Improved Uniqueness Typing for Haskell

Arie Middelkoop
amiddelk@cs.uu.nl

Master's Thesis
INF/SCR-06-04

Daily Supervisor: dr. J. Hage
Supervisor: prof.dr. S.D. Swierstra

Center for Software Technology,
Department of Information and Computing Sciences,
Universiteit Utrecht,
Utrecht, The Netherlands.

October 13, 2006

Abstract

An unused value has special properties in a pure functional language. An optimising compiler can safely throw away code that produces such a value. Likewise, a value that is referred to once has special properties that can be exploited by an optimising compiler [30]. In Clean [34], a type is called unique if the corresponding values are only referred to once. A compiler implementing a uniqueness type system checks, enforces, or infers the uniqueness property of types.

Uniqueness typing is incorporated into the Clean compiler [2]. The input/output model of Clean relies heavily upon this feature. Haskell does not need uniqueness typing in order to perform IO [28], but can benefit from optimisations that are possible on unique values, such as destructive updates on arrays.

In this master thesis, we discuss an implementation of uniqueness typing for Haskell. We take a different strategy than Clean, allowing us to overcome a limitation of the typing in Clean. In Clean, it is not allowed to have a data type with a shared spine and unique components, since uniqueness propagation forces the spine to be unique as well. We treat the components independently from the spine and show the consequences.

Acknowledgements

I want to express my thanks to the people of the Software Technology Group in Utrecht, and the students of the ST-lab specifically. Their insights, and distractions, proved to be a valuable asset during the construction of this master's thesis. Of these people, there are two people that deserve some acknowledgment in particular: my daily supervisor Jurriaan and my roommate Gerrit.

I thank my supervisor Jurriaan for the countless amount of conversations and the patience to comment on my work. I doubt he knew where he was getting into when I was looking for a supervisor and he volunteered to be mine. He knows so now. However, seriously, doing my master's thesis project at the University of Utrecht has been an enjoyable experience, of which Jurriaan has made his contribution.

Also special thanks for Stefan for helping me in fixing spelling and grammar erors in this master's thesis.

Among the students of the ST-lab, Gerrit has been the greatest contributor to the experiences at the ST-lab. We've had a countless number of discussions. Sometimes about my work. Sometimes about Gerrit his work. However, often about other subjects. Despite distracting each other from our work, I think we both managed to have fun and put down a lot of work.

Finally, I thank everybody I worked with this last year, including the ST-lab computer that somehow still works.

Contents

Contents v

1 Introduction 1
- 1.1 Usage analysis, typing, and uniqueness . . . 2
- 1.2 Goals . . . 4
- 1.3 Organisation of this thesis . . . 5

2 Prerequisites 7
- 2.1 Why EH? . . . 7
- 2.2 Language features of EH . . . 8
  - 2.2.1 Quantifiers and qualifiers everywhere . . . 8
  - 2.2.2 Polymorphic kinds . . . 9
  - 2.2.3 Extensible records . . . 9
- 2.3 Experience . . . 10

3 Lambda-calculus without bindings 11
- 3.1 Intuition . . . 11
  - 3.1.1 Annotations . . . 11
  - 3.1.2 Checking the upper-bound usage-count annotation . . . 13
- 3.2 The next step . . . 15
  - 3.2.1 Annotations . . . 16
  - 3.2.2 Checking the annotations . . . 16
  - 3.2.3 Checking the constraints . . . 19
- 3.3 Language . . . 20
- 3.4 Types, Annotations and Constraints . . . 21
  - 3.4.1 The annotations . . . 21
  - 3.4.2 Triple consistency . . . 25
- 3.5 Gathering constraints for first-order functions . . . 26
- 3.6 Gathering constraints for higher-order functions . . . 28
- 3.7 Checking constraints . . . 31
  - 3.7.1 Coercion constraints . . . 31
  - 3.7.2 Aggregation constraint . . . 31
- 3.8 Strictness? . . . 32
- 3.9 Conclusion . . . 33

4 Polyvariant analysis with monomorphic bindings 35
- 4.1 Example . . . 35
- 4.2 Cardinality variables and type rules . . . 37
- 4.3 The inferencer . . . 38
- 4.4 Minimal solution . . . 41
- 4.5 Adding a non-recursive let . . . 43
- 4.6 The inferencer revisited . . . 47

4.7 Graph reduction . . . 49
4.7.1 Preprocessing reductions . . . 49
4.8 Reduction of intermediates . . . 50
4.8.1 Remarks about graph reduction . . . 54
4.9 Conclusion . . . 54

5 Recursion **55**
5.1 Example . . . 55
5.2 Monovariant recursion . . . 56
5.3 Helping the inferencer with manual constraint sets . . . 56
5.3.1 Programmer specified constraint sets . . . 57
5.3.2 The problem with sacred cardinality variables . . . 57
5.3.3 Entailment check . . . 57
5.3.4 Final remarks about helping the inferencer . . . 59
5.4 Full polyvariant recursion . . . 59
5.5 Aliassing and cyclic definitions . . . 60
5.6 Conclusion . . . 61

6 Polymorphic bindings **63**
6.1 Example . . . 63
6.2 Ad-hoc strategy 1: defaulting . . . 64
6.3 Ad-hoc strategy 2: equality . . . 64
6.4 Strategy 3: graph duplication . . . 65
6.5 Other complications . . . 66
6.6 Types in constraints . . . 66
6.7 Existential types . . . 67
6.8 Conclusion . . . 68

7 Parallel execution paths **69**
7.1 Adding an if-then-else expression . . . 69
7.2 Lower bounds and if-then-else . . . 72
7.3 Function application and sequential occurrences of an identifier . . . 72
7.4 Conclusion . . . 73

8 Adding algebraic data types **75**
8.1 Example . . . 75
8.2 Exposed annotations . . . 76
8.2.1 Implementation . . . 77
8.3 Expanded types . . . 77
8.4 Pattern matches . . . 79
8.5 Case expressions . . . 80
8.6 β annotations and constructors . . . 80
8.7 Conclusion . . . 81

9 Polarity and Under-the-arrow analysis **83**
9.1 Introduction . . . 83
9.2 Example . . . 84
9.3 Constraint gathering . . . 85
9.4 Constraint interpretation . . . 86
9.5 Graph simplification . . . 87
9.6 Conclusion . . . 87

10 Dealing with overloading **89**
10.1 Example . . . 89
10.2 Conservative approach . . . 89

10.3 Improved approach . . . 90
10.4 Advanced approach . . . 90
10.5 Probing one step further . . . 91
10.6 Conclusion . . . 91

11 Separate compilation **93**
11.1 Module system . . . 93
11.2 Separate compilation . . . 94

12 Inspecting Results **95**
12.1 HTML Pretty Printing . . . 95
12.2 Signatures . . . 96

13 Conclusion **99**
13.1 Related work . . . 99
13.2 Complexity comparison to Clean . . . 100
13.3 Future work . . . 101

A Implementation **103**
A.1 Installation . . . 103
A.2 Files and architecture . . . 104

Bibliography **105**

Chapter 1

Introduction

Jan is a student with, aside from programming, only one passion in life: a daily shot of coffee at school in the morning. But, Jan is lazy and does not want to bring his own coffee cup with him. Each time he buys a plastic cup and throws it away afterwards, since he does not need the cup anymore after a single use. One day Jan thinks about the cost of buying all these plastic cups and the cost for the government to collect the waste. Jan suddenly gets a brilliant idea: recycling. If he puts his cup in a dish washing machine (normally reserved for teachers) after using the coffee and picks it up the following day, then he does not need to buy plastic cups all the time. Neither does he need to throw them away. This gives him the best of both worlds. He can both be lazy and reduce the cost of acquiring and disposing of the cup.

Taking notice of this life lesson, consider lazy purely functional programming languages. A program written in such a language produces much more garbage than a program written in an imperative language. Referential transparency is to blame. With referential transparency, any expression may be replaced with its result without changing the behavior of the program. Consider the expression:

$$\begin{array}{l}\textbf{let } a \;= \mathit{array}\ (1, 10000)\ (\mathit{zip}\ [1\,..]\ (\mathit{repeat}\ 5)) \\ \quad\; a' = a \backslash [(1, 10)] \\ \textbf{in } \;(a, a')\end{array}$$

Here, a is a huge array with the value five for each element. The array a' is obtained from a by changing the first element to ten. Suppose that this is done by updating the memory where a points to, which is typical for assignments in imperative languages.

Referential transparency tells us that we are allowed to replace an expression by its value. Due to the side-effect, a and a' point to the same value. So, the behavior of this program is that a tuple is constructed where the first component points to an array with the first element having the value ten.

But, if we evaluate this expression by beta reduction, we get a different result. If we beta reduce the above expression a few steps, we get the expression:

$$\begin{array}{l}(\mathit{array}\ (1, 10000)\ (\mathit{zip}\ [1\,..]\ (\mathit{repeat}\ 5)) \\ ,\mathit{array}\ (1, 10000)\ (\mathit{zip}\ [1\,..]\ (\mathit{repeat}\ 5)) \backslash [(1, 10)])\end{array}$$

Further beta reduction results in a tuple being constructed, with the first component pointing to an array, of which the first component has the value five. This tuple differs from the tuple above, so the two evaluations of the expressions give different results. Referential transparency does not hold! The problem is that we changed the value where a pointed to, in order to obtain the value of a'. In general, referential transparency and destructive updates cannot be combined.

To guarantee referential transparency, compilers generate code that treat values as atomic. In other words, code is generated in such a way that the value where a points to, is never changed. This is done by making a copy of the

original value before performing a destructive update. So, in our example, a copy of a is made before updating it. This has a considerable impact on performance.

However, performing a destructive update without making a copy does not always violate referential transparency. Consider the following variation of the above expression:

```
let a  = array (1, 10000) (zip [1 . .] (repeat 5))
    a' = a\[(1, 10)]
in  a'
```

Suppose that a points to a value v, and v is only used once. After making a copy of v to perform the destructive update, v becomes garbage, since there are no references to it anymore. If we directly recycle v, then a copy does not need to be made, and v can be changed instead. Referential transparency holds in this case (verify this).

The lesson that we learn here is that values that are used at most once can by recycled or updated without violating referential transparency. The \ operator is normally implemented as:

```
arr\changes
    = unsafePerformIO
    $ do arr' ← copySpineOfArray arr
         updateInPlace changes arr'
         return arr'
```

If it can be guaranteed that the array is used at most once, then the implementation can be changed into:

```
arr\changes
    = unsafePerformIO
    $ do updateInPlace changes arr
      return arr
```

In case the array is used at most once, then the changed implementation does not copy the array and reuses the old value. The changed implementation of the \ operator does not have the overhead of copying and produces less garbage by reusing the old value of the array.

This optimisation does not only work for arrays, but for many more operations, such as updates on fields of a data type for example, or the addition of integers, as long as such values are used at most once. The identification of values that are used at most once, is the focus of this master's thesis.

1.1 Usage analysis, typing, and uniqueness

Usage analysis is the topic of research that deals with the identification of values that can be recycled. The interest for usage analysis originated from the inefficiency of updating large data structures in functional programming languages, such as arrays. A copy of the entire array is made just to update a single element. Because of this, arrays in functional programming languages have a bad impact on performance. A diversity of approaches to eliminate the inefficiency were investigated [36]. Among these approaches, usage analysis plays an important role, since values that are used at most once can be updated without making a copy, removing the inefficiency effectively.

Usage analysis on values turns out to be difficult as the representation of values can be infinite. Lazy infinite lists are commonly used in Haskell - for instance $fibs = 0 : 1 : zipWith\ (+)\ fibs\ (tail\ fibs)$. But how can uniqueness properties of such a value be described? With first-order logic, some invariants can be defined over infinite values, but the inference of such invariants from the definition of *fibs* is difficult, and the verification of invariants undecidable. However, the properties on values can be approximated by describing the properties on the types of these values. The Fibonacci

sequence may be infinite, but its type *List Int* is not. The type constructors in the type, in this case *List* and *Int*, can be individually given a usage property, for example $List^{20}$ Int^{10}, where the List is considered to be used up to twenty times, and each element of the list up to ten times, for some program that only evaluates a few Fibonacci numbers. Note that this is an approximation of the original problem, since the *Int* ranges over all elements of the list, and each element gets the same property, although given our intuition of the definition of *fibs*, lower numbers are used much more than higher numbers, since the lower numbers are used to obtain a higher number. The approximation makes it easier to deal with usage properties, hence usage analyses are defined in terms of type systems, at the cost of accuracy (see Figure 3.3 for an example why types cannot differentiate between individual bits of memory). In practice this is not much of a problem: if a programmer uses a list, then the elements of this list are typically used in a similar way, otherwise the programmer would have used two lists or another data type.

However, natural numbers as annotations on types are still too difficult to handle. If we would know such a number, we would also know that a program terminates, and the verification of termination is known to be undecidable for languages like Haskell. It is not always possible to know an exact number. Again, an approximation is needed. A usage property of zero of a value is interesting for a compiler, since that means that a compiler does not need to generate code for the construction of the value. Likewise, knowing that a value is used once is useful as we mentioned in the beginning of this chapter. But knowing that a value is used twice, thrice or maybe a zillion times is not that interesting for us, since there at not many optimisations possible for such a value. It can be used to help an optimising compiler in making a trade-off between code size and execution speed, or to identify often used values for caching purposes. However, more detail has a cost at runtime (the number of iterations of the fixpoint iteration process of Section 4.3 depends on the amount of detail), therefore a typical approximation is to distinguish 0 for not used, 1 for at most once, and $*$ for arbitrary usage.

This approximation is clearly visible in linear typing [37] (which has roots in linear logic [11]). In linear typing, values of a non-linear type can be used an arbitrary number of times, but for a type to be linear, the corresponding values must be used exactly once. If we demand that all types are linear, we get a pure linear type system (see Figure 1.1). An elegant aspect of this type system is that it is just a conventional type system with restrictions on the environment that guarantee that each identifier is used exactly once. An identifier can only be taken from the environment if it is the only occupant. An environment is partitioned into two environments at a function application, and identifiers are added to the environment at lambda abstractions. An identifier cannot be used more than once, because it is only once added to the environment. Likewise, an identifier has to be used, otherwise the identifier one of the *Int* or *Var* leafs in the AST of the program has the identifier in their environment, which causes the typing to fail for the leaf.

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

$$\frac{}{\emptyset \vdash n : Int}\ \text{E.INT}_L \qquad \frac{}{\{i \mapsto \tau\} \vdash i : \tau}\ \text{E.VAR}_L \qquad \frac{\Gamma_a \vdash a : \tau_a \quad \Gamma_f \vdash f : \tau_a \to \tau}{\Gamma_f, \Gamma_a \vdash f\ a : \tau}\ \text{E.APP}_L \qquad \frac{x \mapsto \tau_x, \Gamma \vdash b : \tau}{\Gamma \vdash \lambda x \to b : \tau_x \to \tau}\ \text{E.LAM}_L$$

Figure 1.1: Expression type rules (L)

In practice, a linear type system mixes linear type and non-linear types, demanding the environment restrictions on linear types, and no restrictions for non-linear types (see the type systems given by Wadler [37] for more information). In this master's thesis, we use a richer representation than just linear and non-linear to represent uniqueness properties (Section 3.1.1), but assume that linear types are annotated with a 1 (i.e. Int^1), and non-linear types with a $*$ (i.e. Int^*).

Linear types open up a can of optimizations [30, 35, 10, 26]. Let us consider a few of them. Linear typing solves the problem of array updates, since an array with a linear type can be updated without making a copy. But there are also optimizations for smaller values. A function is strict in its linear arguments, so any linear argument can be evaluated before passing it to a function. The place where a linear value is used, is known, so code can be inserted to allocate the value on a heap that is not garbage collected, and code inserted to manually deallocate the value after its use. Expressions with a linear type—functions that are used exactly once for example—can be safely inlined without causing the inliner to loop. These are all optimizations that improve the memory utilization and runtime of functional programs.

Clean [34] uses the results of usage analysis not only for optimisation, but requires it for the IO model [1]. The usage analysis of Clean is called *uniqueness typing*, although it has a close correspondence to linear typing [2, 13]. A type is called unique if it is used at most once (i.e. affine instead of linear). Haskell programs uses monads for IO [30]. A monad guarantees that interaction with the outside world is single-threaded, or, in other words, that there is a guaranteed evaluation order on computations that interact with the outside world, which is made explicit with the **do** notation. Clean programs do not use monads, but use uniqueness typing to enforce that interaction with the outside world is single-threaded. The world is represented by values of the type $World^1$. The initial function of a program gets a world but has to return it as well[1]: $main :: World^1 \rightarrow World^1$. Any function with side-effect, such as $readline :: World^1 \rightarrow (String, World^1)$, requires the current world as parameter and returns a new world. Suppose that w is an identifier representing a unique world. Passing w to a function means that w cannot be used elsewhere. Since the main function demands a unique world as result, this means that if we pass w to a function, the function has to return a unique world, otherwise there is no way to get a unique world. This effectively means that a world is threaded (without sharing) through all function calls, which ensures an evaluation order between functions with side-effects.

What distinguishes Clean from other compilers that perform usage analysis, is that the uniqueness typing is part of the language and performed in the front-end of the compiler. The programmer can interact with the uniqueness type system by writing uniqueness type signatures, or inspecting the signatures returned by the type inferencer. Production compilers such as GHC [12], typically perform some form usage analysis (linear type inference for example) on a core language in the back-end of the compiler, but that is not visible to the programmer. As we discover later in this thesis, it is difficult to see for a human being which uniqueness properties can be derived by the compiler and which not. Since the outcome of the compiler can have a severe impact on the performance of a program, these are actually properties of which the programmer wants to verify that they hold or enforce. An analysis in the back-end cannot provide the required interaction with the programmer.

1.2 Goals

The goal of this master's thesis is to explore uniqueness typing in the context of Haskell using a constraint-based uniqueness-type system. As mentioned in the previous section, some work has already been performed in this area for Clean. The question is how uniqueness typing for Clean translates to Haskell. We show that some of the ideas translate to Haskell, and present some improvements.

Clean uses a mechanism called *marking* that analyses term graphs to find out how often a particular identifier occurs on execution paths. For each occurrence of an identifier, the marking mechanism determines if the value is used for reading or writing. Based upon information about execution order and whether identifiers always occur on the same execution path or always in mutual exclusive execution paths, the marks are combined to a uniqueness type for identifiers. These types are then propagated according to the type system of Clean. In our approach, we do not have a separate procedure for marking, but generate an aggregation constraint of which the interpretation provides similar functionality. Our approach does not take evaluation order in account, as this is largely implicit in Haskell. Our marking-like approach is defined in a more conventional notation of a typed abstract syntax tree of the program (Chapter 7) instead of term graphs.

An improvement is that we consider the components of a value independent of the spine of the value. In Clean, if a component of a value is unique, then the spine of the value must necessarily be unique as well. For example, the following expression is not correctly typed in Clean:

let $xs = map\ (+length\ xs)\ [1, 2, 3, 4]$
in $xs :: List^*\ Int^1$

We add the length of a list to each element of the list. The elements of the list are not touched in order to compute the length (otherwise evaluation of this expression does not terminate), so this expression touches the elements of the list only once. So, the elements of the list are in fact unique, but Clean does not allow it because the list may not be shared

[1] In this thesis, we write all code in Haskell-like notation.

if the elements are unique due to a limitation of the type system of Clean. Our approach does not have this limitation and accepts the program, although we pay a price of it in terms of complexity (Chapter 6).

In fact, the above restriction is rather unfortunate in the presence of partial application, as the result of passing a unique parameter must be unique as well. The order in which parameters are passed to a function matters in the presence of the above restriction, because unique parameters are best passed as late as possible. At the moment a unique parameter is passed to a function, the function itself becomes unique. This is not a severe restriction in Clean because parameters to a function can be passed all at once (although currying is supported), such that the compiler can shuffle the order in which the parameters are passed. This is not the case in Haskell, where we curry one argument at a time. This makes a difference when we abstract from certain recurring patterns. For example, consider the expression:

let $f\ x\ y\ g = g\ x\ y$
    $h = f\ 3\ 4$
in $h\ (+) + h\ const$

Our inferencer discovers that 4 is a unique value and 3 is a shared value. The Clean compiler infers that both 3 and 4 are (coerced to) shared.

Besides for uniqueness, our approach can be used to infer strictness properties [38] as well. Uniqueness and strictness are complementary problems, and we show that we can infer them both with our approach. However, we will not go into strictness in detail.

The results of this master's thesis project are this master's thesis that explains how uniqueness typing can be integrated in a Haskell-like language, using conventional type theory, and a prototype implementation in the EH compiler [4].

We only focus on typing uniqueness and strictness properties. We did not focus on using the inferred types for optimizations. There are several reasons for this. The first reason is that the EHC code generation facility was under construction during the lifetime of this project. The second reason is that there is already a lot of research done in the field of Haskell on, for example, strictness and inlining. Finally, a reason is that a type system that captures uniqueness and strictness in the front-end of the compiler, already posed many questions that demanded an answer, that we leave optimizations as a future work.

1.3 Organisation of this thesis

This thesis is organised as follows. We start with our choice for EHC as starting point for our prototype, followed by several chapters that gradually explain our uniqueness-type system. We introduce language features such as recursion and algebraic data types chapter by chapter. The chosen order allows us to discuss features in isolation, and to let earlier chapters pave the way for subsequent chapters. The following table lists the chapters and their main contents:

Chapter	Contents
Chapter 3	Constraint-based uniqueness type checking on a simply typed lambda calculus. We show how types are annotated, and how constraints are generated between annotations on the types. We show how to interpret constraints to verify that types are correctly annotated.
Chapter 4	Polyvariant uniqueness-type inference on generated constraints, and the support for a monomorphic and non-recursive **let**. Constraints are translated to a graph representation. Heeren [14] provided some inspiration on this topic, although our graph representation is unrelated to type graphs.
Chapter 5	We extend the **let** in this chapter with support for recursion. Recursion influences the constraint-generation process. By means of an instantiation constraint, we support different approaches to instantiate constraint graphs.
Chapter 6	Support for a full Haskell-like **let**, by supporting polymorphism. We show that instantiation of a polymorphic type to a type with more structure, gives complications and we give several approaches to deal with this problem.
Chapter 7	This chapter is yet another step towards the support for data types. In this chapter, we show how to deal with an **if** ... **then** ... **else** expression, which presents an opportunity to write about parallel execution paths. It also allows us to deal with this particular aspect of **case** expressions (Chapter 8) in isolation.
Chapter 8	In this chapter, we add algebraic data types to the language. We do not need many new concepts in order to support algebraic data types, but some complications arise because we need some information about type constructors. A thorough understanding of Chapter 3 and Chapter 4 is required.
Chapter 9	We show how we can use the analysis on the type level (on kinds) for analyses on algebraic data types. As an example, we show how we can use it to determine polarity (variance) for data types of Haskell.
Chapter 10	This chapter covers overloading. Overloading presents some difficulties for the uniqueness-type system. We present several ways of how to deal with overloading and show what the consequences are.
Chapter 11	The remainder of this thesis deals with some practical issues, such as separate compilation what is covered in this chapter. It turns out that it is not difficult for the type system to deal with separate complication, but that we run into problems with code generation.
Chapter 12	There comes a time when the programmer is not satisfied with the results of the uniqueness typing, or wants to make sure that some result is enforced. Our type system is located in the front-end of the compiler, allowing interaction between the type system and the programmer. We discuss some mechanisms that the programmer can use to influence the type system, or to inspect the derived uniqueness types of a program.

Of these chapters, Chapter 3 and Chapter 4 are required to understand the other chapters, and some understanding of these chapters is required to understand the others. The other chapters are incremental in the sense that they build on the results of previous chapters, but less heavily.

Finally, as an aside, we want to disambiguate our uses of the word Haskell. With Haskell, we mean Haskell 98 as defined by Peyton Jones et al.[27]. If required, we assume the presence of some commonly used extensions such as multi-parameter type classes [7], functional dependencies [18], and higher-ranked types [31].

Chapter 2

Prerequisites

We use the Essential Haskell Compiler [4] as a starting point in order to prototype the uniqueness type system. In this chapter we explain why we chose this compiler and to bring you up-to-speed with several language features of the EH language. Design decisions of the compiler and the language features to support, influenced the implementation of the prototype. Although not essential, some background knowledge helps to understand some decisions taken further on in this thesis. The eager reader can skip this chapter and delve directly into uniqueness typing (Chapter 3).

2.1 Why EH?

There are a lot of (partial) compilers for Haskell. Which compiler to use for prototyping uniqueness typing? Choosing the right compiler is important, since uniqueness typing crosscuts almost the entire implementation of type system in a a compiler. For prototyping, one would want a compiler that does not have too much features, but enough to keep it interesting.

Several compilers where considered for prototyping. GHC is an example of a complete, but extensive compiler. A full implementation of uniqueness typing in such a compiler is interesting, since people can directly use it in practice. But, having to deal with all the intimacies of a large compiler is not suitable for prototyping. A lightweight compiler is preferable to stay focused. A compiler such as "Typing Haskell in Haskell" [17] is the other extreme. It lacks infrastructure. For example, adding additional environments is a tedious task, since it requires explicitly threading of the environments as additional parameter and result through all the functions. The compiler to use for experimentation has to lie somewhere in between these extremes.

Now EH enters the picture. EH is a family of languages that resemble Haskell. Of these languages, there are currently nine selected as main languages. These main languages are totally ordered into versions, meaning that each subsequent language extends the other. Starting with a simply typed lambda-calculus, each language adds another language feature until we end up with a language that has a feature set that is roughly comparable to the feature set of Haskell. Each language has an associated compiler. The compilers are structured to inherit implementation from a previous version. The intention is that spreading out language features over multiple versions, allows an implementation and explanation in relative isolation.

The EH project offers what we need for prototyping:

- It allows the implementor to pick a language that has enough features, but omits features that are not required. For example, we do not deal with code generation (EH 8), nor want to touch the intimacies of overloading in EH (EH 9). On the other hand, we want polymorphic types (EH 4) and algebraic data types (EH 5). We also need kind information (EH 7) (Chapter 9). So, our base language is EH 7. We thus have the features we need, but not too many features.

- Infrastructure, including type inference, parser, pretty-printing and facilities for debugging.
- Compatibility with the UUAG system [5]. The UUAG system allows us to encode the uniqueness-type system, without having to touch the existing code much.
- The ruler system that pretty printed the type rules in this master's thesis and a part of the implementation.

2.2 Language features of EH

In this section, we briefly consider some language features of EH that are not in Haskell, but are part of EH. Such features pop up in the implementation now and then. This section touches these features and gives some references for further reading.

2.2.1 Quantifiers and qualifiers everywhere

Haskell makes a distinction between types and type schemes. A type scheme is never shown to the programmer since there is no explicit quantification. Type variables in a type are (universally) quantified or qualified (overloading/type classes) in a type scheme. A type does not contain quantifiers nor qualifiers. In EH this is not the case. Quantifiers and qualifiers may occur everywhere in the type. Allowing quantifiers and qualifiers everywhere in the type has consequences.

The consequences for allowing quantifiers everywhere can best be illustrated an example. The following types for the functions f and g are not equal:

$$f :: \forall\ \alpha.(\alpha \rightarrow Int)$$
$$g :: (\forall\ \alpha.\alpha) \rightarrow Int$$

When calling f, the caller chooses a type to substitute for α. For example, writing $f\ 3$ (α instantiated to *Int*) or $f\ True$ (α instantiated to *Bool*) is allowed. Unifications of types in the body of f may not assume that α is instantiated to some type:

$$\begin{array}{l}\textbf{let}\ f :: \forall\ \alpha.(\alpha \rightarrow Int)\\ \quad f = \lambda x \rightarrow x + 3 \quad \text{-- error}\\ \textbf{in}\ f\ 3 + f\ True \qquad \text{-- OK}\end{array}$$

$$\begin{array}{l}\textbf{let}\ f :: \forall\ \alpha.(\alpha \rightarrow Int)\\ \quad f = \lambda\_ \rightarrow 3 \quad \text{-- OK}\\ \textbf{in}\ f\ 3 + f\ True \quad \text{-- OK}\end{array}$$

For g the situation is reversed. The body of g is allowed to choose the type of α, and the caller is not. In this case, the caller can only pass $\perp$, since that is the only value with type $\forall\ \alpha.\alpha$:

$$\begin{array}{l}\textbf{let}\ g :: (\forall\ \alpha.\alpha \rightarrow \alpha) \rightarrow (Int, Bool)\\ \quad g = \lambda i \rightarrow (i\ 3, i\ True) \quad \text{-- OK}\\ \textbf{in}\ g\ (+1) \quad \text{-- error}\end{array}$$

$$\begin{array}{l}\textbf{let}\ g :: (\forall\ \alpha.\alpha \rightarrow \alpha) \rightarrow (Int, Bool)\\ \quad g = \lambda i \rightarrow (i\ 3, i\ True) \quad \text{-- OK}\\ \textbf{in}\ g\ id \qquad\qquad\qquad\quad \text{-- OK}\end{array}$$

More programs can be typed when quantifiers are allowed everywhere. Programs using the *ST* monad [22] require this feature as the *runST* function has a type with a universal quantifier in a contra-variant position: $runST::\forall\ a.(\forall\ s.ST\ s\ a) \rightarrow a$. Peyton Jones et al. [32], give several examples why such *higher-ranked* types are useful, and what the consequences are for type inference [21, 20]. Further discussion about this topic is beyond the scope of this master's thesis.

Besides quantifiers everywhere, EH also allows qualifiers everywhere. Similar to quantifiers everywhere, the location of the qualifier determines who (caller or callee) supplies the evidence. The evidence is the dictionary resulting from the instance declaration that supplies the implementation of the overloaded function corresponding to the qualifier [24]. For example:

$f :: Eq\ \alpha \Rightarrow \alpha \rightarrow \alpha \rightarrow Bool$
$g :: (\forall\ \alpha.Eq\ \alpha \Rightarrow [\alpha] \rightarrow Bool) \rightarrow Bool$

The caller of f supplies the evidence, and the body of g passes the evidence to the first argument of g. Again, we only mention this feature of EH here. A more thorough discussion is beyond the scope of this master's thesis.

2.2.2 Polymorphic kinds

A type system for Haskell uses *kind inference* to check type signatures. Haskell has monomorphic kinds. At the end of kind inference, all remaining kind variables are defaulted to $*$.

data $App\ f\ a = A\ (f\ a)$ -- given by the programmer
$App :: (* \rightarrow *) \rightarrow * \rightarrow *$ -- inferred by the compiler

On the other hand, EH has polymorphic kinds. After kind inference for some binding group, the remaining kind variables are generalized and universally quantified:

$App :: \forall\ \kappa.(\kappa \rightarrow *) \rightarrow \kappa \rightarrow *$

The difference is that the κ can not only be instantiated to $*$, but also to a more complex kind. The following set of types are not allowed to occur simultaneously in a Haskell program, but can in EH[1]:

type $List\ a = App_1\ [\ ]\ a$
data $Fix\ f = In\ (f\ (Fix\ f))$
type $InfNested = App_2\ Fix\ List$

In this example, the kind of App_1 is instantiated to $(*- > *) \rightarrow * \rightarrow *$ and App_2 to: $((* \rightarrow *) \rightarrow *) \rightarrow (* \rightarrow *) \rightarrow *$. In Haskell, you can either get the first, or the second (with a trick), but not both at the same time.

Polymorphic kinds complicate analysis on data types, similar to complications of polymorphism on our uniqueness analysis (Chapter 6).

2.2.3 Extensible records

EH supports lightweight extensible records as discussed by Jones et al. [19]. A simple way to look at records is as tuples with named components. There are operations to add a component to a record (record extension) and to select a component from it (record extraction):

[1]The subscripts below a *App* type constructor are not part of the code, but we use it to unambiguously refer to an occurrence of *App*.

```
let r = {{() | x = 3} | y = 4}   -- create the record x=3, y=4
    r' = {r | y := 3}            -- record update
in  r.x + r'.y                   -- select components from the record
```

Replace the bars with commas and the comparison with tuples is striking. On the type level, records look like tuples with named components:

$$r :: \{\{() \mid x :: Int\} \mid y :: Int\}$$

Consider a function f that adds a field x with value 3 of type *Int* to a record:

$$f = \lambda r \rightarrow \{r \mid x = 3\}$$

What is the type of r in f? If r has a polymorphic type, then we can pass any record as long as it does not have a field x. The type system encodes the fact that the record is polymorphic by assigning a special, universally quantified, type variable to r, and encodes the fact that the record may not already have a field x by qualifying the type variable with a *Lacks*-predicate:

$$f :: Lacks\ x\ v_1 \Rightarrow v_1 \rightarrow \{v_1 \mid x :: Int\}$$

Calling f requires evidence that argument record does not have a field x. The compiler checks this condition, and inserts as evidence an offset within the record to store the value of x.

