XHASKELL - ADDING REGULAR EXPRESSION TYPES TO HASKELL

KENNY ZHUO MING LU

(B.Science.(Hons), NUS)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

SCHOOL OF COMPUTING, DEPT OF COMPUTING SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2009

XHASKELL - ADDING REGULAR EXPRESSION TYPES TO HASKELL

KENNY ZHUO MING LU

NATIONAL UNIVERSITY OF SINGAPORE

2009

XHASKELL - ADDING REGULAR EXPRESSION TYPES TO HASKELL

KENNY ZHUO MING LU

2009

Acknowledgements

Some people to thank:

- Martin
- Jeremy and Greg
- Edmund, Zhu Ping, Meng, Florin, Corneliu, Hai, David, Beatrice, Christina, Alex, Dana, Shi Kun and those who work and used to work in the PLS-II lab
- Prof Khoo and Prof Dong
- Prof Chin
- Tom and Simon
- Those who reviewed my papers
- The thesis committee and the external examiner
- My family
- Rachel
- Qiming
- Jugui

Summary

Functional programming and XML form a good match. Higher order function and parametric polymorphism equip the programmer with powerful abstraction facilities while pattern matching over algebraic data types allows for a convenient notation to specify XML transformation. Previous works in extending Haskell with XML processing features focus on giving a data model for XML values, so that XML transformations can be expressed in terms of Haskell combinators.

Unfortunately, XML processing in Haskell does not provide the same static guarantees compared to XML processing in domain specific language such as XDuce and CDuce. These languages natively support regular expression type and (semantic) subtype polymorphism. These give much stronger static guarantees about the well-formedness of programs compared to the existing approaches that process XML documents in Haskell. In combination with regular expression pattern matching, we are allowed to write sophisticated and concise XML transformation.

In this thesis, we introduce an extension of Haskell, baptized XHaskell, which integrates XDuce features such as regular expression types, subtyping and regular expression pattern matching into Haskell. In addition, we also support the combination of regular expression types parametric polymorphism and type classes which to the best of our knowledge has not been studied before.

Contents

List of Figures

List of Symbols

t	Types
l	Labels
m	Monomials
n	Normal Forms
$*$	Kleene's Star
$\langle\rangle$	Empty Sequence
$\langle\cdot,\cdot\rangle$	Sequence
$(\cdot\vert\cdot)$	Choice
Γ	Type Environments
θ	Substitutions
$\{\}$	Empty Set
$\bot$	Empty Type
C	Constraints
x	Variables
a	Type Variables
e	System F* Expressions
p	System F* Patterns
w	System F* Values
E	System F Expressions
P	System F Patterns
v	System F Values

Chapter 1

Introduction

XML processing is one of the common tasks in modern computer systems. Programmers are often assisted by XML-aware programming languages and tools when they develop XML processing applications.

Traditional XML processing tools such as XSLT [80] and XML DOM [76] provide *minimal* support for XML manipulation. In particular these traditional approaches do not capture the schema information of the XML documents, e.g. DTD [17]. (Schema information tells us how an XML document is structured.) As a result, programs and applications developed in these languages and tools cannot be guaranteed to produce valid results with repect to the schema.

Such an issue can be addressed by adding *type* information to XML processing languages. In general, types can be viewed as an abstraction of the values which a program expression may evaluate to during run-time. In most of the main-stream programming languages, we use type systems to reason about types arising from the programs. The type soundness property guarantees that a well-typed program will never go wrong during run-time. We can recast this idea into the domain of XML processing. For instance, we find that the relation among XML schema and XML documents is analogical to the relation among program types and program values, as illustrated in Figure 1.1. We can think of XML documents as values in XML

$$\begin{tikzcd}
 & \textit{Programming languages} & \textit{XML} \\
\textit{Static} & \text{Type} & \text{Schema} \\
\textit{Dynamic} & \text{Value} \arrow[u, "\text{has type}"] & \text{XML document} \arrow[u, "\text{is valid w.r.t}"]
\end{tikzcd}$$

Figure 1.1: The connection between types and schema, values and XML documents

procoessing programs and XML schema as types.

There are two pioneering works embracing this idea. XDuce [30, 35] and CDuce [7, 21] are two strongly-typed functional languages for XML processing. In these languages, XML schema information is represented as type. In particular, they introduce a notion called *regular expression type*, which allows us to use regular operations such as Kleene's star, choice and sequence to build type expressions. This gives a natural representation of the XML schema information in the type system of XML processing languages, since XML schema declarations are often defined using regular expressions, too. Consequently, the type soundness property of these strongly typed XML processing languages guarantees that a well-typed program will always generate valid XML documents.

On the other hand, domain specific languages like XDuce often lack good library support. A programmer needs to develop her XML application from scratch. Furthermore, none of these languages (except for a recent version of XDuce) support parametric polymorphism, which is a common feature for most of the main-stream programming languages. Many type-based XML applications would benefit from parametric polymorphism because without parametric polymorphism, code duplication becomes a negative impact on the project development.

In this thesis, we venture with three major goals in mind,

1. We want to enrich a general-purpose language like Haskell with native XML support in XDuce style, i.e, semantic subtyping and regular expression pattern

matching;

2. We would like to study regular expression types, semantic subtyping and regular expression pattern matching in the context of System F [28, 58];

3. Ultimately, we want to develop a primitive calculus which supports formal reasoning about such a system.

As a result, we introduce an extension of Haskell, baptized XHaskell. XHaskell is a smooth integration of XDuce and Haskell. It supports the combination of regular expression types parametric polymorphism and type classes, which to the best of our knowledge have not been studied before. The XHaskell compiler is capable of tracing type errors back to the original locations in the source program. A meaningful error message is delivered to the programmer.

We translate XHaskell programs into the target language System F via a type-directed translation scheme. The translation scheme is developed based on a constructive subtyping proof system, which is an extension of Antimirov's algorithm [4] of regular expression containment check. In this translation scheme, we apply the proofs-are-programs principle (i.e., Curry-Howard isomorphism) to extract proof terms from the subtype proof derivations. The proof terms are realized as coercion functions among types. We translate XDuce's style features such as semantic subtyping and regular expression pattern matching by inserting the coercion functions. We prove that the translation preserves type soundness and coherence.

Another novelty in our work is the use of coercive pattern matching, which is the key to compiling regular expression pattern matching. We show that our implementation using coercive pattern matching is faithful with respect to the regular expression pattern matching relation.

Last but not least, we realize that the usability of XHaskell goes beyond the scope of XML processing. For example, we will show in a later chapter, using the combination of regular expression types, semantic subtyping, regular expression pattern matching and monadic parser combinator, that we are able to describe interesting and sophisticated parsing routines in a concise way.

1.1 Contributions

Our contributions are as follows:

- We formalize an extension of System F called System F*, which integrates semantic subtyping and pattern matching among regular expression types with parametric and ad-hoc polymorphism.
- We present the static and dynamic semantics of System F*.
- We develop a type-directed translation of System F* into System F, and prove that the translation scheme is coherent.
- We formalize a constructive proof system for regular expression subtyping and we derive coercive functions from the proof terms which are used in translating semantic subtyping and regular expression pattern matching.
- We study the regular expression pattern matching problem and develop a regular expression pattern matching algorithm based on regular expression derivatives rewriting. We implement the algorithm in Haskell.
- We develop a coercive pattern matching algorithm by applying proofs-are-programs principle to Antimirov's regular expression containment algorithm. We show that the coercive pattern matching algorithm is faithful with respect to the matching relation.
- We implement the full system in the XHaskell language. We show that the combination of parametric polymorphic regular expression type and type class is highly useful.

1.2 Thesis Outline

We outline this thesis as follows.

In Chapter 2, we further set up the full background of this work with some concrete examples. Readers who are already familiar with XML, XSLT and XDuce may find this chapter less exciting.

In Chapter 3, we highlight the key features of XHaskell by going through a series of examples.

In Chapter 4, we give a formal description of the core language of the XHaskell language, namely System F^*, which extends System F with regular expression type, semantic subtyping and regular expression pattern matching. We also describe a constructive proof system for regular expression subtyping.

In Chapter 5, we develop a source-to-source translation scheme from System F^* to System F. Furthermore, we give a constructive interpretation of the subtyping. The constructive interpretation is realized in terms of coercion functions. We sketch the definitions of these coercion functions. We make use of the coercion functions to translate regular expression pattern matching and subtyping. Furthermore, we also address the classic coherence problem in the context of coercive subtyping. We verify that our translation is coherent.

In Chapter 6, we study the core problem of regular expression pattern matching in detail. We first solve the pattern matching problem by developing a rewriting-based algorithm that make use of the derivative operation. Then we introduce the coercive pattern matching algorithm which is an extension of Antimirov's regular expression containment algorithm. We provide the details of the down/upcast coercion function. We show that our pattern matching algorithm is faithful with respect to the source semantics under the POSIX matching policy.

In Chapter 7, we discuss the details of XHaskell implementation and applications. This is the point where we report the pratical aspect of the system.

In Chapter 8, we provide a discussion of the related work.

In Chapter 9, we conclude the thesis.

Chapter 2

Background

2.1 XML

The e**X**tensible **M**arkup **L**anguage (XML) [75] is designed for data storage and data exchange in the World Wide Web. XML documents are text-based files. Tagged elements are the basic building blocks of XML documents. Each tagged element contains a sequence of attributes (name-value pairs) and a sequence of sub-elements. These sequences can be of any length. Each sub-element again is a tagged element or some text string. An XML document is *well-formed* if every element in it has one opening tag and one closing tag or self-closing. An XML document may have an accompanying **D**ocument **T**ype **D**efinition (DTD) file [17]. The DTD specifies what elements may appear in the XML document and how they can be structured in the document.[1] An XML document is *valid* if it is conformed to its type definitions. Readers with experience in typed programming language can view XML documents as values and DTDs as types. It is then natural to think of XML document validation as a kind of type checking process.

In Figure 2.1, we present a well-formed XML document `library.xml` and a DTD file `library.dtd` that describes the structure of a library. A library consists

[1]There are some advanced schema formats to define XML document types, such as XML Schema [77] and Relax NG [57]. These advanced schema are not discussed in this thesis.

```
library.dtd
<!DOCTYPE library [
<!ELEMENT library (collection)* >
<!ELEMENT collection (book,cdrom?)*>
<!ATTLIST collection type CDATA>
<!ELEMENT book (title, author*, year)>
<!ELEMENT title (#PCDATA) >
<!ELEMENT author (#PCDATA) >
<!ELEMENT year (#PCDATA) >
<!ELEMENT cdrom (title, year)>
]>

library.xml
<library>
  <collection type=''Computer Science''>
    <book>
      <title>Types and Programming Languages</title>
      <author>Benjamin C. Pierce</author>
      <year>2002</year>
    </book>
    <cdrom>
      <title>Types and Programming Languages (The CD-ROM)</title>
      <year>2002</year>
    </cdrom>
  </collection>
</library>
```

Figure 2.1: An XML document and its accompanying DTD

of zero-or-more collections. Each collection has its own set of books. Some books come with a CD-ROM and some do not. Every book has a title, several authors and a year of publication. It is clear that the XML document `library.xml` is valid with respect to its definition `library.dtd`.

2.2 Processing XML

XML processing is a common task in most of the real world computer-based systems. One of the most important applications in XML processing is to transform an XML document into different formats. There are many programming languages and tools that support XML transformation.

lib2bib.xsl

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:output method="xml" indent="yes"/>
  <xsl:template match="/">
     <bib> <xsl:apply-templates/> </bib>
  </xsl:template>
  <xsl:template match="//book">
     <entry>
        <title> <xsl:value-of select="title"/> </title>
        <author> <xsl:value-of select="author" /> </author>
        <year> <xsl:value-of select="year"/> </year>
     </entry>
  </xsl:template>
</xsl:stylesheet>
```

Figure 2.2: An XSLT Example

2.2.1 The untyped approach: XSLT

XSLT [80] is one of the first scripting languages designed for XML transformation. In XSLT, XML data are modelled as generic tree structures. Programmers transform these tree structures via XSLT templates. A template can be viewed like a function. It is applied to an input tree element if the element matches with the template's pattern. The content of the input element is then extracted and used to reconstruct the output document. For example, in Figure 2.2, we define an XSLT program that transforms a library document into a bibliography document. This program consists of two templates. The first template applies to the root of a library document and returns a bib element. The content of the bib is generated by another template. The second template turns a book element into an entry element.

There are some deficiencies in XSLT. XSLT programs are prone to error, because of text-based pattern matching. Furthermore, XSLT only enforces well-formedness of the XML documents and does not guarantee the validity of the resulting documents.

```
type Library = library[Collection*]
type Collection = collection[(Book,CD?)*]
type Book = book[(Title,Author*,Year)]
type Title = title[String]
type Author = author[String]
type Year = year[String]
type CD = cd[(Title,Year)]
type Bib = bib[Entry*]
type Entry = entry[(Title,Author*,Year,Publisher)]

let val libdoc = library[
                  collection[
                            book[ title["Types and Programming Languages"]
                             author["Benjamin C. Pierce"], year["2002"] ]
                            cd[ title["Types and Programming Languages"]
                             , year["2002"] ] ] ]

fun lib2bib (val v as Library) :  Bib =
    match v with library[ c as Collection* ] -> bib[cols2ens c]

fun cols2ens (val v as Collection*) :  Entry* =
    match v with
    (collection[bc as (Book, CD?)*], cs as Collection*) ->
                (bcs2ens bc, cols2ens cs)
    | () -> ()

fun bcs2ens (val v as (Book, CD?)*) :  Entry* =
    match v with
    (book[x as (Title,Author*,Year)], c as CD?, bc as (Book, CD?)*) ->
               (entry[x], bcs2ens bc)
    | () -> ()
```

Figure 2.3: A XDuce Example

2.2.2 The typed approach: XDuce

XDuce (pronounced as "transduce") [30, 35] is designed to overcome these deficiencies of XSLT. XDuce is a strongly typed functional language that is designed for XML processing. XDuce introduces the notion of regular expression types which directly resemble the DTD of XML documents. For instance, in Figure 2.3, we define a XDuce program that behaves the same as the XSLT program we have defined earlier. The first few lines are type declarations. We make use of regular expression types to model the `library.dtd` as well as the DTD for bibliography (which is omitted for its trivialness). The let expression defines a XDuce value `libdoc` representing the document `library.xml`. The rest of the program consists of three function definitions. The function `lib2bib` transforms a library element into a bibliography element. It uses pattern matching to extract the collections from the library element. Then we use a helper function `cols2ens` to build the content of the bibliography from the collection elements. The function `cols2ens` takes a sequence of collection elements and returns a sequence of entries. There are two pattern clauses in this function. The first pattern applies if the sequence has at least one collection element. On the right hand side of this pattern, we apply function `bcs2ens` to the variable `bc` which yields a sequence value of type `Entry*`. Then we apply `cols2ens` recursively to the remaining sequence, which in turn yields a sequence value of type `Entry*`. The final result is the concatenation of these two sequence values. Note that there is a type mismatch since the result is of type `(Entry*,Entry*)`, but `cols2ens`'s signature demands that the result should be of type `Entry*`. This is still well-typed, because XDuce allows for semantic subtyping, which checks that the type `(Entry*,Entry*)` is semantically subsumed by the type `Entry*`. Finally the second pattern applies to the empty sequence, and the same observation applies. The third function `bcs2ens` takes a sequence of books and CDs and returns a sequence of entries. The definition should be clear to the readers. As we can see from this example, XDuce's type soundness property guarantees that XDuce programs never yield run-time errors and the resulting XML documents are always valid with respect to their DTD definitions.

Though XDuce is strongly typed and offers better static guarantee to programs as compared to XSLT, its lack of library support often limits the development of applications that are written in XDuce. XDuce is implemented as an interpreter. In some follow-up work, CDuce [7] provides a compilation scheme which is much more efficient. The first version of XDuce did not support parametric polymorphism. In a latter work [33], an extension of XDuce is devised to support parametric polymorphism. In their extension, type variables are restricted to appear in "guarded positions" only. A detailed discussion can be found in Chapter 8 Section 8.4.

2.3 Our Work

XHaskell combines features from XDuce and Haskell, including regular expression types and regular expression pattern matching (XDuce), algebraic data types, parametric polymorphism and ad-hoc polymorphism (Haskell). Such a language extension is highly useful. With XHaskell, Haskell programmers can enjoy nice language facilities such as regular expression type and pattern matching.

Comparing with the existing works, the *unique* features of XHaskell are summarized as follows,

1. In XHaskell, libraries written in Haskell are made highly accessible to programmers fond of XDuce style programming.

2. XHaskell introduces a more liberal form of parametric polymorphism compared to earlier works by Hosoya et al [33] and Vouillon [67].

3. XHaskell is the first language that combines regular expression types with type classes.

As a language extension, XHaskell provides good error support and gives us ample space for program optimization.

For example, in Figure 2.4, we recast the previous XDuce example `lib2bib` in XHaskell. The first few lines of the program extend data type definitions as found in Haskell. The novelty is the use of regular expression notation on the right-hand side. Thus, we can describe the library DTD in terms of XHaskell data types. The XHaskell function `lib2bib` is very similar to the earlier XDuce function. It takes a library document as input and generates a bibliography document. In function

```
data Library = Library Collection*
data Collection = Collect (Book,CD?)*
data Book = Book (Title,Author*,Year)
data Title = Title String
data Author = Author String
data Year = Year String
data CD = CD (Title,Year)
data Bib = Bib Entry*
data Entry = Entry (Title,Author*,Year)

libdoc :: Library
libdoc = Library
          (Collection
                     ((Book (Title "Types and Programming Languages",
                        Author "Benjamin C. Pierce", Year "2002")),
                     (CD (Title "Types and Programming Languages",
                       , (Year "2002")))))

lib2bib :: Library -> Bib
lib2bib (Library (cols :: Collection*)) = Bib (mapStar col2ens cols)

col2ens :: Collection -> Entry*
col2ens (Collection (bc :: (Book,CD?)*)) = mapStar bc2en bc

bc2en :: (Book,CD?) -> Entry
bc2en (Book (x::(Title,Author*,Year)), y::CD?) = Entry x

mapStar :: (a -> b) -> a* -> b*
mapStar (f :: a -> b) (x :: a, xs :: a*) = (f x, mapStar f xs)
mapStar (f :: a -> b) () = ()
```

Figure 2.4: A XHaskell Example

`lib2bib`, we use the combination of Haskell style patterns and XDuce style type-based regular expression pattern matching to extract the collections from a library. In body of the pattern clause, we make use of a polymorphic function `mapStar` to traverse the content of a library. Like function `map` found in Haskell and ML, function `mapStar` defines a generic traversal over a sequence of elements. Let `col2ens` be a helper function which turns a collection element into a sequence of entries. We apply `(mapStar col2ens)` to the variable `cols` to generate the content of the bibliography element. Note that the result of the application `(mapStar col2ens cols)` has type

`(Entry*)*`. On the other hand, the constructor `Bib` expects its argument of type `Entry*`. Thanks to semantic subtyping, we can safely use the expression of type `(Entry*)*` in the context of `Entry*`. We define function `col2ens` using the same technique.

Polymorphism refers to the language feature that allows a piece of code to be used under different types. XHaskell inherits *subtype polymorphism* from XDuce. In addition, XHaskell supports *parametric polymorphism* and *adhoc polymorphism*.

Parametric polymorphism allows functions and data types to be defined generically. The behavior of a parametric-polymorphic function/data type remains identical under the different concrete type instances. For instance, in our running example, function `mapStar` is used in two different contexts. In function `lib2bib`, `mapStar` describes a traversal over a sequence of collections and in function `cols2ens` `mapStar` is used to define a traversal over a sequence of books. But the semantics of `mapStar` remains the same in these two use sites.

One obvious advantage of using parametric polymorphism is that we can reuse the same function definition in different contexts, thus we save a lot of code duplication. On the other hand, combining parametric polymorphism and regular expression pattern matching is very challenging. As we will see shortly, regular expression pattern matching is a form of type-based pattern matching. The semantics is critically relying on runtime type information. It is hard to develop a static compilation scheme for regular expression pattern with polymorphic types.

This problem has already been recognized by previous works but only a few solutions have been proposed so far [33, 67]. In a nutshell, our approach is to employ a source-to-source translation scheme. By employing a structure representation of values of regular expression types in the target language, we translate subtyping by inserting coercions derived out of subtype proofs. This is a well-established idea and usually referred to as coercive subtyping. Our novel idea is that we can employ a similar method to translate pattern matching. As the run-time values carry enough structure, we are able to perform pattern matching independent of the types. We will provide detailed explanation in Chapter 5. On the other hand, this raises another issue known as "polymorphic faithfulness", i.e. the compiled code must behave the same as the source program. We will have a discussion on this issue in Chapter 8.

Adhoc polymorphism allows one function to have different behaviors in different type contexts. In some languages, this feature is also called *function overloading*. XHaskell supports adhoc polymorphism in terms of type classes [69]. Actually, being a language extension of Haskell, XHaskell inherits type class naturally. For instance, the following program defines a pretty-printer for author elements and title elements.

```
class Pretty a where
  pretty ::  a -> String

instance Pretty Author where
  pretty (Author v) = "<author>" ++ v ++ "</author>"

instance Pretty Title where
  pretty (Title v) = "<title>" ++ v ++ "</title>"
```

The first two lines introduce a type class called `Pretty`, whose member function `pretty` is a function that takes a value of type `a` and pretty-prints it into a string. The `Pretty` class has two instances. The first instance defines a pretty-printing function for an author element. We first print an opening author tag followed by the content of the author element and a closing author tag. The second instance defines a pretty-printing function for a title element. A notable point is that function `pretty`'s meaning changes as it is applied to different types of arguments.

CDuce [7, 21] supports ad-hoc polymorphism via a different type mechanism called type intersection. In Chapter 8, we give a detailed comparison between the two approaches.

Having all these advanced type features, XHaskell programmers are able to write highly expressive programs. For example, as we will see in the upcoming chapter, we are able express some XQuery and XPath style programs in XHaskell.

In addition, as we mentioned earlier, XHaskell programmers are allowed to access Haskell libraries and modules via import keywords. For instance, suppose we would like to make the titles of books in upper-case when they are converted into the entry, we make the following modification to the function `bc2ens`,

```
import Char (toUpper)


bc2en :: (Book,CD?) -> Entry
bc2en (Book (x::(Title,Author*,Year)), y::CD?) = Entry (upper_title x)
upper_title :: (Title,Author*,Year) -> (Title,Author*,year)
upper_title (Title t, auths :: Author*, yr :: Year) =
    (Title (map toUpper t), auths, yr)
```

In the above, we import the Haskell library `Char`, in which the `toUpper` function is defined. The `toUpper` function takes a character value and turns it into upper case if it has not been yet.

Up till now we have been talking about XHaskell only in the context of XML processing. XHaskell is not just "another language designed for XML processing". We discover more good use of regular expression type and pattern matching in combination with parser combinators. This seems to be highly useful and convenient for compiler writing. We leave the details to Chapter 7.

2.4 Summary

In this chapter, we have a short summary on XML processing languages and how we compare our system with these related works. XHaskell is taking the lead to combine regular expression types, regular expression pattern matching, parametric polymorphism and type classes in one programming language. In the upcoming chapter, we highlight the language features of XHaskell via a series of examples.

Chapter 3

The Programmer's Eye View

In this chapter we will give a brief introduction of the XHaskell system by going through a series of examples.

3.1 Regular Expression and Data Types

In XHaskell we can mix algebraic data types and regular expression types. Thus, we can give a recast of the classic XDuce example also found in [34]. First, we provide some type definitions.

```
data Person     = Person  (Name,Tel?,Email*)
data Name       = Name String
data Tel        = Tel String
data Email      = Email String
data Entry      = Entry (Name,Tel)
```

The above extend data type definitions as found in Haskell. The novelty is the use of regular expression notation on the right-hand sides. For example, the type `Email*` makes use of the operator Kleene star `*` and thus we can describe a type holding a sequence of values of type `Email`; the type `Tel?`, a short hand for `(Tel|())` makes use of another operator choice `|` to describe a type which can be either a `Tel` element or an empty sequence `()`. Thus, the first line introduces a data type `Person` whose content is a sequence of a `Name` element, followed by an optional `Tel` element and a sequence of `Email`s.

Like in Haskell, we can now write functions which pattern match over the above data types.

Example 1 The following function (possibly) turns a single person into a phone book entry.

```
person_to_entry :: Person -> Entry?
person_to_entry (Person (n:: Name, t::Tel, es :: Email*)) = Entry (n,t)
person_to_entry (Person (n:: Name, t::(),  es :: Email*)) = ()
```

In the first clause we use the combination of Haskell style patterns and XDuce style type-based regular expression patterns to check whether a person has a telephone number. In the body of the second clause, we use semantic subtyping. The empty sequence value `()` of type `()` is a subtype of `(Entry?)` because the language denoted by `()` is a subset of the language denoted by `(Entry?)`. Hence, we can conclude that the above program is type correct. □

In XHaskell, we can access the items in a sequence by pattern matching against the sequence.

Example 2 The following function turns a sequence of persons into a sequence of phone book entries.

```
persons_to_entries :: Person* -> Entry*
persons_to_entries (Person (n :: Name, t :: Tel, es :: Email*), ps :: Person*) =
    (Entry (n,t), persons_to_entries ps)
persons_to_entries (Person (n :: Name, es :: Email*), ps :: Person*) =
    persons_to_entries ps
persons_to_entries () = ()
```

In the first clause we check whether the first person has a telephone number. In the body we make use of the person's name and telephone number to build a phone book entry. Then we apply the function recursively to the rest of the sequence. In the second pattern we skip the first person element which has no telephone number. The last pattern deals with the empty sequence. □

In the XHaskell language (·, ·) denotes a built-in sequence operator as opposed to the Haskell pair data type. In the presence of regular expression subtyping and pattern matching, XHaskell sequence is more expressive than ordinary Haskell data type such as list. The structure of a sequence is not as rigid as the structure of a list. For instance, we can process a sequence from right to left, which can't be achieved easily with a list.

Example 3 For instance, in the following variant of `addrbook` function, we process the sequence from right to left.

```
persons_to_entries' :: Person* -> Entry*
persons_to_entries' (ps :: Person*, Person (n :: Name, t :: Tel, es :: Email*)) =
    (persons_to_entries' ps, Entry (n,t))
persons_to_entries' (ps :: Person*, Person (n :: Name, es :: Email*)) =
    persons_to_entries' ps
persons_to_entries' () = ()
```

□

Regular expression patterns are often ambiguous. To disambiguate the outcome of matching, we employ the POSIX [56] (longest match) policy.

Example 4 For instance, the following program removes the longest sequence of spaces from the beginning of a sequence of spaces and texts.

```
data Space = Space
data Text = Text String


longestMatch :: (Space|Text)* -> (Space|Text)*
longestMatch (s :: Space*, r :: (Space|Text)*) = r
```

The sub-pattern `(s :: Space*)` is potentially ambiguous because it matches an arbitrary number of spaces. However, in XHaskell we follow the longest match policy which enforces that sub-pattern `(s :: Space*)` will consume the longest sequence of spaces. For example, application of `longestMatch` to the value `(Space, Space, Text "Hello", Space)` yields `(Text "Hello'', Space)`. □

XHaskell also provides support for XML-style attributes.

Example 5 For example, we consider

```
data Book = Book {{author :: Author?, year :: Year}}
type Author = String
type Year = Int

findBooks :: Year -> Book* -> Book*
findBooks yr (b@Book{{year = yr'}},bs :: Book*) =
   if (yr == yr')
      then (b, findBooks yr bs)
      else (findBooks yr bs)
findBooks yr (bs :: ()) = ()
```

The above program filters out all books published in a specified year. The advantage of attributes `author` and `year` is that we can access the fields within a data type by name rather than by position. For example, the pattern `Book{{year = yr'}}` extracts the year out of a book whereas the pattern `b@` allows us to use `b` to refer to this book.

Attributes in XHaskell resemble labeled data types in Haskell. But there are some differences, therefore, we use a different syntax. The essential difference is that attributes may be optional. For example, `Book {{year = 1997}}` defines an author-less book published in 1997. This is possible because the attribute `author` has the optional type `Author?`. In case of

```
findGoethe :: Book* -> Book*
findGoethe (b@Book{{author = "Goethe", year = _}},bs :: Book*) =
        (b, findGoethe bs)
findGoethe _ = ()
```

the first clause applies if the author is present and the author is Goethe. In all other cases, i.e. the author is not Goethe, the book does not have an author at all or the sequence of books is empty, the second clause applies. Another (minor) difference between attributes in XHaskell and labeled data types in Haskell is that in XHaskell an attribute name can be used in more than one data type.

```
data MyBook = MyBook {{author :: Author?, year :: Year, price :: Int}}
```

This is more a matter of convenience and relies on the assumption that we use the attribute in a non-polymorphic context only. □

3.2 Regular Expression Types and Parametric Polymorphism

We can also mix parametric polymorphism with regular expressions. Thus, we can write a polymorphic traversal function for sequences similar to the `map` function in Haskell.

```
mapStar :: (a -> b) -> a* -> b*
mapStar f (x :: ()) = x
mapStar f (x :: a, xs :: a*) =  (f x, mapStar f xs)
```

In the above, we assume that type annotations are lexically scoped. For example, variable `a` in the pattern `x::a` refers to `mapStar`'s annotation.

Example 6 We can now straightforwardly specify a function which turns an address into a phone book by mapping function `person_to_entry` over the sequence of `Persons`.

```
data Book a    = Book a*
type Addrbook  = Book Person
type Phonebook = Book Entry


addrbook :: Addrbook -> Phonebook
addrbook (Book (x :: Person*)) = Book (mapStar person_to_entries x)
```

Notice the we also support the combination of regular expressions and parametric data types. □

Once we have `mapStar` it is easy to define `filterStar` and thus we can express star-comprehension similar to the way list-comprehension is expressed via `map` and `filter` in Haskell. The star-comprehension is a handy notation to write XQuery style programs.

Example 7 Here is a re-formulation of the `findBooks` function using star-comprehension.

```
findBooks' :: Year -> Book* -> Book*
findBooks' yr (bs :: Book*) = [ b | b@Book{{year = yr'}} <- bs, yr == yr']
```

Like list-comprehensions, a star-comprehension consists of a sequence of statements. Concretely, the above star-comprehension has two essential statements. The first statement `b@Book{{year = yr'}} <- bs` is a generator. For each book element `b` in `bs`, we extract the year of publication attribute and bind it to `yr'`. Via the next statement, we then check whether `yr` is equal to `yr'`. If this is the case we return `b`. In XQuery, the above could be written as follows

```
declare function findbooks' ($yr, $bs) {
  for $b in $bs
  where $b/@year = $yr
  return $b
}
```

where the for-clause iterates through a sequence of books, and the where-clause filters out those books that were published in year `$yr`. □

3.3 Regular Expression Types and Type Classes

XHaskell also supports the combination of type classes and regular expression types.

Example 8 For example, we can define `(*)` to be an instance of the `Functor` class.

```
instance Functor (*) where
  fmap = mapStar
```

□

In our next example we define an instance for equality among a sequence of types.

Example 9 Consider

```
instance Eq a => Eq a* where
  (==) (xs::()) (ys::()) = True
  (==) (x::a, xs::a*) (y::a, ys::a*) = (x==y)&&(xs==ys)
  (==) _ _ = False


instance Eq Email where
  (==) (Email x) (Email y) = x == y
```

Now we can make use of the above type class instances to check whether two sequences of `Emails` are equal.

```
eqEmails :: Email* -> Email* -> Bool
eqEmails (es1 :: Email*) (es2 :: Email*) = es1 == es2
```

where the use of `==` in the body of the above function refers to the instance of `Eq (Email*)` which is derivable given the two instances above. □

In the upcoming example, we show how to express a generic set of XPath operations in XHaskell.

Example 10 The following data type declarations introduce the structure of a library.

```
data Library = Library Collection*

data Collection = Collection Book*

data Book   = Book Author Year
```

Let `lib` be a value of type `Library`, we would like to extract all the books from `lib` via XPath-style combinator `lib//Book`.

The insight is to view `(//)` as an overloaded method. For instance, we use the following type class to describe the family of overloaded definitions of `(//)`.

```
class XPath a b where
  (//) :: a -> b -> b*


instance XPath Library Book where
  (//) (Library xs) b = xs // b


instance XPath Collection Book where
  (//) (Collection xs) b = xs // b


instance XPath Book Book where
  (//) x y = x


instance XPath a () where
  (//) _ _ = ()


instance XPath a t => XPath a* t where
  (//) xs t = mapStar (\x -> x // t) xs


instance (XPath a t, XPath b t) => XPath (a|b) t where
  (//) (x::a) t = x // t
  (//) (x::b) t = x // t
```

The operation `e1 // e2` extracts all "descendants" of `e1` whose type is equivalent to `e2`'s type. Thus, we use `lib//Book` to extract all book elements under `lib`. Note that `lib//Book` is desugared to `lib//undefined::Book` internally. □

3.4 Summary

We gave a brief overview of the XHaskell language and showed how to write concise XML transformation in XHaskell using algebraic data type, regular expression types, parametric polymorphism and type classes. There are further details of the XHaskell system such as the integration with GHC, type error reporting, etc. We will postpone the discussion of this detail till Chapter 7.

In the next chapter, we present the core language of XHaskell.

Chapter 4

System F* - The Core Calculus

In this chapter, we formalize System F*, a foundational extension of the polymorphic lambda calculus (also known as System F [28, 58]) with support for structured, recursive data types and regular expression types.

Like System F, System F* is a typed intermediate language. Without being distracted by the source language consideration such as type class resolution and type inference,sem we want to see that types are explicit and type classes have been already resolved (via the dictionary translation). Our focus here is to come up with an elementary semantics which is amenable to efficient compilation. As we will see in the next chapter, we achieve this via a type-driven translation scheme from System F* to System F.

We first give an overview of System F* via some examples. Then we present the formal details of the language, such as syntax, static semantics and dynamic semantics. Finally, we study the various properties of the language such as type decidability and type soundness, etc.

4.1 System F* by examples

Example 11 We consider a re-formulation of the address book example mentioned in Chapter 3,

```
data $Person$ = $Person$ $\langle Name, Tel?, Email^{*}\rangle$
data $Name:$ = $Name:$ String
data $Tel$ = $Tel$ String
data $Email$ = $Email$ String
data $Entry$ = $Entry$ $\langle Name, Tel\rangle$

persons_to_entries :$Person^{*}$ -> $Entry^{*}$
persons_to_entries = $\lambda v : Person^{*}$
    case $v$ of
      $\langle (Person\ \langle n : Name, t : Tel, es : Email^{*}\rangle), ps :: Person^{*}\rangle \rightarrow$
          $\langle Entry\ \langle n, t\rangle,$ persons_to_entries $ps\rangle$
      $\langle (Person\ \langle n : Name, es : Email^{*}\rangle), ps : Person^{*}\rangle \rightarrow$ persons_to_entries $ps$
      $\langle\rangle \rightarrow \langle\rangle$
```

The above is a recast of the function found in Example 2. The difference only lies in syntax. For example, we use : to denote type annotation instead of ::. We use the notation $\langle\cdot,\cdot\rangle$ to denote sequences to avoid confusion with pair data type $(\cdot,\cdot)$. For the same reason, we use $\langle\rangle$ to denote the empty sequence. The change of syntax is to indicate that we are reasoning with the core language (System F*) instead of the surface language XHaskell. Like System F, System F* usually has no `data` keyword. We use the `data` keywords here as if they are syntatic sugar.

In the body of function `persons_to_entries`, we make use of regular expression pattern matching to extract contents from a `person` datatype. For example, the first pattern applies if the input value is a sequence of values where the first value is a *Person* element containing a telephone number. The body of the pattern clause, we recursively call the function `persons_to_entries` which yields a value of type $Entry^{*}$. The overall expression is of type $\langle Entry, Entry^{*}\rangle$ which is a semantic subtype of $Entry^{*}$. A regular expression type t is said to be a semantic subtype of another regular expression type t' if the language denoted by t is a subset of the language denoted by t'. In this case, the type $\langle Entry, Entry^* \rangle$ denotes the set of *Entry* sequences whose lengths are greater than or equal to one. On the other hand the type $Entry^*$ denotes the set of *Entry* sequences whose lengths are greater than or equal to zero. Hence the first pattern clause is type correct. A similar observation applies to the second clause. The last clause only applies if the sequence is empty. Under semantic subtyping $\langle\rangle$ is a subtype of $Entry^*$. Hence, this clause is also type correct.

□

As demonstrated above, the real power of System F* is that thanks to regular expressions we can write expressive patterns/transformations and state powerful semantic subtype relations. This idea is well-explored in the context of monomorphically-typed languages such as XDuce [35, 36] and CDuce [24, 7]. Here, we transfer this idea to the setting of a polymorphically typed language.

Example 12 For instance, we rephrase the `mapStar` function mentioned earlier in System F*. The `mapStar` function applies a function, which takes a a and returns a b, to a sequence of as and yields a sequence of bs.

