, TLP+93], which combines functional and concurrent programming, are used more and more. Hence there is a growing need for analyses for these languages. First it could be interesting to do the development in Chapter 2 for an eager language, like ML, since CML and Facile are build on top of standard ML. Note that for an eager language strictness analysis does not make any sense, however neededness analysis and the termination analysis are useful. We can then extend the types with the type of a `channel`:

$$\mathtt{ut} \ ::= \cdots \mid \mathtt{ut\ channel}$$

and the terms with the concurrency primitives:

$$\mathtt{e} \ ::= \ \cdots \mid \mathtt{spawn} \mid \mathtt{channel} \mid \mathtt{accept} \mid \mathtt{send}$$

The next step is to extend the annotated types:

$$\mathtt{t} \ ::= \ \cdots \mid \mathtt{t\ channel}$$

We can interpret the type ($\mathtt{Int}^{\mathbf{n}}$ `channel`) as the type of a channel where only terms of type $\mathtt{Int}^{\mathbf{n}}$ may be communicated, i.e. terms received on this channel will have the type $\mathtt{Int}^{\mathbf{n}}$. Communication over a channel of the type $(\mathtt{ut\ channel})^{\mathbf{n}}$ will always terminate, i.e. it is always possible to receive a value on this channel.

The semantics of multi-paradigmatic languages like CML [Rep91] and Facile [PGM90, TLP$^+$93] are easy to specify by an inference systems (operational semantics) but their semantics are difficult to specify as an denotational semantics. So in order to construct program analyses for multi-paradigmatic languages we need to be able to prove the analyses sound with respect to a operational semantics, i.e. the proof-technique developed in Chapter 2 may turn out to be useful.

Bibliography

[Abr90] Samson Abramsky. Abstract interpretation, logical relations and Kan extensions. *Journal of Logic and Computation*, 1(1):5–39, 1990.

[Amt93a] Torben Amtoft. Minimal thunkification. In *Proceedings WSA'93*, LNCS 724, pages 218–229, 1993. Available by WWW from
http://www.daimi.aau.dk/˜tamtoft/Papers/WSA93.ps.Z.

[Amt93b] Torben Amtoft. Strictness types: An inference algorithm and an application. Technical Report PB-448, DAIMI, 1993. Available by WWW from
http://www.daimi.aau.dk/˜tamtoft/Papers/PB448.ps.Z.

[Amt94] Torben Amtoft. Local type reconstruction by means of symbolic fixed point iterationa. In *Proceedings of ESOP'94*, LNCS 788, pages 43–57, 1994. Available by WWW from
http://www.daimi.aau.dk/˜tamtoft/Papers/ESOP94.ps.Z.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers — Principles, Techniques and Tools*. Addison-Wesley, 1986.

[Ben93] Nick Benton. *Strictness Analysis of Functional Programs*. PhD thesis, University of Cambridge, 1993. Available as Technical Report No. 309.

[BHA86] Geoffrey L. Burn, Chris Hankin, and Samson Abramsky. Strictness Analysis for Higher-order Functions. *Science of Computer Programming*, 7:249–278, 1986.

[Bon90] Anders Bondorf. Automatic autoprojection of higher order recursive equations. In *Proceedings of ESOP'90*, LNCS 432, pages 70–87, 1990.

[CC94] Patrick Cousot and Radhia Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and per analysis of functional languages), invited paper. In *Proceedings of the 1994 International Conference on Computer Languages, ICCL'94*, pages 95–112. IEEE Computer Society Press, 1994. Available by WWW from lix.polytechnique.fr.

[Con90] C. Consel. Binding time analysis for higher order untyped functional languages. In *Proceedings of LFP'90*, pages ???–???, 1990.

[Dam85] Luis Damas. *Type Assignment in Programming Languages*. PhD thesis, University of Edinburgh, Scotland, 1985.

[Des86] J. Despeyroux. Proof of translation in natural semantics. In *Sumposium on Logic in Computer Science*, 1986.

[EM91] Christine Ernoult and Alan Mycroft. Uniform ideals and strictness analysis. In *Proceeding of ICALP'91*, LNCS 510, 1991.

[FM88] You-Chin Fuh and Prateek Mishra. Type Inference with Subtypes. In *Proceeding of ESOP'88*, LNCS 300, pages 94–114, 1988.

[FM89] You-Chin Fuh and Prateek Mishra. Polymorphic subtype inference: closing the theory-pratice gap. In *Proceeding of TAPSOFT'89*, LNCS 352, pages 167–183, 1989.

[FM90] You-Chin Fuh and Prateek Mishra. Type Inference with Subtypes. *Theoretical Computer Science*, pages 155–175, 1990.

[Hen91] Fritz Henglein. Efficient type inference for higher-order binding time analysis. In *Proceedings of FPCA'91*, LNCS 523, pages 448–472, 1991.

[Hen94] Fritz Henglein. Iterative fixed point computation for type-based strictness analysis. In *Proceedings of SAS'94*, LNCS 864, pages 395–407, 1994.

[HM94a] Chris Hankin and Daniel Le Métayer. Deriving algorithms from type inference systems: Application to strictness analysis. In *Proceedings of POPL'94*, pages 202 – 212, 1994.

[HM94b] Fritz Henglein and Christian Mossin. Polymorphic binding-time analysis. In *Proceeding of ESOP'94*, LNCS 788, pages 278–301, 1994.

[HS91] Sebastian Hunt and David Sands. Binding time analysis: A new perspective. In *Proceedings of PEPM'91*, pages 154–165, 1991.

[Hun91] Sebastian Hunt. *Abstract Interpretation of Functional Languages: From Theory to Practice.* PhD thesis, University of London, 1991.