2.3 Experience

Using EH for prototyping turned out to be an advantage over other compilers for several reasons. One reason is that getting to know the compiler and tools turned out to be relatively easy. Examining the first EH language, and gradually going up to languages with more features—studying the differences—turned out to work in practice. It also helps to discover dependencies between features, which is important to discover how isolated each component is, and thus how easily you can make modifications in a certain area.

A big advantage is compatibility with the UUAG system. The UUAG system allows us to hook into the compiler with virtually no need to touch the original code. Alternatives and attributes are chosen in such a way that new alternatives are most of the time correctly dealt with by code from copy rules, reducing the amount of required code severely. So, due to the UUAG system, the code is extensible and short, which is an asset for (rapid) prototyping.

Unfortunately, EH is not the perfect compiler (yet). EH distinguishes several versions, but individual features cannot be selected. For example, higher-ranked types and existentials are introduced before data types. To support algebraic data types, higher-ranked types and existentials have to be supported as well. These features are not essential for the uniqueness typing story, and are thus needless complications.

Another problem is that the compiler is still in heavy development. This is not a theoretical problem, but sometimes a nuisance in practice. The structure of the compiler is not yet mature, and interfaces change every now and then. This does not make EH a stable framework to build upon. Some functionality that exists today, may be gone tomorrow.

The overall experience, however, is that the EH compiler architecture is well-suited for prototyping and becomes more suitable with each drop of effort that is invested into the EH project.

Chapter 3

Lambda-calculus without bindings

In the introduction (Chapter 1) we showed that functional programs perform a lot of copying in order to preserve referential transparency in the presence of destructive updates. The work in this chapter makes it possible to get performance similar to that of destructive updates in imperative programs. Two steps are needed to get this done: the first step is to analyse which values are guaranteed to be used at most once, and the second step is to insert special code for destructive updates on values that are used at most once. We devote this entire thesis to the first step, and leave the second step as future work.

This chapter presents a uniqueness type system for a severely restricted language: simply-typed lambda-calculus with no **let** bindings, but with integers and an addition operator. Although restricted, this languages serves well to discuss the fundamentals of uniqueness typing, and our implementation of the uniqueness type system. In subsequent chapters, we extend the language with features, until most of Haskell is covered.

This chapter is organized as follows. We start with an overview of the system and some small examples to give a general feeling about our approach. Then we provide a rather large example that demonstrates the approach in detail. This example touches several aspects of our approach, such as the definition of the language, constraint gathering, and constraint solving. In the remainder of the chapter, we treat these aspects in a more formal fashion.

3.1 Intuition

We start with an overview of the uniqueness type system. The system consists of two parts. The first part deals with constraint generation and the second part deals with constraint solving. This separation is not relevant at this point, but is important for separating concerns later in this thesis. The application of an implementation of this type system to an expression, we call a uniqueness analysis.

As we mentioned in the introduction, there are boundaries to what is decidable and what is not. To keep the analysis feasible, we associate uniqueness properties with the type of a value and not with the value itself. The analysis is conservative: if a type is unique, we know that it is used exactly once, but if it is not, it can be unique but is probably not.

3.1.1 Annotations

We consider for a moment what a usage property actually is. During the execution of a program, there are *values* in memory and there are *names* bound to values. These names are actually present in memory as *pointers* to a value[1].

[1] Unless the value is unboxed, then the name only exists conceptually.

Consider the program[2]:

$$(\lambda x \rightarrow (\lambda z \rightarrow z_3)\ x_1 + (\lambda z \rightarrow z_4)\ x_2)\ 3$$

The names for values are the identifiers and their occurrences: x, x_1, x_2, z, z_3 and z_4. Since x_1 and x_2 are occurrences of x, they represent the same value, say v (which happens to be 3). The value v is exactly used once *via* the name x_1. The value v is exactly used once via the name x_2. So, in total v is used more than once via x. Thus, x represents a value that is used more than once. Because the name x_1 represents the same value, it also represents a value that is used more than once, even though this value is used only once via x_1. Similarly, x_2 represents a value that is used more than once. These are two different properties for a name: a property that describes the usage of a value via a name, and a property that describes the entire usage of a value represented by a name.

This concept of names can be generalized to expressions (via referential transparency: an unevaluated expression is a *thunk*, of which the memory is later replaced by the actual value). Suppose that v is a value represented by an expression E. An important observation is that there is a difference between a usage property that specifies that v is used a number of times via E, and the property that specifies how often v is used in total. An analysis typically focuses on one of these two usage properties.

For uniqueness typing as defined in Clean, only usage properties of values are important. In Clean, a unique type represents a value that is used at most once. For each name x with a unique type, we know that the corresponding value is used at most once. For example, the value represented by x is used at most once in $(\lambda x \rightarrow const\ x_5\ x_6)\ 3$, but this property does not specify that it is used at exactly once via x_5 and never through x_6. An optimizer can only assume that if the value is used once via the names, that it will not be used via any of the other names. This is sufficient for destructive updates optimisations. Code that performs destructive updates can be generated for any operation on a value that is used at most once, because at most one of these operations will be performed during a program execution.

However, there are optimisations that benefit from knowing that a value is used exactly once. A value that is used exactly once, can be safely pre-evaluated before passing it to a function. Or a function that is used exactly once, can be inlined safely, always resulting into better performing code and a smaller code size. Furthermore, an expression that will not be used at all, can be omitted. These are realistic and beneficial optimisations, thus for our analysis, we support both types of usage properties.

We define the *cardinality* property of a type as follows:

$$\jmath \bowtie \jmath$$

The left component defines the usage of some individual name or expression. The second component defines the total usage of some value represented by the name or expression. The usage $\jmath$ is defined as:

$$\begin{array}{lll}
\jmath ::= & 0 & \text{-- not used: zero} \\
\mid & 1^{-} & \text{-- used at most once: affine, unique} \\
\mid & 1 & \text{-- exactly used once: linear} \\
\mid & 1^{+} & \text{-- used at least once: strict} \\
\mid & * & \text{-- arbitrary usage: shared}
\end{array}$$

A possible future generalisation of this approach is to support $\jmath$-values up to $k > 0$, such that we can specify that some value is exactly used 3 times, or at most 10 times. Although this generalisation is fairly straight forward, it is also doubtful if there is a use for determining that some value is used at most a certain number of times. Since it only complicates our explanation, we stick to the above definition.

Some optimizations demand that a value is exactly used once, while for other optimizations it is sufficient that a value is used at most once, which is slightly less demanding. Our approach can verify both uniqueness properties and strictness

[2]The subscripts on x are not part of the code, but used as a textual disambiguation of the occurrences of x.

properties. Although our focus is on uniqueness properties, we consider strictness as well, since we get it almost for free with our approach. We still call our approach uniqueness typing, but it is technically more than that.

A cardinality property is attached to type constructors as *cardinality annotation*. Cardinality originates from set theory where it is defined as the number of elements in a set. Some authors use the word cardinality as alternative for arity. Our definition of cardinality is unrelated to arity, which is the reason why we stress the difference here.

Suppose that x is a name for value v of type *Int*, i.e. $x :: Int$ with $memory\ (x) = v$. To specify that x uses v exactly once via x, and that x requires that v is used exactly once in total, we write: $x :: Int^{1 \bowtie 1}$. To specify that x does not use v via x, and that x requires v to be used exactly once in total, we write: $x :: Int^{0 \bowtie 1}$. Similarly, suppose that f is the name for a closure cl, i.e. a function $f :: Int \to Int$. Then $f :: Int^{1 \bowtie *} \to^{1 \bowtie 1} Int^{1 \bowtie *}$ specifies that f uses cl exactly once via f, and that f requires that cl is used exactly once in total. Furthermore, the annotated type specifies that for an argument-value a, f provides that it uses a exactly once, and makes no demands about the total usage of a. Finally, the annotated type specifies that for a result-value r, f requires that r is used exactly once, and gives no guarantees about the total usage of r.

The analysis expects a program annotated in this fashion and verifies that the annotations are correct (inference in Chapter 4). Unfortunately, we cannot easily formulate the conditions of when an annotation is correct or not. We have to check two things: does the annotation fit with related annotations (the context), and does the annotation fit with the actual uses of the value. With context we mean that if two types are unified, that their annotations also unify. For example, a value with cardinality $1 \bowtie 1$ cannot be passed to a function that has $1^- \bowtie 1$ cardinality for its parameter, because the function does not guarantee that the value will always be used. With the actual uses of the value, we mean the guaranteed upper and lower bound on the number of times the value will be evaluated or referenced. For example, a value that is referenced at least once cannot have the cardinality annotation $0 \bowtie 0$. We have sufficient information for the context check (which is basically a form of unification, and checks the right-hand side of the cardinality annotation), but insufficient information for the check on the usages (the left-hand side of the cardinality annotation).

We therefore ask the programmer to specify an upper and lower bound on the type of an expression (or name). The types in an expression are thus annotated with three (!) annotations: A cardinality annotation, a lower-bound usage-count annotation, and an upper-bound usage-count annotation. The lower-bound usage-count annotation tells us that a value is at least used a certain number of times via the expression, and the upper-bound usage-count annotation tells us that a value is at most used a certain number of times via the expression. The usage-check demands that left-hand side of the cardinality annotation *fits* within these bounds (Figure 3.6). The upper and lower bound can be verified by means of an analysis, which in turn means that we can verify the cardinality annotations.

We explain by means of an example how these three annotations interact and how we formulate the checks as a constraint satisfaction problem. Unfortunately, that is rather hard to grasp at this point, so we defer the explanation to Section 3.2. Instead, we discuss how to check the upper-bound usage-count annotations here, since the checks on the lower-bound usage-count and also the context-check on the usage annotations themselves are quite similar. This will become clear in Section 3.2.

3.1.2 Checking the upper-bound usage-count annotation

An upper-bound usage-count annotation u specifies that the corresponding values are used at most u times:

$$
\begin{array}{rcl}
\iota & ::= & 0 \quad \text{-- Not used.} \\
& | & 1 \quad \text{-- Used at most once.} \\
& | & * \quad \text{-- Arbitrary usage. However, typically represents a number greater than one.}
\end{array}
$$

We assume that expressions are annotated with upper-bound usage-count annotations. That means that all type constructors occurring in the types of an expression have an upper-bound usage-count annotation. For example, an annotated version of the expression $(\lambda x \to x)\ 3$ is[3]:

[3] Again, note that the subscripts are not part of the code. They are used in the text to refer to a particular type.

$(\lambda x \to x)\ 3$ -- original expression
$(((\lambda(x :: Int) \to x :: Int) :: Int \to Int)\ (3 :: Int)) :: Int$ -- typed expression
$(((\lambda(x :: Int_a{}^1) \to x :: Int_b{}^1) :: Int_c{}^1 \to_d{}^1 Int_e{}^1)\ (3 :: Int_f{}^1)) :: Int_g{}^1$ -- typed expression with annotations

This looks quite intimidating, but let us consider what is displayed here. The outermost expression is annotated with $Int_g{}^1$, which means that the result of the expression is used at most once. The outermost expression is an application of an argument to a function. The annotated type of the function is $Int_c{}^1 \to_d{}^1 Int_e{}^1$. The annotation on the arrow type constructor, $(\to_d)^1$, specifies that the function is used at most once. The annotation $Int_c{}^1$ specifies that the parameter is used at most once, and $Int_e{}^1$ demands that the result is used at most once. The annotation $Int_f{}^1$ specifies that the argument is used at most once. The annotation on the parameter of the function and the annotation on the argument of the function are related to each other. Likewise, the annotation on the result of the function is related to the annotation on the function application. The annotation of the parameter of the function is related to the annotation of the result of the function, but that depends on the structure of the body of the function. Take a moment to get a clear picture of the annotations and their meaning.

Are these annotations correct? For this example, it is not difficult to see that indeed the value 3 is not used more than once, but we need a systematic approach to verify this. We check the annotations by a traversal over the structure of the expression, and approximate the upper bound from below. By computing an approximation of a least upper bound, we check that the upper bound provided by the programmer is high enough to be an upper bound for any execution of the program. In other words, we determine from the structure of the program that the upper bound should be at least m to be an upper bound, and verify that the upper bound given by the programmer is at least m. The upper bound is allowed to be higher than is strictly necessary (safe approximation).

The root of the expression is the starting point. When executing the expression, the root is evaluated once to obtain the value. So, the upper bound for the root of the expression is at least 1. The annotated type of the root, Int^1, is therefore correct. The root of the expression is a function application. Since the annotation is correct, it means that the upper bound for the function application is at least the upper bound of the root of the expression, thus at least 1. The function is used at least as often as the function application, so the upper bound of the function is at least the upper bound of the function application, thus at least 1. The annotation on $\to_d{}^1$ is correct. The same reasoning applies to the result of the function. The upper bound of the result of the function is at least the upper bound of the function application. The annotation on $Int_e{}^1$ is correct.

Now the function and the argument. Consider the argument first. The function fully determines how often it is used. The upper bound of the argument is at least the upper bound of the parameter of the function. The annotation on $Int_c{}^1$ is the upper bound of the parameter, thus the upper bound of the argument is at least 1. $Int_f{}^1$ is correct. Of course, we have to verify that the upper bound of the parameter of the function is 1. We take a look at the function. The body of the function has an upper bound of 1, because the result of the function has an upper bound of 1. The body of the function is a use of parameter x. So, the upper bound for this occurrence of x is 1. The annotation on $Int_b{}^1$ is correct. The upper bound for the parameter x is at least the sum of the upper bounds of the occurrences. There are no other occurrences of x, so the aggregated upper bound of x is 1. We come back to this specific subject later. The annotation on $Int_a{}^1$ is correct. Again, take a moment to get a mental picture of how the abstract syntax tree is traversed and how annotations are compared from outermost expressions to innermost expressions and from left to right.

An important step is the aggregation of upper bounds for occurrences of an identifier. The following variation of the above example shows why it is important:

$(\lambda x \to x + x)\ 3$ -- original expression
$(((\lambda(x :: Int) \to ((x :: Int) + (x :: Int)) :: Int) :: Int \to Int)\ (3 :: Int)) :: Int$ -- typed expression
$(\lambda(x_0 :: Int_a{}^*) \to (x_1 :: Int_b{}^1) + (x_2 :: Int_c{}^1))\ (3 :: Int_d{}^*)$ -- typed expression with annotations

The number of times a value bound to x_0 is used, is the aggregated number of times the value is used as x_1 and x_2. Any actual usage count for x_1 or x_2 is lower than their corresponding upper bound. A safe approximation for a least upper bound of x_0 is the sum of the upper bounds of x_1 and x_2. As the example shows, the upper bounds of x_1 and x_2 are both 1, so the sum is $*$ and the annotation is correct.

As we mentioned several times already, a conservative approximation of the upper bound is allowed. The upper bound may be higher than is strictly necessary. For example, if we know that a value is not used, we may also assume that it is at most used once. Furthermore, assuming that some value is used an arbitrary number of times is always allowed. It is safe to pretend that the upper bound is higher than is actually the case, but it makes the analysis result less accurate.

The following example is a variation upon the earlier example where some of the annotations are higher than is strictly necessary:

$(\lambda x \to x)\ 3$ -- original expression
$(((\lambda(x :: Int) \to x :: Int) :: Int \to Int)\ (3 :: Int)) :: Int$ -- typed expression
$(((\lambda(x :: Int_a{}^*) \to x :: Int_b{}^1) :: Int_c{}^* \to_d{}^* Int_e{}^1)\ (3 :: Int_f{}^*)) :: Int_g{}^1$

We verify again that these annotations are correct. The outermost expression is used at most once, so the upper bound is at least 1. The annotation on $Int_g{}^1$ is correct. The upper bound for the function itself is at least as high as the upper bound for the function application. The upper bound for the function is $*$, which is at least as high as the upper bound of 1 of the function application. The annotation on $\to_d{}^*$ is correct. The result of the function has an upper bound of 1, which is at least as high as the upper bound of 1 of the function application. The annotation on $(Int_e)^1$ is correct. The aggregated upper bound for x is at least the sum of the upper bounds of the occurrences of x, which in this case is $Int_b{}^1$. The annotation on $Int_a{}^*$ is higher, but that is allowed, so it is correct. Consequently, our traversal of the expression demands that the annotations on $Int_c{}^*$ and $Int_f{}^*$ are at least the annotation on $Int_a{}^*$, which is correct.

To summarize: what our analysis does is to start with some known information about the root of the expression and 'push' it through the abstract syntax tree from outermost to innermost, and in case of functions from result to arguments based on the body. A mental picture is that there is an 'flow' of count-information from the root of the expression to the leaves. We say that the analysis 'propagates' counts. We see in later sections that dealing with a lower bound, and dealing with cardinality annotations, is not much different than what we did in this section.

3.2 The next step

With the intuitions gained in the previous section, we introduce a constraint-based approach to uniqueness typing. For example, for the upper bound analysis of the previous section, we generate a constraint $... \triangleright ...$ for each time we wrote 'at least', and a constraint $... \odot ... \odot ... \leqslant ...$ where we aggregated upper bounds. It that it? Yes, that is it basically, because we will show that the same constraints can be used for the lower bound analysis and the analysis for the actual cardinality values.

The running example in this section is the following expression:

$(\lambda x\ y \to (y + x) + x)\ 3\ 4$

The result of this program is evaluated once when the program is executed. Consequently, the results of both additions are used once, and thus is the x parameter used twice and the y parameter once. Which in turn means that the value 3 is used twice and the value 4 once. In terms of usage, the value 3 is linear (exactly once), and the value 4 is strict (more than once).

Efficient code can be produced with this usage information. A conventional implementation of addition in a pure, lazy language, evaluates both arguments, allocate memory for an integer, and store the result of the addition there. But in this case, there is no need to allocate memory for the result of the additions. Since the value passed to y is linear, the value of x can be added directly to the value of y (destructively updating the value of y). The result of the addition is again linear, and the value of x can be added directly again for the second addition. The result: less memory allocation, less copying, and better cache utilization. If the compiler also inlines functions that are at most used once, and evaluates strict values before passing it to a function, then we believe that this code is as efficient as the imperative program:

int y;
$y = 4$;
$y += 3$;
$y += 3$;
return y;

3.2.1 Annotations

Usage information is encoded as a cardinality annotation on the type constructors in the type of a value. The type constructors specify how a value is represented in memory. For example, the value 3 is represented as a sequence of 32 bits somewhere in memory. The type constructor *Int* of the type of the value 3, corresponds with this area of memory. The cardinality annotation determines the cardinality properties of the area of memory. Likewise, the function type constructor ($\rightarrow$) corresponds to a closure, which is an area of memory which contains a pointer to the code of the function, and pointers to values where the function depends on, among other information [40]. The exact contents of the closure does not matter for our explanation. Just assume that it represents the bookkeeping required for applying the function to some value. The cardinality annotation on the function type constructor specifies the cardinality property of this closure. To know the cardinality of a memory area (part of a value), lookup the annotation on the type constructor that corresponds to the area of memory (Section 3.4).

As mentioned in the introduction, a type has three annotations. This looks horrible, but fortunately, we can do something against it the next chapter. As an example of a triple of annotations, consider the type $Int^{(1,1^{+}\bowtie 1^{+},*)}$. The first annotation is the lower-bound usage-count annotation, which specifies that the value is used at least once. The second annotation is the cardinality annotation, specifying that the value is strict, meaning that it is used one or more times. The third annotation is the upper-bound usage-count annotation, which specifies that the value is at most used an arbitrary number of times. This triple is consistent if the cardinality annotation fits the two bounds (Figure 3.6) and the left-hand side of the cardinality annotation does not conflict with the right-hand side of the cardinality annotation (Section 3.4).

For the above example, a possible annotation is:

$$
\begin{array}{l}
(((\lambda(x :: Int^{(1,1\bowtie 1^{+},*)}) \rightarrow \\
\quad \lambda(y :: Int^{(1,1\bowtie 1,*)}) \rightarrow \\
\quad\quad (\ \ (\ \ y :: Int^{(1,1\bowtie 1,1)} \\
\quad\quad\quad\ + x :: Int^{(1,1\bowtie 1^{+},*)} \\
\quad\quad\quad) :: Int^{(1,1\bowtie 1,1)}) \\
\quad\quad\ + x :: Int^{(1,1\bowtie 1^{+},*)} \\
\quad\quad) :: Int^{(1,1\bowtie 1,1)} \\
\quad) :: Int^{(1,1\bowtie 1^{+},*)} \rightarrow^{(1,1\bowtie 1,1)} Int^{(1,1\bowtie 1,1)} \rightarrow^{(1,1\bowtie 1,1)} Int^{(1,1\bowtie 1,1)}) \\
\quad (3 :: Int^{(1,1\bowtie 1^{+},*)}) \\
\quad) :: Int^{(1,1\bowtie 1,1)} \rightarrow^{(1,1\bowtie 1,1)} Int^{(1,1\bowtie 1,1)}) \\
\quad (4 :: Int^{(1,1\bowtie 1,1)}) \\
) :: Int^{(1,1\bowtie 1,1)}
\end{array}
$$

This looks quite intimidating, but that is just because there are a lot of types involved and the annotations take much space. In the next chapter, we infer all annotations, and the lower and upper bound annotations become implicit, resulting in cleaner types (Section 4.3). But we first have to do the dirty work here.

3.2.2 Checking the annotations

How to check that the annotated typing of a program is valid? First we check that the program is correctly typed for the conventional type system of simply typed lambda-calculus. Secondly, we check that all triples of annotations are

consistent. Then it is time for the real work: gather a set of constraints from the annotated program, and check if the constraints are satisfied. When the constraints are satisfied, the annotations are correct and the program is correctly typed according to the uniqueness type system.

As an aside, in order to keep the explanation compact, we are a bit sloppy in distinguishing identifiers, values and types. For example, if we talk about the upper bound of a function, we intend the upper-bound usage-count annotation on the type constructor associated with the value representing the function. Or when we talk about the cardinality of an expression, we actually mean how often the value corresponding to the expression is used. So, assume that the notion of obtaining a cardinality annotation, upper bound, and lower bound is overloaded for identifiers, expressions, values, types and type constructors in the obvious way.

We illustrate the constraints by analyzing the example[4]. For a description of the constraints and their meaning, check Section 3.4. We reason about a program in an outermost-to-innermost way. We start with the entire program (expression):

$$(...) :: Int^{(1^1, 1 \bowtie^2 1, 1^3)}$$

The result of the entire expression is exactly used once. However, we are allowed to be less accurate for the result of the entire expression. Any usage value that is consistent is allowed. We generate the constraint $(1, 1 \bowtie *, 1) \triangleright_s (1^1, 1 \bowtie^2 1, 1^3)$. This constraint specifies that the lower bound is at most 1, the upper bound is at least 1, and the cardinality annotation can be anything as long as it is consistent with the two bounds in the triple. The subscript s indicates that the cardinality annotation is ignored in this constraint. A $\triangleright$ constraint without the s suffix is encountered later in the example. The generated constraint expresses what we know about the result of the entire expression. We call this constraint a flow or propagation constraint, because with some imagination, it 'pushes' values on the left-hand side to the right-hand side. We often call a $\triangleright$ constraint a coercion, which is explained in the next section.

One step into the expression, an application is encountered:

$$((... :: Int^{(1^4, 1 \bowtie^5 1, 1^6)} \rightarrow^{(1^7, 1 \bowtie^8 1, 1^9)} Int^{(1^{10}, 1 \bowtie^{11} 1, 1^{12})}) \; (4 :: Int^{(1^{13}, 1 \bowtie^{14} 1, 1^{15})})) :: Int^{(1^{16}, 1 \bowtie^{17} 1, 1^{18})}$$

Several constraints are generated for a function application:

- The function application and the function. The number of usages of the function depends on the number of usages of the function application. We generate the constraint $(1^{16}, 1 \bowtie^{17} 1, 1^{18}) \triangleright_s (1^7, 1 \bowtie^8 1, 1^9)$. The lower bound of the function (1^7) is at most the lower bound of the function application (1^{16}). The upper bound of the function (1^{18}) is at least the upper bound of the function application (1^9). The constraint expresses precisely how often the function is used, if we know how often the function application itself is used. The cardinality annotations are unrelated, since the memory occupied by the closure (function) is unrelated to the memory occupied by the result of the function.

- The function application and result of the function. The number of uses of the result of the function application depends on how often the function application itself is used. We generate the constraint $(1^{16}, 1 \bowtie^{17} 1, 1^{18}) \triangleright (1^{10}, 1 \bowtie^{11} 1, 1^{12})$. The lower bound of the result of the function (1^{10}) is at most the lower bound of the function application (1^{16}). The upper bound of the result of the function (1^{18}) is at least the upper bound of the function application (1^{12}). The cardinality annotation ($1 \bowtie^{11} 1$) of the result of the function should be equal (modulo coercion/weakening) to the cardinality annotation of the function application ($1 \bowtie^{17} 1$), since they represent the same value (via referential transparency).

- The function and the argument of the function application. The uses of the argument of the function application depend on the annotations of the parameter of the function. We generate the constraint $(1^4, 1 \bowtie^5 1, 1^6) \triangleright (1^{13}, 1 \bowtie^{14} 1, 1^{15})$. The semantics of the constraint capture the relations between the three annotations on the type of the function parameter and the type of the argument value.

[4]For clarity, the annotations values have a number superscripted to show where they appear in the constraints.

The remainder of the constraints are obtained by analysing the function expression and the argument expression. There is no constraint generated for the argument expression, since that is an *Int* expression, and each consistent triple is valid. In other words: the value 3 can represent a linear *Int*, but also a strict *Int* or an affine *Int*. So, verifying the function expression is what remains.

The function expression is again an application:

$$
\begin{array}{l}
(... :: Int^{(1^{19},1^{+}\bowtie^{20}1^{+},*^{21})} \rightarrow^{(1^{22},1\bowtie^{23}1,1^{24})} (Int^{(1^{25},1\bowtie^{26}1,1^{27})} \rightarrow^{(1^{28},1\bowtie^{29}1,1^{30})} Int^{(1^{31},1\bowtie^{32}1,1^{33})}))\\
\quad (3 :: Int^{(1^{34},1^{+}\bowtie^{35}1^{+},*^{36})})\\
\qquad :: Int^{(1^{4},1\bowtie^{5}1,1^{6})} \rightarrow^{(1^{7},1\bowtie^{8}1,1^{9})} Int^{(1^{10},1\bowtie^{11}1,1^{12})}
\end{array}
$$

This is basically the same situation as above, except with one important difference. The type of the result of the function application is a function. Its type has more than one type constructor, and multiple triples of annotations. Each type constructor in one type has a corresponding type constructor in the other type. The $\triangleright$ constraint is generated between triples attached on corresponding type constructors. This way we check annotations that represent the same values. The direction of the $\triangleright$ constraint depends on co-variance and contra-variance. The direction is reversed for $\triangleright$ constraints between *contra-variant* type constructors. The reason is that a co-variant type constructor corresponds to values for which the annotation requires some property ("I must be used exactly once"), whereas contra-variant type constructors correspond to values for which the annotations provide some property ("I am used exactly once"), which is essentially the inverse direction. See Peyton Jones [31] for an elaborate explanation on co-variance, contra-variance, and subtyping.

Applying the same reasoning as with the previous function application, we generate the following constraints:

$$
\begin{array}{lll}
(1^{7}, 1 \bowtie^{8} 1, 1^{9}) & \triangleright_{s} (1^{22}, 1 \bowtie^{23} 1, 1^{24}) & \text{-- use of the function}\\
(1^{10}, 1 \bowtie^{11} 1, 1^{12}) & \triangleright (1^{31}, 1 \bowtie^{32} 1, 1^{33}) & \text{-- use of the function value (resulting Int, co-variant)}\\
(1^{7}, 1 \bowtie^{8} 1, 1^{9}) & \triangleright (1^{28}, 1 \bowtie^{29} 1, 1^{30}) & \text{-- use of the function value (function type, co-variant)}\\
(1^{25}, 1 \bowtie^{26} 1, 1^{27}) & \triangleright (1^{4}, 1 \bowtie^{5} 1, 1^{6}) & \text{-- use of the function value (argument Int, contra-variant)}\\
(1^{19}, 1^{+} \bowtie^{20} 1^{+}, *^{21}) & \triangleright (1^{34}, 1^{+} \bowtie^{35} 1^{+}, *^{36}) & \text{-- use of the argument}
\end{array}
$$

Yet to analyse are the function of the function application, and the argument of the function application. The argument is again an *Int* expression, which does not result in constraints, so we proceed with the function expression.

The function expression is a lambda abstraction:

$$
\begin{array}{l}
(\lambda(x :: Int^{(1^{19},1^{+}\bowtie^{20}1^{+},*^{21})}) (y :: Int^{(1^{25},1\bowtie^{26}1,1^{27})}) \rightarrow ...y :: Int^{(1^{37},1\bowtie^{38}1,1^{39})} ... x_1 :: Int^{(1^{40},1\bowtie^{41}1^{+},*^{42})} ... x_2 :: Int^{(1^{43},1\bowtie^{44}1^{+},*^{45})}\\
) :: Int^{(1^{19},1^{+}\bowtie^{20}1^{+},*^{21})} \rightarrow^{(1^{22},1\bowtie^{23}1,1^{24})} (Int^{(1^{25},1\bowtie^{26}1,1^{27})} \rightarrow^{(1^{28},1\bowtie^{29}1,1^{30})} Int^{(1^{31},1\bowtie^{32}1,1^{33})})
\end{array}
$$

For a lambda abstraction, we perform two tasks: analyze the body of the lambda abstraction, and combine the results of the uses of the values represented by the parameters. The bounds of each occurrence of some parameter are individually established. We generate an aggregation constraint to combine the bounds and to check that the cardinality value is properly split over the occurrences: $triple\ (use\_site_1) \odot ... \odot triple\ (use\_site_n) \leqslant triple\ (def\_site)$. We use the safe approximation that the upper bounds are combined by addition and the lower bounds by taking the minimum. This approach is improved in Chapter 7. This constraint ensures that we check that the lower-bound is low enough and the upper-bound is high enough. Furthermore, the constraint ensures that if the parameter has some cardinality value u, then the combined occurrences correspond to u. For example, if u is linear, it means that one of the occurrences must be linear and all the other occurrences not used.

With only one use-site of y, and two use-sites of x, the constraints are:

$$
\begin{array}{ll}
(1^{40}, 1 \bowtie^{41} 1^{+}, *^{42}) \odot (1^{43}, 1 \bowtie^{44} 1^{+}, *^{45}) \leqslant (1^{19}, 1^{+} \bowtie^{20} 1^{+}, *^{21}) & \text{-- for x}\\
(1^{37}, 1 \bowtie^{38} 1, 1^{39}) \leqslant (1^{25}, 1 \bowtie^{26} 1, 1^{27}) & \text{-- for y}
\end{array}
$$

The function body remains to be checked. The function body consists of the additions:

$$((y :: Int^{(1^{37},1 \bowtie^{38} 1,1^{39})}) + (x_1 :: Int^{(1^{40},1^{+} \bowtie^{41} 1^{+},*^{42})})) :: Int^{(1^{46},1 \bowtie^{47} 1,1^{48})}$$
$$(((...) :: Int^{(1^{46},1 \bowtie^{47} 1,1^{48})}) + (x_2 :: Int^{(1^{43},1^{+} \bowtie^{44} 1^{+},*^{45})})) :: Int^{(1^{31},1 \bowtie^{32} 1,1^{33})}$$

The addition operator uses both the arguments at least as many times as the result is used. The default operational semantics for addition stores the result of the addition in a memory location unrelated to the arguments of the addition. So, cardinality annotations of the arguments of the addition are unrelated to the cardinality annotation on the result of the addition. But the lower-bound and upper-bound annotations of the arguments are related to the lower-bound and upper-bound annotations of the result of the expression. The cardinality annotation on the result of the addition may be any value that fits the two bounds. For the relation between the arguments of the addition and the result of the addition, we generate a weaker version of the $\triangleright$ constraint that ignores the cardinality annotation, but does not ignore the two bounds: $\triangleright_s$, with the *s* of *soft*. When we want to stress the difference between these two $\triangleright$ constraints, we write the normal $\triangleright$ constraint as $\triangleright_h$, with the *h* of *hard*.