```
mapStar :  $\forall a, b.(a \rightarrow b) \rightarrow a^* \rightarrow b^*$
mapStar = $\Lambda a, b.\lambda f : (a \rightarrow b)\lambda(v : a^*)$
   case $v$ of
      $\langle x : a, xs : a^* \rangle \rightarrow \langle x,$ mapStar $a\ b\ f\ xs \rangle$
      $\langle\rangle \rightarrow \langle\rangle$
```

Like in System F, type abstraction and application are explicit. For example, we define `mapStar` in terms of a type abstraction. In the body of `mapStar`, we make a recursive call to `mapStar`, which needs to be first applied to the type arguments a and b then applied to the value arguments f and xs. □

Via these examples, we have a rough idea of System F*. In the following, we will look at the formal description of the language.

Declarations

$$\begin{array}{lll}
prog & ::= & \overline{decl}; e \\
decl & ::= & data\ T\ \overline{a} = \overline{K\ \overline{t}}
\end{array}$$

Types

$$\begin{array}{llll}
t & ::= & l \| r & \\
l & ::= & a \| T\ t_1 ... t_n \| t \rightarrow t \| \forall a.t & \text{Labels} \\
r & ::= & t^* & \text{Kleene star} \\
 & \| & (t \mid t) & \text{Choice} \\
 & \| & \langle\rangle & \text{Empty sequence} \\
 & \| & \langle t, t\rangle & \text{Pair sequence} \\
m & ::= & \langle l, t\rangle & \text{Monomials} \\
n & ::= & m \| \langle\rangle \| \langle n \mid n\rangle & \text{Normal form}
\end{array}$$

Expressions

$$\begin{array}{llll}
e & ::= & x \| K & \text{Variables and constructors} \\
 & \| & \lambda x : t.e \| e\ e & \text{Expr abstraction/application} \\
 & \| & \Lambda a.e \| e\ t & \text{Type abstraction/application} \\
 & \| & \mathsf{let}\ x : t = e\ \mathsf{in}\ e & \text{Let definition} \\
 & \| & \mathsf{case}\ e\ \mathsf{of}\ [p_i \rightarrow e_i]_{i \in I} & \text{Pattern matching} \\
 & \| & \langle\rangle & \text{Empty sequence} \\
 & \| & \langle e, e\rangle & \text{Pair sequence}
\end{array}$$

$$\begin{array}{llll}
p & ::= & x : t \| K\ \overline{t}\ p...p \| \langle\rangle \| \langle p, p\rangle & \text{Pattern}
\end{array}$$

Figure 4.1: Syntax of System F*

4.2 Syntax

We first consider the syntax of the language. The syntax of System F* is described in Figure 4.1. We use $\|$ in EBNF syntax to avoid confusion with the regular choice operator $|$.

The type language is mainly divided into two categories: Label types l and regular expression types r. The third and the fourth categories, monomials and their normal forms, will only become relevant when we discuss subtyping. A label types l is either a variable or a type built using the familiar data, function and polymorphic type constructors. A regular expression type r is built using regular expression operators such as Kleene star etc. The option operator $t?$ is syntactic

Environments
$\Gamma \ ::= \ \emptyset \| \{x : t\} \| \Gamma \cup \Gamma$

Constraints
$C \ ::= \ \emptyset \| \{t \leq t\} \| C \cup C$

Substitutions

v	$::=$	$a \| x$	Variables
o	$::=$	$t \| e$	Objects
θ	$::=$	$\{\} \| \{o/v\} \| \theta \cup \theta$	

Syntactic sugar
$t? \equiv t|\langle\rangle \qquad \bar{t} \equiv t_1...t_n$
$\{\overline{t/a}\} \equiv \{t_1/a_1\} \cup ... \cup \{t_n/a_n\}$
$\forall a_1, ..., a_n.t \equiv \forall a_1 ... \forall a_n.t$
$\forall t \equiv \forall a_1, ..., a_n.t \quad \text{where } fv(t) = \{a_1, ..., a_n\}$
$\Lambda a_1, ..., a_n.e \equiv \Lambda a_1 ... \Lambda a_n.e$
$\lambda x_1, ..., x_n.e \equiv \lambda x_1 ... \lambda x_n.e$

Figure 4.2: Syntactic Categories and Notations

sugar for $t|\langle\rangle$. We can arbitrarily mix label and regular expression types. Thus, we can effectively support regular hedges which are trees of regular expressions. As we will see later, regular hedges admit slightly stronger type relations which is in our opinion unnecessary for practical examples. Our sequences $\langle ... \rangle$ should not be confused with pairs as found in ML or Haskell. Sequences admit stronger type relations as compared to pairs. For example, sequences are associative and have $\langle\rangle$ as the identity.

Example 13 In System F*, $\langle t_1, \langle t_2, t_3\rangle\rangle = \langle\langle t_1, t_2\rangle, t_3\rangle$ and $\langle t, \langle\rangle\rangle = \langle\langle\rangle, t\rangle = t$ are valid equations, where t_1, t_2, t_3 and t are types in System F*and we consider $t_1 = t_2$ as the shorthand for $t_1 \leq t_2$ and $t_1 \geq t_2$. We write $t_1 \leq t_2$ to denote that t_1 is a subtype of t_2 and $t_1 \geq t_2$ to denote t_1 is a supertype of t_2. □

The expression language is the familiar one from System F extended with sequences, let definitions and pattern matching support. The types of constructors K of a data type T are recorded in an initial type environment. We assume that

$K : \forall a_1, ..., a_n.t_1 \rightarrow ... \rightarrow t_m\ T\ a_1...a_n \in \Gamma_{init}$ where $fv(t_1, ..., t_m) = \{a_1, ..., a_n\}$ and the function $fv(\cdot)$ computes the free variables in a type. Our patterns employ a mix of ML/Haskell style pattern matching over data types and regular expression pattern matching using sequences. We assume that K-patterns are always fully saturated, that is, a constructor typed as a n-ary function in a datatype declaration must be applied to exactly n sub-patterns when it is used in a pattern. Other works like [67] also support function and choice patterns which we do not support. We disallow choice patterns to keep the language simple. We disallow function patterns due to a technical reason, which will be discussed later in Chapter 5. Note that pattern variables must always carry a type annotation. As usual, we assume that variables in patterns are distinct.

Example 14 For example, in System F*, the pattern $\langle x : a, x : b\rangle$ is considered invalid, because the pattern variable x occurs more than once. □

Figure 4.2 contains further syntactic categories and notations which will become relevant when introducing the static and dynamic semantics of System F*. For example, we will write $\{t_2/a\}t_1$ for the capture avoiding substitution of variable a by type t_2 in the type t_1. Similarly, $\{e_2/x\}e_1$ stands for the capture avoiding substitution of variable x by expression e_2 in the expression e_1. We write $\{o_2/v_1\}o_1 \cup \{o_4/v_2\}o_3$ to denote the capture avoiding substitution of v_1 by o_2 in o_1 and of v_2 by o_4 in o_3. We will always assume that variables v_1 and v_2 are distinct. We write $\{\}$ to denote the identity substitution.

4.3 Static Semantics

We consider the static semantics of System F*. The static semantics is given in Figure 4.3. The first set of typing rules make use of judgments $\Gamma \vdash e : t$ to describe well-typing of expressions. Rules (Var) - (Let) contain no surprises and are already found in System F.

$$\boxed{\Gamma \vdash e : t}$$

$$\text{(Var)} \quad \frac{x : t \in \Gamma}{\Gamma \vdash x : t} \qquad \text{(EAbs)} \quad \frac{\Gamma \cup \{x : t_1\} \vdash e : t_2}{\Gamma \vdash \lambda x : t_1.e : t_1 \to t_2}$$

$$\text{(EApp)} \quad \frac{\Gamma \vdash e_1 : t_2 \to t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1\ e_2 : t_1} \qquad \text{(TAbs)} \quad \frac{\Gamma \vdash e : t \quad a \notin fv(\Gamma)}{\Gamma \vdash \Lambda a.e : \forall a.t}$$

$$\text{(TApp)} \quad \frac{\Gamma \vdash e : \forall a.t_1}{\Gamma \vdash e\ t_2 : \{t_2/a\}t_1} \qquad \text{(Let)} \quad \frac{\Gamma \cup \{x : t_1\} \vdash e_1 : t_1 \quad \Gamma \cup \{x : t_1\} \vdash e_2 : t_2}{\Gamma \vdash \mathsf{let}\ x : t_1 = e_1\ \mathsf{in}\ e_2 : t_2}$$

$$\text{(EmptySeq)} \quad \Gamma \vdash \langle\rangle : \langle\rangle \qquad \text{(PairSeq)} \quad \frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash \langle e_1, e_2\rangle : \langle t_1, t_2\rangle}$$

$$\text{(Sub)} \quad \frac{\Gamma \vdash e : t_1 \quad \vdash_{\text{sub}} t_1 \leq t_2}{\Gamma \vdash e : t_2}$$

$$\text{(Case)} \quad \frac{\Gamma \vdash e : t \quad \Gamma_i \vdash_{\text{pat}} p_i : t_i \quad \vdash_{\text{sub}} t_i \leq t \quad \Gamma \cup \Gamma_i \vdash e_i : t' \quad \text{for } i \in I}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ [p_i \to e_i]_{i \in I} : t'}$$

$$\boxed{\Gamma \vdash_{\text{pat}} p : t}$$

$$\emptyset \vdash_{\text{pat}} \langle\rangle : \langle\rangle \qquad \{x : t\} \vdash_{\text{pat}} (x : t) : t \qquad \frac{\Gamma_1 \vdash_{\text{pat}} p_1 : t_1 \quad \Gamma_2 \vdash_{\text{pat}} p_2 : t_2}{\Gamma_1 \cup \Gamma_2 \vdash_{\text{pat}} \langle p_1, p_2\rangle : \langle t_1, t_2\rangle}$$

$$\frac{\begin{array}{c}\Gamma_{init} \vdash K : \forall \overline{a}.t'_1 \to \ldots \to t'_m \to T\ \overline{a} \\ \Gamma_i \vdash_{\text{pat}} p_i : t''_i \quad \vdash_{\text{sub}} t''_i \leq \{\overline{t/a}\}t'_i \quad \text{for } i = 1, \ldots, m\end{array}}{\Gamma_1 \cup \ldots \cup \Gamma_m \vdash_{\text{pat}} K\ \overline{t}\ p_1 \ldots p_m : T\ \overline{t}}$$

Figure 4.3: System F* Typing Rules

Notice that let-defined functions can be recursive. See rule (Let) where we can make use of the type assumption $x : t_1$ when typing e_1. The remaining expression typing rules are non-standard. Rules (EmptySeq) and (PairSeq) allow us to build sequences. Via rule (Sub) we can change the type of an expression from t_1 to t_2 if t_1 and t_2 are in subtype relation. The set of valid subtype relations is described using a combination of a semantic subtype relation among regular expressions and a structural subtype relation among data, function and polymorphic types. The details are a few paragraphs away. Subtyping is also employed in the (Case) rule

$$\boxed{C \vdash_{\text{sub}} t_1 \leq t_2}$$

$$\text{(Hyp)}\ \frac{t_1 \leq t_2 \in C}{C \vdash_{\text{sub}} t_1 \leq t_2} \qquad \text{(Norm)}\ \frac{\vdash_{\text{norm}} t_1 \leadsto n_1 \quad \vdash_{\text{norm}} t_2 \leadsto n_2 \quad C \cup \{t_1 \leq t_2\} \vdash_{\text{lnf}} n_1 \leq n_2}{C \vdash_{\text{sub}} t_1 \leq t_2}$$

$$\boxed{C \vdash_{\text{lab}} l_1 \leq l_2}$$

$$\text{(Var)}\ C \vdash_{\text{lab}} a \leq a \qquad \text{(T)}\ \frac{\Gamma_{init} \vdash K : \forall \overline{a}.t''_1 \to ... \to t''_m \to T\ \overline{a} \quad C \vdash_{\text{sub}} \{\overline{t/a}\}t''_i \leq \{\overline{t'/a}\}t''_i \ \text{ for } i = 1, ..., m}{C \vdash_{\text{lab}} T\ \overline{t} \leq T\ \overline{t'}}$$

$$\text{(Arrow)}\ \frac{C \vdash_{\text{sub}} t'_1 \leq t_1 \quad C \vdash_{\text{sub}} t_2 \leq t'_2}{C \vdash_{\text{lab}} t_1 \to t_2 \leq t'_1 \to t'_2} \qquad \text{(Forall)}\ \frac{C \vdash_{\text{sub}} t_1 \leq t_2}{C \vdash_{\text{lab}} \forall a.t_1 \leq \forall a.t_2}$$

$$\boxed{C \vdash_{\text{lnf}} n_1 \leq n_2}$$

$$\text{(LE)}\ C \vdash_{\text{lnf}} \langle\rangle \leq \langle\rangle \qquad \text{(LN)}\ \frac{C \vdash_{\text{lab}} l_1 \leq l_2 \quad C \vdash_{\text{sub}} t_1 \leq t_2}{C \vdash_{\text{lnf}} \langle l_1, t_1\rangle \leq \langle l_2, t_2\rangle}$$

$$\text{(LR1)}\ \frac{C \vdash_{\text{lnf}} n_1 \leq n_2}{C \vdash_{\text{lnf}} n_1 \leq (n_2|n_3)}$$

$$\text{(LR2)}\ \frac{C \vdash_{\text{lnf}} n_1 \leq n_3}{C \vdash_{\text{lnf}} n_1 \leq (n_2|n_3)} \qquad \text{(LL)}\ \frac{C \vdash_{\text{lnf}} n_1 \leq n_3 \quad C \vdash_{\text{lnf}} n_2 \leq n_3}{C \vdash_{\text{lnf}} (n_1|n_2) \leq n_3}$$

Figure 4.4a: System F* Subtype Relation

via which we support pattern matching. The type t_i of each pattern p_i only needs to be a subtype of the type t of the case expression e.

The next set of rules concern typing of patterns using judgments of the form $\Gamma \vdash_{\text{pat}} p : t$. In the rule for constructors K we find again a use of subtyping. Thus, we can fully embed regular expression patterns inside "normal" data type constructor patterns.

Example 15 For instance, given data type declaration

data *List a* = *Cons a* (*List a*) | *Nil*

$$\boxed{\vdash_{\text{norm}}\ t \rightsquigarrow n}$$

$$\text{(N1)}\quad \frac{\vdash_{\text{empty}}\ \langle\rangle \in t}{\vdash_{\text{norm}}\ r \rightsquigarrow \langle\rangle | (|_{l \in \Sigma(t) \wedge pd(l|t) \neq \{\}} \langle l, d(l \mid t)\rangle)}$$

$$\text{(N2)}\quad \frac{\neg(\vdash_{\text{empty}}\ \langle\rangle \in t)}{\vdash_{\text{norm}}\ r \rightsquigarrow |_{l \in \Sigma(t) \wedge pd(l|t) \neq \{\}} \langle l, d(l \mid t)\rangle}$$

$$\boxed{\Sigma(t)}$$

$$\Sigma(a) = \{a\} \quad \Sigma(T\ t_1 ... t_n) = \{T\ t_1 ... t_n\}$$

$$\Sigma(t \to t') = \{t \to t'\} \quad \Sigma(\forall a.t) = \{\forall a.t\} \quad \Sigma(t^*) = \Sigma(t)$$

$$\Sigma(\langle t_1, t_2\rangle) = \Sigma(t_1) \cup \Sigma(t_2) \quad \Sigma(t_1|t_2) = \Sigma(t_1) \cup \Sigma(t_2)$$

$$\boxed{\vdash_{\text{empty}}\ \langle\rangle \in t}$$

$$\text{(ES)}\quad \vdash_{\text{empty}}\ \langle\rangle \in t^*$$

$$\text{(EE)}\quad \vdash_{\text{empty}}\ \langle\rangle \in \langle\rangle$$

$$\text{(EP)}\quad \frac{\vdash_{\text{empty}}\ \langle\rangle \in t_i \quad \forall i \in \{1,2\}}{\vdash_{\text{empty}}\ \langle\rangle \in \langle t_1, t_2\rangle}$$

$$\text{(EC)}\quad \frac{\vdash_{\text{empty}}\ \langle\rangle \in t_i \quad \exists i \in \{1,2\}}{\vdash_{\text{empty}}\ \langle\rangle \in (t_1|t_2)}$$

$$\boxed{d(l \mid t)}$$

$$d(l \mid t) = t_1|...|t_n$$
where $pd(l \mid t) = \{t_1, ..., t_n\}$ and $n > 0$

$$\boxed{\{t_1, ..., t_n\} \odot t}$$

$$\{\} \odot t = \{\} \qquad \{\langle\rangle\} \odot t = \{t\}$$
$\{t_1, ..., t_n\} \odot t = \{\langle (t_1|...|t_n), t\rangle\}$ where $n > 0$

$$\boxed{pd(l \mid t)}$$

$$pd(l \mid \langle\rangle) = \{\}$$

$$pd(l \mid t^*) = pd(l \mid t) \odot t^* \qquad pd(l_1 \mid l_2) = \begin{cases} \{\langle\rangle\} & ; l_1 = l_2 \\ \{\} & ; \mathit{otherwise} \end{cases}$$

$$pd(l \mid (t_1|t_2)) = pd(l \mid t_1) \cup pd(l \mid t_2)$$

$$pd(l \mid \langle t_1, t_2\rangle) = \begin{cases} pd(l \mid t_1) \odot t_2 \cup pd(l \mid t_2) & ; \vdash_{\text{empty}}\ \langle\rangle \in t_1 \\ pd(l \mid t_1) \odot t_2 & ; \mathit{otherwise} \end{cases}$$

Figure 4.4b: System F* Type Normalization

---

we can match a value of type $List\ a^*$ against the pattern $Cons\ (x : a)\ (xs : List\ a^*)$, because the following derivation is valid.

$$\frac{\begin{array}{c} \Gamma_{init} \vdash\ Cons : \forall b.b \to List\ b \to List\ b \\ \{x : a\} \vdash_{\text{pat}}\ (x : a) : a \qquad \{xs : List\ a^*\} \vdash_{\text{pat}}\ (xs : List\ a^*) : List\ a^* \\ \vdash_{\text{sub}}\ a \leq a^* \quad \vdash_{\text{sub}}\ List\ a^* \leq List\ a^* \end{array}}{\{x : a, xs : List\ a^*\} \vdash_{\text{pat}}\ Cons\ a\ (x : a)\ (xs : List\ a^*) : List\ a^*}$$

□

The subtype rules constitute the most complicated aspect of our system. In Figure 4.4a, we describe the valid subtype relation in System F*. The proof rules are defined in terms of judgments $C \vdash_{\text{sub}} t_1 \leq t_2$. We write $\vdash_{\text{sub}} t_1 \leq t_2$ for short if the constraint set C is empty. The constraint set C is necessary because our proof rules make use of co-induction. Cycles in subtype proofs typically arise in case of recursive types [10]. We do not support recursive types but cycles can still arise because of the Kleene star. The specification of our subtype proof system has an operational flavor in the sense that types represent states in a DFA and therefore subtyping among types corresponds to an inclusion testing among DFAs. We will come back to this point shortly.

Co-induction is used in rule (Norm) where we add the "to-be-proven statement" $t_1 \leq t_2$ as an assumption to the constraint set. In rule (Hyp) we can make use of such assumptions. To guarantee that this rule is sound we need to ensure that we make progress in a subtype proof. Otherwise, any statement $t_1 \leq t_2$ would hold trivially. We make progress by normalizing types and then switching to a proof system which checks for subtyping among normalized types.

In Figure 4.4b, we describe the type normalization for System F*. Normalization is carried out by judgments $\vdash_{\text{norm}} t \leadsto n$. A type t is normalized to the form $\langle l_1, t_1\rangle|...|\langle l_n, t_n\rangle$ where labels l_i refer to base types such as polymorphic variables, data, function and polymorphic types. Components $\langle l_i, t_i\rangle$ are referred to as monomials and expression t_i is the residual of t by removing the first label l_i. In essence, t_i represents the state of the underlying DFA after accepting the label l_i. The computation of the residual of a type (state) t for label l follows the standard definition employed for regular expressions [4, 59]. Function $pd(l \mid t)$ builds the set of states reachable (partial derivatives) from t after accepting l. The transitions and therefore the underlying automata seem to be indeterministic. However, function $d(l \mid t)$ then turns this set of states into a single state (derivative) by using the regular expression choice operator, thus, making the automata deterministic. This is of course only possible if the set is non-empty. The operation $\cdot \odot \cdot$ concatenates a set of states with a type to form a new set of states. We apply simplification rule $\langle\langle\rangle, t\rangle = t$. Rules (N1) and (N2) perform the actual normalization of types. We can guarantee that at least one of the sets computed via the $pd(\cdot \mid \cdot)$ will be non-empty. Thus, we can ensure that normalizing of types is well-defined.

If we ignore subtyping among labels, the normalized subtype proof system specified via judgments $C \vdash_{\text{lnf}} n_1 \leq n_2$ and rule (Norm) can be simplified as follows:

$$\frac{\begin{array}{c} C \cup \{t \leq t'\} \vdash_{\text{sub}} d(l_1 \mid t) \leq d(l_1 \mid t') \\ \ldots \\ C \cup \{t \leq t'\} \vdash_{\text{sub}} d(l_n \mid t) \leq d(l_n \mid t') \\ \Sigma(t') = \{l_1, ..., l_n\} \end{array}}{C \vdash_{\text{sub}} t \leq t'}$$

In the above, we assume that $pd(l_i \mid t)$ and $pd(l_i \mid t')$ are non-empty for each l_i arising out of type t'. For any valid subtype proof $C \vdash_{\text{sub}} t \leq t'$ the labels in t are a subset of the labels in t'. Therefore, for brevity we only compute the labels in t' via $\Sigma(t')$. For each label l_i we then check that $d(l_i \mid t)$ is a subtype of $d(l_i \mid t')$. In general, some of the $pd(l_i \mid t)$ and $pd(l_i \mid t')$ may be empty. In case $pd(l_i \mid t)$ is non-empty and $pd(l_i \mid t')$ is empty this immediately leads to failure. That is, the statement $C \vdash_{\text{sub}} t \leq t'$ does not hold.

Our normalized subtype proof rules support subtyping among labels. See rule (LN). Label subtyping is defined via judgments $C \vdash_{\text{lab}} l_1 \leq l_2$. The label subtyping rules should not contain any surprises. We apply the standard structural subtyping rules among data, function and polymorphic types [53].

In general, we also need to cover the case that types are empty. Emptiness of a type is defined via judgments $\vdash_{\text{empty}} \langle\rangle \in t$. In case types are formed using regular expression operators we need to check the subcomponents for emptiness. See rules (ES), (EP) and (EC). Rule (EE) represents the base case. Labels are always non-empty.

Normalization:

$$\dfrac{\begin{array}{c}\vdash_{\text{empty}}\ \langle\rangle \in \langle A^*, A^*\rangle\\ \Sigma(\langle A^*, A^*\rangle) = \{A\}\\ pd(A \shortmid \langle A^*, A^*\rangle) = \{\langle A^*, A^*\rangle, A^*\}\\ d(A \shortmid \langle A^*, A^*\rangle) = (\langle A^*, A^*\rangle | A^*)\end{array}}{\vdash_{\text{norm}}\ \langle A^*, A^*\rangle \rightsquigarrow (\langle\rangle | \langle A, (\langle A^*, A^*\rangle | A^*)\rangle)}\,(\text{N1}) \quad \dfrac{\begin{array}{c}\vdash_{\text{empty}}\ \langle\rangle \in A^*\\ \Sigma(A^*) = \{A\}\\ pd(A \shortmid A^*) = \{A^*\}\\ d(A \shortmid A^*) = A^*\end{array}}{\vdash_{\text{norm}}\ A^* \rightsquigarrow (\langle\rangle | \langle A, A^*\rangle)}\,(\text{N1}) \quad \dfrac{\begin{array}{c}\vdash_{\text{empty}}\ \langle\rangle \in (\langle A^*, A^*\rangle | A^*)\\ \Sigma((\langle A^*, A^*\rangle | A^*)) = \{A\}\\ pd(A \shortmid (\langle A^*, A^*\rangle | A^*)) = \{\langle A^*, A^*\rangle, A^*\}\\ d(A \shortmid (\langle A^*, A^*\rangle | A^*)) = (\langle A^*, A^*\rangle | A^*)\end{array}}{\vdash_{\text{norm}}\ (\langle A^*, A^*\rangle | A^*) \rightsquigarrow (\langle\rangle | \langle A, (\langle A^*, A^*\rangle | A^*)\rangle)}\,(\text{N1})$$

Constraints:

$$C_1 = \{\langle A^*, A^*\rangle \le A^*\} \quad C_2 = C_1 \cup \{(\langle A^*, A^*\rangle | A^*) \le A^*\}$$

Main Proof:

$$\dfrac{\dfrac{}{C_1 \vdash_{\text{lnf}}\ \langle\rangle \le \langle\rangle}\,(\text{LE}) \quad \begin{array}{c}\dfrac{\dfrac{\begin{array}{c}\dfrac{}{C_2 \vdash_{\text{lnf}}\ \langle\rangle \le \langle\rangle}\,(\text{LE}) \quad \dfrac{\dfrac{(\langle A^*, A^*\rangle | A^*) \le A^* \in C_2}{C_2 \vdash_{\text{sub}}\ (\langle A^*, A^*\rangle | A^*) \le A^*}\,(\text{Hyp})}{C_2 \vdash_{\text{lnf}}\ \langle A, (\langle A^*, A^*\rangle | A^*)\rangle \le \langle A, A^*\rangle}\,(\text{LN})\\ \vdash_{\text{norm}}\ (\langle A^*, A^*\rangle | A^*) \rightsquigarrow (\langle\rangle | \langle A, (\langle A^*, A^*\rangle | A^*)\rangle) \quad \vdash_{\text{norm}}\ A^* \rightsquigarrow (\langle\rangle | \langle A, A^*\rangle)\end{array}}{C_1 \vdash_{\text{sub}}\ (\langle A^*, A^*\rangle | A^*) \le A^*}\,(\text{Norm})}{C_1 \vdash_{\text{lnf}}\ \langle A, (\langle A^*, A^*\rangle | A^*)\rangle \le \langle A, A^*\rangle}\,(\text{LN})\\ \vdash_{\text{norm}}\ \langle A^*, A^*\rangle \rightsquigarrow (\langle\rangle | \langle A, (\langle A^*, A^*\rangle | A^*)\rangle) \quad \vdash_{\text{norm}}\ A^* \rightsquigarrow (\langle\rangle | \langle A, A^*\rangle)\end{array}}{\{\} \vdash_{\text{sub}}\ \langle A^*, A^*\rangle \le A^*}\,(\text{Norm})$$

Figure 4.5: A subtype proof of $\vdash_{\text{sub}}\ \langle A^*, A^*\rangle \le A^*$

Checking for emptiness:

$$\dfrac{\dfrac{}{\vdash_{\text{empty}} \langle\rangle \in A^*}\,(\text{ES})}{\vdash_{\text{empty}} \langle\rangle \in \langle A^*, A^*\rangle}\,(\text{EP}) \qquad \dfrac{}{\vdash_{\text{empty}} \langle\rangle \in A^*}\,(\text{ES}) \qquad \dfrac{\vdash_{\text{empty}} \langle\rangle \in \langle A^*, A^*\rangle \quad \vdash_{\text{empty}} \langle\rangle \in A^*}{\vdash_{\text{empty}} \langle\rangle \in (\langle A^*, A^*\rangle | A^*)}\,(\text{EC})$$

Computing partial derivatives:

$$\begin{array}{cl} & pd(A \mid \langle A^*, A^*\rangle) \\ = & \{\langle t, A^*\rangle | t \in pd(A \mid A^*)\} \cup pd(A \mid A^*) \\ = & \{\langle\langle\langle\rangle, A^*\rangle, A^*\rangle\} \cup \{\langle\langle\rangle, A^*\rangle\} \\ = & \{\langle\langle\langle\rangle, A^*\rangle, A^*\rangle, \langle\langle\rangle, A^*\rangle\} \end{array}$$

$$\begin{array}{cl} & pd(A \mid A^*) \\ = & \{\langle t, A^*\rangle | t \in pd(A \mid A)\} \\ = & \{\langle\langle\rangle, A^*\rangle\} \end{array}$$

$$pd(A \mid A) = \{\langle\rangle\}$$

$$\begin{array}{cl} & pd(A \mid (\langle A^*, A^*\rangle | A^*)) \\ = & pd(A \mid \langle A^*, A^*\rangle) \cup pd(A \mid A^*) \\ = & \{\langle\langle\langle\rangle, A^*\rangle, A^*\rangle, \langle\langle\rangle, A^*\rangle\} \cup \{\langle\langle\rangle, A^*\rangle\} \\ = & \{\langle\langle\langle\rangle, A^*\rangle, A^*\rangle, \langle\langle\rangle, A^*\rangle\} \end{array}$$

Figure 4.6: A subtype proof of $\vdash_{\text{sub}} \langle A^*, A^*\rangle \leq A^*$ (Cont'd.)

Example 16 Figures 4.5 and 4.6 give the proof of the statement $\vdash_{\text{sub}} \langle A^*, A^* \rangle \leq A^*$. We assume that $(A : A) \in \Gamma_{init}$. That is, value A is of singleton type. We read the proof from bottom to top. Hence, rule applications should be interpreted as reduction steps. We first normalize $\langle A^*, A^* \rangle$ to the form $(\langle\rangle | \langle A, (\langle A^*, A^* \rangle | A^*) \rangle)$ and A^* to the form $(\langle\rangle | \langle A, A^* \rangle)$. The top part of the figure contains the sub-calculations necessary to carry out the normalization steps. Then, we proceed and compare the normal forms via the normalized proof rules. We shorten the (Norm) rule step slightly by immediately breaking apart the normal forms and compare their respective monomials and empty sequences. This leads to $C_1 \vdash_{\text{lnf}} \langle\rangle \leq \langle\rangle$ and $C_1 \vdash_{\text{lnf}} \langle A, (\langle A^*, A^* \rangle | A^*) \rangle \leq \langle A, A^* \rangle$ where constraint C_1 now contains the "to-be-proven statement" $\langle A^*, A^* \rangle \leq A^*$. The first statement $C_1 \vdash_{\text{lnf}} \langle\rangle \leq \langle\rangle$ can be verified immediately. The second statement is reduced via rule (LN) to $C_1 \vdash_{\text{sub}} (\langle A^*, A^* \rangle | A^*) \leq A^*$. We perform a further normalization step which then eventually leads to $C_2 \vdash_{\text{sub}} (\langle A^*, A^* \rangle | A^*) \leq A^*$. The constraint C_2 contains this statement. Hence, we can reduce this statement via rule (Hyp) which concludes the proof. □

We can mix structural and semantic subtyping. For example, suppose we have a data type T with the single constructor $K : \forall a.a \to T\ a$. Based on the above calculations and the label subtyping rule (T), we can verify that $\vdash_{\text{sub}} T\ \langle A^*, A^* \rangle \leq T\ A^*$. However, structural and semantic subtyping are strictly separated from each other. The statement $\vdash_{\text{sub}} (T\ A \mid T\ B) \leq T\ (A \mid B)$ is not provable because regular expression operators such as choice do not distribute over data types.

In XDuce, the statement $(\langle A[B], C \rangle | \langle A[C], B \rangle) \leq \langle A[(B|C)], (C|B) \rangle$ is valid (where A is a tag) because XDuce supports regular hedge types which enjoy more expressive subtype relations compared to data types. In theory, it's possible to add regular hedge types plus the additional subtyping rules to our system. For example, see our earlier work in [47]. In our experience, the combination of regular expression and data types is sufficient, therefore, the extra complexity of regular hedges unnecessary.

Note that so far, we omit exhaustiveness check for patterns. Exhaustiveness guarantees that a pattern will not get "stuck" during run-time. In the context of regular expression pattern alone, the exhaustiveness of the pattern can be verified by checking whether the incoming type is a subtype of the union of the pattern's types.

Example 17 For instance, we consider

```
countA :: $(A|B)^* \rightarrow Int$
countA = $\lambda x : (A|B)^*$
       case $x$ of
       $\langle\rangle \rightarrow 0$
       $\langle x : B^*, xs : (A|B)^*\rangle \rightarrow$ countA $xs$
       $\langle x : A, xs : (A|B)^*\rangle \rightarrow 1 +$ (countA $xs$)
```

The above pattern is exhaustive, because the incoming type $(A|B)^*$ is a subtype of pattern's types $(\langle\rangle|\langle B^*, (A|B)^*\rangle|\langle A, (A|B)^*\rangle)$. □

In general, to check whether the pattern $(p_1|...|p_n)$ is exhaustive given an input type t, we need to extract the types from the patterns say $(t_1|...|t_n)$, then we check whether $t \leq (t_1|...|t_n)$. This checking technique is also employed by XDuce [30] and CDuce [21]. In the presence of data types, this technique is not applicable any more.

Example 18 Consider,

```
buggy_count :: $\forall a.(List\ a) \rightarrow Int$
buggy_count = $\Lambda a \lambda x : (List\ a)$
            case $x$ of
            $Cons\ a\ (x : a)\ (xs : (List\ a)) \rightarrow 1 +$ (buggy_count $a$ $xs$)
```

The above pattern is *not* exhaustive, because the Nil case is not handled. On the other hand, if we apply the above-mentioned techinque, it is obvious that we can conclude that the pattern has type *List a*. It follows that $\vdash_{\text{sub}}$ *List a* $\leq$ *List a*. That means this technique is not applicable. □

Thus we conclude that the exhaustiveness check via subtyping is too weak to handle data type patterns. The checking mechanism is beyond the scope of this thesis. We believe that static analysis techniques such as [50] can be applied here.

4.4 Dynamic Semantics

The dynamic semantics of System F* is defined in Figure 4.7 via a (strict) small-step operational semantics [71]. The reduction rules for expressions are standard. The interesting part is the pattern matching relation $w \lhd p \leadsto \theta$ which states that matching the value w against the pattern p yields the matching substitution θ. In rule (Case) we use the pattern matching relation to select a pattern clause. We leave the order in which we select pattern clauses unspecified. In a concrete implementation, we could employ a top to bottom selection strategy. We also do not catch pattern matching failure which therefore results in a "stuck" expression. Let us take a closer look at the pattern matching relation which is a mix of pattern matching based on structure, see rule (Pat-K), and unstructured pattern matching, see rule (Pat-Seq).

Rule (Pat-Var) deals with variable patterns. We use here the type attached to each pattern variable to perform the matching by checking whether w has type t in the initial type environment. This means that our semantics is type-based and we therefore cannot discard (erase) type information at run-time. Rule (Pat-K) is the standard pattern matching rule also found in ML and Haskell. Rule (Pat-$\langle\rangle$) matches the empty sequence value against the empty sequence pattern. In rule (Pat-Seq), we pattern match against sequences. Via the statement $w \sim \langle w_1, w_2 \rangle$ we split the value w into two sub-components w_1 and w_2 which we then match against the sub-patterns p_1 and p_2. Pattern variables are distinct. Hence, there will not be any clashes when combining the matching substitutions θ_1 and θ_2. Splitting of values into sub-

Values

$$w \quad ::= \quad \Lambda a.e \| \lambda x : t.e \| K\ \bar{t}\ w_1 ... w_n \| \langle\rangle \| \langle w, w \rangle$$

Evaluation contexts:

$$\frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']} \qquad \begin{array}{rcl} E & ::= & [\,] \| E\ w \| E\ t \| K\ \bar{t}\ E...E \\ & \| & \langle E, E\rangle \| \mathsf{let}\ x : t = E\ \mathsf{in}\ e \\ & \| & \mathsf{case}\ E\ \mathsf{of}\ [p_i \rightarrow e_i]_{i \in I} \end{array}$$

Reduction rules

$$\begin{array}{ll}
\text{(TBeta)} & (\Lambda a.e)\ t \quad \longrightarrow \quad \{t/a\}e \\
\text{(Beta)} & (\lambda x.e)\ w \quad \longrightarrow \quad \{w/x\}e \\
\text{(Let)} & \mathsf{let}\ x : t = w\ \mathsf{in}\ e \quad \longrightarrow [w/x]e \\
\text{(Case)} & \dfrac{w \lhd p_j \rightsquigarrow \theta \quad \text{for some } j \in I}{\mathsf{case}\ w\ \mathsf{of}\ [p_i \rightarrow e_i]_{i \in I} \longrightarrow \theta(e_j)} \\
\text{(Pat-Var)} & \dfrac{\Gamma_{init} \vdash w : t}{w \lhd (x : t) \rightsquigarrow \{w/x\}} \\
\text{(Pat-K)} & \dfrac{w_i \lhd p_i \rightsquigarrow \theta_i \quad \text{for } i = 1, ..., n}{K\ \bar{t}\ w_1...w_n \lhd K\ \bar{t'}\ p_1...p_n \rightsquigarrow \theta_1 \cup ... \cup \theta_n} \\
\text{(Pat-}\langle\rangle\text{)} & \langle\rangle \lhd \langle\rangle \rightsquigarrow \{\} \\
\text{(Pat-Seq)} & \dfrac{w \sim \langle w_1, w_2\rangle \quad w_1 \lhd p_1 \rightsquigarrow \theta_1 \quad w_2 \lhd p_2 \rightsquigarrow \theta_2}{w \lhd \langle p_1, p_2\rangle \rightsquigarrow \theta_1 \cup \theta_2}
\end{array}$$

$$\text{(Pat-Norm)} \quad \begin{array}{c} (\langle\langle w_1, w_2\rangle, w_3\rangle \succ \langle w_1, \langle w_2, w_3\rangle\rangle \\ \dfrac{w_1 \succ^* w \quad w_2 \succ^* w}{w_1 \sim w_2} \end{array} \qquad \begin{array}{c} \langle w, \langle\rangle\rangle \succ w \\ \\ \langle\langle\rangle, w\rangle \succ w \end{array}$$

Figure 4.7: Operational Semantics

components is performed by rule (Pat-Norm). $\succ$ denotes the sequence normalization operation. The rule $\langle\langle w_1, w_2\rangle, w_3\rangle \succ \langle w_1, \langle w_2, w_3\rangle\rangle$ normalizes a word from the left associated form into the right associated form by applying the associativity law. The rules $\langle w, \langle\rangle\rangle \succ w$ and $\langle\langle\rangle, w\rangle \succ w$ remove redundent empty sequences by applying the identity law.

Example 19 We normalize the word $\langle\langle A, \langle A, \langle\rangle\rangle\rangle, A\rangle$ to $\langle A, \langle A, A\rangle\rangle$ by applying the

$\succ$ rules

$$\langle\langle A,\langle A,\langle\rangle\rangle\rangle,A\rangle \succ \langle\langle A,A\rangle,A\rangle \succ \langle A,\langle A,A\rangle\rangle$$

□

$\succ^*$ denotes the reflexive and transitive closure of $\succ$. We only require that w and $\langle w_1,w_2\rangle$ have the same normal form. We compute the normal form by applying associativity and identity laws for sequences. Pattern matching is therefore indeterministic as shown by the following example.

Example 20 We find that $\langle A,\langle A,A\rangle\rangle \sim \langle A,\langle A,A\rangle\rangle$ and $\langle A,\langle A,A\rangle\rangle \sim \langle\langle A,A\rangle,A\rangle$ where A is value of singleton type. Hence,

$$\langle A,\langle A,A\rangle\rangle \lhd \langle x:A^*,y:A^*\rangle \rightsquigarrow \{A/x,\langle A,A\rangle/y\} \qquad (1)$$

$$\langle A,\langle A,A\rangle\rangle \lhd \langle x:A^*,y:A^*\rangle \rightsquigarrow \{\langle A,A\rangle/x,A/y\} \qquad (2)$$

are two possible pattern matching results. In some intermediate steps of the derivation (1) we find $A \lhd x : A^* \rightsquigarrow \{A/x\}$ and $\langle A,A\rangle \lhd y : A^* \rightsquigarrow \{\langle A,A\rangle/y\}$ because of $\Gamma_{init} \vdash A : A^*$ and $\Gamma_{init} \vdash \langle A,A\rangle : A^*$. The last two statements are derived from $\vdash_{\text{sub}} A \leq A^*$ and $\vdash_{\text{sub}} \langle A,A\rangle \leq A^*$. □

In a concrete implementation, we can make pattern matching deterministic by, for example, applying the POSIX policy [66]. We replace rule (Pat-Seq) by the following rule.

$$\frac{\begin{array}{c} w \sim \langle w_1,w_2\rangle \quad w_1 \lhd_{\text{lm}} p_1 \rightsquigarrow \theta_1 \quad w_2 \lhd_{\text{lm}} p_2 \rightsquigarrow \theta_2 \\ \neg\left(\begin{array}{c} \exists w_3,w_4 : \neg(w_3 \sim \langle\rangle) \wedge \langle w_3,w_4\rangle \sim w_2 \wedge \\ \langle w_1,w_3\rangle \lhd_{\text{lm}} p_1 \rightsquigarrow \theta_1' \wedge w_4 \lhd_{\text{lm}} p_2 \rightsquigarrow \theta_2' \end{array}\right) \end{array}}{w \lhd_{\text{lm}} \langle p_1,p_2\rangle \rightsquigarrow \theta_1 \cup \theta_2}$$

The above rule says that when a value w is matched against $\langle p_1, p_2\rangle$, the sub-pattern p_1 will consume the longest possible prefix from w while the remaining suffix is consumed by sub-pattern p_2. This is the POSIX/Longest matching policy. For example, we find that $\langle A, \langle A, A\rangle\rangle \lhd_{\text{lm}} \langle x : A^*, y : A^*\rangle \leadsto \{\langle A, \langle A, A\rangle\rangle/x, \langle\rangle/y\}$.

4.5 Type Checking, Type Soundness and Semantic Subtyping

We establish some essential properties of System F* such as decidability of type checking and type soundness. First, we take a look at type checking.

In System F, type checking is completely deterministic because lambda-bound, let-defined and pattern variables carry a type annotation and type abstraction and application is made explicit in the expression language. Type checking in System F* is slightly less deterministic because of the non-syntax directed (Sub) rule. However, we can easily make the typing rules syntax-directed by integrating rule (Sub) with rules (App), (Let) and (Case). This is the standard approach and we omit the details for brevity. The point is that type checking in System F* reduces to checking for subtyping among types whereas in System F we only need to check for syntactic type equivalence (modulo variable renaming).

However, subtyping in System F* is potentially undecidable because of *nested data type definitions* [9]. Informally, a nested datatype is a parametrized datatype, one of whose value constructors is taking a "larger" instance of the datatype as the argument.

For example, consider the data type T with constructor $K : \forall a.T\ [a] \to T\ a$, where we use $[a]$ as a short hand of $(List\ a)$. This data type is nested because the constructor's argument is of type $T\ [a]$ which is further nested compared to the result type $T\ a$. The trouble with nested data types is that when trying to verify $\vdash_{\text{sub}} T\ a \leq T\ b$ we encounter in an intermediate step $\vdash_{\text{sub}} T\ [a] \leq T\ [b]$ because of label subtype proof rule (T). We are clearly in a cycle and checking for subtyping is therefore potentially undecidable. One possibility to recover decidability is to restrict label subtyping. We replace rule (LN) in Figure 4.3 via the following rule:

$$\text{(LN')} \quad \frac{C \vdash_{\text{sub}} t_1 \leq t_2}{C \vdash_{\text{lnf}} \langle l, t_1 \rangle \leq \langle l, t_2 \rangle}$$

The following lemma states that subtyping is decidable if we restrict label subtyping.

Lemma 1 (Decidability of Subtyping I) *If we omit subtyping among labels, then for any two types t and t' we can decide whether $\vdash_{sub} t \leq t'$ is valid or not.*

It follows that the type checking process is decidable, too.

Theorem 1 (Decidability of Type Checking I) *If we omit subtyping among labels, then we can decide whether $\Gamma \vdash e : t$ holds or not.*

Note that omitting label subtyping is not an onerous restriction. We can always mimic it by writing explicit coercion functions.

Example 21 The following function making use of label subtyping

```
label_subtype :: $\forall a.(List\ a^*) \rightarrow (List\ a)$
label_subtype = $\Lambda a \lambda x : (List\ a^*)$
       case $x$ of
        $(y : (List\ a)) \rightarrow y$
```

can be redefined as follows,

```
no_label_subtype :: $\forall a.(List\ a^*) \rightarrow (List\ a)$
no_label_subtype = $\Lambda a \lambda x : (List\ a^*)$
          case $x$ of
           $(Cons\ a\ (y : a)\ (ys : (List\ a^*))) \rightarrow Cons\ a\ y$ (no_label_subtype $a\ ys$)
           $Nil \rightarrow Nil$
```

which does not make use of label subtyping. □

Another possibility to recovery decidability is to simply reject nested data types. We first give a proper definition of non-nested datatypes.

Definition 1 (Strongly-connected Data Types) *We say that two data types T and T' are strongly-connected if there are constructors $K : \forall a_1, ..., a_n.t_1 \rightarrow ... \rightarrow t_m\ T\ a_1...a_n$ and $K' : \forall a_1, ..., a_l.t'_1 \rightarrow ... \rightarrow t'_k\ T\ a_1...a_l$ and T occurs in some t'_i and T' occurs in some t_j. This also covers the special case that $T = T'$ and $K = K'$. In other words, T and T' appear (indirectly) in each other's definition.*

In a nutshell strongly-connected datatypes refer to a group of datatypes whose definitions are recursively related to each other.

Example 22 For instance,

```
data T a = K (T’ a) | N

data T’ a = K’ (T a)
```

datatypes T and T' are strongly-connected. □

Definition 2 (Non-nested Data Types) *We say that a data type T is* non-nested *iff for each of its constructors $K : \forall a_1, ..., a_n.t_1 \rightarrow ... \rightarrow t_m\ T\ a_1...a_n$ and each occurrence of a strongly-connected data type T' in some t_i is of the form $T'\ b_1...b_k$ where $\{b_1, ..., b_k\} \subseteq \{a_1, ..., a_n\}$. We say a type t is* non-nested *if it is not composed of any nested data types.*

Example 23 For example the following data type $(T\ a)$ is nested.

```
data T a = K (T’ a)

data T’ a = K’ (T [a])
```

Because in the definition of $T'\ a$ which is a strongly connected data type of $T\ a$, $T\ [a]$ appears in the argument position. Proving $\vdash_{\text{sub}}\ T\ a \leq T\ a$ will lead to an infinite derivation tree consisting of $\vdash_{\text{sub}}\ T\ [a] \leq T\ [a]$, $\vdash_{\text{sub}}\ T\ [[a]] \leq T\ [[a]]$ and etc. □

The next lemma says that if we restrict nested-datatype, the subtyping is decidable.

Lemma 2 (Decidability of Subtyping II) *Let Γ_{init} be an initial type environment which only contains non-nested data types. Then, for any two types t and t' we can decide whether $\vdash_{sub}\ t \leq t'$ is valid or not.*

Then the type checking is also decidable if there is no nested-datatype.

Theorem 2 (Decidability of Type Checking II) *Let Γ be a type environment which only contains non-nested data types, e an expression and t a type. Then, we can decide whether $\Gamma \vdash e : t$ holds or not.*

To establish type soundness we need to show that types are preserved when performing reduction steps (also known as subject reduction) and the evaluation of expressions will not get stuck (also known as progress). The first property follows straightforwardly.

Theorem 3 *(Subject Reduction) Let e and e' be System* F^* *expressions and t be a type such that $\Gamma \vdash e : t$ and $e \longrightarrow e'$. Then $\Gamma \vdash e' : t$.*

The progress property only holds if patterns are exhaustive. In case patterns are non-exhaustive evaluation gets stuck. We could catch such cases by adding a default (always matching) pattern clause.

Theorem 4 *(Progress) Let e be a System* F^* *expression and t be a type such that $\Gamma_{init} \vdash e : t$ and all patterns in e are exhaustive. Then either e is a value or there exists e' such that $e \longrightarrow e'$.*

We write $L(r)$ to denote the language described by a regular expression r. The following lemma states that System F^* subtype system presented in Figures 4.4a and 4.4b is a semantic subtyping system if we restrict the label type l.

Lemma 3 *(Semantic Subtyping) If we restrict label types l to be data types of form data $T = T$, then for any two types t and t', we have $\vdash_{sub}\ t \leq t'$ iff $L(t) \subseteq L(t')$.*

A reader who is curious about the proofs of the semantic subtyping lemma, the decidability lemmas and theorems may find them in Appendix B Section B.1. The proofs for the Subject Reduction and Progress theorems are standard practices [71], thus we omit the details.

A property which does not carry over from System F to System F* is type erasure. Type erasure means that we erase all types as well as type application and abstraction from typed expressions (respectively replacing them by expression abstraction/application). In case of System F, type erasure will not change the meaning of programs [53]. The situation is different for System F*. The operational semantics described in Figure 4.7 relies on type information to perform the pattern match. See rule (Pat-Var). The consequence is that the System F* semantics must carry around additional type parameters which causes some overhead. More seriously, it is impossible to statically compile pattern matches because the actual pattern match relies on dynamic type information. Our goal is to address this issue by giving a more elementary semantics to System F* expressions which admits type erasure. This is the topic of the next section.

4.6 Summary

We have formalized the syntax and semantics of System F*. We show that the type checking in System F* is decidable under certain conditions and the subjection reduction and progress results show that the type system we developed is sound. In the next chapter, we are going to develop a static compilation scheme for System F*.

Chapter 5

Translation Scheme from System F* to System F

We have defined the syntax and semantics of System F* in Chapter 4. In this chapter, we study how to develop a compilation scheme for System F*.

We adopt a popular compilation technique known as "source-to-source translation" to compile System F* programs into System F with data types.

We use a structured representation of values of regular expression types. Semantic subtyping is translated by extracting proof terms out of subtype proofs which are inserted in the translated program. Similarly, we translate regular expression pattern matching to pattern matching over structured data (for which efficient compilation schemes exist).

The layout of this chapter is as follows. We first briefly look at the syntax and semantics of the target language System F. We then discuss how to derive coercion functions out of the subtype proof, followed by how semantic subtyping and pattern matching can be translated by using these coercion functions. Finally, we show that our translation is type-preserving.

Declarations

$$\begin{array}{lcl} prog & ::= & \overline{decl}; E \\ decl & ::= & data\ T\ \overline{a} = \overline{K\ \overline{t}} \end{array}$$

Types

$$t \ ::= \ a \| T\ t_1 ... t_n \| t \rightarrow t \| \forall a.t$$

Expressions

$$\begin{array}{lcll} E & ::= & x \| K & \text{Variables and constructors} \\ & \| & \lambda x : t.E \| E\ E & \text{Expr abstraction/application} \\ & \| & \Lambda a.E \| E\ t & \text{Type abstraction/application} \\ & \| & \mathsf{let}\ x : t = E\ \mathsf{in}\ E & \text{Let definition} \\ & \| & \mathsf{case}\ E\ \mathsf{of}\ [P_i \rightarrow E_i]_{i \in I} & \text{Pattern matching} \\ P & ::= & x \| K\ \bar{t}\ P ... P & \text{Pattern} \\ v & ::= & \Lambda a.E \| \lambda x : t.E \| K\ \bar{t}\ v_1 ... v_n & \text{Values} \end{array}$$

Evaluation contexts:

$$\frac{E \longrightarrow E'}{F[E] \longrightarrow F[E']} \qquad \begin{array}{lcl} F & ::= & [\,] \| F\ v \| F\ t \| K\ \bar{t}\ F ... F \| \mathsf{let}\ x : t = F\ \mathsf{in}\ E \\ & \| & \mathsf{case}\ F\ \mathsf{of}\ [P_i \rightarrow E_i]_{i \in I} \end{array}$$

Reduction rules

$$\begin{array}{ll} \text{(TBeta)} & (\Lambda a.E)\ t \ \longrightarrow\ \{t/a\}E \\ \text{(Beta)} & (\lambda x.E)\ v \ \longrightarrow\ \{v/x\}E \\ \text{(Let)} & \mathsf{let}\ x : t = v\ \mathsf{in}\ E \ \longrightarrow [v/x]E \\ \text{(Case)} & \dfrac{v \lhd_{\mathrm{F}} P_j \rightsquigarrow \theta \quad \text{for some } j \in I}{\mathsf{case}\ v\ \mathsf{of}\ [P_i \rightarrow E_i]_{i \in I} \longrightarrow \theta(E_j)} \\ \text{(Pat-Var)} & v \lhd_{\mathrm{F}} x \rightsquigarrow \{v/x\} \\ \text{(Pat-K)} & \dfrac{v_i \lhd_{\mathrm{F}} P_i \rightsquigarrow \theta_i \quad \text{for } i = 1, ..., n}{K\ \bar{t}\ v_1 ... v_n \lhd_{\mathrm{F}} K\ \overline{t'}\ P_1 ... P_n \rightsquigarrow \theta_1 \cup ... \cup \theta_n} \end{array}$$

Figure 5.1: Syntax and Operational Semantics of System F

5.1 System F with Data Types

We first take a brief look at the target language System F. In Figure 5.1, we describe the syntax and operational semantics of System F. We use E to denote System F expressions, v to denote System values, and P to denote System F patterns in order to distinguish from their System F*counter-parts.

Note that the static semantics of System F is simpler compared to System F*, because there is no semantic subtyping. We refer to Figure 5.2 for the typing rules.

$$\boxed{\Gamma \vdash_{\mathrm{F}} E : t}$$

$$\text{(Var)}\ \frac{x : t \in \Gamma}{\Gamma \vdash_{\mathrm{F}} x : t} \qquad \text{(EAbs)}\ \frac{\Gamma \cup \{x : t_1\} \vdash_{\mathrm{F}} E : t_2}{\Gamma \vdash_{\mathrm{F}} \lambda x : t_1.E : t_1 \to t_2}$$

$$\text{(EApp)}\ \frac{\Gamma \vdash_{\mathrm{F}} E_1 : t_2 \to t_1 \quad \Gamma \vdash_{\mathrm{F}} E_2 : t_2}{\Gamma \vdash_{\mathrm{F}} E_1\ E_2 : t_1} \qquad \text{(TAbs)}\ \frac{\Gamma \vdash_{\mathrm{F}} E : t \quad a \notin fv(\Gamma)}{\Gamma \vdash_{\mathrm{F}} \Lambda a.E : \forall a.t}$$

$$\text{(TApp)}\ \frac{\Gamma \vdash_{\mathrm{F}} E : \forall a.t_1}{\Gamma \vdash_{\mathrm{F}} E\ t_2 : \{t_2/a\}t_1} \qquad \text{(Let)}\ \frac{\begin{array}{c}\Gamma \cup \{x : t_1\} \vdash_{\mathrm{F}} E_1 : t_1 \\ \Gamma \cup \{x : t_1\} \vdash_{\mathrm{F}} E_2 : t_2\end{array}}{\Gamma \vdash_{\mathrm{F}} \mathsf{let}\ x : t_1 = E_1\ \mathsf{in}\ E_2 : t_2}$$

$$\text{(Case)}\ \frac{\Gamma \vdash_{\mathrm{F}} E : t \quad \Gamma_i \vdash_{\mathrm{pat}} P_i : t \quad \Gamma \cup \Gamma_i \vdash_{\mathrm{F}} E_i : t' \quad \text{for } i \in I}{\Gamma \vdash_{\mathrm{F}} \mathsf{case}\ E\ \mathsf{of}\ [P_i \to E_i]_{i \in I} : t'}$$

$$\boxed{\Gamma \vdash_{\mathrm{pat}} P : t}$$

$$\{x : t\} \vdash_{\mathrm{pat}} x : t \qquad \frac{\begin{array}{c}\Gamma_{init} \vdash K : \forall \overline{a}.t'_1 \to \ldots \to t'_m \to T\ \overline{a} \\ \Gamma_i \vdash_{\mathrm{pat}} P_i : \{\overline{t/a}\}t'_i \quad \text{for } i = 1, \ldots, m\end{array}}{\Gamma_1 \cup \ldots \cup \Gamma_m \vdash_{\mathrm{pat}} K\ \overline{t}\ P_1 \ldots P_m : T\ \overline{t}}$$

Figure 5.2: System F typing rules

We assume that there are some predefined data types in System F as follows,

```
data Maybe a = Just a | Nothing

data Or a b = L a | R b

data List a = Cons a (List a) | Nil

data Pair a b = Pair a b

data Unit = Unit
```

Sometimes we use some shorthands for these data types and their constructors. For instance, for types we use $[a]$ for *List* a, () for *Unit* and (a, b) for *Pair* a b; and for constructors we use $(x : xs)$ for (*Cons* x xs), [] for *Nil* and (x, y) for *Pair* x y. Note that for convenience, we omit type application for the constructors when there is no confusion arising.

We assume that there exists a built-in string type *String* and a special built-in

$$\boxed{[\![t]\!]}$$

$$[\![a]\!] = a \qquad [\![t_1 \to t_2]\!] = [\![t_1]\!] \to [\![t_2]\!]$$

$$[\![T\ \overline{t}]\!] = T\ \overline{[\![t]\!]} \qquad [\![\forall a.t]\!] = \forall a.[\![t]\!]$$

$$[\![t^*]\!] = [\ [\![t]\!]\ ] \qquad [\![(t_1|t_2)]\!] = Or\ [\![t_1]\!]\ [\![t_2]\!]$$

$$[\![\langle\rangle]\!] = () \qquad [\![\langle t_1, t_2\rangle]\!] = ([\![t_1]\!], [\![t_2]\!])$$

Figure 5.3: Translating source to target types

function

$$error : \forall a.String \to a$$

which signals a run-time error.

5.2 Constructive Interpretation of Subtyping

The coercion functions that we build operate on target expressions. Hence, we need to find appropriate target representations for source types. The natural choice is to represent Kleene star by lists, sequences by pairs and choice by the data type *Or*. All other source types can be literally adopted. The translation from source to target types is specified via function $[\![\cdot]\!]$. See Figure 5.3 for the details.

Example 24 For example,

$$\begin{array}{lcl} [\![A^*]\!] & = & [A] \\ [\![(A|\langle B,C\rangle)]\!] & = & (Or\ A\ (B,C)) \\ [\![A?]\!] & = & (Or\ A\ ()) \end{array}$$

□

To derive coercions out of subtype proofs $\vdash_{\text{sub}}\ t_1 \leq t_2$ we apply the proofs-are-programs principles. We write $\vdash_{\text{sub}}\ t_1 \leq^u_d t_2$ to denote that out of the subtype proof for $\vdash_{\text{sub}}\ t_1 \leq t_2$ we derive an up-cast coercion $u : \forall [\![t_1]\!] \rightarrow [\![t_2]\!]$ and a down-cast coercion $d : \forall [\![t_2]\!] \rightarrow \mathit{Maybe}\ [\![t_1]\!]$. An up-cast coercion injects a target expression of type $[\![t_1]\!]$ into the "larger" target type $[\![t_2]\!]$. This is the behavior we expect from coercive subtyping. The down-cast coercion d represents the pattern match of matching a value of type $[\![t_2]\!]$ against a pattern of type $[\![t_1]\!]$. We often call it coercive pattern matching. The pattern type is "smaller" than the incoming type. Hence, pattern matching may fail. We signal pattern matching failure by using the *Maybe* data type.

Example 25 For example, the subtype proof $\vdash_{\text{sub}}\ A \leq A^*$ should give rise to the following coercions:

$$\begin{array}{ll} & d : [A] \rightarrow \mathit{Maybe}\ A \\ u : A \rightarrow [A] & \\ & d\ [x] = \mathit{Just}\ x \\ u\ x = [x] & \\ & d\ \_ = \mathit{Nothing} \end{array}$$

For convenience, we use Haskell syntax for pattern matching (which can be obviously represented in System F with data types). For example, we write

```
f : ∀a.t → t'
f p₁ = e₁
...
f pₙ = eₙ
```

as a short hand of

```
f : ∀a.t → t'
f = Λa.λ(v : t). case v of
                 p₁ → e₁
                 ...
                 pₙ → eₙ
```

Subtype proofs with coercions

$$
\begin{array}{c}
C \cup \{t \leq^{u}_{d} t'\} \vdash_{\mathbf{sub}} d(l_1 \mid t) \leq^{u_1}_{d_1} d(l_1 \mid t') \\
\ldots \\
C \cup \{t \leq^{u}_{d} t'\} \vdash_{\mathbf{sub}} d(l_n \mid t) \leq^{u_n}_{d_n} d(l_n \mid t') \\
\Sigma(t') = \{l_1, ..., l_n\} \\
\begin{array}{lcl}
dd_1 : \forall[\![l_1]\!] \rightarrow [\![d(l_1 \mid t')]\!] \rightarrow \mathit{Maybe}\ [\![t]\!] & & dd_n : \forall[\![l_n]\!] \rightarrow [\![d(l_n \mid t')]\!] \rightarrow \mathit{Maybe}\ [\![t]\!] \\
dd_1\ l_1\ v'_1 = \mathit{case}\ (d_1\ v'_1)\ \mathit{of} & & dd_n\ l_n\ v'_n = \mathit{case}\ (d_n\ v'_n)\ \mathit{of} \\
\quad \mathit{Just}\ x \rightarrow \mathit{Just}\ \mathit{inj}_{(l_1,t)}\ l_1\ x & & \quad \mathit{Just}\ x \rightarrow \mathit{Just}\ \mathit{inj}_{(l_n,t)}\ l_n\ x \\
\quad \mathit{Nothing} \rightarrow \mathit{Nothing} & \ldots & \quad \mathit{Nothing} \rightarrow \mathit{Nothing} \\
 & & \\
uu_1 : \forall[\![l_1]\!] \rightarrow [\![d(l_1 \mid t)]\!] \rightarrow [\![t']\!] & & uu_n : \forall[\![l_n]\!] \rightarrow [\![d(l_n \mid t)]\!] \rightarrow [\![t']\!] \\
uu_1\ l_1\ v'_1 = \mathit{inj}_{(l_1,t')}\ l_1\ (u_1\ v'_1) & & uu_n\ l_n\ v'_n = \mathit{inj}_{(l_n,t')}\ l_n\ (u_n\ v'_n) \\
d : \forall[\![t']\!] \rightarrow \mathit{Maybe}\ [\![t]\!] & & u : \forall[\![t]\!] \rightarrow [\![t']\!] \\
d\ v = \mathit{if}\ \mathit{isEmpty}_{t'}\ v\ \mathit{then}\ \mathit{Just}\ \mathit{mkEmpty}_{t} & & u\ v = \mathit{if}\ \mathit{isEmpty}_{t}\ v\ \mathit{then}\ \mathit{mkEmpty}_{t'} \\
\qquad\qquad \mathit{else}\ \mathit{select}_{(l_1,...,l_n,t')}\ v\ dd_1 ... dd_n & & \qquad\qquad \mathit{else}\ \mathit{select}_{(l_1,...,l_n,t)}\ v\ uu_1 ... uu_n
\end{array} \\
\hline
C \vdash_{\mathbf{sub}} t \leq^{u}_{d} t'
\end{array}
$$

Figure 5.4a: Deriving coercions from subtype proofs

Let us go back to the definitions of u and d. u denotes an upcast function that injects a value of type A into A^*. d denotes a downcast function that fits a value of type A^* into A. As we discussed earlier, the coercions operate on target expressions. According to the type translation rules in Figure 5.3, we have $[\![A]\!] = A$ and $[\![A^*]\!] = [A]$. Thus, u has type $A \rightarrow [A]$. and d has type $[A] \rightarrow \mathit{Maybe}\ A$. $\Box$

The last example gives a rough idea of how the coercions u and d look like. However, the definitions provided in that example are almost "hand-coded". Next we show how to derive coercion from the subtype proof algorithm to cover the general cases.

Helper functions

$$\mathit{proj}_{(l,t'')} : \forall[\![t'']\!] \to \mathit{Maybe}\ ([\![l]\!], [\![d(l \shortmid t'')]\!])$$

$$\mathit{inj}_{(l,t'')} : \forall[\![l]\!] \to [\![d(l \shortmid t'')]\!] \to [\![t'']\!]$$

$$\mathit{isEmpty}_{t''} : \forall[\![t'']\!] \to \mathit{Bool}$$

$$\mathit{mkEmpty}_{t''} : \forall[\![t'']\!]$$