[Jen91] Thomas P. Jensen. Strictness analysis in logical form. In *Proceedings of FPCA'91*, LNCS 523, pages 352 – 366, 1991.

[Jen92a] Thomas P. Jensen. *Abstract Interpretation In Logical Form.* PhD thesis, University of London, Imperial College, 1992.

[Jen92b] Thomas P. Jensen. Disjunctive strictness analysis. In *Proceedings of LICS'92*, pages 174 – 185, 1992.

[Jør92] Jesper Jørgensen. Generating a compiler for a lazy language by partial evaluation. In *Proceedings of POPL'92*, pages 258–268, 1992.

[KM89] Tsung-Min Kuo and Prateek Mishra. Strictness analysis: A new perspective based on type inference. In *Proceedings of FPCA'89*, pages 260 – 272. ACM Press, 1989.

[Lau91] John Launchbury. *Projection Factorization in Partial Evaluation.* Cambridge University Press, 1991.

[LM91] Allen Leung and Prateek Mishra. Reasoning about Simple and Exhaustive Demand in Higher-Order Lazy Languages. In *Proceedings of FPCA'91*, LNCS 523, 1991.

[Mil78] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348 – 375, 1978.

[Mit91] John C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(3):245–285, 1991.

[MN83] Alan Mycroft and Flemming Nielson. Strong abstract interpretation using power domain (extended abstract). In *Proceedings of ICALP'83*, LNCS 154, 1983.

[MS95] Alan Mycroft and Kirsten Lackner Solberg. Uniform PERs and comportment analysis. In *Proceedings of PLILP'95*, LNCS 982, pages 169–187, 1995.

[Myc80] Alan Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *Proceedings of the 4th International Symposium on Programming*, LNCS 83, pages 269–281, 1980.

[Myc81] Alan Mycroft. *Abstract Interpretation and Optimising Transformation for Applicative programs.* PhD thesis, University of Edinburgh, Scotland, 1981.

[NN88] Hanne Riis Nielson and Flemming Nielson. Automatic binding time analysis for a typed lambda calculus. *Science of Computer Programming*, pages 139–176, 1988.

[NN89] Hanne Riis Nielson and Flemming Nielson. Transformations on higher-order functions. In *Proceedings of FPCA'89*, pages 129 – 143, 1989.

[NN92] Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages.* Cambridge University Press, 1992.

[NN95] Flemming Nielson and Hanne Riis Nielson. Termination analysis based on operational semantics. Technical Report DAIMI PB-492, Aarhus University, 1995.

[PGM90] Sanjiva Prasad, Alessandro Giacalone, and Prateek Mishra. Operational and algebraic semantics for Facile: A symmetric integration of concurrent and functional programming. In *ICALP 90*, 1990.

[Plo77] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Technical report, Aarhus University, 1981. DAIMI FN-19.

[Rep91] John H. Reppy. CML: A higher-order concurrent language. In *Proceedings of the Conference on Programming Language Design and Implementation*, June 1991.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[San90] David Sands. Complexity analysis for a lazy higher-order language. In *Proceedings of ESOP'90*, LNCS 432, pages 361–376, 1990.

[SNN92] Kirsten Lackner Solberg, Hanne Riis Nielson, and Flemming Nielson. Inference Systems for Binding Time Analysis. In *Proceedings of Workshop on Static Analysis '92*, pages 247 – 254, 1992.

[SNN94] Kirsten Lackner Solberg, Hanne Riis Nielson, and Flemming Nielson. Strictness and totality analysis. In *Proceedings of SAS'94*, LNCS 864, pages 408 – 422, 1994.

[Sol93] Kirsten Lackner Solberg. Inference systems for binding time analysis. Technical Report 25, Odense University, Denmark, 1993. ISSN No. 0903-3920.

[Sol95] Kirsten Lackner Solberg. Strictness and Totality Analysis with Conjunction. In *Proceedings of TAPSOFT'95*, LNCS 915, pages 501 – 515, 1995.

[Tan94] Yan-Mei Tang. *Control-Flow Analysis by Effect Systems and Abstract Interpretation.* PhD thesis, Ecole National Superieure des Mines de Paris, 1994.

[TJ92a] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):162 – 173, 1992.

[TJ92b] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect disipline. In *Proceedings of LICS'92*, 1992.

[TLP+93] Bent Thomsen, Lone Leth, Sanjiva Prasad, Tsung-Min Kuo, Andre Kramer, Fritz Knabe, and Alessandro Giacalone. Facile Antigua Release Programming Guide. Technical Report ECRC-93-20, European Computer-industry Research Centre, Munich, Germany, 1993.

[TT94] Mads Tofte and Jean-Pierre Talpin. Data region inference for polymorphic functional languages. In *Proceedings of POPL'94*, pages 188 – 201, 1994.

[Tur85] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Proceedings of FPCA'85*, LNCS 201, pages 1 – 16, 1985.

[Wad87] Phil Wadler. Strictness analysis on non-flat domains (by abstract interpretation over finite domains. In *Abstract Interpretation of Declarative Languages, Samson Abramsky and Chris Hankin (eds.)*, pages 266 – 275. Ellis Horwood, 1987.

[Wad91] Phil Wadler. Is there a use for linear logic? In *Proceedings of PEPM'91*, pages 255 – 273, 1991.

[WBF93] David A. Wright and Clement A. Baker-Finch. Usage analysis with natural reduction types. In *Proceeding of WSA'93*, LNCS 724, pages 254–266, 1993.

[WH87] Phil Wadler and John Hughes. Projections for strictness analysis. In *Proceedings of FPCA'87*, LNCS 27, 1987.

[Wri91] David A. Wright. A new technique for strictness analysis. In *Proceedings TAPSOFT'91*, LNCS 494, pages 260 – 272. Springer Verlag, 1991.