The following constraints relate the result of the additions properly to the arguments of the addition:

$(1^{46}, 1 \bowtie^{47} 1, 1^{48}) \triangleright_s (1^{37}, 1 \bowtie^{38} 1, 1^{39})$
$(1^{46}, 1 \bowtie^{47} 1, 1^{48}) \triangleright_s (1^{40}, 1 \bowtie^{41} 1^{+}, *^{42})$
$(1^{31}, 1 \bowtie^{32} 1, 1^{33}) \triangleright_s (1^{46}, 1 \bowtie^{47} 1, 1^{48})$
$(1^{31}, 1 \bowtie^{32} 1, 1^{33}) \triangleright_s (1^{43}, 1 \bowtie^{44} 1^{+}, *^{45})$

3.2.3 Checking the constraints

The final part of this example is to verify that the constraints hold. We show by an example how we interpret the constraints. We define when a constraint holds more precisely in Section 3.7. We assume that the triples are already checked for consistency, meaning that the cardinality annotation fits between the upper and lower bound, and that the cardinality annotation itself is consistent. A cardinality annotation is consistent if the left-hand side does not conflict with the right-hand side. The table in Figure 3.1 lists the constraints and their interpretation (see Figure 3.14 for the definition of *isSplit*).

constraint	lower bound	usage	upper bound
$(1, *, 1) \triangleright_s (1^1, 1 \bowtie^2 1, 1^3)$	$1 \sqsubseteq_L 1^1$		$1 \leqslant 1^3$
$(1^{16}, 1 \bowtie^{17} 1, 1^{18}) \triangleright_s (1^7, 1 \bowtie^8 1, 1^9)$	$1^{16} \sqsubseteq_L 1^7$		$1^{18} \sqsubseteq_U 1^9$
$(1^{16}, 1 \bowtie^{17} 1, 1^{18}) \triangleright (1^{10}, 1 \bowtie^{11} 1, 1^{12})$	$1^{16} \sqsubseteq_L 1^{10}$	$1 \bowtie^{17} 1 \sqsubseteq_C 1 \bowtie^{11} 1$	$1^{18} \sqsubseteq_U 1^{12}$
$(1^4, 1 \bowtie^5 1, 1^6) \triangleright (1^{13}, 1 \bowtie^{14} 1, 1^{15})$	$1^4 \sqsubseteq_L 1^{13}$	$1 \bowtie^5 1 \sqsubseteq_C 1 \bowtie^{14} 1$	$1^6 \sqsubseteq_U 1^{15}$
$(1^7, 1 \bowtie^8 1, 1^9) \triangleright_s (1^{22}, 1 \bowtie^{23} 1, 1^{24})$	$1^7 \sqsubseteq_L 1^{22}$		$1^9 \sqsubseteq_U 1^{24}$
$(1^{10}, 1 \bowtie^{11} 1, 1^{12}) \triangleright (1^{31}, 1 \bowtie^{32} 1, 1^{33})$	$1^{10} \sqsubseteq_L 1^{31}$	$1 \bowtie^{11} 1 \sqsubseteq_C 1 \bowtie^{32} 1$	$1^{12} \sqsubseteq_U 1^{33}$
$(1^7, 1 \bowtie^8 1, 1^9) \triangleright (1^{28}, 1 \bowtie^{29} 1, 1^{30})$	$1^7 \sqsubseteq_L 1^{28}$	$1 \bowtie^8 1 \sqsubseteq_C 1 \bowtie^{29} 1$	$1^9 \sqsubseteq_U 1^{30}$
$(1^{25}, 1 \bowtie^{26} 1, 1^{27}) \triangleright (1^4, 1 \bowtie^5 1, 1^6)$	$1^{25} \sqsubseteq_L 1^4$	$1 \bowtie^{26} 1 \sqsubseteq_C 1 \bowtie^5 1$	$1^{27} \sqsubseteq_U 1^6$
$(1^{19}, 1^{+} \bowtie^{20} 1^{+}, *^{21}) \triangleright (1^{34}, 1^{+} \bowtie^{35} 1^{+}, *^{36})$	$1^{19} \sqsubseteq_L 1^{34}$	$1^{+} \bowtie^{20} 1^{+} \sqsubseteq_C 1^{+} \bowtie^{35} 1^{+}$	$*^{20} \sqsubseteq_U *^{36}$
$(1^{40}, 1 \bowtie^{41} 1^{+}, *^{42}) \odot (1^{43}, 1 \bowtie^{44} 1^{+}, *^{45})$	1^{40} ‘max‘ 1^{43}	$\{1_{41}, 1_{44}\}$ ‘$isSplit$‘ $1^{+}{}_{20}$	$1^{42} + 1^{45}$
$\leqslant (1^{19}, 1^{+} \bowtie^{20} 1^{+}, *^{21})$	$\sqsubseteq_L 1^{19}$	$\wedge\ 1^{+}{}_{20} \equiv 1^{+}{}_{20} \equiv 1^{+}{}_{20})$	$\sqsubseteq_U 1^{21}$
$(1^{37}, 1 \bowtie^{38} 1, 1^{39}) \leqslant (1^{25}, 1 \bowtie^{26} 1, 1^{27})$	$1^{37} \sqsubseteq_L 1^{25}$	$\{1_{38}\}$ ‘$isSplit$‘ $1_{26} \wedge 1_{38} \equiv 1_{26}$	$1^{39} \sqsubseteq_U 1^{27}$
$(1^{46}, 1 \bowtie^{47} 1, 1^{48}) \triangleright_s (1^{37}, 1 \bowtie^{38} 1, 1^{39})$	$1^{46} \sqsubseteq_L 1^{37}$		$1^{48} \sqsubseteq_U 1^{39}$
$(1^{46}, 1 \bowtie^{47} 1, 1^{48}) \triangleright_s (1^{40}, 1 \bowtie^{41} 1^{+}, *^{42})$	$1^{46} \sqsubseteq_C 1^{40}$		$1^{48} \sqsubseteq_U *^{42}$
$(1^{31}, 1 \bowtie^{32} 1, 1^{33}) \triangleright_s (1^{46}, 1 \bowtie^{47} 1, 1^{48})$	$1^{31} \sqsubseteq_L 1^{46}$		$1^{33} \sqsubseteq_U 1^{48}$
$(1^{31}, 1 \bowtie^{32} 1, 1^{33}) \triangleright_s (1^{43}, 1 \bowtie^{44} 1^{+}, *^{45})$	$1^{13} \sqsubseteq_L 1^{43}$		$1^{33} \sqsubseteq_U *^{45}$

Figure 3.1: Constraint interpretation

The relation $\sqsubseteq_L$ should be interpreted as the relation $\geqslant$ on natural numbers. The relation $\sqsubseteq_U$ as $\leqslant$ on natural numbers. The $\sqsubseteq_C$ relation is somewhat complicated. For the left component in the cardinality annotation, the $\sqsubseteq_C$ relation specifies 'is more specific', and for the right component the relation is equality modulo coercion. We cover these relations in more detail in Section 3.4.

All constraints are satisfied, so the program was properly annotated. But this is only one of the annotations for this program that satisfy the above constraints. For example, take 0 for all lower bounds, $* \bowtie *$ for all cardinalities, and $*$ for all upper bounds and verify that the constraints are satisfied. However, set lower bounds to 0, cardinalities to $0 \bowtie 0$, and upper bounds to 0, and verify that some constraints are violated (in fact, the constraint at the root of the expression is already violated).

In the next chapter, we determine a least solution to the constraints, which means a least upper bound, a highest lower bound, and a most precise usage annotation. The remainder of this chapter delves more precisely into subjects encountered in this section.

3.3 Language

As mentioned before, the language in this chapter is a simply typed lambda-calculus without **let** bindings. The latter means no (recursive) **let**, but there are bindings by lambda abstraction:

```
expr ::= int              -- integer expression (Int)
       | var              -- variable expression (Var)
       | expr + expr      -- plus operator (Plus)
       | expr expr        -- application (App)
       | λvar → expr      -- lambda abstraction (Lam)
```

Types are assigned to each subexpression. The language of types is:

```
τ ::= Int      -- tycon int (Int)
    | (→)      -- tycon arrow (Arrow)
    | τ τ      -- type application (App)
```

We assume that all programs are correctly typed according to the type system in Figure 3.2. In this specification of the type system, Γ is the environment that contains a type for each identifier in scope, and τ represents types. An integer expression has type *Int*. The type of a variable is recorded in the environment. The result and operands of an addition are *Int*s. When applying a function to a value, the parameter type and argument type should unify. Finally, a lambda abstraction brings the type of an identifier in scope, which means into the environment Γ of the body.

$$\boxed{\Gamma \vdash^{\textbf{type}} e : \tau}$$

$$\frac{}{\Gamma \vdash^{\textbf{type}} n : Int}\ \text{E.INT}_E \qquad \frac{i \mapsto \tau \in \Gamma}{\Gamma \vdash^{\textbf{type}} i : \tau}\ \text{E.VAR}_E \qquad \frac{\Gamma \vdash^{\textbf{type}} x : Int \quad \Gamma \vdash^{\textbf{type}} y : Int}{\Gamma \vdash^{\textbf{type}} x + y : Int}\ \text{E.PLUS}_E \qquad \frac{\Gamma \vdash^{\textbf{type}} a : \tau_a \quad \Gamma \vdash^{\textbf{type}} f : \tau_f \quad \tau_f \cong \tau_a \to \tau}{\Gamma \vdash^{\textbf{type}} f\ a : \tau}\ \text{E.APP}_E$$

$$\frac{x \mapsto \tau_x, \Gamma \vdash^{\textbf{type}} b : \tau}{\Gamma \vdash^{\textbf{type}} \lambda x \to b : \tau_x \to \tau}\ \text{E.LAM}_E$$

Figure 3.2: Expression type rules (E)

3.4 Types, Annotations and Constraints

For the uniqueness type system, we extend the language of types to include the annotations encountered in the examples. These annotations are attached to type constructors:

$$
\begin{array}{lll}
\widetilde{\tau} ::= Int^{(\ell,c,\iota)}) & & \text{-- tycon int} \\
\quad | \ \rightarrow^{(\ell,c,\iota)} & & \text{-- tycon arrow} \\
\quad | \ \widetilde{\tau}\,\widetilde{\tau} & & \text{-- type application} \\
c ::= \jmath \bowtie \jmath & & \text{-- cardinality annotation} \\
\ell ::= 1 \mid 0 & & \text{-- lower-bound usage-count annotation} \\
\iota ::= 0 \mid 1 \mid * & & \text{-- upper-bound usage-count annotation} \\
\jmath ::= 0 \mid 1^{-} \mid 1 \mid 1^{+} \mid * & & \text{-- cardinality annotation}
\end{array}
$$

Why annotate the type constructors? Types describe the structure of a value in a concise way. A type constructor corresponds with a certain area of memory. For example, the *Int* type constructor represents all bits of an integer value. But in *List Int* (algebraic data types are introduced later in this master's thesis, Chapter 8), the *List* type constructor represents the spine of a list, and the *Int* type constructor represents each element of the list (Figure 3.3). In fact, how values are treated depends on an operation semantics (for example, Peyton Jones [25]), although we leave that implicit in this master's thesis.

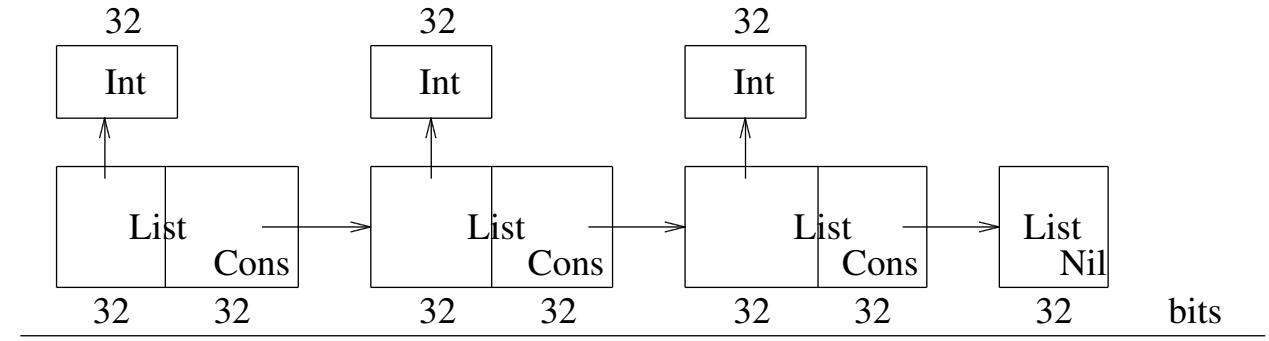


Figure 3.3: Representation of a *List* in memory

Each bit of the list belongs to a single type constructor, the *corresponding* type constructor. The cardinality of a single bit, is the cardinality of the corresponding type constructor. Or, the other way around, the cardinality of a type constructor is the *join* over all the cardinalities of the bits that correspond to the type constructor. So, the reason is that the type constructors allow us to specify cardinalities of bits in a concise way, although we loose the ability to give different cardinalities to bits represented by the same type constructor.

A related question is: why is the arrow type constructor annotated? There are several reasons. From a practical view considering code generation: functions are also values: *closures*. Such a value is represented by an area of memory, and can have a cardinality like any other value. From a more theoretical point of view, functions that are used exactly once have the property that they can be safely inlined [39]. So, the arrow type constructor requires an annotation as well.

3.4.1 The annotations

In this section, we introduce the annotations on a type constructor, their values and some orderings on this values. A type constructor contains a tripple with a lower-bound usage-count annotation (which we call the lower bound), a cardinality annotation, and an upper-bound usage-count annotation (which we call the upper bound). We consider these

annotations in more detail and define some orderings on it which will be used when checking constraints or inferring values later (see the summary at the end of this section).

The lower-bound usage-count annotation ℓ on a type constructor of a type originating from expression E, specifies that the value of E is used at least 0 or 1 times via E. A lower bound of 0 means that there is no guarantee that the value is used via E. The ℓ values are totally ordered with $1 \sqsubseteq_L 0$[5]. Note that the relation $\sqsubseteq_L$ corresponds to the relation $\geqslant$ if the ℓ values are considered natural numbers. We use this relation later to specify that when we know that E uses its value at least once, that we are allowed to forget it and think that we do not have any guarantee about how often the value of E is used. The definition of lower bound values can be generalised to lower bounds up to some value k, but we will not complicate this description with this generalisation.

The upper-bound usage-count annotation ι is the dual of the lower-bound usage-count annotation. It specifies that the value of E is used at most 0, 1, or $*$ times via E. An upper bound of 0 means that the value of E is never used via E. An upper bound of 1 specifies that E used its value at most once. It can occur that the value of E is never used via E, but an upper bound of 1 does not guarantee that. However, it does guarantee that E is not used twice or thrice via E. Finally, an upper bound of $*$ means that the value of E is used an arbitrary number of times via E, but probably twice or more. The ι values are totally ordered with: $0 \sqsubseteq_U 1$ and $1 \sqsubseteq_U *$. The $\leqslant$ relation on natural numbers corresponds to the above relation. The above relation is used in the checks to allow relaxation of an upper bound to a higher upper bound.

We use the $\sqsubseteq_L$ ordering to allow weakening of lower bounds, and the $\sqsubseteq_U$ ordening to weaken upper bounds.

These two bounds are not independent of each other. The upper bound should be greater than the lower bound, which is defined by the following predicate $\sqsubseteq_B$:

$$
\begin{aligned}
&\sqsubseteq_B :: \ell \to \iota \to \mathit{Bool} \\
&0 \sqsubseteq_B \_ = \mathit{True} \\
&1 \sqsubseteq_B u = 1 \sqsubseteq_U u \\
&\_ \sqsubseteq_B \_ = \mathit{False}
\end{aligned}
$$

This $\sqsubseteq_B$ predicate tests if the lower and upper bound are consistent with each other. For example, a lower bound of 1 is inconsistent an the upper bound of 0. This predicate is used to verify the consistency of a tripple. It is not used for weakening purposes.

An cardinality annotation consists of two $\jmath$ values. The left $\jmath$ value defines the usage of the value via E, and the right $\jmath$ value defines the total usage of the value. The left $\jmath$ value must fit the two bounds (Figure 3.6). The right $\jmath$ value is at most as specific as the left $\jmath$ value.

We define several orderings on $\jmath$ values. We start with an ordering that is used in the definition for consistency of cardinality annotations. This ordering, $\sqsubseteq_X$, allows the right-hand side of a cardinality annotation to deviate from the left-hand side. The amount of deviation is determined by $\sqsubseteq_X$. If $a \sqsubseteq_X b$ then a does not conflict with b. There is a conflict when the left $\jmath$ value specifies that the value is used at least k times and the right $\jmath$ value specifies that the value is used strictly less than k times. For example, $a = 1$ and $b = 0$ conflict. The following partial order defines when two values do not conflict (see Figure 3.4):

$$
\begin{aligned}
0 &\sqsubseteq_X 1^- \\
0 &\sqsubseteq_X 1^+ \\
1 &\sqsubseteq_X 1^+ \\
1^- &\sqsubseteq_X 1^+ \\
1^- &\sqsubseteq_X * \\
1^+ &\sqsubseteq_X *
\end{aligned}
$$

If the left-hand side of a cardinality annotation is $*$, then the only possibility is $*$ for the right-hand side. If we do not know the usage of one particular occurrence of some value, then we neither know it for the aggregation of all occurrences. Another interesting example is $1^- \sqsubseteq_X 1^+$. A specific occurrence of a value can be used at most time,

[5] All the $\sqsubseteq$ relations that we define are partial orders.

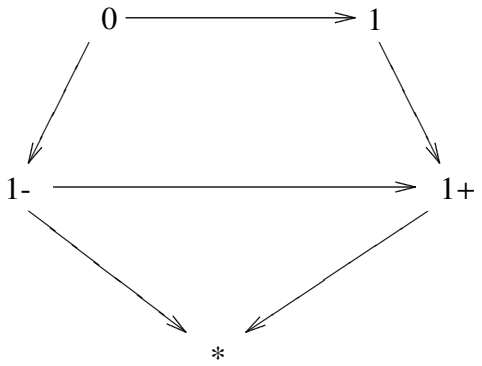


Figure 3.4: The does-not-conflict semi-lattice

but then it is still possible that the value is in total guaranteed to be used more than once. So, $\sqsubseteq_X$ is used to test if a cardinality annotation is valid (i.e. it is used as a predicate like $\sqsubseteq_B$).

Furthermore, we have a partial order for checking that one $\jmath$ value is more specific than another $\jmath$ value. If $a \sqsubseteq_Q b$, then a is more specific than b. It is defined as follows (see also Figure 3.5):

$0 \sqsubseteq_Q 1^-$
$1^- \sqsubseteq_Q *$
$1 \sqsubseteq_Q 1^+$
$1^+ \sqsubseteq_Q *$

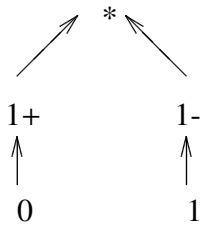


Figure 3.5: A semi-lattice on specificness of $\jmath$ values

The above ordering is (only) used in Chapter 4 to specify which right-hand side values of a cardinality annotation are preferable over others. $\jmath$ values of 0 and 1 are preferable over 1^- or $*$ because they are more precise. We do not use it yet in this chapter, since all the annotations are already given. But in the next chapter, we infer the annotations and start with the lowest value according to $\sqsubseteq_Q$ and gradually weaken the results according to $\sqsubseteq_Q$ until a fixpoint is reached. We already mention this ordering here for completeness.

Similar to $\sqsubseteq_Q$ is $\sqsubseteq_P$, which also specify that one $\jmath$ value is more specific than another $\jmath$ value, but then for the left-hand side values of cardinality annotations:

$0 \sqsubseteq_P *$
$1^- \sqsubseteq_P *$
$1 \sqsubseteq_P *$
$1^+ \sqsubseteq_P *$

The rationale behind this ordering is that the join between the left-hand side values of two cardinality annotations, does

not violate the lower bound and upper bound corresponding to these values. In the next chapter, there is a fixpoint iteration phase that starts with the most precise values for $\sqsubseteq_P$ that are possible for the bounds that were derived, which are then weaked according to $\sqsubseteq_P$ until a fixpoint is reached. In order not to violate any bounds during this weakening, weakening to $*$ is only allowed, since $*$ never violates any bounds.

We now only need two more orderings: one to weaken left-hand sides of cardinality annotations, and one to weaken the right-hand side of cardinality annotations. This orderings are subjected to compiler optimisations.

We define an ordering for coercions on the right-hand side $\jmath$ values of a cardinality annotation. A value with a total usage of u may be relaxed into a value with total usage u' (denoted by $u' \sqsubseteq_Z u$) if there is a coercion function that changes a value with usage of u to a value with usage u', such that optimisations of the compiler are still valid.

For example, we can coerce a value with total usage of 1^- to a value with total usage of $*$ by means of the identity function if we assume that the compiler only uses cardinality annotations to insert code for destructive updates on unique values. The reason for this particular example is that due to the fact that the coercion is the first use of the value, that there can only be unused references (besides the coercion) to the value that consider it to be used at most once. And after the coercion, any active reference assumes that the value is used in an arbitrary way, such that no inplace updates can be performed on it. On the other hand, if the compiler generates code that allocates values with a total usage of 1^- on a special compile-time garbage collected heap, then such a coercion is not allowed, unless there is an coercion function inserted into the generated code that copies the value to a normal garbage collected heap.

So, it depends on optimisations of the compiler and the availability of coercion functions to allow certain coercions between right-hand side values of cardinality annotations. Therefore, by default $\sqsubseteq_Z$ is an equality relation, but we assume that it can be a subset of $\sqsubseteq_S$ with left and right arguments swapped. Note that $a \sqsubseteq_Z b$ defines that the total usage b can be relaxed by coercion to total usage a. This is the inverse direction compared to the other orderings in this section. This is intentional: the two bound annotations and the left-hand side value of a cardinality annotation combine usage information of individual occurrences of a value into usage information about the total usage of some value. Subsequently, this aggregate result is passed back to the individual occurrences via the right-hand side value of the cardinality annotations. We are only allowed to forget some property about the total usage of a value, if our uses of the value do not conflict with optimized uses of the value at other places, except by doing some additional administrative work (the coercion).

We define a similar ordering as the ordering above for left-hand side of a cardinality annotation is relaxed. If $a \sqsubseteq_S b$, then a can be relaxed to b. This ordering is typically an equality relation, unless a relaxation does not conflict with optimisations of the back-end or can be resolved using a coercion function.

Or in other words: we can relax the left-hand side of a cardinality annotation only to $*$. However, there is an important restriction here: if we make a relaxation for one particular occurrence of some value, then we must make this relaxation for all annotations of this particular occurrence. For example,

The orderings on the left-hand side and right-hand side of a cardinality annotation can be combined into a single ordering on cardinality annotations as a whole:

$$
\begin{aligned}
a \bowtie b \sqsubseteq_C c \bowtie d \\
&= a \sqsubseteq_S c \\
&\wedge b \sqsubseteq_Z d \\
&\wedge a \sqsubseteq_X b \\
&\wedge c \sqsubseteq_X d
\end{aligned}
$$

This ordering tells us that a can be weakened to c, and b can be weakened to d, if there is a coercion from a to c and from b to d, and both a and b, and c and d, do not conflict with each other.

So, to summarize, we have three orderings used for weakening of results:

- $\sqsubseteq_L$ for the lower bound;

- $\sqsubseteq_U$ for the upper bound;
- $\sqsubseteq_C$ for the cardinality annotation (with $\sqsubseteq_S$ for the left-hand side and $\sqsubseteq_Z$ for the right-hand side).

And there are some orderings that are basically predicates:

- $\sqsubseteq_B$ for the two bounds;
- $\sqsubseteq_X$ for the left-hand side and right-hand side of a cardinality annotation.

3.4.2 Triple consistency

A triple is consistent if the left-hand side of the cardinality annotation fits into the boundaries given by the two boundary annotations, and the left-hand side of the cardinality annotation is consistent with the right-hand side. See Figure 3.6 for a specification. We do not allow a $\jmath$ value that might violate the bounds. For example, the $\jmath$ value 1^+ is not allowed when the upper bound is 0 or 1, because 'at least zero' or 'at least once' does not guarantee that the upper bound is violated. We make one exception to this rule for $*$. It always fits the bounds. We know that once $*$ has been derived for an individual occurrence of a value, that the value in total will have a $\jmath$ value of $*$. Nowhere in the program can guarantees be made about a value with $\jmath$ value of $*$, so the bounds can be ignored for this value. Strictly speaking, this is not required: we could enforce a stronger notion of fitting and require that the boundaries itself are relaxed to 0 for the lower bound or $*$ for the upper bound when the $\jmath$ value is $*$. However, in Chapter 4, we present an approach that infers the annotations, and the above relaxation allows us to infer lower and upper bounds separately from the cardinality annotations (Figure 4.3).

```
trippleConsistent (a, b ⋈ c, d)
    = a `leftFit`  b
    ∧ b `rightFit` d
    ∧ b ⊑X         c
    ∧ a ⊑B         d


leftFit :: ℓ → ȷ → Bool
leftFit 0  _  = True   -- bound gives no info
leftFit _  *  = True   -- 'don't know' fits bounds
leftFit 1  1  = True
leftFit 1  1+ = True
leftFit _  _  = False


rightFit :: ȷ → ι → Bool
rightFit _  *  = True   -- bound gives no info
rightFit *  _  = True   -- 'don't know' fits bounds
rightFit 0  1  = True
rightFit 1  1  = True
rightFit 1− 1  = True
rightFit 0  0  = True
rightFit _  _  = False
```

Figure 3.6: Consistency of triples

The orderings are important. The analysis gets as input an annotated program and produces a set of constraints that hold between the annotations, if the annotations are valid. The semantics of these constraints make heavy use of the fact that one annotation is greater or equal to another annotation, as we will see in Section 3.7.

There are three types of constraints for the type system in this chapter:

$$
\begin{array}{lll}
\mathit{constr} ::= (\ell, c, \imath) \triangleright (\ell, c, \imath) & & \text{-- coercion} \\
\quad | \ (\ell, c, \imath) \triangleright_s (\ell, c, \imath) & & \text{-- coercion (ignores use)} \\
\quad | \ (\ell, c, \imath) \odot \ldots \odot (\ell, c, \mathit{up}) \leqslant (\ell, c, \imath) & & \text{-- aggregation}
\end{array}
$$

As demonstrated in the examples, the coercion constraint ($\triangleright$) is used to propagate usage information. It specifies that the lower bound on the left-hand side is greater or equal than the lower bound on the right-hand side ($\mathit{left}\lambda \sqsubseteq_L \ \mathit{right}$). Vice versa, it specifies that the upper bound on the left-hand side is smaller or equal to the upper bound on the right-hand side ($\mathit{left} \ \sqsubseteq_U \ \mathit{right}$). Finally, it specifies how the cardinality annotations are relaxed: for the left-hand side $\jmath$ value from left to right, and for the right-hand side $\jmath$ value from right to left. In the last case only when there is a coercion.

With the coercion constraint, we essentially check that all boundary information is properly propagated from outermost expression to innermost expression, from body to parameters (Figure 3.7). The cardinalities of values bound to identifiers are treated separately for each use-site of an identifier and added to each other by using the aggregation constraint $\odot$. Section 3.7 describes how the constraints are checked.

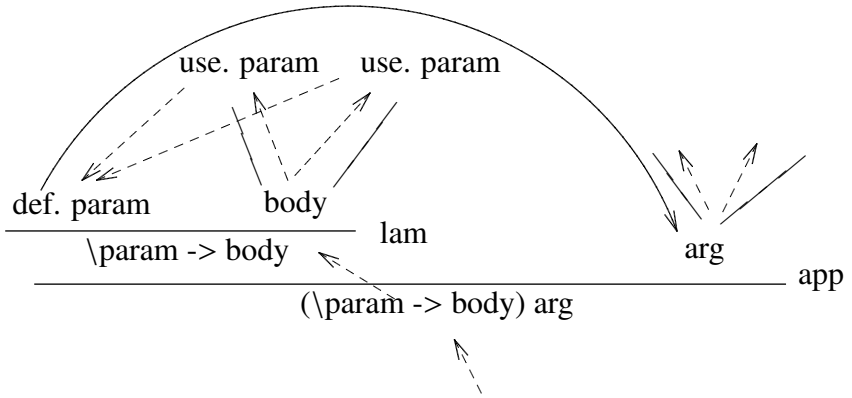


Figure 3.7: Count flow

3.5 Gathering constraints for first-order functions

This section lists type rules for the uniqueness type system. Figure 3.8 defines that an annotated program is valid when the gathered constraints are satisfied. Section 3.7 deals with constraint satisfiability. Constraints are gathered according to the expression type rules in Figure 3.9. The constraints are gathered bottom up: this only holds if all functions are first order. In the next section (Section 3.6), we present type rules that support higher-order functions (which are slightly more complicated). We will now consider the rules in more detail.

The type rule for the whole program (Figure 3.8) states that the resulting expression has a lower and upper bound of 1. By assuming that all the triples are consistent, this means that the resulting expression is allowed to have a cardinality of which the left-hand side is either 1 or $*$, although the total usage may be any $\jmath$ value except 0. We pattern match on the annotated type of the expression to obtain the annotation δ on the outermost type constructor and generate a constraint that specifies that the lower bound is at least 1 and the upper bound is at most 1. This constraint, and the constraints gathered from the expression itself, must be satisfied for the program to be valid according to the uniqueness type system.

Types are written in two ways in the type rules. A type written as $\widetilde{\tau}$ has annotations. A type written as τ has no annotations. We sometimes write τ_a to refer to $\widetilde{\tau}_a$ without annotations. A $\widetilde{\tau}$ is also denoted as τ^δ, where δ is the

$$\boxed{\Gamma \vdash^{prog} p}$$

$$\frac{\begin{array}{c} \textit{satisfied } \zeta_f \zeta \\ \Gamma \vdash^{unq} e : \widetilde{\tau} \rightsquigarrow \zeta \\ \widetilde{\tau} \equiv \_^{\delta} \\ \zeta_f \equiv \{(1, * \bowtie *, 1) \triangleright_s \delta\} \\ \textit{all triples consistent} \end{array}}{\Gamma \vdash^{prog} e : \widetilde{\tau}} \text{P}_{U1}$$

Figure 3.8: Program type rule (U1)

outermost[6] annotation.

The structure of the type rules of Figure 3.9 is as follows. Γ represents an environment that contains mappings from an identifier to a type (not annotated). The e stands for an expression with the annotated type $\widetilde{\tau}$. Finally ζ stands for a set of generated constraints.

A few words on syntax: we write $\zeta_1\ \zeta_2$ to denote a constraint set obtained by taking the union of ζ_1 with ζ_2. We write $\tau_a \cong \tau_b$ for the unification of τ_a with τ_b. Finally, unbound identifiers such as δ in the *Var* and *Lam* rule, specify that there are no limitations on that identifier.

$$\boxed{\Gamma \vdash^{unq} e : \widetilde{\tau} \rightsquigarrow \zeta}$$

$$\frac{\delta \textit{ arbitrary}}{\Gamma \vdash^{unq} n : \mathit{Int}^{\delta} \rightsquigarrow \emptyset} \text{E.INT}_{U1} \qquad \frac{i \mapsto \tau \in \Gamma \quad \widetilde{\tau} \equiv \tau^{\delta}}{\Gamma \vdash^{unq} i : \widetilde{\tau} \rightsquigarrow \emptyset} \text{E.VAR}_{U1}$$

$$\frac{\begin{array}{c} \Gamma \vdash^{unq} x : \mathit{Int}^{a} \rightsquigarrow \zeta_a \\ \Gamma \vdash^{unq} y : \mathit{Int}^{b} \rightsquigarrow \zeta_b \\ \zeta_{c1} \equiv \{\delta \triangleright_s a\} \\ \zeta_{c2} \equiv \{\delta \triangleright_s b\} \end{array}}{\Gamma \vdash^{unq} x\ +\ y : \mathit{Int}^{\delta} \rightsquigarrow \zeta_{c1}\zeta_{c2}\zeta_a\zeta_b} \text{E.PLUS}_{U1}$$

$$\frac{\begin{array}{c} \Gamma \vdash^{unq} a : \widetilde{\tau}_a \rightsquigarrow \zeta_a \\ \Gamma \vdash^{unq} f : \widetilde{\tau}_f \rightsquigarrow \zeta_f \\ \widetilde{\tau}_f \cong \widetilde{\tau}_a^{1} \rightarrow^{\delta_1} \widetilde{\tau}_r \\ \zeta_{m2} \equiv \{\delta \triangleright_s \delta_1\} \\ \widetilde{\tau} \equiv \_^{\delta} \\ \vdash^{flow} \widetilde{\tau} \triangleright \widetilde{\tau}_r \rightsquigarrow \zeta_r \\ \tau_p \cong \tau_a \\ \vdash^{flow} \widetilde{\tau}_p \triangleright \widetilde{\tau}_a \rightsquigarrow \zeta_{m1} \\ \tau_r \cong \tau \end{array}}{\Gamma \vdash^{unq} f\ a : \widetilde{\tau} \rightsquigarrow \zeta_{m1}\zeta_{m2}\zeta_r\zeta_a\zeta_f} \text{E.APP}_{U1}$$

$$\frac{\begin{array}{c} \delta \textit{ arbitrary} \\ x \mapsto \tau_x, \Gamma \vdash^{unq} b : \widetilde{\tau} \rightsquigarrow \zeta_b \\ x \vdash^{gather} b \rightsquigarrow \{x_1 : \widetilde{\tau}_1^{a_1}, ..., x_n : \widetilde{\tau}_n^{a_n}\} \\ \widetilde{\tau}_x \equiv \_^{a} \\ \zeta_{gat} \equiv \{a_1 \odot ... \odot a_n \leqslant a\} \end{array}}{\Gamma \vdash^{unq} \lambda x \rightarrow b : \widetilde{\tau}_x \rightarrow^{\delta} \widetilde{\tau} \rightsquigarrow \zeta_{gat}\zeta_b} \text{E.LAM}_{U1}$$

Figure 3.9: Expression type rules (U1)

Constraints for an expression are gathered by a syntax directed traversal over the expression. The case for integers is the easiest: integers can be produced for any usage annotation and nothing has to be checked for them. We neither have to check anything for the use-sites of identifiers, since we deal with these identifiers in the case for lambda abstraction. The idea is that the occurrences of identifiers are independent, and we combine the results at the lambda abstraction. This works only when the identifiers are not functions. If an identifier is a function, we need the constraint set corresponding to the function (Section 3.6).