```
$select_{(l_1,...,l_n,t'')} : \forall[\![t'']\!] \to ([\![l_1]\!] \to [\![d(l_1 \shortmid t'')]\!] \to a)$
  $\to ... \to ([\![l_n]\!] \to [\![d(l_n \shortmid t'')]\!] \to a) \to a$
$select_{(l_1,...,l_n,t'')}\ v\ e_1 ... e_n =$
          $let\ v_1 = proj_{(l_1,t'')}\ v$
                $...$
              $v_n = proj_{(l_n,t'')}\ v$
          $in\ case\ (v_1, ..., v_n)\ of$
              $(Just\ (l_1, v_1'), ....) \to e_1\ l_1\ v_1'$
              $...$
              $(...., Just\ (l_n, v_n')) \to e_n\ l_n\ v_n'$
```

Figure 5.4b: Helper Functions

As discussed earlier, the general (simplified) shape of subtype proofs is as follows.

$$\frac{\begin{array}{c} C \cup \{t \leq t'\} \vdash_{\text{sub}} d(l_1 \shortmid t) \leq d(l_1 \shortmid t') \\ ... \\ C \cup \{t \leq t'\} \vdash_{\text{sub}} d(l_n \shortmid t) \leq d(l_n \shortmid t') \\ \Sigma(t') = \{l_1, ..., l_n\} \end{array}}{C \vdash_{\text{sub}} t \leq t'}$$

What remains is to simply attach proof terms (coercions) to subtype proofs. The details are in Figures 5.4a and 5.4b.

In Figure 5.4a, the proofs for sub-statements $C \cup \{t \leq_d^u t'\} \vdash_{\text{sub}} d(l_i \shortmid t) \leq_{d_i}^{u_i} d(l_i \shortmid t')$ give rise to up-cast coercions u_i and down-cast coercions d_i from which we then need to build the up-cast coercion u and down-cast coercion d for the statement

Semantic equivalence among source and target values

$$\langle\rangle \overset{\langle\rangle}{\approx} () \qquad \frac{w \sim \langle w_1', w_2'\rangle \quad w_1' \overset{t_1}{\approx} v_1 \quad w_2' \overset{t_2}{\approx} v_2}{w \overset{\langle t_1,t_2\rangle}{\approx} (v_1, v_2)} \qquad \frac{w \overset{t_1}{\approx} v_1}{w \overset{(t_1|t_2)}{\approx} L\ v_1} \qquad \frac{w \overset{t_2}{\approx} v_2}{w \overset{(t_1|t_2)}{\approx} R\ v_2}$$

$$\langle\rangle \overset{t^*}{\approx} [] \qquad \frac{w \sim \langle w_1', w_2'\rangle \quad w_1' \overset{t}{\approx} v_1 \quad w_2' \overset{t^*}{\approx} v_2}{w \overset{t^*}{\approx} (v_1 : v_2)} \qquad \frac{\begin{array}{c} K : \forall \bar{a}.t_1' \to ... \to t_n' \to T\ \bar{a} \in \Gamma_{init} \\ w_i \overset{\{\overline{t/a}\}t_i'}{\approx} w_i' \quad \text{for } i = 1, ..., n \end{array}}{K\ \bar{t}\ w_1...w_n \overset{T\ \bar{t}}{\approx} K\ [\![\bar{t}]\!]\ v_1'...v_n'}$$

$$\frac{\begin{array}{c} \text{for any values } w \textit{ and } v' \text{ such that } w \overset{t_1}{\approx} v' \\ \{w/x\}e \overset{t_2}{\approx} \{v'/x\}E \end{array}}{\lambda x : t_1.e \overset{t_1 \to t_2}{\approx} \lambda x : [\![t_1]\!].E} \qquad \frac{\begin{array}{c} \text{for any closed type } t' \\ \{t'/x\}e_1 \overset{t}{\approx} \{[\![t']\!]/a\}E_2 \end{array}}{\Lambda a.e_1 \overset{\forall a.t}{\approx} \Lambda a.E_2}$$

Semantic equivalence among source and target expressions

$$\begin{array}{lll} e_1 \overset{t}{\approx} E_2 & \text{iff} & \text{for any } v_1 \text{ such that } e_1 \longrightarrow^* w_1 \\ & & \text{there exists } v_2 \text{ such that } E_2 \longrightarrow^* v_2 \text{ and } w_1 \overset{t}{\approx} v_2 \end{array}$$

Semantic equivalence among target expressions

$$\begin{array}{lll} E_1 \overset{t}{\leftrightarrow} E_2 & \text{iff} & e_3 \overset{t}{\approx} E_1 \text{ and } e_3 \overset{t}{\approx} E_2 \\ & & \text{for some System F}^* \text{ expression } e_3 \end{array}$$

Figure 5.5: Semantic Equivalence Relations

$C \vdash_{\text{sub}} t \leq_d^u t'$. Due to the co-inductive nature of our subtype proof system, the proofs for the sub-statements might already make use of u and d. The statement $t \leq_d^u t'$ is added to the assumption set. Therefore, coercions can be recursive.

For the construction of u and d we introduce some helper functions, see Figure 5.4b. The helper functions are indexed by types. They represent a family of helper functions. The helper functions must satisfy the following properties.

Definition 3 (Helper Function Properties) *Let t be a System* F* *type and l a System* F* *label type.*

Is empty *If v is a System F expression of type $[\![t]\!]$ such that $\langle\rangle \overset{t}{\approx} v$ then $isEmpty_t \longrightarrow^*$ True. Otherwise, $isEmpty_t \longrightarrow^*$ False.*

Make empty *We have that* $\langle\rangle \overset{t}{\approx} \mathit{mkEmpty}_t$.

Projection *If* v_1 *is a System F value of type* $[\![t]\!]$ *and* $\langle l, w\rangle \overset{t}{\approx} v_1$ *for some System* F* *value* w *of type* $d(l \shortmid t)$ *then*

$$\mathit{proj}_{(l,t)}\ v_1 \longrightarrow^* \mathit{Just}\ (v_2, v_3)$$

for some System F values v_2 *and* v_3 *such that* $w \overset{d(l \shortmid t)}{\approx} v_3$ *and* $l \overset{l}{\approx} v_2$. *In all other cases,* $\mathit{proj}_{(l,t)}\ v_1 \longrightarrow^* \mathit{Nothing}$.

Injection *If* w *is a System* F* *value of type* $d(l \shortmid t')$, v_1 *and* v_2 *are System F values such that* v_1 *of type* $[\![l]\!]$ *and* v_2 *of type* $[\![d(l \shortmid t')]\!]$. *Then,* $\mathit{inj}_{(l,t)}\ v_1\ v_2 \longrightarrow^* v_3$ *for some System F value* v_3 *such that* $\langle l, w\rangle \overset{t}{\approx} v_3$.

Function $\mathit{mkEmpty}_{t''}$ embeds the empty sequence into a target type $[\![t'']\!]$ whereas $\mathit{isEmpty}_{t''}$ checks whether a target value is empty. We make use of a semantic equivalence relation $w \overset{t}{\approx} v$ to compare a source System F* value w of type t against a target System F value v of type $[\![t]\!]$. The details of the semantic equivalence relation (in essence a logical relation) are given in Figure 5.5. Function $\mathit{proj}_{(l,t'')}$ (possibly) projects a value of type $[\![t'']\!]$ onto the type $[\![d(l \shortmid t'')]\!]$. Recall that types t'' are normalized to the form $\langle l_1, d(l_1 \shortmid t'')\rangle|...|\langle l_n, d(l_n \shortmid t'')\rangle$ where $l_i \in \Sigma(t'')$. The choice type $|$ is translated to Or. Hence, a value of type $[\![t'']\!]$ contains only one of the monomials $\langle l_i, d(l_i \shortmid t'')\rangle$. Hence, function $\mathit{proj}_{(l,t'')}$ simply checks which particular monomial is present and extracts this monomial out of $[\![t'']\!]$ and fails otherwise. Function $\mathit{inj}_{(l,t'')}$ performs the opposite operation. We apply it to the label value l and the remaining value of type of $[\![d(l \shortmid t'')]\!]$, which is then injected into $[\![t'']\!]$.

Next, we show how to build coercions u and d via these helper functions. In case of the up-cast coercion we first check if the incoming value is empty. If it is, we embed the (target representation of the) empty sequence into the type $[\![t']\!]$. We assume here that both types t and t' contain the empty sequence. If the incoming value is not empty, we extract the monomial v out via the helper projection functions. The target representation of this monomial is $[\![d(l_i \mid t)]\!]$. For a valid subtype proof there is exactly one of the monomials present in v. We then perform the up-casting using one of the coercions u_i which gives us a monomial in target representation $[\![d(l_i \mid t')]\!]$. What remains is to inject this monomial into $[\![t']\!]$ via one of the helper injection functions. We carry out these steps via the helper function $\mathit{select}_{(l_1,\ldots,l_n,t)}$ (here $t'' = t$) which performs the extraction and takes as arguments the specific up-cast followed by injection functions uu_i.

Building of the down-cast coercion d works similarly. We first deal with the empty sequence case. Remember that we assume that both types t and t' contain the empty sequence. Then, we first extract the monomial on which we then apply the appropriate down-cast d_i. The remaining step is to apply the injection function unless the down-cast d_i failed (i.e. resulted in *Nothing*). We make use again of the helper function $\mathit{select}_{(l_1,\ldots,l_n,t')}$ (here $t'' = t'$) and the specific down-cast followed by injection functions dd_i.

To have a better idea of the process of deriving of the coercion functions from the subtype proof, let us consider an example.

Example 26 Recall from the previous example that we hand-coded the "artificial" definitions of the upcast function u and the downcast function d, which can be derived from the subtype proof $\vdash_{\text{sub}} A \leq^u_d A^*$. Now let us make use of the definitions in Figure 5.4a and 5.4b to derive the "real" definitions of these coercion functions.

$$\frac{\begin{array}{l} \{A \leq^u_d A^*\} \vdash_{\text{sub}} d(A \mid A) \leq^{u_1}_{d_1} d(A \mid A^*) \\ \\ dd_1 : \forall A \rightarrow [\![d(A \mid A^*)]\!] \rightarrow \mathit{Maybe}\ [\![A]\!] \\ dd_1\ l_1\ v'_1 = \mathit{case}\ (d_1\ v'_1)\ \mathit{of} \\ \quad \mathit{Just}\ x \rightarrow \mathit{Just}\ \mathit{inj}_{(A,A)}\ l_1\ x \\ \quad \mathit{Nothing} \rightarrow \mathit{Nothing} \\ d : [\![A^*]\!] \rightarrow \mathit{Maybe}\ [\![A]\!] \\ d\ v = \mathit{if}\ \mathit{isEmpty}_{A^*}\ v\ \mathit{then}\ \mathit{Nothing} \\ \qquad\qquad \mathit{else}\ \mathit{select}_{(A,A^*)}\ v\ dd_1 \end{array}}{\vdash_{\text{sub}} A \leq^u_d A^*}$$

In this example, we only consider the downcast coercion, the upcast coercion is derivable similarly. We read the above derivation from bottom to top. d extracts a single A from a sequence of As. In the body of d we use the helper function $\mathit{isEmpty}_{A^*}$ to test whether the incoming value is empty. This is necessary because A^* can potentially be empty. If the incoming value is empty, the application of d fails (because the empty word does not inhabits in A). Otherwise, we apply the selection function $\mathit{select}_{(A,A^*)}$ to convert the incoming value into the monomial form. The definition of the selection function is as follows,

$$\begin{array}{l} \mathit{select}_{(A,A^*)} : \forall [\![A^*]\!] \rightarrow ([\![A]\!] \rightarrow [\![d(A \mid A^*)]\!] \rightarrow a) \rightarrow a \\ \mathit{select}_{(A,A^*)}\ v\ e_1 = \\ \qquad\qquad \mathit{let}\ v_1 = \mathit{proj}_{(A,A^*)}\ v \\ \qquad\qquad \mathit{in}\ \mathit{case}\ v_1\ \mathit{of} \\ \qquad\qquad\quad (\mathit{Just}\ (l_1, v'_1)) \rightarrow e_1\ l_1\ v'_1 \end{array}$$

In the selection function, we make use of the helper function $proj_{(A,A^*)}$ to extract the first A out of the incoming value. Some reader may wonder what if the extraction fails. Note that the property of function $isEmpty_{A^*}$ given in Definition 3 guarantees that if the incoming value is empty, the empty test definitely yields $True$. That means we will not apply the selection function if the incoming value is empty. Hence, we can be sure that the value being sent to the $inj_{(A,A^*)}$ function must be non-empty. There must be at least an A in the incoming value. Based on the property of the $inj_{(A,A^*)}$ given in Definition 3, we can conclude that the extraction must be successful. Finally, we apply the helper function dd_1 to the extracted label A and the remaining value.

In the body of dd_1, we use the downcast coercion d_1. Note that d_1 is the proof-term of the sub-proof $\{A \leq_d A^*\} \vdash_{\text{sub}} d(A \shortmid A) \leq_{d_1} d(A \shortmid A^*)$, which can be simplified to $\{A \leq_d A^*\} \vdash_{\text{sub}} \langle\rangle \leq_{d_1} A^*$. Now it is clear that we use d_1 to test whether the remaining incoming value (after removing the leading A) can be fit into $\langle\rangle$. If this test is successful, i.e., $Just\ x$ is returned, we use the helper function $inj_{(A,A)}$ to inject A back to the value of x, which is the final result. If the test is unsuccessful, we return $Nothing$ to signal that the entire downcast coercion results in a failure. $\square$

Note that the helper functions are derivable from the auxilary judgment $\vdash_{\text{empty}} \langle\rangle \in t$ and operation $d(l \shortmid t) = t'$. It is straightforward but tedious to give System F definitions of the helper functions. We will provide the concrete definitions of these helper functions in Chapter 6. The important point to note is that there is a design space for the helper function definitions which depends on the particular pattern matching policy employed.

We first consider some definitions of $isEmpty_t$ and $proj_{(l,t)}$, which are fixed by the type.

Example 27 For instance, $isEmpty_{A^*}$ can be defined as

```
isEmpty_A* v = case v of { [] → True; _ → False }
```

$$mkEmpty_{(\langle A^*,A^*\rangle | A^*)} = L\ ([],[]) \qquad (1)$$

$$mkEmpty_{(\langle A^*,A^*\rangle | A^*)} = R\ [] \qquad (2)$$

$$\begin{array}{l}
inj_{(A,\langle A^*,A^*\rangle)} : A \rightarrow Or\ ([A],[A])\ [A] \rightarrow ([A],[A]) \qquad (3) \\
inj_{(A,\langle A^*,A^*\rangle)}\ v\ (L\ (xs,ys)) = (v : (xs{+}{+}ys), []) \\
inj_{(A,\langle A^*,A^*\rangle)}\ v\ (R\ zs) = (v : zs, [])
\end{array}$$

$$\begin{array}{l}
inj_{(A,\langle A^*,A^*\rangle)} : A \rightarrow Or\ ([A],[A])\ [A] \rightarrow ([A],[A]) \qquad (4) \\
inj_{(A,\langle A^*,A^*\rangle)}\ v\ (L\ (xs,ys)) = (v : xs, ys) \\
inj_{(A,\langle A^*,A^*\rangle)}\ v\ (R\ zs) = ([v], zs)
\end{array}$$

Figure 5.6: The possible ways of defining $mkEmpty_{(\langle A^*,A^*\rangle | A^*)}$ and $inj_{(A,\langle A^*,A^*\rangle)}$

□

Example 28 Recall that $d(A \shortmid A^*) = A^*$. Function $proj_{(A,A^*)}$ can be defined as

$proj_{(A,A^*)}$ `v = case v of { (x,xs) → Just (x,xs); [] → Nothing }`

□

On the other hand, we have choices in defining of *mkEmpty* and *inj* functions.

Example 29 For example, we give two valid definitions of $mkEmpty_{(\langle A^*,A^*\rangle | A^*)}$, (1) and (2) in Figure 5.6. In the same figure, we also find that there are two possible ways of defining $inj_{(A,\langle A^*,A^*\rangle)}$, namely (3) and (4), which is based on the partial derivative result, $d(A \shortmid \langle A^*, A^*\rangle) = \langle A^*, A^*\rangle \mid A^*$. □

Note that by changing the combination of different implementation of $mkEmpty_{(\langle A^*,A^*\rangle | A^*)}$ and $inj_{(A,\langle A^*,A^*\rangle)}$, the resulting coercion function implements different matching policy. We will experience such a result in the upcoming example, where we consider building a downcast coercion using these helper functions.

Example 30 For example, we consider building the downcast coercion function d out of the proof derivation of $\vdash_{\text{sub}} \langle A^*, A^*\rangle \leq A^*$ as follows,

$$\dfrac{\dfrac{\begin{array}{c} \ldots \\ d_1 : [\![A^*]\!] \to \mathit{Maybe}[\![\langle A^*, A^*\rangle | A^*]\!] \\ d_1\ v = \mathit{isEmpty}_{A^*}\ v\ \mathit{then}\ \mathit{Just}\ \mathit{mkEmpty}_{\langle A^*,A^*\rangle|A^*}\ \mathit{else}\ \ldots \end{array}}{\{\langle A^*, A^*\rangle \leq_d A^*\} \vdash_{\text{sub}} d(A \shortmid \langle A^*.A^*\rangle) \leq_{d_1} d(A \shortmid A^*)} \quad \begin{array}{l} dd_1 : [\![A]\!] \to [\![d(A \shortmid A^*)]\!] \to \mathit{Maybe}\ [\![\langle A^*, A^*\rangle]\!] \\ dd_1\ l_1\ v_1' = \mathit{case}\ (d_1\ v_1')\ \mathit{of} \\ \quad \mathit{Just}\ x \to \mathit{Just}\ \mathit{inj}_{(A,\langle A^*,A^*\rangle)}\ l_1\ x \\ \quad \mathit{Nothing} \to \mathit{Nothing} \\ d : [\![A^*]\!] \to \mathit{Maybe}\ [\![\langle A^*, A^*\rangle]\!] \\ d\ v = \mathit{if}\ \ \mathit{isEmpty}_{A^*}\ v\ \mathit{then}\ \mathit{Just}\ \mathit{mkEmpty}_{\langle A^*,A^*\rangle} \\ \qquad \mathit{else}\ \mathit{case}\ \mathit{proj}_{(A,A^*)}\ v\ \mathit{of} \\ \qquad\qquad \mathit{Just}\ (l, v') \to dd_1\ l\ v' \\ \qquad\qquad \mathit{Nothing} \to \mathit{Nothing} \end{array}}{\vdash_{\text{sub}} \langle A^*, A^*\rangle \leq_d A^*}$$

Note that for simplicity, we inline the selection function. Suppose we apply d to value $[A]$, the evaluation of $(d\ [A])$ proceeds as follows,

$$\begin{array}{lcll} d\ [A] & \longrightarrow & dd_1\ A\ [] & \text{because } \mathit{proj}_{(A,A^*)}\ [A] \longrightarrow^* \mathit{Just}\ (A, []) \\ & \longrightarrow & \mathit{Just}\ \mathit{inj}_{(A,\langle A^*,A^*\rangle)}\ A\ x & \text{where } d_1\ [] \longrightarrow^* \mathit{Just}\ x \end{array}$$

Note that d_1 is defined in in terms of $\mathit{isEmpty}_{A^*}$ and $\mathit{mkEmpty}_{(\langle A^*,A^*\rangle|A^*)}$. Suppose we choose definition (1) for $\mathit{mkEmpty}_{(\langle A^*,A^*\rangle|A^*)}$ defined in Figure 5.6, we have that $x = L\ ([], [])$. Suppose we choose definition (3) for $\mathit{inj}_{(A,\langle A^*,A^*\rangle)}$ defined in Figure 5.6, we have

$$\mathit{inj}_{(A,\langle A^*,A^*\rangle)}\ A\ x \longrightarrow^* \mathit{Just}\ ([A], [])$$

In other words, the above coercion function implements the POSIX/Longest matching policy which is discussed earlier. Suppose we switch to a different combination of definitions such as (2) and (3) in Figure 5.6, the above evaluates to $Just\ ([], [A])$ which means we apply the shortest matching policy. If we use the combination of (1) and (4), we achieve a "random" matching policy. $\Box$

Our implementation of the helper functions adheres to the POSIX (or longest-match) policy. We postone a discussion of the matching policy to Chapter 6.

However, the constructive interpretation of subtyping we are using does not support downcast of higher order function.

Example 31 Consider the subtype proof

$$\frac{\vdash_{\text{sub}} A \leq^{u_1}_{d_1} (A|B) \quad \vdash_{\text{sub}} A \leq^{u_2}_{d_2} (A|C)}{\vdash_{\text{sub}} (A|B) \rightarrow A \leq^{u}_{d} A \rightarrow (A|C)}$$

From the above, we can easily define the upcast function u in terms of u_1 and u_2 as follows,

```
u f = \ x -> u2 (f (u1 x))
```

However we cannot define the downcast function d, even if we have d_1 and d_2.

```
d f = ???
```

The reason is that in downcast function, we need to test whether the input value can be fitted into the output type by examing its structure. However, d_1 and d_2 can only be used to test the structure of `f`'s input and output, but not `f` itself. Furthermore, `f` is a System F function, which has no structure at all. $\Box$

Thus, our system disallows subtyping on function type.

We conclude that by applying the proofs-are-programs principle we derive upcast as well as down-cast coercions out of subtype proofs. We obtain the following results.

Lemma 4 (Coercive Subtyping and Pattern Matching) *Let* $\vdash_{sub}\ t_1 \leq^u_d t_2$*. Then, u and d are well-typed in System F with data types and their types are* $u : \forall[\![t_1]\!] \rightarrow [\![t_2]\!]$ *and* $d : \forall[\![t_2]\!] \rightarrow$ *Maybe* $[\![t_1]\!]$.

The proof of this lemma is straightforward based on the description of proof term construction in Figures 5.4a and 5.4b.

The above lemma states that subtyping and pattern matching in System F* can be reduced to coercive subtyping and pattern matching which is definable in System F. We make use of the above shortly to translate System F* expressions to System F.

In addition, we can also guarantee that up-/ and (successful) down-casting will not lose any data. This property is formalized in the next lemmas.

Lemma 5 (Semantic Preservation (Upcast)) *Let* $\vdash_{sub}\ t_1 \leq^u t_2$*,* v_1 *be a target value of type* $[\![t_1]\!]$*,* w *be a source value such that* $w \overset{t_1}{\approx} v_1$*. Then* $u\ v_1 \longrightarrow^* v_2$ *implies that* $w \overset{t_2}{\approx} v_2$*.*

Lemma 6 (Semantic Preservation (Downcast)) *Let* $\vdash_{sub}\ t_1 \leq_d t_2$*,* v_2 *be a target value of type* $[\![t_2]\!]$*,* w *be a source value such that* $w \overset{t_2}{\approx} v_2$*. Then* $d\ v_2 \longrightarrow^*$ *Just* v_1 *implies that* $w \overset{t_1}{\approx} v_1$*.*

Now we can state that the combination of upcast and downcast does not break the semantic preservation.

Lemma 7 (Semantic Preservation) *Let* $\vdash_{sub}\ t_1 \leq^u_d t_2$*,* v_1 *be a value of type* $[\![t_1]\!]$ *and* v_2 *be a value of type* $[\![t_2]\!]$*. Then,*

(1) $d\ (u\ v_1) \longrightarrow^*$ *Just* v_3 *such that* $v_1 \overset{t_1}{\leftrightarrow} v_3$*, and*

(2) if $d\ v_2 \longrightarrow^*$ *Just* v_4 *then* $v_2 \overset{t_2}{\leftrightarrow} u\ v_4$*.*

The first property states that up-casting followed by down-casting yields back the original value but possibly in a different structural target representation. We express this via the relation $e_1 \overset{t}{\leftrightarrow} e_2$ (defined in Figure 5.5) which states that two target expressions e_1 and e_2 of type $[\![t]\!]$ are equal if both are equivalent to a common source expression of type t. Similarly, the second property states that a successful down-cast followed by an up-cast is effectively the identity operation. Both properties follow from the conditions imposed on our helper functions (see Definition 3).

The technical proof details of the last two lemma as well as any other subsequent results stated in this chapter can be found in Appendix B Section B.2.

In the upcoming Chapter 6, we will give a comprenhesive account of how to derive the up/down-cast coercions.

5.3 System F* to System F Translation Scheme

5.3.1 Translating Expressions via Coercive Subtyping

Translating System F* source expressions to System F target expressions is straightforward by simply inserting coercions derived out of subtype proofs. Formally, we introduce judgments $\Gamma \vdash e_1 : t \rightsquigarrow e_2$ which derive a System F expression e_2 from a System F* expression e_1, given the System F*'s expression typing derivations. The translation rules (Var) - (Sub) for expressions are in Figure 5.7 and should not contain any surprises. We maintain the invariant that expression e_2 is of type $[\![t]\!]$ under the environment $[\![\Gamma]\!]$, where $[\![\Gamma]\!]$ is the translation of source type environment Γ. It is defined as

$$[\![\Gamma]\!] = \{x : [\![t]\!] | (x : t) \in \Gamma\}$$

5.3.2 Translating Patterns via Coercive Pattern Matching

To translate case expressions we make use of down-cast coercions derived out of subtype proofs. See rule (Case). Via the auxiliary judgment $\Gamma \vdash_{\text{pat}} p : t \rightsquigarrow P$ each System F* pattern is translated to a corresponding System F pattern. We

$$\boxed{\Gamma \vdash e : t \rightsquigarrow E}$$

$$\text{(Var)}\ \frac{x : t \in \Gamma}{\Gamma \vdash x : t \rightsquigarrow x} \qquad \text{(EAbs)}\ \frac{\Gamma \cup \{x : t_1\} \vdash e : t_2 \rightsquigarrow E}{\Gamma \vdash \lambda x : t_1.e : t_1 \to t_2 \rightsquigarrow \lambda x : [\![t_1]\!].E}$$

$$\text{(EApp)}\ \frac{\Gamma \vdash e_1 : t_2 \to t_1 \rightsquigarrow E_1 \quad \Gamma \vdash e_2 : t_2 \rightsquigarrow E_2}{\Gamma \vdash e_1\ e_2 : t_1 \rightsquigarrow E_1\ E_2} \qquad \text{(TAbs)}\ \frac{\Gamma \vdash e : t \rightsquigarrow E \quad a \notin fv(\Gamma)}{\Gamma \vdash \Lambda a.e : \forall a.t \rightsquigarrow \Lambda a.E}$$

$$\text{(TApp)}\ \frac{\Gamma \vdash e : \forall a.t_1 \rightsquigarrow E}{\Gamma \vdash e\ t_2 : [t_2/a]t_1 \rightsquigarrow E\ [\![t_2]\!]} \qquad \text{(Let)}\ \frac{\Gamma \vdash e_1 : t_1 \rightsquigarrow E_1 \quad \Gamma \cup \{x : t_1\} \vdash e_2 : t_2 \rightsquigarrow E_2}{\Gamma \vdash \mathsf{let}\ x : t_1 = e_1\ \mathsf{in}\ e_2 : t_2 \rightsquigarrow \mathsf{let}\ x : [\![t_1]\!] = E_1\ \mathsf{in}\ E_2}$$

$$\text{(EmptySeq)}\ \Gamma \vdash \langle\rangle : \langle\rangle \rightsquigarrow ()$$

$$\text{(PairSeq)}\ \frac{\Gamma \vdash e_1 : t_1 \rightsquigarrow E_1 \quad \Gamma \vdash e_2 : t_2 \rightsquigarrow E_2}{\Gamma \vdash \langle e_1, e_2\rangle : \langle t_1, t_2\rangle \rightsquigarrow (E_1, E_2)} \qquad \text{(Sub)}\ \frac{\Gamma \vdash e : t_1 \rightsquigarrow E \quad \vdash_{\text{sub}} t_1 \leq^u t_2}{\Gamma \vdash e : t_2 \rightsquigarrow u\ E}$$

$$\text{(Case)}\ \frac{\begin{array}{c}\Gamma \vdash e : t \rightsquigarrow E \quad \Gamma_i \vdash_{\text{pat}} p_i : t_i \rightsquigarrow P_i \quad \vdash_{\text{sub}} t_i \leq_{d_i} t \quad \Gamma \cup \Gamma_i \vdash e_i : t' \rightsquigarrow E_i \\ g_i = \lambda c.\mathsf{case}\ d_i\ E\ \mathsf{of}\ \{Just\ P_i \to E_i; Nothing \to c\} \quad \text{for } i \in I\end{array}}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ [p_i \to e_i]_{i \in I} : t' \rightsquigarrow g_1\ (...\ (g_n\ (error\ \text{``}pattern\ is\ not\ exhaustive\text{''})))}$$

Figure 5.7: Translation from System F* to System F

refer to Figure 5.8 for details. We again maintain the invariant that pattern P is of type $[\![t]\!]$ under environment $[\![\Gamma]\!]$. The pattern rules for variables and sequences contain no surprises. In the pattern rule for constructors the source constructor K belonging to data type T is translated to a target constructor K_T, which is assumed to exist in the target initial type environment Γ^{target}_{init}. Thus, we encode Haskell/ML style pattern matching via down-casting, see rule (K), which leads to a uniform translation scheme. We give an example below.

For each pattern clause, we translate the body e_i using the translation rules for expressions. We translate each pattern p_i by deriving a down-cast coercion $\vdash_{\text{sub}} t_i \leq_{d_i} t$. Via the down-cast coercion we then check which pattern clause applies. We check pattern clauses from top to bottom. In case of a successful pattern match, the result is bound to the target pattern P_i.

Here is a simple example which shows the translation rules in action.

$$\boxed{\Gamma \vdash_{\text{pat}} p : t \leadsto P}$$

$$\emptyset \vdash_{\text{pat}} \langle\rangle : \langle\rangle \leadsto () \quad \{x : t\} \vdash_{\text{pat}} (x : t) : t \leadsto x \quad \frac{\Gamma_1 \vdash_{\text{pat}} p_1 : t_1 \leadsto P_1 \quad \Gamma_2 \vdash_{\text{pat}} p_2 : t_2 \leadsto P_2}{\Gamma_1 \cup \Gamma_2 \vdash_{\text{pat}} \langle p_1, p_2\rangle : \langle t_1, t_2\rangle \leadsto (P_1, P_2)}$$

$$\frac{\begin{array}{c}\Gamma_{init} \vdash K : \forall \overline{a}.t'_1 \to ... \to t'_m \to T\ \overline{a} \\ \Gamma_i \vdash_{\text{pat}} p_i : t''_i \leadsto P_i \quad \vdash_{\text{sub}} t''_i \leq [\overline{t/a}]t'_i \quad \text{for } i = 1, ..., m\end{array}}{\Gamma_1 \cup ... \cup \Gamma_m \vdash_{\text{pat}} K\ \overline{t}\ p_1...p_m : K_T\ t''_1\ ...t''_m \leadsto K_T\ P_1...P_m}$$

Data type Pattern Matching

$$\text{(K)} \quad \frac{\begin{array}{c}\Gamma_{init} \vdash K : \forall \bar{a}.t_1 \to ... \to t_n \to T\ \bar{a} \\ \Gamma^{target}_{init} \vdash K_T : \forall \bar{b}.b_1 \to ... \to b_n \to K_T\ \bar{b} \\ \vdash_{\text{sub}} t'_i \leq_{d_i} [\overline{t/a}]t_i \quad \text{for } i = 1, ..., n \\ \begin{array}{ll} d\ (K\ x_1...x_n) = \mathsf{case}\ (d_1\ x_1, ..., d_n\ x_n)\ \mathsf{of} & \\ \qquad (Just\ v_1, ..., Just\ v_n) \to Just\ (K_T\ v_1...v_n) & \\ \qquad \_ \to Nothing & \\ d\ \_ = Nothing & \end{array}\end{array}}{\vdash_{\text{sub}} K_T\ t'_1...t'_n \leq_d T\ \bar{t}}$$

Figure 5.8: Translating Pattern Matching

Example 32 Recall the address book example we mentioned in Chapter 4. In this example, we are interested in translating the `foreach` function.

`data` $Person \;=\; Person\ \langle Name, Tel?, Email*\rangle$

`foreach :` $Person \to Entry?$

`foreach v = case v of`

$$(Person\ \langle n : Name, \langle t : Tel, es : Email^*\rangle\rangle) \to Entry\ \langle n, t\rangle$$

We apply the pattern rule (K) to the above pattern,

$$\frac{\dfrac{\begin{array}{c}\{n : \mathit{Name}\} \vdash_{\text{pat}} (n : \mathit{Name}) : \mathit{Name} \leadsto n \\ \{t : \mathit{Tel}\} \vdash_{\text{pat}} (t : \mathit{Tel}) : \mathit{Tel} \leadsto t \\ \{es : \mathit{Email}^*\} \vdash_{\text{pat}} (es : \mathit{Email}^*) : \mathit{Email}^* \leadsto es\end{array}}{\Gamma \vdash_{\text{pat}} \langle n : \mathit{Name}, \langle t : \mathit{Tel}, es : \mathit{Email}^*\rangle\rangle : \langle \mathit{Name}, \langle \mathit{Tel}, \mathit{Email}^*\rangle\rangle \leadsto (n, (t, es))} \quad \begin{array}{c}\Gamma_{\mathit{init}} \vdash \mathit{Person} : \langle \mathit{Name}, \langle \mathit{Tel}?, \mathit{Email}^*\rangle\rangle \rightarrow \mathit{Person} \\ \vdash_{\text{sub}} \langle \mathit{Name}, \langle \mathit{Tel}, \mathit{Email}^*\rangle\rangle \leq \langle \mathit{Name}, \langle \mathit{Tel}?, \mathit{Email}^*\rangle\rangle\end{array}}{\begin{array}{c}\Gamma \vdash_{\text{pat}} \mathit{Person}\ \langle n : \mathit{Name}, \langle t : \mathit{Tel}, es : \mathit{Email}^*\rangle\rangle : \mathit{Person}_T\ \langle \mathit{Name}, \langle \mathit{Tel}, \mathit{Email}^*\rangle\rangle \\ \leadsto \mathit{Person}_T\ (n, (t, es))\end{array}}$$

where $\Gamma = \{n : \mathit{Name}, t : \mathit{Tel}, es : \mathit{Email}^*\}$. We build the down-cast coercion

$$\vdash_{\text{sub}} \mathit{Person}_T\ \langle \mathit{Name}, \langle \mathit{Tel}, \mathit{Email}^*\rangle\rangle \leq_d \mathit{Person}$$

which according to the data type pattern translation rule (K) in Figure 5.8 yields

```
d (Person (n,t,es)) =
   case(d1 n, d2 t, d3 es) of
   (Just v1, Just v2, Just v3) -> PersonT (v1,(v2 ,v3))
   _                           -> Nothing


d1 x = Just x
d3 x = Just x
d2 (L x) = Just x
d2 _ = Nothing
```

The last coercion results from $\vdash \mathit{Tel} \leq_{d_2} \mathit{Tel}?$. Recall that *Tel*? is a short-hand for $\mathit{Tel}|\langle\rangle$. Thus, the translation of the above program text yields

```
data Person = Person (Name, (Or Tel (), [Email]))
data Person_T a = Person_T a
foreach :   Person → Or Entry ()
foreach x = case (d x) of
                Just (Person_T (n,(t,es))) = Entry (n,t)
```

□

5.4 Type Preservation

Based on Lemma 4 we can verify that the resulting System F expressions are well-typed.

Theorem 5 (Type Preservation) *Let* $\Gamma_{init} \vdash e : t \leadsto E$. *Then* $\Gamma^{target}_{init} \vdash_F E : [\![t]\!]$.

Obviously, we also would like to verify that the semantic meaning of programs has not changed in any essential way. Some form of semantic preservation should hold. For example, see Lemma 7 which states that in the target program up-casting followed by down-casting behaves like the identity.

Before we relate source against target expressions, we first need to guarantee that all possible target translations resulting from the same source expressions are related. This property is usually referred to as coherence and non-trivial because of the coercions derived out of subtype proofs and the non-syntax directed typing rule (Sub). In the next section, we identify conditions under which we achieve coherence.

5.5 The Coherence Problem

The coherence problem was first studied in [11]. In essence, subtyping permits a program to be type-checked in more than one way. One must prove that the meaning of the program does not depend on the way it is typed.

In our context, the coherence problem is caused by coercive subtyping. To illustrate this point, let us consider an example.

Example 33 Consider the translating the following System F* program,

```
data A = A
v:A*
v = A
```

According the translation rules described in Figure 5.7 in Chapter 5, there are (at least) two ways to translate the above program,

$$\frac{\vdash A : A \leadsto A \qquad \vdash_{\text{sub}} A \leq^{u_1} A^*}{\vdash A : A^* \leadsto u_1\ A}\text{(Sub)} \tag{5.1}$$

$$\frac{\dfrac{\vdash A : A \leadsto A}{\vdash A : A? \leadsto (u_2\ A)}\text{(Sub)} \qquad \vdash_{\text{sub}} A? \leq^{u_3} A^*}{\vdash A : A^* \leadsto u_3\ (u_2\ A)}\text{(Sub)} \tag{5.2}$$

□

As we can observe from the above, the translation rules are not syntax-directed thanks to the subsumption rule. Therefore, the translation result is not unique.

Things may get more complicated when we combine semantic subtyping with regular expression pattern matching,

Example 34 For example, when translating the expression $\mathsf{case}\ A\ \mathsf{of} x : A \to x$, we may have two different derivations, thanks to the non-syntax directed rule (Sub),

$$\frac{\Gamma \vdash A : A \leadsto A \qquad \vdash_{\text{sub}} A \leq_{d_1} A}{\Gamma \vdash \mathsf{case}\ A\ \mathsf{of}\ \{x : A \to x\} : A \leadsto \mathsf{case}\ d_1\ A\ \mathsf{of}\{\mathit{Just}\ x \to x\}} \tag{5.3}$$

$$\frac{\dfrac{\Gamma \vdash A : A \leadsto A \qquad \vdash_{\text{sub}} A \leq^{u} A^{*}}{\Gamma \vdash A : A^{*} \leadsto u\ A} \qquad \vdash_{\text{sub}} A \leq_{d_2} A^{*}}{\Gamma \vdash \mathsf{case}\ A\ \mathsf{of}\ \{x : A \to x\} : A \leadsto \mathsf{case}\ d_2\ (u\ A)\ \mathsf{of}\{\mathit{Just}\ x \to x\}} \tag{5.4}$$

□

As demonstrated, the translation of a case expression can be different if we apply additional subsumption rule in the conditional expression position.

On the other hand, since the coercion function preserves the semantics, the translation results should share the same semantic meaning, though they are syntactically different. In other words, our translation should be coherent. However, our coherence does not come for free. To establish coherence, we need to impose some conditions on the source programs.

5.6 Establishing Coherence

In a first step we establish conditions to guarantee coherence and transitivity of coercive subtyping (i.e. up-cast coercions). These are the classic conditions to guarantee coherence in the presence of subtyping. In addition, we also need to ensure that our use of coercive pattern matching (i.e. down-cast coercions) to translate regular expression pattern matching and their interaction with coercive subtyping will not break coherence. First, we take a look at coercive subtyping.

5.6.1 Coherence and Transitivity of Coercive Subtyping

We first observe that a subtype statement can give rise to incomparable coercions. For example, consider the statement $\vdash_{\text{sub}} (T\ A) \leq ((T\ A?)|(T\ A^*))$ where T is a data type with the single constructor $K : \forall a.a \to T\ a$. We can verify the statement using either of the following two subtype proofs.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dots}{\vdash_{\text{sub}}\ A \leq A?}}{\vdash_{\text{lab}}\ (T\ A) \leq (T\ A?)} \quad \vdash_{\text{lnf}}\ \langle\rangle \leq \langle\rangle}{\vdash_{\text{lnf}}\ \langle (T\ A), \langle\rangle\rangle \leq \langle (T\ A?), \langle\rangle\rangle}}{\vdash_{\text{lnf}}\ \langle (T\ A), \langle\rangle\rangle \leq \langle (T\ A?), \langle\rangle\rangle | \langle (T\ A^*), \langle\rangle\rangle}}{\vdash_{\text{sub}}\ (T\ A) \leq ((T\ A?)|(T\ A^*))} \tag{5.5}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dots}{\vdash_{\text{sub}}\ A \leq A^*}}{\vdash_{\text{lab}}\ (T\ A) \leq (T\ A^*)} \quad \vdash_{\text{lnf}}\ \langle\rangle \leq \langle\rangle}{\vdash_{\text{lnf}}\ \langle (T\ A), \langle\rangle\rangle \leq \langle (T\ A^*), \langle\rangle\rangle}}{\vdash_{\text{lnf}}\ \langle (T\ A), \langle\rangle\rangle \leq \langle (T\ A?), \langle\rangle\rangle | \langle (T\ A^*), \langle\rangle\rangle}}{\vdash_{\text{sub}}\ (T\ A) \leq (T\ A?)|(T\ A^*)} \tag{5.6}$$

The up-cast coercions arising out of both subtype proofs are incomparable. In the first case, we inject $T\ A$ into the left component via the coercion arising from the sub-statement $\vdash_{\text{sub}} (T\ A) \leq (T\ A?)$. In the other case, $T\ A$ is injected into the right component via the sub-statement $\vdash_{\text{sub}} (T\ A) \leq (T\ A^*)$. The problem is that we can observe the difference in behavior of both up-cast coercions by first testing for the pattern $(x : T\ A^*)$ followed by testing for $(y : T\ A?)$.

The source of the problem is that data types $T\ A?$ and $T\ A^*$ are treated as different labels and cannot be combined into a single monomial. However, they share some common values. Similar observations can be made for function and polymorphic types. On the other hand, $(A|A)$ is turned into the single monomial $\langle A, \langle\rangle\rangle$. Therefore, there is exactly one subtype proof (up-cast coercion) for the statement $\vdash_{\text{sub}} A \leq (A|A)$. To ensure that up-cast coercions arising out of subtype proofs behave deterministically we simply restrict label subtyping.

There is still a minor issue we haven't addressed. Recall that the partial derivative operation $d(l \shortmid t)$ is not deterministic. For instance, $pd(A \shortmid (\langle A^*, A^*\rangle) = \{\langle A^*, A^*\rangle, A^*\}$. Note that the result is a set. The order among the list elements does not matter. Thus $pd(A \shortmid (\langle A^*, A^*\rangle)) = \{A^*, \langle A^*, A^*\rangle\}$ is valid, too. Therefore, $d(A \shortmid (\langle A^*, A^*\rangle))$ has two solutions, $(\langle A^*, A^*\rangle | A^*)$ and $(A^* | \langle A^*, A^*\rangle)$. Although these two types are semantically equivalent, they are still syntactically different. As a result, the proof and the resulting proof terms may be syntactically different, even though their meanings are the same. To fix that we impose an order on the resulting set of $pd(l \shortmid t)$, such that the order respects the construction of $pd(l \shortmid t)$ as stated in Figure 4.4b. In other words, we view the result is a list with unique elements instead of a set. For instance, in the above example we expect $d(A \shortmid (\langle A^*, A^*\rangle))$ evaluates to $(\langle A^*, A^*\rangle | A^*)$ but not $(A^* | \langle A^*, A^*\rangle)$. Similarly, we expect $d(A \shortmid (\langle A, B\rangle | \langle A, C\rangle))$ evaluates to $(B|C)$ but not $(C|B)$.

Lemma 8 (Coherence of Coercive Subtyping) *If we replace (LN) in Figure 4.4a by rule (LN') (see Section 4.5), then for each subtype statement there exists at most one subtype proof.*

The above lemma immediately guarantees that our coercive subtype proof system is coherent. Each provable subtype statement gives rise to exactly one up-cast coercion.

Because of the non-syntax directed typing rule (Sub), we can simplify a sequence of subtype steps $t_1 \leq ... \leq t_n$ (1) by a single subtype step $t_1 \leq t_n$ (2). Hence, we also need to guarantee that the composition of up-cast coercions from (1) is compatible with the up-cast coercion from (2).

Such a guarantee is provided by the following lemma.

Lemma 9 (Transitivity of Coercive Subtyping) *Let $\vdash_{sub} t_1 \leq^{u_1} t_2$, $\vdash_{sub} t_2 \leq^{u_2} t_3$, $\vdash_{sub} t_1 \leq^{u_3} t_3$ and v be a value of type $[\![t_1]\!]$. Then, $u_2\ (u_1\ v) \overset{t_3}{\leftrightarrow} u_3\ v$.*

We use here the relation $\cdot \overset{\cdot}{\leftrightarrow} \cdot$ introduced in Figure 5.5 to ensure the compatibility between (1) and (2).

5.6.2 Coherence of Coercive Pattern Matching in Combination with Coercive Subtyping

We have yet to study interactions among coercive subtyping and coercive pattern matching. This can have subtle consequences on the translation of expressions. For example, consider the case expression $\mathsf{case}\ e\ \mathsf{of}\ p \rightarrow e'$. Suppose we find two translations for the above making use of the following two intermediate translations $\vdash\ e : t_1 \leadsto E_1$ and $\vdash\ e : t_2 \leadsto E_2$. It is possible to assign different types to e due to the non-syntax directed subtyping rule (Sub).

Suppose t is the type of the pattern p. In one case, we express the pattern p via the down-cast d_1 derived from $\vdash_{\text{sub}}\ t \leq_{d_1} t_1$ and in the other case we make use of the down-cast d_2 derived from $\vdash_{\text{sub}}\ t \leq_{d_2} t_2$. To guarantee that both translations of the pattern match behave the same, we need to show that d_1 applied to E_1 and d_2 applied to E_2 yield the same result. Let's denote this property by $E_1 \overset{t_1}{\rightarrow}\overset{t_2}{\leftarrow} E_2$. Our actual task is to show that this property is preserved under coercive subtyping. Here are the formal details.

Definition 4 (Coercive Pattern Matching Equivalence) *Let t_1 and t_2 be two source types and E_1 and E_2 be two target expressions such that $\vdash_F\ E_1 : [\![t_1]\!]$ and $\vdash_F\ E_2 : [\![t_2]\!]$. We define $E_1 \overset{t_1}{\rightarrow}\overset{t_2}{\leftarrow} E_2$ iff for any source type t where $\vdash_{sub}\ t \leq_{d_1} t_1$ and $\vdash_{sub}\ t \leq_{d_2} t_2$ we have that $d_1\ E_1 = d_2\ E_2$.*

Example 35 For example, we consider $\vdash_{\mathrm{F}} (L\ A) : [\![(A|\langle\rangle)]\!]$ and $\vdash_{\mathrm{F}} [A] : [\![A^*]\!]$. By Definition 4, we find that $L\ A \overset{(A|\langle\rangle) A^*}{\rightarrow\leftarrow} [A]$ is a valid statement. It is based on the following observation. There are three common subtypes of $(A|\langle\rangle)$ and A^*, namely, $\langle\rangle$, A and $(A|\langle\rangle)$, where

1. $\vdash_{\mathrm{sub}} \langle\rangle \leq_{d_1} (A|\langle\rangle)$ and $\vdash_{\mathrm{sub}} \langle\rangle \leq_{d'_1} A^*$, where
2. $\vdash_{\mathrm{sub}} A \leq_{d_2} (A|\langle\rangle)$ and $\vdash_{\mathrm{sub}} A \leq_{d'_2} A^*$,
3. $\vdash_{\mathrm{sub}} (A|\langle\rangle) \leq_{d_3} (A|\langle\rangle)$ and $\vdash_{\mathrm{sub}} (A|\langle\rangle) \leq_{d'_3} A^*$,

We have $d_i\ (L\ A) = d'_i\ [A]$ for $i \in \{1,2,3\}$. For instance, from

d_1 `(L v) = Nothing`

d_1 `(R v) = Just v`

d'_1 `[] = Just ()`

d'_1 `(x:xs) = Nothing`

we find that $d_1\ (L\ A) \longrightarrow \mathit{Nothing}$ and $d'_1\ [A] \longrightarrow \mathit{Nothing}$. Similarly,

d_2 `(R v) = Nothing`

d_2 `(L v) = Just v`

d'_2 `[A] = Just A`

d'_2 `_ = Nothing`

we find that $d_2\ (L\ A) \longrightarrow \mathit{Just}\ A$ and $d'_2\ [A] \longrightarrow \mathit{Just}\ A$. Similar observation applies to d_3 and d'_3.

On the other hand, we consider $\vdash_{\mathrm{F}} (R\ \langle\rangle) : [\![(A|\langle\rangle)]\!]$ and $\vdash_{\mathrm{F}} [A] : [\![A^*]\!]$. It is clear that $R\ \langle\rangle \overset{(A|\langle\rangle) A^*}{\rightarrow\leftarrow} [A]$ does not hold, because $d_1\ (R\ \langle\rangle) \longrightarrow \mathit{Just}\ ()$, but $d'_1\ [A] \longrightarrow \mathit{Nothing}$. □

Lemma 10 (Preservation of Coercive Pattern Matching Equivalence) *Let t_1 and t_2 be two source types and E_1 and E_2 be two target expressions such that $\vdash_F E_1 : [\![t_1]\!]$ and $\vdash_F E_2 : [\![t_2]\!]$ and $E_1 \overset{t_1\ t_2}{\rightarrow\leftarrow} E_2$. Let t_3 be a source type such that $\vdash_{sub} t_2 \leq^u t_3$. Then, we have that $E_1 \overset{t_1\ t_3}{\rightarrow\leftarrow} u\ E_2$.*

Lemma 11 (Semantic Equiv. Implies Coercive Pattern Matching Equiv.) *Let t be a source type and E_1 and E_2 be two target expressions such that $\vdash_F E_1 : [\![t]\!]$ and $\vdash_F E_2 : [\![t]\!]$ and $E_1 \overset{t}{\leftrightarrow} E_2$. Then, we have that $E_1 \overset{t}{\rightarrow}\overset{t}{\leftarrow} E_2$.*

5.6.3 Coherence of Translation

Finally, we can conclude that our translation is coherent.

Theorem 6 (Coherence) *Let $\vdash e : t_1 \rightsquigarrow E_1$ and $\vdash e : t_2 \rightsquigarrow E_2$. Then, $E_1 \overset{t_1}{\rightarrow}\overset{t_2}{\leftarrow} E_2$.*

5.7 Summary

We have developed a type-directed translation scheme from System F* to System F. The idea is to apply proofs-are-programs principle to extract subtype coercion functions out of the subtype proof. We sketched the definitions of these coercion functions. Using the extracted coercion functions, we translate semantic subtyping and pattern matching. The formal result shows that the translated program is always well-typed.

We studied the coherence problem in the context of semantic subtyping and regular expression pattern matching. We established some conditions under which our translation scheme is coherent.

In the next chapter, we will study the regular expression pattern matching problem and provide the complete details of coercive subtyping and coercive pattern matching.

Chapter 6

Regular Expression Pattern Matching

In this chapter, we give an in-depth discussion of regular expression pattern matching. We first give a general characterization of the regular expression pattern matching and its many matching policies (Section 6.1). We then develop a regular expression pattern matching algorithm by rewriting regular expressions using Brzozowski's derivative operation on regular expressions (Section 6.2). This then leads to another rewriting based algorithm for coercive pattern matching (Section 6.3).

The material covered in this chapter is related to the previous chapter as follows.

- We fill in the details of the previously "sketched" coercive pattern matching approach which was first mentioned in Chapter 5.
- We verify the correctness of the coercive pattern matching algorithm under the POSIX/Longest matching policy.
- Then it follows that the translation of System F* to System F introduced in Chapter 5 is correct under the POSIX/Longest matching policy.

All algorithms in this chapter are written in Haskell-style pseudo code.

6.1 The Regular Expression Pattern Matching Problem

The main difference between regular expression pattern matching and pattern matching found in ML/Haskell is that we cannot pattern match by comparing the structure of the pattern against the structure of the incoming value. The reason is that a pattern like $\langle x : A^*\rangle$ matches with $\langle\rangle$ (the "epsilon"), A, $\langle A, A\rangle$ and so on. This is because the regular expression A^* semantically denotes the set of words constructed by repeating A for zero or more times. The challenge is that we need to take into account the semantic meaning of the regular expression when performing the pattern matching. Here is an example.

Example 36 Matching the word $\langle A, B\rangle$ against the pattern $(x : A?, y : ((A, B)|B))$ yields two possible value bindings, namely $\{(A/x), (B/y)\}$ and $\{(\langle\rangle/x), (\langle A, B\rangle/y)\}$. $\square$

This example shows that regular expression pattern matching is indeterministic. (We may also say the regular expression pattern is ambiguous.) Regular expression pattern matching can be made deterministic if we fix a specific pattern matching policy.

In the following, we consider a regular expression pattern language which is a subset of the System F* language specified in Figure 4.1, Chapter 4,

$$\begin{array}{rrcl}
\text{Patterns} & p & ::= & (x : t) \| \langle p, p\rangle \| (p|p) \\
\text{Types} & t & ::= & l \| \langle t, t\rangle \| \langle\rangle \| t^* \| (t|t) \\
\text{Words} & w & ::= & \langle\rangle \| l \| \langle w, w\rangle \\
\text{Literals} & l & ::= & A \| B ...
\end{array}$$

The language in the above defines the regular expression type fragment of System F*. In addition, we extend the pattern language with choice pattern $(p|p)$. The choice

$$\text{(Var)}\ \frac{w \in L(t)}{w \lhd (x : t) \leadsto \{w/x\}} \qquad \text{(Seq)}\ \frac{\begin{array}{c} w \sim \langle w_1, w_2 \rangle \\ w_1 \lhd p_1 \leadsto \theta_1 \\ w_2 \lhd p_2 \leadsto \theta_2 \end{array}}{w \lhd \langle p_1, p_2 \rangle \leadsto \theta_1 \cup \theta_2}$$

$$\text{(Choice1)}\ \frac{w \lhd p_1 \leadsto \theta_1}{w \lhd (p_1|p_2) \leadsto \theta_1} \qquad \text{(Choice2)}\ \frac{w \lhd p_2 \leadsto \theta_2}{w \lhd (p_1|p_2) \leadsto \theta_2}$$

Figure 6.1: Pattern Matching Relation

patterns will appear in some intermediate steps of the pattern matching algorithm.

The pattern matching relation is described in terms of judgment $w \lhd p \leadsto \theta$. $w \lhd p \leadsto \theta$ is pronounced as "a word w matches a pattern p and produces a value binding environment θ". In Figure 6.1, we describe all valid pattern matching relation. We write $L(t)$ to denote the language described by a regular expression t. Rule (Var) states that a word matches with a binder pattern if the word is in the language denoted by the pattern annotation. In rule (Seq), we pattern match a word against a sequence pattern. We split the word w in-deterministically via $w \sim \langle w_1, w_2 \rangle$, such that w_1 is matching with sub-pattern p_1 and w_2 is matching with p_2. The operation $\cdot \sim \cdot$ was defined earlier in Figure 4.7 in Chapter 4. In rules (Choice1) and (Choice2), we match a choice pattern in-deterministically.

We consider some examples,

Example 37 Consider $\langle A, A \rangle \lhd \langle x : A^*, y : A^* \rangle \leadsto \theta$. According to the matching relation defined above, we find the following derivations are possible.

$$\frac{\langle A, A \rangle \sim \langle \langle \rangle, \langle A, A \rangle \rangle \quad \dfrac{\langle \rangle \in L(A^*)}{\langle \rangle \lhd x : A^* \leadsto \{\langle \rangle / x\}} \quad \dfrac{\langle A, A \rangle \in L(A^*)}{\langle A, A \rangle \lhd y : A^* \leadsto \{\langle A, A \rangle / y\}}}{\langle A, A \rangle \lhd \langle x : A^*, y : A^* \rangle \leadsto \{\langle \rangle / x, \langle A, A \rangle / y\}} \tag{6.1}$$

$$\frac{\langle A,A\rangle \sim \langle A,A\rangle \quad \dfrac{A \in L(A^*)}{A \lhd x : A^* \leadsto \{A/x\}} \quad \dfrac{A \in L(A^*)}{A \lhd y : A^* \leadsto \{A/y\}}}{\langle A,A\rangle \lhd \langle x : A^*, y : A^*\rangle \leadsto \{A/x, A/y\}} \tag{6.2}$$

$$\frac{\langle A,A\rangle \sim \langle\langle A,A\rangle,\langle\rangle\rangle \quad \dfrac{\langle A,A\rangle \in L(A^*)}{\langle A,A\rangle \lhd x : A^* \leadsto \{\langle A,A\rangle/x\}} \quad \dfrac{\langle\rangle \in L(A^*)}{\langle\rangle \lhd y : A^* \leadsto \{\langle\rangle/y\}}}{\langle A,A\rangle \lhd \langle x : A^*, y : A^*\rangle \leadsto \{\langle A,A\rangle/x, \langle\rangle/y\}} \tag{6.3}$$

Based on three different ways of splitting the sequence $\langle A, A\rangle$, the three derivations in the above lead to three different pattern matching results. For instance, in derivation 6.1, we split $\langle A, B\rangle$ into $\langle\rangle$ and $\langle A, A\rangle$, so that $\langle\rangle$ matches with sub-pattern $x : A^*$ and $\langle A, A\rangle$ matches with sub-pattern $y : A^*$. □

Example 38 Matching $\langle A, B\rangle$ against $\langle x : A?, y : (\langle A, B\rangle|B)\rangle$ yields two possible value substitutions.

$$\frac{\langle A,B\rangle \sim \langle A,B\rangle \quad \dfrac{A \in L(A?)}{A \lhd x : A? \leadsto \{A/x\}} \quad \dfrac{B \in L(\langle A,B\rangle|B)}{B \lhd y : (\langle A,B\rangle|B) \leadsto \{B/y\}}}{\langle A,B\rangle \lhd \langle x : A?, y : (\langle A,B\rangle|B)\rangle \leadsto \{A/x, B/y\}} \tag{6.4}$$

$$\frac{\begin{array}{c}\langle A,B\rangle \sim \langle\langle\rangle,\langle A,B\rangle\rangle \\ \dfrac{\langle\rangle \in L(A?)}{\langle\rangle \lhd x : A? \leadsto \{\langle\rangle/x\}} \quad \dfrac{\langle A,B\rangle \in L(\langle A,B\rangle|B)}{\langle A,B\rangle \lhd y : (\langle A,B\rangle|B) \leadsto \{\langle A,B\rangle/y\}}\end{array}}{\langle A,B\rangle \lhd \langle x : A?, y : (\langle A,B\rangle|B)\rangle \leadsto \{\langle\rangle/x, \langle A,B\rangle/y\}} \tag{6.5}$$

□

In this section, we presented the regular expression pattern matching problem. We found that regular expression pattern matching is indeterministic unless a matching policy is fixed. In the next section we develop an algorithm for regular expression pattern matching.

6.2 Derivative Based Pattern Matching

Our idea is to solve the pattern matching problem $w \lhd p \leadsto \theta$ by rewriting p. We rewrite the regular expression pattern using the classic derivative operation [14] extended to regular expression pattern.

We illustrate the concept of derivatives by first considering the word problem: Given word w and regular expression t check if $w \in L(t)$. From the derivative-based word problem algorithm we will then derive our derivative-based regular expression pattern matching algorithm.

6.2.1 The Word Problem

We solve the word problem by rewriting the regular expression t. The word matching algorithm can be described in terms of the following function `match`.