[6] In this chapter, this is the triple on the outermost type constructor of the type expression. In the presence of algebraic data types, this is the outermost annotation on the result-kind of the type expression.

The case for integer addition is slightly more complicated. Aside from consistency, there are no demands on the uniqueness component of the result of the addition. On the other hand, we check that the upper and lower bound annotations of the parameters are properly propagated from result of the addition to the arguments of the addition. For that, a $\triangleright_s$ constraint is generated between the triple on the result type and the triples on the argument types.

The function application is perhaps the most difficult case in the uniqueness type system. Three things have to be done: check that the bounds of the function are in agreement with the bounds of the function application. Check that the bounds of the function application and the cardinality annotation are in agreement with the bounds and the cardinality annotation of the value of the function. Finally, check that the bounds and the cardinality annotation of the argument is in agreement with the bounds and usage annotation of the parameters.

For checking propagation, we bring to the attention that the value passed to a function, or the value returned by the function, can be a function itself. Propagation deals with all annotated type constructors in the type to make sure that argument and parameter, and function result and result, are properly connected. Note that variance plays a role here: the direction of an argument of a function is the opposite to the result of the function. Suppose that a function f is passed to function g as parameter, then g applies the function, and passes some argument x. Propagation is from x, to the parameter of g, and inside g to the result-value of g. The propagation of boundaries and usage information for such an annotation is the other way around than the usual flow. In Figure 3.10 we list the type rules that walk over the type and generate the $\triangleright$-constraints for each two corresponding annotations in the proper direction.

$$\boxed{\vdash^{flow} \widetilde{\tau} \triangleright \widetilde{\tau} \rightsquigarrow \zeta}$$

$$\frac{\zeta \equiv \{a \triangleright b\}}{\vdash^{flow} Int^a \triangleright Int^b \rightsquigarrow \zeta}\ \text{INT}_{U1} \qquad \frac{\begin{array}{c}\vdash^{flow} \widetilde{\tau}_b \triangleright \widetilde{\tau}_d \rightsquigarrow \zeta_1 \\ \vdash^{flow} \widetilde{\tau}_c \triangleright \widetilde{\tau}_a \rightsquigarrow \zeta_2 \\ \zeta_f \equiv \{a \triangleright b\}\end{array}}{\vdash^{flow} \widetilde{\tau}_a \rightarrow^a \widetilde{\tau}_b \triangleright \widetilde{\tau}_c \rightarrow^b \widetilde{\tau}_d \rightsquigarrow \zeta_f\zeta_1\zeta_2}\ \text{ARROW}_{U1}$$

Figure 3.10: Coercion generating type rules (U1)

The last case to deal with is lambda abstraction. We use an auxiliary rule that gathers all annotated types of the use-sites of the identifier of the lambda and turns these into an aggregation constraint. There is no constraint generated between the triple of the result of the body and the result-part of the annotated type of the lambda function. As is visible in the type rules, we force the annotated types to be exactly the same. This means that we cannot weaken the annotations of the body. But, a function application can weaken the bounds and usage information of the result, if needed.

As a final note about the type rules: the type system can be formulated in a less complicated, but not syntax directed, way. By adding an additional case for coercions, the places where we generated a $\triangleright$-constraint can be extracted and handled by this special non-deterministic rule. However, that would make it unclear at which places the $\triangleright$ constraint is generated, and that is actually what we want to make explicit, so we choose for a more elaborate, syntax directed, representation of the type rules.

3.6 Gathering constraints for higher-order functions

Consider the expression $(\lambda x \rightarrow x + x)\ 3$. Since x is not a function, it only has one triple (say δ) that describes how x is used. With only a single triple, there cannot be constraints between triples on the type of x. Since two occurrences of x can have a different triple than δ, there is actually a constraint for x: the aggregation constraint generated at the lambda. Conceptually, this constraint belongs to the (empty) constraint set of the two occurrences of x. However, there is no need to transport this constraint set to the occurrences of x because it is already included in the constraint set generated for the lambda. Figure 3.11 illustrates this: the thick lines indicates the propagation of constraint sets from the leafs to the root. The dotted lines correspond to conceptual constraint sets.

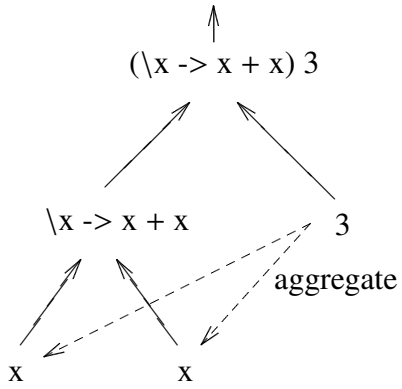


Figure 3.11: Propagation of constraint sets for first-order functions

Now consider the expression $(\lambda f \rightarrow f\ 3 + f\ 3)\ (\lambda x \rightarrow x)$. Since f is a function, there are relations between triples occurring on the type constructors of the type of f. In this case, there is a coercion between the result of f and the argument of f. The aggregation constraint only aggregates the triple on the spine of f, not the triples on the argument and result part of the type of f. This directly means that we do not check if each occurrence of f is correctly used according to the expression that is passed as f!

Since all annotations in this chapter are monovariant, we have an easy way out of this problem. The triples deeper into the type of an occurrence of f need to be equal to the corresponding triple in the definition-type of f. For example, if the definition site of f has the type $f :: Int^{(1,1\bowtie 1,1)} \rightarrow^{(1,1^{+}\bowtie 1^{+},*)} Int^{(1,1\bowtie 1,1)}$, then the occurrences of f are allowed to have a different annotation for the spine, such as: $Int^{(1,1\bowtie 1,1)} \rightarrow^{(1,1\bowtie 1^{+},*)} Int^{(1,1\bowtie 1,1)}$, but not: $Int^{(1,*\bowtie *,1)} \rightarrow^{(1,1\bowtie 1^{+},*)} Int^{(1,*\bowtie *,1)}$. We can enforce this by generating an equality constraint between these deeper triples of f during the generation of an aggregation constraint.

However, in the next chapter (Chapter 4) we support polyvariant cardinality annotations, such that we are able to apply f to arguments with different cardinalities. This requires a rather complicated propagation of constraint sets, as the use-sites of a higher-order parameter need the constraint set belonging to the expression that is passed as higher-order argument. Figure 3.12 illustrates this. The thick lines represents the leafs to root propagation of constraint sets as presented in Section 3.5. The tiny dotted lines are the original aggregation rules that only aggregate the spine annotation of use-sites of parameters. The dotted lines represent the propagation of constraint sets from argument expression to use site of a parameter. For example, the constraint set for the value 3 (which is empty) needs to be passed at the function application with f as the constraint set of the use-site of x in the constraint set of f. Likewise, at the application at the root, the constraint set of the argument should be passed as the constraint set for the two occurrences of f (this constraint set is not complete as it still requires a constraint set, which is provided by an application of f). We solve this constraint passing problem in the next chapter in a similar way as the spread function in Heeren [14], with receive points indicated using a special *Inst* constraint to represent a receive point and spread points as a special annotation on the arrow type constructor.

As preparation for the next chapter, we present a set of type rules that implement the above approach. It does not have any benefits for this chapter due to monovariance, but serves as a preparation for Chapter 4. Figure 3.13 lists the modified set of type rules. We now have two additional environments and function arrows have an additional annotation (a unique identifier). The environment Ψ consists of mappings from a unique identifier β to a constraint set. The environment Φ consists of mappings from identifiers of the expression to such a unique identifier β. The idea is that we give each parameter of a function a unique identifier β, which is recorded in Φ at a lambda abstraction. This β represents a receive point of a constraint set. At each use of an identifier, we lookup what the corresponding receive point is and demand that the future constraint set will become the constraint set of the use of the identifier. Because this β ends up as annotation on the function arrow and is used during unifications, we know at applications of this function how to refer to this β. The future constraint set is then provided by the function application by demanding that the receive point β corresponds with the constraint set resulting from argument expression.

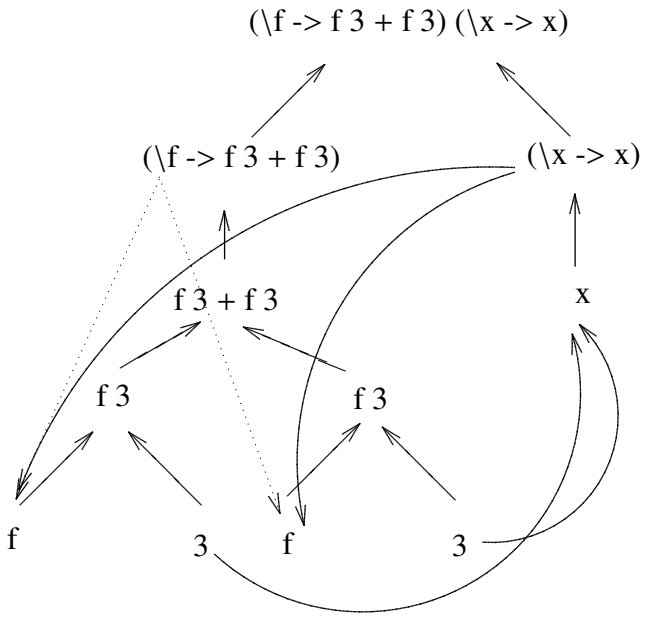


Figure 3.12: Propagation of constraint sets for higher-order functions

$$\boxed{\Phi;\Gamma \vdash^{unq} e : \widetilde{\tau} \rightsquigarrow \zeta;\Psi}$$

$$\frac{\delta \text{ arbitrary}}{\Phi;\Gamma \vdash^{unq} n : Int^{\delta} \rightsquigarrow \emptyset;\emptyset}\ \text{E.INT}_{U2} \qquad \frac{\begin{array}{c} i \mapsto \tau \in \Gamma \\ i \mapsto \beta \in \Phi \\ \widetilde{\tau} \equiv \tau^{\delta} \end{array}}{\Phi;\Gamma \vdash^{unq} i : \widetilde{\tau} \rightsquigarrow \zeta_{\psi};\{(\beta,\zeta_{\psi})\}}\ \text{E.VAR}_{U2}$$

$$\frac{\begin{array}{c} \Phi;\Gamma \vdash^{unq} x : Int^{a} \rightsquigarrow \zeta_a;\Psi_a \\ \Phi;\Gamma \vdash^{unq} y : Int^{b} \rightsquigarrow \zeta_b;\Psi_b \\ \zeta_{c1} \equiv \{\delta \triangleright_s a\} \\ \zeta_{c2} \equiv \{\delta \triangleright_s b\} \end{array}}{\Phi;\Gamma \vdash^{unq} x\ +\ y : Int^{\delta} \rightsquigarrow \zeta_{c1}\zeta_{c2}\zeta_a\zeta_b;\Psi_a\Psi_b}\ \text{E.PLUS}_{U2}$$

$$\frac{\begin{array}{c} \Phi;\Gamma \vdash^{unq} a : \widetilde{\tau}_a \rightsquigarrow \zeta_a;\Psi_a \\ \Phi;\Gamma \vdash^{unq} f : \widetilde{\tau}_f \rightsquigarrow \zeta_f;\Psi_f \\ \beta \mapsto \zeta_a \in \Psi_f \\ \widetilde{\tau}_f \cong \widetilde{\tau}_p \rightarrow_{\beta}^{\delta_1} \widetilde{\tau}_r \\ \zeta_{m2} \equiv \{\delta \triangleright_s \delta_1\} \\ \widetilde{\tau} \equiv \_^{\delta} \\ \vdash^{flow} \widetilde{\tau} \triangleright \widetilde{\tau}_r \rightsquigarrow \zeta_r \\ \vdash^{flow} \widetilde{\tau}_p \triangleright \widetilde{\tau}_a \rightsquigarrow \zeta_{m1} \\ \tau_p \cong \tau_a \\ \tau_r \cong \tau \end{array}}{\Phi;\Gamma \vdash^{unq} f\ a : \widetilde{\tau} \rightsquigarrow \zeta_{m1}\zeta_{m2}\zeta_r\zeta_a\zeta_f;\Psi_f\Psi_a}\ \text{E.APP}_{U2}$$

$$\frac{\begin{array}{c} \delta \text{ arbitrary} \\ x \mapsto \beta, \Phi; x \mapsto \tau_x, \Gamma \vdash^{unq} b : \widetilde{\tau} \rightsquigarrow \zeta_b;\Psi \\ x \vdash^{gather} b \rightsquigarrow \{x_1 : \widetilde{\tau}_1^{a_1}, ..., x_n : \widetilde{\tau}_n^{a_n}\} \\ \widetilde{\tau}_x \equiv \_^{a} \\ \zeta_{gat} \equiv \{a_1 \odot ... \odot a_n \leqslant a\} \end{array}}{\Phi;\Gamma \vdash^{unq} \lambda x \rightarrow b : \widetilde{\tau}_x \rightarrow_{\beta}^{\delta} \widetilde{\tau} \rightsquigarrow \zeta_{gat}\zeta_b;\Psi}\ \text{E.LAM}_{U2}$$

Figure 3.13: Expression type rules (U2)

Note that a constraint set of node e in the abstract syntax tree cannot spread to a node that is a child of e due to the fact that we only spread from argument to function in a function application. This ensures that we can write this spreading concisely using a few AG attributes: unifications already took place so we know what β is at the function application and at the use-sites of parameters. For a function application, we can then just add the synthesized constraint set of a function application paired with β to the inherited environment Ψ of the function expression, take out the constraint set at the use sites of the corresponding parameter, and then synthesize the constraints of the function expression, argument expression, and the coercion constraints of the function application as the constraint set of the function application. This is not the case for a recursive **let**. However, a solution will be presented in Chapter 5, based upon instantiation and the constraint graphs of Chapter 4.

3.7 Checking constraints

The following definition defines precisely when a constraint is satisfied or not:

$$
\begin{aligned}
&\mathit{satisfied} :: \mathit{Check} \\
&\mathit{satisfied}\ (a \triangleright b) &&= a\ \text{`}\mathit{coercionSatisfied}\text{`}\ b \\
&\mathit{satisfied}\ (a \triangleright_s b) &&= a\ \text{`}\mathit{softCoercionSatisfied}\text{`}\ b \\
&\mathit{satisfied}\ (a_1 \odot \ldots \odot a_2 \leqslant a) &&= [a_1 \ldots a_n]\ \text{`}\mathit{aggregationSatisfied}\text{`}\ a
\end{aligned}
$$

This function is lifted to work on constraint sets as a whole.

3.7.1 Coercion constraints

For a coercion constraint, the lower bound of the left-hand side should be greater than the lower bound of the right-hand side. It is the other way around for the upper bound. If the coercion is weak, then the cardinality annotation is ignored, otherwise the right-hand side should be more specific than the left-hand side:

$$
\begin{aligned}
&\mathit{coercionSatisfied}\ (a,b,c)\ (d,e,f) \\
&\quad = a \sqsubseteq_L d \\
&\quad \wedge\ b \sqsubseteq_C e \\
&\quad \wedge\ c \sqsubseteq_U f \\
&\mathit{softCoercionSatisfied}\ (a,b,c)\ (d,e,f) \\
&\quad = a \sqsubseteq_L d \\
&\quad \wedge\ c \sqsubseteq_U f
\end{aligned}
$$

This constraint is generated in such a way that we may weaken what we know of the right-hand side, to obtain the left-hand side. Depending on code generation, some coercion function may need to be inserted, for example, to move a value from a compile-time garbage collected heap to a runtime garbage collected heap. For a code generator that only selects special operations for types that are used at most once, no coercion function is required.

3.7.2 Aggregation constraint

For the aggregation constraint, we conservatively approximate the upper and lower bound. Things are a bit more complicated for the cardinality annotation. We check that the left-hand side of the cardinality annotation is properly split up to the individual use sites (Figure 3.14) and that the right-hand side is equal for all use-sites. We mean with this that if a value is linear, that the left-hand side of the cardinality annotation is linear for one use site, and not used for the other use sites. For the right-hand side of the cardinality, we check that all use-sites consider the total usage of the value to be linear. See Figure 3.15 for the definition.

$$
\begin{array}{lll}
\mathit{isSplit\ as\ 0} & = \mathit{all}\ (== 0)\ \mathit{as} \\
\mathit{isSplit\ as\ 1} & = \textbf{let}\ (\mathit{eq0}, \mathit{neq0}) = \mathit{partition}\ (== 0)\ \mathit{as} \\
 & \quad \textbf{in}\ \ \mathit{length\ neq0} == 1 \\
 & \qquad \wedge\ \mathit{head\ neq0} == 1 \\
\mathit{isSplit\ as\ 1^+} & = \mathit{length\ as} > 0 \\
 & \wedge\ \mathit{any}\ (== 1^+)\ \mathit{as} \\
 & \wedge\ \mathit{all}\ (\not\equiv 0)\ \mathit{as} \\
 & \wedge\ \mathit{all}\ (\not\equiv 1)\ \mathit{as} \\
\mathit{isSplit\ as\ 1^-} & = \mathit{all}\ (== 1^-)\ \mathit{as} \\
\mathit{isSplit\ as} * & = \mathit{all}\ (== *)\ \mathit{as}
\end{array}
$$

Figure 3.14: The isSplit function.

$$
\begin{array}{l}
\mathit{aggregationSatisfied}\ [(a_1, b_1 \bowtie c_1, d_1), ..., (a_n, b_n \bowtie c_n, d_n)]\ (a, b \bowtie c, d) \\
\quad = a_1\ `\mathit{max}`\ ...\ `\mathit{max}`\ a_n\ \sqsubseteq_L\ a \\
\quad \wedge\ \mathit{isSplit}\ [b_1, ..., b_n]\ b \\
\quad \wedge\ c_1 \equiv ... \equiv c_n \\
\quad \wedge\ d_1 + ... + d_n\ \sqsubseteq_U\ d
\end{array}
$$

Figure 3.15: aggregation constraint satisfaction

Note that when $n = 0$, $a_1\ `\mathit{max}`\ ...\ `\mathit{max}`\ a_n \equiv 0$ and $c_1 + ... + c_n \equiv 0$. If an identifier does not occur syntactically, then the value cannot be touched through that identifier, and the lower bound corresponding to the type of the identifier is forced to 0 and the upper bound can be anything.

All occurrences of an identifier point to the same value. So, if one occurrence of an identifier has a lower bound of 1, it means that the occurrence is evaluated at least once, so the value is used at least once. We approximate the lower bound by taking the maximum of all lower bounds.

At this moment, we do not distinguish between parallel and sequential occurrences of values. For parallel occurrences, we can use the maximum of the upper bounds instead of the sum, which is a better approximation. And in a similar way take the sum of the lower bounds instead of the maximum (note the duality here). We discuss this subject in Chapter 7.

3.8 Strictness?

As we mentioned in Chapter 1, we can also infer strictness with our approach. Strictness is defined as follows: consider a function $f = \lambda x \rightarrow ...$ and the application $r = f\ \bot$. If reduction of r to weak-head normal form results into $\bot$, then f is strict in x.

After performing the analysis that we explained in this chapter, we know for some parameters that they are used. This information almost tells us that the parameters are strict, but not entirely: in our analysis it is possible that we consider a value used, because the result of the function is reduced further than weak-head normal form.

But, we can obtain strictness properties of a function f with result type r. Suppose we only have the expression of f. If we use $(1, 1 \bowtie 1^+, 1)$ for the outermost annotation on r and $(0, 0 \bowtie 0, 0)$ for all other annotations of r, and expect a valid annotation, then f is strict in those parameters that can have a cardinality annotation with 1^+ or 1 as left-hand side value. An optimisation is that we can *seq* (Section 6.2 of the Haskell 98 report [29]) the parameters before applying them to the function. The annotations on the types further specify how deep the *seq* function can evaluate the value.

3.9 Conclusion

We showed in this chapter what we mean with uniqueness typing, albeit with a simple language. Our approach consists of a set of type rules that specify when a program is correctly annotated with cardinality annotations. We turned this problem into a constraint satisfaction problem, which gives us a separation between the gathering of the constraints and checking them. In the next chapter, we show that the constraints also allow us to *infer* cardinality annotations, instead of requiring that they are supplied by the programmer.

In this chapter, we also stated that cardinality annotations are not sufficient. We required lower-bound and upper-bound annotations as well. In the next chapter, we switch to inferring cardinality properties, and there we infer these bounds as well.

Chapter 4

Polyvariant analysis with monomorphic bindings

In the previous chapter, we generate constraints from an annotated program, and call the annotations valid if and only if the constraints are satisfied. In this chapter, we take a closer look at the constraints. We take a validly typed, but not annotated, program and decorate the type constructors with *cardinality variables* δ instead of concrete annotations. Then we generate the set of constraints, which now contain variables instead of concrete values. A solution is an assignment of concrete cardinality values to the annotations such that the constraints are satisfied. The constraints thus specify a whole family of solutions. In this chapter, we discuss a technique that infers the *minimal* solution to the constraints (Section 4.4).

To make this chapter slightly more challenging, we add a monomorphic and non-recursive **let** to the expression language defined in Chapter (Section 3). With the existence of cardinality variables, we turn the analysis into a polyvariant analysis, which means that we infer a most general uniqueness typing for a binding, and allow different instantiations for each use of an identifier.

This chapter is organized as follows. We illustrate the concept of cardinality inference by an example. Then we go into the concept of the type inference process (Section 4.3). We follow this up with a discussion about the addition of bindings to the expression language, and consequently the concept of binding groups and instantiation(Section 4.6). The constraints play an important role here, but we go to yet another representation: the constraints are rewritten to a graph-representation on which we can perform graph reduction (Section 4.7) to simplify the constraints.

4.1 Example

A type now only has cardinality variables as annotations. Upper and lower-bound cardinality annotations do not exist anymore, but are used internally by the inferencer. The following example demonstrates some annotated types and constraint sets:

$$
\begin{array}{ll}
f :: \forall\ \delta_1 & .Int^{\delta_1} \\
g :: \forall\ \delta_1 & .Int^{\delta_1}, (1^- \bowtie *) \triangleright \delta_1 \\
h :: \forall\ \delta_2 & .Int^{\delta_2} \rightarrow^{\delta_2} Int^{\delta_2} \\
j :: \forall\ \delta_3\ \delta_4\ \delta_5 & .Int^{\delta_4} \rightarrow^{\delta_3} Int^{\delta_5}, \delta_5 \triangleright \delta_4, \delta_5 \triangleright_s \delta_3
\end{array}
$$

All cardinality variables are universally quantified, but are constrained by a constraint set. For example, f represents a value with a type that has an unconstrained annotation δ_1. This annotation can be instantiated with any consistent cardinality value. The annotation on the type of g can only be $1^- \bowtie *$, or $* \bowtie *$, since the constraint disallows other

combinations (Section 3.7). The type of h corresponds to a function that may have any of the five cardinality values as long as the function, the argument, and the result type have the same cardinality value. The three annotations on the type of j can have any cardinality value as long as the two coercions are satisfied.

We annotate a program with cardinality variables, perform the constraint gathering of the previous chapter, and perform cardinality inference on the constraints. We remark that a cardinality annotation is a cardinality variable (i.e. δ), and a concrete cardinality annotation is a cardinality value (i.e. $1 \bowtie *$).

The first step in this process is to take a typed program and annotate it with fresh annotations:

$(\lambda x \to x)\ 3$
$((\lambda(x :: Int) \to x :: Int) :: Int \to Int)\ (3 :: Int) :: Int$

Each type constructor in the types is annotated with a fresh cardinality variable (the x parameter gets its annotation from the type of the function).

$(\lambda x \to x)\ 3$
$((\lambda(x :: Int^{\delta_1}) \to x :: Int^{\delta_2}) :: Int^{\delta_3} \to^{\delta_4} Int^{\delta_5})\ (3 :: Int^{\delta_6}) :: Int^{\delta_7}$

Constraint gathering of Section 3.5 results in the following constraints:

$\delta_0 \triangleright_s \delta_7$ -- root of the expression
$\delta_7 \triangleright \delta_5$ -- function application to function result
$\delta_7 \triangleright_s \delta_4$ -- function application to function spine
$\delta_3 \triangleright \delta_6$ -- function parameter to function type
$\delta_5 \triangleright \delta_2$ -- function result to function body
$\delta_2 \leqslant \delta_1$ -- aggregation of use-sites of x
$\delta_1 \triangleright \delta_3$ -- parameter to function

With a fixed cardinality variable δ_0 of which we set the initial substitution for the upper bound to 0 and the lower bound to 0.

These constraints specify a whole family of solutions. A solution is a substitution that maps each cardinality variable to a cardinality value and satisfies the constraints. The substitution that maps each cardinality variable to $* \bowtie *$ always satisfies the constraint. But in this case, a substitution that maps each cardinality variable to $1 \bowtie 1$, $1 \bowtie 1^+$, or $1^+ \bowtie 1^+$, also satisfies the constraints. Annotate the program with corresponding upper and lower bound annotations and apply the constraint checking of the previous chapter if in doubt.

The above constraints are passed to the inferencer (we call it the constraint solver), which infers the least solution to the constraints. With a least solution we mean a solution that is as specific as possible for each cardinality annotation (see the ordering $\sqsubseteq_S$ in Section 3.4). The $\jmath$ values 1 and 0 are the most specific (and mutual exclusive), the annotations 1^- and 1^+ annotations are in between (also mutual exclusive) and $*$ is the least specific $\jmath$ value. There is exactly one least solution to the constraints (see Section 4.4 for a proof). Keep in mind that this solution is still a conservative approximation of the actual usage of a value.

We obtain the solution as follows. Based upon the constraints, a lower and an upper bound is inferred for each cardinality annotation. After the lower and upper bounds are inferred, we choose the most specific $\jmath$ value for both $\jmath$ values of the cardinality annotations. We then relax (make them less specific according to $\sqsubseteq_S$) the left-hand side and right-hand side of the cardinality values independently until they satisfy the constraints. A relaxation of the left-hand side possibly leads to a relaxation of the right-hand side to keep the cardinality value consistent (see $\sqsubseteq_B$ in Section 3.4.1). Satisfying one constraint by relaxing one of the $\jmath$ values of a cardinality annotation, can cause a need for relaxation of the cardinalities of another constraint. This process is repeated until all cardinality values satisfies the constraints. This process terminates since we cannot relax $\jmath$ values beyond $*$.

The upper and lower bound are obtained in a similar way. An initial substitution is created that maps the lower bound of each cardinality variable to 1. Each coercion constraint that is not satisfied has a higher lower bound in the substitution for the right-hand side than the left-hand side. The constraint becomes satisfied by lowering the lower bound of the right-hand side to the lower-bound of the left-hand side. An aggregation constraint that is not satisfied, becomes satisfied by changing the right-hand side to the result of the computation of the left-hand side. This process is repeated until all constraints are satisfied. The same process is performed for the right-hand side, except that we start with the value 0. Performing this approach on the constraints of the example results in lower-bound values of all 1 (none of the constraints are satisfied to start with), and upper bound values of 1 (some work had to be done for this one).

The most specific cardinality value that fits (Figure 3.6) an upper bound and lower bound of 1, is the value 1 (linear). A cardinality of 1 satisfies all the above constraints. The substitution $S\ \delta_i = 1$, with $1 \leqslant i \leqslant 7$ is a least solution to the constraints of the above example.

4.2 Cardinality variables and type rules

In contrast to the previous chapter where the type constructors are annotated by a triple of annotations, we now only annotate the type constructors with a fresh cardinality variable. The example in Section 4.1 showed several types annotated with such cardinality variables.

Consider the following expression with types of every identifier and subexpression:

$$\begin{array}{l}\textbf{let}\ (f :: Int \to Int) = (\lambda x :: Int \to (x :: Int) + (1 :: Int) :: Int) :: Int \to Int \\ \textbf{in}\ \ (f :: Int \to Int)\ (1 :: Int) :: Int\end{array}$$

There are several ways in which we can annotate a program. One annotation strategy is to annotate all of these types with fresh cardinality variables. For example:

$$\begin{array}{l}\textbf{let}\ (f :: Int^{\delta_1} \to^{\delta_2} Int^{\delta_3}) = (\lambda(x :: Int^{\delta_4}) \to (x :: Int^{\delta_5}) + (1 :: Int^{\delta_6}) :: Int^{\delta_7}) :: Int^{\delta_8} \to^{\delta_9} Int^{\delta_{10}} \\ \textbf{in}\ \ (f :: Int^{\delta_{11}} \to^{\delta_{12}} Int^{\delta_{13}})\ (1 :: Int^{\delta_{14}}) :: Int^{\delta_{15}}\end{array}$$

As demonstrated in Chapter 3, the constraints between annotations make sure that the cardinality variables are properly connected to each other.

The above approach allows cardinality coercions everywhere in the abstract syntax tree. A cardinality coercion allows a relaxation of an established cardinality result to a weaker variant. For example, we can coerce a value with a total *usage* value of 1^- to a value with a total usage of $*$ by forgetting that the value is used at most once in total, if optimisations do not conflict with this coercion and possibly requires a coercion to be applied to the value at the moment of relaxation. There is some redundancy between annotations on some types. For example, a parameter of a function can use the cardinality variables of the enclosing function for its annotated type. It does not need fresh cardinality variables. Since we allow a coercion from argument to parameter, there is no need for a coercion from parameter to use-site of the parameter in the body of the function. Similarly, for a function application like such as $f\ 3$, it is possible to coerce the result of f at the function application. However, the result of $f\ 3$ can be used at three places: as argument of a function application, as body of a lambda, or as right-hand side of a **let** binding (later in this chapter). In case it is used as argument to another function application (i.e. $g\ (f\ 3)$), then the function application of g already allows an coercion. In case it is used as the body of lambda, we already allow a coercion from the type of the body to the type of the result of the lambda. Finally, we also already allow a coercion of the right-hand side of a **let**. So, a coercion of the result of a function at a function application is redundant. Coercions possibly lead to additional code depending on the code generator. We therefore want to limit the number of coercions that can be inserted, and restrict it to clearly defined places. Therefore, we restrict coercion to the three places mentioned above. This has no influence on the quality of the inference process. It leads to fewer coercion constraints and fewer annotations, which is beneficial for the performance of the compiler and possibly the performance of the program itself.

Consider the difference between the following example and the previous example:

$$\begin{array}{l}\textbf{let } (f :: Int^{\delta_1} \rightarrow^{\delta_2} Int^{\delta_3}) = (\lambda(x :: Int^{\delta_1}) \rightarrow (x :: Int^{\delta_4}) + (1 :: Int^{\delta_5}) :: Int^{\delta_6}) :: Int^{\delta_1} \rightarrow^{\delta_2} Int^{\delta_3} \\ \textbf{in } (f :: Int^{\delta_7} \rightarrow^{\delta_8} Int^{\delta_9}) \ (1 :: Int^{\delta_{10}}) :: Int^{\delta_{11}}\end{array}$$

With this strategy, fewer different cardinality annotations are used compared with the annotation strategy above. The difference is that we allow coercions only at restricted places. The result is less constraints and less coercion variables, which results in better performance of the inferencer. To scale up to typing big real-life programs, a cardinality inferencer can decide to reduce accuracy for certain unimportant pieces of a program, by limiting the coercions and reusing cardinality variables even more, to reduce the number of annotations severely, thus preventing choking on the number of annotations (Section 13.3). Limiting coercions can also positively impact runtime performance, depending on the additional code that is inserted for it, and what the code-improvements are, up to the point of the coercion.

Take a look at the example again. The types at the use-site of an identifier are freshly annotated, but this is unrelated to coercions. The reason for freshly annotating the use-sites of an identifier is because we treat each use of an identifier independently, and combine the results at the definition site with the aggregation constraint. For similar reasons, integer expressions and the spine of a lambda are freshly annotated: we can produce integers and functions for any cardinality annotation.

Figure 4.1 lists the type rules for the uniqueness type system with cardinality inference. The constraint generation remained virtually the same (see Section 3.6). The notable changes are the places where identifiers are freshly annotated, and a reduction in the number of coercion constraints. Furthermore, we take a copy of the constraint set of an identifier with the annotations of the definition site replaced by annotations of the use-site (the relation *rename* in the type rules), which is needed to ensure that we still know which types we are talking about.

The generated constraints are passed to the inferencer, which results in annotated types for each expression, and a substitution. The substitution maps each cardinality variable to a cardinality value.