```
$w$ ‘matches‘ $t$ =  case $w$ of
                     $\langle\rangle \rightarrow$ isEmpty $t$
                     $\langle l, v\rangle \rightarrow v$ ‘matches‘ $(t/l)$
```

Note that the pseudo-code we used is in Haskell-style, e.g., w ‘matches‘ t is the infix representation of the application (matches w t), in which the function application (isEmpty t) yields `True` if t accepts the empty word $\langle\rangle$, and yields `False` otherwise. t/l denotes the *derivative* of t with respect to l. We compute the derivative t/l by “taking away” the “leading” literal l from t. Semantically, we can explain the

$$
\begin{array}{lcl}
w \text{ `matches` } t & = & \mathsf{case}\ w\ \mathsf{of} \\
 & & \langle\rangle \rightarrow \text{isEmpty } t \\
 & & \langle l, v\rangle \rightarrow v\text{`matches` } t/l \\
 & & \\
\text{isEmpty } t^* & = & True \\
\text{isEmpty } (t_1|t_2) & = & (\text{isEmpty } t_1)\ \ ||\ \ (\text{isEmpty } t_2) \\
\text{isEmpty } \langle\rangle & = & True \\
\text{isEmpty } \langle t_1, t_2\rangle & = & (\text{isEmpty } t_1)\ \&\&\ (\text{isEmpty } t_2) \\
\text{isEmpty } l & = & False \\
\text{isEmpty } \bot & = & False \\
 & & \\
\bot/l & = & \bot \\
\langle\rangle/l & = & \bot \\
l_1/l_2 & = & \mathsf{if}\ l_1 == l_2\ \mathsf{then}\ \langle\rangle\ \mathsf{else}\bot \\
(t_1|t_2)/l & = & (t_1/l)|(t_2/l) \\
\langle t_1, t_2\rangle/l & = & \mathsf{if}\ \text{isEmpty } t_1\ \mathsf{then}\ (\langle t_l/l, t_2\rangle|t_2/l)\ \mathsf{else}\ \langle t_1/l, t_2\rangle \\
t^*/l & = & \langle t/l, t^*\rangle
\end{array}
$$

Figure 6.2: The algorithm for the word problem

derivative operation as follows,

$$L(t/l) = \{w|\langle l, w\rangle \in L(t)\}$$

Operationally, we can define a function that computes a regular expression representing the derivative of t with respect to l.

In Figure 6.2, we present the algorithm for the word problem. We use case expression for pattern matching, and use && to denote the “logical and operator”, || to denote the “logical or operator”. $\bot$ denotes the empty regular expression, that is $L(\bot) = \emptyset$.

Example 39 Consider the word problem A ‘matches‘ $(A|B)^*$, since $A \sim \langle A, \langle\rangle\rangle$, we reduce the word problem to $\langle\rangle$ 'matches' $(A|B)^*/A$ Since

$$\begin{array}{rcl} (A|B)^*/A & \longrightarrow & \langle (A|B)/A, (A|B)^* \rangle \\ & \longrightarrow & \langle ((A/A)|(B/A)), (A|B)^* \rangle \\ & \longrightarrow & \langle ((\langle\rangle|\bot), (A|B)^* \rangle \end{array}$$

It is clear that

$$\begin{array}{rcl} \text{isEmpty } \langle ((\langle\rangle|\bot), (A|B)^* \rangle & \longrightarrow & (\text{isEmpty } (\langle\rangle|\bot))\&\&(\text{isEmpty } (A|B)^*) \\ & \longrightarrow & ((\text{isEmpty } \langle\rangle) \mid\mid (\text{isEmpty } \bot))\&\&True \\ & \longrightarrow & (True \mid\mid False)\&\&True \\ & \longrightarrow & True \end{array}$$

Therefore, running the algorithm with the word problem A 'matches' $(A|B)^*$ yields $True$. □

A complete, runnable implementation of the word algorithm in Haskell is given in Appendix A.1.

We solved the word problem by rewriting the regular expression using derivative operation. The rewriting idea applies to the regular expression pattern matching problem as we show in the up-coming section.

6.2.2 Towards a Regular Expression Pattern Matching Algorithm

Let us go back to the regular expression pattern matching problem, $w \lhd p \rightsquigarrow \theta$. Our idea is to extend the derivative operation to regular expression patterns.

Extending derivative operation to regular expression pattern

Recall that in the previous subsection, we solve the word problem by reducing

$$\langle l, w \rangle \text{ `matches` } t$$

to

$$w \text{ `matches` } t/l$$

If we are able to extend the derivative operation to regular expression pattern, we can solve the regular expression pattern matching problem by reducing

$$\langle l, w \rangle \lhd p \rightsquigarrow \theta$$

to

$$w \lhd p/l \rightsquigarrow \theta$$

which effectively means that a sequence $\langle l, w \rangle$ matches the pattern p and yields a substitution θ iff w matches the pattern derivative p/l producing the substitution θ.

The next problem is how to define the derivative operation for regular expression pattern, p/l. We first consider the easy case, where we define $(p_1|p_2)/l = (p_1/l)|(p_2/l)$. What about $(x : t)/l$? It is wrong to define $(x : t)/l = (x : (t/l))$. The reason is that building the derivative p/l implies that we have consumed the input symbol l. We must record somewhere that we have consumed l.

A simple solution is that in the variable pattern we record the sequence of literals which has been consumed by the variable pattern so far. To record the sequence of consumed literals, we adjust the syntax of the binder pattern.

$$\text{Patterns} \quad p \quad ::= \quad ([w]\ x : t) \| \ldots$$

where w denotes for the sequence of literals consumed by the pattern $(x : t)$ so far.

Thus any valid pattern binding will look like $\{(\langle w, v\rangle/x)\}$, where v is some word which we yet need to consume with t, in other words, $v \in L(t)$. Thus the pattern matching relation for variable pattern is adjusted as follows.

$$\text{(Var)}\quad \frac{w \in L(t)}{w \lhd ([w']\ x : t) \leadsto \{\langle w', w\rangle/x\}}$$

From this point onwards, we often write $(x : t)$ where we mean $([\langle\rangle]\ x : t)$.

The derivative definition for variable patterns is then straightforwardly defined as $([w]\ x : t)/l = ([\langle w, l\rangle]\ x : (t/l))$. Here is an example,

Example 40 We consider the pattern matching problem $\langle A, B\rangle \lhd x : (A|B)^* \leadsto \theta$. We solve the above pattern matching problem by building derivatives.

First, we reduce $\langle A, B\rangle \lhd (x : (A|B)^*) \leadsto \theta$ to

$$B \lhd (x : (A|B)^*)/A \leadsto \theta \tag{6.6}$$

To proceed, we want to compute the derivative $(x : (A|B)^*)/A$. According to the definition, we have $([\langle\rangle]\ x : (A|B)^*)/A = ([A]\ x : ((A|B)^*/A)) = ([A]\ x : (A|B)^*)$. Thus, the problem 6.6 is reduced to

$$B \lhd ([A]\ x : (A|B)^*) \leadsto \theta \tag{6.7}$$

In the next step, we reduce the problem 6.7 to

$$\langle\rangle \lhd ([A]\ x : (A|B)^*)/B \leadsto \theta \tag{6.8}$$

where $([A]\ x : (A|B)^*)/B = ([\langle A, B\rangle]\ x : (A|B)^*/B) = ([\langle A, B\rangle]\ x : (A|B)^*)$. Since the input word is fully consumed by the pattern, we can construct the resulting substitution by reading the binding. Therefore, we have the result $\theta = \{(\langle A, B\rangle/x)\}$.

$\Box$

What about the derivative definition for pair patterns? The immediate definition we can think of is as follows,

$$\begin{array}{rcl} \langle p_1, p_2\rangle/l & = & \mathsf{if}\ \text{isEmpty (stript } p_1) \\ & & \mathsf{then}\ \langle p_1/l, p_2\rangle|\langle\Box, p_2/l\rangle \\ & & \mathsf{else}\ \langle p_1/l, p_2\rangle \end{array}$$

where stript p extracts the regular expression types from a pattern p. The definition will be provided shortly.

The question is what to replace $\Box$ with. If p_1 accepts the empty word, this means we could match all further input just with p_2. But we simply cannot discard p_1 because we have recorded the variable binding in the pattern itself. On the other hand, we cannot keep p_1 as it is. We must somehow indicate that the matching using p_1 is finished.

The idea is to replace $\Box$ by a variation of p_1 where we make all regular expressions empty whichever is possible.

Example 41 We consider the pattern $\langle [\langle A, B\rangle]\ x : (A|B)^*, [\langle\rangle]\ y : C^*\rangle$, where $x : (A|B)^*$ has already consumed $\langle A, B\rangle$ and C is the remaining input.

We expect that,

$$\begin{array}{ll} \langle ([\langle A, B\rangle]\ x : (A|B)^*),\ ([\langle\rangle]\ y : C^*)\rangle/C & \longrightarrow \\ \langle ([\langle A, B, C\rangle]\ x : \bot),\ ([\langle\rangle]\ y : C^*)\rangle|\langle ([\langle A, B\rangle]\ x : \langle\rangle),\ ([C]\ y : C^*)\rangle & \end{array}$$

where $(A|B)^*/C = \bot$. This shows that if consuming C with sub-pattern $([\langle A, B\rangle]\ x : (A|B)^*)$ leads to a "failure state" from which we can't obtain any valid pattern binding.

On the other hand, since $(A|B)^*$ accepts the empty word $\langle\rangle$, we can stop matching using the sub-pattern $([\langle A, B\rangle]\ x : (A|B)^*)$ by replacing $(A|B)^*$ with $\langle\rangle$ and we

$$
\begin{array}{lcl}
\text{allmatch } w\ p & = & \textsf{let } p' = \text{build } w\ p \textsf{ in } \text{collect } p' \\
\\
\text{build } w\ p & = & \text{fold } (\lambda l.\lambda p'.p'/l)\ p\ w \\
\\
\text{fold } f\ p\ \langle\rangle & = & p \\
\text{fold } f\ p\ \langle l, w\rangle & = & \text{fold } f\ (f\ l\ p)\ w \\
\\
\text{collect } ([w]\ x : t) & = & \textsf{if } \text{isEmpty } t \textsf{ then } \{\{(w/x)\}\} \textsf{ else } \{\} \\
\text{collect } (p_1|p_2) & = & (\text{collect } p_1) \cup (\text{collect } p_2) \\
\text{collect } \langle p_1, p_2\rangle & = & \text{combine } (\text{collect } p_1)\ (\text{collect } p_2) \\
\\
\text{combine } xs\ ys & = & \{x \cup y | x \in xs, y \in ys\} \\
\\
(p_1|p_2)/l & = & (p_1/l)|(p_2/l) \\
([w]\ x : t)/l & = & ([\langle w, l\rangle]\ x : (t/l)) \\
\langle p_1, p_2\rangle/l & = & \textsf{if } \text{isEmpty } (\text{stript } p_1) \\
 & & \textsf{then } \langle p_1/l, p_2\rangle | \langle \text{mkEmpPat } p_1, p_2/l\rangle \\
 & & \textsf{else } \langle p_1/l, p_2\rangle \\
\\
\text{mkEmpPat } ([w]\ x : t) & = & \textsf{if } \text{isEmpty } t \textsf{ then } ([w]\ x : \langle\rangle) \textsf{ else } ([w]\ x : \bot) \\
\text{mkEmpPat } \langle p_1, p_2\rangle & = & \langle \text{mkEmpPat } p_1, \text{mkEmpPat } p_2\rangle \\
\text{mkEmpPat } (p_1|p_2) & = & (\text{mkEmpPat } p_1)|(\text{mkEmpPat } p_2) \\
\\
\text{stript } ([w]\ x : t) & = & t \\
\text{stript } \langle p_1, p_2\rangle & = & \langle \text{stript } p_1, \text{stript } p_2\rangle \\
\text{stript } (p_1|p_2) & = & (\text{stript } p_1)|(\text{stript } p_2)
\end{array}
$$

Figure 6.3: A pattern matching algorithm that implements “all match” semantics

consume C with the sub-pattern $([\langle\rangle]\ y : C^*)$. □

In general, to stop matching using sub-pattern p_1 in $\langle p_1, p_2\rangle$,we need to make the sub pattern p_1 empty by replacing all regular expressions which accept $\langle\rangle$ by $\langle\rangle$ and all others we replace by $\bot$.

We extended the derivative operation to regular expression pattern. We are ready to put everything together and implement a matching algorithm.

Implementing the "all match" semantics

In Figure 6.3, we describe the derivative based regular expression pattern matching algorithm in terms of a function allmatch w p. allmatch w p computes *all* possible results of matching w against p (i.e. a set of value binding environments $\{\theta_1, ..., \theta_n\}$).

The algorithm consists of two parts.

- In the first part, we use a helper function build w p which builds the derivative of a regular expression pattern p with respect to the input word w. This operation is carried out by "pushing all labels in w into p" by a fold operation.

- In the second part, we retrieve the matching results by collecting the recorded value bindings from the (re-written) pattern. This is achieved by using the other helper function collect p to access a particular matching result.

In this above definition, we are implementing the "all match" semantics, under which we collect all the possible matching results. The first clause of collect extracts the value binding from a variable pattern, if the pattern accepts the empty word $\langle\rangle$. Otherwise, an empty set is returned to signal pattern matching failure. The second clause extracts value bindings from a choice pattern. We have to union the two sub results, since we are implementing the "all match" semantics. The fourth clause deals with pattern $\langle p_1, p_2\rangle$. We use a helper function combine to build the aggregated result set from the two intermediate results coming from allmatch $\langle\rangle$ p_1 and allmatch $\langle\rangle$ p_2. p/l computes the pattern derivative of p with respect to l. We often write $p/\langle l_1, ..., l_n\rangle$ where we actually mean $(p/l_1) \ldots /l_n$. Function mkEmpPat p replaces all regular expression types in p by $\langle\rangle$ if possible. Function stript p extracts the regular expression type from p.

Example 42 We are finding all possible match results from the pattern matching problem "matching $\langle A, A, A\rangle$ against the pattern $\langle (x : A^*), (y : A^*)\rangle$.". We apply the pattern matching algorithm in Figure 6.3 to this problem as follows,

$$\begin{array}{rl}
& \text{allmatch } \langle A, A, A\rangle\ \langle([\langle\rangle]\ x : A^*), ([\langle\rangle]\ y : A^*)\rangle \\
\longrightarrow & \mathsf{let}\ p = \text{build } \langle A, A, A\rangle\ \langle([\langle\rangle]\ x : A^*), ([\langle\rangle]\ y : A^*)\rangle\ \mathsf{in}\ \text{collect } p
\end{array}$$

We proceed with the above execution by breaking it into the building phase and the collection phase.

- The building phase

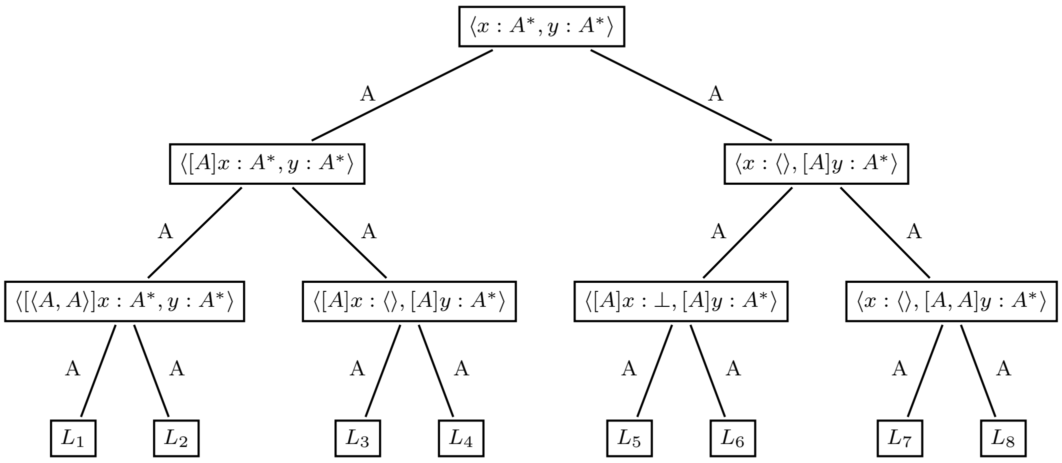
$$\begin{array}{rlr}
& \text{build } \langle A,A,A\rangle\ \langle([\langle\rangle]\ x : A^*), ([\langle\rangle]\ y : A^*)\rangle & (1) \\
\longrightarrow & \text{fold } (\lambda l.\lambda p.p/l)\ \langle([\langle\rangle]\ x : A^*), ([\langle\rangle]\ y : A^*)\rangle\ \langle A,A,A\rangle & (2) \\
\longrightarrow & \text{fold } (\lambda l.\lambda p.p/l)\ \langle([\langle\rangle]\ x : A^*), ([\langle\rangle]\ y : A^*)\rangle/A\ \langle A,A\rangle & (3) \\
\longrightarrow & \text{fold } (\lambda l.\lambda p.p/l)\ (\langle([\langle\rangle]\ x : A^*)/A, ([\langle\rangle]\ y : A^*)\rangle|\langle\text{mkEmpPat } ([\langle\rangle]\ x : A^*), ([\langle\rangle]\ y : A^*)/A\rangle)\ \langle A,A\rangle & (4) \\
\longrightarrow & \text{fold } (\lambda l.\lambda p.p/l)\ (\langle([A]\ x : A^*), ([\langle\rangle]\ y : A^*)\rangle|\langle([\langle\rangle]\ x : \langle\rangle), ([A]\ y : A^*)\rangle)\ \langle A,A\rangle & (5) \\
\longrightarrow & \text{fold } (\lambda l.\lambda p.p/l)\ ((\langle([A]\ x : A^*), ([\langle\rangle]\ y : A^*)\rangle|\langle([\langle\rangle]\ x : \langle\rangle), ([A]\ y : A^*)\rangle)/A)\ A & (6) \\
\longrightarrow & \text{fold } (\lambda l.\lambda p.p/l)\ (\langle([A]\ x : A^*), ([\langle\rangle]\ y : A^*)\rangle/A|\langle([\langle\rangle]\ x : \langle\rangle), ([A]\ y : A^*)\rangle/A)\ A & (7) \\
\longrightarrow & \text{fold } (\lambda l.\lambda p.p/l)\ (\langle([A]\ x : A^*)/A, ([\langle\rangle]\ y : A^*)\rangle|\langle\text{mkEmpPat } ([A]\ x : A^*), ([\langle\rangle]\ y : A^*)/A\rangle) & (8) \\
& \qquad\qquad |\ (\langle([\langle\rangle]\ x : \langle\rangle)/A, ([A]\ y : A^*)\rangle|\langle\text{mkEmpPat } [\langle\rangle]\ x : \langle\rangle, ([A]\ y : A^*)/A\rangle)\ A & \\
\longrightarrow & \text{fold } (\lambda l.\lambda p.p/l)\ (\langle[\langle A,A\rangle]\ x : A^*), ([\langle\rangle]\ y : A^*)\rangle|\langle([A]\ x : \langle\rangle), ([A]\ y : A^*)\rangle & \\
& \qquad\qquad |\ \langle([A]\ x : \bot), ([A]\ y : A^*)\rangle|\langle([\langle\rangle]\ x : \langle\rangle), ([\langle A,A\rangle]\ y : A^*)\rangle\ A & (9) \\
\longrightarrow & \text{fold } (\lambda l.\lambda p.p/l)\ (\langle([\langle A,A\rangle]\ x : A^*), ([\langle\rangle]\ y : A^*)\rangle/A) & (10) \\
& \qquad\qquad |\ \langle([A]\ x : \langle\rangle), ([A]\ y : A^*)\rangle/A & \\
& \qquad\qquad |\ \langle([A]\ x : \bot), ([A]\ y : A^*)\rangle/A & \\
& \qquad\qquad |\ \langle([\langle\rangle]\ x : \langle\rangle), ([\langle A,A\rangle]\ y : A^*)\rangle/A)\ \langle\rangle & \\
\longrightarrow & \text{fold } (\lambda l.\lambda p.p/l)\ (\langle([\langle A,A,A\rangle]\ x : A^*), ([\langle\rangle]\ y : A^*)\rangle|\langle([\langle A,A\rangle]\ x : \langle\rangle), ([A]\ y : A^*)\rangle & (11) \\
& \qquad\qquad |\ \langle([\langle A,A\rangle]\ x : \bot), ([A]\ y : A^*)\rangle|\langle([A]\ x : \langle\rangle), ([\langle A,A\rangle]\ y : A^*)\rangle & \\
& \qquad\qquad |\ \langle([\langle A,A\rangle]\ x : \bot), ([A]\ y : A^*)\rangle & \\
& \qquad\qquad |\ \langle([A]\ x : \bot), ([\langle A,A\rangle]\ y : A^*)\rangle|\langle([\langle\rangle]\ x : \langle\rangle), ([\langle A,A,A\rangle]\ y : A^*)\rangle\ \langle\rangle & \\
\longrightarrow & (\langle([\langle A,A,A\rangle]\ x : A^*), ([\langle\rangle]\ y : A^*)\rangle|\langle([\langle A,A\rangle]\ x : \langle\rangle), ([A]\ y : A^*)\rangle & (12) \\
& \qquad\qquad |\ \langle([\langle A,A\rangle]\ x : \bot), ([A]\ y : A^*)\rangle|\langle([A]\ x : \langle\rangle), ([\langle A,A\rangle]\ y : A^*)\rangle & \\
& \qquad\qquad |\ \langle([\langle A,A\rangle]\ x : \bot), ([A]\ y : A^*)\rangle & \\
& \qquad\qquad |\ \langle([A]\ x : \bot), ([\langle A,A\rangle]\ y : A^*)\rangle|\langle([\langle\rangle]\ x : \langle\rangle), ([\langle A,A,A\rangle]\ y : A^*)\rangle &
\end{array}$$

- The collection phase

$$
\begin{array}{rl}
& \text{collect } (\langle([\langle A,A,A\rangle]\ x : A^*),([\langle\rangle]\ y : A^*)\rangle | \langle([\langle A,A\rangle]\ x : \langle\rangle),([A]\ y : A^*)\rangle \qquad (13)\\
& \quad | \ \langle([\langle A,A\rangle]\ x : \bot),([A]\ y : A^*)\rangle | \langle([A]\ x : \langle\rangle),([\langle A,A\rangle]\ y : A^*)\rangle \\
& \quad | \ \langle([\langle A,A\rangle]\ x : \bot),([A]\ y : A^*)\rangle \\
& \quad | \ \langle([A]\ x : \bot),([\langle A,A\rangle]\ y : A^*)\rangle | \langle([\langle\rangle]\ x : \langle\rangle),([\langle A,A,A\rangle]\ y : A^*)\rangle \\
\longrightarrow & \{\{\langle A,A,A\rangle/x,\langle\rangle/y\},\{\langle A,A\rangle/x,A/y\},\{A/x,\langle A,A\rangle/y\},\{\langle\rangle/x,\langle A,A,A\rangle/y\}\}
\end{array}
$$

Steps (1) - (12) correspond to the *building phase.* During this phase, we build the pattern derivatives by pushing the input literals into the pattern. Step (13) corresponds to the *collection phase.* We collect *all* matching results by traversing the pattern derivative built in the previous step.

The execution trace above can be visualized in terms of a tree of all possible pattern matches as follows,



$$
\begin{array}{lcllcl}
L_1 & = & \langle[\langle A,A,A\rangle]x : A^*, y : A^*\rangle & L_2 & = & \langle[\langle A,A\rangle]x : \langle\rangle, [A]y : A^*\rangle \\
L_3 & = & \langle[\langle A,A\rangle]x : \bot, [A]y : A^*\rangle & L_4 & = & \langle[A]x : \langle\rangle, [\langle A,A\rangle]y : A^*\rangle \\
L_5 & = & \langle[\langle A,A\rangle]x : \bot, [A]y : A^*\rangle & L_6 & = & \langle[A]x : \bot, [\langle A,A\rangle]y : A^*\rangle \\
L_7 & = & \langle[A]x : \bot, [\langle A,A\rangle]y : A^*\rangle & L_8 & = & \langle x : \langle\rangle, [\langle A,A,A\rangle]y : A^*\rangle
\end{array}
$$

As we can observe from the tree above, every branch in the search tree corresponds to a choice operator in the pattern derivative. Every leaf node in the search tree denotes a match result (which can be a failure.) In this example, all leaf nodes denote successful match results.

We collect match results by visiting leaf nodes $\{L_1, ..., L_8\}$. Leaf nodes $\{L_3, L_5, L_6, L_7\}$ consist of $\perp$ and denote matching failures. $\{L_1, L_2, L_4, L_8\}$ denote successful matches. Therefore, we can conclude that the following are *all* possible results of matching $\langle A, A, A\rangle$ against the pattern $\langle (x : A^*), (y : A^*)\rangle$,

$$\begin{array}{ll} \{ & \{\langle A, A, A\rangle/x, \langle\rangle/y\} \\ , & \{\langle A, A\rangle/x, A/y\} \\ , & \{A/x, \langle A, A\rangle/y\} \\ , & \{\langle\rangle/x, \langle A, A, A\rangle/y\} \\ \} & \end{array}$$

$\square$

We extended the derivative operation to regular expression pattern. We solved the regular expression pattern matching problem by rewriting the regular expression pattern. The construction of pattern derivatives can be visualized as a parse tree. We compute *all* possible matchings from the parse tree by visiting the leaf nodes.

6.2.3 Formal Results

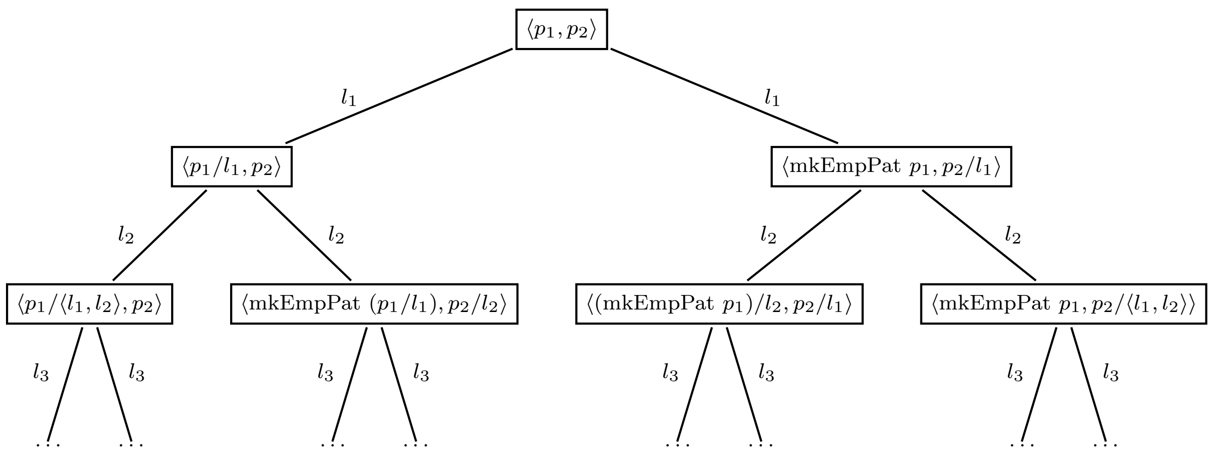
The termination of our pattern matching algorithm is always guaranteed.

Lemma 12 (All Match Termination) *Let w be a word and p be a pattern. Then allmatch w p always terminates.*

Lastly, we conclude that our allmatch w p algorithm is correct with respect to the pattern matching relation $w \lhd p \rightsquigarrow \theta$.

Lemma 13 (All Match Correctness) *Let w be a word and p be a pattern. Both of the following are valid.*

1. *Let $w \lhd p \rightsquigarrow \theta$. Then $\theta \in$ allmatch w p.*



$\langle p_1, p_2\rangle$
l_1
l_1
$\langle p_1/l_1, p_2\rangle$
$\langle \text{mkEmpPat } p_1, p_2/l_1\rangle$
l_2
l_2
l_2
l_2
$\langle p_1/\langle l_1, l_2\rangle, p_2\rangle$
$\langle \text{mkEmpPat } (p_1/l_1), p_2/l_2\rangle$
$\langle (\text{mkEmpPat } p_1)/l_2, p_2/l_1\rangle$
$\langle \text{mkEmpPat } p_1, p_2/\langle l_1, l_2\rangle\rangle$
l_3

Figure 6.4: A generic parse tree representing regular expression pattern matching

2. *Let $\theta \in$ allmatch w p. Then $w \lhd p \rightsquigarrow \theta$.*

The technical proofs of these lemmas can be found in the Appendix B, Section B.3.

6.2.4 Complexity Analysis and Optimization

We can always represent the pattern derivatives construction $\langle p_1, p_2\rangle/\langle l_1, ..., l_n\rangle$ in terms of a parse tree as described in Figure 6.4. The height of the parse tree is bounded by the length of the input $\langle l_1, ..., l_n\rangle$, that is n. There are in total 2^n leaf nodes. In the worst case, we need to vist all leaf nodes to collect the match result. In case that p_1 and p_2 are not simple variable patterns, the tree will grow exponentially.

Example 43 Consider the pattern $p = \langle p_1, p_2\rangle$ and the word $w = \langle l_1, l_2, l_3, l_4\rangle$. We will have $2^4 = 16$ leaf nodes by distributing the labels among p_1 and p_2, i.e.

$$
\begin{array}{ll}
\{ & \langle p_1/\langle l_1, l_2, l_3, l_4\rangle, p_2\rangle, \\
& \langle p_1/\langle l_1, l_2, l_3\rangle, p_2/l_4\rangle, \langle p_1/\langle l_1, l_2, l_4\rangle, p_2/l_3\rangle, \langle p_1/\langle l_1, l_3, l_4\rangle, p_2/l_2\rangle, \langle p_1/\langle l_2, l_3, l_4\rangle, p_2/l_1\rangle, \\
& \langle p_1/\langle l_1, l_2\rangle, p_2/\langle l_3, l_4\rangle\rangle, \langle p_1/\langle l_1, l_3\rangle, p_2/\langle l_2, l_4\rangle\rangle, \langle p_1/\langle l_1, l_4\rangle, p_2/\langle l_2, l_3\rangle\rangle, \\
& \langle p_1/\langle l_2, l_3\rangle, p_2/\langle l_1, l_4\rangle\rangle, \langle p_1/\langle l_2, l_4\rangle, p_2/\langle l_1, l_3\rangle\rangle, \langle p_1/\langle l_3, l_4\rangle, p_2/\langle l_1, l_2\rangle\rangle, \\
& \langle p_1/l_1, p_2/\langle l_2, l_3, l_4\rangle\rangle, \langle p_1/l_2, p_2/\langle l_1, l_3, l_4\rangle\rangle, \langle p_1/l_3, p_2/\langle l_1, l_2, l_4\rangle\rangle, \langle p_1/l_4, p_2/\langle l_1, l_2, l_3\rangle\rangle, \\
& \langle p_1, p_2/\langle l_1, l_2, l_3, l_4\rangle\rangle \quad \}
\end{array}
$$

In case that p_1 and p_2 are nested, say $p_1 = \langle p_3, p_4 \rangle$ and $p_2 = \langle p_5, p_6 \rangle$, we need to further distribute the labels among p_3, p_4, p_5 and p_6. Hence the total number of leaf nodes in the parse tree will be

$$\begin{aligned}
& 2^4 * 2^0 + 4 * 2^3 * 2^1 + 6 * 2^2 * 2^2 + 4 * 2^1 * 2^3 + 2^0 * 2^4 \\
= \; & \Sigma_{i=0}^{4}(C_4^i * 2^{4-i} * 2^i) \\
= \; & (2^4 * 2^4) \\
= \; & 2^{4*2}
\end{aligned}$$

□

Based on the above example, we can conclude that the time complexity of pattern matching algorithm is 2^{n*m} in the worst case, where n is the length of the input word and m is the maximum level of nesting pairs in the pattern.

There is certainly ample space for optimization here. For instance, among the 16 nodes that we listed in Example 43, there are many of them are *definitely failures*. For example, $\langle p_1/\langle l_1, l_2, l_4 \rangle, p_2/l_3 \rangle$ is a failing case, because, when we match $\langle l_1, l_2 \rangle$ with p_1 and l_3 with p_2, it is obviously impossible to match l_4 with $p_1/\langle l_1, l_2 \rangle$, because $p_1/\langle l_1, l_2 \rangle$ have been "made empty". We can prune the parse tree by tossing out these cases. We can also eliminate backtracking by employing similar techniques found in [23] and [39]. The application of these optimization techniques is beyond the scope of this thesis. We will pursue this topic in the near future.

6.2.5 Implementing the POSIX/Longest matching policy

We want to implement a specific pattern matching policy - the POSIX [56] matching policy. Under the POSIX matching policy, the sub pattern p_1 in $\langle p_1, p_2 \rangle$ will always consume the longest possible sequence from the input, while the remaining input is matched by p_2. Therefore, we also call this policy the POSIX/Longest matching policy.

$$\text{(LM-Seq)} \quad \frac{\begin{array}{c} w \sim \langle w_1, w_2 \rangle \quad w_1 \lhd_{\text{lm}} p_1 \rightsquigarrow \theta_1 \quad w_2 \lhd_{\text{lm}} p_2 \rightsquigarrow \theta_2 \\ \neg \left(\begin{array}{c} \exists w_3, w_4 : \neg(w_3 \sim \langle\rangle) \wedge \langle w_3, w_4 \rangle \sim w_2 \wedge \\ \langle w_1, w_3 \rangle \lhd_{\text{lm}} p_1 \rightsquigarrow \theta_3 \wedge w_4 \lhd_{\text{lm}} p_2 \rightsquigarrow \theta_4 \end{array} \right) \end{array}}{w \lhd_{\text{lm}} \langle p_1, p_2 \rangle \rightsquigarrow \theta_1 \cup \theta_2}$$

$$\text{(LM-Choice1)} \quad \frac{w \lhd_{\text{lm}} p_1 \rightsquigarrow \theta_1}{w \lhd_{\text{lm}} (p_1|p_2) \rightsquigarrow \theta_1}$$

$$\text{(LM-Choice2)} \quad \frac{\neg(w \lhd_{\text{lm}} p_1 \rightsquigarrow \theta_1) \quad w \lhd_{\text{lm}} p_2 \rightsquigarrow \theta_2}{w \lhd_{\text{lm}} (p_1|p_2) \rightsquigarrow \theta_2}$$

$$\text{(Var)} \quad \frac{w \in L(t)}{w \lhd ([w']\ x : t) \rightsquigarrow \{\langle w', w \rangle / x\}}$$

Figure 6.5: Pattern Matching Relation in POSIX/Longest matching policy

In Figure 6.5, we define the regular expression pattern matching relation for the POSIX/Longest matching policy. Rule (LM-Seq) enforces that the sub pattern p_1 will be matched with the longest possible prefix. Rules (LM-Choice1) and (LM-Choice2) restrict that a choice pattern $(p_1|p_2)$ must be matched from left to right. The rule for variable pattern remains unchanged.

Example 44 We recall from the earlier example that matching $\langle A, B \rangle$ against $\langle x : A^*, y : (\langle A, B \rangle^* | B) \rangle$ yields two possible substitutions. Let's fix the longest matching policy, the matching derivation is as follows,

$$\frac{\langle A, B \rangle \sim \langle A, B \rangle \quad \dfrac{A \in L(A^*)}{A \lhd_{\text{lm}} x : A^* \rightsquigarrow \{A/x\}} \quad \dfrac{B \in L(\langle A, B \rangle^* | B)}{B \lhd_{\text{lm}} y : (\langle A, B \rangle^* | B) \rightsquigarrow \{B/y\}}}{\langle A, B \rangle \lhd_{\text{lm}} \langle x : A^*, y : (\langle A, B \rangle^* | B) \rangle \rightsquigarrow \{A/x, B/y\}} \tag{6.9}$$

In the above derivation, we break the input sequence $\langle A, B \rangle$ into A and B to match with the sub-patterns. No other breaking is allowed. For instance, we cannot break the input sequence into $\langle\rangle$ and $\langle A, B \rangle$, because it does not satisfy the longest match-

```
longmatch $w\ p$          =  let $p' = \text{build}\ w\ p$ in collect $p'$

collect $([w]\ x : t)$    =  if isEmpty $t$
                             then $Just\ \{(w/x)\}$
                             else $Nothing$
collect $(p_1|p_2)$       =  case collect $p_1$ of
                             $Nothing \rightarrow \text{collect}\ p_2$
                             $Just\ \theta \rightarrow Just\ \theta$
collect $\langle p_1, p_2\rangle$  =  case (collect $p_1$) of
                             $Nothing \rightarrow Nothing$
                             $Just\ \theta_1 \rightarrow$ case (collect $p_2$) of
                                  $Nothing \rightarrow Nothing$
                                  $Just\ \theta_2 \rightarrow Just\ (\theta_1 \cup \theta_2)$
```

Figure 6.6: The POSIX/longest pattern matching algorithm

ing condition,

$$\neg \left(\begin{array}{l} \exists w_3, w_4 : \neg(w_3 \sim \langle\rangle) \wedge \langle w_3, w_4\rangle \sim \langle A, B\rangle \wedge \\ \langle \langle\rangle, w_3\rangle \lhd_{\text{lm}} x : A^* \rightsquigarrow \theta_3 \wedge w_4 \lhd_{\text{lm}} y : A^* \rightsquigarrow \theta_4 \end{array} \right)$$

which can be disproved with $w_3 = A$ and $w_4 = B$. □

We can derive a POSIX/Longest matching algorithm from the allmatch $w\ p$ algorithm as follows. In Figure 6.6, we define the POSIX/Longest matching algorithm longmatch $w\ p$. longmatch $w\ p$ is a variant of allmatch $w\ p$. The key difference is that instead of collecting all possible matching of matching w against p, longmatch $w\ p$ collects the first successful matching. It performs a depth-first left to right search across the tree of all possible matches, and stops when the first successful leaf node is found or there is no valid match. We use `Maybe` data type to represent successful match and match failure, e.g. $Just\ \theta$ denotes a match is found, $Nothing$ denotes a match failure.

We can straightforwardly verify that the POSIX/longest matching algorithm is terminating and correct.

Lemma 14 (POSIX/Longest Match Termination) *Let v be a value and p be a pattern, Then longmatch w p always terminates.*

Lemma 15 (POSIX/Longest Match Correctness) *Let p be a pattern and w be a value, longmatch w p $\longrightarrow^{*}$ θ iff $w \lhd_{lm} p \rightsquigarrow \theta$.*

The technical proofs of these lemmas can be found in the Appendix B, Section B.3. Complete, runnable Haskell implementations of both algorithms are given in Appendix A.2.

In this section, we developed two algorithms for regular expression pattern matching based on rewriting. The algorithm allmatch w p implements the "all match" semantics, whilst on the other hand, the algorithm longmatch w p implements the POSIX/Longest matching policy.

6.3 Coercive Pattern Matching

In this section, we consider a more specific problem setting where in addition the set of possible input words are described by a regular expression r. Given the regular expression pattern p, our goal is to derive a coercion function which executes the pattern match for the specified set of input words. Our novel idea is to derive this coercion from Antimirov's regular expression containment algorithm by applying the proofs-are-programs principle. The following section reviews Antimirov's algorithm and highlights the challenges we face to extract the coercion function.

6.3.1 From Regular Expression Containment to Coercive Pattern Matching

We consider the classic regular expression containment problem, $t_1 \leq t_2$, which is to check that all words described by the language $L(t_1)$ can be found in the language

$$\boxed{C \vdash_{\text{sub}} t \leq t'}$$

$$\text{(Hyp)}\ \frac{t \leq t' \in C}{C \vdash_{\text{sub}} t \leq t'} \qquad \text{(Norm)}\ \frac{\begin{array}{c} \langle\rangle \in t \text{ implies } \langle\rangle \in t' \quad \Sigma(t) \cup \Sigma(t') = \{l_1, ..., l_n\} \\ C \cup \{t \leq t'\} \vdash_{\text{sub}} d(l_1 \shortmid t) \leq d(l_1 \shortmid t') \\ \dots \\ C \cup \{t \leq t'\} \vdash_{\text{sub}} d(l_n \shortmid t) \leq d(l_n \shortmid t') \end{array}}{C \vdash_{\text{sub}} t \leq t'}$$

$$\boxed{\langle\rangle \in t}$$

$$\langle\rangle \in \langle\rangle \qquad \frac{\langle\rangle \in t_1 \quad \langle\rangle \in t_2}{\langle\rangle \in \langle t_1, t_2 \rangle}$$

$$\langle\rangle \in t^* \qquad \frac{\langle\rangle \in t_i \quad i \in \{1,2\}}{\langle\rangle \in (t_1|t_2)}$$

$$\boxed{\Sigma(t)}$$

$$\Sigma(l) = \{l\} \qquad \Sigma(t^*) = \Sigma(t) \qquad \Sigma(\langle\rangle) = \{\}$$

$$\Sigma(\langle t_1, t_2 \rangle) = \Sigma(t_1) \cup \Sigma(t_2)$$

$$\Sigma(t_1|t_2) = \Sigma(t_1) \cup \Sigma(t_2)$$

$$\boxed{pd(l \shortmid t)}$$

$$pd(l \shortmid \langle\rangle) = \{\}$$

$$pd(l \shortmid t^*) = pd(l \shortmid t) \odot t^* \qquad pd(l_1 \shortmid l_2) = \begin{cases} \{\langle\rangle\} & ; l_1 = l_2 \\ \{\} & ; \mathit{otherwise} \end{cases}$$

$$pd(l \shortmid (t_1|t_2)) = pd(l \shortmid t_1) \cup pd(l \shortmid t_2)$$

$$pd(l \shortmid \langle t_1, t_2 \rangle) = \begin{cases} pd(l \shortmid t_1) \odot t_2 \cup pd(l \shortmid t_2) & ; \vdash_{\text{empty}} \langle\rangle \in t_1 \\ pd(l \shortmid t_1) \odot t_2 & ; \mathit{otherwise} \end{cases}$$

$$\boxed{d(l \shortmid t)} \qquad \boxed{\{t_1, ..., t_n\} \odot t}$$

$$d(l \shortmid t) = t_1|...|t_n \qquad \{\} \odot t = \{\} \qquad \{\langle\rangle\} \odot t = \{t\}$$

$$\text{where } pd(l \shortmid t) = \{t_1, ..., t_n\} \text{ and } n > 0 \qquad \{t_1, ..., t_n\} \odot t = \{\langle (t_1|...|t_n), t \rangle\} \text{ where } n > 0$$

Figure 6.7: Antimirov's Containment Algorithm

$L(t_2)$, in other words, to check $L(t_1) \subseteq L(t_2)$. We sometimes refer to the regular expression containment problem as the regular expression subtyping problem.

In [4], Antimirov developed an algorithm in terms of a co-inductive term-rewriting system. In Figure 6.7 we describe his algorithm in terms of the judgment $C \vdash_{\text{sub}} t_1 \leq t_2$. Like most of the subtyping algorithms, such as [10], the algorithm can be elegantly formalized using co-induction by recording previously processed inequalities in the context C. When an inequality $t \leq t'$ is encountered again, we conclude the proof by applying the co-inductive hypothesis via rule (Hyp). Otherwise, we apply rule (Norm) to reduce the subtype inequality. In the premise of (Norm), we first need to enforce that if t contains the empty sequence, t' must also contain the empty sequence. For each label l in $\Sigma(t) \cup \Sigma(t')$, we compute the set of partial derivatives of t and t' with respect to l, say $pd(l \shortmid t)$ and $pd(l \shortmid t')$. We union the partial derivatives in the sets to compute the *canonical forms*, $d(l \shortmid t)$ and $d(l \shortmid t')$. Finally the proof proceeds as we verify the new sub-goals $C \cup \{t \leq t'\} \vdash_{\text{sub}} d(l \shortmid t) \leq d(l \shortmid t')$. Definitions of $pd(l \shortmid t)$ and $d(l \shortmid t)$ are given in Figure 6.7.

To have a better understanding of the algorithm, let us consider an example.

Example 45 We consider the regular expression containment problem $A^* \leq (A|B)^*$. To motivate the need for partial derivatives, let us try to solve this containment problem using a "naive" algorithm which is based on on derivative rewriting.

$$\begin{array}{l} t_1 \leq t_2 \text{ iff} \\ \quad (1) \quad \langle\rangle \in L(t_1) \text{ implies } \langle\rangle \in L(t_2) \text{ and} \\ \quad (2) \quad \forall l.t_1/l \leq t_2/l \end{array}$$

The above algorithm says that t_1 is a subtype of t_2 iff both of the following are valid. First, if we find $\langle\rangle$ in t_1, then we can also find it in t_2; second, their derivatives t_1/l and t_2/l are in subtype relation. The flaw of this "naive" algorithm is that it may not terminate.

Suppose we recursively apply the derivative operation to both sides of the inequality $A^* \leq (A|B)^*$, we have

$$A^* \leq (A|B)^* \longrightarrow \langle\langle\rangle, A^*\rangle \leq \langle(\langle\rangle|\bot), (A|B)^*\rangle \longrightarrow \ldots$$

which has infinitely many steps. This is because the derivative operation t/l that we defined in Figure 6.2 is producing infinitely many new regular expressions.

$$\begin{array}{l} A^*/A = \langle\langle\rangle, A^*\rangle \\ \langle\langle\rangle, A^*\rangle/A = (\langle\bot, A^*\rangle|\langle\langle\rangle, \langle\langle\rangle, A^*\rangle\rangle) \\ \ldots \end{array}$$

and

$$\begin{array}{l}(A|B)^*/A = \langle(\langle\rangle|\bot),(A|B)^*\rangle \\ \langle(\langle\rangle|\bot),(A|B)^*\rangle/A = (\langle(\bot|\bot),(A|B)^*\rangle|\langle(\langle\rangle|\bot),\langle(\langle\rangle|\bot),(A|B)^*\rangle\rangle) \\ \ldots\end{array}$$

The crucial property is that, for each regular expression the set of derivatives is infinite, but the set of partial derivatives is bound. This is the main result in [4]. (We can think of partial derivatives as connected to derivatives like NFAs are connected to DFAs.) Hence, Antimirov employs partial derivatives in his regular expression containment algorithm.

Back to the running example, we apply the partial derivative operation to $A^* \leq (A|B)^*$, we have

$$pd(A \shortmid A^*) = \{A^*\} \quad d(A \shortmid A^*) = A^*$$

and

$$pd(A \shortmid (A|B)^*) = \{(A|B)^*\} \quad d(A \shortmid (A|B)^*) = (A|B)^*$$

Therefore, the correct proof derivation is as follows,

$$\cfrac{\cfrac{A^* \leq (A|B)^* \in \{A^* \leq (A|B)^*\}}{\{A^* \leq (A|B)^*\} \vdash_{\text{sub}} A^* \leq (A|B)^*}\text{(Hyp)}}{\{\} \vdash_{\text{sub}} A^* \leq (A|B)^*}\text{(Norm)}$$

□

At this point, we have a constructive regular expression subtyping algorithm. Let us apply the proofs-are-programs principle to this algorithm to derive the downcast coercion. There are some challenges faced by us.

The main challenge is how to compose the coercions among the partial derivatives to obtain a coercion among the top-level types. The issue is that parts of the original structures are lost when applying the partial derivative set operation. Thus gluing the pieces together is non-trivial. We illustrate the point using the following example.

Example 46 For instance, we consider $\langle A,B\rangle \leq_d (\langle A,B\rangle|\langle A,C\rangle)$. Our goal is to derive a downcast coercion $d : [\![(\langle A,B\rangle|\langle A,C\rangle)]\!] \rightarrow \mathit{Maybe}\ [\![\langle A,B\rangle]\!]$. Applying Antimirov's algorithm to the problem, we have a proof derivation as follows,

$$\dfrac{\dfrac{\cdots}{\{\langle A,B\rangle \leq (\langle A,B\rangle|\langle A,C\rangle)\} \vdash_{\text{sub}} d(A \shortmid \langle A,B\rangle) \leq_{d'} d(A \shortmid (\langle A,B\rangle|\langle A,C\rangle))}\text{(Norm)}}{\{\} \vdash_{\text{sub}} \langle A,B\rangle \leq_d (\langle A,B\rangle|\langle A,C\rangle)}\text{(Norm)}$$

In the above derivation, we verify the containment problem $\langle A,B\rangle \leq_d (\langle A,B\rangle|\langle A,C\rangle)$ by applying partial derivative operation to both sides. The goal is therefore reduced to a sub-goal $\{\langle A,B\rangle \leq (\langle A,B\rangle|\langle A,C\rangle)\} \vdash_{\text{sub}} d(A \shortmid \langle A,B\rangle) \leq_{d'} d(A \shortmid (\langle A,B\rangle|\langle A,C\rangle))$.

The idea is to make use of $d' : [\![d(A \shortmid (\langle A,B\rangle|\langle A,C\rangle))]\!] \rightarrow \mathit{Maybe}\ [\![d(A \shortmid \langle A,B\rangle)]\!]$ to define d. The question is how to make use of d'? Note that the input type of d, $(\langle A,B\rangle|\langle A,C\rangle)$, is connected to d''s input type via the partial derivative operation $pd(l \shortmid t)$ and $d(l \shortmid t)$.

Let `f` be a function that coerces values from $[\![(\langle A,B\rangle|\langle A,C\rangle)]\!]$ to $[\![(A, d(A \shortmid (\langle A,B\rangle|\langle A,C\rangle))]\!]$, we can make use of d' to define d as follows,

```
d :: (Or (A,B) (A,C)) -> Maybe (A,B)
d v = let (A,v') = f v
      in case d' v' of
         Just v'' -> Just (A,v'')
         Nothing  -> Nothing
```

For simplicity, we omit the definition of `d'`. The problem is that it is hard to fix the type of the function `f`. We recall the partial derivative operation as follows,

$$pd(A \shortmid (\langle A,B\rangle|\langle A,C\rangle)) = pd(A \shortmid (A,B)) \cup pd(A \shortmid (A,C)) = \{B\} \cup \{C\} = \{B,C\}$$

$$\text{(E1) } (\bot|t) = t \qquad \text{(E2) } (t|\bot) = t \qquad \text{(E3) } (t|t) = t \qquad \text{(E4) } ((t_1|t_2)|t_3) = (t_1|(t_2|t_3))$$

$$\text{(E5) } \langle\langle\rangle, t\rangle = t \qquad \text{(E6) } \langle\bot, t\rangle = \bot$$

Figure 6.8: Simplification Rules

Note that elements in a set are un-ordered. Therefore, $d(A \shortmid (\langle A, B\rangle|\langle A, C\rangle))$ can be either $(B|C)$ or $(C|B)$, which implies the type of `f` can be `(Or (A,B) (A,C)) -> Maybe (A, (Or B C))` or `(Or (A,B) (A,c)) -> Maybe (A, (Or C B))`. □

The problem is caused by the set operations used in the partial derivative operation. The purpose of using set operations is to remove unreachable states $\bot$ and duplicate states $(t|t)$ in the partial derivatives, therefore the resulting canonical form $d(l \shortmid t)$ is always minimized. To attack this problem, we replace the set operations by employing a set of simplification rules. We apply these simplification rules to the derivative and obtain the canonical form.

The set of simplification rules are described in Figure 6.8, The simplification rules can be obtained by orienting these equations from left to right. Rules (E1) and (E2) remove the $\bot$ type in a choice. Rule (E3) collapses a choice type whose left and right alternatives are identical. Rule (E4) applies the associativity law for choice operator. Rule (E5) removes the leading empty sequence. Rule (E6) simplifies a pair type whose first component is $\bot$ to $\bot$. We may argue why some other valid equations such as $\langle t, \langle\rangle\rangle = t$ and $\langle t, \bot\rangle = \bot$ are not included. This is because in the derivative operation, $\bot$ and $\langle\rangle$ are only introduced in the first position of a sequence type. Therefore, we do not need these two extra rules in the set of simplification rules.

Definition 5 *Let $|t|$ denote the regular expression obtained by applying simplification rules, (E1) to (E6), to t exhaustively.*

Lemma 16 *Let $pd(l \shortmid t) = \{t_1, ..., t_n\}$. Then $|t/l| = (t_1|...|t_n)$.*

$$(\text{Norm})\quad \frac{\begin{array}{c}\langle\rangle \in t \text{ implies } \langle\rangle \in t' \\ C \cup \{t \leq t'\} \vdash_{\text{sub}} |(t/l_1)| \leq |(t'/l_1)| \\ \dots \\ C \cup \{t \leq t'\} \vdash_{\text{sub}} |(t/l_n)| \leq |(t'/l_n)| \\ \Sigma(t') \cup \Sigma(t) = \{l_1, ..., l_n\}\end{array}}{C \vdash_{\text{sub}} t \leq t'}$$

Figure 6.9: Using simplification rules in the refined (Normal) rule

The proof of this lemma can be found in Appendix B.3.

We can replace the $d(l \shortmid t)$ operation in the containment algorithm with $|t/l|$. The refined (Norm) rule is given in Figure 6.9.

Let us use the simplification rules to solve the problem which we encountered earlier in Example 46.

Example 47 Recall that in Example 46 we want to derive a coercion function `f` from the partial derivative operation $d(A \shortmid (\langle A, B\rangle | \langle A, C\rangle))$. We were unable to fix the type of `f` due to the set operations used in the computation of $d(A \shortmid (\langle A, B\rangle | \langle A, c\rangle))$.

Thanks to the result of Lemma 16, we can compute the canonical forms by first computing the derivative, then applying the simplification rules to the derivative.

$$(\langle A, B\rangle | \langle A, C\rangle)/A = (B|C)$$
$$|(B|C)| = (B|C)$$

Therefore, we can fix the type of `f` as `(Or (A,B) (A,C)) -> (A, (Or B C))`. □

There is another minor problem we yet need to address. The regular expression containment proof requires co-induction. To derive the corresponding downcast coercion, we build a recursive function whenever we apply co-inductive hypothesis in the proof. The following example demonstrates this idea.

Example 48 We note that proving $\vdash_{\text{sub}} A^* \leq A^*$ requires co-induction. The proof derivation is as follows,

$$\dfrac{\dfrac{A^* \leq A^* \in \{A^* \leq_d A^*\}}{\{A^* \leq_d A^*\} \vdash_{\text{sub}} A^* \leq_{d'} A^*}(\text{Hyp}) \quad A^*/A = \langle\langle\rangle, A^*\rangle \quad |\langle\langle\rangle, A^*\rangle| = A^*}{\{\} \vdash_{\text{sub}} A^* \leq_d A^*}\ (\text{Norm})$$

In the above derivation, we reduce the original goal by rewriting the regular expressions into their canonical forms. We then discover that the sub-goal we obtain is in the context. Thus, we apply co-inductive hypothesis to conclude the proof.

When we derive the downcast coercion from the above derivation, we need to make use of the sub downcast coercion d' to define the main downcast coercion d. Since we conclude the sub-goal by co-induction, we simply build a recursion by letting d' equal to d.

```
d :: [A] -> Maybe [A]
d v = case v of
    [] -> Just []
    (l:rest) ->
         case d rest of
         Just rest' -> Just (l:rest')
         Nothing -> Nothing
```

For simplicity, we "compile away" the coercion functions that convert values back and forth between their original forms and the respective canonical forms by using a case expression. In the top level of `d`, we check the input value for emptiness. In the case that v is empty, we need to produce an empty value for the output type, which is `[]`, too. In case that `v` is not empty, we have to rewrite it to the canonical form. Since `v` is a list of As, this can be done easily via a list pattern `(l:rest)`. Now we process the remainder `rest`. We want to make use of the sub-proof $\vdash_{\text{sub}} |A^*/A| \leq_{d'} |A^*/A|$. As we pointed out, we verified this sub-goal by applying co-induction. We use $A^* \leq_d A^*$ as the co-induction hypothesis, we define `d' = d`. As a result, applying `d` to `rest` yields `rest'`. From `l` and `rest'`, we construct the final result `(l:rest')`, which is of type `[A]`. □

Now we are ready to derive the coercion function from the containment proof. Before going into the details of the coercion function, we want to highlight that there are still some ambiguity issues which are connected to the kind of pattern matching policy we employ. For instance, there are multiple ways of defining the downcast function that coerces values from A to $(A|A)$. For instance, we can define

```
d :: A -> Maybe (Or A A)
d v = Just (L v)
```

or

```
d v = Just (R v)
```

We will give a detailed discussion on this issue shortly.

6.3.2 Deriving Downcast Coercion

Now let us go into the details of the downcast coercion. The challenge is to derive the appropriate downcast coercion which performs the actual pattern matching. We apply the proofs-are-programs principle (Curry-Howard Isomorphism) to the containment proof. Each rewriting step in the proof derivation will yield a pair of coercion functions. Recall the problem that the structure of the regular expressions changes when we rewrite them into to the canonical forms. The key insight here is to maintain a correspondence between derivatives and the canonical forms via two helper functions `to` and `from`, which can be derived from the simplification rules.

Applying Curry-Howard Isomorphism to Antimirov's algorithm, we turn the algorithm in Figure 6.7 into a coercive pattern matching algorithm. In particular we

$$\frac{\begin{array}{c}
C \cup \{t \leq_d t'\} \vdash_{\text{sub}} |(t/l_1)| \leq_{d_1} |(t'/l_1)| \\
\ldots \\
C \cup \{t \leq_d t'\} \vdash_{\text{sub}} |(t/l_n)| \leq_{d_n} |(t'/l_n)| \\
\Sigma(t') = \{l_1, ..., l_n\} \\
\begin{array}{lcl}
\begin{array}{l}
\mathit{dd_1} : \forall[\![l_1]\!] \to [\![t'/l_1]\!] \to \mathit{Maybe}\ [\![t]\!] \\
\mathit{dd_1}\ l_1\ v_1' = \mathit{case}\ d_1\ (\mathit{to_1}\ v_1')\ \mathit{of} \\
\quad \mathit{Just}\ x \to \mathit{Just}\ \mathit{inj}_{(l_1,t)}\ l_1\ (\mathit{from_1}\ x) \\
\quad \mathit{Nothing} \to \mathit{Nothing}
\end{array}
& \ldots &
\begin{array}{l}
\mathit{dd_n} : \forall[\![l_n]\!] \to [\![t'/l_n]\!] \to \mathit{Maybe}\ [\![t]\!] \\
\mathit{dd_n}\ l_n\ v_n' = \mathit{case}\ d_n\ (\mathit{to_n}\ v_n')\ \mathit{of} \\
\quad \mathit{Just}\ x \to \mathit{Just}\ \mathit{inj}_{(l_n,t)}\ l_n\ (\mathit{from_n}\ x) \\
\quad \mathit{Nothing} \to \mathit{Nothing}
\end{array}
\end{array} \\
\begin{array}{l}
d : \forall[\![t']\!] \to \mathit{Maybe}\ [\![t]\!] \\
d\ v = \mathit{if}\ \mathit{isEmpty}_{t'}\ v\ \mathit{then}\ \mathit{Just}\ \mathit{mkEmpty}_t \\
\qquad\qquad \mathit{else}\ \mathit{select}_{(l_1,...,l_n,t')}\ v\ \mathit{dd_1}...\mathit{dd_n}
\end{array}
\end{array}}{C \vdash_{\text{sub}} t \leq_d t'}$$

Helper functions

$$\begin{array}{l}
\mathit{proj}_{(l,t'')} : \forall[\![t'']\!] \to \mathit{Maybe}\ ([\![l]\!], [\![t''/l]\!]) \\
\mathit{inj}_{(l,t'')} : \forall[\![l]\!] \to [\![t''/l]\!] \to [\![t'']\!] \\
\mathit{isEmpty}_{t''} : \forall[\![t'']\!] \to \mathit{Bool} \\
\mathit{mkEmpty}_{t''} : \forall[\![t'']\!] \\
to_1 : \forall[\![(t'/l_1)]\!] \to [\![|(t'/l_1)|]\!] \\
\ldots \\
to_n : \forall[\![(t'/l_n)]\!] \to [\![|(t'/l_n)|]\!] \\
\\
from_1 : \forall[\![|(t/l_1)|]\!] \to [\![(t/l_1)]\!] \\
\ldots \\
from_n : \forall[\![|(t/l_n)|]\!] \to [\![(t/l_n)]\!]
\end{array}$$

$$\begin{array}{l}
\mathit{select}_{(l_1,...,l_n,t'')} : \forall[\![t'']\!] \to ([\![l_1]\!] \to [\![t''/l_1]\!] \to a) \\
\quad \to ... \to ([\![l_n]\!] \to [\![t''/l_n]\!] \to a) \to a \\
\mathit{select}_{(l_1,...,l_n,t'')}\ v\ e_1...e_n = \\
\qquad \mathit{let}\ v_1 = \mathit{proj}_{(l_1,t'')}\ v \\
\qquad\quad \ldots \\
\qquad\quad v_n = \mathit{proj}_{(l_n,t'')}\ v \\
\qquad \mathit{in}\ \mathit{case}\ (v_1, ..., v_n)\ \mathit{of} \\
\qquad\quad (\mathit{Just}\ (l_1, v_1'), ...) \to e_1\ l_1\ v_1' \\
\qquad\quad \ldots \\
\qquad\quad (...., \mathit{Just}\ (l_n, v_n')) \to e_n\ l_n\ v_n'
\end{array}$$

Figure 6.10: Deriving downcast from subtype proofs

are only interested in the (Norm) rule, which is refined in Figure 6.9. The resulting pattern matching algorithm is given in Figure 6.10.

The construction of the downcast coercions is driven by value rewriting. In the general setting, the downcast coercion d consists of three main steps,

- We rewrite the input value from type t' into some intermediate value that is correspondent to the derivative t'/l. Then we further rewrite it to the canonical form $|t'/l|$;

- After rewriting the input into canonical form, we proceed by applying the sub coercion, another downcast coercion derived from the sub proof $\vdash_{\text{sub}} |t/l| \leq |t'/l|$, to the intermediate value. The result of this intermediate application must be of type $|t/l|$.

- Finally we need to rewrite this result from the canonical form $|t/l|$ back to derivative t/l and to the type t.

The above has outlined the main procedures that are applied in the downcast coercion.

Let us get into the details of the downcast coercion as stated in Figure 6.10:

1. d has type $[\![t']\!] \rightarrow Maybe\ [\![t]\!]$, where $[\![t]\!]$ refers to the target representation of t. We use $Maybe$ type in the output, in order to catch downcast failures.

2. At the entry point of d, we verify whether the input value v denotes an empty sequence. This is accomplished by applying the helper function $isEmpty_{t'}$ to v. We skip the definitions of all the helper functions at the moment and will come back to them shortly. If v is empty, we create an empty value of type $[\![t]\!]$, by using $mkEmpty_t$.

3. Otherwise, we need to rewrite v into its derivative type. To do that, we need to find out what the leading label of v is. Knowing that the set of possible labels appearing in v is $\{l_1, ..., l_n\}$, we iterate through the set of labels and look for the particular label l_i that we can extract from v. This task is carried out within the helper function $select_{(l_1,..,l_n,t')}$. In this function, we apply a set of projection functions $proj_{(l_1,t')}, ..., proj_{(l_n,t')}$ to v one by one. Each projection function $proj_{(l_i,t')}$ tries to extract the label l_i from the input value, which may fail. Therefore, $proj_{(l_i,t')}$'s output has a $Maybe$ type. We will eventually find one successful projection.

4. Once we find a successful projection, let's say $proj_{(l_i,t')}$, we rewrite the input

value into the derivative form (l_i, v'_i). At this point, we want to apply the sub-downcast d_i to the remaining v'_i as planned. Note that v'_i is of type $[\![t'/l_i]\!]$. We have to further rewrite it to the canonical form $[\![|t'/l_i|]\!]$ so that its structure as well as its type matches with the input type of d_i. This rewriting step is carried out by another helper function to_i. When v'_i is rewritten, we apply the sub downcast d_i to v'_i.

5. After we successfully downcast v'_i to v_i, we expect that v_i is of the canonical form (type) $[\![|t/l_i|]\!]$. We need to rewrite v_i to type $[\![t/l_i]\!]$, so that we can rewrite the result back to $[\![t]\!]$ by "inserting" l_i into the result. There are two more operations happening here. The $from_i$ function rewrites the value from its canonical form to the (unabridged) derivative form. The injection function $inj_{(l_i,t)}$ injects the label l_i back to the value, so that the result is of type $[\![t]\!]$. This completes the execution of a successful downcast application $(d\ v)$.

Now let us go through the helper functions in details.

1. Function $isEmpty_t$ and constructor $mkEmpty_t$

   In Figure 6.11, we give a generic description of helper functions $isEmpty_t$ and $mkEmpty_t$. Note that they are both type indexed. They have different definitions given different type parameters. The definitions are driven by the empty test judgment $\vdash_{\text{empty}} \langle\rangle \in t$ which is defined in Figure 4.4b (Section 4.3 Chapter 4).

   Function $isEmpty_t$ tests whether a target value of type $[\![t]\!]$ is equivalent to the source value $\langle\rangle$. The first pattern clause applies if the type parameter is $\langle\rangle$. We immediately return $True$ since the input value must be $()$ and $\langle\rangle \overset{\langle\rangle}{\approx} ()$. (This is required by Definition 3 in Chapter 5.) The second clause applies if the input type is t^*. The second pattern clause has two pattern guards. The first guard is valid if we can't find the empty word $\langle\rangle$ in t. In the body, we examine the structure of the input value v. In the case where v is an empty

```
$\mathbf{isEmpty_t} : \forall[\![\mathbf{t}]\!] \to \mathbf{Bool}$
 $isEmpty_{\langle\rangle}\ () = True$
 $isEmpty_{t^*}\ v$
     $|\neg(\vdash_{\text{empty}} \langle\rangle \in t) = \textsf{case}\ v\ \textsf{of}\ [] \to True$
                                          $(x : xs) \to False$
     $|\vdash_{\text{empty}} \langle\rangle \in t = \textsf{case}\ v\ \textsf{of}\ [] \to True$
                                   $(x : xs) \to (isEmpty_t\ x)\&\&(isEmpty_{t^*}\ xs)$
 $isEmpty_{(t_1|t_2)}\ v$
     $|\vdash_{\text{empty}} \langle\rangle \in t_1 \wedge \vdash_{\text{empty}} \langle\rangle \in t_2 = \textsf{case}\ v\ \textsf{of}$
                                                  $L\ v_1 \to isEmpty_{t_1}\ v_1$
                                                  $R\ v_2 \to isEmpty_{t_2}\ v_2$
     $|\vdash_{\text{empty}} \langle\rangle \in t_1 \wedge \neg(\vdash_{\text{empty}} \langle\rangle \in t_2) = \textsf{case}\ v\ \textsf{of}$
                                                  $L\ v_1 \to isEmpty_{t_1}\ v_1$
                                                  $R\ v_2 \to False$
     $|\vdash_{\text{empty}} \langle\rangle \in t_2 \wedge \neg(\vdash_{\text{empty}} \langle\rangle \in t_1) = \textsf{case}\ v\ \textsf{of}$
                                                  $R\ v_2 \to isEmpty_{t_2}\ v_2$
                                                  $L\ v_1 \to False$
 $isEmpty_{\langle t_1,t_2\rangle}\ (v_1, v_2) = (isEmpty_{t_1}\ v_1)\&\&(isEmpty_{t_2}\ v_2)$

 $\mathbf{mkEmpty_t} : \forall[\![\mathbf{t}]\!]$
  $mkEmpty_{\langle\rangle} = ()$
  $mkEmpty_{t^*} = []$
  $mkEmpty_{(t_1|t_2)}\ |\vdash_{\text{empty}} \langle\rangle \in t_1 \wedge \vdash_{\text{empty}} \langle\rangle \in t_2 = L\ mkEmpty_{t_1}$
                 $|\vdash_{\text{empty}} \langle\rangle \in t_1 \wedge \neg(\vdash_{\text{empty}} \langle\rangle \in t_2) = L\ mkEmpty_{t_1}$
                 $|\vdash_{\text{empty}} \langle\rangle \in t_2 \wedge \neg(\vdash_{\text{empty}} \langle\rangle \in t_1) = R\ mkEmpty_{t_2}$
  $mkEmpty_{\langle t_1,t_2\rangle} = (mkEmpty_{t_1}, mkEmpty_{t_2})$
```

Figure 6.11: An implementation of $isEmpty_t$ and $mkEmpty_t$ functions

list, we return $True$, otherwise, we can immediately return $False$. The second pattern guard is valid when $\vdash_{\text{empty}} \langle\rangle \in t$. In the body, we return $True$ if the input is an empty list, otherwise we have to check whether all elements in the list are empty. We simply apply $isEmpty_t$ to every element in the list. The rest of the definitions contain no surprise.

Function $mkEmpty_t$ creates an empty value of type $[\![t]\!]$. The most interesting case is when we need to build an empty value of type $[\![(t_1|t_2)]\!]$. In the case where both t_1 and t_2 contain the empty sequence $\langle\rangle$, we have two ways of defining $mkEmpty_{(t_1|t_2)}$. We can either make an empty value using (1) $mkEmpty_{t_1}$ (the

```
$\mathbf{proj_{(l,t)}} : \forall[\![\mathbf{t}]\!] \rightarrow \mathbf{Maybe}\ ([\![\mathbf{l}]\!], [\![\mathbf{t/l}]\!])$
 $proj_{(l,\langle\rangle)}\ v = Nothing$
 $proj_{(l_1,l_2)}\ v \mid l_1 \neq l_2 = Nothing$
                   $\mid$ otherwise $= Just\ (v, ())$
 $proj_{(l,t^*)}\ v =$ case $v$ of $[] \rightarrow Nothing$
                                  $(x : xs) \rightarrow$ case $proj_{(l,t)}\ x$ of
                                    $Just\ (v_l, v') \rightarrow Just\ (v_l, (v', xs))$
                                    $Nothing \rightarrow Nothing$
 $proj_{(l,(t_1|t_2))}\ v =$ case $v$ of
     $L\ v_1 \rightarrow$ case $proj_{(l,t_1)}\ v_1$ of
         $Just\ (v_l, v'_1) \rightarrow Just\ (v_l, L\ v'_1)$
         $Nothing \rightarrow Nothing$
     $R\ v_2 \rightarrow$ case $proj_{(l,t_2)}\ v_2$ of
         $Just\ (v_l, v'_2) \rightarrow Just\ (v_l, R\ v'_2)$
         $Nothing \rightarrow Nothing$
 $proj_{(l,\langle t_1,t_2\rangle)}\ (v_1, v_2)$
     $\mid \vdash_{\text{empty}}\ \langle\rangle \in t_1 =$ if $isEmpty_{t_1}\ v_1$
                              then case $proj_{(l,t_2)}\ v_2$ of
                                $Just\ (v_l, v'_2) \rightarrow Just\ (v_l, R\ v'_2)$
                                $Nothing \rightarrow Nothing$
                              else case $proj_{(l,t_1)}\ v_1$ of
                                $Just\ (v_l, v'_1) \rightarrow Just\ (v_l, L\ (v'_1, v_2))$
                                $Nothing \rightarrow Nothing$
     $\mid \neg(\vdash_{\text{empty}}\ \langle\rangle \in t_1) =$ case $proj_{(l,t_1)}\ v_1$of
                                  $Just\ (v_l, v'_1) \rightarrow Just(v_l, (v'_1, v_2))$
                                  $Nothing \rightarrow Nothing$
```

Figure 6.12: An implementation of $proj_{(l,t)}$ function

left alternative) or (2) $mkEmpty_{t_2}$ (the right alternative). We favor the left alternative (1). Note that this is another source of ambiguity. This decision will affect the result of the coercive pattern matching algorithm. The impact of this decision is discussed shortly.

2. Functions $proj_{(l,t)}$ and $inj_{(l,t)}$

Example 49 Recall the earlier running example (Example 48) in which we were proving $\vdash_{\text{sub}} A^* \leq_{d3} A^*$. In one of the steps, we rewrite the type A^* to A^*/A, which is $\langle\langle\rangle, A^*\rangle$. On the value level, we want to rewrite the value from type $[\![A^*]\!]$ to a pair consisting of the leading label A and the remaining part of

```
$\mathbf{inj_{(l,t)}} : \forall[\![\mathbf{l}]\!] \to [\![\mathbf{t/l}]\!] \to [\![\mathbf{t}]\!]$
 $inj_{(l,\langle\rangle)}\ l\ v = error\ \text{“}undefined\text{”}$
 $inj_{(l_1,l_2)}\ l\ ()\ |\ l_1 \neq l_2 = error\ \text{“}undefined\text{”}$
                  $|\ \text{otherwise} = l$
 $inj_{(l,t^*)}\ l\ v = \mathsf{let}\ \{(v_1,v_2) = v\}\ \mathsf{in}\ (inj_{(l,t)}\ l\ v_1) : v_2$
 $inj_{(l,(t_1|t_2))}\ l\ v = \mathsf{case}\ v\ \mathsf{of}$
                   $L\ v_1 \to L\ (inj_{(l,t_1)}\ l\ v_1)$
                   $R\ v_2 \to R\ (inj_{(l,t_2)}\ l\ v_2)$
 $inj_{(l,\langle t_1,t_2\rangle)}\ l\ v$
     $|\ \vdash_{\text{empty}}\ \langle\rangle \in t_1 = \mathsf{case}\ \ v\ \mathsf{of}$
           $L\ v_1 \to \mathsf{let}\ (v_1',v_2') = v_1\ \mathsf{in}\ (inj_{(l,t_1)}\ l\ v_1', v_2')$
           $R\ v_2 \to (mkEmpty_{t_1}, inj_{(l,t_2)}\ l\ v_2)$
     $|\neg(\vdash_{\text{empty}}\ \langle\rangle \in t_1) = \mathsf{let}\ (v_1,v_2) = v\mathsf{in}\ (inj_{(l,t_1)}\ l\ v_1, v_2)$
```

Figure 6.13: An implementation of $inj_{(l,t)}$ function

type $[\![\langle\langle\rangle, A^*\rangle]\!]$. We refer to this value re-writing operation as the projection function. Since $[\![A^*]\!] = [A]$ and $[\![((),[A])]\!] = ((),[\mathtt{A}])$, we demand the projection function $proj_{(A,A^*)}$ to coerce values from type `[A]` to `(A,((),[A]))`.

In this particular example, the definition of the projection function is fairly straight-forward.

```
proj_(A,A*) ::  [A] -> Maybe (A,((),[A]))
proj_(A,A*) v = case v of
               [] -> Nothing
               Just (x:xs) -> case proj_(A,A) x of
                              Just (x1,x2) -> Just (x1, (x2, xs))
                              Nothing -> Nothing


proj_(A,A) v = Just (v,())
```

note that the function's output has a `Maybe` type, this is because the projection operation might fail. □

In Figure 6.12, we give general definitions of the projection function $proj_{(l,t)}$. $proj_{(l,t)}$ takes a value of type $[\![t]\!]$ as the input and attempts to rewrite it into the derivative form which is of type $[\![\langle l, t/l\rangle]\!]$. Note that the attempt could fail. Therefore, the result has to be of a `Maybe` type. The first clause tries to extract a label from an empty sequence type. This is impossible. Therefore, `Nothing` is returned to signal a definite failure. The second clause extracts a label from a label. This is only successful if the two labels are identical. The third clause deals with the kleene's star type, which is easy. The fourth clause is more than interesting. We want to extract the label l from a choice type $(t_1|t_2)$. Depending on whether the left or the right alternative is present, we extract the label from the left or the right alternative. The last clause extracts the label from a pair type $\langle t_1, t_2\rangle$. If t_1 does not possess the empty sequence, we can immediately extract the label from the first component of the pair and leave the the second component untouched. Otherwise, we need to test whether the first component of the pair, v_1, is indeed empty. If v_1 is empty, we extract the label from the second component v_2. Otherwise, we have to to extract it from v_1.

Note that the result of the projection function is always a pair, whose first component is the leading label and the second component is the remainder. However, as we observe from the above example, the remainder is not in its canonical form. For instance, the output of $proj_{(A,A^*)}$ is of type `(A,((),[A]))`. A notable point is that we have to maintain the correspondence between the derivatives and their canonical forms. In the subsequent steps, we will take away the leading `A` and apply sub-downcast to the remainder. However, the remainder is of type `((),[A])` which needs to turn into the canonical form. To make the type right, we have to apply another simplification function *to* to the remainder. We will get back to the details of the *to* function shortly.

The injection function $inj_{(l,t)}$ is the inverse of the projection function. It takes two arguments, a label value of type $[\![l]\!]$ and a value of type $[\![t/l]\!]$, then returns a result of type $[\![t]\!]$ where the first argument l is injected to the second argument.

Example 50 For instance, in our running example (Example 48) which solves $\vdash_{\text{sub}} A^* \leq_{d3} A^*$, we have used the projection function $proj_{(A,A^*)}$ to rewrite a value of type $[\![A^*]\!]$ into type $[\![(A^*/A)]\!]$. Now we demand another opposite operation, the injection operation, $inj_{(A,A^*)}$, which turns values from type $[\![(A^*/A)]\!]$ back to $[\![A^*]\!]$.

```
inj_(A,A*) ::  A -> ((),[A]) -> [A]
inj_(A,A*) l v = let (v1,v2) = v
                 in (inj_(A,A) l v1) :  v2
inj_(A,A) l v = l
```

The first argument of $inj_{(A,A^*)}$ is a label A. The second argument is the result from the $from$ function. A $from$ function rewrites the result from the canonical form which is A. back to the "unabridged" derivative form $\langle\langle\rangle, A\rangle$. We will soon explain the $from$ function together with the *to* function. $\Box$

The detail definition of the injection function is given in Figure 6.13. The first pattern is raising a run-time error. This is because we try to construct an empty value by injecting a label into some value. In the actual program this will not happen because the injection function will be derived from a valid derivative operation, and there is no regular expression r satisfying $\langle\rangle/l = r$. In the second pattern, a label l_1 is injected into an empty value to form a value of type l_2. This is only possible when both types agree. The third pattern handles kleene's star. By $t^*/l = \langle t/l, t^*\rangle$, we know that v must be a pair value. We inject l into the first component of v, which will be concatenated with with the rest of v. The fourth pattern deals with choice type. We inject the label either to the left or right alternative depending on whichever is present as the second argument. In the last pattern, we create a value of type $\langle t_1, t_2\rangle$ by injecting the label l into the second argument. We proceed with a case analysis.

(a) In the first case, t_1 possesses the empty sequence $\langle\rangle$. Therefore, we note that the second argument must be of a choice type $\langle t_1/l, t_2\rangle|(t_2/l)$. Otherwise the type will not agree. If the second argument is present as a left alternative ($L\ (v_1, v_2)$), we inject the label to the first component of the pair, v_1. Else, we inject the label to the second argument to form a value of type $[\![t_2]\!]$. In addition, we need to create an empty value of type $[\![t_1]\!]$. Putting these two values into a pair, we construct the result of the injection.

(b) In the case where t_1 does not possess the empty sequence, the second argument must be a pair value, (v_1, v_2). This is valid because l can only be extracted from t_1 but not t_2. In this case, we simply inject the label l to v_1.

3. Functions *to* and $from$

As we discussed, we need to maintain the correspondence between the derivatives and their canonical forms. The projection functions and injection functions need to be used in combination with the *to* and $from$ functions, in order to coerce values between their original form and their canonical forms. We derive the *to* and $from$ functions from the simplification rules.

Recall the set of simplification rules from Figure 6.8 as follows,

$$\text{(E1)}\ (\bot|t) = t \qquad \text{(E2)}\ (t|\bot) = t \qquad \text{(E3)}\ (t|t) = t \qquad \text{(E4)}\ ((t_1|t_2)|t_3) = (t_1|(t_2|t_3))$$
$$\text{(E5)}\ \langle\langle\rangle, t\rangle = t \qquad \text{(E6)}\ \langle\bot, t\rangle = \bot$$

On the value level, each simplification rule gives us a pair of coercion functions. Suppose we use `data Phi` to represent $\bot$ in the target language, we then use Haskell type classes to define the *to* and $from$ functions in Figure 6.14.

The type class `Canonical t c` defines a relation between `t` and `c` which says `c` is the canonical form of `t`, and it is uniquely determined by `t` via the depen-

```
class Canonical t c | t -> c where
      to ::  t -> c
      from ::  c -> t
instance Canonical t t' => Canonical (Or Phi t) t' where -- (E1)
      to (R v) = to v
      from v = R (from v)
instance Canonical t t' => Canonical (Or t Phi) t' where -- (E2)
      to (L v) = to v
      from v = L (from v)
instance Canonical t t' => Canonical (Or t t) t' where   -- (E3)
      to (L v) = to v
      to (R v) = to v
      from v = L (from v)
instance (Canonical t1 t1', Canonical (Or t2 t3) t5) =>
      Canonical (Or (Or t1 t2) t3) (Or t1' t4) where     -- (E4)
      to (L (L v)) = L (to v)
      to (L (R v)) = R (to (L v))
      to (R v) = R (to (R v))
      from (L v) = L (L (from v))
      from (R v) = case from v of { (L v) -> L (R v)
                                  ; (R v) -> R v }
instance Canonical t t' => Canonical ((),t) t' where     -- (E5)
      to ((),v) = to v
      from v = ((),from v)
instance Canonical (Phi,t) Phi where                     -- (E6)
      to = error "undefined"
      from = error "undefined"
instance Canonical A A where                             -- (Base case)
      to x = x
      from x = x
...
```

Figure 6.14: Implementing *to* and *from* functions using Haskell type classes

dency `| t -> c`. The first six instances correspond to the six simplification rules (E1) - (E6). Note that these instances are overlapping. This can be easily resolved by specializing the instances. For simplicity, we omit the specialization. We also omit some base cases since most of them are trivial. Each instance defines a pair of specific *to* and $from$ functions. All the definitions are simple except (E3). In the instance of (E3), we collapse a choice type into a single type, which incurs a loss of information on the value level. We need to coerce a value of type $[\![t]\!]$ to type $[\![t|t]\!]$. In the definition of $from$ above, we coerce the input value to the left component. The following alternative is of course a valid definition,

```
instance Canonical t t' => Canonical (Or t t) t' where      -- (E3)
   ...
   from v = R (from v)
```

where we coerce the value to the right component instead of the left component. In short, the definition of the *from* function is ambiguous. This effectively impacts the result of the downcast coercion. We will discuss the resolution shortly.

Given all the helper functions defined above, let us consider the coercive pattern matching algorithm in action.

Example 51 Now let us apply the coercive pattern matching algorithm to the running example $\vdash_{\text{sub}} A^* \leq_{d3} A^*$ (Example 48) in full detail. We recall the proof derivation as follows,

$$\dfrac{\dfrac{A^* \leq A^* \in \{A^* \leq A^*\}}{\{A^* \leq A^*\} \vdash_{\text{sub}} A^* \leq A^*}\text{(Hyp)} \quad A^*/A = \langle\langle\rangle, A^*\rangle \quad |\langle\langle\rangle, A^*\rangle| = A^*}{\vdash_{\text{sub}} A^* \leq A^*}\text{(Norm)}$$

Putting all the helper functions together, we define the "unabridged" version of the downcast coercion function as follows,

```
d ::  [A] -> Maybe [A]
d v = if $isEmpty_{A^*}$ v then Just $mkEmpty_{A^*}$
      else case $proj_{(A,A^*)}$ v of
            Just (l,rest) -> case $d$ (to rest) of
```

```
                                Just rest' -> Just ($inj_{(A,A^*)}$ l (from rest'))
                                Nothing -> Nothing
```

In the above definition, we inline the *select* function in the body of d. Note that we have defined $proj_{(A,A^*)}$ and $inj_{(A,A^*)}$ in Examples 49 and 50. The other helper functions are defined as follows,

```
$isEmpty_{A^*}$ [] = True
$isEmpty_{A^*}$ _ = False

$mkEmpty_{A^*}$ = []
```

By resolving the type class constraint `Canonical ((),[A]) [A]`, we derive the `to` and `from` functions as follows,

```
to ((),v) = v
from v = ((),v)
```

□

6.3.3 The Ambiguity Problem

As we observed earlier, there are multiple choices in building the downcast coercion. The choices reflect different kinds of pattern matching policies. There are two parameters leading to the ambiguity, i.e., the *mkEmpty* and the $from$ functions.

We use the following example to illustrate that the helper function *mkEmpty* is one of the parameters that leads to the ambiguity problem.

Example 52 Let us consider a downcast function d that is derived from the subtype proof $\vdash_{\text{sub}} \langle A^*, A^* \rangle \leq_d A^*$,

$$\frac{\begin{array}{c} \vdash_{\text{sub}} (\langle A^*, A^* \rangle | A^*) \leq_{d'} A^* \\ (A^*/A) = \langle \langle \rangle, A^* \rangle \quad |\langle \langle \rangle, A^* \rangle| = A^* \\ (\langle A^*, A^* \rangle / A) = \langle \langle \langle \rangle, A^* \rangle, A^* \rangle | \langle \langle \rangle, \langle \langle \rangle, A^* \rangle \rangle \quad |\langle \langle \langle \rangle, A^* \rangle, A^* \rangle | \langle \langle \rangle, \langle \langle \rangle, A^* \rangle \rangle| = (\langle A^*, A^* \rangle | A^*) \end{array}}{\vdash_{\text{sub}} \langle A^*, A^* \rangle \leq_d A^*}$$

where the definition of d can be found in Figure 6.15. In the definition of d, we inline the *select* function into the body of d. Definitions of the helper functions $\mathit{isEmpty}_{A^*}$, $\mathit{proj}_{(A,A^*)}$ and *to* can be found in Example 51. Definitions of functions $\mathit{inj}_{(A,\langle A^*,A^* \rangle)}$, from and $\mathit{mkEmpty}_{\langle A^*,A^* \rangle}$ can be found in Figure 6.15 right below the definition of d.

Note that these definitions are not interesting, since the definition of $\mathit{mkEmpty}_{\langle A^*,A^* \rangle}$ is unique. The interesting bit appears in the definition of the sub proof d'. For simplicity, we only consider the case in which the input to d' is empty, as stated in Figure 6.15.

In function d', we employ a helper function $\mathit{mkEmpty}_{(\langle A^*,A^* \rangle | A^*)}$ to construct an empty value of type `(Or ([A],[A]) [A])`. Note that $\mathit{mkEmpty}_{(\langle A^*,A^* \rangle | A^*)}$ has more than one valid definition, which can be found in Figure 6.15. The result of the downcast coercion varies depending on the definition of $\mathit{mkEmpty}_{(\langle A^*,A^* \rangle | A^*)}$,

```
      d [A]
⟶    if $\mathit{isEmpty}_{A^*}$ [A] then...
      else case $\mathit{proj}_{(A,A^*)}$ [A] of
        Just $(l, v')$ → case $d'$ $(\mathit{to}\ v')$ of
          Just $v''$ → Just $(\mathit{inj}_{(A,\langle A^*,A^* \rangle)}\ l\ (\mathit{from}\ v''))$
          Nothing → Nothing
        Nothing → Nothing
```

```
d :  [A] -> Maybe ([A],[A])
d v = if $isEmpty_{A^*}$ v then Just $mkEmpty_{\langle A^*,A^*\rangle}$
      else case $proj_{(A,A^*)}$ v of
           Just (l,v') -> case d' ($to$ v') of
                          Just v'' -> Just ($inj_{(A,\langle A^*,A^*\rangle)}$ l ($from$ v''))
                          Nothing -> Nothing
           Nothing -> Nothing

$mkEmpty_{\langle A^*,A^*\rangle}$ ::  ([A],[A])
$mkEmpty_{\langle A^*,A^*\rangle}$ = ([],[])

-- derived from $|\langle\langle\langle\rangle, A^*\rangle, A^*\rangle|\langle\langle\rangle, \langle\langle\rangle, A^*\rangle\rangle| = (\langle A^*, A^*\rangle|A^*)$
$from$ ::  (Or ([A],[A]) [A]) -> (Or (((),[A]),[A]) ((),[A]))
$from$ (L (x,y)) = L (((),x),y)
$from$ (R y) = R ((),y)

-- derived from $(\langle A^*, A^*\rangle/A) = \langle\langle\langle\rangle, A^*\rangle, A^*\rangle|\langle\langle\rangle, \langle\langle\rangle, A^*\rangle\rangle$
$inj_{(A,\langle A^*,A^*\rangle)}$ ::  A -> (Or (((),[A]),[A]) ((),[A])) -> ([A],[A])
$inj_{(A,\langle A^*,A^*\rangle)}$ l (L ((),x),y) = (l:x,y)
$inj_{(A,\langle A^*,A^*\rangle)}$ l (R ((),y) = ([],l:y)

d' ::  [A] -> Maybe (Or ([A],[A]) [A])
d' v = if $isEmpty_{A^*}$ v then $mkEmpty_{(\langle A^*,A^*\rangle|A^*)}$ else ...

$mkEmpty_{(\langle A^*,A^*\rangle|A^*)}$ ::  (Or ([A],[A]) [A])
$mkEmpty_{(A\langle A^*,A^*\rangle|A^*)}$ = L ([],[]) -- (1)
-- or
$mkEmpty_{(A\langle A^*,A^*\rangle|A^*)}$ = R []   -- (2)
```

Figure 6.15: A running example proving $\vdash_{\text{sub}} \langle A^*, A^*\rangle \leq_d A^*$

$\longrightarrow$ case $proj_{(A,A^*)}$ $[A]$ of (because $isEmpty_{A^*}$ $[A]$ $\longrightarrow^*$ $False$)

Just $(l, v') \to$ case d' $(to\ v')$ of

Just $v'' \to$ Just $(inj_{(A,\langle A^*,A^*\rangle)}\ l\ (from\ v''))$

Nothing $\to$ Nothing

Nothing $\to$ Nothing

$\longrightarrow$ case d' $(to\ ((), []))$ of (because $proj_{(A,A^*)}$ $[A]$ $\longrightarrow^*$ $(A, ((), [])))$

Just $v'' \to$ Just $(inj_{(A,\langle A^*,A^*\rangle)}\ A\ (from\ v''))$

Nothing $\to$ Nothing

$\longrightarrow$ case d' $[]$ of (because to $((), [])$ $\longrightarrow$ $[])$

Just $v'' \to$ Just $(inj_{(A,\langle A^*,A^*\rangle)}\ A\ (from\ v''))$

Nothing $\to$ Nothing

Depending on how we define $mkEmpty_{(A^*,A^*|A^*)}$, the above execution proceeds differently.

1. Suppose we favor definition `(1)` as given in Figure 6.15, the application $(d'\ [])$ is evaluated as follows,

$$\begin{array}{ll} & d'\ [] \\ \longrightarrow & \mathsf{Just}\ mkEmpty_{(A^*,A^*)} \\ \longrightarrow & \mathsf{Just}\ (L\ ([],[])) \end{array}$$

   Therefore, the main execution yields the following,

$$\begin{array}{ll} \longrightarrow & \mathsf{Just}\ (inj_{(A,\langle A^*,A^*\rangle)}\ A\ (from\ (L\ ([],[])))) \\ & \mathsf{Just}\ (inj_{(A,\langle A^*,A^*\rangle)}\ A\ (((),[]),[]) \\ & \mathsf{Just}\ ([A],[]) \end{array}$$

   Note that in the final result, `([A],[])`, `A` is located in the first component of the pair structure, which means that the above is implementing the POSIX/longest matching policy.

2. Suppose we favor definition `(2)` as given in Figure 6.15, the application $(d'\ [])$ is evaluated as follows,

$$\begin{array}{ll} & d'\ [] \\ \longrightarrow & \mathsf{Just}\ mkEmpty_{(A^*,A^*)} \\ \longrightarrow & \mathsf{Just}\ (R[]) \end{array}$$

   Therefore, the main execution proceeds as follows,

$$\begin{array}{ll} \longrightarrow & \mathsf{Just}\ (inj_{(A,\langle A^*,A^*\rangle)}\ A\ (from\ (R\ []))) \\ & \mathsf{Just}\ (inj_{(A,\langle A^*,A^*\rangle)}\ A\ ((),[]) \\ & \mathsf{Just}\ ([],[A]) \end{array}$$

Note that the final result `([],[A])`, `A` is located in the second component of the pair structure, which means that the above is implementing the *shortest* matching policy, which is the opposite of the POSIX/Longest matching policy.

□

The above example shows that by switching the implementation of $mkEmpty_t$ we change the meaning of resulting downcast coercion.

In the next example, we show that the function $from$ is another factor that changes the behavior of the downcast coercion.

Example 53 Consider $\vdash_{\text{sub}} (A|A) \leq A$, whose derivation is as follows,

$$\frac{\begin{array}{c}\vdash_{\text{sub}} \langle\rangle \leq_{d'} \langle\rangle \\ A/A = \langle\rangle \\ (A|A)/A = (\langle\rangle|\langle\rangle) \quad |\langle\rangle|\langle\rangle| = \langle\rangle\end{array}}{\vdash_{\text{sub}} (A|A) \leq_d A}$$

According to the coercive pattern matching algorithm, we define d as follows,

```
d v = case $proj_{(A,A)}$ A of
          Just (1,v') -> case $d'$ v' of
                              Just v'' ($inj_{(A,(A|A))}$ 1 (from v''))
                              Nothing -> Nothing
          Nothing -> Nothing


-- derived from $A/A = \langle\rangle$
$proj_{(A,A)}$ ::  A -> Maybe (A,())
$proj_{(A,A)}$ v = Just (v,())


-- derived from $\vdash_{\text{sub}} () \leq_{d'} ()$ and simplified
```

d' `::  () -> Maybe ()`

d' `v = Just ()`

`-- derived from` $(A|A)/A = (\langle\rangle|\langle\rangle)$

$inj_{(A,(A|A)}$ `::  A -> (Or () ()) -> (Or A A)`

$inj_{(A,(A|A)}$ `l (L x) = (L l)`

$inj_{(A,(A|A)}$ `l (R y) = (R l)`

`-- derived from` $|\langle\rangle|\langle\rangle| = \langle\rangle$

`from ::  () -> (Or () ())`

Note that there are two valid definitions of $from$ we can give

```
from v = L v -- (3)
-- or
from v = R v -- (4)
```

Let us apply d to a value `A`. Depending on which definition of $from$ we choose, the application (`(`d `A)`) yields either `(L A)` or `(R A)`. □

We have illustrated the ambiguity problem of the coercive pattern matching by going through two complete examples. The main problem is that when we rewrite regular expression into canonical forms, we lose the structure of the original expression. Under such circumstances, we often find that we may give more than one valid definition of $mkEmpty_t$ and $from$. The result varies depends on which implementations of functions $mkEmpty_t$ and $from$ are used.

From this point onwards, we adopt the following strategy to resolve ambiguity: When there are more than one valid definition, we always favor the definition producing a left alternative. For instance, recall in Example 52 where we favored the definition `(1)` in Figure 6.15, for $mkEmpty_{(\langle A^*,A^*\rangle|A^*)}$. In the previous Example 53, we favored the definition `(3)` when we define the function $from$.

Using this resolution strategy, the coercive pattern matching algorithm that we develop is implementing the POSIX/Longest matching policy. In the next section, we verify the correctness of the coercive pattern matching algorithm under this decision.

6.3.4 Correctness

In this subsection, we consider the correctness result of the coercive pattern matching algorithm. We make a comparison between the POSIX/Longest matching algorithm and the coercive pattern matching algorithm . Since we have proven that the former is correct in the earlier section, we can easily show that the latter is correct too by establishing an isomorphism among the two.

Let us consider an example.

Example 54 Recall the earlier running example,

$\texttt{f = } \lambda(z : A^*).\mathsf{case}\ z\ \mathsf{of}\ \langle x : A^*, y : A^*\rangle \rightarrow \langle x, y\rangle$

Suppose now we apply the POSIX/Longest matching algorithm to match the input A with the pattern $\langle x : A^*, y : A^*\rangle$.

$$\begin{array}{ll}
 & \text{longmatch } A\ \langle x : A^*, y : A^*\rangle \\
\longrightarrow & \text{longmatch } \langle\rangle\ \langle x : A^*, y : A^*\rangle/A \\
\longrightarrow & \text{longmatch } \langle\rangle\ \langle [A]x : \langle\langle\rangle, A^*\rangle, y : A^*\rangle | \langle x : \langle\rangle, [A]y : \langle\langle\rangle.A^*\rangle\rangle) \\
\longrightarrow & \text{longmatch } \langle\rangle\ \langle [A]x : \langle\langle\rangle, A^*\rangle, y : A^*\rangle \\
\longrightarrow & \mathit{Just}\ [(x, A), (y, \langle\rangle)]
\end{array}$$

In the above, we use pattern derivative to extract leading label from input value and store the label into the pattern derivative. We keep doing that until the input value is empty. When the input is empty, we extract value from the pattern derivative.

Recall that we can visualize the POSIX/Longest matching algorithm in terms of a tree that contains all possible matches as follows,

$$\begin{tikzcd}
& \langle x : A^*, y : A^* \rangle \arrow[dl, dash, "A"'] \arrow[dr, dash, "A"] & \\
\langle [A]x : A^*, y : A^* \rangle & & \langle x : \langle\rangle, [A]y : A^* \rangle
\end{tikzcd}$$

Under the POSIX/Longest matching policy, we search for the *first* successful match from left to right. In the tree above, $\langle [A]x : A^*, y : A^* \rangle$ is the left most leaf node that yields a match $\{A/x, \langle\rangle/y\}$. It is the result "favored" by the POSIX/Longest match algorithm. □

Comparing the above example with Example 52, we can easily realize that the POSIX/Longest matching algorithm is in essence very similar to the coercive pattern matching algorithm. In both algorithms,

1. we perform pattern matching by rewriting input into derivative forms;

2. we have choice points in which selecting a particular choice point affects the match results;

3. we favor the left alternatives which leads to the longest matched results.

Since we have already proven that the POSIX/Longest matching algorithm longmatch $w\ p$ is correct, we can verify that the coercive pattern matching algorithm is also correct if we are able to establish an isomorphic relation among the two approaches.

However, there are still two issues that prevent us from building the isomorphic relation.

- First, the two approaches use different runtime representations. The POSIX/Longest pattern matching algorithm operates on uniform (unstructured) values, whilst the coercive pattern matching algorithm operates on structured values;

- In the coercive pattern matching algorithm, we further rewrite derivative into canonical forms, which are not available in the POSIX/Longest pattern matching algorithm.

To address the first issue, we define the following isomorphic relation which associates the unstructured value with the structured value,

Let w and w' be two System F* values (uniform representation), we define w/w' to be the resulting value by removing the prefix w' from w. We define a relation between the downcast result and the (partial) pattern derivative as follows,

Definition 6 (Isomorphism between structure values and pattern derivatives) *Let v be a structured value resulting from coercive pattern matching. Let θ be a value environment which maps pattern variables to uniform representation values. Let p be a pattern. We define $v \overset{p}{\sim} \theta$ as follows,*

$$\begin{array}{rll}
v \overset{([w]\ x:t)}{\sim} \{(w'/x)\} & \textit{iff} & w/w' \overset{t}{\dot{\approx}} v \\
(v_1, v_2) \overset{\langle p_1, p_2 \rangle}{\sim} \theta_1 \cup \theta_2 & \textit{iff} & v_1 \overset{p_1}{\sim} \theta_1 \wedge v_2 \overset{p_2}{\sim} \theta_2 \\
L\ v \overset{(p_1|p_2)}{\sim} \theta & \textit{iff} & v \overset{p_1}{\sim} \theta \\
R\ v \overset{(p_1|p_2)}{\sim} \theta & \textit{iff} & v \overset{p_2}{\sim} \theta
\end{array}$$

$v \overset{p}{\sim} \theta$ ties up the connection between a structure value v with a match result θ in uniform representation under a pattern p. The first rule says that a structure value v is isomorphic to a single value binding $\{w'/x\}$ under a variable pattern $([w]\ x : t)$ if we compare w/w', which is what remains after taking away w from w', with v, we find that they are semantically equivalent. Note that the semantic equivalence relation $\cdot \dot{\approx} \cdot$ is defined in Figure 5.5 Section 5.2 Chapter 5. The second rule defines the isomorphic relation between a pair value and the union of two value bindings. The third and forth rules handle `Or` type values.

Example 55 For example, let `v = ([A],[])` be a structured value, $\theta = \{A/x, \langle\rangle/y\}$ be a value binding and $p = \langle x : A^*, y : A^* \rangle$. We find that

$$v \overset{p}{\sim} \theta$$

$$\text{(E1) } (\bot|t) = t \qquad \text{(E2) } (t|\bot) = t \qquad \text{(E3) } (t|t) = t \qquad \text{(E4) } ((t_1|t_2)|t_3) = (t_1|(t_2|t_3))$$

$$\text{(E5) } \langle\langle\rangle, t\rangle = t \qquad \text{(E6) } \langle\bot, t\rangle = \bot$$

Figure 6.16: The Simplification Rules (repeated)

because

$$[A] \overset{x:A^*}{\sim} \{A/x\} \text{ and } [] \overset{y:A^*}{\sim} \{\langle\rangle/y\}$$

hold.

On the other hand, let `v' = ([],[A])` be another structured value, we find that

$$\neg(v' \overset{p}{\sim} \theta)$$

because neither

$$[] \overset{x:A^*}{\sim} \{A/x\} \text{ nor } [A] \overset{y:A^*}{\sim} \{\langle\rangle/y\}$$

holds. □

To address the second issue, we introduce some pruning rules to the POSIX/Longest matching algorithm that correspond to the simplification rules used in the coercive pattern matching algorithm.

We repeat the definitions of the simplification rules given in the early Section 6.3.2, in Figure 6.16. We also recall the type class implementation of the simplification rules in Figure 6.14 in Section 6.3.2.

For each simplification rule defined for the coercive pattern matching algorithm, we introduce a correspondent pruning rule in the POSIX/Longest pattern matching algorithm.

For simplicity we consider two interesting simplification rules, namely (E3) and (E5).

In (E3), we collapse a choice type $(t|t)$ into t. We can derive two possible pruning operations from (E3).

1. In the first case, we consider the pattern $([w]\ x : (t|t))$. It is clear that we can collapse the type annotation to obtain $([w]\ x : t)$. These two pattern should have the same semantics in any matching policy.

2. In the second case, we consider the pattern $(p_1|p_2)$, where stript $p_1 =$ stript $p_2 = t$. Since both patterns match with the same set of words, we can drop either one. However under the POSIX/Longest matching policy, the only safe operation is to drop p_2, because of the following property,

   Lemma 17 *Let p_1 and p_2 be patterns such that stript $p_1 =$ stript p_2. Let w be a word. Then $w \lhd_{lm} p_1 \leadsto \theta$ iff $w \lhd_{lm} (p_1|p_2) \leadsto \theta'$ where $\theta = \theta'$.*

   The above guarantees that it is safe to prune away p_2, because under POSIX/Longest matching policy, we always favor the first successful match. Since any word matching with p_2 must match with p_1, it is therefore safe to prune away p_2. In other words, we can add the following "pruning rule" which is applied to the result of p/l.

   $$(p_1|p_2) = \ \mathsf{if}(\text{stript } p_1 == \text{stript } p_2)\ \mathsf{then}\ p_1\ \mathsf{else}\ (p_1|p_2) \quad \text{(Prune1)}$$

   which checks for syntactic equality of two alternative patterns' type annotations. If they are the same, we prune away p_2, otherwise, the pattern remains unchanged.

   Hence, when we apply simplification rule (E3) in our subtype proof, we commit ourselves to the POSIX/Longest matching policy and this is correspondent to making use of the above pruning rule in the POSIX/Longest matching algorithm.

   When choosing a proper definition for function `from` under the instance of

(E3), we favor the definition injecting the value to the left component, over the following alternative,