4.3 The inferencer

The question that now remains is: how to obtain the substitution that assigns a cardinality value to a cardinality variable? Since we have a finite number of cardinality variables and a finite number of cardinality values, an obvious approach is to iterate through all combinations, and pick a least solution from the combinations that satisfy the constraints. A least solution is a lowest value in the ordering $\sqsubseteq_M$ such that the cardinality values satisfy the constraints:

$$\begin{array}{l}(c_1, ..., c_n) \ \sqsubseteq_M \ (d_1, ..., d_n) \\ \quad = c_1 \ \sqsubseteq_C \ d_1 \\ \quad \wedge \ ... \\ \quad \wedge \ c_n \ \sqsubseteq_C \ d_n\end{array}$$

Although it is not obvious from the definition $\sqsubseteq_M$, we show in Section 4.4 that there is only one least solution to a set of constrains. Unfortunately, the above approach is not feasible in practice. The problem is for n annotations, there are 5^n combinations, which is not feasible to calculate in practice.

Fortunately, there is no need to compute all possible combinations. Due to the orderings defined on the cardinalities, upper bound and lower bounds, we can use a fixpoint computation to obtain a least fixpoint. Actually, we need three of these computations: one for the lower bound, one for the upper bound, and one for the cardinalities. We first compute the lower and upper bounds. Then choose for each cardinality the most specific cardinality that fits with the two bounds. The third iteration process relaxes the cardinalities according to $\sqsubseteq_C$, until all constraints are satisfied. This means that there are three fixpoint computations on the same constraint set, each time with a different solving function. Figure 4.2 gives an overview of the implementation.

$$\boxed{\Phi; \Gamma \vdash^{unq} e : \widetilde{\tau} \rightsquigarrow \zeta; \Psi}$$

$$\frac{\delta fresh}{\Phi; \Gamma \vdash^{unq} n : Int^{\delta} \rightsquigarrow \emptyset; \emptyset}\ \text{E.INT}_{UX} \qquad \frac{\begin{array}{c}\vdash^{annot} \widetilde{\tau}_{env} \rightsquigarrow \widetilde{\tau} \\ \widetilde{\tau}_{env}; \widetilde{\tau} \vdash^{rename} \zeta_i \rightsquigarrow \zeta \\ i \mapsto \widetilde{\tau}_{env} \in \Gamma \\ i \mapsto \beta \in \Phi\end{array}}{\Phi; \Gamma \vdash^{unq} i : \widetilde{\tau} \rightsquigarrow \zeta; \{(\beta, \zeta_i)\}}\ \text{E.VAR}_{UX}$$

$$\frac{\begin{array}{c}\Phi; \Gamma \vdash^{unq} x : Int^{a} \rightsquigarrow \zeta_a; \Psi_a \\ \Phi; \Gamma \vdash^{unq} y : Int^{b} \rightsquigarrow \zeta_b; \Psi_b \\ \zeta_{c1} \equiv \{\delta \triangleright_s a\} \\ \zeta_{c2} \equiv \{\delta \triangleright_s b\} \\ \delta fresh\end{array}}{\Phi; \Gamma \vdash^{unq} x\ +\ y : Int^{\delta} \rightsquigarrow \zeta_{c1}\zeta_{c2}\zeta_a\zeta_b; \Psi_a\Psi_b}\ \text{E.PLUS}_{UX} \qquad \frac{\begin{array}{c}\Phi; \Gamma \vdash^{unq} a : \widetilde{\tau}_a \rightsquigarrow \zeta_a; \Psi_a \\ \Phi; \Gamma \vdash^{unq} f : \widetilde{\tau}_f \rightsquigarrow \zeta_f; \Psi_f \\ \beta \mapsto \zeta_{at} \in \Psi_f \\ \widetilde{\tau}_f \cong \widetilde{\tau}_p \rightarrow_{\beta}^{\delta_f} \widetilde{\tau} \\ \widetilde{\tau} \equiv \tau^{\delta} \\ \zeta_{md} \equiv \{\delta \triangleright_s \delta_f\} \\ \widetilde{\tau}_a; \widetilde{\tau}_p \vdash^{rename} \zeta_a \rightsquigarrow \zeta_{at} \\ \vdash^{flow} \widetilde{\tau}_p \triangleright \widetilde{\tau}_a \rightsquigarrow \zeta_{ma} \\ \tau_p \cong \tau_a\end{array}}{\Phi; \Gamma \vdash^{unq} f\ a : \widetilde{\tau} \rightsquigarrow \zeta_{md}\zeta_{ma}\zeta_a\zeta_f; \Psi_f\Psi_a}\ \text{E.APP}_{UX}$$

$$\frac{\begin{array}{c}x \mapsto \beta, \Phi; x \mapsto \tau_x, \Gamma \vdash^{unq} b : \widetilde{\tau}_b \rightsquigarrow \zeta_b; \Psi \\ x \vdash^{gather} b \rightsquigarrow \{x_1 : \widetilde{\tau}_1^{a_1}, ..., x_n : \widetilde{\tau}_n^{a_n}\} \\ \beta fresh \\ \vdash^{flow} \widetilde{\tau} \triangleright \widetilde{\tau}_b \rightsquigarrow \zeta_r \\ \delta fresh \\ \widetilde{\tau}_x \equiv \_^{a} \\ \zeta_{sum} \equiv \{a_1 \odot ... \odot a_n \leqslant a\}\end{array}}{\Phi; \Gamma \vdash^{unq} \lambda x \rightarrow b : \widetilde{\tau}_x \rightarrow_{\beta}^{\delta} \widetilde{\tau} \rightsquigarrow \zeta_{sum}\zeta_r\zeta_b; \Psi}\ \text{E.LAM}_{UX}$$

Figure 4.1: Expression type rules (UX)

```
infer :: ζ → Subst Cardinality
infer cs
  = let lowerSubst = worklistFix lowerSolveF cs [δ ↦ 1 | δ ∈ cs]
        upperSubst = worklistFix upperSolveF cs [δ ↦ 0 | δ ∈ cs]
    in  worklistFix cardSolveF cs [δ ↦ upperSubst δ `bestFit` lowerSubst δ | δ ∈ cs]


bestFit :: ℓ → ι → c
bestFit l u
    =  let z = l `bestFitAux` u
       in  z ⋈ z
  where
    bestFitAux :: ℓ → ι → ȷ
    bestFitAux 0 0 = 0
    bestFitAux 1 1 = 1
    bestFitAux 1 _ = 1⁺
    bestFitAux _ 1 = 1⁻
    bestFitAux _ _ = ∗


worklistFix :: (Constr a → Constr a) → ζ → Subst a → Subst a
worklistFix solveF cs initial
  = iter (toList cs) initial
  where
    deps = [(c, [d ∈ cs, c `shareVar` d, c ≢ d]) | c ∈ cs]
    iter [] subst = subst
    iter (c : cs) subst
      = let c1     = subst `apply` c
            c2     = solveF c1
            subst' = c2 `update` subst
        in  if subst' == subst
            then iter cs subst'
            else iter ((c `lookup` deps) ++ cs) subst'
```

Figure 4.2: Pseudo code of the inferencer

The fixpoint iteration proceeds as follows. We start with the best possible substitution, which is the lowest value in the ordering. If one particular value v does not satisfy a constraint, then v is too low in the ordering and we select a value v', with $v \sqsubseteq v'$ and v' as low as possible in the ordering and satisfying the constraint. We relaxed v to v', since v' can only be higher in the ordering. Such a value can always be found since the orderings are finite join-semilattices, which implies there is a value t such that $v \sqsubseteq t$ for any x. It is possible that satisfying one constraint makes another constraint not-satisfied. Processing the constraints once is called an iteration, and we repeat iterations until the substitution does not change anymore. If a substitution does not change during such an interation, then no relaxations took place, which implies that all constraints are satisfied. Furthermore, the number of relaxations is limited, since we have a finite substitution of which the value for each variable can be relaxed a finite number of times. A property of the fixpoint procedure is that a least fixpoint is reached after the process terminates.

For our constraints and their interpretation, there is only one such least fixpoint, and it is the optimal solution to the constraint set (Section 4.4). This approach is fast: when the constraints are topologically ordered, with the left-hand side(s) as dependency on the right-hand-side, and a special treatment for constraints that form a cycle (a change of the substitution of a variable in a cycle is directly propagated to the substitution of the other variables in the cycle), at most four iterations are required for the upper bound computation: three iterations (in the worst case) to get $*$ everywhere in the substitution, and one iteration to discover that the substitution did not change. With an implementation by means of a worklist algorithm, it takes even less time in practice, due to several reasons, including that the constraints represented as a graph (Section 4.7) are sparse. The worklist version takes a single constraint, ensures that it is satisfied, and if it changes the substitution, adds all dependent constraints on the stack of constraints to process. See Figure 4.2 for the worklist implementation.

The remaining question is what the solve functions are that make the substitution satisfy constraints. Figure 4.3 lists the solve functions. The worklist function applies the current substitution to a constraint, such that the solve function gets a constraint with concrete values filled in for the variables. The solve function returns the constraint with possibly increased values for the variables. The worklist function takes these values out of the constraint again and updates the substitution. This happens over and over again until the substitution does not change anymore. Compare these functions with the constraint checking functions of Section 3.7. The solve functions are similar to the constraint checking functions, which is not surprising since the iteration functions just increase values until they fit.

Figure 4.3 lists the solve functions. The solve functions for the lower bound and the upper bound are fairly straightforward. The β in the coercion constraint means that we do not differentiate between soft and hard coercion constraints for the lower and upper bounds. The solve function for cardinality values requires some explanation. The *increase* function minimally relaxes both $\jmath$ parameters until they coerce into each other. The *split* function relaxes $\jmath$ values minimally until usages are properly split over the individual occurrences. This means for example that if we know that the combined occurrences are used in total exactly once, then once of the occurrences must be used exactly once and the others not. If we know that the total is used more than once, then one of the individual occurrences must be used more than once and the others can be arbitrarily used.

4.4 Minimal solution

We mentioned in the introduction that the inferencer infers the minimal solution to the constraints. This minimal solution is computed by obtaining a least solution by means of a fixpoint iteration [23]. However, is this really a single least solution? We verify that this is the case. To simplify matters, only the upper bound annotations are considered. Proofs for the lower bound and the cardinality annotations are similar.

A solution in the context of this section is an upper bound substitution for the cardinality variables, such that the upper bound values satisfy the constraints.

Theorem 4.1 *There is exactly one least solution S to constraints ζ.*

Note that there is always a solution *top* to the constraints. We therefore have to prove that there is at most one least solution. For that, we take a few sidesteps first:

```
lowerSolveF (a ▷β b)
    = a ▷β (a `min` b)
lowerSolveF (a1 ⊙ ... ⊙ an ⩽ a)
    = let z = a1 `max` ... `max` an    -- 0 if n == 0
      in  (a1 ⊙ ... ⊙ an ⩽ (a `min` z))


upperSolveF (a ▷β b)
    = a ▷β (a `max` b)
upperSolveF (a1 ⊙ ... ⊙ an ⩽ a)
    = let z = a1 + ... + an    -- 0 if n == 0
      in  (a1 ⊙ ... ⊙ an ⩽ (a `max` z))


cardSolveF c @(_ ▷s _)    -- soft constraint ignores cardinality
    = c
cardSolveF (a ⋈ b ▷h c ⋈ d)
    = let (a1, c1) = increase (⊑P) (⊑S) a c
          (b1, d1) = increase (⊑Q) (⊑Z) b d
          b2 = a1 `⊔⊑X` b1
          d2 = c1 `⊔⊑X` d1
      in  (a1 ⋈ b2 ▷h c1 ⋈ d2)
cardSolveF (a1 ⋈ b1 ⊙ ... ⊙ an ⋈ b1 ⩽ a ⋈ b)
    = let (a11, ..., a1n, a1) = split (a1, ..., ab, a)
          b11 = a11 `⊔⊑X` b1
          ...
          b1n = a1n `⊔⊑X` bn
          b1  = a1  `⊔⊑X` b
          b2 = b11 `⊔⊑Q` ... `⊔⊑Q` b1n
      in  (a11 ⋈ b2 ⊙ ... ⊙ a1n ⋈ b2 ⩽ a1 ⋈ b2)


split c @(0, ..., 0, 0)       -- all exactly 0
    = c
split c @(a1, ..., an, 1)    -- one exactly 1, the others exactly 0
    | length neq0 == 1 ∧ head neq0 == 1
      = c
  where
    (_, neq0) = partition (== 0) [a1, ..., an]
split (a1, ..., an, a)          -- all 1+ or all 1- or all *
    = let z = a1 `⊔⊑P` ... `⊔⊑P` an `⊔⊑P` a
      in  (z, ..., z, z)


increase relaxR coercionR a b
    = min(⊑Q,⊑Q) [(a', b') | a' ← ȷ, b' ← ȷ
                           , a `relaxR` a'
                           , b `relaxR` b'
                           , a `coercionR` b' ]
```

Figure 4.3: Solve functions

Lemma 4.2 *Substitution S is a solution of the constraints ζ if and only if $S = solve\ \zeta\ S$, with $S_0 \leqslant S$, where S_0 is the default initial substitution.*

Proof The fixpoint iteration process does not terminate until all constraints in ζ are satisfied. If S is a solution, then all constraints are satisfied, and the fixpoint iteration results in S. ■

Collorary 4.3 *Substitution S is a solution of constraint set ζ if the substitution during any fixpoint iteration step of solve ζ S remains invariant.*

Proof By induction on the structure of the *solve* function, using Lemma 4.2. ■

With Lemma 4.2 we can prove that a substitution is a solution by showing that a substitution remains invariant during a fixpoint iteration step. We use this lemma to prove the following lemma:

Lemma 4.4 *Given solutions S and R, the minimum Z of these solutions, defined as $Z \equiv S\ `min`\ R \equiv \{S\ (\delta)\ `min`\ R\ (\delta) \mid \delta \in annotations\}$, is also a solution.*

Proof Take S, R, and Z as specified above. Now, consider a solve step with constraint C for *infer* ζ Z.

Suppose $C \equiv \delta_1 \triangleright \delta_2$. Since fixpoint iteration is completed for S and R, $S\ (\delta_1) \leqslant S\ (\delta_2)$ and $R\ (\delta_1) \leqslant R\ (\delta_2)$. Since $Z\ (\delta_1) \equiv S\ (\delta_1)\ `min`\ R\ (\delta_1)$, we conclude that by transitivity $Z\ (\delta_1) \leqslant S\ (\delta_2)$ and $Z\ (\delta_1) \leqslant R\ (\delta_2)$, which implies that $Z\ (\delta_1) \leqslant S\ (\delta_2)\ `min`\ S\ (\delta_2) = Z\ (\delta_2)$. This means that $Z\ (\delta_1)\ `\sqcup`\ Z\ (\delta_2) = Z\ (\delta_2)$, and we conclude that Z does not change.

Suppose that $C \equiv \sum_i\ \delta_i \leqslant \delta$. Since fixpoint iteration is completed for S and R, $\sum_i\ S\ (\delta_i) \leqslant S\ (\delta)$ and $\sum_i\ R\ (\delta_i) \leqslant R\ (\delta)$. Analogous to the previous section, we conclude that $\sum_i\ Z\ (\delta_i) \leqslant S\ (\delta)$ and $\sum_i\ Z\ (\delta_i) \leqslant R\ (\delta)$, which means that $\sum_i\ Z\ (\delta_i) \leqslant S\ (\delta)\ `min`\ R\ (\delta) = Z\ (\delta)$, and we conclude that Z does not change.

We therefore conclude that Z remains invariant during the fixpoint iteration process and conclude by Lemma 4.2 that Z is a solution. ■

So, taking the minimum of two solutions gives another solution. With this fact, we prove Theorem 4.1:

Proof Assume that the opposite is the case, such that P and Q are different least solutions. In order for P and Q to be different, there must be some annotation δ_0 for which $Q\ (\delta_0) < P\ (\delta_0)$ (the other way around as well, but we do not need that here). Consider the solution obtained by taking the minimum from P and Q: $M \equiv P\ `min`\ Q$. For each annotation δ, $M\ (\delta) \leqslant P\ (\delta)$. But for δ_0, $M\ (\delta_0) = Q\ (\delta_0) < P\ (\delta_0)$. This means that M is a smaller solution that P. Thus P cannot be a least solution, and that there is only one least solution. ■

Proofs for the lower bound, and even the cardinality values, are analogous to this proof. This means that we compute the best solution or 'most general' solution to the constraints. The constraints can be considered part of the type language, which means that our analysis gives a principal uniqueness typing.

4.5 Adding a non-recursive let

In Chapter 3 we showed how to gather the constraints. In this chapter, we showed how to infer the uniqueness types given the constraints, using a fixpoint iteration approach. We are now going to make life a bit interesting by supporting a non-recursive **let**. This will complicate the inferencer, especially since we are paving the way to support a polymorphic and recursive **let** for Chapter 5 in Chapter 6. A gentle warning: make sure you have a feeling of the inference process up to this point, otherwise subsequent sections and chapters will be difficult to understand!

The remainder of this chapter describes how to deal with a non-recursive **let**. We show that we can deal with this language construct by means of constraint duplication. We can handle this constraint duplication during the constraint gathering phase, but also during the inference phase. There are several reasons to deal with constraint duplication

during the inference phase. Some of the reasons will only become clear in Chapter 5 and Chapter 6, but we can already mention an advantage in this chapter. Constraint duplication leads to an explosion in the number of constraints, but we keep this constraint set manageable by applying several simplification strategies (Section 4.7). Some of these strategies can benefit from upper and lower bounds that are incrementally computed during the inference phase.

Before we go into the details and complications of the uniqueness inference process for the **let**, we first discuss how to deal with it during the constraint gathering phase. Assume we have the following **let**-construct:

let *id* = *def_expr*
in *body_expr*

Technically speaking, there is no use for allowing more than one binding to occur in a **let**, since the **let** is non-recursive and monomorphic as well. However, in preparation for subsequent chapters, we assume there can be many bindings inside the same **let**, although we often only use one binding in each **let** for type rules and examples.

For the conventional type system, a non-recursive, monomorphic **let** is a trivial addition to the type system: infer a type τ for *def_expr* with the same environment Γ as the entire expression, then use τ for each occurrence of *id* by adding it to Γ for *body_expr*. Since the **let** is non-recursive, the environment for *def_expr* does not need knowledge about *id*, but that will change once we want to support a recursive **let** in Chapter 5. This gives the type rule:

$$\frac{x_m \mapsto \tau_x, \Gamma \vdash^{\mathbf{type}} body : \tau \qquad \Gamma \vdash^{\mathbf{type}} expr : \tau_x}{\Gamma \vdash^{\mathbf{type}} \mathbf{let}_m x = expr \ \mathbf{in}\ body : \tau}\ \text{E.LET}_{EL}$$

We can apply the same strategy for the uniqueness type system. A set of constraints S is inferred for *expr_def*. For each use of *id*, S is copied and becomes the constraint set of the identifier. To show why this approach works, remember that beta-reduction allows us to substitute an identifier by its definition.

Suppose the occurrences $1 \ldots n$ of *id* in *body_expr* are replaced by *def_expr* to $inst\_expr_1 \ldots inst\_expr_n$. In other words, in the expression:

let *id* = *def_expr*
in ... id_1 ... id_n ...

The definition for *id* is inlined to (beta-reduction):

... $inst\_expr_1$ ... $inst\_expr_n$ ...

We now compare the constraint sets resulting from constraint gathering on *def_expr* and each $inst\_expr_i$. The constraint sets of such an $inst\_expr_i$ is a copy of S, except that the cardinality variables are different. The reason for this difference is due to the fresh annotations on the types occurring in $inst\_expr_1 \ldots inst\_expr_n$ (Section 4.2).

There is one other difference in the constraint set of the whole expression. If *def_expr* uses some identifier x from an outer binding group, then the x occurs n times more often in the inlined version. The corresponding aggregation constraint will be n times bigger. We solve this difference with a trick: when instantiating the constraint set of *def_expr*, the cardinality variables occurring on the use-site-types of identifiers are not given fresh variables (unless they are an argument or result of a function, but that is not important for this explanation). This gives us n of the smaller aggregation constraints, slightly different, but gives a conservative result towards the n times as big aggregation constraint, as the following example shows:

let x = ...
    $y = x \ldots x$
in $y \ldots y$

let $x = ...$
in $(x \; ... \; x) \; ... \; (x \; ... \; x)$

In the version where y is inlined, we compute the bounds of x directly. In the inlined version, the bound of x are computed from the bounds of y. This potentially leads to a less accurate, but conservative result, since we already approximate the bounds of y. This means that the constraint set that we obtain by separately analysing y and inserting this constraint set at each use-site of y, gives solutions that are equal, or conservatively less accurate, compared to solutions obtained from analysing an expression without a **let**.

There is a caveat here. The above assumption only holds in the presence of functions if we assume that all free variables of a function are at least as often used as the function itself. Consider the following expression:

let $y = 1$
$\quad f = \lambda x \to x + y$
in $f \; 1 + f \; 2$

The free variable y is hidden by f, but a use of f indirectly leads to a use of y. Normally, the aggregation constraint deals with this situation, but the aggregation constraint does not go deeper than the spine of the function; it does not aggregate the parameters or result of a function. By putting a soft coercion constraint between y and f (outermost annotation of y and the annotation on the function arrow of the lambda abstraction), we ensure that the aggregation constraint does aggregate uses of y. Later in this chapter we introduce constraint graphs, which can be used to improve this approximation. For example, if there is no path between the result of a function g and a free variable v (for an instantiation of g), then the constraint between g and v may be omitted. This is similar to an approach in Section 7.2. However, we leave this particular topic as future work.

There is an alternative way to discover how to deal with a **let**. Consider the function $(\lambda f \to f \; 3 + f \; 3) \; (\lambda x \to x)$ from Chapter 3.6. There is an equivalent way to write this function:

let $f = \lambda x \to x$
in $f \; 3 + f \; 3$

This expression is exactly the same because the let is monomorphic. Since we use a full polyvariant analysis, we expect the constraint set of the let-expression to equal the constraint set of the function-expression. The function expression copied the constraint set of $\lambda x \to x$ to the use-sites of f. In our description of how to deal with a **let**, we copy the constraint set of f—which is the constraint set of $\lambda x \to x$—to the use-sites of f. So, technically speaking, this is nothing new!

We now present a type system that uses the above approach to type **let**s. The type scheme has an additional result ω, which represents the constraints of a binding group. Note that we consider an argument expression to be a (pseudo) nested binding group of the binding group of the function application, since conceptually that is the same. The type scheme is now:

$\Gamma \vdash^{unq} e : \widetilde{\tau} \rightsquigarrow \zeta; \omega$

The inferencer performs the constraint duplication. In the inferred constraint set, we insert a placeholder to represent the constraints that are to be inserted, by means of the following constraint:

Inst < *binding_group_id* >	-- Identifier of binding group
[(*replace_binding_group_id*, *by_binding_goup_id*)]	-- How to map Inst constraints
[(<*definition_site_uniqueness_variable*>, <*use_site_uniqueness_variable*>)]	-- How to map the annotations

This constraint is inserted at the use-site of an identifier and at a function application. We use the relation *genInst* in the type rule to convert a source and destination type into an *Inst* constraint. This *Inst* constraint specifies that we take the constraint set for the identifier (or argument expression) and insert a fresh copy at the use-site. This constraint sets can

contain *Inst* constraints. These *Inst* constraint point to some binding group β, which is known at the use site. Therefore, the *Inst* constraint contains a mapping how to rename the *Inst* constraints of the definition site, such that they refer to the binding groups of the use site. These β annotations can be considered universally quantified phantom variables on type constructors, which are instantiated at a use site. We have to keep track of where they are instantiated too; this is why we add a list of pairs to the *Inst* constraint. We already mentioned that the constraint set is the same, but the cardinality annotations differ. The list of pairs in the constraint specify which cardinality variables in the constraint set of the definition-site are mapped to which cardinality variables of the use-site. The other cardinality variables occurring in the definition-site constraint set are replaced by fresh variables in the inferencer (Section 4.3). This gives us the following rule for identifiers:

$$\frac{\begin{array}{c}\vdash^{annot} \widetilde{\tau}_b \rightsquigarrow \widetilde{\tau} \\ \widetilde{\tau}_i;\widetilde{\tau};pairs \vdash^{genInst} \beta \rightsquigarrow \zeta \\ \tau_b \rightsquigarrow \tau;pairs \\ i_\beta \mapsto \widetilde{\tau}_b \in \Gamma\end{array}}{\Gamma \vdash^{unq} i : \widetilde{\tau} \rightsquigarrow \zeta;\emptyset}\ \text{E.VAR}_{UL} \qquad \frac{\begin{array}{c}\Gamma \vdash^{unq} e : \widetilde{\tau}_a \rightsquigarrow \zeta_a;\omega_a \\ \Gamma \vdash^{unq} e : \widetilde{\tau}_f \rightsquigarrow \zeta_f;\omega_f \\ \tau_r \cong \tau \\ \widetilde{\tau}_f \cong \widetilde{\tau}_p \rightarrow^{\delta_1}_{\beta} \widetilde{\tau}_r \\ \widetilde{\tau}_a;\widetilde{\tau}_p;[\,] \vdash^{genInst} \beta \rightsquigarrow \zeta_{ap} \\ \vdash^{flow} \widetilde{\tau}_p \triangleright \widetilde{\tau}_a \rightsquigarrow \zeta_{m1} \\ \zeta_{m2} \equiv \{\delta \triangleright_s \delta_1\} \\ \widetilde{\tau} \equiv \_^{\delta} \\ \vdash^{flow} \widetilde{\tau} \triangleright \widetilde{\tau}_r \rightsquigarrow \zeta_r \\ \tau_p \cong \tau_a\end{array}}{\Gamma \vdash^{unq} e : \widetilde{\tau} \rightsquigarrow \zeta_{m1}\zeta_{m2}\zeta_r\zeta_{ap}\zeta_f;\{(\beta,\zeta_a)\}\omega_f\omega_a}\ \text{E.APP}_{UL}$$

A nice result is that the type rules do not have the seemingly complex propagation of constraint sets: the inferencer takes care of it.

There is a slight optimization possible. If a parameter of a function is not a function itself, then we do not need to generate an *Inst* constraint for it. The reason is that the aggregation constraint properly relates the annotations of such a variable.

For the **let**, we have the following type rule:

$$\frac{\begin{array}{c}\vdash^{annot} \widetilde{\tau}_e \rightsquigarrow \widetilde{\tau}_x \\ \Gamma \vdash^{unq} expr : \widetilde{\tau}_e \rightsquigarrow \zeta_e;\omega_e \\ x \vdash^{gather} expr, body \rightsquigarrow \{x_1 : \widetilde{\tau}_1^{a_1}, ..., x_n : \widetilde{\tau}_n^{a_n}\} \\ x_m \mapsto \widetilde{\tau}_x, \Gamma \vdash^{unq} body : \widetilde{\tau} \rightsquigarrow \zeta_b;\omega_b \\ \widetilde{\tau}_x \equiv \_^{\delta} \\ \zeta_{sum} \equiv \{a_1 \odot ... \odot a_n \leqslant \delta\}\end{array}}{\Gamma \vdash^{unq} \mathbf{let}_m x = expr\ \mathbf{in}\ body : \widetilde{\tau} \rightsquigarrow \zeta_{sum}\zeta_b;\{(m,\zeta_e)\}\omega_e\omega_b}\ \text{E.LET}_{UL}$$

The term *binding group* refers to the bindings of the **let**. We assume that each **let** is numbered. This number identifies the binding group. In EH, each **let** only contains a single binding group, but technically speaking, the **let** can be partitioned in multiple binding groups (such as in Haskell). Binding groups are nested. Binding group A is a parent of binding group B if and only if B occurs in an expression of A. The body of the **let** belongs to the same binding group of the enclosing expression. An identifier in binding group C is also in binding group D if and only if D equals C, or D is a parent of C. Two identifiers are in the same binding group if and only if the binding group of one identifier is the parent of the binding group of the other identifier. A binding group is outermost if it has no parent.

The type rule for the **let** specifies that a compiler performs two tasks for a **let**: generate a constraint to aggregate the cardinalities of individual occurrences of the identifier bound to the **let**, and collect constraints of the binding group.

In the end, a result of the constraint gathering phase is a set of constraints for each binding group. These constraints are passed to the inferencer. The inferencer combines the individual results and computes a least substitution.

4.6 The inferencer revisited

The inferencer processes the constraint sets of binding groups in the order such that if A is a constraint set containing an *Inst* constraint referencing binding group B, then B is processed before A. In other words: the binding groups are processed in the order specified by a topological sort on the dependency graph between constraint sets. Since the binding groups are non-recursive, it is guaranteed that when the inferencer processes a binding group, that all the binding groups it depends on have already been processed.

The processing of a binding group consists essentially of getting rid of the *Inst* constraints, and replacing them by normal constraints by making a fresh copy of the originating constraint set. Not every cardinality variable may be replaced by a fresh version when making a copy: the originating and destination binding group may share some annotation variables of which the relations need to be preserved. These annotations we call the *sacred* variables.

The sacred variables, or sacred annotations, are those cardinality variables that may not be replaced by a fresh cardinality variable. These sacred variables are the annotations on the definition-site types of identifiers and the annotations on use-site types of identifiers that are in scope for a binding group. Basically, the sacred annotations are the annotations on identifiers that are directly influenced by and visible to the binding group. The number of sacred annotation variables is linear in the size of the abstract syntax tree. The sacred variables are collected and passed to the inferencer. We omitted this detail in the type rules.

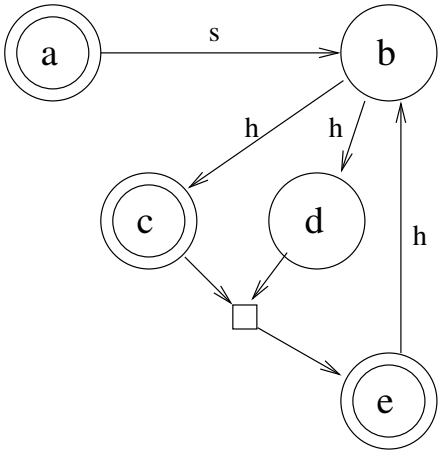
After copying the constraint sets, we end up with a normal constraint set again, and can continue with the cardinality inference as specified in the first part of this chapter. However, replacing the *Inst* constraint by the constraint set of the corresponding binding group, tends to make the number of constraints explode. We take a different approach, by converting each constraint set to an equivalent graph representation: the *constraint graph*. By simplifying the graph, it becomes small enough for practical use (Section 4.7).

The graph representation of a constraint set ζ is obtained as follows. Let $G = (V, E, H)$ be a directed graph with vertices V, edges E, and hyper edges H. The vertices of G are the cardinality annotations of ζ: $\delta \in V$ if and only if δ `` `isAnnotation` `` ζ. The labeled edges of G are the coercion constraints of ζ: $E\ (\delta_1, \delta_2) = \beta$ if and only if $(\delta_1 \triangleright_\beta \delta_2) \in \zeta$. The labeled hyper edges of G are aggregation constraints of ζ: $H\ [\delta_1, ..., \delta_n] = (\delta, \mathit{Nothing})$ if and only if $(\delta_1 \odot ... \odot \delta_n \leqslant \delta) \in \zeta$. This *Nothing* value is a label on the block symbol of a hyper edge. We come back to this value later.

As an example of the above approach, take the following constraint set (we make an explicit difference between the soft and hard versions of the coercion constraints):

$$
\begin{aligned}
&a \triangleright_s b \\
&b \triangleright_h c \\
&b \triangleright_h d \\
&c \odot d \leqslant e \\
&e \triangleright_h b
\end{aligned}
$$

The variables $a\,..\,e$ become vertices in the graph. The coercion constraints become the edges and the aggregation constraints become hyper edges. Suppose that the variables a, c, and e are sacred cardinality variables. We visually represent sacred vertices with a double circle, and normal vertices with just a single circle. This results in the graph:



To calculate an upper bound substitution from initial substitution $S\ (a) = 1, S\ (\delta) = 0, \delta \in [b\,..\,e]$, we iterate over the (hyper) edges of the graph. The edge between a and b, combined with the substitution $S\ (a) = 1$, force $S\ (b)$ to 1. $S\ (b) = 1$ forces both $S\ (c)$ and $S\ (d)$ to 1. The hyper edge combines $S\ (c) = 1$ and $S\ (d) = 1$, resulting in $*$, and forces $S\ (b)$ to $*$. Repeating the above procedure forces $S\ (c)$ and $S\ (d)$ to $*$. The hyper edge combines $S\ (c) = *$ and $S\ (d) = *$ in $*$ for $S\ (b)$, which is already the case. None of the edges in the graph change the substitution anymore, and the iteration stops with a final substitution S', with $S'\ (a) = 1, S'\ (\delta) = *, \delta \in [b\,..\,e]$.