```
instance Canonical t t' => Canonical (Or t t) t' where      -- (E3)
   ...
   from v = R (from v)
```

This is in sync with the above "pruning" that takes place in the POSIX/Longest matching algorithm.

In (E5), we simplify a sequence type $\langle\langle\rangle, t\rangle$ to t. Similarly, there are two possible pruning operations derivable.

1. In the first case, we can apply the following rule to prune the pattern,

   $$([w]\ x : \langle\langle\rangle, t\rangle) = ([w]\ x : t) \quad \text{(Prune2)}$$

   ll which is trivial.

2. In the second case, we consider the pattern $\langle([w]\ x : \langle\rangle), p\rangle$. Since the sub-pattern $([w]\ x : \langle\rangle)$ will not consume any label, we conclude that any word that matches with p will match with $\langle([w]\ x : \langle\rangle), p\rangle$. We can apply the operation because the following property holds,

   Lemma 18 *Let p be a pattern and w be a word. Then $w \lhd_{lm} p \rightsquigarrow \theta$ iff $w \lhd_{lm} \langle([w]\ x : \langle\rangle), p\rangle \rightsquigarrow \theta'$ where $\{w/x\} \cup \theta = \theta'$.*

   In other words, we should apply the following "short-cut" to the longmatch $w\ p\theta$ algorithm,

   ```
   longmatch $w$ $\langle [w']\ x : \langle\rangle, p\rangle =$
     case longmatch $w$ $p$ of
     $Just\ \theta \rightarrow Just\ \{(w'/x)\} \cup \theta$
     $Nothing \rightarrow Nothing$
   ```

When we apply simplification rule (E5) in our subtype proof, we apply function $from_{(E5)}$ to the "abridged" value to "recover" the empty sequence component which should be part of the "unabridged" value. Note that the definition of $from_{(E5)}$ is definite, thus there is no need to care about the matching policy here.

Similar observations can be made on the remaining simplification rules. We omit the details.

In summary, each simplification rule applied in the subtype proof corresponds to a pruning operation on the search tree. Under POSIX/Longest matching policy, we always prune the sub-tree which does not affect the matching result. This is enforced by choosing the *correct* implementation of the $from_i$ coercion function.

Example 56 We recall Example 54, in which we have a rough comparison between the two pattern matching approaches by looking at the result of two pattern matching implementations, namely

$$\text{longmatch } A \ \langle x : A^*, y : A^* \rangle \longrightarrow^* \mathit{Just}\ \{(A/x), (\langle\rangle/y)\}$$

versus

$$d\ [A] \longrightarrow^* \mathit{Just}\ ([A], [])$$

where d is derived from the proof of $\vdash_{\text{sub}} \langle A^*, A^* \rangle \leq_d A^*$.

Now let us make a detail comparison between these two approaches. Let us first consider the POSIX/Longest matching algorithm.

$$\begin{array}{lll}
 & \text{longmatch } A\ \langle x : A^*, y : A^* \rangle & (1) \\
\longrightarrow & \text{longmatch } \langle\rangle\ (\langle [A]x : A^*, y : A^* \rangle | \langle x : \langle\rangle, [A]y : A^* \rangle) & (2) \\
\longrightarrow & \text{longmatch } \langle\rangle\ \langle [A]x : A^*, y : A^* \rangle & (3) \\
\longrightarrow & \mathit{Just}\ [(x, A), (y, \langle\rangle)] & (4)
\end{array}$$

From step (1) to (2), we compute $\langle x : A^*, y : A^*\rangle/A = (\langle [A]x : \langle\langle\rangle, A^*\rangle, y : A^*\rangle | \langle x : \langle\rangle, [A]y : \langle\langle\rangle.A^*\rangle\rangle)$. Then we apply the pruning rule (Prune2) to prune $(\langle [A]x : \langle\langle\rangle, A^*\rangle, y : A^*\rangle | \langle x : \langle\rangle, [A]y : \langle\langle\rangle.A^*\rangle\rangle)$ to $(\langle [A]x : A^*, y : A^*\rangle | \langle x : \langle\rangle, [A]y : A^*\rangle)$.

On the other hand, we recall the coercive pattern matching algorithm,

```
d :  [A] -> Maybe ([A],[A])
d v = if $isEmpty_{A^*}$ v then Just $mkEmpty_{\langle A^*,A^*\rangle}$
        else case $proj_{(A,A^*)}$ v of
              Just (l,v') -> case d' v' of
                              Just v'' -> Just ($inj_{(A,\langle A^*,A^*\rangle)}$ l v'')
                              Nothing -> Nothing
              Nothing -> Nothing
```

(We omit the definitions of the helper functions as they can be found in Figure 6.15.)

The interesting observation is that at each (intermediate) level, we find that the result of the POSIX/Longest matching algorithm is always in isomorphic relation with the result of the coercive pattern matching algorithm.

- Let us consider the top level of the POSIX/Longest matching algorithm, the pattern is $p = \langle x : A^*, y : A^*\rangle$. The match result of the POSIX/Longest pattern matching algorithm is $\{(A/x), (\langle\rangle/y)\}$. The result of the downcast application `(d [A])` is `Just ([A],[])`. We find the following

  $$([A], []) \overset{\langle x:A^*, y:A^*\rangle}{\sim} \{(A/x), (\langle\rangle/y)\}$$

  holds, because

  $$[A] \overset{x:A^*}{\sim} \{(A/x)\} \text{ and } [] \overset{y:A^*}{\sim} \{(\langle\rangle/x)\}$$

- In the intermediate step (2), the pattern is,

  $$(\langle [A]x : A^*, y : A^*\rangle | \langle x : \langle\rangle, [A]y : A^*\rangle)$$

Whilst in the intermediate step of the downcast coercion, we find that `(d' [])` is reduced to `Just (L ([],[]))`.

We realize that the two intermediate results agree with the isomorphism,

$$L\ ([], []) \overset{(\langle [A]x{:}A^*, y{:}A^* \rangle | \langle x{:}\langle\rangle, [A]y{:}A^* \rangle)}{\sim} \{(A/x), (\langle\rangle/y)\}$$

On the contrary, suppose we implement `(d [A])` using the shortest matching policy, by using a different implementation of $mkEmpty_{\langle A^*, A^* \rangle | A^*}$,

$mkEmpty_{\langle A^*, A^* \rangle | A^*}$ `= R [] -- (2)`

Using this definition, we can no longer maintain the isomorphic relation among the structure result of the downcast coercion and the result of the POSIX/Longest matching algorithm, because

$$\neg(R\ []) \overset{(\langle [A]x{:}A^*, y{:}A^* \rangle | \langle x{:}\langle\rangle, [A]y{:}A^* \rangle)}{\sim} \{(A/x), (\langle\rangle/y)\})$$

since

$$\neg([] \overset{\langle x{:}\langle\rangle, [A]y{:}A^* \rangle}{\sim} \{(A/x), (\langle\rangle/y)\})$$

and

$$\neg(([], [A]) \overset{\langle x{:}A^*, y{:}A^* \rangle}{\sim} \{(A/x), (\langle\rangle/y)\})$$

$\square$

From the above example, we find that the isomorphic relation $\cdot \dot{\sim} \cdot$ is always maintained by $mkEmpty_t$, $inj_{(l,t)}$ and $from$ functions.

Lemma 19 (Make empty maintains isomorphism) *Let p be a regular expression pattern. Let stript $p = t$ such that $\vdash_{empty} \langle\rangle \in t$. Then longmatch $\langle\rangle$ $p \longrightarrow^*$ Just θ where $mkEmpty_t \overset{p}{\sim} \theta$.*

Lemma 20 (Injection maintains isomorphism) *Let p and p' be two patterns such that $p/l = p'$. Let stript $p = t$ and stript $p' = t/l$. Let θ be a value binding environment. Let v be a System F value such that $v : [\![t/l]\!]$ and $v \overset{p'}{\sim} \theta$. Then $(inj_{(l,t)}\ v) \overset{p}{\sim} \theta$.*

The proof details of these lemma can be found in Appendix B.3.

Lemma 21 (From coercion maintains isomorphism) *Let p and p' be two patterns such that p' is the pruned version of p. Let stript $p = t$ and stript $p' = |t|$. Let θ be a value binding environment. Let v be a System F value such that $v : [\![|t|]\!]$ and $v \overset{p'}{\sim} \theta$, Let $from$ be the function that is derived from the simplification going from t to $|t|$. Then $(from\ v) \overset{p}{\sim} \theta$.*

The proof details of these lemma can be found in Appendix B.3.

It is clear that Lemma 19, 20, 21 and 22 are the key parts for us to verify that the coercive pattern matching algorithm is faithful with respect to the POSIX/Longest matching algorithm.

Why do we need to maintain the isomorphic relation? We discover that the isomorphic relation $v \overset{p}{\sim} \theta$ guarantees that matching the structured value v (obtained via a downcast operation) against the System F pattern (obtained by translation from p) always yields the correct value bindings with respect to the System F* value binding θ.

Lemma 22 *Let p, θ be a System F^* pattern and a System F^* value environment respectively. Let $\Gamma \vdash_{pat} p : t \rightsquigarrow P$. Let v be a System F value such that $v : [\![t]\!]$ and $v \overset{p}{\sim} \theta$. Let θ_F be a System F value environment, such that $v \lhd_F P \rightsquigarrow \theta_F$. Then $\forall x.\theta(x) \overset{\Gamma(x)}{\approx} \theta_F(x)$*

We provide the detail proofs of this lemma in Appendix B.3.

In the following, we conclude that the coercive pattern matching algorithm is faithful with respect to the POSIX/Longest matching algorithm.

Lemma 23 (Downcast is faithful w.r.t POSIX matching) *Let stript $p = t_1$ and $\vdash_{sub} t_1 \leq_d t_2$. Let w be a System* F* *value such that $w : t_2$ and v_2 be a System F value such that $w \overset{t_2}{\leftrightarrow} v_2$. Then we have*

1. *$d\ v_2 \longrightarrow^* Just\ v_1$ iff longmatch $w\ p \longrightarrow^* Just\ \theta$, where $v_1 \overset{p}{\sim} \theta$;*

2. *$d\ v_2 \longrightarrow^* Nothing$ iff longmatch $w\ p \longrightarrow^* Nothing$.*

This lemma can be verified easily, since we note that $mkEmpty_t$ and $inj_{(l,t)}$ always preserve the isomorphic relation. Therefore, the proof of this lemma leverages on the results of Lemma 19, Lemma 20 and Lemma 21. The proof also depends on the fact that the simplification rules do not break the isomorphic relation. We provide the detail proofs of this lemma in in Appendix B.3.

We conclude this section with the following theorems.

Theorem 7 (Faithful Downcast) *The coercive pattern matching algorithm is faithful with respect to pattern matching relation under POSIX/Longest matching policy.*

The proof follows from lemma 15, 22 and 23.

Theorem 8 (Faithful Translation) *Let e be a system* F**program and E be a system F program such that $\vdash\ e : t \rightsquigarrow E$. Let $e \longrightarrow^* w$ and $E \longrightarrow^* v$. Then $w \overset{t}{\approx} v$.*

The proof follows from Lemma 5 and Theorem 7.

In this section, we present a coercion-based matching algorithm for regular expression pattern matching. The core of the algorithm is to derive a downcast coercion from the regular expression subtype proof. We show that this algorithm is faithful with respect to the POSIX/Longest matching algorithm.

6.4 Summary

In this chapter, we presented the most technical component of this thesis. We studied various techniques of implementing regular expression pattern matching.

Inspired by the regular expression word problem, we developed a novel regular expression pattern matching algorithm based on pattern derivatives. We showed that the algorithm is terminating and correct with respect to the matching relation.

We developed a coercive pattern matching algorithm which operates on a specific set of inputs. This coercive pattern matching algorithm is heavily influenced by the regular expression containment problem. The core of the coercive pattern matching algorithm is the downcast coercion, which can be derived from the regular expression subtyping proof. We provided a complete development of the downcast coercion. We showed that the coercive pattern matching algorithm is faithful with respect to the POSIX/Longest matching algorithm.

Note that we have not mentioned the counter-part of the downcast coercion, namely, the upcast coercion. The development of the upcast coercion is similar to and simpler than that of the downcast coercion. The details of the upcast coercion can be found in Appendix A.3.

Chapter 7

XHaskell Implementation and Applications

In the previous chapters, we have studied the core language of XHaskell. We have developed a translation scheme to System F. We have shown that the translation is coherent and faithful. In this chapter, we present the implementation of XHaskell by putting all these ideas together. The XHaskell system includes a source to source translator from XHaskell to Haskell and a DTD conversion tool which generates XHaskell data types from a DTD file. The XHaskell system prototype is available at [74].

7.1 XHaskell Implementation

The XHaskell source-to-source translator translates XHaskell source programs to Haskell programs, which can be used in combination with GHC [27] version 6.8. Choosing Haskell as the target language has several advantages. First of all, it allows us to incorporate new language features into the prototype with the least effort. For instance, without re-implementing the existing techniques, we added type classes to XHaskell and left the tasks of evidence translation for type classes to the GHC compiler. Furthermore, GHC is well-developed compiler which generates highly optmized executable code for pattern matching.

In this section, we elaborate on some design decisions that we made in the XHaskell implementation.

7.1.1 Regular Expression Type and Type Classes

As we mentioned earlier, XHaskell inherits type classes from Haskell. However, implementing a language that combines regular exprssion types and type class requires some care.

Examples in the earlier chapters show that regular expression types may appear in the type parameters of type classes.

Example 57 For instance, we consider the following type class and one of its instances,

```
class Eq a where
  (==) :: a -> a -> Bool


instance Eq a => Eq a* where ...
```

Suppose some program text gives rise to `Eq (a,a)`. In our subtype proof system, we find that

$$\vdash\ a^* \rightarrow a^* \rightarrow Bool \leq^u (a,a) \rightarrow (a,a) \rightarrow Bool$$

We apply here the co-/contra-variant subtyping rule for functions, which leads to $\vdash\ (a,a) \leq a^*$. The last statement holds. Hence, we can argue that the dictionary E for $Eq\ (a,a)$ can be expressed in terms of the dictionary E' for $Eq\ a^*$ where $E = u\ E'$. □

This suggests a refinement of the type class resolution (also known as context reduction) strategy. Instead of looking for exact matches when resolving type classes with respect to instances, we look for subtype matches. Then, the resolution of `Eq (a,a)` with respect to the above instance yields `Eq a`. The trouble is that type class resolution becomes easily non-terminating. For example, `Eq a` resolves to `Eq a` and so on because of $\vdash a \leq a^*$. We have not found (yet) any simple conditions which guarantees termination under a "subtype match" type class resolution strategy. Therefore, we employ an "exact match" type class resolution strategy which in our experience is sufficient. Thus, we can guarantee decidability of type checking.

7.1.2 Local Type Inference

In XHaskell we give up on automatic type inference because, in the presence of type-based pattern matching, type annotations are crucial [35]. We demand that functions and their arguments are type annotated, and that type instances of polymorphic functions are given. Like many other languages, we employ local type inference methods [54] to avoid an excessive amount of type annotation.

Example 58 For example, we consider the following,

```
filter :: (a|b)* -> b*

filter (x :: b, xs :: (a|b)*) = (x, filter xs)
```

we infer that `filter` is used at type instance `(a|b)* -> b*`. □

To get better results, our implementation takes into account subtyping when building type instances. The following example illustrates this point.

Example 59 We first specify a `foldStar` function for sequences.

```
foldStar :: (a -> b -> a)-> a -> b* -> a

foldStar f x (y::()) = x

foldStar f x (y::b, ys::b*) =  foldStar f (f x y) ys
```

We can straightforwardly infer the missing pattern annotations which are `f::a->b->a` and `x::a`. Thus, we can infer that `foldStar` is used at type instance `(a -> b -> a)-> a -> b* -> a`. Now comes the interesting part.

Suppose we use `foldStar` to build more complex transformations. For example, we want to transform a sequence of alternate occurrences of `a`'s and `b`'s such that all `a`'s occur before the `b`'s. We can specify this transformation via `foldStar` as follows

```
transform :: (a|b)* -> (a*,b*)
transform xs =
   foldStar
     ((\x -> \y ->
        case y of
         (z::a) -> (z,x)
          (z::b) -> (x,z)
      ) :: (a*,b*) -> (a|b) -> (a*,b*))
     () xs
```

The challenge here is to infer that `foldStar` is used at type instance

```
((a*,b*)->(a|b)->(a*,b*))->(a*,b*)->(a|b)*->(a*,b*)
```

From the types of the arguments and the result type of `transform`'s annotation we infer the type

```
((a*,b*)->(a|b)->(a*,b*))->()->(a|b)*->(a*,b*)
```

But this type does not exactly match the above type. The mismatch occurs at the second argument position. Therefore, we take into account subtyping when checking for type instances. We find that $\vdash$ `()` $\leq (a^*, b^*)$ which resolves the mismatch. Hence, our implementation accepts the above program.

□

7.1.3 Pattern Inference

In XHaskell, we can also omit the annotations of patterns.

Example 60 In case of

```
filter :: (a|b)* -> b*

filter (x :: b, xs) = (x, filter xs)
```

we infer that the type `(a|b)*` can reach the pattern `xs`. □

Pattern inference must dictate the pattern matching operational semantics, thus it depends on the particular pattern matching policy [66]. As we have seen earlier in Chapter 5, in our system, pattern matching is translated via down-cast coercions. In Chapter 6, we have shown that our down-cast coercions implement the POSIX matching policy. In XHaskell, the missing pattern annotations are inferred under the POSIX matching policy.

Example 61 We consider the following (contrived) example

```
data A = A

g :: A* -> ()

g (x :: A*, y :: A*) = y
```

The point is that we could distribute the input vlue to `x` and `y` in any arbitrary way. Under the POSIX matching policy, the sub-pattern `(x :: A*)` consumes all `A`s greedily and leaves `y` with the empty sequence `()`.

If we omit the pattern annotations, for example consider the following variant

```
g2 :: A* -> ()

g2 (x, y) = y
```

POSIX pattern match inference yields that `x` has type `A*` and `y` has type `()`. □

7.1.4 Type Error Support

A challenge for any compiler system is to provide meaningful type error messages. This is in particular important in case the expressiveness of the type system increases. The XHaskell compiler is built on top of the Chameleon system [63] and thus we can take advantage of Chameleon's type debugging infrastructure [61, 62] to provide concise location and explanation information in case of a type error.

The following program has a type error in the function body because the value `x` of type `(B|A)*` is not a subtype of the return type `(B|C)*`.

```
data A = A
data B = B
data C = C


f :: (B|A)* -> (B|C)*
f (x :: (B|A)*) = x
```

The compiler reports the following error.

```
ERROR: XHaskell Type Error
Expression at:
f (x ::  (B|A)*) = x
has an inferred type (B|A)* which is not a subtype of (B|C)*.
Trivial inconsistencies probably arise at:
f ::  (B|A)* -> (B|C)*
f (x ::  (B|A)*) = x
```

The error report contains two parts. The first part says that a subtyping error is arising from the body of function `f`, namely the expression `x`. The second part points out the cause of the type error. We found the data type `A` in `x`'s inferred type, which is not part of the expected type. This is a very simple example but shows that we can provide fairly detailed information about the possible cause of a type error.

Instead of highlighting the entire expression we only highlight sub-expressions which are involved in the error.

As an extra feature we allow the postponement of certain type checks till run-time. Let's consider the above program again. The program contains a static type error because the value `x` of type `(B|A)*` is not a subtype of `(B|C)*`. In terms of our translation scheme, we cannot derive the up-cast coercion among the target expression because the subtype proof obligation $\vdash A \leq C$ cannot be satisfied. But if `x` only carries values of type `B*` the subtype relation holds. Hence, there is the option not to immediately issue a static type error here. For each failed subtype proof obligation such as $\vdash A \leq C$ we simply generate an "error" case which then yields for our example the following up-cast coercion.

```
u :: [Or B A] -> [Or B C]
u (L b:xs) = (L b):(u xs)
u (R a:xs) = error "run-time failure: A found where B or C is expected"
```

The program type checks now, but the translated program will raise a run-time error if the sequence of values passed to function `f` consists of an `A`.

The option of mixing static with dynamic type checking by "fixing" coercions is quite useful in case the programmer provides imprecise type information. In case of imprecise pattern annotations, we can apply pattern inference to infer a more precise type. The trouble is that the standard pattern inference strategy [35] may fail to infer a more precise type as shown by the following contrived example.

```
g :: (A,B)|(B,A) -> (A,B)|(B,A)
g (x :: (A|B), y :: (A|B)) = (x,y)
```

It is clear that either (1) `x` holds a value of type `A` and `y` holds a value of type `B`, or (2) `x` holds a `B` and `y` an `A`. Therefore, the above program ought to type check. The problem is that pattern inference computes a type binding for each pattern variable. The best we can do here is to infer the pattern binding $\{(x : (A|B)), (y : (A|B))\}$.

But then `(x,y)` in the function body has type `((A|B),(A|B))` which is not a subtype of `(A,B)|(B,A)`. Therefore, the above programs fails to type check.

The problem of imprecise pattern inference is well-known [35]. We can offer a solution by mixing static with dynamic type checking. Like in the example above, we generate an up-cast coercion u_2 out of the subtype proof obligation $\vdash ((A|B),(A|B)) \leq^{u_2} ((A,B)|(B,A))$ where we use "error" cases to fix failed subtype proofs. This means that application of coercion u_2 potentially leads to a run-time failure. In fact, for our example we know there will not be any run-time failure because either case (1) or (2) applies.

For the above example, we additionally need to fix the subtype proof $\vdash ((A|B),(A|B)) \leq ((A,B)|(B,A))$ resulting from the pattern match check. This check guarantees that the pattern type is a subtype of the incoming type. Out of each such subtype proof we compute a down-cast coercion to perform the pattern match. In case of $\vdash A \leq B$ the pattern match should clearly fail. We can apply the same method for fixing up-cast coercions to also fix down-cast coercions. Each failed subtype proof is simply replaced by an "error" case. The pattern match belonging to the failed subtype proof $\vdash A \leq B$ is fixed by generating

```
\x -> error "run-time failure: we can't pattern match A against B"
```

In our case, we fix $\vdash ((A|B),(A|B)) \leq ((A,B)|(B,A))$ by generating

```
d2 :: Or (A,B) (B,A) -> Maybe (Or A B, Or A B)
d2 (L (a,b)) = Just (L a, R b)
d2 (R (b,a)) = Just (R b, L a)
```

Notice that there are no "error" and not even any "Nothing" cases because each of the two components of the incoming type $(A,B)|(B,A)$ fits into the pattern type $((A|B),(A|B))$.

7.1.5 GHC As a Library

One of the critical factors for the acceptance of any language extension is the availability of library support and how much of the existing code base can be re-used. XHaskell supports a module system and makes use of GHC-as-a-library to process Haskell modules which are imported by a XHaskell program. We make use of these features in the application below.

```
module RSStoXHTML where


import IO           -- Haskell IO module
import RSS          -- RSS XHaskell module generated by dtdToxhs rss.dtd
import XHTML        -- XHTML module genereated by dtdToxhs xhtml.dtd
import XConversion -- XHaskell module defining parseXml and writeXml etc


filepath1 = "rss1.xml"
filepath2 = "rss2.xml"


row :: (Link, Title) -> Div
row (Link link, Title title) =
    Div ("RSS Item", B title, "is located at", B link)


filter_rss :: Rss -> Div*
filter_rss rss = [ (row (l,t)) | (Item ( (t :: Title)
                                       , (ts :: (Title|Description)*)
                                       , (l :: Link)
                                       , rs )) <- rss/Channel/Item ]


main :: IO ()
main =  do (rss1 :: Rss) <- parseXml filepath1
           (rss2 :: Rss) <- parseXml filepath2
```

```
          let filter_rss1 = filter_rss rss1
              filter_rss2 = filter_rss rss2
          html = Html (Body
                 (I ("This document is generated by RSStoXHTML convertor, \
                     a program written in XHaskell.")
                 , Hr, filter_rss1, filter_rss2))
          writeXml "myrss.xhtml" html
```

As we mentioned before, our implementation comes with a tool called `dtdToxhs` which we use here to automatically generate XHaskell data types from the RSS and XHTML DTD specifications, for example `RSS`, `Link`, `Title`, `Div` etc. We can then import these data types into our main application. Another XHaskell module `XConversion` provides two functions `parseXml :: String -> IO Rss` to read and validate the RSS (XML) document and `writeXml :: Xhtml -> IO ()` to store the XHTML values into a (XML) file. We read and print from standard I/O. Therefore, we import the Haskell module `IO`. We make use of GHC-as-a-library to extract type information out of the imported Haskell module `IO`. We use this information to type check and translate the XHaskell program parts.

Function `filter_rss` extracts all `Item` elements out of the RSS document. For each `Item` element we call function `row` to generate an XHTML `Div` element which has the title and the link of this item. We make use of XQuery and XPath-style combinators to extract the immediate child elements of type `t` in expression `e`. As discussed earlier, we can de-sugar these combinators in terms of plain XHaskell. The main function finally generates an XHTML document in which part of the body content is generated using function `filter_rss`. For instance, given the input file `rss1.xml` as follows,

```
<rss>
 <channel>
   <item>
     <title>XHaskell</title>
```

```
      <link>http://www.comp.nus.edu.sg/~luzm/xhaskell</link>
    </item>
  </channel>
</rss>
```

and `rss2.xml` as follows,

```
<rss>
  <channel>
    <item>
      <title>Haskell</title>
      <link>http://www.haskell.org/</link>
    </item>
  </channel>
</rss>
```

executing the program RSStoXHTML yields the following XHTML document,

```
<html>
  <body>
   <i>This document is generated by RSStoXHTML convertor,
      a program written in XHaskell.</i>
   <hr/>
   <div> RSS Item
     <b>XHaskell</b> is located at <b>http://www.comp.nus.edu.sg/~luzm/xhaskell</b>
   </div>
   <div> RSS Item
    <b>Haskell</b> is located at <b>http://www.haskell.org</b>
   </div>
  </body>
</html>
```

7.1.6 Integration with HaXML

HaXML has been popular in the Haskell community in providing XML manipulation facilities. There have been quite a number of applications written in HaXML.

XHaskell programmers are allowed to reuse legacy codes written in HaXML. To allow for easier integration of XHaskell with HaXML legacy code, we provide two XHaskell library functions `toHaXml` and `fromHaXml` to convert data from its XHaskell type representation to HaXml type representation and vice versa.

In the following example, we incorporate some HaXML legacy code into the `RSStoXHTML` program given in the last sub-section.

Example 62 Suppose that `haxml_row` is a HaXml legacy function which generates a `Div` element out of a `Link` element and a `Title` element. Then we can redefine the function `row` from above as follows.

```
import MyHaXmlLib (haxml_row)
row' :: (Link, Title) -> Div
row' x = fromHaXml (haxml_row (toHaXml x))
```

□

7.2 XHaskell Applications

7.2.1 XML Processing

One of the main applications of XHaskell is XML processing. XHaskell equips programmers with advanced language features such as regular expression types, regular pattern matching, paramtric polymoprhism and type classes, which have not been implemented in a single language or system in the past. This makes the XHaskell language fit perfectly in any XML application development.

We have already gone through a lot of XML processing examples in this thesis, such as the address book example, the library example and the RSS-to-XHTML example. Yet there are many more real-world XML applications implemented in XHaskell unmentioned here, which can be found in the XHaskell homepage [74].

7.2.2 Parser Combinators

As we mentioned earlier, XHaskell is not restricted in XML processing. In the next application, we show that XHaskell is highly useful in parser writing. Suppose we would like to write a parser for Bibtex documents. A Bibtex document is a text file consisting of a sequence of entries. A Bibtex entry looks like the following,

```
@InProceedings{HaXML,
   author = {M. Wallace and C. Runciman},
   title = {Haskell and XML: Generic Combinators or Type-Based Translation?},
   booktitle = {ICFP ’99},
   publisher = {ACM Press},
   pages = {148-159},
   year = {1999}
}
```

a proceeding entry consisting of a key, an author, a title, a booktitle, the name of the publisher, the page index of the paper and a year. Note that some of the fields are optional. Thus, naturally we use the following XHaskell data type to encode an `InProceedings` entry,

```
data InProc = InProc Key Author Title Btitle? Year? Pages? Pub?
data Key = Key String
data Author = Author String
data Title = Title String
data Btitle = Btitle String
data Year  = Year String
data Pages = Pages String
data Pub = Pub String
```

We employ a parsing techinque, called monadic parser combinators, which has been studied in [37]. For instance, we declare a monadic parser data type as follows,

```
data Parser a = Parser (String -> [(a,String)])


instance Monad Parser where
    -- return :: a -> Result a
    return = let f :: a -> Result a
                 f (x::a) = Succ x
             in f
    -- (>>=) ::  Result a -> (a -> Result b) -> Result b
    (>>=) = let f :: (Result a) -> (a -> Result b) -> Result b
                f p g = case p of
                        (Succ (x :: a))      -> g x
                        (Err (s :: [Char]))  -> Err s
            in f
```

For example, a parser that parses a Author field can be specified as follows,

```
author :: Parser Author
author = do { (_::String) <- parse_string "author"
            ; (_::String) <- parse_string "={"
            ; (s :: String) <- everythingUntilString "}"
            ; (return :: Author -> Parser Author) (Author s)
            }
```

where `parse_string` is a helper function that parses a given string, `everthingUntilString` consumes everything until the specified string. In a similar way we can define the other parser combinators

```
title :: Parser Title

btitle :: Parser Btitle

year :: Parser Year

pages :: Parser Pages

pub :: Parser Pub
```

The novelty of this application is the mixing of regular expression types and parser combinators, which allows us to specify the composition of parser combinators.

```
star :: Parser a -> Parser a*

choice :: Parser a -> Parser b -> Parser (a|b)
```

The `star` combinator allows us to build repetition. For instance, `(star author)` denotes a parser that parses a sequence of `Author`s. The `choice` combinator allows us to build a choice parser. For example, `(choice year btitle)` denotes a parser that parses either a `Year` or a `Btitle`.

With all these combinators ready, we can describe a complex parser that parses an `InProceedings` entry.

```
inproceedings :: Parser InProc

inproceedings =

  do { (header::String) <- parse_string "@inproceedings{"

     ; (keys :: String) <- key ; (auth :: Author) <- author

     ; (titl :: Title)  <- title

     ; (chunk :: (Btitle|Year|Pages|Pub)*)

          <- star (btitle `choice` year `choice` pages `choice` pub) -- (1)

     ; case chunk of                                                 -- (2)

        (bt :: Btitle , yr :: Year, pg :: Pages, pub :: Pub) ->

           do { (cp :: String) <- parse_string "}"

              ; return (InProc (Key keys) auth titl bt yr pg pub) }
        (bt :: Btitle , yr :: Year, pub :: Pub, pg :: Pages) ->
           do { (cp :: String) <- parse_string "}"
              ; return (InProc (Key keys) auth titl bt yr pg pub) }
      ...
  }
```

The first three statements in the function body parse the header, the key, the author and the title. The remaining items are slightly harder to parse., since the fields, year, book title, publisher and pages, may appear in any order. Therefore, at `(1)` we use the composition of `star` and `choice` to parse everything remaining as a `chunk`. At `(2)` we use regular expression pattern matching to perform a case analysis.

In short, regular expression types, regular expression pattern matching and parser combinators make a perfect match. Regular expression types and parre combinators allow us to specify the complex parsing routines in simple syntax. Regular expression pattern matching helps us in analyzing and extracting the parsed data in a concise way.

7.3 Summary

We have discussed the design decisions which we made in the implementation of the XHaskell system. We have shown that XHaskell is not solely designed for XML proccessing. The combination of regular expression types and pattern matching seems to be useful in other applications such as building complex parser combinators.

Chapter 8

Related Work and Discussion

In this chapter, we study the related work and provide detailed discussion.

In Section 8.1, we give a literature survey on languages and systems that support XML processing. In Section 8.2, we juxtapose our regular expression pattern compilation scheme with other compilation schemes. In Section 8.3, we compare various XML value encoding schemes. In Section 8.4, we have a discussion on regular expression type and parametric polymorphism. In Section 8.5, we summarize different techniques on regular expression pattern inferences. In Section 8.6, we review some related work on adding subtyping to functional languages.

8.1 Related Works

First, we review various programming languges and systems that support XML processing.

8.1.1 XDuce and CDuce

XDuce Hosoya and Pierce pioneered the work, XDuce [30, 35], a type-safe functional language for XML processing. XDuce has native support for XML values. Regular expression type and pattern matching were first introduced in this work.

Regular expression type resembles XML DTD directly. Regular expression pattern allows programmer to express sophisticated XML transformation in a concise fashion. Further more, the XDuce type system guarantees that well-typed programs will generate well-formed and valid XML documents. XDuce is implemented in the form of an interpreter.

In [32], Hosoya introduced the regular expression filter. The regular expression filter is an advanced language construct in XDuce that allows programmers to define generic traversal and transformation of XML documents. This powerful feature operates in a flavor of polymorphic functions, such as `map` and `filter`, found in XHaskell. It is a simple solution towards generic XML programming without getting into the issues of combining regular expression types with parametric polymorphism, which will be discussed in Section 8.4.

Note that there is an extension of XDuce [33] that support parametric polymorphism, we defer the discussion to Section 8.4.

CDuce The CDuce language [7, 21] extends XDuce with higher-order function and function overloading. The CDuce system includes an interpretor and a compiler. CDuce provides an advanced compilation scheme for regular expression pattern matching. Experiemental results show that its run-time performance is faster than XSLT. Like the regular expression filter, CDuce also provides several language constructs, such as `map`, `transform` and `xtransform`, to support generic XML processing. On the other hand, parametric polymorphism is not available and user-defined polymorphic function is not supported by CDuce.

One of the novel features of CDuce is function overloading. For example, consider the following CDuce program,

```
type MPerson = MPerson
type FPerson = FPerson
type Male = Male
type Female =  Female

let fun f (MPerson -> Male ; FPerson -> Female)

   (x :: MPerson) -> Male

   (x :: FPerson) -> Female
```

Just like in XDuce, In CDuce the `type` keyword introduces a new datatype. Function `f` has two type signatures, `MPerson -> Male` and `FPerson -> Female`. There are two patterns in the function body. The first pattern applies when the input is a male person, `MPerson`, a value of `Male` type is returned. The second pattern applies when the input is a female person, `FPerson`, a value of type `Female` type is returned. We say function `f` is overloaded because it returns different types of results depending on the type of the input value.

In XHaskell, we can encode the above via a type class as follows,

```
data MPerson = MPerson

data FPerson = FPerson

data Male = Male

data Female =  Female


class F a b where

  f :: a -> b


instance F MPerson Male where

  f (x :: MPerson) = Male


instance F FPerson Female where

  f (x :: FPerson) = Female
```

In the above, the type class `F` describes valid relations among type `a` and type `b`. There are two instances of `F`. In the first instance, function `f` takes a value of type

MPerson and returns a value of type Male. On the other hand, in the second instance, function f takes a value of type FPerson and and returns a Female.

The above XHaskell program behaves almost the same as the CDuce program we define earlier. Except that XHaskell type class implements the "open world model", while the CDuce function overloading implements the "close world model". For instance, we can extend the above XHaskell type class with a new instance as follows,

```
data Foo = Foo
data Bar = Bar


instance F Foo Bar where
  f (x :: Foo) = Bar
```

In CDuce, it is not easy to extend an overloaded function with new instances.

CDuce has a richer type system which allows for subtyping among higher-order type which is not supported by XHaskell for the moment. This is because we cannot find a proper translation for the downcast pattern matching of function type, which we have already discussed in Chapter 5.

CDuce does support parametric polymorphism.

8.1.2 XML Processing in ML and Haskell

There are several projects of enhancing ML and Haskell with XML processing capability.

HaXml In [70], Wallace and Runcimann brought XML and Haskell together, by studying two different XML encoding schemes in Haskell.

In one scheme, XML values are represented in terms of some *uniform* data types. For instance, we use the following data types to represent XML documents

```
data Element = Element Name [Content]

data Name = Name String

data Content = CE Element | CT String
```

Elements `<A/>` and `<B/>` are represented as `Element (Name "A") []` and `Element (Name "B") []`.

In this scheme, the type information of the XML document is not captured by the encodings. Functions which manipulate these data types do not need to respect the type of the XML document. As a result, by using this encoding scheme, we can only guarantee the resulting XML document is well-formed but we cannot gaurantee it is valid.

In the other scheme, XML values are represented as *type-specific* data types. For instance, consider

```
data A = A String

data B = B String
```

Elements `<A/>` and `<B/>` are represented as `A ""` and `B ""`. In this encoding scheme, the schema information of the XML documents is embedded into the type system of the host language. For example, we use the above data type to represent a DTD as the following,

```
<!ELEMENT A (#PCDATA)>

<!ELEMENT B (#PCDATA)>
```

As a result, the validity of program output is guaranteed by type system in the host language Haskell.

In XHaskell, we leverage on the second encoding scheme and extend it with regular expression type. With the native support of regular expression type and regular expression pattern (which are missing in HaXml), we are able to express XML transformation in a more concise way.

Harp The Harp project [12] is a preprocessor of Haskell that supports regular expression pattern matching over Haskell list values. Nevertheless, the pattern matching is limited to list values. Furthermore, the type system is not extended to support regular expression type, thus, the system provides no static guarantee to the result of pattern matching.

UUXML In [19] (and later [6]), Atanassow, Clarke and Jeuring study the method of modelling XML Schema using Haskell data bindings. Since XML Schema is more expressive than XML DTD, the problems imposed in this work are more challenging. For instance, they address the issue of how to model Schema Derivative (a kind of nominal subtyping). But they do not directly deal with semantics subtyping as what we do in XHaskell. Their ideas are really novel, we plan to adopt some of their ideas when we consider supporting XML Schema in XHaskell as future works.

In a follow up work, Guerra, Jeuring and Swierstra present a typed based approach to validate an XPath with respect to the XML schema [29]. This permits a type-safe embedding of XPath expression into a general purpose functional language like Haskell. In this system, a valid function can be defined by induction on the type structure generically. The implementation relies on a Haskell extension, *Generic Haskell*. Comparing with our work, this work does not directly deal with regular expression type, instead, the reasonings are based on Haskell data-type representation of the XML schema (via HaXml or UUXML).

Haskell XML Toolbox The Haskell XML Toolbox is a collection of Haskell library extensions that support XML programming in Haskell. The Haskell XML Toolbox is based on the idea of HaXml (basically the universal encoding scheme). It introduces more generic ways of processing XML. For instance, it supports XPath-style child-/descendant-axis operations. It has an integrated XSLT transformer. Nevertheless, it has no support for regular expression types and pattern matching. Thus, the static guarantee provided by the system is limited.

Wash Thiemann [65] introduced WASH, a combinator library which can be used to generate XML values. He made use of the Haskell type class to check for correctness of constructed values. The gist of the ideas is to use type class context reduction to mimic the automata transition when testing for the validity of the XML document. However, this approach does not consider deconstruction of values.

HSXML Kiselyov developed a library extension for writing and transforming typed semi-structured data in Haskell, called HSXML [38]. In this approach, XML data are also represented as structured Haskell data type. The validity of the constructed document is enforced via type class constraints.

The main focus of the above-mentioned projects is to encode XML-specific types using types from the host languages. Thus XML type definitions can be expressed in terms of the type combinators. Programmers can define XML transformations as functions and procedures in the host language. The host languages are strictly typed, the validity of the program output is automatically ensured by type system. At the same time, they can easily make use of some existing libraries.

However, all these approaches exist in the form of libraries. They lack good (type) error support in general. In case the program fails to type check, the error message is phrased by some general typing information from the host type system. Therefore, the error message could be incomprehensible to the programmer. Furthermore, there is less space to optimize the XML transformation in these approaches. Lastly, the lack of native support for XML in the type system makes these approaches provide less static guarantee to the XML transformation as compared to languages such as XDuce, which we will describe shortly.

Pre-XHaskell In [47], Lu and Sulzmann explored the possibility of implementing regular expression subtyping in type classes by encoding Antimirov's regular expression containment algorithm in terms of type class constraints. This was the initial idea of XHaskell. The difference is their system is implemented as a library extension. Regular expression types are encoded as some singleton type. The subtype proof derivation reduces to the type class resolution problem which demands a coinductive type class extension.

$\mathcal{L}_{dtd}$ In [81], Zhu and Xi gave a type-safe encoding of XML documents using *Guarded Recursive Data type* (which is also known as GADT or EADT). In their approach, each XML value has two components. In the first component, a universal representation (similar to the first encoding scheme of HaXml) is used to store the value. In the second component, the additional (DTD or Schema) type information is captured as a dependent typed value. Well-typedness of the functions guarantees that the resulting values are well-formed and valid. Their system does not support regular expression subtyping nor regular expression pattern matching.

XMλ In [49], a functional language with direct support of XML is presented. The language is designed for processing XML in a type-safe manner. The language supports parametric polymorphism. On the other hand, they do not consider regular expression types nor regular expression pattern matching. The ideas of this work have been later adopted in the C_ω project (see below).

OCamlDuce In [22], Frisch introduced OCamlDuce, a merger of OCaml and XDuce. The main focus of his work is to develop a type inference algorithm to infer types for the OCaml components and most of the XDuce components using a global flow analysis. A significant difference in comparison with our work is that OCamlDuce strictly separates the XDuce components from the OCaml counterparts by syntax restriction. Hence, it is not possible to directly use OCaml polymorphism in XDuce programs.

8.1.3 XML Processing in Imperative Languages

We now look at some imperative languages and extentions that support XML processing.

XOBE XOBE [41, 40] is a language extension to Java. XOBE supports XML elements as first class values. XOBE's type system extends the Java type system to support regular expression subtyping. It is one of the first projects that extend Antimirov's algorithm (an algorithm of solving regular expression inequalities) to handle regular hedge. They also employed a source-to-source translation from XOBE to Java. The XML elements are represented as universal tree structures. Thus the use of subtyping in XOBE does not require insertion of coercion. On the other hand, XOBE does not support regular expression pattern matching. XML values in XOBE are deconstructed by XPath extractors.

Scala Scala [60] is another language extension to Java which supports algebraic data types and pattern matching. In [18], Emir extends the Scala pattern matching with regular expression patterns. We will conduct a comparison between Scala and XHaskell in Section 8.2.

Xact Xact [72] is a language extension to Java that supports XML processing in the style of XPath combinators. As studied in [16, 1], Xact relies on a data-flow analysis to provide static guarantees that the program output is valid if the program input is valid. In a follow up work, [15], the ideas are extended to support XML Schema and Relax NG.

Next we take a look at works that extends C♯ with type-safe XML processing features.

Xtatic Xtatic is a language extension that combines features from XDuce and C♯. In [26], Gapeyev and Pierce formalized a smooth integration of feather-weight Java and XDuce. Similar to XHaskell, Xtatic employs a source-to-source translation to C♯. Regular expression types are compiled into C♯ classes.

$\mathbf{C}_\omega$ Meijer, Schulte, Bierman proposed an XML extension to *C♯* by generalizing the type system [48]. With regular expression types built-in, XML values can be constructed and accessed as first class objects, and the static information can be reasoned about at compile time. In case of ambiguity, the programmer needs to provide extra annotation to resolve it, otherwise a compile time error is generated. The language does not support semantics subtyping. Value deconstruction relies on `foreach` statement and `(.)` operator, which yields a query-style language.

In a follow-up work [8], Bierman, Meijer and Schulte presented the core of the language C_ω, which is a novel extension to *C♯* supporting first-class type-based query for relational database and XML. The main ideas is to introduce stream type, anonymous struct type, choice type and generalized member access into a main-stream language like *C♯* and Java. The main contribution in this work is that their language stratification allows a free mixing of objects and XML/database values. They implemented subtyping in a limited way. For instance, in their framework, $\texttt{choice}\{\texttt{A},\texttt{B}\} <: \texttt{choice}\{\texttt{choice}\{\texttt{A},\texttt{C}\},\texttt{choice}\{\texttt{B},\texttt{D}\}\}$ (equivalent to $(A|B) \leq ((A|C)|(B|D))$ in XHaskell) cannot be verified. (Or `(A|B)` $\leq$ `((A|C)|(B|D))` in our syntax). Furthermore, their focus is on a query language extension, hence pattern matching is not considered. The compilation scheme of this language extension is based on source-to-source translation. The formal result states that the translation is type-preserving.

8.1.4 XML Query Language

XQuery [79] is a query language designed to query a collection of XML data from a XML document based on programmable criteria. Its syntax is very close to SQL (a query language for relational database). XQuery is a typed language. Queries and functions are guaranteed to generate well-formed and valid XML data. The XPath [78] is a language designed for extracting some sub-components from a XML structure. Most of the time it is used in combination with XQuery. In a later chapter of this thesis, we will show that XHaskell allows for XQuery and XPath style programming.

Xcerpt [73, 13] is a rule-based deductive query language for XML and other semi-structured data. Xcerpt allows programmers to specify XML query and transformation in the style of logic programming.

8.2 Regular Expression Pattern Compilation

In this thesis, we have presented a new compilation scheme for regular expression pattern matching. The novelty comes from the use of partial pattern derivative, which allows us to perform pattern matching based on term rewriting. We compare our compilation scheme with a few closely related works on this specific topic.

CDuce compiles regular expression patterns into non-uniform automata [20] which is a combination of top-down tree automata and bottom-up tree automata. As a result, the pattern matching algorithm takes into account the static information (thanks to the top-down automata) and eliminates backtracking (thanks to the bottom-up automata).

In [18], Emir extended the Scala language with regular expression patterns. In his work, regular expression patterns are compiled into sequential machines. Sequential machines can be viewed as finite automata with transition symbols annotated with pattern variables. Executing the sequence machine with a word yields the matching result.

The Xtatic research team has put in great effort in compiling regular expression pattern. In [44], Levin developed a compilation scheme for XDuce patterns by using a notion of *Matching Automata*. Matching Automata is a variant of tree automata in which we can easily identify sub-tests (or sub-automata) that are isomorphic. Therefore, we can easily combine them to minimize the number of tests performed on the input value. For instance, we consider the following pattern,

```
case x of                   -- (1)
 (A, (y::B)) -> y
 (A, (z::C)) -> z
```

The test "whether the first element in `x` is an `A`" will be executed twice if `x` does not match with the first pattern `(A, (y::B))`. With matching automata we realize that pattern `(1)` is equivalent to the following pattern,

```
case x of                   -- (2)
 (A, x') -> case x' of
         y :: B -> y
         z :: C -> z
```

in which the test of the leading `A` is not repeated. However their approach does not take into account the static information given by the input type of the pattern. In a follow-up work [46], Levin and Pierce proposed an extension of the above compilation scheme by incorporating the input type into the checking process. Given the input type which may denote a subset of the values being matched by the pattern, we can remove unused tests. This is also known as *regular expression pattern inference* for which we will give a discussion shortly.

All these optimization techniques on regular expression pattern matching are surely great. On the other hand, we find that our system gets most of this optimization for free. In our system, we use downcast coercions to compile regular expression pattern matching. With the help of pattern inference, we always find that the pattern's type is a subtype of the function input type. The downcast coercion is derived from the subtype relation amongst the pattern type and the input type. Therefore, the static information has already been propogated into the coercion. Furthermore, there have been a lot of well-studied existing techniques on Haskell pattern matching optimization. We can reuse them by translating regular expression pattern matching into Haskell pattern matching. In particular, we translate XHaskell patterns to Haskell patterns and leave some optmization task to the backend compiler GHC. For instance, as we can observe from the intermediate *GHC core* code generated by the GHC compiler (with compiler flags `-O2 -fext-core`), the pattern `(1)` above is compiled to the following (abridged) GHC core code,

```
case x of
  (A, x') -> x'
```

Note that GHC uses aggressive inlining, thus the downcast coercions and the intermediate data structures are compiled away.

In some cases, we even get better optimized code compared to any of the above-mentioned systems. For instance, In a recent work [52], Peyton-Jones presented an optimization technique by specializing recursive function calls according to the shape of the input. For example, we recast one of his examples in XHaskell as follows,

```
last :: a* -> a
last (_ :: ()) = error "The sequence is empty!"
last (x :: a) = x
last (x :: a, xs :: a+) = last xs
```

Function `last` retrieves the last item in a sequence. However such an implementation is not very efficient. In the third pattern clause, we apply `last` recursively to `xs` which is of type `a+`. It forces us to check `xs` against the first pattern `(_ :: ())` though it will of course never apply. According to the proposed solution in [52], the above function should be re-written as follows,

```
last :: a* -> a
last (_ :: ()) = error "The sequence is empty!"
last (xs :: a+) = last' x xs
  where last' :: a+ -> a
        last' (x :: a) = x
        last' (x :: a, xs :: a+) = last' xs
```

where the recursive call is specialized as `last'`. Thus the redundant test is eliminated.

As we said, we do not need to implement this optimization technique in XHaskell. By using GHC as the back end compiler, we get this optimization for free. To the best of our knowledge, this technique has not been mentioned in any of the systems above.

However, we also note that all the above-mentioned approaches deal with regular hedges, which is not supported by our system. We believe our approach can be extended to regular hedges. Most of the practical examples we have encountered so far can be implemented using regular expression data types, without sacrificing the conciseness of the programs.

Lastly, we found some related work implementing a regular expression pattern matching interpretor. In [23], Frisch and Cardelli studied the regular expression pattern matching problem in the context of using structured representation as run-time values. The goal of their work is to provide a language extension to main stream languages such as Java and C$\sharp$ with regular expression pattern matching. They adopted an automata based approach. They showed that their implementation has linear time complexity with respect to the size of the input word and the pattern. In contrast, our interpretor implementation $posixMatch$ proposed in Chapter 6 has a linear time complexity, too. This is not a surprise because the height of the search tree is bounded by the size of the pattern and the size of input word. Under POSIX/Longest matching, we obtain the matching result by traversing the tree from root to the first successful leaf node on the left most. Assuming we apply aggressive pruning on the search tree, the time taken to find the matching result is proportional to the height of the search tree.

8.3 Run-Time Encodings of Semi-Structured Value

XHaskell is strongly inspired by HaXML and uses type-specific encoding for values of regular expression types. This simplifies the compilation scheme of regular expression pattern matching and smoothens the integration with existing libraries.

Opposed to our work, CDuce adapts a uniform run-time representation for XML values. Therefore, CDuce run-time values do not carry any structure. Semantic subtyping has no effect on the run-time values.

Like CDuce, Xtatic employs a universal run-time representation for XML values. For instance, a sequence of `A` is translated into a linked-list object. For each element in the linked-list, a specific field is used to store the source type information. In this case the field identifies the element is of type `A`. Optimization techniques such as [45] are needed to eliminate run-time value tests in Xtatic. On the other hand, our work avoids run-time value testing by using structure representation for XML values. The run-time value has enough structure which can be used to identify its type.

In [25], Gapeyev et al. were investigating an alternative runtime representation for Xtatic XML values. One interesting choice is to use lazy linked-lists instead of ordinary (eager) linked-lists as the run-time representation of sequences. Their experiment shows that lazy run-time representation may improve the run-time performance. There are some cases where the improvements are significant as only part of the input sequence is needed to compute the output. In our system, we use Haskell as the target language. By default the language uses lazy evaluation and uses linked-list to represent sequence. Of course, we can also switch to the strict evaluation strategy by annotating the run-time representation data structure with the strict symbol `!` which is available in Haskell.

8.4 Regular Expression Types and Polymorphism

There is no doubt that the combination of regular expression types and parametric polymorphism is highly useful, exciting and challenging. There are two main issues arising.

First of all, adding polymorphism naively to XDuce leads to undecidable type inference. For example, we consider the following program