Figure 4.4 gives an overview of inferencer. The inferencer constructs the constraint graph of the outermost binding group and performs a fixpoint substitution calculation on it. To obtain this graph, the inferencer takes the constraint set without *Inst* constraints and creates a constraint graph out of it. Then we fill these missing holes by constructing the constraint graphs belonging to the *Inst* constraints. Note that we make a difference between constraint graphs and binding groups here. A binding group can have multiple constraint graphs if it contains higher-order functions. In these cases, there is a separate constraint graph for each instantiation of an identifier of the binding group occurring in the program.

```
inferSubst constrSets
    =  infer (construct (outermostBndgId, [ ]))
  where
    construct key @(bndgId, idMap)
      | inMemo key = key `lookup` memo
      | otherwise  = let cs  = bndgId `lookup` constrSets
                         cs' = rename idMap cs
                         gr  = csToGraph cs'
                         gr' = foldr (instantiate bndgId) gr (insts cs')
                     in returnAndStoreInMemo gr'
    instantiate to (Inst from idMap tups) gr
      = let key  = (from, idMap)
            src  = construct key
            src' = fresh (sacred to) tups src
            dst  = src ∪ gr
            dst' = reduce dst
        in dst'
```

Figure 4.4: Overview of the inferencer

The last step results in a cardinality substitution for each cardinality variable occurring in types of the outermost binding group of the program. It does not directly give a substitution for another binding group, since the substitution is potentially different for each use of an identifier occurring in such a binding group. We can construct such a definition-site substitution from a use-site substitution. Suppose we know the cardinality substitution R for binding group k, and k contains the use-site of an identifier x defined in binding group j. We can construct the substitution S for the upper bound

and the substitution W for the lower bound from R by translating the guarantees that left-hand side $_J$ values of cardinality values of R give to upper and lower bounds. Consider an upper bound substitution S for example. Take S such that it maps to 0 for each cardinality variable of j, except for the cardinality variables mentioned by the *Inst* constraint. These we set to a value determined by the cardinality of the use-site counterpart in R. If the left-hand side cardinality value is 0, the upper bound is 0, and for 1^- and 1 the upper bound is 1, and otherwise $*$. By feeding S as initial substitution to the inferencer for the fixpoint computations on the constraint graph of j, we get a final substitution for the upper bound of each cardinality variable of j. This allows us to get a use-site dependent typing for a binding-group, which can be exploited by a code generator that performs code specialization.

4.7 Graph reduction

The performance of an implementation of the uniqueness type system is directly connected to the size of the uniqueness graphs. Without special treatment, these graphs explode in size. For example, consider the program:

let *inc1* = $\lambda x \to x + 1$
in **let** *inc4* = $\lambda x \to$ *inc1* (*inc1* (*inc1* (*inc1 x*))
in **let** *inc16* = $\lambda x \to$ *inc4* (*inc4* (*inc4* (*inc4 x*))
in *inc16* 4

With constraint duplication, the size of the graph corresponding to *inc16* is at least four times as big as the graph of *inc4*, and at least sixteen times as big as the graph of *inc1*. However, we can reduce this graph, because only the sacred variables matter. These are all use-site and definition-site cardinality variables in scope of the binding group. The other cardinality variables, for example, those taken freshly when instantiating a constraint graph, are intermediates. The vertices corresponding to sacred cardinality variables are sacred vertices. The other vertices are intermediates. We present a reduction strategy that removes all intermediates from the graph, and preserves the substitution that the graph represents.

The theoretical maximum number of constraints between these sacred vertices is only dependent on the number of sacred vertices, which are in turn only dependent on the size of the abstract syntax tree. Without reduction, the number of constraints depends on the amount of duplication, which can easily cause the number of constraints to explode, as the example shows. With the reduction of intermediate vertices, and n sacred vertices, the theoretical maximum number of normal edges is $O(n^2)$, but the theoretical number of hyper edges is still high: $O(n!)$. In practice, these numbers are not reached. The constraint graphs are sparse, with the number of edges proportional (on average) to the number of vertices. The reason is that the sacred annotations corresponds to values, and evaluation of some value depends typically on only a few other values. Furthermore, there are only a few hyper edges with many sources (those related to functions that are used a lot, such as prelude functions), but most identifiers in a program have only a handful of occurrences.

4.7.1 Preprocessing reductions

We present several relatively simple reduction strategies. It will turn out that these reduction strategies are insufficient. However, these strategies already reduce the number of paths in the graph, which makes the real reduction strategy (Section 4.8) performs better in practice.

Again, some cardinality variables may not be eliminated. The sacred cardinality variables will have counterparts in the destination graph during instantiation and should not be eliminated. The other variables are allowed to be eliminated, since their values can be reconstructed once we have a substitution for the sacred variables, by filling in the values in the original graph, and performing the fixpoint computation again, as we discussed earlier in this chapter.

A simple elimination strategy is splitting the constraint graph in connected components, and dropping those components without a sacred cardinality variable. These components cannot influence the substitution on sacred cardinality variables, since there is no path to them. This strategy is only effective when there are multiple, independent (mutual

exclusive), bindings in a binding group. A possible use for this strategy arises in the context of a module system, where bindings from another module can be considered to be in the same binding group.

Another elimination strategy is cycle reduction. Thus far, however, cycles in the constraint graph cannot occur, since each binding is non-recursive in this chapter. In Chapter 5, recursive **let**s are discussed, which can cause cycles in the constraint graph. A cycle in the constraint graph can be reduced to all sacred nodes in the cycle, or to an arbitrary node if there are no sacred nodes in the cycle. The edges of a singleton cycle, *self edges*, can be removed entirely.

There is an effective reduction strategy that can be applied in this chapter. Each non-sacred node u that is not connected by hyper edges, can be eliminated. For each pair of nodes a and b, such that there exists only an edge a to u, and only an edge from u to b, create an edge from a to b. Then remove u and the corresponding edges from the graph. Unfortunately, this strategy removes not all of the intermediate nodes; intermediate vertices with an in-degree greater than one are not reduced for example.

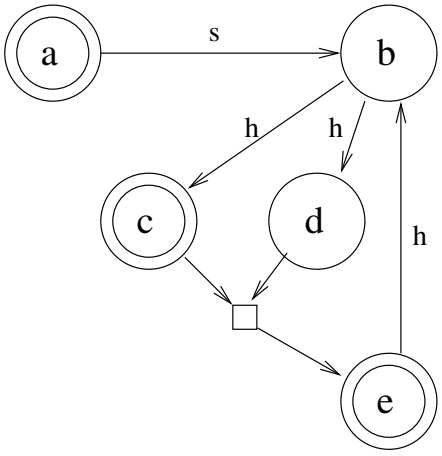
A final reduction strategy that we discuss makes use of upper and lower bound information. By applying the fixpoint substitution computations for the lower and upper bound on the constraint graph (while the graph is being constructed), we can get bounds for the cardinality variables. Take the upper bound values for example. Most of these values will still be 0, since these values depend largely on the usage of the binding group. But constraints from annotations of the user (Section 12.2), or ways to deal with a module system (Chapter 11), can give some information that allow $*$ to be inferred as an upper bound. If we also get a lower bound value of 0, we know that the two substitutions do not change anymore, and that the only possible cardinality annotation is $* \bowtie *$. Suppose that this is the case for vertex n. Then for each vertex m with an edge from m to n, we know that m also has a lower bound substitution of 0 and an upper bound substitution of $*$, and a corresponding cardinality value of $* \bowtie *$, because that is the result of one solve step on this the edge from m to n. In that case, we can remove the edge from m to n from the constraint graph. This approach disconnects nodes from the graph for which we already know that we cannot optimize the corresponding values.

All the reduction strategies discussed in this subsection have the property that they can be implemented in an efficient way. Unfortunately, not all intermediate vertices are removed. Therefore we need a more complicated approach, which we present in the next section (Section 4.8). Still, these reductions are important, because the performance of the reduction of the next section depends on the size of the graph that it gets as input.

4.8 Reduction of intermediates

Figure 4.4 mentioned the function *reduce*. We describe the implementation of this function in this section. This is an important function: it ensures that graphs do not explode due to constraint duplication.

We take the graph $G = (N_G, E_G)$ (encountered above) as example:



From this graph, we construct a reduced graph R that does not have the vertices b and d. We take a two step approach: first we create a reduced graph K that contains only normal edges, and then a reduced graph L that contains only hyper edges. The reduced graph R is the union of K and L.

We start with the reduced graph $K = (V_K, E_K, \emptyset)$. This reduced graph only contains the sacred vertices of G: $V_K = [v \in V_G \mid v \; sacred]$. The set of edges E_K is defined as follows:

$[(\delta_1, \delta_2, h) \in E_K \vee \delta_1 \; `hardPath` \; \delta_2]$
$[(\delta_1, \delta_2, s) \in E_K \vee \delta_1 \; `softPath` \; \delta_2 \wedge not \; (\delta_1 \; `hardPath` \; \delta_2)]$

There is a hard path between x and y ($x \not\equiv y$) if there is a simple path between x and y that:

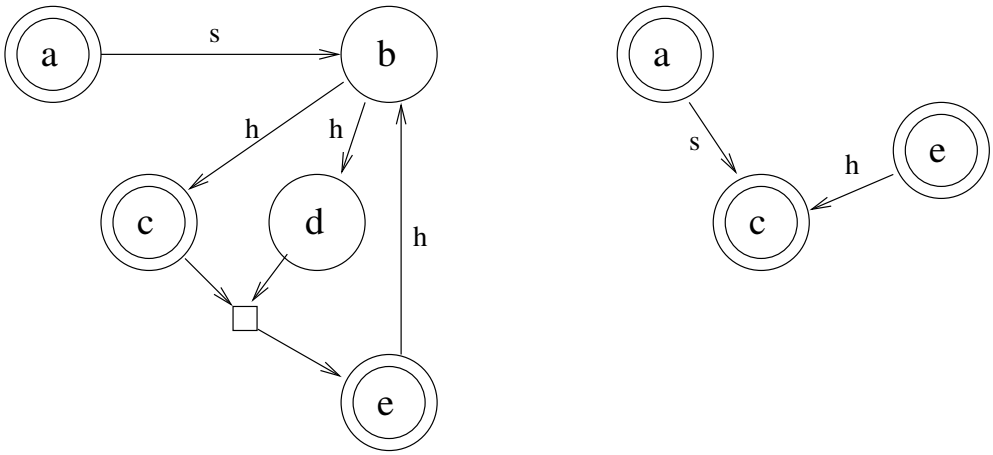
- Contains no sacred vertices except x and y.
- All edges are labeled with h.

Similarly, a soft path between x and y ($x \not\equiv y$) if there is a simple path between x and y that:

- Contains no sacred vertices except x and y.
- Has an edge labeled with s.

The idea behind these two definitions is that a soft edge is overruled by a hard edge.

When we look at the example, we see that there is only a soft path between a and c. There is a hard path between e and c (via b). The result is:

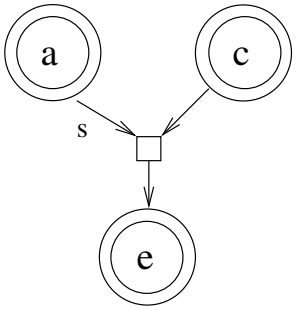


Now the graph $L = (V_L, \emptyset, H_L)$ containing only hyper edges. This reduced graph only contains the sacred vertices of G: $V_L = [v \in V_G \mid v \; sacred]$. The set of hyper edges H_L is defined as follows:

$H_L \; [\delta_1, ..., \delta_n] = (\delta, \eta)$, if:

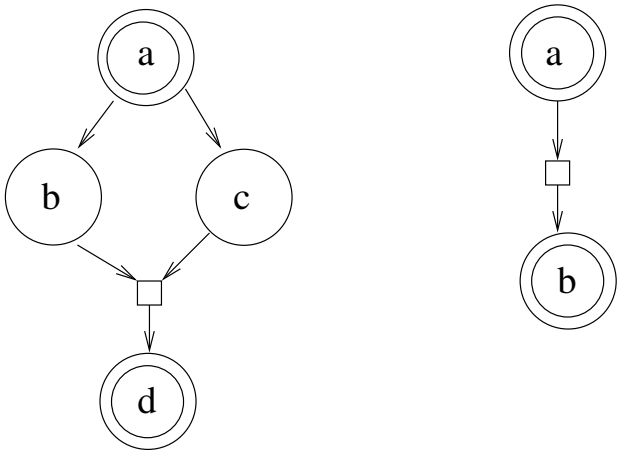
- δ is a sacred vertex.
- Each δ_i is a sacred vertex.
- There is a path P between δ_i and δ in G, not containing any other sacred vertices.
- P traverses a hyper edge (possibly more). The hyper edge HG is the hyper edge closest to δ.
- The value η is the value η taken from HG if all the vertices δ_i are the sources of HG. Otherwise η is the value $Just \; X$ where X is a constraint graph that describes how $\delta_1 \; ... \; \delta_n$ are combined into δ.

Before we talk about the graph X, we consider the example again. There is a path from both a end c to e traversing a hyper edge. We therefore expect L to be:



Unfortunately, there is a coercion (actually two) between a and the source vertex d on the hyper edge in G. The problem is that the hyper edge can force some value for the substitution of d which is relaxed by the coercion before it reaches a. If we blindly connect a to the hyper edge (as in the example), then the hyper edge can force the value directly on a, which is not the case with the original graph G. Or formulated in terms of constraints, there is a difference between $a \triangleright_s d, d \oplus c \leqslant e$ and $a \oplus c \leqslant e$.

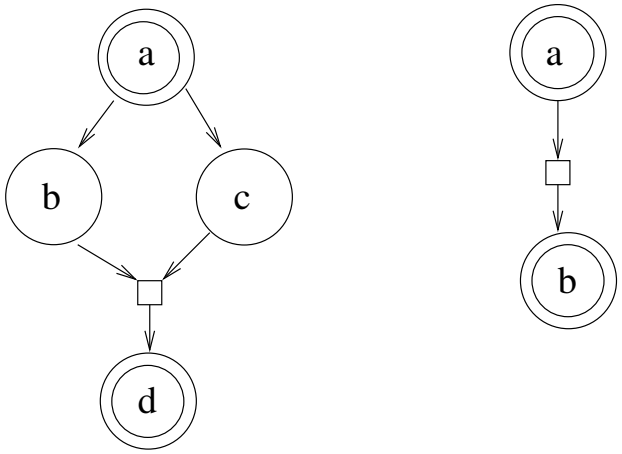
Even worse, consider a slightly different graph:



A reduction from the left graph to the right graph gives a constraint graph that results in inconsistent solutions! This is where $\eta = \mathit{Just}\ X$ enters the picture. We store a graph on a hyper edge (HL) in L that describes how the sources of HL are combined. This is basically a subgraph of G representing HL. When performing a solve step for HL, the inferencer either applies the solve step for an aggregation constraint ($\eta = \mathit{Nothing}$), or recurses on the graph X ($\eta = \mathit{Just}\ X$). In that case, the inferencer temporarily extends the substitution with initial values for the vertices in X that are not on HL.

We construct the constraint graph $X = (V_X, E_X)$ as follows. We first construct a graph X_1 which is a subgraph of G such that it contains only vertices and edges on simple paths from δ_i to δ that traverse hyper edge HG. In other words, it is just the part of G that is related to the hyper edge that we are inserting into L. The second step is that we clone X_1 to X_2 with a fresh variable on each vertex of X_2, except for the source and destination vertices of HL. Now comes the trick: we mark all source and destination vertices of hyper edges in X_2 sacred and recursively reduce X_2 to X. Since all sources and destinations of the hyper edges are marked as sacred, reduction of this graph does not recurse further, but creates a small graph that connects the sources of HL properly to the destination of HL.

Applying this approach on the running example of this section, gives us the following graphs for L and X:



Unfortunately, this reduction is insufficient. It saves us from an explosion of coercion constraints, but does not safe us from an explosion of hyper edges. Consider the example at the beginning of Section 4.7. The resulting graph has a hyper edge containing four hyper edges who each in turn contain four hyper edges. This problem can be solved by merging hyper edges in a constraint graph to a single hyper edge.

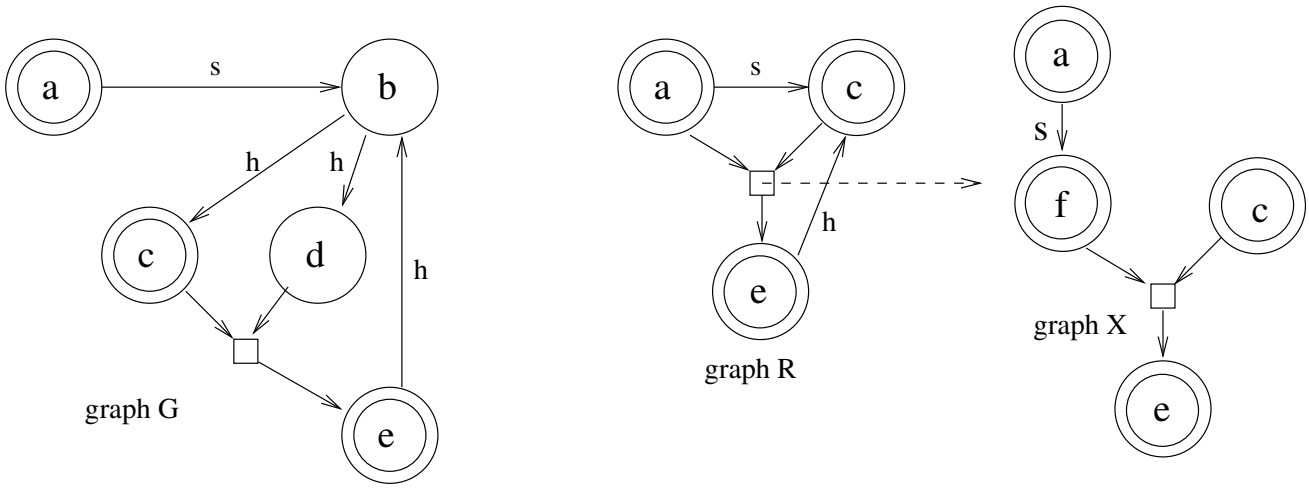
Merging hyper edges basically corresponds to simplifying an expression consisting of additions and maximums (or minimums), if we assume that coercions between destinations of a hyper edge and a source of another hyper edge are equalities (which is a conservative approximation). Expressions with additions and maximums can be rewritten to a canonical form, called *max-min-plus canonical form* [3]. This is achieved by applying the following rewrite rules:

$$max\ (\alpha,\beta) + \gamma \equiv max\ (\alpha + \gamma, \beta + \gamma)$$
$$max\ (\alpha,\beta) + max\ (\gamma,\delta) \equiv max\ (\alpha + \gamma, \alpha + \delta, \beta + \gamma, \beta + \delta)$$

Note that the rewrite rules for plus/max and max/min are equivalent, such that there is no difference between the graph for upper and lower bound.

This leads to a possible future work: we generated constraint graph in such a way that the edges correspond to constraints. But what if we store constraints as computations in the vertices, such that the edges connect inputs and outputs of these computations. A constraint graph is then a *Max-Min-Plus System* which we can reduce to a canonical form of a fixed size in the number of annotations. This should be possible for the upper and lower bounds, although it is not clear if this graph can be used for the cardinality propagation.

The last step is to take the union between K and L to obtain:



4.8.1 Remarks about graph reduction

The concept of sacred cardinality variables, or sacred vertices, is important. The number of sacred cardinality variables is a subset of all cardinality variables involved, especially where instantiations occur (which creates intermediates). The reduction strategies that we discussed, reduce the graphs such that the size is determined by the number of sacred vertices instead of the amount of constraint duplication (which would be a severe problem otherwise). This helps to improve the performance of the compiler on real-life programs. The number of sacred cardinality variables is a good indicator of the required processing time of the type system. To take this a step further, by controlling the number of sacred cardinality variables, a compiler could trade accuracy conservatively for compilation speed. For example, the use-site types can be replaced by definition-site types, reducing the number of cardinality annotations severely, but also reducing the quality of the inference for these types. In Chapter 8 we introduce data types, which leads to an explosion of sacred cardinality variables, and we discuss several ways to allow control over the number of cardinality variables, and thus to control the size of the constraint graphs, and the performance of the compiler.

4.9 Conclusion

The inferencing of cardinality annotations on types is rather indirect. The typing problem is translated into a constraint problem, which in turn is converted to a graph problem. This may seem to be overkill at first thought, but it allows a certain separation of concerns and abstraction, which is further exploited in subsequent chapters. However, in this chapter, we already use this separation to allow non-recursive and monomorphic **let**-bindings in the expression language.

The translation of the constraints to a graph representation allows another way of looking at the typing problem. Paths in the constraint graph tell how cardinality variables are related to each other. Furthermore, we are only interested in the cardinality values of a small subset of the nodes of the constraint graph. By looking at the paths between these special, sacred, nodes, we eliminate the other, intermediate, nodes, so that the resulting constraint graphs are small, which is important for the performance of the compiler.

In the next chapter, Chapter 5, the notion of a constraint graph will be exploited further, to allow polymorphic and recursive **let** bindings. The reduction work on graphs will become even more important, as the recursion will cause a fixpoint construction of a constraint set of a binding group, which would explode beyond imagination without the reduction work of this chapter.

Chapter 5

Recursion

In Chapter 4, we discussed the inferencing of cardinalities for a non-recursive **let**. We go one step further in this chapter, in order to support a (mutual) recursive Let. It is important to fully understand Chapter 4 before proceeding with this chapter, especially the part about the inferencer (Section 4.3 and Section 4.6), since we set up the type system by means of the *Inst* constraint in such a way that only the inferencer is affected to support recursion.

5.1 Example

Consider the following, incorrect[1], implementation of the faculty function:

let $fac_a :: Int \rightarrow Int = \lambda n \rightarrow n * fac_b\ (n - 1)$
in $fac\ 3$

So, what is the challenge of the recursion? The problem is that when gathering the constraint set of (the binding group of) fac_a, we encounter the variable fac_b, which results in an *Inst* constraint that refers to fac_a. So, the constraint graph of fac_a is needed to construct the constraint graph of fac_a, which is a problem, since it is not yet there.

This is a familiar problem. This problem already occurs with a conventional type system. Hindley-Milner type inference solves this problem by treating an identifier, that occurs recursively, monomorphically. That is also a solution to our problem. By giving up a polyvariant typing for an identifier occurring in an expression of its own binding group, we do not need the constraint graph when dealing with the corresponding *Inst* constraint (Section 5.2). A downside is that we loose some accuracy this way [8]. In a conventional type system, this problem can be solved when a programmer provides a polymorphic type signature [32]. Similarly, we can bypass this problem, when the programmer supplies a polyvariant constraint set (Section 5.3).

Contrary to the conventional typing problem, full polyvariant cardinality inference is possible in the presence of recursion. We perform Kleene-Mycroft Iteration [9], which amounts to a fixpoint iteration on the constraint graph, which assumes initially an empty constraint graph for a binding group and keeps expanding the graph, until a fixpoint is reached. We come to this in Section 5.4.

[1] We discuss how to make expressions conditional in Chapter 8 about data types

5.2 Monovariant recursion

A solution is to forget instantiation for recursive identifiers, and choose the definition-site annotated of an identifier as use-site annotated type instead of a freshly annotated type. For example, for the *fac* function:

$$\begin{array}{l}\textbf{let}\ fac :: Int^{\delta_2} \rightarrow^{\delta_1} Int^{\delta_3} = \lambda n \rightarrow n * (fac :: Int^{\delta_2} \rightarrow^{\delta_1} Int^{\delta_3})\ (n-1)\\ \textbf{in}\ \ fac :: Int^{\delta_5} \rightarrow^{\delta_4} Int^{\delta_6}\end{array}$$

In our implementation, we always use fresh identifiers for the use-site annotated type. An *Inst* constraint contains a mapping from annotations on the definition-site type to the (fresh) annotations on the use-site type. Then it is just a matter of equating the annotations on the definition-site type to those on each use-site type. So, with this approach we interpret *Inst* constraints belonging to recursive identifiers differently, and instead of inserting the constraint set of the corresponding binding group, we just insert the constraints $a \triangleright b$ and $b \triangleright a$ for each pair (a, b) mentioned in the *Inst* constraint. An *Inst* constraint corresponds to a recursive identifier if it occurs in a right-hand side expression of a binding-group.

To support recursion, only a minor change of the inferencer is required. But, in this approach, individual recursive identifiers are treated in exactly the same way. This gives suboptimal typings for some, perhaps contrived, programs [32], such as:

$$\begin{array}{l}\textbf{let}\ f = \lambda n \rightarrow \textbf{let}\ x = x\\ \qquad\qquad\ \ \textbf{in}\ f\ x + f\ (n-1)\\ \textbf{in}\ \ f\ 1\end{array}$$

What happens here is that the x is discovered to be shared. It is passed as argument to f, which makes the parameter of f shared. The n is also an argument of f, and since the parameter of f is considered to be shared, it makes n shared as well, although n can be unique.

This approach is the easiest way to deal with recursion and corresponds closely to how recursion is treated with conventional type inference.

5.3 Helping the inferencer with manual constraint sets

In Section 5.2, we by-passed the problem of needing a constraint graph while building it, by removing the requirement on a constraint graph. This approach has limitations, although it is questionable if these limitations are ever a problem in practice. However, we can consider what the impact on the type system is when circumventing this limitation.

Instead of dropping the requirement on constraint graph A while building A, we can change the requirement on A to some other graph B, which is at least as restrictive as A. This means that any valid substitution of B is also a valid substitution of A, such that it can replace A. When constructing the graph of A, the type system can just insert a copy of B where A is expected. This basically shifts the problem to another place, since we still need this graph B, but we come to that in a second.

How can we tell the type system to use another graph for a certain identifier? The *Inst* constraint helps us again. Suppose that for some identifier $f :: \widetilde{\tau}$, we have a constraint set S, where S does not contain any *Inst* constraints. We then create some new binding group n for S, and for each use $f :: \widetilde{\tau}_2$, generate an *Inst* constraint that refers to n and maps uniqueness variables of $\widetilde{\tau}$ to $\widetilde{\tau}_2$. Then the constraint graph obtained from S is used as replacement for the constraint graph of the binding group of f.

5.3.1 Programmer specified constraint sets

The constraint set is specified by the programmer. The programmer gives annotated types of the recursive identifiers in the binding group, and a constraint set. For example:

$$f :: a : Int \to (1,1) : Int, (1,1) \triangleright a$$

Each type constructor is prefixed with a cardinality variable chosen by the programmer, a concrete cardinality value, or nothing. A concrete cardinality value v is converted into a freshly invented variable δ, together with a special equality constraint $v \equiv \delta$. We can represent the $\equiv$ constraint internally with two $\triangleright$ constraints without coercions. Specifying no annotation has the same meaning as the concrete cardinality annotation $* \bowtie *$.

To construct the desired constraint set S, the type given by the programmer is extended with cardinality variables for the concrete or missing annotations, and the constraint set of the user updated accordingly. By overlaying the type of the programmer on the type inferred for the identifier, the cardinality variables in the constraints are renamed to the actual cardinality variables, instead of the symbols of the programmer. The result is S.

5.3.2 The problem with sacred cardinality variables

There is a problem with S. Binding groups are not independent. Sacred cardinality variables occur all over the place, and create links between binding groups. Unfortunately, the programmer cannot specify the effects on the sacred cardinality variables, since they do not have a name which the programmer can refer to. Conventional type signatures have this problem as well, as the following example shows:

$$\begin{aligned}&\textbf{let } f :: \forall\ \alpha.\alpha \to \alpha\\ &\quad f\ x = \textbf{let } y :: ???\\ &\qquad\qquad\ \ y = x\\ &\qquad\ \textbf{in } y\end{aligned}$$

In Haskell, we are unable to specify the type for y in this case. A solution is to give a name to the skolemnised version of type variable α, such that this name can be used in the signature of y. Unfortunately, this requires naming all sacred variables that are affected by the expression for which we want to specify the constraint set. I.e. this requires that we name the annotations on the type of each toplevel function that is used by the expression. That is infeasible, just because of the amount of annotations.

On the other hand, if we assume that we only specify replacement signatures for top-level functions, then we only have dependencies on annotations of other toplevel definitions. These annotations are on the spine of toplevel functions or values. Since the idea is that toplevel functions and values are shared, we can suffice by automatically enforcing a $* \bowtie *$ cardinality value for these annotations (by means of specifying some additional constraints during the constraint gathering process). This way, we assume that all toplevel values are fully shared and toplevel functions have a shared spine. As a result, it is impossible for any other definition to influence these annotations (they already have the highest value in the lattice). So, the programmer does not need to mention these annotations in the constraint set, which allows us to bypass the problem at the loss of precision for toplevel functions. This is not a very bad assumption to make, since from a design perspective, a function should only be toplevel if it is intended to be shared by multiple functions or modules. With this approach, constraint sets specified by the programmer are feasible.

5.3.3 Entailment check

Unfortunately, there is another problem: Suppose we have such an S. How do we know that the programmer specified the right constraint set? For example, how can we be sure that the constraint set of the programmer does not forget a constraint between some cardinality variables? For that, the compiler has to prove that S *entails* S', where S' is

the constraint set resulting from constraint gathering on the expression. This means that a substitution on the sacred uniqueness variables that is valid for S, is also valid for S'.

As a side note, the word *substitution* is a bit ambiguous here, since there are two solve phases. Normally, we talk about the final substitution, so the substitution resulting from the cardinality inference phase. In this case, however, it is enough that we can prove that S entails S', for substitutions resulting from the lower and upper bound inference phase, since if S' is more restrictive in terms of bounds than S, it also is more restrictive in the cardinality values (ignoring soft coercion constraints).

One attempt to verify this condition is to check the paths between sacred vertices. After reducing (Section 4.7) the graph S to R and S' to R', then S entails S' if each path in R is also a path in R' (assuming that hyper edges are reduced to canonical form). However, a constraint set specified by the user typically contains concrete annotations; these concrete annotations can hide a complete graph, such that the graphs are incomparable. This approach is not going to work.

One way to prove that the constraint set is valid, is to enumerate each combination of upper/lower bound values and check the implication $S \Rightarrow S'$, but this is not doable in practice. For propositional logic, there are many satisfiability checkers that can help us out here, but our final constraint set (which only has coercion and aggregation constraints) is an expression in a three-valued logic for the upper bound. We can rewrite the upper bounds to propositional logic, but that results into many propositions.

For each cardinality variable δ, we require two propositional symbols for the upper bound, P and Q:

δ	P	Q
0	0	0
1	0	1
*	1	0
-	1	1

With this scheme, we can construct truth-tables for the constraints, and construct a proposition from it:

p	q	$p \triangleright q$	p_1	p_2	q_1	q_2	
0	0	1	0	0	0	0	$\neg p_1 \wedge \neg p_2 \wedge \neg q_1 \wedge \neg q_2$
0	1	1	0	0	0	1	$\neg p_1 \wedge \neg p_2 \wedge \neg q_1 \wedge q_2$
0	*	1	0	0	1	0	$\neg p_1 \wedge \neg p_2 \wedge q_1 \wedge \neg q_2$
1	0	0	0	1	0	0	–
1	1	1	0	1	0	1	$\neg p_1 \wedge p_2 \wedge \neg q_1 \wedge q_2$
1	*	1	0	1	1	0	$\neg p_1 \wedge p_2 \wedge q_1 \wedge \neg q_2$
*	0	0	1	0	0	0	–
*	1	1	1	0	0	1	$p_1 \wedge \neg p_2 \wedge \neg q_1 \wedge q_2$
*	*	1	1	0	1	0	$p_1 \wedge \neg p_2 \wedge q_1 \wedge \neg q_2$

The propositional variant of the constraint is the disjunction of the above propositions.

Instead of translating an arbitrary aggregation constraint directly, we split it first up into multiple, but smaller, aggregation constraints. Since the $\odot$ is commutative, and the $\leqslant$ transitive, we can introduce some fresh cardinality variables, and shape the aggregation constraints into a tree form (see Section 7.1).

Each aggregation constraint has at most three cardinality variables this way. The translation scheme is then similar to the scheme for $\triangleright$, only three times as big.

After converting the entire constraint set into propositions, tools from propositional logic are used to verify that the proposition holds, or to come up with a counter example. Converting such a counter example into a piece of information understandable by the programmer is not trivial, and left as a subject for further study.

5.3.4 Final remarks about helping the inferencer

Helping the inferencer by specifying a replacement constraint set, gives some mixed feelings. Technically it is possible, but the effort it takes to get it implemented, is excessively huge, compared to the gains.

5.4 Full polyvariant recursion

There is a way to support full polyvariant recursion. The *Inst* constraints are a specification: is there a finite graph G, such that when a graph G' is constructed using G at the places of the corresponding *Inst* constraints, that G' is equivalent to G? The set of sacred annotations is finite, so according to the graph reduction section (Section 4.7, we know that any graph for these annotations can be reduced to a finite graph. A graph that always fits the specification, is the graph that has an edge between each pair of sacred annotations, and a hyper edge between each subset of sacred annotations, including with and without coercions. Of course, a smaller graph is desired.