```
f :: (a|b) -> Int
f (x :: a) = 1
f (x :: b) = 2
```

Applying `f` to a value of type (`A`|`B`) yields two possible type substitutions, namely (1) { `(A/a), (B/b)` } or (2) { `(A/b), (B/a)` }. Second, it is hard to guarantee the compiled code is faithful under substitution. For example, we consider the following program,

```
g :: (c|A) -> Int
g (x :: c) = 1
g (x :: A) = 2
```

Depending on how `c` is instantiated, the behavior of `g` differs. Suppose we send `c` to `A`. The application `(g A)` always returns `1` as result. Otherwise, it is clear that `(g A)` should yield `2`.

At this stage we are only aware of two projects that address the above issues.

In [33], Hosoya, Frisch and Castagna showed how to integrate parametric polymorphism into XDuce. Their idea is to only accept programs which do not make any specific assumptions about polymorphic variables. Under this condition they can adopt the existing simple semantics of XDuce. Thus, they can deal with programs whose polymorphic variables are guarded by an XML tag. But they generally cannot deal with programs which require polymorphic types of the form $(a|b)$, a^*

etc. It is obvious that problematic programs like the two stated above as well as `mapStar` are ruled out by the syntactic restriction.

In some follow-up work [67, 68], Vouillon studied a system of polymorphic regular tree types and patterns. His work improves over the work in [33] by introducing a variant of System F with regular hedges. In his language polymorphic regular expression type such as $(a|b)$ is allowed. The ambiguity of type instantiation is resolved by demanding explicit type application. Furthermore, he established several conditions, which can be used to analyze statically whether a faithful compilation exists for a given program. However, his analysis is still too restrictive to reject many useful programs such as `mapStar`.

Similar to Vouillon's solution, we demand explicit type application in XHaskell to address the first problem. However, our current formulation is too liberal by ignoring the faithfulness issue of parametric polymorphism, namely the second problem. (Note that the faithfulness result in Chapter 6 only covers monomorphic programs.) To establish polymorphic faithfulness without rejecting useful programs such as `mapStar`, a refined checking technique is needed. We believe that the static analysis should be performed at the function application sites instead of the function definition sites. The checking procudure should take type application into account. For instance, in our system the function `g` above will be translated as follows,

```
g :: (Or c A) -> Int
g (L _) = 1
g (R _) = 2
```

For simplicity, we inline the downcast coercion which the backend GHC compiler eventually does. The argument here is that the above compiled code will only behave unfaithfully when we instantiate `c` with `A`. As we discuss earlier, we should expect the result to be `1`, regardless of the input having target language representation `(L A)` or `(R A)`. However, this is not true given the above translation. On the other hand, `g` is safe to used under the type instance which sends `c` to `B`. Therefore, we should just accept the above translation of `g`, and reject any application of `g` which instantiates `c` with `A`. We plan to pursue this idea in the near future.

8.5 Regular Expression Pattern Inference

In [31], Hosoya proposed a simpler design for XDuce's regular expression pattern matching. In particular, he formalized a pattern inference algorithm based on non-deterministic pattern matching policy. His algorithm can be viewed as an extension of the product automata which recognizes the intersection of two regular languages. He extends the product automata with variable bindings which captures the inference results.

In [66] Vansummeran studied the pattern inference problem for regular expression patterns. He formalized three pattern matching policies and showed the difference amongst them. Then he gave an algorithm of pattern inference for each pattern matching policy. In our implementation, we adopt Vansummeran's POSIX pattern inference algorithm.

8.6 Subtyping and Functional Languages

In [10], Brandt and Henglein presented a sound and complete axiomization of type equality and subtype inequality for first order typed language with recursive types. Their work shows that coinductive proof technique is needed to verify subtyping proof in the presence of recursive types.

Kieβling and Luo presented an extension to Hindley-Milner Systems with coercions [42]. The main objective of this work is to make certain un-typable terms in HM system become typable. The main idea is to look up the appropriate coercion function when there is a type mismatch. In contrast to our work, their coercion functions are statically declared, and the constructions of these functions are considered external to the system. With the presence of polymorphism, their coercion derivation may not be unique and the coherence of the program may be lost. In case of ambiguity, the system rejects the program. The authors also pointed out that transitivity (i.e., coercion composition) can be added to the coercion derivation. However the decidability of coercion search with transitivity is unclear.

Norlander introduced a variant of Haskell, O'Haskell [51], which supports nominal subtyping and parametric polymorphism whereas we support semantic subtyping and parametric polymorphism. To our knowledge his system does not support type class overloading. On the other hand, his system supports explicit subtype constraints in function signatures which we currently disallow.

In some earlier work [2], Aiken and Murphy formalized a system that implements regular tree expression. The regular tree expression system was then used in reasoning with a dynamic typed language [3].

Our system adds regular expression subtyping to Haskell (or ML). Our subtype algorithm is an extension of Antimirov's work [5, 4], in which he uses partial derivative as the basic operation in solving regular expression equalities and inequalities.

Chapter 9

Conclusion And Future Work

Writing XML transformation using untyped language such as XSLT is error-prone. Regular expression type and regular expression pattern matching are two silver bullets for writing type-safe XML transformation. Existing languages and systems which implement regular expression type and regular expression pattern matching are mostly domain specific. Programmers are restricted to limited access of existing standard libraries. There are some approaches that try to bridge the gap. However they all exist as library extensions and lack of strong static guarantee and type error reporting.

In this thesis we present XHaskell, a language extension of Haskell, that combines regular expression type and pattern matching with algebraic data type, parametric polymorphism and ad-hoc polymorphism. In XHaskell, programmers are able to write XDuce-style transformation via regular expression patterns. At the same time programmers still have access to all existing libraries provided by Haskell. In the fusion of subtype polymorphism, parametric polymorphism and ad-hoc polymorphism, programmers are allowed to specify XML transformation program in the style of XQuery and XPath. XHaskell is not solely designed for XML processing. For instance, one of our applications shows that regular expression types and pattern matching can be used in combination with monadic parser combinators to build advanced parsers.

In this thesis, we have formalized the core language of XHaskell, System F* which is an extension of System F with regular expression types and pattern matching. We have developed a source-to-source translation scheme from System F* to System F. The translation relies on a constructive interpretation of semantic subtyping. We employed Curry-Howard isomorphism to derive coercion functions from the subtype proof. The upcast coercion is used in translating semantic subtyping. The downcast coercion is used to translate regular expression pattern matching. Based on the result that the upcast coercion does not change the semantics of the result, we proved that our translation scheme is coherent.

We had an in-depth discussion of the regular expression pattern matching problem. We studied various techniques of implementing regular expression pattern matching. We proposed a novel algorithm that solves the regular expression pattern matching problem based on derivative operation. On the other hand we developed a specialized regular expression pattern matching algorithm which operates on a smaller set of inputs. We called this algorithm the coercive pattern matching algorithm. We showed that the algorithm is faithful with respect to the pattern matching relation under the POSIX/Longest matching policy. It follows that our translation scheme is faithful under the POSIX/Longest matching policy.

We believe that using structured data to represent values of regular expression type at run time have several advantages over systems that use uniform representation. First of all, the run-time values carry enough structure which resembles the type information on the source level. It costs us little effort to develop a compilation scheme for regular expression pattern matching whose semantics is type-dependent. Furthermore, the same compilation scheme extends to polymorphic programs without making any adjustment.

9.1 Future Work

The framework developed in this thesis supports the combination parametric polymorphism and regular expression type. As we mentioned in Section 8.4 Chapter 8, there is still an interesting issue of polymorphic faithfulness. We have some initial ideas of addressing this issue without imposing severe restrictions on the language. This will be the immediate step we are going to take.

As we have stepped into the multi-core era, we would like to adapt our software system so that it scales well with a multi-core system. This is definitely applicable to XML processing applications. For example, in a recent work [64], Sun et. al. study the problems of parallelizing XML transformation in the context of XSLT. In our system, we are interested in extending our compilation scheme to support an automatic generation of parallel programs. One obvious parallelizable program location lies in a pattern that mixes data type pattern and regular expression pattern. Consider the following example,

```
data T = T A*
data A = A


f :: T* -> A*
f (T (x::A+), y :: T*) = (x,f y)
f (T (x::()), y :: T*) = f y
f () = ()
```

The first pattern applies if the incoming sequence is starting with a non-empty `T`, and the remainder is a sequence of `T`. The pattern matching routine can be broken down to two independent sub-tasks. That is, (1) checking whether the leading element in the value is a `(T A+)`, and (2) checking the remaining element is a sequence of `T*`. Note that these two tasks can be perform concurrently, since neither one depends on the result of the other. A less obvious program location for parallelization can be found in a sequence pattern whose sub-sequences are clearly separated. Consider the example,

```
g :: (A*,B*) -> (A+,B+)
g (x :: A+, y :: B+) = (x,y)
```

Note that the input is a sequence of `A`s followed by a sequence of `B`s. If we clearly know where to break the sequence into `A`s and `B`s, because the run-time value has a pair structure, we can again perform the pattern matching of `( x :: A+)` and the pattern matching `(y :: B+)` in parallel. It gets harder to check for parallelization opportunity when the sequence pattern becomes more complicated. We believe that there exists a systematic approach to identify the right parallelizable program location. This is the second item on our to-do list.

The XHaskell prototype currently relies on an external XML parser provided by [70]. This incurs a lot of overhead during run-time because of the data conversion between XHaskell and HaXml representations of XML data. In Section 7.2, we showed how to write a parser in XHaskell using regular expression type, pattern matching and monadic parser combinators. This suggests that we should write an XML parser for XHaskell applications natively in XHaskell. Previous work [37, 43, 55] show that naive implementation of parser combinators does not perform well in practice and therefore some special treatment is needed for optimization. We plan to pursue further along their ideas.

The current XHaskell prototype generates Haskell codes from XHaskell source programs. Users are expected to invoke the GHC compiler to compile these generated Haskell codes manually to obtain the final executables. In future, we shall make use of GHC-as-a-library to generate binary executable directly from XHaskell source programs.

Bibliography

[1] A. Møller A. S. Christensen, C. Kirkegaard. A runtime system for XML transformations in java. In *Proc. of XSym 2004*, pages 143–157, 2004.

[2] A. Aiken and B. R. Murphy. Implementing regular tree expressions. In *Proc. of FPCA'91*, volume 523 of *LNCS*, pages 427–447. Springer, 1991.

[3] A. Aiken and B. R. Murphy. Static type inference in a dynamically typed language. In *Proc. of POPL'91*, pages 279–290. ACM Press, 1991.

[4] V. M. Antimirov. Rewriting regular inequalities. In *Proc. of FCT'95*, volume 965 of *LNCS*, pages 116–125. Springer-Verlag, 1995.

[5] V. M. Antimirov and P. D. Mosses. Rewriting extended regular expressions. *Theoretical Computer Science*, 143(1):51–72, 1995.

[6] F. Atanassow and J. Jeuring. Customizing an xml-haskell data binding with type isomorphism inference in generic haskell. *Science of Computer Programming*, 65(2):72–107, 2007.

[7] V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-centric general-purpose language. In *Proc. of ICFP '03*, pages 51–63. ACM Press, 2003.

[8] G. Bierman, E. Meijer, and W. Schulte. The essence of data access in c_ω. In *Proc. of ECOOP 05*, pages 287–311. Spring-Verlag, 2005.

[9] R. S. Bird and L. G. L. T. Meertens. Nested datatypes. In *Proc. of the Mathematics of Program Construction*, LNCS, pages 52–67. Springer-Verlag, 1998.

[10] M. Brandt and F. Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundam. Inf.*, 33(4):309–338, 1998.

[11] V. Breazu-Tannen, T. Coquand, C. A. Gunter, and A. Scedrov. Inheritance as implicit coercion. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 197–245. The MIT Press, Cambridge, MA, 1994.

[12] N. Broberg, A. Farre, and J. Svenningsson. Regular expression patterns. In *Proc. of ICFP'04*, pages 67–78. ACM Press, 2004.

[13] F. Bry and S. Schaffert. Towards a declarative query and transformation language for xml and semistructured data: Simulation unification. In *Proc. Int. Conf. on Logic Programming.* LNCS, (2401), Springer-Verlag, 2002.

[14] J. A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM (JACM) archive*, 11(4):481–494, 1964.

[15] A. Møller C. Kirkegaard. Type checking with xml schema in XACT. In *Proc of PLAN-X 2006*, pages 14–23, Charleston, South Carolina, USA, 2006.

[16] M. I. Schwartzbach C. Kirkegaard, A. Møller. Static analysis of xml transformations in java. *IEEE Transaction Software Engineering*, 30(3):181–192, 2004.

[17] Document type definition. http://www.w3.org/TR/REC-xml/♯dt-doctype.

[18] B. Emir. Compiling regular patterns to sequential machines. In *Proceedings of ACM Symposium on Applied Computing, Programming Languages Track Santa Fe*, New Mexico, 2005.

[19] D. Clarke F. Atanassow and J. Jeuring. UUXML – a type-preserving XML schema-haskell data binding. In *Proc of Practical Aspects of Declarative Programming (PADL)*, Dallas, Texas, USA, June 2004.

[20] A. Frisch. Regular tree language recognition with static information. In *Proc. of (TCS 2004)*, 2004.

[21] A. Frisch. *Théorie, conception et réalisation d'un langage adapté à XML*. PhD thesis, I'Université Paris 7-Denis Diderot, Dec 2004.

[22] A. Frisch. OCaml + XDuce. In *Proc. of ICFP'06*, pages 192–200. ACM Press, 2006.

[23] A. Frisch and L. Cardelli. Greedy regular expression matching. In *Proc. of ICALP'04*, pages 618– 629. Spinger-Verlag, 2004.

[24] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping. In *Proc. of LICS '02*, pages 137–146. IEEE Computer Society, 2002.

[25] V. Gapeyev, M. Levin, B. Pierce, and A. Schmitt. Xml goes native: Run-time representations for xtatic, 2004.

[26] V. Gapeyev and B. C. Pierce. Regular object types. In *ECOOP '03*, volume 2743 of *LNCS*, pages 151–175. Springer, 2003. A preliminary version was presented at FOOL '03.

[27] Glasgow haskell compiler home page. http://www.haskell.org/ghc/.

[28] J. Girard. *Interpretation fonctionnelle et elimination des coupures de l'arithmetique d'ordre superieur*. Thèse d'état, Université Paris 7, June 1972.

[29] R. Guerra, J. Jeuring, and S. D. Swierstra. Generic validation in an XPath-Haskell data binding. In *PLAN-X '05 Informal Proceedings*, 2005.

[30] H. Hosoya. *Regular Expression Types for XML*. PhD thesis, The University of Tokyo, December 2000.

[31] H. Hosoya. Regular expression pattern matching - a simpler design -, Feb 2003.

[32] H. Hosoya. Regular expression filters for XML. *Journal of Functional Programming*, 16(6):711–750, 2006.

[33] H. Hosoya, A. Frisch, and G. Castagna. Parametric polymorphism for XML. In *Proc. of POPL'05*, pages 50–62. ACM Press, 2005.

[34] H. Hosoya and B. C. Pierce. XDuce: A typed XML processing language (preliminary report). In *Proc. of Third International Workshop on the Web and Databases (WebDB2000)*, volume 1997, pages 226–244, 2000.

[35] H. Hosoya and B. C. Pierce. Regular expression pattern matching for XML. In *Proc. of POPL'01*, pages 67–80. ACM Press, 2001.

[36] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. *ACM Trans. Program. Lang. Syst.*, 27(1):46–90, 2005.

[37] G. Hutton and E. Meijer. Monadic parser combinators, 1996.

[38] Hsxml. http://okmij.org/ftp/Scheme/xml.html.

[39] Steven M. Kearns. Extending regular expressions with context operators and parse extraction. *Software - Practice and Experience*, 21(8):787–804, 1991.

[40] M. Kempa and V. Linnemann. The XOBE Project. In *Proceedings of the First Hangzhou-Lbeck Conference on Software Engineering*, pages 80–87, Hangzhou, P.R. China, November 1-2, 2003.

[41] M. Kempa and V. Linnemann. Type Checking in XOBE. In Gerhard Weikum, Harald Schning, and Erhard Rahm, editors, *BTW 2003, Datenbanksysteme fr Business, Technologie und Web, Tagungsband der 10. BTW-Konferenz*, volume P-26 of *Lecture Notes in Informatics (LNI)*, pages 227–246, Leipzig, Germany, 26.-28. Februar 2003. Gesellschaft fr Informatik (GI).

[42] R. Kierssling and Z. Luo. Coercions in hindley-milner systems. In *Proc. of Inter. Conf. on Proofs and Types*, volume 3085 of *LNCS*, pages 259–275. Springer-Velag, 2004.

[43] D. Leijen and E. Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.

[44] M. Y. Levin. Compiling regular patterns. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Uppsala, Sweden*, 2003.

[45] M. Y. Levin and B. C. Pierce. Typed-based optimization for regular patterns. In *First International Workshop on High Performance XML Processing*, 2004.

[46] M. Y. Levin and B. C. Pierce. Type-based optimization for regular patterns. In *Database Programming Languages (DBPL)*, August 2005.

[47] K. Z. M. Lu and M. Sulzmann. An implementation of subtyping among regular expression types. In *Proc. of APLAS'04*, volume 3302 of *LNCS*, pages 57–73. Springer-Verlag, 2004.

[48] E. Meijer, W. Schulte, and G. Bierman. Programming with circles, triangles and rectangles. In *Proc. of XML 2003*. deepX Ltd., 2003.

[49] E. Meijer and M. Shields. XMλ: A functional language for constructing and manipulating XML documents. (Draft), 1999.

[50] N. Mitchell and C. Runciman. Unfailing haskell: a static checker for pattern matching. In *Proc of 6th Symposium on Trends in Functional Programming (TFP)*, pages 313–328, 2005.

[51] J. Nordlander. Polymorphic subtyping in O'Haskell. *Science of Computer Programming*, 43(2):93–127, May 2002.

[52] S. Peyton-Jones. Call-pattern specialization for haskell programs. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Freiburg, Germany*, 2007.

[53] B. C. Pierce. *Types and programming languages.* MIT Press, Cambridge, MA, USA, 2002.

[54] B. C. Pierce and D. N. Turner. Local type inference. *ACM Transactions on Programming Languages and Systems*, 22(1):1–44, 2000.

[55] Polyparse : alternative parser combinator libraries. http://www.cs.york.ac.uk/fp/polyparse/.

[56] POSIX portable operating system interface. http://standards.ieee.org/regauth/posix/.

[57] Relax NG. http://relaxng.org/.

[58] J. C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 408–423. Springer-Verlag, 1974.

[59] J. J. M. M. Rutten. Automata and coinduction (an exercise in coalgebra). Technical report, CWI (Centre for Mathematics and Computer Science), 1998.

[60] Scala programming language. http://scala-lang.org/.

[61] P. J. Stuckey, M. Sulzmann, and J. Wazny. The Chameleon type debugger. In *Proc. of Fifth International Workshop on Automated Debugging (AADEBUG 2003)*, pages 247–258. Computer Research Repository (http://www.acm.org/corr/), 2003.

[62] P. J. Stuckey, M. Sulzmann, and J. Wazny. Type processing by constraint reasoning. In *Proc. of APLAS'06*, volume 4279 of *LNCS*, pages 1–25. Springer-Verlag, 2006.

[63] M. Sulzmann and J. Wazny. Chameleon. http://www.comp.nus.edu.sg/˜sulzmann/chameleon.

[64] Y. Sun, T. Li, Q. Zhang, J. Yang, and S. Liao. Parallel xml transformations on multi-core processors. In *Proc of IEEE International Conference on e-Business Engineering (ICEBE'07)*, pages 701–708, 2007.

[65] P. Thiemann. A typed representation for HTML and XML documents in Haskell. *Journal of Functional Programming*, 12(4 and 5):435–468, July 2002.

[66] S. Vansummeren. Type inference for unique pattern matching. *ACM TOPLAS*, 28(3):389–428, May 2006.

[67] J. Vouillon. Polymorphic regular tree types and patterns. In *Proc. of POPL'06*, pages 103–114. ACM Press, 2006.

[68] J. Vouillon. Polymorphism and XDuce style patterns. In *Informal Proc. of PLAN-X'06*, pages 49–60, 2006.

[69] P. Wadler and S. Blott. How to make *ad-hoc* polymorphism less *ad-hoc*. In *Proc. of POPL'89*, pages 60–76. ACM Press, 1989.

[70] M. Wallace and C. Runciman. Haskell and XML: Generic combinators or type-based translation? In *ICFP '99*, pages 148–159. ACM Press, 1999.

[71] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[72] Xact language. http://www.brics.dk/Xact/.

[73] The Xcerpt Language. http://www.xcerpt.org.

[74] Xhaskell. http://www.comp.nus.edu.sg/˜luzm/xhaskell.

[75] Extensible markup language (XML). http://www.w3.org/XML/.

[76] XML DOM. http://www.w3schools.com/dom/default.asp.

[77] Xml schema. http://www.w3.org/XML/Schema.

[78] The XPath Language. http://www.w3.org/TR/xpath/.

[79] The XQuery Language. http://www.w3.org/TR/xquery/.

[80] XSL transformations. http://www.w3.org/TR/xslt.

[81] D. Zhu and H. Xi. A Typeful and Tagless Representation for XML Documents. In *Proceedings of the First Asian Symposium on Programming Languages and Systems*, pages 89–104, Beijing, China, November 2003. Springer-Verlag LNCS 2895.

Appendix A

Implementation Details

A.1 Implementing the Word Problem in Haskell

In this section, we implement the word problem in Haskell. We use the `RE` Haskell datatype to build regular expression language.

```
data RE a where
  Phi :: RE a                        -- empty language, bottom
  Empty :: RE a                      -- empty word
  L :: a -> RE a                     -- single letter taken from alphabet a
  Choice :: RE a -> RE a -> RE a     -- r1 + r2
  Seq :: RE a -> RE a -> RE a        -- (r1,r2)
  Star :: RE a -> RE a               -- r*
 deriving Eq
```

To encode a word, we use a list of alphabet letters.

```
type Word a = [a]
```

The word problem can be implemented as the following `matchWord` function.

```
matchWord :: Eq a => RE a -> Word a -> Bool
matchWord r [] = isEmpty r
matchWord r (l:w) = matchWord (partDeriv r l) w

isEmpty :: RE a -> Bool
isEmpty Phi = False
isEmpty Empty = True
isEmpty (Choice r1 r2) = (isEmpty r1) || (isEmpty r2)
isEmpty (Seq r1 r2) = (isEmpty r1) && (isEmpty r2)
isEmpty (Star r) = True
isEmpty (L _) = False

partDeriv :: Eq a => RE a -> a -> RE a
partDeriv Phi _ = Phi
partDeriv Empty _ = Phi
partDeriv (L l1) l2 = if l1 == l2  then Empty else Phi
partDeriv (Choice r1 r2) l = Choice (partDeriv r1 l) (partDeriv r2 l)
partDeriv (Seq r1 r2) l =
    if isEmpty r1
    then Choice (Seq (partDeriv r1 l) r2) (partDeriv r2 l)
    else Seq (partDeriv r1 l) r2
partDeriv (this@(Star r)) l =
   Seq (partDeriv r l) this
```

We use the helper function `isEmpty` to check whether a regular expression is accepting the empty word and we use the helper function `partDerive` to compute the (partial) derivative of a regular expression with respect to a label, namely t/l.

A.2 Implementing Pattern Matching Algorithm in Haskell

We implement the pattern matching algorithm described in Chapter 6 in Haskell. We build the regular expression pattern language via datatypes `Pat a`.

```
data Pat a where
 PVar :: Int -> Word a -> RE a-> Pat a
 PPair :: Pat a -> Pat a -> Pat a
 PChoice :: Pat a -> Pat a -> Pat a

   --  PVar var w r
   -- variables var are represented by Ints
   -- w represents the part we have already seen
   -- r represents the remaining part we yet have to match
```

Next, we define a function that builds (partial) derivative of patterns, that is p/l.

```
pdPat :: Eq a => Pat a -> a -> Pat a
pdPat (PVar x w r) l = PVar x (w ++ [l]) (partDeriv r l)
pdPat (PPair p1 p2) l =
  if (isEmpty (strip p1))
  then PChoice (PPair (pdPat p1 l) p2) (PPair (mkEmpPat p1) (pdPat p2 l))
  else PPair (pdPat p1 l) p2
pdPat (PChoice p1 p2) l =
  PChoice (pdPat p1 l) (pdPat p2 l)
```

The following function implements stript p,

```
strip :: Pat a -> RE a
strip (PVar _ w r) = r
strip (PPair p1 p2) = Seq (strip p1) (strip p2)
strip (PChoice p1 p2) = Choice (strip p1) (strip p2)
```

Lastly, we need a function that makes a pattern empty, so that it stops matching with further input.

```
-- replace all (<w> x : r) by (<w> x: <>) if isEmpty r
-- otherwise yield (<w> x: Phi)
mkEmpPat :: Pat a -> Pat a
mkEmpPat (PVar x w r)
 | isEmpty r = PVar x w Empty
 | otherwise = PVar x w Phi
mkEmpPat (PPair p1 p2) = PPair (mkEmpPat p1) (mkEmpPat p2)
mkEmpPat (PChoice p1 p2) = PChoice (mkEmpPat p1) (mkEmpPat p2)
```

We use a list of int-word pairs to represent the subsitution.

```
type Env a = [(Int,Word a)]
```

With all these helper function, we can implement the allmatch   function as follows,

```
allMatch :: Eq a => Pat a -> Word a -> [Env a]
allMatch p (l:w) =
   allMatch (pdPat p l) w
allMatch (PVar x w r) [] =
   if isEmpty r then [[(x,w)]] else []
allMatch (PChoice p1 p2) [] =
  (allMatch p1 []) ++ (allMatch p2 [])
            -- indet choice
allMatch (PPair p1 p2) [] =
   (allMatch p1 []) `combine` (allMatch p2 [])
          -- build all possible combinations
 where
   combine xss yss = [ xs ++ ys |  xs <- xss, ys <- yss]
```

Thanks to Haskell laziness, we can implement the longest matching policy by taking the first element of `allMatch p w`, and implement the shortest matching policy by taking the last element of `allMatch p w`.

```
longPatMatch :: Eq a => Pat a -> Word a -> Maybe (Env a)
longPatMatch p w =
     first (allMatch p w)
  where
    first (env:_) = return env
    first _ = Nothing


shortPatMatch :: Eq a => Pat a -> Word a -> Maybe (Env a)
shortPatMatch p w =
     last (allMatch p w)
  where
```

```
last [env] = return env
last (_:xs) = last xs
last _  = Nothing
```

A.3 Deriving Upcast Coercion

We consider deriving the upcast coercion from the regular expression subtype algorithm $\vdash_{\text{sub}} t_1 \leq t_2$. Recall that the upcast function is used in the translation of semantic subtyping in System F* in Chapter 5.

In Section 6.3.2, we have shown how to derive the downcast coercion $d : [\![t_2]\!] \rightarrow Maybe\ [\![t_1]\!]$ from the proof of $\vdash_{\text{sub}} t_1 \leq_d t_2$. The main idea was to apply the "proofs-are-programs" principle to the regular expression containment algorithm. Every rewriting step in the proof gives us a pair of coercion functions. The downcast coercion d is defined in terms of these (small) coercion functions.

Given the regular expression subtyping proof $\vdash_{\text{sub}} t_1 \leq^u t_2$, the derived upcast coercion u coerces value from type $[\![t_1]\!]$ to $[\![t_2]\!]$. We can consider that the upcast coercion is the dual of the downcast coercion. Therefore, we can apply the idea of downcast derivation in the context of upcast derivation.

Similar to downcast coercion, the upcast coercion is driven by value rewriting. The upcast coercion u consists of three main steps,

- We rewrite the input value from type t into some intermediate value that is correspondent to the derivative t/l. Then we further rewrite it to the canonical form $|t/l|$;
- After rewriting the input into canonical form, we proceed by applying the sub coercion, another upcast coercion derived from the sub proof $\vdash_{\text{sub}} |t'/l| \leq |t/l|$, to the intermediate value. The result of this intermediate application must be of type $|t'/l|$.
- Finally we need to rewrite this result from the canonical form $|'t/l|$ back to derivative t'/l and to the type t'.

The above outlined the main procedures that are applied in the upcast coercion. If we compare it against the procedures of the downcast coercion which was oulined in Section 6.3.2, we find that the upcast function defines an inverse function of the downcast function.

Let us get into the details of the downcast coercion as stated in Figure A.1:

1. u has type $[\![t]\!] \rightarrow [\![t']\!]$, where $[\![t]\!]$ refers to the target representation of t. In contrast with the downcast coercion, we do *not* use *Maybe* type in the output. This is because the subtype proof $\vdash_{\text{sub}} t \leq t'$ guarantees that all the values of t can be fit into type t'. Thus the coercion is *safe*.
2. At the entry point of u, we verify whether the input value v denotes an empty sequence. This is accomplished by applying the helper function $isEmpty_t$ to v. If v is empty, we create an empty value of type $[\![t']\!]$, by using $mkEmpty_{t'}$.

$$\frac{\begin{array}{c} C \cup \{t \leq^{u} t'\} \vdash_{\text{sub}} |(t/l_1)| \leq^{u_1} |(t'/l_1)| \\ ... \\ C \cup \{t \leq^{u} t'\} \vdash_{\text{sub}} |(t/l_n)| \leq^{u_n} |(t'/l_n)| \\ \Sigma(t') = \{l_1, ..., l_n\} \\ \begin{array}{lcl} \mathit{uu_1} : \forall\llbracket \mathit{l_1}\rrbracket \rightarrow \llbracket t/\mathit{l_1}\rrbracket \rightarrow \llbracket t'\rrbracket & & \mathit{uu_n} : \forall\llbracket \mathit{l_n}\rrbracket \rightarrow \llbracket t/\mathit{l_n}\rrbracket \rightarrow \llbracket t'\rrbracket \\ \mathit{uu_1}\ \mathit{l_1}\ \mathit{v_1} = \mathit{let}\ x\ =\ \mathit{u_1}\ (\mathit{to_1}\ \mathit{v_1}) & ... & \mathit{uu_n}\ \mathit{l_n}\ \mathit{v_n} = \mathit{let}\ x\ =\ \mathit{u_n}\ (\mathit{to_n}\ \mathit{v'_n})\ \mathit{of} \\ \quad \mathit{in}\ \mathit{inj}_{(\mathit{l_1},t)}\ \mathit{l_1}\ (\mathit{from_1}\ x) & & \quad \mathit{in}\ \mathit{inj}_{(\mathit{l_n},t)}\ \mathit{l_n}\ (\mathit{from_n}\ x) \end{array} \\ \begin{array}{l} u : \forall\llbracket t\rrbracket \rightarrow \llbracket t'\rrbracket \\ u\ v = \mathit{if}\ \mathit{isEmpty}_t\ v\ \mathit{then}\ \mathit{Just}\ \mathit{mkEmpty}_{t'} \\ \qquad\qquad \mathit{else}\ \mathit{select}_{(\mathit{l_1},...,\mathit{l_n},t)}\ v\ \mathit{uu_1}...\mathit{uu_n} \end{array} \end{array}}{C \vdash_{\text{sub}} t \leq^{u} t'}$$

Helper functions

$$\begin{array}{l} \mathit{proj}_{(l,t'')} : \forall\llbracket t''\rrbracket \rightarrow \mathit{Maybe}\ (\llbracket l\rrbracket, \llbracket t''/l\rrbracket) \\ \mathit{inj}_{(l,t'')} : \forall\llbracket l\rrbracket \rightarrow \llbracket t''/l\rrbracket \rightarrow \llbracket t''\rrbracket \\ \mathit{isEmpty}_{t''} : \forall\llbracket t''\rrbracket \rightarrow \mathit{Bool} \\ \mathit{mkEmpty}_{t''} : \forall\llbracket t''\rrbracket \\ \mathit{to}_1 : \forall\llbracket (t/l_1)\rrbracket \rightarrow \llbracket |(t/l_1)|\rrbracket \\ ... \\ \mathit{to}_n : \forall\llbracket (t/l_n)\rrbracket \rightarrow \llbracket |(t/l_n)|\rrbracket \\ \\ \mathit{from}_1 : \forall\llbracket |(t'/l_1)|\rrbracket \rightarrow \llbracket (t'/l_1)\rrbracket \\ ... \\ \mathit{from}_n : \forall\llbracket |(t'/l_n)|\rrbracket \rightarrow \llbracket (t'/l_n)\rrbracket \end{array}$$

$$\begin{array}{l} \mathit{select}_{(\mathit{l_1},...,\mathit{l_n},t'')} : \forall\llbracket t''\rrbracket \rightarrow (\llbracket \mathit{l_1}\rrbracket \rightarrow \llbracket t''/\mathit{l_1}\rrbracket \rightarrow a) \\ \quad \rightarrow ... \rightarrow (\llbracket \mathit{l_n}\rrbracket \rightarrow \llbracket t''/\mathit{l_n}\rrbracket \rightarrow a) \rightarrow a \\ \mathit{select}_{(\mathit{l_1},...,\mathit{l_n},t'')}\ v\ \mathit{e_1}...\mathit{e_n} = \\ \qquad\qquad \mathit{let}\ \mathit{v_1} = \mathit{proj}_{(\mathit{l_1},t'')}\ v \\ \qquad\qquad\qquad ... \\ \qquad\qquad\quad \mathit{v_n} = \mathit{proj}_{(\mathit{l_n},t'')}\ v \\ \qquad\qquad \mathit{in}\ \mathit{case}\ (\mathit{v_1}, ..., \mathit{v_n})\ \mathit{of} \\ \qquad\qquad\quad (\mathit{Just}\ (\mathit{l_1}, \mathit{v'_1}), ...) \rightarrow \mathit{e_1}\ \mathit{l_1}\ \mathit{v'_1} \\ \qquad\qquad\quad ... \\ \qquad\qquad\quad (...., \mathit{Just}\ (\mathit{l_n}, \mathit{v'_n})) \rightarrow \mathit{e_n}\ \mathit{l_n}\ \mathit{v'_n} \end{array}$$

Figure A.1: Deriving upcast from subtype proofs

3. Otherwise, we need to rewrite v into its derivative type. To do that, we need to find out what is the leading label of v. Knowing that the set of possible labels appearing in v is $\{l_1, ..., l_n\}$, we iterate through the set of labels and look for the particular label l_i that we can extract from v. This task is carried out within the helper function $\mathit{select}_{(l_1,..,l_n,t)}$. In this function, we apply a set of projection functions $\mathit{proj}_{(l_1,t)}, ..., \mathit{proj}_{(l_n,t)}$ to v one by one. Recall that each projection function $\mathit{proj}_{(l_i,t)}$ tries to extract the label l_i from the input value which may fail. We will eventually find one successful projection.

4. Once we find a successful projection, let's say $\mathit{proj}_{(l_i,t)}$, we rewrite the input value into the derivative form (l_i, v_i). At this point, we want to apply the sub-upcast u_i to the remaining to v_i as planed. Note that v_i is of type $\llbracket t/l_i\rrbracket$. We have to further rewrite it to the canonical form $\llbracket|t/l_i|\rrbracket$ so that its structure as well as its type matches with the input type of u_i. This rewriting step is

carried out by another helper function to_i. When v_i is rewritten, we apply the sub-upcast u_i to v_i.

5. After we successfully upcast v_i to v'_i, we expect that v'_i is of the canonical form (type) $[\![|t'/l_i|]\!]$. We need to rewrite v'_i to type $[\![t'/l_i]\!]$, so that we can rewrite the result back to $[\![t']\!]$ by "inserting" l_i into the result. There are two more operations happening here. The $from_i$ function rewrites the value from its canonical form to the (unabridged) derivative form. The injection function $inj_{(l_i,t')}$ injects the label l_i back to the value, so that the result is of type $[\![t']\!]$. This completes the execution of an upcast application $(u\ v)$.

We omit the details of the helper functions $proj_{l,t}$, $inj_{l,t}$, to_i, $from_i$, $mkEmpty_t$ and $isEmpty_t$. We refer to readers to Section 6.3.2 for the details.

Let us consider an example,

Example 63 We consider the proof of $\vdash_{\text{sub}} A^* \leq A^*$ as follows,

$$\dfrac{\dfrac{A^* \leq^u A^* \in \{A^* \leq^{u'} A^*\}}{\{A^* \leq A^*\} \vdash_{\text{sub}} A^* \leq^{u'} A^*}(\text{Hyp}) \quad A^*/A = \langle\langle\rangle, A^*\rangle \quad |\langle\langle\rangle, A^*\rangle| = A^*}{\vdash_{\text{sub}} A^* \leq^u A^*}\ (\text{Norm})$$

We apply the algorithm described in Figure A.1 and define the upcast function as follows,

```
u ::  [A] -> [A]
u v = if $isEmpty_{A^*}$ v then $mkEmpty_{A^*}$
      else case $proj_{(A,A^*)}$ v of
           Just (l,rest) -> let rest' = $u$ (to rest)
                            in $inj_{(A,A^*)}$ l (from rest)
           Nothing -> error ``this will not happen.''
```

In the above derivation, we inline the *select* function in the body of u. Note we apply of co-induction hypothesis $A^* \leq^u A^* \in \{A^* \leq^{u'} A^*\}$ in the proof derivation. Therefore, we define the upcast function u by building a recursive function, i.e., by letting $u' = u$. The helper functions are defined as follows,

```
$isEmpty_{A^*}$ [] = True
$isEmpty_{A^*}$ _ = False

$mkEmpty_{A^*}$ = []

$proj_{(A,A^*)}$ [] = Nothing
$proj_{(A,A^*)}$ (x:xs) = Just (x, ((),xs))

$inj_{(A,A^*)}$ x ((),xs) = (x:xs)
```

Recall that the *to* and $form$ functions are defined in Figure 6.14 in Section 6.3.2. By resolving the type class constraint `Canonical ((),[A]) [A]`, we derive the `to` and `from` functions as follows,

```
to ((),v) = v
from v = ((),v)
```

□

Like the downcast coercion, there are multiple ways in defining an upcast coercion, to which we refer as the ambiguity problem. For instance, we can define an upcast function that coerce value from A to $(A|A)$ in multiple ways,