To obtain a smallest graph that satisfies the specification, we perform a fixpoint computation on the constraint graph. Each binding group initially has an empty graph, and the insertion of graphs at the *Inst* constraints is replayed for each iteration. The graphs are reduced at the end of the iteration. This process proceeds until the graphs do not change anymore (see Figure 5.1). The graph reduction ensures convergence: hyper edges cannot grow indefinitely, and if one coercion is in the constraint set, then it will continue to be in the constraint set the next iteration, which guarantees that the number of coercions cannot grow indefinitely.

```
inferSubst constrSets
  =  let initial  = [((outermostBndgId, [ ]), emptygraph)]
         graphs   = iterate (λcurrent → foldr iteration current
                             . keys
                             $ topsortByInstDeps current) initial
         fixpnt   = snd.last.takeWhile (λ(g, g') → g ≢ g')
                             . zip graphs
                             $ tail graphs
     in  infer (outermost fixpnt)
  where
    iteration key grs
      = update grs key (construct key grs)
    construct key @(bndgId, idMap) grs
      = let cs     = bndgId `lookup` constrSets
            cs'    = rename idMap cs
            gr     = lookupWithDefault (csToGraph cs') key grs
            gr'    = foldr (instantiate grs bndgId) gr (insts cs')
        in  reduce gr' ∪ gr    -- ensure monotonicity
    instantiate grs to (Inst from idMap tups) gr
      = let key  = (from, idMap)
            src  = lookupWithDefault emptygraph key grs
            src' = fresh (sacred to) tups src
            dst  = src ∪ gr
        in  dst
```

Figure 5.1: Fixpoint iterating inferencer

This approach is less obvious than it appears. The constraint reduction is essential: without it, the constraint graph can grow forever. In fact, without reduction, the hyper edges increase in size with each iteration. But, graph reduction does not reduce to a canonical form. A result of this is that monotonicity is not guaranteed. We solve this by taking the union of the newly computed (and reduced) graph for the binding group (gr') with the previous (reduced) graph of

the binding group (gr). This graph has two important properties. It can only stay the same or grow between iterations (monotonically increasing). It has the same vertices as both gr and gr' (due to reduction). Only the number of edges can grow between iterations, and the number of edges is bounded (depending on the number of vertices). These two properties ensure that a fixpoint is reached.

Implementation of this procedure in the inferencer has only a slight impact. It does not influence the other operations of the inferencer much, and given functions that reduce a graph, and functions for dealing with the *Inst* constraint, implementing functionality as discussed in Chapter 4, this procedure is not very difficult to implement.

But there is a downside: even with the reduction strategies, the graphs for even toy programs are big. Fixpoint iterations on these graphs require at least the procedure to be performed twice, and depending on the nesting level and amount of recursion, several more times. For most programs, this fixpoint iteration process will be overkill. The question is whether this approach will be worth it.

We give this choice to the programmer. This choice is possible due to the *Inst* constraint. The programmer specifies for which binding groups a fixpoint strategy is required. The inferencer uses the implementation of this section as main inference strategy. All binding groups are put in the worklist in the earlier descried topologically sorted order (Section 4.3). When a recursive *Inst* constraint is encountered while processing a binding-group, then depending on the choice of the programmer, either the quick equational approach of Section 5.2 is taken, or the iterative approach (Section 5.4) by reinserting this binding group and the binding groups dependent on it, back in the work list. We can thus have best of both worlds, without a performance penalty if the fixpoint approach is never selected.

However, if we support the fixpoint approach, we pay a price: allowing full user-defined constraint sets (Section 5.3) becomes a problem. We can no longer verify the constraint set given by the programmer beforehand, since the full constraint set is only known at the use-site of an identifier. We can check that the constraint set is correct for each use of an identifier, just by checking if the substitution for the use-site is also a valid substitution for the constraints obtained by augmenting the programmer-supplied constraint set with the gathered constraint set. Unfortunately, there are many subtleties, due to dependencies on other binding-groups, and differences between use-site type and definition-site type. Perhaps most important, the problem is detected at a very late time, with probably no reasonable way to relate the error back to a location in the constraint set specified by the programmer. Letting the programmer specify additional constraints, is not a problem, however (Section 12.2).

5.5 Aliassing and cyclic definitions

A question that arises is: does our approach correctly deal with aliassing and cyclic definitions. The following example demonstrates a possible problem:

```
let x   = 1 + f 3
    f z = z + y
    y   = x
in f 1
```

The fact that x is a cyclic definition is not directly visible, since x only occurs one. Our analysis can detect this situation: the spine of f is discovered to be shared, because f is used in the **in** expression and in the body of x. As a result, y is discovered to be shared, and subsequently x is discovered to be shared.

If there is a cyclic definition, then there is at least an identifier v that occurs twice in the abstract syntax tree that causes the definition to be cyclic. Since our approach assumes that identifiers hidden by a function or hidden by an alias, have the same usage counts as the identifier that hides it, we in the end discover that the occurrences of v are individually used and that v itself is shared. Subsequently, this propagates to the aliassed and hidden identifiers, which ensures that any cyclic definition is discovered to be shared.

5.6 Conclusion

The *Inst* constraint plays an important role in dealing with recursion. At least two different ways of dealing with this constraint are possible. One way is fast, but less accurate. One way is accurate, but slow. The constraint creates an abstraction barrier between the two ways, allowing both to exist side by side. In practice, the fast approach is virtually always sufficient, but in case the programmer really needs the additional power, it is possible to turn on the slow approach for specific parts of the program.

This chapter also showed the need for constraint graphs and the corresponding reductions of Chapter 4. Without the reductions, the fixpoint iteration process is not possible for all but the tiniest programs. That said, this almost concludes the story about constraint graphs. Only Chapter 6 about polymorphism will mention constraint graphs again, since instantiation to a type with more structure requires another form of duplication of graphs.

Chapter 6

Polymorphic bindings

This chapter finishes the story about the **let** by explaining how to deal with polymorphism. At first glance, the addition of polymorphism does not seem to be a very involving change. In essence, we need to deal with annotated type variables in the type rules and that is all. Unfortunately, there is one big complication. A type variable can be instantiated by a function type, which has more than one type constructor, and consequently, more than one uniqueness annotation. Why this is a problem, and how to deal with it, is the central theme of this chapter.

6.1 Example

To demonstrate what the problem is, we show what the changes are to support polymorphism, until we run into the problem. For that, we need type variables, and some changes in the typing rules to support them.

The type language is extended with type variables. A type variable is annotated like a type constructor, and is a placeholder for a real annotated type. In the case of our language, this can only be an Int or a function.

We assume that we have the final types at our disposal, which means that each encountered type variable is universally quantified. Existentially quantified type variables are not allowed, we defer the discussion of them to Section 6.7. Applying these changes to the type language results in:

$$
\begin{array}{lll}
\widetilde{\tau} ::= \alpha^{\delta} & & \text{-- tyvar} \\
\quad | \;\; Int^{\delta} & & \text{-- tycon int (Int)} \\
\quad | \;\; (\rightarrow)^{\delta} & & \text{-- tycon arrow (Arrow)} \\
\quad | \;\; \widetilde{\tau}\,\widetilde{\tau} & & \text{-- type application (App)} \\
\quad | \;\; \forall\, \alpha.\widetilde{\tau} & & \text{-- universal quantification (silently ignored)}
\end{array}
$$

There are two places in the type rules that need to be changed. These are the flow type rules (Figure 3.10), and the type rule that generates the *Inst* constraint (Figure 4.1). Both have one and the same problem, so let us look only at instantiation.

Consider the following example:

$$
\begin{array}{l}
f :: Int^{\delta_2} \rightarrow^{\delta_1} Int^{\delta_3} \\
f = ... \\
id :: \forall\, \alpha.\alpha^{\delta_5} \rightarrow^{\delta_4} \alpha^{\delta_6} \\
id = \lambda x \rightarrow x \\
g :: Int^{\delta_8} \rightarrow^{\delta_7} Int^{\delta_9} \\
g = (id :: Int^{\delta_{12}} \rightarrow^{\delta_{11}} Int^{\delta_{13}} \rightarrow^{\delta_{10}} Int^{\delta_{15}} \rightarrow^{\delta_{14}} Int^{\delta_{16}})\, f
\end{array}
$$

Here we see that the use-site type of *id* has more annotations than the definition-site type of *id*. From the constraints of *id*, we get information about how δ_8 and δ_9 are related. With these, we can relate δ_{11} and δ_{14}. However, what to do with: δ_{12}, δ_{13}, δ_{15}, and δ_{16}?

Leaving these annotations unconstrained is not an option. Assuming that the argument and value of f are related, then the argument and value of g (which is identical to f), should also be related. How this is done is up to *id*, and thus there must be some constraints between δ_{12}, δ_{13}, δ_{15}, and δ_{16}, and the annotations of *id*. There are two ad-hoc options to get these constraints in place.

6.2 Ad-hoc strategy 1: defaulting

The first option uses a defaulting strategy. Suppose we instantiate some annotated type variable α^{δ_1} to $\widetilde{\tau}_a \rightarrow^{\delta_2} \widetilde{\tau}_b$. We generate additional constraints during instantiation that equate the uniqueness variables of $\widetilde{\tau}_a$ and $\widetilde{\tau}_b$ with δ_2. This effectively puts δ_2 on the type constructors of $\widetilde{\tau}_a$ and $\widetilde{\tau}_b$. The annotations are now properly connected, albeit in a crude way, resulting into poor typings when *id* is used.

In case of the above example, during instantiation the following additional constraints are generated:

$$\delta_{11} \triangleright \delta_{12}, \delta_{12} \triangleright \delta_{11}$$
$$\delta_{11} \triangleright \delta_{13}, \delta_{13} \triangleright \delta_{11}$$
$$\delta_{14} \triangleright \delta_{15}, \delta_{15} \triangleright \delta_{14}$$
$$\delta_{14} \triangleright \delta_{16}, \delta_{16} \triangleright \delta_{14}$$

Which just specifies that the occurrence of *id* in g has the annotated type:

$$g = (id :: Int^{\delta_{11}} \rightarrow^{\delta_{11}} Int^{\delta_{11}} \rightarrow^{\delta_{10}} Int^{\delta_{14}} \rightarrow^{\delta_{14}} Int^{\delta_{14}})\, f$$

6.3 Ad-hoc strategy 2: equality

The second option is less conservative. Assume that each type variable is instantiated to exactly the same annotated type. For example:

$$g = (id :: Int^{\delta_{12}} \rightarrow^{\delta_{11}} Int^{\delta_{13}} \rightarrow^{\delta_{10}} Int^{\delta_{12}} \rightarrow^{\delta_{11}} Int^{\delta_{13}})\, f$$

Again, this is implemented by generating additional constraints during instantiation.

This works, since the underlying type system will guarantee that each occurrence of the same type variable is instantiated to exactly the same type, thus we can always put the same annotations there. For the *id* function, this is a perfect solution, because now the uniqueness typing of g is equivalent to that of f. In fact, this approach works just fine for any function where a quantified type variable belongs to a single value.

However, in general, this approach is still too conservative as it ignores how the polymorphic function handles it's arguments, as is demonstrated by the following alternative version of the *const* function:

$$first :: \forall\ \alpha.\alpha \rightarrow \alpha \rightarrow \alpha$$
$$first = \lambda x\ y \rightarrow x$$

In this example, the type variable α is used for both the identifiers x and y. These identifiers potentially represent different values, but are treated the same with this approach, because they get the same annotations. The y parameter is not evaluated, but that will not be discovered as x and y have the same annotated type.

6.4 Strategy 3: graph duplication

We take an approach in this chapter that deals with the problem of polymorphism by investigating the constraint graph belonging to the binding group of a definition that is instantiated with a polymorphic type. A function with a polymorphic type cannot touch the contents of a value with a quantified type variable as type, because it does not know what the exact type is. However, it can pass the value to functions that do know the contents. For example:

```
fix :: (a → a) → (a → a → Bool) → a → a
fix f p v
  | p v v'       = v'
  | otherwise = fix f p v'
  where
    v' = f v
```

The function *fix* itself cannot touch v because it does not know what v is. But, depending on f and p, it does cause v to be use (and shared).

This duplication and passing around of values is captured in the constraint graph[1]. Each simple path between uniqueness annotations on a type variable (not necessarily the same type variable), gives information about the relation of cardinality information between the annotations on type variables. For the above example, the constraint graph explicitly indicates that v is duplicated and passed to f and p, just by looking at paths between vertices corresponding to the types of f, p, and v in the constraint graph.

However, this example is a bit too complicated for a more precise explanation. Therefore, consider $g = f\ id$. The cardinality information of the result of the *id* function is related (i.e. there is a path in the graph) to the argument. When *id* is passed the function f, then the result g is also a function. Now, there should be a relation between the annotations on the result of g to the result of f, and an information flow between the parameter of g and the parameter of f. Due to variance differences between argument and result, the relation between the arguments should be the opposite of the relation between the results. This creates one encapsulating chain of relations: from the value of g, to the value of f, from this value to the parameter of f and finally to the parameter of g.

Another example. Consider the *first* function again. There is a relation between the result of *first* and the first parameter of *first*. However, there is no relation with respect to the second parameter. So, if the type variable α is instantiated to some function type, then there should be a similar relation as in the above example: from the result of *first* to the function passed as first argument to *first*.

Formulating this in general is a bit tricky. Consider Figure 6.1 as illustration to the description that follows. Suppose there is an instantiation of a definition-site type to a use-site type. Suppose that the variables $(\alpha_1)^{\delta_1}, ..., (\alpha_n)^{\delta_n}$ are a subset of the annotated type variables of the definition-site type. Also suppose that the use-site type has annotated types $\widetilde{\tau}_1, ..., \widetilde{\tau}_n$ corresponding to these type variables. These types have an important property: their structure is exactly equal ($\tau_1 \equiv ... \equiv \tau_n$) because they are all the same instantiation of type variable α. Now, suppose there is a path P in the definition-site constraint graph between $\delta_1, ..., \delta_n$. We duplicate this path for each set of corresponding annotations in the use-site type (see Figure 6.1 for an example of a duplicated path). More precisely, let $\delta_{i,j}$ denote the jth annotation of τ_i. For each j, we duplicate the path P, taking $\delta_{i,j}$ for the annotation δ_i that occurred on $(\alpha_i)^{\delta_i}$). A small caveat: the path is reversed if the variance of $\delta_{i,j}$ differs from δ_i (note that the path represents constraints and constraints are directional depending on variance).

With this approach, instantiation is correctly dealt with in the presence of polymorphism, but there is a price due to constraint duplication. This price is not high: after graph reduction, the constraints will only be between sacred uniqueness variables, which are typically not many for the "deeper" variables.

[1] For simplicity, we assume that there are no hyper edges in the constraint graph.

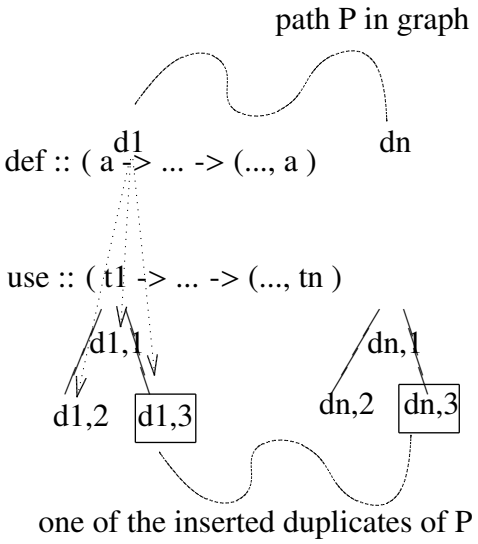


Figure 6.1: Path duplication illustration

6.5 Other complications

The problem with instantiation shows up at other places as well, e.g. in the presence of impredicativity. Consider the following example:

let $id :: \forall\ \alpha.\alpha \to \alpha$
  $f :: ((Int \to Int) \to (Int \to Int)) \to Int$
  $f = \lambda g \to g\ id\ 3$
in $f\ id$

A problem occurs in the presence of impredicativity in the application of *id* to f. Constraint generation of coercion constraints dictates that we generate coercion constraints between cardinality annotations on the type of the parameter of f and the cardinality annotations on the type of *id*. But the type of the parameter of f has a deeper structure than that of the *id* function. The same strategies that we discussed above can be applied in this case.

In practice, this means that the position of an annotation in a type and constraint is starting to play an important role. This role gets even more important in Chapter 8 about data types. In the type system, dealing with position information of an annotation shows up at two places; during constraint gathering and during inferencing. It complicates the implementation of both parts of the type system. In the remainder of this chapter, we factor this subject out of both parts of the type system and give it its own place. This results in a better separation of functionality, easier understanding of the implementation, more code reuse, and not to forget, allows us to vary the strategy that deals with the "deeper" annotations.

6.6 Types in constraints

The trick is to put annotated types in the constraints, instead of a single annotation:

$$\begin{aligned} constr ::= &\ \delta \rhd_{h/s} \delta \\ \mid &\ \overline{\tau} \rhd_{h/s} \overline{\tau} \end{aligned}$$

$$\mid \ \sum_i \widetilde{\tau}_i \leqslant \widetilde{\tau}$$
$$\mid \ \mathit{Inst}\ [(n,n)]\ n\ \widetilde{\tau}\ \widetilde{\tau}$$

The coercion constraint on single cardinality variables is still present/allowed, in case a particular annotation needs to be constrained.

This simplifies the constraint gathering process, as there is no longer a need to recurse into the annotated type and to keep track of variance. For example, the type rule of Figure 3.10 is no longer required, and is replaced by the generation of a single constraint.

For the inferencer, the abstraction comes in the form of *annotation trees*. This is a representation that clearly shows the recursive structure of annotations on a type:

```
data AnnTree
  = AnnNode Annotation [AnnTree] [Annotation]
  | ...    -- some other constructors that are only there for technical purposes
```

Each type constructor corresponds to an *AnnNode*, and the parameters of a type constructor are the children of the type constructor. The list of annotations is empty in this chapter, but in Chapter 8 this list will represent additional annotations on a type constructor. For example, the type $(,)^{\delta_1}\ \mathit{Int}^{\delta_2}\ \mathit{Int}^{\delta_3}$ is represented as:

$$\mathit{AnnNode}\ \delta_1\ [\mathit{AnnNode}\ \delta_2\ [\,]\ [\,]]\ [\mathit{AnnNode}\ \delta_3\ [\,]\ [\,]]$$

The idea is that we simultaneously convert a set of types to annotation trees, one annotation tree for each type, with the property that all annotation trees have exactly the same shape, assuming that the types are unifiable, which is guaranteed by the type inference process[2]. If one type $\widetilde{\tau}_1$ has an annotation δ that has no counter part in some other type $\widetilde{\tau}_2$, then $\widetilde{\tau}_1$ has a deeper structure at this place than $\widetilde{\tau}_2$. We fill up the gaps in $\widetilde{\tau}_2$ by duplicating an annotation δ_2 in $\widetilde{\tau}_2$. δ_2 is the deepest annotation in $\widetilde{\tau}_2$, such that its counter part in $\widetilde{\tau}_1$ is a parent of δ. This constructs the annotation trees by means of the defaulting strategy of Section 6.2. Note that we only use this for constructing the annotation trees. Which parts of the annotation tree are used, depends on the actual instantiation strategy of the inferencer. With the concept of annotation trees, we merely make sure that the inferencer only needs to deal with types that have an identical shape.

The gap between the constraint gathering process and the inferencer is filled by the procedure that constructs the annotation trees, and several environments that capture properties of annotations. Among these properties is the variance of an annotation, and whether or not it is attached to a type variable. Changes to the type language that we made in order to support polymorphism are covered by this procedure. The only task left to the inferencer is to perform the constraint duplication, which is now considerably easier with the annotation trees and some annotation environments as input.

6.7 Existential types

Since we support universal quantification, the question arises what we have to do for existential quantification. For our implementation, it is a similar problem. Universal quantification has as problem that the use-site may have more structure than the definition site. Existential quantification has as problem that the definition-site may have more structure than the use-site. Unfortunately, there is a difference: with universal quantification, it is the definition-site binding-group that needs to be augmented with additional paths. For existential quantification, it is the use-site binding-group needs to be augmented with paths. We can only do that as a post-processing phase after all *Inst* constraints have been dealt with (otherwise the vertices are not there yet). For universal quantification, we can already do this when inserting the constraint graph of the definition site. So, basically the difference between dealing with universal quantification and existential quantification is in which constraint graph is consulted for additional paths.

[2]Not all types in an aggregation constraint are unifiable, but the annotation trees are lazily constructed and the inferencer only examines only the unifiable parts of the annotation trees for a type correct program.

6.8 Conclusion

Dealing with polymorphism is interesting. Up front, one would expect that it has severe consequences on the uniqueness type system, especially in the presence of higher-ranked annotated types. But with the right abstractions in place, these changes are rather isolated.

The work we did to put the types into the constraints will be useful in Chapter 8. Dealing with data types is essentially the problem of additional annotations, and will cause some additional annotations to appear in the annotation trees. To support data types, we do not need to change anything beyond the annotation trees. This means that with this chapter, we close the subjects about the inferencer and constraint graphs.

Chapter 7

Parallel execution paths

In the languages covered up to this chapter, there is no syntactic way to write that an expression is conditionally evaluated. However, conditional evaluation is already possible, for example, by using Church booleans [33]:

$$
\begin{array}{ll}
\mathit{true} & = \lambda t\ f \to t \\
\mathit{false} & = \lambda t\ f \to f \\
\mathit{ite} & = \lambda g\ t\ e \to g\ t\ e \\
\mathit{example} & = \mathit{ite}\ \mathit{true}\ 1\ \bot \\
 & \quad \text{-- the result: example = 1}
\end{array}
$$

Some occurrences of an identifier are sequential. If one occurrence is referenced on an execution path, then the other is referenced as well if they are sequential. An example of this is the expression $\lambda x \to x + x$. Both occurrences of x are always on the same execution path. This is not the case for the expression $\lambda x \to \mathit{ite}\ \mathit{true}\ x\ x$. In this expression, the two occurrences are mutually exclusive or parallel. They never occur simultaneously on an execution path. And sometimes, no guarantees can be made: $\lambda g\ x \to \mathit{ite}\ g\ x\ 1 + x$. In this case, it depends entirely on g if the two occurrences of x will be always be on the same execution path, or not.

If occurrences are sequential, we can add up the lower bounds instead of taking the maximum. If occurrences are parallel, we can take the maximum of the upper bounds instead of adding them up. This gives a better approximation of the result, but requires us to discover which occurrences are parallel and which are sequential.

Analyzing function applications to determine whether the operators are guaranteed to be sequential, or guaranteed parallel, is tricky. We discuss this subject in Section 7.3. Instead of analyzing the function applications the hard way, we take a look at syntactic constructs that guarantee that some occurrences of an identifier are sequential or parallel. For example, in practice, most splits in an execution path are explicitly indicated by alternatives for a function definition, branches of a **case** expression, or the **then** and **else** branch of an **if** . . **then** . . **else** expression. Since **if** . . **then** . . **else** expressions capture the essence of this section and are a prelude to **case** expressions of Chapter 8, we introduce **if** . . **then** . . **else** expression in this chapter (Section 7.1).

7.1 Adding an if-then-else expression

The addition of an **if** . . **then** . . **else** expression is discussed in this section. Such an expression guarantees syntactically that occurrences on a **then** branch are parallel to occurrences on an **else** branch. It also guarantees that an occurrence of the guard is sequential with occurrences in the branches. Unfortunately, the bad news that we (in general) cannot determine the lower bound of the branches. If we know that the entire expression is evaluated at least once, we know

that one of the branches is evaluated at least once, but we do not know which one. So, we are conservative and assume that the lower bound of the branches is 0.

The following type rule specifies what the analysis does when it encounters an **if** . . **then** . . **else** nonterminal:

$$\frac{\begin{array}{c}\Gamma \vdash^{unq} e : \widetilde{\tau}_e \rightsquigarrow \zeta_e, \omega_e \\ \Gamma \vdash^{unq} g : Bool^{\delta} \rightsquigarrow \zeta_g, \omega_g \\ \Gamma \vdash^{unq} t : \widetilde{\tau}_t \rightsquigarrow \zeta_t, \omega_t \\ \tau \cong \tau_e \\ \tau \cong \tau_t \\ \widetilde{\tau} \equiv \tau^{\delta}{}_e \\ \zeta_{ec} \equiv \{\widetilde{\tau} \triangleright_q \widetilde{\tau}_e\} \\ \zeta_{tc} \equiv \{\widetilde{\tau} \triangleright_q \widetilde{\tau}_t\} \\ \zeta_{gc} \equiv \{\delta_e \triangleright_s \delta\}\end{array}}{\Gamma \vdash^{unq} \textbf{let } g \textbf{ then } t \textbf{ else } e : \widetilde{\tau} \rightsquigarrow \zeta_{gc}\zeta_{tc}\zeta_{ec}\zeta_g\zeta_t\zeta_e, \omega_g\omega_t\omega_e}\ \text{E.ITE}_{UF}$$

The guard is evaluated as often as the expression is evaluated, which we specify by a soft coercion between the outermost annotation of the result type of the expression and the annotation on the *Bool* type of the guard. The least upper bound of the **then** and **else** branch is at most the upper bound of the entire expression. The lower bound of the **then** and **else** branch depends on which branch is taken. We do not know this in general, so we set these lower bounds to 0, with a special coercion constraint $\triangleright_q$, which is interpreted as follows:

$$\begin{array}{l}\textit{satisfied } (a,b,c) \triangleright_q (d,e,f) \\ \quad = 0 \sqsubseteq_L d \quad \text{-- force lower bound to 0} \\ \quad \wedge\ b \sqsubseteq_C e \\ \quad \wedge\ c \sqsubseteq_U f\end{array}$$

So, an **if** . . **then** . . **else** expression gives us some information about upper bounds, at the expense of lower bounds for branches. We take a closer at this particular problem in section 7.2. We are now going to use the information that the **if** . . **then** . . **else** expression provides to improve the aggregation constraints.

The syntax of an **if** . . **then** . . **else** expression guarantees that occurrences of an identifier on a **then** branch are parallel to identifiers occurring on an **else** branch. There is also a syntactic guarantee that occurrences in the guard are sequential to both the **then** and the **else** branch. There is no other syntax that guarantees that two occurrences are sequential to each other. For example, in the case of $f\ x\ x$, it depends on how f treats its parameters in order to know if the two occurrences of x are parallel or not. But, we can already improve the solution with only the knowledge that some occurrences of identifiers are necessarily parallel, and some are necessarily sequential.

The aggregation constraint consists of a sequence of annotated types of which the aggregated result needs to be smaller than some annotated type. Since $\odot$ is commutative, we can change the order of the aggregation. For example, $a \odot b \odot c \odot d \leqslant e$ can be written as $((a \odot b) \odot c) \odot d \leqslant e$, or as $(a \odot b) \odot (c \odot d) \leqslant e$. Our improved approach uses this property and we change the structure of the aggregation constraint accordingly:

$$\begin{array}{lll}\textit{sum\_constr} ::= \textit{sum\_expr} \leqslant \textit{uty} & & \\ \textit{sum\_expr} ::= \textit{uty} & & \\ \qquad\qquad \mid\ \textit{sum\_expr} \oplus \textit{sum\_expr} & \text{-- sequential only} \\ \qquad\qquad \mid\ \textit{sum\_expr} \odot \textit{sum\_expr} & \text{-- sequential or parallel} \\ \qquad\qquad \mid\ \textit{sum\_expr} \ominus \textit{sum\_expr} & \text{-- parallel only}\end{array}$$

The difference with the previous structure of the aggregation constraint is that the structure of the aggregation is not a list anymore, but a tree, where the internal nodes are aggregations by means of one of the three aggregation operators. The annotated types of the occurrences form the leafs of the tree. The representation in the graph is still a hyper edge, where the leafs are cardinality variables of the corresponding types in the constraint, and the computation is attached as label to the hyper edge.

After conversion to graph form and during the inference of a substitution, the operators form a computation on lower and upper bound values. The operators are defined as follows:

$$
\begin{aligned}
\mathit{lower}\ (a \ominus b) &= \mathit{lower}\ a\ \text{`}\mathit{max}\text{`}\ \mathit{lower}\ b \\
\mathit{lower}\ (a \odot b) &= \mathit{lower}\ a\ \text{`}\mathit{max}\text{`}\ \mathit{lower}\ b \\
\mathit{lower}\ (a \oplus b) &= \mathit{lower}\ a + \mathit{lower}\ b \\
\mathit{lower}\ \delta &= \delta\ \text{`}\mathit{lookup}\text{`}\ \mathit{subst} \\
\mathit{upper}\ (a \ominus b) &= \mathit{upper}\ a\ \text{`}\mathit{max}\text{`}\ \mathit{upper}\ b \\
\mathit{upper}\ (a \odot b) &= \mathit{upper}\ a + \mathit{upper}\ b \\
\mathit{upper}\ (a \oplus b) &= \mathit{upper}\ a + \mathit{upper}\ b \\
\mathit{upper}\ \delta &= \delta\ \text{`}\mathit{lookup}\text{`}\ \mathit{subst}
\end{aligned}
$$

The operator $\odot$ can be used anywhere, but $\oplus$ and $\ominus$ are less conservative versions in case we know that some occurrences are parallel or sequential.

The idea is that we construct a computation from the structure of an expression (Figure 7.1). An occurrence of occurrence $x :: \widetilde{\tau}$ is turned into the computation $\widetilde{\tau}$. If we have such a computation for the function expression and argument expression of a function, we combine these two computations with the $\odot$ operator, since it depends on the function whether or not they are sequential or parallel, and our approach does not have this knowledge. For an **if** . . **then** . . **else** expression, the computations of the two branches are parallel, and are combined by the $\ominus$ operator. The $\oplus$ operator is used between the guard and the branches, since it is guaranteed that identifier occurrences of the guard are on the same execution path as identifier occurrences in the branches. There are some additional rules in case there is no occurrence in a subexpression. We omit the case for the **let** in the above type rules, but assume that occurrences in expressions of a **let** are combined by means of the $\odot$ operator.

$$\boxed{\hbar \vdash^{\odot} e \rightsquigarrow \mathfrak{I}}$$

$$\frac{}{x \vdash^{\odot} n : \widetilde{\tau} \rightsquigarrow \widetilde{\tau}}\ \text{E.VAR}_B \qquad \frac{\begin{array}{c} x \vdash^{\odot} a \rightsquigarrow c_a \\ x \vdash^{\odot} f \rightsquigarrow c_f \end{array}}{x \vdash^{\odot} f\ a \rightsquigarrow c_f \odot c_a}\ \text{E.APP}_B \qquad \frac{x \vdash^{\odot} f \rightsquigarrow c_f}{x \vdash^{\odot} f\ a \rightsquigarrow c_f}\ \text{E.APP.1}_B \qquad \frac{x \vdash^{\odot} a \rightsquigarrow c_a}{x \vdash^{\odot} f\ a \rightsquigarrow c_a}\ \text{E.APP.2}_B$$

$$\frac{\begin{array}{c} x \vdash^{\odot} g \rightsquigarrow c_g \\ x \vdash^{\odot} l \rightsquigarrow c_l \\ x \vdash^{\odot} r \rightsquigarrow c_r \end{array}}{x \vdash^{\odot} \textbf{if}\ g\ \textbf{then}\ l\ \textbf{else}\ r \rightsquigarrow c_g \oplus (c_l \ominus c_r)}\ \text{E.ITE}_B \qquad \frac{\begin{array}{c} x \vdash^{\odot} l \rightsquigarrow c_l \\ x \vdash^{\odot} r \rightsquigarrow c_r \end{array}}{x \vdash^{\odot} \textbf{if}\ g\ \textbf{then}\ l\ \textbf{else}\ r \rightsquigarrow c_l \ominus c_r}\ \text{E.ITE.1}_B$$

$$\frac{x \vdash^{\odot} g \rightsquigarrow c_g}{x \vdash^{\odot} \textbf{if}\ g\ \textbf{then}\ l\ \textbf{else}\ r \rightsquigarrow c_g}\ \text{E.ITE.2}_B \qquad \frac{\begin{array}{c} x \vdash^{\odot} g \rightsquigarrow c_g \\ x \vdash^{\odot} l \rightsquigarrow c_l \end{array}}{x \vdash^{\odot} \textbf{if}\ g\ \textbf{then}\ l\ \textbf{else}\ r \rightsquigarrow c_g \oplus c_l}\ \text{E.ITE.3}_B$$

$$\frac{\begin{array}{c} x \vdash^{\odot} g \rightsquigarrow c_g \\ x \vdash^{\odot} r \rightsquigarrow c_r \end{array}}{x \vdash^{\odot} \textbf{if}\ g\ \textbf{then}\ l\ \textbf{else}\ r \rightsquigarrow c_g \oplus c_r}\ \text{E.ITE.4}_B \qquad \frac{x \vdash^{\odot} l \rightsquigarrow c}{x \vdash^{\odot} \textbf{if}\ g\ \textbf{then}\ l\ \textbf{else}\ r \rightsquigarrow c}\ \text{E.ITE.5}_B$$

$$\frac{x \vdash^{\odot} r \rightsquigarrow c}{x \vdash^{\odot} \textbf{if}\ g\ \textbf{then}\ l\ \textbf{else}\ r \rightsquigarrow c}\ \text{E.ITE.6}_B$$

Figure 7.1: Aggregation type rule (B)

In the case for a lambda and a **let**, we generate the aggregation constraint for an identifier $x :: \widetilde{\tau}_{def}$ by taking the computation $\mathfrak{I}$ and turn it into the constraint $\mathfrak{I} \leqslant \widetilde{\tau}_{def}$.