```
u :: A -> (Or A A)
u v = L v            -- (1)
```

or

```
u v = R v            -- (2)
```

To resolve the ambiguity we apply the strategy which we mentioned in Section 6.3.2. That is we always favor the “left injection” (i.e. (1)).

Appendix B

Proof Details

B.1 Technical Proofs for Chapter 4

Lemma 1 (Decidability of Subtyping I) *If we omit subtyping among labels, then for any two types t and t' we can decide whether $\vdash_{sub}\ t \leq t'$ is valid or not.* To prove this lemma, we first consider a technical definition which helps us to enumerate all (sub) goals derived from $\vdash_{\text{sub}}\ t \leq t'$. Basically they are subtype relations among the derivatives of t and t'.

Definition 7 (Word Derivative and Type Derivative) *Let w be a source value and t be a type, we define word derivative as follows,*

$$wd(\langle\rangle \shortmid t) = t \qquad wd(\langle l, w\rangle \shortmid t) = wd(w \shortmid d(l \shortmid t))$$

Let t_1 and t_2 be two types, we define the type derivative as follows,

$$td(t_1 \shortmid t_2) = \{wd(w \shortmid t_2) | w : t_1\}$$

Example 64 For instance, $wd(\langle A, B\rangle \shortmid (A|B)^*) = (A|B)^*$. □

We now consider proving Lemma 1.

> **Proof:** It is clear that proof is decidable if the size of the proof derivation tree is finite. Recall that all subtype proof derivations are of the following shape,
>
> $$\frac{\begin{array}{c} C \cup \{t \leq t'\} \vdash_{\text{sub}}\ d(l_1 \shortmid t) \leq d(l_1 \shortmid t') \\ \ldots \\ C \cup \{t \leq t'\} \vdash_{\text{sub}}\ d(l_n \shortmid t) \leq d(l_n \shortmid t') \\ \Sigma(t') = \{l_1, ..., l_n\} \end{array}}{C \vdash_{\text{sub}}\ t \leq t'}\text{(Norm)}$$
>
> As we can observe, every time we apply the (Norm) rule in the derivation, we add the current goal into the constraint context C. The (Hyp) rule is applied at every leaf node in the proof derivation tree by looking up the

current goal in the constraint context. Note that the height of the proof tree is proportional to the size of the constraint context C. Therefore if we can show that for any subtype proof derivation, the sizes of the constraint contexts are bounded, then we can conclude that all subtype proof $\vdash_{\text{sub}} t \leq t'$ are decidable.

Since label subtyping is disallowed, the constraint contexts are sets of the subtype relations among t and t' and word derivatives of t and t'. By Definition 7, $td(\Sigma(t)^* \shortmid t)$ denotes the set of all word derivative of t that we can ever build. Similarly $td(\Sigma(t')^* \shortmid t')$ denotes the set for t'. Therefore, we can conclude that any constraint context C must be a subset of the Cartesian product of $d(\Sigma(t)^* \shortmid t)$ and $d(\Sigma(t')^* \shortmid t')$.

A result from [4] states that $pd(\Sigma(t)^* \shortmid t)$ and $pd(\Sigma(t')^* \shortmid t')$ are bounded. Therefore the set of all possible $d(\Sigma(t)^* \shortmid t)$ and $d(\Sigma(t')^* \shortmid t')$ is also finite. As a result, we can conclude that the set $d(\Sigma(t)^* \shortmid t) \times d(\Sigma(t')^* \shortmid t')$ is finite. It follows that the height of the proof derivation tree must be finite. Therefore the subtype proof of $\vdash_{\text{sub}} t \leq t'$ is decidable for any type t and t'. □

Theorem 1 (Decidability of Type Checking I) *If we omit subtyping among labels, then we can decide whether* $\Gamma \vdash e : t$ *holds or not.*

Proof: It follows straight-forwardly from Lemma 1 □

Lemma 2 (Decidability of Subtyping II) *Let* Γ_{init} *be an initial type environment which only contains non-nested data types. Then, for any two types* t *and* t' *we can decide whether* $\vdash_{sub} t \leq t'$ *is valid or not.*

Proof: We prove this lemma using a similar idea that is used in proving Lemma 1. Except that we need to take into account label subtypings. In the presence of label subtyping, the constraint context C not only contains subtype relations among word derivatives of t and t', but also those (subsequent) subtyping relation of arising from $\vdash_{\text{lab}} l \leq l'$, where $\vdash_{\text{lab}} l \leq l'$ is a label subtyping arising from $\vdash_{\text{lab}} t \leq t'$.

Since t and t' are non-nested, by definition, we know that we can build a *finite* closure of all labels appearing in t, t' and their strongly connected components via the following operation

$$\begin{array}{c}
lclosure(()) = \{\} \quad lclosure(t_1, t_2) = lclosure(t_1) \cup lclosure(t_2) \\
lclosure(t_1|t_2) = lclosure(t_1) \cup lclosure(t_2) \quad lclosure(t^*) = lclosure(t) \\
lclosure(a) = \{a\} \quad lclosure(\forall a.t) = \{\forall a.t\} \cup lclosure(t) \\
lclosure(t_1 \to t_2) = \{t_1 \to t_2\} \cup lclosure(t_1) \cup lclosure(t_2) \\
lclosure(T\ \bar{t}) = \{T\ \bar{t}\} \cup \bigcup_{i=1,\ldots,n} lclosure(\{\overline{t/a}\}t'_i) \\
\text{where } K : \forall \bar{a}.t'_1 \to \ldots \to t'_n \to T\ \bar{a} \in \Gamma_{init}
\end{array}$$

Since $lclosure(t)$ and $lclosure(t')$ are finite, therefore, the number of label subtyping relations $\vdash_{\text{lab}} \ l \leq l'$ arising from $\vdash_{\text{sub}} \ t \leq t'$ is finite, too.

In Figure 4.4a label subtype rules (Arrow), (T) and (Forall) introduce new proof goals, which are eventually recorded by the constraint context C. (We omit (Var), because it does not introduce any new subgoal.) Thus, we find that any subtype constraint $\vdash_{\text{sub}} \ t_1 \leq t_2$ in C, we have either

1. $t_1 \in td(\Sigma(t)^* \shortmid t)$ and $t_2 \in td(\Sigma(t')^* \shortmid t')$; or
2. $t_1 \in td(\Sigma(t_1')^* \shortmid t_1')$ and $t_2 \in td(\Sigma(t_2')^* \shortmid t_2')$ where $\vdash_{\text{sub}} \ t_1' \leq t_2'$ is an immediate subgoal of $\vdash_{\text{lab}} \ l \leq l'$ and $\vdash_{\text{lab}} \ l \leq l'$ is a literal arising from $\vdash_{\text{sub}} \ t \leq t'$.

Therefore, we can conclude that the constraint context C is still bounded. Thus, $\vdash_{\text{sub}} \ t \leq t'$ is decidable. □

Theorem 2 (Decidability of Type Checking II) *Let Γ be a type environment which only contains non-nested data types, e an expression and t a type. Then, we can decide whether $\Gamma \vdash e : t$ holds or not.*

Proof: It follows from Lemma 2. □

Lemma 3 (Semantic Subtyping) *If we restrict label types l to be data types of form data $T = T$, then for any two types t and t', we have $\vdash_{sub} \ t \leq t'$ iff $L(t) \subseteq L(t')$.*

Proof: To prove this lemma, we make use of the correctness results of the $\vdash_{\text{empty}} \ \langle\rangle \in t$ system and the $d(l \shortmid t)$ operations. These results are stated in Lemma 24 and 25.

Let us first consider the $\Rightarrow$ direction (i.e. the soundness) . We want to show that

$$\textit{Let } \vdash_{\text{sub}} \ t_1 \leq t_2 \textit{ and } w \in L(t_1) \textit{ ,then } w_2 \in L(t_2).$$

We verify the above by induction over w.

Case $w = \langle\rangle$. Since $\langle\rangle \in L(t_1)$, by Lemma 24 we conclude that

$$\vdash_{\text{empty}} \ \langle\rangle \in t_1 \tag{B.1}$$

From the premise $\vdash_{\text{sub}} \ t_1 \leq t_2$ and B.1 we can deduce that

$$\vdash_{\text{empty}} \ \langle\rangle \in t_2 \tag{B.2}$$

By Lemma 24 and B.2 we can conclude that $\langle\rangle \in L(t_2)$. That verifies the case where $w = \langle\rangle$.

Case $w = \langle l, w' \rangle$. Since l is restricted, we find that the proof of $\vdash_{\text{sub}} t_1 \leq t_2$ will be as follows,

$$\frac{\vdash_{\text{sub}} d(l \shortmid t_1) \leq d(l \shortmid t_2) \\ \ldots}{\vdash_{\text{sub}} t_1 \leq t_2}$$

Since the conclusion in the above derivation is our assumption, thus the premise

$$\vdash_{\text{sub}} d(l \shortmid t_1) \leq d(l \shortmid t_2) \tag{B.3}$$

must hold. By Lemma 25 and the definition of derivative, we have

$$L(d(l \shortmid t_1)) = \{w'' | \langle l, w' \rangle \in L(t_1)\} \tag{B.4}$$

It follows that

$$w' \in L(d(l \shortmid t_1)) \tag{B.5}$$

Since w' is the suffix of w by taking away l, we can apply the induction hypothesis to B.4 and B.3 to conclude that

$$w' \in L(d(l \shortmid t_2)) \tag{B.6}$$

From B.6 we conclude that $\langle l, w' \rangle \in L(t_2)$, which is what we want to prove in this case.

We conclude that the $\Rightarrow$ direction of this lemma holds.

Now let us consider the $\Leftarrow$ direction (i.e. the completeness).

We want to show

$$\textit{Let } L(t_1) \subseteq L(t_2), \textit{then } \vdash_{\text{sub}} t_1 \leq t_2$$

We prove by induction over the derivation tree of $\vdash_{\text{sub}} t_1 \leq t_2$. Since l is restricted, to verify that $\vdash_{\text{sub}} t_1 \leq t_2$, we only need to verify

$$\vdash_{\text{empty}} \langle\rangle \in t_1 \text{ implies } \vdash_{\text{empty}} \langle\rangle \in t_2 \tag{B.7}$$

and

$$\{t_1 \leq t_2\} \vdash_{\text{sub}} d(l \shortmid t_1) \leq d(l \shortmid t_2) \text{ for all } l \in \Sigma(t_1) \cup \Sigma(t_2), \tag{B.8}$$

Let us consider B.7. Suppose $\vdash_{\text{empty}} \langle\rangle \in t_1$, based on Lemma 24, we can immediately conclude that

$$\langle\rangle \in L(t_1) \tag{B.9}$$

Since $L(t_1) \subseteq L(t_2)$, it follows that from B.9 that

$$\langle\rangle \in L(t_2) \tag{B.10}$$

We apply Lemma 24 to B.10 to conclude that $\vdash_{\text{empty}} \langle\rangle \in t_2$, which is what we want to show in B.7.

We now consider B.8. Let $\langle l, w\rangle$ be an arbitary word such that $\langle l, w\rangle \in L(t_1)$. By applying Lemma 25, we deduce that,

$$w \in L(d(l \mid t_1)) \tag{B.11}$$

From the result of Lemma 25, we have

$$L(d(l \mid t_1)) = \{w' | \langle l, w'\rangle \in L(t_1)\} \tag{B.12}$$

and

$$L(d(l \mid t_2)) = \{w' | \langle l, w'\rangle \in L(t_2)\} \tag{B.13}$$

From the assumption $L(t_1) \subseteq L(t_2)$, B.11, B.12 and B.13, we can conclude that

$$w \in L(d(l \mid t_2)) \tag{B.14}$$

Since $\langle l, w'\rangle$ is randomly choosen, w.l.o.g., we conclude that

$$L(d(l \mid t_1) \leq L(d(l \mid t_2)) \tag{B.15}$$

Now we can appy induction hypothesis to B.15 to conclude that $\vdash_{\text{sub}} d(l \mid t_1) \leq d(l \mid t_2)$, which is what we want to prove in B.8.

We conclude that the $\Leftarrow$ direction of this lemma holds. □

Lemma 24 $\vdash_{empty} \langle\rangle \in t$ *iff* $\langle\rangle \in L(t)$.

Proof: The proof is straight-forward. The $\Rightarrow$ direction can be proven by induction over the derivation of $\vdash_{\text{empty}} \langle\rangle \in t.$. The $\Leftarrow$ direction can be verifiied by structural induction of t. □

Lemma 25 *Let $d(l \mid t) = t'$, then t' is a derivative of t with respect to l.*

Proof: Note that by definition t' is a derivative of t with respect to l iff $L(t') = \{w' | \langle l, w'\rangle \in L(t)\}$. Thus the proof of this lemma is straight-forward by structural induction over t. We omit the details. □

B.2 Technical Proofs for Chapter 5

Lemma 5 (Semantic Preservation (Upcast)) *Let $\vdash_{sub} t_1 \leq^u t_2$, v_1 be a target value of type $[\![t_1]\!]$, w be a source value such that $w \overset{t_1}{\approx} v_1$. Then $u\ v_1 \longrightarrow^* v_2$ implies that $w \overset{t_2}{\approx} v_2$.*

Proof: We prove by induction over the size of w and we make use of the properties of the helper functions defined in Definition 3.

Case: $w \sim \langle\rangle$*:* From the assumption, we note that $\langle\rangle \overset{t_1}{\approx} v_1$. Based on the property of $isEmpty_t$, we have that $isEmpty_{t_1}\ v_1 \longrightarrow^* True$ (1). Thus, by making use of (1), we have that $u\ v_1 \longrightarrow mkEmpty_{t_2}$ (1). Based on the property of $mkEmpty_t$, we can conclude that $\langle\rangle \overset{t_2}{\approx} mkEmpty_{t_2}$ (3). By definition, it immediately follows from (3) that $\langle\rangle \overset{t_2}{\approx} v_2$ where $mkEmpt_{t_2} \longrightarrow^* v_2$, which means that $w \overset{t_2}{\approx} v_2$.

Case: $w \sim \langle l, w'\rangle$*:* From the assumption, we note that $\langle l, w\rangle \overset{t_1}{\approx} v_1$ (4). Based on the property of $pdtProj_{(l,t)}$, from (4) we can conclude that $pdtProj_{(l,t_1)}\ v_1 \longrightarrow^* Just\ (v_l, v_1')$ where $w' \overset{pdt(l \,\shortmid\, t_1)}{\approx} v_1'$ (5) and $l \overset{l}{\approx} v_l$ (6). Since $\vdash_{\mathrm{sub}} t_1 \leq t_2$, it is clear that $\vdash_{\mathrm{sub}} pdt(l \shortmid t_1) \leq^{u'} pdt(l \shortmid t_2)$ must hold for some u'. By making use of (5) and (6), thus we can conclude that

$$u\ v_1 \longrightarrow \ (pdtInj_{(l,t_2)}\ v_l\ (u'\ v_1'))$$

For simplicity we inline the *select* and *uu* functions in the above. Let $u'\ v_1' \longrightarrow^* v_2'$ for some v_2'. We apply induction hypothesis to the result of (5), we can conclude that $w' \overset{pdt(l \,\shortmid\, t_2)}{\approx} v_2'$ (7). Based on the property of $pdtInj_{(l,t)}$, from (6) and (7) we can conclude that

$$pdtInj_{(l,t_2)}\ v_l\ v_2' \longrightarrow^* v_2$$

where $\langle l, w'\rangle \overset{t_2}{\approx} v_2$. Therefore, we conclude that $w \overset{t_2}{\approx} v_2$. □

Lemma 6 (Semantic Preservation (Downcast)) *Let* $\vdash_{sub} t_1 \leq_d t_2$*,* v_2 *be a target value of type* $[\![t_2]\!]$*,* w *be a source value such that* $w \overset{t_2}{\approx} v_2$*. Then* $d\ v_2 \longrightarrow^* Just\ v_1$ *implies that* $w \overset{t_1}{\approx} v_1$*.*

The proof of this lemma is almost a repetition of the proof for Lemma 5. Hence, we omit the details.

Lemma 7 (Semantic Preservation) *Let* $\vdash_{sub} t_1 \leq_d^u t_2$*,* v_1 *be a value of type* $[\![t_1]\!]$ *and* v_2 *be a value of type* $[\![t_2]\!]$*. Then,*
(1) $d\ (u\ v_1) \longrightarrow^* Just\ v_3$ *such that* $v_1 \overset{t_1}{\leftrightarrow} v_3$*, and*
(2) if $d\ v_2 \longrightarrow^* Just\ v_4$ *then* $v_2 \overset{t_2}{\leftrightarrow} u\ v_4$*.*

Proof: We apply the results of Lemma 5 and 6 to verify this lemma.

We first consider (1). Let w be a source value such that $w \overset{t_1}{\approx} v_1$. By Lemma 5, we can conclude that $w \overset{t_2}{\approx} u\ v_1$. Thus, we can apply Lemma

6 to conclude that $w \overset{t_1}{\approx} v_3$. Finally, by definition, we can conclude that $v_1 \overset{t_1}{\leftrightarrow} v_3$.

We can verify (2) in a similar way. □

Theorem 5 (Type Preservation) *Let* $\Gamma_{init} \vdash e : t \rightsquigarrow E$. *Then* $\Gamma_{init}^{target} \vdash_F E : [\![t]\!]$.

Proof: We prove the theorem by verifying a stronger result.

Let Γ be a source type environment and $\Gamma_{init} \cup \Gamma \vdash e : t \rightsquigarrow E$. Then $\Gamma_{init}^{target} \cup [\![\Gamma]\!] \vdash_{\mathrm{F}} E : [\![t]\!]$.

Prove by induction over $\Gamma \vdash e : t \rightsquigarrow E$, for simplicity, we omit the initial type environment Γ_{init} and Γ_{init}^{target}.

Case (Sub):

$$\frac{\Gamma \vdash e : t_1 \rightsquigarrow E \qquad \vdash_{\mathrm{sub}} t_1 \leq^u t_2}{\Gamma \vdash e : t_2 \rightsquigarrow u\ E}$$

By Lemma 4, from the second premise $\vdash_{\mathrm{sub}} t_1 \leq^u t_2$ we have that $u : [\![t_1]\!] \rightarrow [\![t_2]\!]$ (1). We apply induction hypothesis to the first premise $\Gamma \vdash e : t_1$ we have hat $[\![\Gamma]\!] \vdash_{\mathrm{F}} E : [\![t_1]\!]$ (2). It follows from (1) and (3) that $[\![\Gamma]\!] \vdash_{\mathrm{F}} u\ E : [\![t_2]\!]$.

Case (Case):

$$\frac{\begin{array}{c}\Gamma \vdash e : t \rightsquigarrow E \quad \Gamma_i \vdash_{\mathrm{pat}} p_i : t_i \rightsquigarrow P_i \quad \vdash_{\mathrm{sub}} t_i \leq_{d_i} t \quad \Gamma \cup \Gamma_i \vdash e_i : t' \rightsquigarrow E_i \\ g_i = \lambda c.\mathsf{case}\ d_i\ E\ \mathsf{of}\ \{Just\ P_i \rightarrow E_i; Nothing \rightarrow c\} \quad \text{for } i \in I\end{array}}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ [p_i \rightarrow e_i]_{i \in I} : t' \rightsquigarrow g_1\ (...\ (g_n\ (error\ \text{"}pattern\ is\ not\ exhaustive\text{"})))}$$

Applying induction hypothesis to the first premise $\Gamma \vdash e : t \rightsquigarrow E$, we find that $[\![\Gamma]\!] \vdash_{\mathrm{F}} E : [\![t]\!]$ (4). By Lemma 4, from $\vdash_{\mathrm{sub}} t_i \leq_{d_i} t$ we can conclude that $d : [\![t]\!] \rightarrow Maybe\ [\![t_i]\!]$ (5). From (4) and (5) we can deduce that $[\![\Gamma]\!] \vdash_{\mathrm{F}} d_i\ E : Maybe\ [\![t_i]\!]$. Since the premise $\Gamma_i \vdash_{\mathrm{pat}} p_i : t_i \rightsquigarrow P_i$ maintains the invariance of $[\![\Gamma_i]\!] \vdash_{\mathrm{F}} P_i : [\![t_i]\!]$, we can conclude that for every pattern variable x appearing P_i the relation $[\![\Gamma_i]\!] \vdash_{\mathrm{F}} x : [\![\Gamma_i(x)]\!]$ must hold (6). It means that the translated System F pattern is well-typed. What we yet need to show is that the body of the pattern clause must be well-typed, too. We apply induction hypothesis to the premise $\Gamma \cup \Gamma_i \vdash e_i : t' \rightsquigarrow E_i$ to conclude that $[\![\Gamma \cup \Gamma_i]\!] \vdash_{\mathrm{F}} E_i : [\![t']\!]$. Thus, the whole expression $g_1\ (...\ (g_n\ (error\ \text{"}pattern\ is\ not\ exhsaustive\text{"})))$ has type $[\![t']\!]$ under the type environment $[\![\Gamma]\!]$.

The other cases are similar. □

Lemma 8 (Coherence of Coercive Subtyping) *If we replace (LN) in Figure 4.4a by rule (LN') (see Section 4.5), then for each subtype statement there exists at most one subtype proof.*

Proof: Without loss of generality, consider a proof derivation of $\vdash_{\text{sub}} \langle l_1, t_1\rangle \leq (\langle l_2, t_2\rangle | \langle l_3, t_3\rangle)$ as follows,

$$\frac{\vdash_{\text{lnf}} \langle l_1, t_1\rangle \leq (\langle l_2, t_2\rangle | \langle l_3, t_3\rangle)}{\vdash_{\text{sub}} \langle l_1, t_1\rangle \leq (\langle l_2, t_2\rangle | \langle l_3, t_3\rangle)} \tag{B.16}$$

From the assumption, we replace (LN) with (LN'). To reduce B.16, we can apply (LN')

1. to $\langle l_1, t_1\rangle$ and $\langle l_2, t_2\rangle$ if $l_1 = l_2$; or
2. to $\langle l_1, t_1\rangle$ and $\langle l_3, t_3\rangle$ if $l_1 = l_3$.

According to the definition of type normalization (see Figure 4.4b), $l_2 \neq l_3$. Therefore, only one derivation out of the two possibilities will be valid. Thus we can conclude that there exists at most one subtype proof. □

Lemma 9 (Transitivity of Coercive Subtyping) *Let $\vdash_{sub} t_1 \leq^{u_1} t_2$, $\vdash_{sub} t_2 \leq^{u_2} t_3$, $\vdash_{sub} t_1 \leq^{u_3} t_3$ and v be a value of type $[\![t_1]\!]$. Then, $u_2\ (u_1\ v) \overset{t_3}{\leftrightarrow} u_3\ v$.*

Proof: It follows straight-forwardly from Lemma 5 that u_1 and u_2 and u_3 are semantic preserving. Thus, there exists a source expression w such that

$$w \overset{t_1}{\approx} v \tag{B.17}$$

$$w \overset{t_3}{\approx} u_2\ (u_1\ v) \tag{B.18}$$

$$w \overset{t_3}{\approx} u_3\ v \tag{B.19}$$

By definition of $\cdot \leftrightarrow \cdot$, from B.18 and B.19 we can conclude that $u_2\ (u_1\ v) \overset{t_3}{\leftrightarrow} u_3\ v$. □

Lemma 10 (Preservation of Coercive Pattern Matching Equivalence) *Let t_1 and t_2 be two source types and E_1 and E_2 be two target expressions such that $\vdash_F E_1 : [\![t_1]\!]$ and $\vdash_F E_2 : [\![t_2]\!]$ and $E_1 \overset{t_1}{\rightarrow}\overset{t_2}{\leftarrow} E_2$. Let t_3 be a source type such that $\vdash_{sub} t_2 \leq^u t_3$. Then, we have that $E_1 \overset{t_1}{\rightarrow}\overset{t_3}{\leftarrow} u\ E_2$.*

Proof:(Sketch) To motivate the intuition of the proof, let us consider an example. Let $t_1 = (A^+|B)$, $t_2 = A^*$ and $t_3 = (A|B)^*$. Let E_1 and

E_2 be target expressions such that $\vdash_F E_1 : [\![t_1]\!]$ and $\vdash_F E_2 : [\![t_2]\!]$. The assumption $E_1 \overset{t_1}{\rightarrow}\overset{t_2}{\leftarrow} E_2$ implies the results of downcasting E_1 and E_2 are identical. Which means that E_1 and E_2 must share the same semantic meaning. Since t_2 is A^*, we can conclude that neither E_1 nor E_2 contains any B label. Note that for an arbitrary expression e_3 of type t_3, $E_1 \overset{t_1}{\rightarrow}\overset{t_3}{\leftarrow} E_3$ might not hold, because E_3 might potentially contain some B label, which is not in E_1. On the other hand, by Lemma 5 we know that the upcast coercion u from type t_2 to type t_3 does not change the labels in its argument. As a result, the semantic meaning of expression $(u\ E_2)$ should be the same as E_2. In other words, $(u\ E_2)$ does not contain label B. Thus, $E_1 \overset{t_2}{\rightarrow}\overset{t_3}{\leftarrow} u\ E_2$ must hold.

As motivated by the example, we can prove this lemma by (recursively) comparing all the (partial) derivatives of t_1, t_2 and t_3, and examining the construction of the upcast and downcast coercions generated from the subtype proofs.

□

Lemma 11 (Semantic Equiv. Implies Coercive Pattern Matching Equiv.) *Let t be a source type and E_1 and E_2 be two target expressions such that $\vdash_F E_1 : [\![t]\!]$ and $\vdash_F E_2 : [\![t]\!]$ and $E_1 \overset{t}{\leftrightarrow} E_2$. Then, we have that $E_1 \overset{t}{\rightarrow}\overset{t}{\leftarrow} E_2$.*

Proof: We start by making use of the definition of $E_1 \overset{t}{\leftrightarrow} E_2$, from which we note that if E_1 evaluates to v_1 then E_2 must evaluate to v_2 such that $v_1 \overset{t}{\leftrightarrow} t_2$. That is when we "flatten" v_1 and v_2, they are equivalent. It is also not hard to verify that v_1 and v_2 share the same time $[\![t]\!]$. By the preservation property of System F, E_1 and E_2 are of type $[\![t]\!]$ too.

To show that $E_1 \overset{t}{\rightarrow}\overset{t}{\leftarrow} E_2$, we need to show that for any t' which is a subtype of t. the result of downcasting E_1 from t to t' will be exactly the same as the one obtained by downcasting E_2 to t'. Note that E_1 and E_2 share the same "flatten" value, say $\langle l_1, ...l_n\rangle$. We guarantee that the downcast function $d :: [\![t]\!] \rightarrow Maybe\ [\![t']\!]$ must be determinstic. In the applications of of $(d\ E_1)$ and $(d\ E_2)$, we destruct the input values by pulling l_1 to l_n out from them, then we build the results by applying the $from$ operations to l_1 up to l_n. Since we are passing through the same set of labels to the $from$ operations in the same order, the results of the downcast operation must be the same. The same observation applies when E_1 and E_2 are empty. □

Theorem 6 (Coherence) *Let $\vdash e : t_1 \rightsquigarrow E_1$ and $\vdash e : t_2 \rightsquigarrow E_2$. Then, $E_1 \overset{t_1}{\rightarrow}\overset{t_2}{\leftarrow} E_2$.*

Proof: (Sketch) In the presence of pattern matching, we prove the coherence theorem by proving a stronger result which takes into account the value bindings.

We first extend the definition of equivalence relation $\cdot \overset{\cdot}{\leftrightarrow} \cdot$ to express semantic equivalence among target value bindings.

Definition 8 *Let θ_1 and θ_2 be two target value substitutions and Γ be source type environment such that $\theta_1 \vdash [\![\Gamma]\!]$ and $\theta_2 \vdash [\![\Gamma]\!]$. We say $\theta_1 \overset{\Gamma}{\leftrightarrow} \theta_2$ iff $\forall x$ we have $\theta_1(x) \overset{\Gamma(x)}{\leftrightarrow} \theta_2(x)$.*

Then we would like to show a stronger result as follows.

Let $\Gamma \vdash e : t_1 \leadsto E_1$ and $\Gamma \vdash e : t_2 \leadsto E_2$. Let θ_1 and θ_2 be two target value substitutions such that $\theta_1 \overset{\Gamma}{\leftrightarrow} \theta_2$. Then $\theta_1(E_1) \overset{t_1\ t_2}{\rightarrow\leftarrow} \theta_2(E_2)$.

Note that it is safe to assume that the two derivations share the same type environment Γ, because the program is fully type-annotated, Γ is always built deterministically.

Ideally, the proof proceeds by induction over the two derivation $\Gamma \vdash e : t_1 \leadsto E_1$ and $\Gamma \vdash e : t_2 \leadsto E_2$. Note that the derivations are non syntax-directed thanks to the subsumption rule. That is we cannot guarantee that the two derivations are reduced by applying the same rules.

Without loss of generality, we assume that we can apply the subsumption rule to both derivations *exhaustively*, until they both reach the same typing judgement $\Gamma \vdash e : t$, where $\Gamma \vdash e : t$ is not reduced by the subsumption rule, that is, t is the type of e by looking up the type environment Γ.

Case: (Sub)

Assume we apply the (Sub) rule to the first derivation for n times,

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash e : t_n \leadsto E \quad \vdash_{\text{sub}} t_n \leq^{u_n} t_{n-1}}{...}}{\Gamma \vdash e : t_2 \leadsto (u_2\ ...\ (u_n\ E)) \quad \vdash_{\text{sub}} t_2 \leq^{u_1} t_1}}{\Gamma \vdash e : t_1 \leadsto (u_1\ (u_2\ ...\ (u_n\ E)))}$$

and we apply the (Sub) rule to the second derivation for m times,

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash e : t'_m \leadsto E' \quad \vdash_{\text{sub}} t'_m \leq^{u'_m} t'_{m-1}}{...}}{\Gamma \vdash e : t'_2 \leadsto (u'_2\ ...\ (u'_n\ E')) \quad \vdash_{\text{sub}} t'_2 \leq^{u'_1} t'_1}}{\Gamma \vdash e : t'_1 \leadsto (u'_1\ (u'_2\ ...\ (u'_n\ E')))}$$

where $t_n = t'_m$ and t_n is the type of e by looking up the type environment Γ.

We wish to apply induction hypothesis to show that $(u_1 \; ... \; (u_n \; E))$ behaves the same as $(u'_1 \; ... \; (u'_m \; E'))$ (1). Note that there exist infinitely many ways of coercing E from t_n to t_1 and similarly for coercingE' from t'_m to t'_1. Fortunately, Lemma 8 guarantees that the subtype proof is coherence and Lemma 9 and Lemma 11 guarantee that the result of subtype coercion preserved under transitivity. As a result, we simplify the above derivations as follows,

$$\frac{\Gamma \vdash e : t_n \rightsquigarrow E \qquad \vdash_{\text{sub}} t_n \leq^u t_1}{\Gamma \vdash e : t_1 \rightsquigarrow (u \; E)}$$

and

$$\frac{\Gamma \vdash e : t'_m \rightsquigarrow E' \qquad \vdash_{\text{sub}} t'_m \leq^{u'} t'_1}{\Gamma \vdash e : t'_1 \rightsquigarrow (u' \; E')}$$

And we note that $(u \; E)$ (resp. $(u' \; E')$) behaves the same as $(u_1 \; ... \; (u_n \; E))$ (resp. $(u'_1 \; ... \; (u'_m \; E'))$) Hence, (1) is proven if we can verify $\theta_1(u \; E) \overset{t_1}{\rightarrow}\overset{t'_1}{\leftarrow} \theta_2(u' \; E')$ (2). By induction hypothesis, we conclude that $\theta_1(E) \overset{t_n}{\rightarrow}\overset{t'_m}{\leftarrow} \theta_2(E')$ (3). Applying the result of Lemma 10, to the both sides of (3), we can further conclude that (2) is valid.

Therefore, we conclude that we have proven the case for subsumption rule.

From this point onwards, we assume that the subsumption rule is exhaustively applied. As a consequence, for the remaining cases, we can assume that the derivation $\Gamma \vdash e : t \rightsquigarrow E_1$ and $\Gamma \vdash e : t \rightsquigarrow E_2$ are reduced by applied the same rule.

Case: (Case)

Applying (Case) rule to the first derivation, we have

$$\frac{\begin{array}{c}\Gamma \vdash e : t_0 \rightsquigarrow E \quad \Gamma_i \vdash_{\text{pat}} p_i : t_i \rightsquigarrow P_i \\ \vdash_{\text{sub}} t_i \leq_{d_i} t_0 \quad \Gamma \cup \Gamma_i \vdash e_i : t \rightsquigarrow E_i \\ g_i = \lambda c.\mathsf{case} \; d_i \; E \; \mathsf{of} \; [\mathit{Just} \; P_i \rightarrow E_i, \mathit{Nothing} \rightarrow c] \quad \text{for } i \in I\end{array}}{\begin{array}{c}\Gamma \vdash \mathsf{case} \; e \; \mathsf{of} \; [p_i \rightarrow e_i]_{i \in I} : t \rightsquigarrow \\ g_1 \; (... \; (g_n \; (\mathit{error} \; \text{"}\mathit{pattern \; is \; not \; exhaustive}\text{"})))\end{array}}$$

Applying (Case) rule to the second derivation, we have

$$\frac{\begin{array}{c}\Gamma \vdash e : t'_0 \rightsquigarrow E' \quad \Gamma_i \vdash_{\text{pat}} p_i : t_i \rightsquigarrow P_i \\ \vdash_{\text{sub}} t_i \leq_{d'_i} t'_0 \quad \Gamma \cup \Gamma_i \vdash e_i : t \rightsquigarrow E'_i \\ g_i = \lambda c.\mathsf{case} \; d'_i \; E' \; \mathsf{of} \; [\mathit{Just} \; P_i \rightarrow E'_i, \mathit{Nothing} \rightarrow c] \quad \text{for } i \in I\end{array}}{\begin{array}{c}\Gamma \vdash \mathsf{case} \; e \; \mathsf{of} \; [p_i \rightarrow e_i]_{i \in I} : t \rightsquigarrow \\ g_1 \; (... \; (g_n \; (\mathit{error} \; \text{"}\mathit{pattern \; is \; not \; exhaustive}\text{"})))\end{array}}$$

We first need to show that $d_i\ E \longrightarrow^* \mathit{Just}\ v_i$ iff $d'_i\ E'_i \longrightarrow^* \mathit{Just}\ v'_i$ and $d_i\ E \longrightarrow^* \mathit{Nothing}$ iff $d'_i\ E'_i \longrightarrow^* \mathit{Nothing}$ for $i = 1, ..., n$. That means the downcast coercions from the two derivations will succeed at the same pattern (1). By applying induction hypothesis, we can conclude that $E \overset{t_0}{\rightarrow}\overset{t'_0}{\leftarrow} E'$ (2).

By Definition 4, from (2) we can derive $d_i\ E = d'_i\ E'$ (3). That implies that (1) is valid.

What remains is to show that the E_i behaves the *same* as E'_i. From (3), we know that for $i = 1, ..., n$ we have that $d_i\ E = d'_i\ E'$. In case the pattern clause applies, we must have that $d_i\ E \longrightarrow^* \mathit{Just}\ v_i$ and $d'_i\ E' \longrightarrow^* \mathit{Just}\ v'_i$. Then we can immediately conclude that $v_i = v'_i$ (4). Now we would like to apply the induction hypothesis to the pattern bodies E_i and E'_i. Since (4), we conclude that $\theta_i = \theta'_i$ (5) where $v_i \lhd P_i \leadsto \theta_i$ and $v'_i \lhd P_i \leadsto \theta'_i$. From (5), we can derive that $\theta_i \overset{\Gamma_i}{\leftrightarrow} \theta'_i$ (6). From the assumption, we note that the global value bindings θ and θ' are in relation $\cdot \overset{\cdot}{\leftrightarrow} \cdot$. It follows from (6) immediately that $\theta \cup \theta_i \overset{\Gamma \cup \Gamma_i}{\leftrightarrow} \theta' \cup \theta'_i$. Thus, we can apply induction hypothesis to the pattern bodies to conclude that $\theta \cup \theta_i(E_i) \overset{t}{\longrightarrow}\overset{t}{\longleftarrow} \theta' \cup \theta'_i(E'_i)$. We have proven the case for case-expression rule.

□

B.3 Technical Proofs for Chapter 6

Lemma 12 (All Match Termination) *Let w be a word and p be a pattern. Then allmatch w p always terminates.*

Proof: By definition, the application of allmatch w p always applies the first pattern clause when w is not empty. The size of w decreases in the subsequent recursive calls, until w is empty. Then the remaining pattern clauses apply, in which the size of p decreases in the recursive calls. Thus, we conclude that the function application is terminating. □

Lemma 13 (All Match Correctness) *Let w be a word and p be a pattern. Both of the following are valid.*

1. *Let $w \lhd p \leadsto \theta$. Then $\theta \in$ allmatch w p.*
2. *Let $\theta \in$ allmatch w p. Then $w \lhd p \leadsto \theta$.*

Proof: We first consider the $\Rightarrow$ direction. Suppose $w \lhd p \rightsquigarrow \theta$, we would like to show that $\theta \in$ allmatch $w\ p$. We prove it by induction over the size of w:

Case $w = \langle\rangle$. Let $\langle\rangle \lhd p \rightsquigarrow \theta$. We would like to show

$$\theta \in \text{allmatch } \langle\rangle\ p \tag{B.20}$$

We verify B.20 by an inner induction over the structure of p.
The case of $p = ([w]\ x : t)$ is straight-forward. We consider a more interesting case $p = (p_1|p_2)$. By definition, $\langle\rangle \lhd (p_1|p_2) \rightsquigarrow \theta$ holds if $\langle\rangle \lhd p_1 \rightsquigarrow \theta$ (1) or $\langle\rangle \lhd p_2 \rightsquigarrow \theta$ (2). We apply the inner induction to (1) and (2) to obtain $\theta \in$ allmatch $w\ p_1$ or $\theta \in$ allmatch $w\ p_2$. In other words, in either case, $\theta \in$ allmatch $\langle\rangle\ (p_1|p_2)$ always holds because the result of allmatch $\langle\rangle\ (p_1|p_2)$ is the union of allmatch $\langle\rangle\ p_1$ and allmatch $\langle\rangle\ p_2$. Similar observation can be applied to the case $p = \langle p_1, p_2\rangle$.
Therefore, B.20 is valid.

Case $w = \langle l, w\rangle$. Let $\langle l, w\rangle \lhd p \rightsquigarrow \theta$. We would like to show

$$\theta \in \text{allmatch } \langle l, w\rangle\ p \tag{B.21}$$

To proof this case, we need to find a way to reduce of $\langle l, w\rangle$ to w, so that we can apply induction.
To do that, we need an auxillary property,

$$\langle l, w\rangle \lhd p \rightsquigarrow \theta \text{ implies } w \lhd p/l \rightsquigarrow \theta \tag{B.22}$$

which says that the match result will not change under pattern derivative operation. This property is valid, as we will provide the proof shortly.
With the above property, we can deduce that $w \lhd p/l \rightsquigarrow \theta$. To which we apply induction hypothese to conclude that $\theta \in$ allmatch $w\ p/l$ (3). By definition, we find that allmatch $\langle l, w\rangle\ p \longrightarrow$ allmatch $w\ p/l$ (4). With (3) and (4) we can conclude that B.21 is valid.

Hence we have verified the $\Rightarrow$ direction. The proof for the $\Leftarrow$ direction follows in a similar fashion, except that we need a different auxilary property to establish the induction.

$$w \lhd p/l \rightsquigarrow \theta \text{ implies } \langle l, w\rangle \lhd p \rightsquigarrow \theta \tag{B.23}$$

which is valid, too. The proof follows immediately. $\square$

We now verify the properties which we used in the previous proof is valid. Let $w/l = w'$ such that $\langle l, w'\rangle \sim w$.

Lemma 26 (Matching Preservation) *Let $p/l = p'$ and $w/l = w'$. Then $w \lhd p \rightsquigarrow \theta$ iff $w' \lhd p' \rightsquigarrow \theta$*

The proof of this lemma is straight-forward by the induction over the evaluation of p/l.

Lemma 14 (POSIX/Longest Match Termination) *Let v be a value and p be a pattern, Then longmatch w p always terminates.*
The proof is similar to the proof of Lemma 12.

Lemma 15 (POSIX/Longest Match Correctness) *Let p be a pattern and w be a value, longmatch w $p \longrightarrow^* Just\ \theta$ iff $w \lhd_{lm} p \rightsquigarrow \theta$.*
The proof is similar to the proof of Lemma 13.

Lemma 16 *Let $pd(l \shortmid t) = \{t_1, ..., t_n\}$. Then $|t/l| = (t_1|...|t_n)$.*

Proof: We prove by induction over the structure of t. We only consider the most interesting case $t = \langle r_1, r_2\rangle$ where $\langle\rangle \in t_1$. Our goal is to show that

$$|\langle r_1, r_2\rangle/l| = (t_1|...|t_n) \text{ and } pd(l \shortmid \langle r_1, r_2\rangle) = \{t_1, ..., t_n\}$$

We know that

$$\langle r_1, r_2\rangle/l = (\langle r_1/l, r_2\rangle|r_2/l) \quad \text{(B.24)}$$

Therefore

$$|\langle r_1, r_2\rangle/l| = |(\langle r_1/l, r_2\rangle|r_2/l)| \quad \text{(B.25)}$$

By definition of $pd(l \shortmid t)$, we note that

$$pd(l \shortmid \langle r_1, r_2\rangle) = pd(l \shortmid r_1) \odot r_2 \cup pd(l \shortmid r_2) \quad \text{(B.26)}$$

We apply induction hypothesis to B.25 and B.26 to conclude that

$$|\langle r_1/l\rangle| = (t'_1|...|t'_n) \text{ where } pd(l \shortmid r_1) = \{t'_1, ..., t'_n\} \quad \text{(B.27)}$$

and

$$|\langle r_2/l\rangle| = (s'_1|...|s'_n) \text{ where } pd(l \shortmid r_2) = \{s'_1, ..., s'_n\} \quad \text{(B.28)}$$

By definition of $\cdot \odot \cdot$, we have

$$pd(l \shortmid r_1) \odot r_2 = \{\langle(t'_1|...|t'_n), r_2\rangle\} \quad \text{(B.29)}$$

From B.27, we can conclude that

$$\langle|\langle r_1/l\rangle|, t_2\rangle = \langle(t'_1|...|t'_n), r_2\rangle \quad \text{(B.30)}$$

Applying B.28, B.29 and B.30 we conclude that

$$|\langle r_1, r_2\rangle/l| = \langle(t'_1|...|t'_n), r_2\rangle|s'_1|...|s'_n \quad \text{(B.31)}$$

and

$$pd(l \shortmid \langle r_1, r_2\rangle) = \{\langle(t'_1|...|t'_n), r_2\rangle\} \cup \{s'_1, ..., s'_n\} \quad \text{(B.32)}$$

which is what we want to show. $\square$

Lemma 19 (Make Empty maintains isomorphism) *Let p be a pattern in derivative form. Let stript $p = t$ such that $\vdash_{empty}$ $\langle\rangle \in t$. Then longmatch $\langle\rangle$ $p \longrightarrow$ Just θ where $mkEmpty_t \overset{p}{\sim} \theta$.*

Proof: We prove by induction over the structure of p:

Case $([w]\ x : t)$*:* From the assumption, we have $\vdash_{\text{empty}}$ $\langle\rangle : t$, According to the definition of longmatch , we can immediately conclude that longmatch $\langle\rangle$ $([w]\ x : t) \longrightarrow^* Just\ \{(w/x)\}$. Note that stript $([w]\ x : t) = t$. By Definition 3, $mkEmpty_t$ guarantees the following,

$$\langle\rangle \overset{t}{\leftrightarrow} mkEmpty_t$$

Since $w/w = \langle\rangle$ clearly holds, we can conclude that $mkEmpty_t \overset{([w]\ x:t)}{\sim} \{(w/w)\}$. Thus we have verified this case.

Case $\langle p_1, p_2\rangle$*:* Let stript $\langle p_1, p_2\rangle = \langle$stript p_1, stript $p_2\rangle = \langle t'_1, t'_2\rangle$. Since $\vdash_{\text{empty}}$ $\langle\rangle \in \langle t'_1, t'_2\rangle$, we have $\vdash_{\text{empty}}$ $\langle\rangle \in t'_1$ and $\vdash_{\text{empty}}$ $\langle\rangle \in t'_2$. By definition, we have $mkEmpty_{(t'_1,t'_2)} = (mkEmpty_{t'_1}, mkEmpty_{t'_2})$. In addition, we evaluate

```
      longmatch ⟨⟩ ⟨p₁, p₂⟩
⟶    case (longmatch ⟨⟩ p₁) of
      Just θ₁ → case (longmatch ⟨⟩ p₂) of
            Just θ₂ → θ₁ ∪ θ₂
            Nothing → Nothing
      Nothing → Nothing
```

We apply induction hypothesis to conclude that longmatch $\langle\rangle$ $p_1 \longrightarrow^*$ $Just\ \theta_1$ and longmatch $\langle\rangle$ $p_2 \longrightarrow^* Just\ \theta_2$ where $mkEmpty_{t'_1} \overset{p_1}{\sim} \theta_1$ and $mkEmpty_{t'_2} \overset{p_2}{\sim} \theta_2$. By Definition 6 we conclude $mkEmpty_{\langle t'_1, t'_2\rangle} \overset{\langle p_1,p_2\rangle}{\sim} \theta_1 \cup \theta_2$.

Case $(p_1|p_2)$*:* Let stript $(p_1|p_2) = ($stript $p_1|$stript $p_2) = (t_1|t_2)$. Since $\vdash_{\text{empty}}$ $\langle\rangle \in (t_1|t_2)$, we have either

1. $\vdash_{\text{empty}}$ $\langle\rangle \in t_1$ and $\vdash_{\text{empty}}$ $\langle\rangle \in t_2$ or;
2. $\vdash_{\text{empty}}$ $\langle\rangle \in t_1$ or;
3. $\vdash_{\text{empty}}$ $\langle\rangle \in t_2$.

We consider the first case.

Suppose $\vdash_{\text{empty}} \langle\rangle \in t_1$ and $\vdash_{\text{empty}} \langle\rangle \in t_2$, Note that $\langle\rangle$ inhabits in both alternative. As we mentioned, we favor the definition $mkEmpty_{(t_1|t_2)} = L\ mkEmpty_{t_1}$. We now consider the evaluation of $posixMatch\ (p_1|p_2)\ \langle\rangle$ as follows,

$$\begin{array}{rl} & \text{longmatch } \langle\rangle\ (p_1|p_2) \\ \longrightarrow & \mathsf{case}(\text{longmatch } \langle\rangle\ p_1)\ \mathsf{of} \\ & Just\theta_1 \rightarrow \theta_1 \\ & Nothing \rightarrow (\text{longmatch } \langle\rangle\ p_2) \end{array}$$

We apply induction hypothesis to conclude that longmatch $\langle\rangle\ p_1 \longrightarrow Just\ \theta_1$ and longmatch $\langle\rangle\ p_2 \longrightarrow Just\ \theta_2$ where $mkEmpty_{t_1} \overset{p_1}{\sim} \theta_1$ and $mkEmpty_{t_2} \overset{p2}{\sim} \theta_2$.

Thus,

$$\begin{array}{rl} & \mathsf{case}\ (\text{longmatch } \langle\rangle\ p_1)\ \mathsf{of} \\ & Just\ \theta_1 \rightarrow Just\ \theta_1 \\ & Nothing \rightarrow (\text{longmatch } \langle\rangle\ p_2) \\ \longrightarrow^* & Just\ \theta_1 \end{array}$$

Therefore, we can conclude that $L\ mkEmpty_{t_1} \overset{(p_1|p_2)}{\sim} \theta_1$. We have proven the first case (1), out of the three difference cases. The other two cases (2 and 3) can be verified in similar way, of which we omit the detail. □

Lemma 20 (Injection maintains isomorphism) *Let p and p' be two patterns such that $p/l = p'$. Let stript $p = t$ and stript $p' = t/l$. Let θ be a value binding environment. Let v be a System F value such that $v : [\![t/l]\!]$ and $v \overset{p'}{\sim} \theta$. Then $(pdtInj_{(l,t)}\ v) \overset{p}{\sim} \theta$.*

Proof: We prove by induction over the structure of p.

Case ([w] x:t): Applying $p = ([w]\ x : t)$ to the assumption, we have the following,

$$p' = p/l = ([\langle w, l\rangle]\ x : (t/l)) \tag{B.33}$$

$$\text{stript } p = t \tag{B.34}$$

$$\text{stript } p/l = (t/l) \tag{B.35}$$

$$v \overset{([\langle w,l\rangle]\ x:(t/l))}{\sim} [(x, w')] \tag{B.36}$$

Our goal is to show that

$$pdtInj_{(l,t)}\ v \overset{([w]\ x:t)}{\sim} \{(w'/x)\} \tag{B.37}$$

Let $w'' = w'/\langle w, l\rangle$ (1). From B.36, we can deduce that $w'' \overset{(t/l)}{\approx} v$. By Definition 3, we have $\langle l, w''\rangle \overset{t}{\approx} pdtInj_{(l,t)}\ v$ (2). From (1) we deduce that $\langle l, w'' \rangle = w'/w$ (3) Thus by Definition 6, from (2) and (3) we can conclude that B.37 is valid.

Case $\langle p_1, p_2 \rangle$*:* We first need to decide what p/l is. Depending on whether $\vdash_{\text{empty}} \langle\rangle \in \text{stript } p_1$, $\langle p_1, p_2 \rangle / l$ gives two possible outcomes.

$$\langle p_1, p_2 \rangle / l = \begin{cases} \langle p_1/l, p_2 \rangle & ; \neg(\vdash_{\text{empty}} \langle\rangle \in \text{stript } p_1) \\ \langle p_1/l, p_2 \rangle | \langle \epsilon(p_1), p_2/l \rangle & ; \textit{otherwise} \end{cases}$$

We first consider the simpler case. Suppose $\neg(\vdash_{\text{empty}} \langle\rangle \in \text{stript } p_1)$, we have $\langle p_1, p_2 \rangle / l = \langle p_1/l, p_2 \rangle$. Applying this information to the assumption we have

$$\text{stript } \langle p_1, p_2 \rangle = \langle \text{stript } p_1, \text{stript } p_2 \rangle = \langle t_1, t_2 \rangle \tag{B.38}$$

$$(\langle t_1, t_2 \rangle / l) = \langle (t_1/l), t_2 \rangle \tag{B.39}$$

$$\text{stript } \langle p_1/l, p_2 \rangle = \langle \text{stript } p_1/l, \text{stript } p_2 \rangle = \langle (t_1/l), t_2 \rangle \tag{B.40}$$

Let $v = (v_1', v_2)$ such that $(v_1', v_2) : [\![\langle (t_1/l), t_2 \rangle]\!]$ and

$$(v_1', v_2) \overset{\langle p_1/l, p_2 \rangle}{\sim} \theta_1 \cup \theta_2 \tag{B.41}$$

Our goal is to show

$$\textit{pdtInj}_{(l, \langle t_1, t_2 \rangle)} \ (v_1', v_2) \overset{\langle p_1, p_2 \rangle}{\sim} \theta_1 \cup \theta_2 \tag{B.42}$$

By definition of $\textit{pdtInj}_{(l, \langle t_1, t_2 \rangle)}$ we have,

$$\begin{array}{ll} & \textit{pdtInj}_{(l, \langle t_1, t_2 \rangle)} \ (v_1', v_2) \\ \longrightarrow & (\textit{pdtInj}_{(l, t_1)} \ v_1', v_2) \end{array}$$

To verify B.42, we need to show

$$\textit{pddInj}_{(l, t_1)} \ v_1' \overset{p_1}{\sim} \theta_1 \tag{B.43}$$

and

$$v_2 \overset{p_2}{\sim} \theta_2 \tag{B.44}$$

From B.41, we can deduce that both B.44 and

$$v_1' \overset{p_1/l}{\sim} \theta_1 \tag{B.45}$$

hold. Applying induction hypothesis to B.45, we can conclude that B.43 is valid. Therefore we conclude that B.42 holds.

We now consider the other (the harder) case. Suppose $\vdash_{\text{empty}} \langle\rangle \in \text{stript } p_1$, we have $\langle p_1, p_2 \rangle / l = (\langle p_1/l, p_2 \rangle | \langle \text{mkEmpPat } p_1, p_2/l \rangle)$. Applying the information to the assumption we have

$$(\langle t_1, t_2 \rangle / l) = \langle (t_1/l), t_2 \rangle | \langle \langle\rangle, (t_2/l) \rangle \tag{B.46}$$

Our goal is to show

$$pdtInj_{(l,\langle t_1,t_2\rangle)}\ v \overset{\langle p_1,p_2\rangle}{\sim} \theta_1 \cup \theta_2 \tag{B.47}$$

where $pdtInj_{(l,\langle t_1,t_2\rangle)}$ is defined as follows,

$$\begin{array}{l} pdtInj_{(l,\langle t_1,t_2\rangle)}\ v_l\ v = \\ \quad \mathsf{case}\ v\ \mathsf{of} \\ \quad L\ (v_1,v_2) \rightarrow (pdtInj_{(l,t_1)}\ v_1, v_2) \\ \quad R\ ((),v_2) \rightarrow (mkEmpty_{t_1}, v_2) \end{array}$$

We perform a case analysis on the value v:

Let $v = L\ (v_1,v_2)$ where $(v_1,v_2) : [\![\langle (t_1/l), t_2\rangle]\!]$ and

$$L\ (v_1,v_2) \overset{(\langle p_1/l,p_2\rangle | \langle \text{mkEmpPat}\ _{p_1,p_2}\rangle)}{\sim} \theta_1 \cup \theta_2 \tag{B.48}$$

By Definition 6 from B.48 we can deduce that

$$(v_1,v_2) \overset{\langle p_1/l,p_2\rangle}{\sim} \theta_1 \cup \theta_2 \tag{B.49}$$

Applying Definition to B.49, we have

$$v_1 \overset{p_1/l}{\sim} \theta_1 \tag{B.50}$$

and

$$v_2 \overset{p_2}{\sim} \theta_2 \tag{B.51}$$

Applying induction hypothesis to B.50 we can conclude that

$$pdtInj_{(l,t)}\ v_1 \overset{p_1}{\sim} \theta_1 \tag{B.52}$$

By Definition 6, from B.51 and B.52, we conclude that

$$(pdtInj_{(l,t)}\ v_1, v_2) \overset{\langle p_1,p_2\rangle}{\sim} \theta_1 \cup \theta_2 \tag{B.53}$$

Since $pdtInj_{(l,\langle t_1,t_2\rangle)}\ v_l\ v \longrightarrow^* (pdtInj_{(l,t)}\ v_1, v_2)$, we conclude that B.47 is valid.

The case of $v = R\ ((),v_2)$ where $((),v_2) : [\![\langle\rangle, (t_2/l)]\!]$ is similar.

This concludes the case of $\langle p_1,p_2\rangle$.

The other cases are similar, hence we omit the details. □

Lemma 21 ("From" maintains isomorphism) *Let p and p' be two patterns such that p' is the pruned version of p. Let stript $p = t$ and stript $p' = |t|$. Let θ be a value binding environment. Let v be a System F value such that $v : [\![|t|]\!]$ and $v \overset{p'}{\sim} \theta$,* *Let $from$ be the function that is derived from the simplification going from t to $|t|$. Then $(from\ v) \overset{p}{\sim} \theta$.*

Proof: As we mentioned earlier, the function $from$ are defined in terms of the basic coercion functions $from_{(E_1)}$ to $from_{(E_5)}$. If we can show that for $i = \{1, ..., 5\}$, $from_{(E_i)}$ preserves the isomorphic relation, by a simple induction over the series simplification steps, we can conclude that $from$ must preserve the relation, too.

We would like to verify that $from_{(E_i)}$ indeed preserves the isomorphic relation $\cdot \overset{\cdot}{\sim} \cdot$.

Subcase (E3):

$$from_{(E3)}\ v = L\ v$$

We note that (E3) corresponds to a pruning operation which turns $(p_1|p_2)$ into p_1. Let v be a value such that $v \overset{p_1}{\sim} \theta$ for some value binding θ. By Definition 6, we can conclude that $from_{(E3)}\ v \overset{(p_1|p_2)}{\sim} \theta$.

Subcase (E5):

$$from_{(E5)}\ v = ((), v)$$

(E5) corresponds to a pruning operation that turns $\langle [w]\ x : \langle\rangle, p\rangle$ into p. Let v be a value such that $v \overset{p}{\sim} \theta$ for some value binding θ. By Definition 6, we can conclude that $from_{(E5)}\ v \overset{\langle [w]\ x:\langle\rangle, p\rangle}{\sim} \{w/x\} \cup \theta$.

The other sub-cases are similar.

Thus we conclude that $from_{(E_i)}$ preserves the isomorphic relation. □

Lemma 23 (Downcast is faithful w.r.t. POSIX matching) *Let $stript\ p = t_1$ and $\vdash_{sub} t_1 \leq_d t_2$. Let w be a System* F* *value such that $w : t_2$ and v_2 be a System F value such that $w \overset{t_2}{\leftrightarrow} v_2$. Then we have*

1. *$d\ v_2 \longrightarrow^* Just\ v_1$ iff $longmatch\ w\ p \longrightarrow^* Just\ \theta$, where $v_1 \overset{p}{\sim} \theta$;*
2. *$d\ v_2 \longrightarrow^* Nothing$ iff $longmatch\ w\ p \longrightarrow^* Nothing$.*

Proof: (Sketch) First of all we would like to show that $d\ v_2 \longrightarrow^* Just\ v_1$ implies longmatch $w\ p \longrightarrow^* Just\ \theta$, where $v_1 \overset{p}{\sim} \theta$. We prove by induction over size of w.

Case $w \sim \langle\rangle$:

$$\frac{\begin{array}{l} d : \forall[\![t_2]\!] \rightarrow Maybe\ [\![t_1]\!] \\ d\ v = if\ \ isEmpty_{t_2}\ v\ then\ Just\ mkEmpty_{t_1} \\ \qquad\qquad else\ ... \end{array}}{\vdash_{\text{sub}}\ t_1 \leq_d t_2}$$

First of all we know that $isEmpty_t$ satisfies the properties stated in Definition 3. From the assumption $\langle\rangle \overset{t_2}{\leftrightarrow} v_2$ we can deduce that $isEmpty_{t_2}\ v_2 \longrightarrow^* True$. Therefore, we have $d\ v_2 \longrightarrow Just\ mkEmpty_{t_1}$, from which we also find that $\vdash_{\text{empty}}\ \langle\rangle \in t_1$.

To proceed, we need to show

$$\text{longmatch } \langle\rangle\ p\ \longrightarrow Just\ \theta \text{ where } mkEmpty_{t_1} \overset{p}{\sim} \theta \tag{B.54}$$

By applying Lemma 19, we can immediately conclude that B.54 is valid.

Case $w \sim \langle l, w'\rangle$*:*

$$\dfrac{\dfrac{\dots}{\{t_1 \leq_d t_2\} \vdash_{\text{sub}} (t_1/l)' \leq_{d'} (t_2/l)'} \quad \begin{array}{l} to : \forall[\![(t'/l)]\!] \to [\![(t'/l)']\!] \\ from : \forall[\![(t/l)']\!] \to [\![(t/l)]\!] \\ d : \forall[\![t_2]\!] \to Maybe\ [\![t_1]\!] \\ d\ v = if\ \ isEmpty_{t_2}\ v\ then\ Just\ mkEmpty_{t_1} \\ \qquad\qquad else\ case\ pdtProj_{(l,t_2)}\ v\ of \\ \qquad\qquad\quad Just\ (v_l, v_2') \to case\ d'\ (to\ v_2')\ of \\ \qquad\qquad\qquad\qquad Just\ v_1' \to Just\ (pdtInj_{(l,t_1)}\ v_l\ (from\ v_1')) \\ \qquad\qquad\qquad\qquad Nothing \to Nothing \\ \qquad\qquad\quad Nothing \to Nothing \end{array}}{\vdash_{\text{sub}}\ t_1 \leq_d t_2}$$

Since v_2 is not empty, the evaluation of $d\ v_2$ looks like the following,

$$\begin{array}{ll} & d\ v_2 \\ \longrightarrow & case\ pdtProj_{(l,t_2)}\ v\ of \\ & Just\ (v_l, v_2') \to case\ d'\ (to\ v_2')\ of \\ & \qquad\qquad\qquad Just\ v_1' \to Just\ (pdtInj_{(l,t_1)}\ v_l\ (from\ v_1')) \\ & \qquad\qquad\qquad Nothing \to Nothing \\ & Nothing \to Nothing \\ \longrightarrow & Just\ (pdtInj_{(l,t_1)}\ v_l\ v_1') \end{array}$$

where $w' \overset{(t_2/l)}{\leftrightarrow} v_2'$.

To simplify the proof we first make an assumption. Assuming $(t_1/l) \equiv (t_1/l)'$ (A), which means that no simplification takes place and functions to and $from$ are identity functions. We apply induction hypothesis to conclude that longmatch $w'\ p/l \longrightarrow^* Just\ \theta$ where $v_1' \overset{p/l}{\sim} \theta$. Now we are ready to apply Lemma 20 to conclude that $pdtInj_{(l,t_1)}\ v_l\ v_1' \overset{p}{\sim} \theta$, which is what we wanted to show.

However the assumption (A) does not always hold for any p and t_1.

We consider the case when (A) is lifted. For the proof to get through we just need to show that the function *from* preserves the isomorphic relation $\cdot \overset{\cdot}{\sim} \cdot$. According to Lemma 21, the *from* function preserves the isomorphicm relation. Thus, lifting (A) does not invalidate the proof.

We have proven the "only-if" direction.

The other direction of the lemma is straight-forward. It is obvious that longmatch $w\ p \longrightarrow Just\ \theta$ implies w : stript p. We note that there exists another downcast property which guarantees that $d\ v_2 \longrightarrow^* Just\ v_1$. For simplicity, we omit the details. $\square$

Lemma 22 *Let* p, θ *be a System* F^* *pattern and a System* F^* *value environment repsectively. Let* $\Gamma \vdash_{pat} p : t \rightsquigarrow P$. *Let* v *be a System F value such that* $v : [\![t]\!]$ *and* $v \overset{p}{\sim} \theta$. *Let* θ_F *be a System F value environment, such that* $v \lhd_F P \rightsquigarrow \theta_F$. *Then* $\forall x.\theta(x) \overset{\Gamma(x)}{\approx} \theta_F(x)$

Proof: We prove the lemma straight-forwardly by induction over the structure of p. *Case* $(x : t)$*:* From the assumption, we note that

$$\{x : t\} \vdash_{\mathrm{pat}} (x : t) : t \rightsquigarrow x \tag{B.55}$$

and

$$v \overset{([]\ x:t)}{\sim} \{(w/x)\} \tag{B.56}$$

for some System F^* value w. From B.56, we conclude that

$$w \overset{t}{\approx} v \tag{B.57}$$

From the assumption, we also note that

$$v \lhd_{\mathrm{F}} x \rightsquigarrow \{(v/x)\} \tag{B.58}$$

From B.56 and B.58 we can immediately conclude $\theta(x) \overset{\Gamma(x)}{\approx} env(x)$ is valid.

Case $\langle p_1, p_2\rangle$*:* from the assumption, we note that

$$\frac{\Gamma_1 \vdash_{\mathrm{pat}} p_1 : t_1 \rightsquigarrow P_1 \quad \Gamma_2 \vdash_{\mathrm{pat}} p_2 : t_2 \rightsquigarrow P_2}{\Gamma_1 \cup \Gamma_2 \vdash_{\mathrm{pat}} \langle p_1, p_2\rangle : \langle t_1, t_2\rangle \rightsquigarrow (P_1, P_2)} \tag{B.59}$$

Let $v = (v_1, v_2)$ we have

$$(v_1, v_2) \overset{\langle p_1,p_2\rangle}{\sim} \theta_1 \cup \theta_2 \tag{B.60}$$

From B.60, we can deduce that

$$v_1 \overset{p_1}{\sim} \theta_1 \tag{B.61}$$

and

$$v_2 \overset{p_2}{\sim} \theta_2 \tag{B.62}$$

From the assumption we note that

$$\frac{v_1 \lhd_{\mathrm{F}} P_1 \rightsquigarrow \theta_1 \quad v_2 \lhd_{\mathrm{F}} P_2 \rightsquigarrow \theta_2}{(v_1, v_2) \lhd_{\mathrm{F}} (P_1, P_2) \rightsquigarrow \theta_1 \cup \theta_2} \tag{B.63}$$

Applying induction hypothesis to B.61, B.62 and the premises of B.59 and B.63, we have that

$$\forall x.\theta_1(x) \overset{\Gamma_1(x)}{\approx} \theta_1(x) \tag{B.64}$$

and

$$\forall x.\theta_2(x) \overset{\Gamma_2(x)}{\approx} \theta_2(x) \tag{B.65}$$

From B.64 and B.65 we can conclude that

$$\forall x.(\theta_1 \cup \theta_2)(x) \overset{(\Gamma_1 \cup \Gamma_2)(x)}{\approx} (\theta_1 \cup \theta_2)(x) \tag{B.66}$$

We have verified this case. □