This approach allows us to improve the upper bounds of identifiers that occur only parallel. The approach does not help much with the lower bound of identifiers (see next section). However, in general the approximation of the bounds in presence of multiple execution paths, is improved. Perhaps as important, is that it is a fairly easy improvement, since we know exactly from the structure of the abstract syntax tree where the execution paths are split up.

7.2 Lower bounds and if-then-else

The **if** . . **then** . . **else** is a problem for the lower bound of the branches, because we do not know which branch is taken. In the general case with a **case**-expression, it is even possible that none of the branches is taken. We can say something about all branches of an **if** . . **then** . . **else** (or **case**) expression). Suppose that some identifier x occurs (indirectly) on each branch and has a lower bound of 1 if each branch has a lower bound of 1. This means that the value represented by x is used if a branch is evaluated. Every branch evaluates x, so it does not matter which branch is taken, thus the lower bound of x is at least the lower bound of the entire expression.

One way to detect this situation is by performing a strictness analysis on the **if** . . **then** . . **else** or case expression (Section 3.8). If we know that some δ_1 in the definition-site type of x is used if some δ_2 in the type of the expression is used, then we can add δ_2 to the aggregation of δ_1 (i.e. $(...) \odot \delta_2 \leqslant \delta_1$ instead of $(...) \leqslant \delta_1$). However, this can depend on functions that are passed to a function where the **if** . . **then** . . **else** appears is, so that we cannot perform the analysis beforehand.

Alternatively, we can detect this situation in the constraint graph. The above description translates informally as follows: if there is a path from an annotation on a type of the branch to an annotation of the type of an occurrence of x in that branch, and all branches have such a path, then we can use the lower bound of the corresponding annotation on the result of the **if** . . **then** . . **else** expression in the aggregation of the lower bound of the annotation on the identifier.

More formally, the implementation is as follows. We add the following information to the aggregation constraint at the definition-site of an identifier:

BranchCombine
  Ty -- definition-site annotated type of the identifier
  [*Ty*] -- result annotated types of each branch
  Ty -- result annotated type of the entire if-then-else or case expression

Suppose the definition-site annotated type is D, the annotated types of each branch are T_1 ... T_n, and the annotated type of the result of the if-then-else is R. Take an annotation δ from D. Subsequently, take an annotation δ_r from R. Assuming that the types of the branches and the result of the expression have the same structure, take annotations $\delta_1 ... \delta_n$ respectively from T_1 ... T_n, such that each δ_i corresponds to δ_r. Now, if there is a path from each δ_i to δ in the constraint graph, then the lower bound of δ is allowed to be δ_r. If there is a hyper edge on such a path, then all leaves of the hyper edge must be on such a path as well.

We only have to check for this path once. This can be done just before graph reduction. If there is a path, we change the aggregation constraint such that it includes those δ_r. The only problem is that when the graph is constructed using a fixpoint iteration on the graph, that the path may not exist yet (for example, due to recursion in one of the branches).

7.3 Function application and sequential occurrences of an identifier

There is still a deficiency in combination with function applications. Arguments to a function application are usually sequential, except for functions that make an exclusive choice between a parameter a and a parameter b, depending on some other condition or parameter. An example is the function *ite* given in the introduction of this chapter. The arguments passed to a and b are unlikely to share occurrences of the same identifier, because one can ask himself why

one would make a choice then. So, we would like to assume that occurrences of an identifier appearing as arguments to a function, are sequential to each other.

Assuming that the relation between occurrences is sequential when it is actually parallel, has only minor influence on the upper bound. We are a bit too conservative in this case, but it rarely results in much worse upper bounds in practice. Unfortunately, this story does not hold for the lower bound. If we assume that a relation is sequential when it is parallel, then we add lower bounds when we are only allowed to take the maximum value and we can give a wrong estimation for the lower bound. Thus, we can only assume that the relation between occurrences is sequential for the lower bound if it is guaranteed to be the case.

This is rather unfortunate, since most of the occurrences of an identifier on function arguments will be sequential, but we cannot make use of this fact for the lower bound. Unless we can prove that the occurrences are sequential. Unfortunately, this is difficult to prove. Consider for example:

let $f\ g\ t\ e =$ **if** g **then** t **else** e
in $\lambda x \rightarrow f$ (*...complicated expression...*) $x\ x$

The two occurrences of x are parallel to each other, although they appear to be sequential. Determining whether or not the occurrences of x are sequential is a difficult problem. The function f can be a function that puts the arguments into a list and passes it to some function that extracts the second element if the first element is greater than ten. This makes it difficult to capture the relations between occurrences of a variable.

However, there is an elegant way to discover from most of the perhaps-sequential occurrences that they are in fact sequential. Consider the following expression:

$f\ x_1\ x_2$

If we know that both x_1 and x_2 are used at least once, then the relation between the two is sequential. Why? A lower bound of 1 means that the value represented by x_1 is used at least once on each execution path. The same for x_2. So, they occur both on every execution path, and thus occur together on each execution path.

This means that the $\odot$ can be replaced by the $\oplus$ operator if the lower bound of both arguments is at least 1. Now compare the definition of $\odot$ and $\oplus$. The only difference is in the computation of the lower bound. Lower bound values are special in the sense that it is addition up to at most 1, which results into the following properties:

$1 + 1 = 1 = 1$ `` `max` `` 1
$1 + 0 = 1 = 1$ `` `max` `` 0

There is no distinction between the three aggregation operators due to the limited possibilities for the lower-bound usage annotation! There is no such deficiency.

However, this is only the case because a lower bound of 1 is the highest allowed lower bound. If we generalize our approach by allowing higher lower bounds, then the above property does not hold anymore. Then there is a difference between the $\oplus$ operator and the $\odot$ operator for the lower bound. In such a situation, the property that two occurrences are sequential if they both have a lower bound of at least 1, does still hold and we can replace the $\odot$ operator with the $\oplus$ operator to improve the approximation in this case. Although this fact is not useful for our analysis, it is important enough to keep it in mind for possible generalizations of our approach.

7.4 Conclusion

Parallel execution paths pose interesting problems. It puts the difference between upper and lower bounds into perspective. Properly dealing with the existence of parallel execution paths require us to determine which occurrences of an

identifier are parallel and which occurrences are sequential.

The usage analysis of Clean (called labeling or marking) is related to our approach in this chapter. Like Clean, we make a distinction between the **then** and **else** branch of an **if** . . **then** . . **else** expression. Unlike Clean, our analysis can detect that some identifiers are not used, which has consequences for the aggregation result.

However, the two approaches are difficult to compare. The usage analysis of Clean is described in term graphs, where we describe it in terms of abstract syntax trees. Clean allows the use of a strict **let**, which makes evaluation order explicit. With an explicit evaluation order, it is possible to distinguish read and write access. If it is guaranteed that a value is accessed for writing only once, and all other operations on the value are read accesses that occur before the write access, then the value can still be updated in place. This distinction cannot be easily made in Haskell since the evaluation order is implicit and difficult to analyse, as evaluation order is determined by (lazy) pattern matches.

Chapter 8

Adding algebraic data types

We almost have an industry-ready language from a typing point of view. The major type system feature that we lack is that of algebraic data types. Most of the machinery to support data types is already in place; only support for pattern matches is missing.

Support for data types involves changes at both the expression level and the type level. The expression level needs to deal with pattern matches and case-expressions. At the type level, the types become more complicated. Types have more annotations, dictated by the types in an algebraic data type definition. Although algebraic data types influence a lot of components of the inferencer, the required changes are surprisingly isolated. The constraint gathering phase needs to deal with case expressions and pattern matches. The solver is unaffected. The only place that is heavily affected, is the conversion to annotation trees (Section 6.6). In order to perform this conversion, it turns out that we have to perform two analyses on the data type definitions (Chapter 9).

This chapter is organised as follows. We start with an example that illustrates what the problems are to support algebraic data types. Each of these problems is discussed in a subsequent section in isolation. The contents of these sections require a fair understanding of the previous chapters.

8.1 Example

The example data type that we take in this chapter, are lists:

$$\begin{aligned}&\textbf{data}\ \mathit{List}\ a = \mathit{Cons}\ a\ (\mathit{List}\ a)\\ &\qquad\qquad\ \ |\ \mathit{Nil}\end{aligned}$$

The algebraic data type definition dictates how the representation of a *List Int* in memory is (see Figure 3.3). In this case, a list is a region of memory representing a *Cons* or a *Nil*. In the first case, the area of memory has two pointers, one to a value of type a, and a pointer to the remainder of the list. Each area of memory has a corresponding type in the algebraic data type definition. The other way around, each type in a algebraic data type definition, corresponds to zero or more memory regions of a value of that data type. Similar to Section 3.4, annotations on the type constructors of the data type, classify the uniqueness properties of a piece of memory. This example also shows why annotations on the types are useful: a value can be infinite, but the representation of the type is finite.

With the annotation problem taken care of, the existing infrastructure knows how to deal with expressions that have user-defined data types as types. But, in order to construct and deconstruct such values we need a few alterations in the expression language, by means of value constructors, pattern matching, and case-expressions. Value constructors are

just functions with a type derived from the corresponding algebraic data type definition. We already know how to deal with those. But pattern matches and case-expressions deserve some attention.

The annotations are important in the case of pattern matching. Pattern matching touches the spine of a value, and if it succeeds, can bring identifiers into scope. Constraints needs to be generated in order to properly flow use-counts into the annotations on the spine of data types. Annotations for the types of the identifiers introduced by the pattern match, need to be obtained from the type of the pattern and the corresponding algebraic data type definition (Section 8.4). For example, if some value has annotated type $\widetilde{\tau}^{\delta_0}$, where $\tau = \mathit{List\ Int}$, and we perform a successful pattern match on $\mathit{Cons}\ (x :: {\tau_1}^{\delta_3}\ (xs :: {\tau_2}^{\delta_4})$, then each use of x and xs can potentially result in a use of the spine of the value towards the location where x or xs is stored, so we generate the constraints: $\delta_3 \triangleright_s \delta_0$ and $\delta_4 \triangleright_s \delta_0$. For the strict parts in the pattern, in this case the *Cons* constructor alone (with δ_0 being the corresponding annotation), we generate a constraint $\delta \triangleright_s \delta_0$, where δ is the number of uses of the expression containing the pattern match.

Case expressions are a generalisation of the **if** . . **then** . . **else** expressions of Chapter 7. We are not going to explain this subject again here: the generalisation is straightforward after we know how to deal with pattern matches.

8.2 Exposed annotations

Data types complicate the annotation process. Consider the following algebraic data type definition:

data *Tuple* = *T Int Int*

Only the *Tuple* type constructor gets an annotation δ. But what are the annotations to give to both integers? There are several ways to deal with this situation. A straightforward way is to annotate each *Int* with δ. That way we treat the components of a data type the same as the data type itself. Although possible, this does not provide the accuracy we aim for. A second way is to ask the programmer for the annotations on the components of the type, and assume that the annotation is $* \bowtie *$ if that is not the case. This is the approach taken by Clean [2]. The third way is to expose the annotations on inner types such that the use-site of a data type can choose the annotations [39]:

$$\textbf{data}\ \mathit{Tuple}_{[\delta_1,\delta_2]}\ (\delta)) = T\ \mathit{Int}^{\delta_1}_{[]}\ \mathit{Int}^{\delta_2}_{[]}$$

This way, the use-site can choose if the *Tuple* contains two unique integers, two strict integers, or perhaps a unique integer and an arbitrarily used integer. The list subscripted on a type constructor is called the set of exposed annotations. This list contains all annotations occurring inside the algebraic data type definition.

When referencing a type constructor from another binding group, the exposed annotations are instantiated to fresh versions:

$$\textbf{data}\ \mathit{IntWrap}^{\delta_1}_{[\delta_2]} = \mathit{Wrap\ Int}^{\delta_2}_{[]}$$
$$\textbf{data}\ \mathit{IntTup}^{\delta_3}_{[\delta_4,\delta_5,\delta_6,\delta_7]} = \mathit{Tup\ IntWrap}^{\delta_4}_{[\delta_5]}\ \mathit{IntWrap}^{\delta_6}_{[\delta_7]}$$

This approach is slightly complicated in case of (mutual) recursive data types, since what list of annotations should we pass to the recursive type constructor? This is a familiar problem, for which there is a common solution: in a first pass, pick fresh annotations for type constructors from another binding group, and take the empty list for annotations from the same binding group. Collect all the exposed annotations for each binding-group. In the second pass, set the exposed annotations for type constructors of the same binding-group to the list of the entire binding-group. Now, each annotation occurring inside the algebraic data type definition, occurs in the list of exposed annotations of the corresponding type constructor.

For example:

$$\textbf{data}\ \mathit{InfIntList}^{\delta}_{[\delta_1,\delta_2]} = \mathit{Cell\ Int}^{\delta_1}_{[]}\ \mathit{InfIntList}^{\delta_2}_{[\delta_1,\delta_2]}$$

But what about named type variables in a algebraic data type definition? The problem with type variables is that the structure of a type variable can be substituted to virtually any structure at the use-site of a data type, and we cannot give the annotations beforehand. But that is not required either: the annotated types passed as arguments to a type constructor, contain exactly the annotations that we did not know yet. For example:

data $Embed^{\delta}_{[\,]}\ a = Emb\ a$

The annotations to choose for a come from the use-site. For example, the type $Embed^{\delta_1}_{[\,]}\ Int^{\delta_2}_{[\,]}$ specifies that a value of this type has δ_1 as annotation on the Emb constructor, and δ_2 on the integer that it contains.

8.2.1 Implementation

After obtaining the set of annotations for each data type, the implementation of this approach is straightforward after the work that we did for annotation trees in Section 6.6. When annotating a type constructor in a type, pick fresh annotations for the exposed annotations, and when processing a constraints with these type in them, construct an annotation tree with the list of annotations as attachment on the corresponding node in the annotation tree. The treatment of exposed annotations in the annotation tree is the same as the other annotations in the annotation trees. There is a complication, however, since we need to know the variance of annotations and the fact if they are below a function arrow or not. These two subjects are more involved in the presence of data types. Chapter 9 provides a way to obtain the required information about the location of annotations.

The described approach is flexible and the changes are fairly isolated. There is a downside to this approach, which is that types can contain a lot of annotations this way. We can control this number of annotations by giving up accuracy, for example by enforcing that the annotations of a certain data type are never taken freshly. This saves annotations when the data type is used a lot, but has as consequence that there is only one uniqueness variant of the data type. All values of such a data type then have the same cardinality values, which is less flexible.

8.3 Expanded types

The exposed annotations approach has limitations. If we look at a list, then each *Cons* in the list is treated the same way. We cannot make a distinction between certain elements of a list. On the other hand, the reason for taking a list (which is of varying size), is to treat each element in the same way, so that is not a big issue. A bigger limitation is that a type variable can only be used in one fashion. For example:

data *Tuple a = T a a*

In this example, both occurrences of a will get the same cardinality annotation, since we take the annotations from the type passed as a to *Tuple*. But in the memory representation of such a data type, each a is a separate area and thus can have different cardinalities. So, can we change the system such that we can treat occurrences of the same type variable differently?

We can achieve this by using an ad-hoc trick: by unfolding the algebraic data type definitions a couple of times. For example, unfolding *Tuple Int* results into the type *<T Int Int>*, and can be annotated with annotations in the conventional way. The value-constructor name does not need an annotation, but is part of the type in order to determine of which constructor the fields are an expansion. Type expansion is dangerous for recursive types, since the expansion process will not terminate, and the representation can explode quickly, especially for nested data types.

The expansion approach and the exposed-annotations approach can be combined. The exposed-annotations approach can be used on any type, and the expansion approach on any type of kind *star*. By default, due to the dangerous nature of expansions, we do not perform expansions and require the programmer to specify when an expansion is allowed. We start with an expanded annotated type. Assume that $\widetilde{\tau}$ is a type annotated by the exposed-annotations approach.

Consider that during a topdown visit a type $\widetilde{\tau}_0$ is encountered, with outermost type constructor T and parameters $\widetilde{\tau}_1$, ..., $\widetilde{\tau}_n$. If $\widetilde{\tau}_0$ is of kind *star* and the occurrence of T in $\widetilde{\tau}$ is allowed for expansion, then the definition of T is unfolded with annotations taken from $\widetilde{\tau}_i$ at each occurrence of type variable i. This procedure thus works by performing one-step expansions. In the end, we replace each individual variable by a fresh one.

For example, consider the algebraic data type definition of *List*:

$$\begin{aligned}&\textbf{data } List^{\delta_1}_{[\delta_2]}\ a = Cons\ a\ (List^{\delta_2}_{[\delta_2]}\ a)\\ &\qquad\qquad\ \ |\ Nil\end{aligned}$$

Suppose that we annotate the type *List Int*. By means of exposed annotations, we obtain the type:

$$List^{\delta_3}_{[\delta_4]}\ Int^{\delta_5}_{[\,]}$$

A one step expansion gives:

$$<Cons\ Int^{\delta_5}_{[\,]}\ (List^{\delta_4}_{[\delta_4]}\ Int^{\delta_5}_{[\,]}), Nil>^{\delta_3}$$

Note how the exposed annotation δ_4 is used on its corresponding occurrence on the data type during the expansion. Expanding yet another step further, gives:

$$<Cons\ Int^{\delta_5}_{[\,]}\ <Cons\ Int^{\delta_5}_{[\,]}\ (List^{\delta_4}_{[\delta_4]}\ Int^{\delta_5}_{[\,]}), Nil>^{\delta_4}, Nil>^{\delta_3}$$

We now end up with a type that is just an expanded version of the exposed annotated type. This type reflects how exposed annotations are mapped to the structure of an algebraic data type. We use this type for pattern matches, because we can easily 'overlay' it over constructors. For freshly annotating a type we go one step further and replace each individual annotation variable with a fresh one:

$$<Cons\ Int^{\delta_6}_{[\,]}\ <Cons\ Int^{\delta_7}_{[\,]}\ (List^{\delta_9}_{[\delta_8]}\ Int^{\delta_{10}}_{[\,]}), Nil>^{\delta_{11}}, Nil>^{\delta_{12}}$$

This type can be used as expanded annotated type of *List Int*. Note that this type allows more precise results than the exposed annotated type, because the expanded annotated type allows the first two constructors to be annotated differently than the remainder of the list.

These one-step expansions are not only used for annotating the data types, but also for expanding a type for a certain constructor in pattern matches, and in constructing an equivalent annotation tree structure if one type has exposed type, and the other an expanded type. So, in a way, the infrastructure that is needed to support expanded types, is already needed for the system itself.

So, how does a programmer specify when a type may be expanded? We use *path expressions*, which is a sequence of constructor names with wildcards. A type constructor T is allowed to be expanded, when T occurs in some expanded type, which as parents $T_1, ..., T_n$, and there is a prefix of a constructor sequence equal to $T_1, ..., T_n, T$. It must explicitly mention T, but it is allowed to have a wildcard for some of the constructor names.

A path expression is defined by:

$$\begin{aligned}&path ::= T\ path \quad \text{-- constructor}\\ &\qquad |\ *\ T\ path \quad \text{-- wildcard}\\ &\qquad |\ empty\end{aligned}$$

A wildcard stands for any number of constructors not equal to the constructor after the wildcard. For example, the sequence $*List * List * List$ specifies that a list may be unfolded thrice.

With expanded types, two things can be accomplished: type constructors of kind $*$ can get different annotations for each occurrence in the body of a data type, and the head of a data type can be annotated differently than the (possibly infinite) tail. This power comes at a price: the types grow and so does the number of annotations. Therefore, expanded types are turned off by default, only to be used in special circumstances, specified by the programmer using path expressions.

8.4 Pattern matches

Pattern matches bring identifiers into scope, and force evaluation of a certain part of a value to determine if a pattern matches or not. We assume that we have only pattern matches in a **case** expression, not in left-hand sides of functions.

The structure of a pattern match dictates how to pattern match on a type to obtain the type of the subpatterns. An expanded type (Section 8.3) can directly be overlayed on top of a pattern match. An exposed-annotated type can be overlayed on a pattern match after a one-step expansion for each pattern match on a constructor. For example, if we take the one-step expansion of Section 8.3 as example, then a match *Cons x xs*, with the type of the entire pattern match being $<Cons\ Int^{\delta_5}_{[]}\ (List^{\delta_4}_{[\delta_4]}\ Int^{\delta_5}_{[]}), Nil>^{\delta_3}$, gives $Int^{\delta_5}_{[]}$ for *x* and $(List^{\delta_4}_{[\delta_4]}\ Int^{\delta_5}_{[]})$ for *xs*. So, for pattern matches, the annotated type is pushed inwards, expanding it if needed for each match on a constructor. This ensures that we have a definition-site type for every part of a pattern.

We now discuss which coercion and aggregation constraints are generated for a pattern match. Take the following pattern as example:

$$Cons\ x\ (Cons\ 3\ y\ @(Cons\ \_\sim(Cons\ z\ zs)))$$

This pattern shows some possibilities for a pattern, such as aliasing (y@...) and irrefutable patterns (∼(*Cons*...)) and an underscore (_).

If a pattern has a topmost constructor (in our case the *Cons* on the left), then this constructor is special: it has the same annotation as the outermost annotation of the type of the pattern. There is guaranteed to be an evaluation of a value corresponding to the match if all patterns in branches above this constructor have topmost constructors. These constructors have one thing in common: the corresponding type constructor (and thus the annotation) are equal. So the case expression will always lead to a weak-head normal form evaluation of the value which we match. Thus we can safely set the annotation corresponding to such a special constructor to the upper and lower bound of the enclosing expression.

For the non-special parts of the pattern, we proceed as follows. The lower bound of each annotation is forced to 0, except for function arguments or parameters, since we do not have guarantees of evaluation. The upper bound depends on the specific part of a pattern.

Suppose that the pattern is a constructor. A constructor is irrefutable if it is inside a irrefutable pattern (∼), otherwise it is refutable. For a refutable pattern we force the upper bound to at least the upper bound of the enclosing expression, since such a match only needs to be performed at most once for each evaluation of the **case** expression. This is not the case for an irrefutable pattern. For an irrefutable pattern, we can force the upper bound to at least the minimum of the upper bound of the enclosing expression, and the maximum of the upper bounds of all the variables inside the same irrefutable pattern. The reason is that the irrefutable pattern is evaluated at most as often as the enclosing expression, but can be less if none of the identifiers it brings in scope are used. If there is an alias connected to the constructor, then we generate an aggregation constraint between the annotation of the constructor and the corresponding annotation on the use-sites of such an alias.

Suppose that the pattern is a variable or underscore. We force, by means of an aggregation constraint, the upper bound to at least the upper bound of the sum of the corresponding upper bounds from all aliases and the aggregation result of the individual uses of the variable. For the above example, we create an aggregation constraint for all uses of z and combine this aggregation with the annotations from the corresponding part of the definition site of y.

Pattern matches are not really complicated, but present some technical details due to forced evaluation and patterns that introduce aliasing.

8.5 Case expressions

Constraint gathering for case expressions is fairly straightforward now we know how to deal with pattern matches. For the pattern matches and the subexpressions, we apply the constraint gathering rules as discussed before. A freshly annotated type $\widetilde{\tau}$ is generated for the result type τ of the case-expression. Each branch i, with annotated type $\widetilde{\tau}_i$, is coerced to the result type of the case-expressions by generating the constraint $\widetilde{\tau} \triangleright_q \widetilde{\tau}_i$. We discussed this particular subject in the previous chapter when we dealt with **if** . . **then** . . **else** expressions.

8.6 β annotations and constructors

What about the β annotations that were mentioned in Chapter 3? These β annotations are basically constraint set variables, and are required for higher-ranked uniqueness types. A constraint set can be associated with each parameter to a constructor, so with each field of a data type.

We therefore annotate each data type with a β annotation for each field in one of the constructors. So, a data type with n fields gets annotations $\beta_1 \ldots \beta_n$. These β annotations participate with the general unification process (just assume they are additional phantom type variables).

From the data type definition, we derive an annotated signature from the data type definition, using the conventional approach to get a signature from a data type definition, except that we take the annotated types of the fields.

Consider the *List* data type. The algebraic data type definition of a *List* has two fields, so the *List* has two β annotations. It also has one exposed annotation δ. This gives us the following annotated algebraic data type declaration (in a slightly different notation):

$$
\begin{array}{l}
\forall\, \beta_1\ \beta_2. \\
\quad \forall\, \delta_1\ \delta_2. \\
\quad\quad \forall\, \alpha. \\
\quad\quad\quad \textbf{data}\ List^{\delta_1}_{[\delta_1,\delta_2]\ [\beta_1,\beta_2]}\ \alpha = Cons\ \alpha_{\beta_1}\ (List^{\delta_2}_{[\delta_1,\delta_2]}\ \alpha)_{\beta_2} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid\ Nil
\end{array}
$$

From this definition, we generate the signature for *Cons* in the obvious way:

$$
\begin{array}{l}
\forall\, \beta_1\ \beta_2. \\
\quad \forall\, \delta_1\ \delta_2. \\
\quad\quad \forall\, \alpha. \\
\quad\quad\quad Cons :: \alpha \rightarrow_{\beta_1} (List^{\delta_2}_{[\delta_1,\delta_2]\ [\beta_1,\beta_2]}\ \alpha \rightarrow_{\beta_2} (List^{\delta_1}_{[\delta_1,\delta_2]\ [\beta_1,\beta_2]}\ \alpha)
\end{array}
$$

This allows us to use constructors as arbitrary functions. Also note the interaction with the β annotations. We can put functions into lists, but the β annotations will ensure that we still know which constraints belong to a function:

$$xs = (Cons\ f\ (Cons\ g\ (Cons\ h\ Nil)))$$

Since we assume that β annotations are part of the underlying type system, unifications on β annotations take place when constructing the list, so that β_1 is the same for each *Cons*. This means that the constraint sets of f, g, and h are merged into a single constraint set. When pattern matching on a function in this *List*, we do not know which of the three

functions it is. But the constraint set that we get is the merged version of the three constraint sets, which is at least as restrictive than any of the individual functions, thus correct.

8.7 Conclusion

Algebraic data types are an interesting language feature in terms of what complications they give in order to support them. Good abstractions are required in the implementation, otherwise code soon becomes so interconnected that it is no longer feasible to grasp. Our approach with constraint gathering, constraint graph construction with annotation trees, and constraint solving, provides ways of splitting concerns in relatively small pieces.

An important aspect of the approach taken in this chapter is that the part that is concerned with the analysis itself (constraint solving), is almost not affected. This means that algebraic data types do not require us to change our way of reasoning about uniqueness typing, and only complicate the analysis on a technical level, instead of a conceptual level. Perhaps the work on algebraic data types can be reused for other analyses as well.

One aspect that we did not cover in this chapter, is how we determine the variance of an annotation, and whether or not the values it classifies are parameters or values of a function. We discuss this in Chapter 9.

Chapter 9

Polarity and Under-the-arrow analysis

Chapter 8 introduces data types. For annotations on the type constructors of these data types, we need to know what their variance is and if they occur as function parameter or value. In this chapter we discuss how to infer this information from the data type definitions and the types in the program. For that, we reuse the entire analysis that we explained in this thesis, and apply it to the type language instead of the expression language. Notice that this chapter is not about uniqueness typing, but about a whole other analysis!

9.1 Introduction

In this section we introduce the concepts of covariance and below-the-function-arrow on simple types without data types.

An annotation is *covariant* if it only occurs on type constructors that are covariant. Likewise, an annotation is *contra-variant* if it only occurs on type constructors that are contra-variant. When we do not know the difference, then it is called *variant*. Without algebraic data types, a type is either an *Int*, or the function type $\tau \to \tau$. Suppose that we also have tuples (τ, τ). We then define the variance as follows:

$$
\begin{aligned}
&\mathit{variance}\ t = \mathit{var}\ t\ \oplus \\
&\mathit{var}\ \mathit{Int}^{\delta}\ c \qquad\quad = [(\delta, c)] \\
&\mathit{var}\ (t_1 \to^{\delta} t_2)\ c = [(\delta, c)] +\!\!+ \mathit{var}\ t_1\ (\mathit{neg}\ c) +\!\!+ \mathit{var}\ t_2\ c \\
&\mathit{var}\ (t_1\ ,^{\delta}\ t_2)\ c \quad = [(\delta, c)] +\!\!+ \mathit{var}\ t_1\ c +\!\!+ \mathit{var}\ t_2\ c \\
&\mathit{neg}\ \oplus = \ominus \\
&\mathit{neg}\ \ominus = \oplus
\end{aligned}
$$

Effectively, this definition means that the variance of a function argument is the inverse of a function result. Apply the *variance* function to $((\mathit{Int}^{\delta_1} \to^{\delta_4} \mathit{Int}^{\delta_2}) \to^{\delta_5} \mathit{Int}^{\delta_3})$ to discover that δ_3 and δ_5 are covariant, that δ_2 and δ_4 are contra variant (because of the negation), and that δ_1 is covariant again.

We give a similar definition for below-the-function arrow. An annotation is *below-the-function-arrow* if it only occurs on type constructors that are below-the-function-arrow. Likewise, an annotation is *not-below-the-function-arrow* if it only occurs on type constructors that are not-below-the-function-arrow. If it is neither, then we assume that it is both. Again, we can give a definition in terms of types without algebraic data types:

$$
\begin{aligned}
&\mathit{below}\ t = \mathit{bel}\ t\ \mathit{NotBelow} \\
&\mathit{bel}\ \mathit{Int}^{\delta}\ c \qquad = [(\delta, c)]
\end{aligned}
$$

$$bel\ (t_1 \rightarrow^{\delta} t_2)\ c = [(\delta, c)] +\!\!+ bel\ t_1\ Below +\!\!+ bel\ t_2\ Below$$
$$bel\ (t_1\ ,^{\delta}\ t_2)\ c \quad = [(\delta, c)] +\!\!+ bel\ t_1\ c +\!\!+ bel\ t_2\ c$$

For the example $((Int^{\delta_1} \rightarrow^{\delta_4} Int^{\delta_2}) \rightarrow^{\delta_5} Int^{\delta_3})$, this definition means that $\delta_.$ (1), δ_2, δ_3, and δ_4 are below the function arrow, and δ_5 is not.

But, these are just definitions for a type language without algebraic data types. Algebraic data types complicate these concepts because it depends on the definition of a data type what is done with the type arguments of a type constructor. We demonstrate this by an example in the following section.

9.2 Example

Suppose we have a slight alteration of the commonly known *GRose* data type, which we will call *FRose*:

$$\textbf{data}\ FRose^{\delta_1}_{\delta_2}\ f\ a = FBranch\ (f\ a\ (FRose^{\delta_1}_{\delta_1}\ f\ a))$$
$$FRose :: \forall\ a.(a \rightarrow * \rightarrow *) \rightarrow a \rightarrow *$$

And consider the following types:

$$FRose^{\delta_3}_{\delta_4}\ (,)^{\delta_5}\ Int^{\delta_6}$$
$$FRose^{\delta_7}_{\delta_8}\ (\rightarrow)^{\delta_9}\ Int^{\delta_{10}}$$

We now ask ourself the following questions:

- What is the variance of δ_6 and δ_{10}?
- If the type will be expanded indefinitely, will δ_6 and δ_{10} always occur below a $(\rightarrow)$? If that is the case, then δ_{10} always occurs on type constructors of types of values that are function arguments or results.

An analysis on the types is required to answer these questions. For δ_6 we see that the type definition represents an right-infinitely deep tuple:

$$(Int^{\delta_6}, (Int^{\delta_6}, (Int^{\delta_6}, (Int^{\delta_6}, (Int^{\delta_6}, ...)))))$$

Applying the variance rules given in Section 9.1 results in δ_6 in being covariant and not below a function arrow.

The inverse is case for δ_{10}. The type with δ_{10} represents the type of a function with infinitely many parameters:

$$(Int^{\delta_{10}} \rightarrow (Int^{\delta_{10}} \rightarrow (Int^{\delta_{10}} \rightarrow (Int^{\delta_{10}} \rightarrow (Int^{\delta_{10}} \rightarrow ...)))))$$

Applying the techniques of Section 9.1 again, we see that for the variance, each δ_6 is a multiple of two negations away from an other δ_6, and that if there would be a final δ_6, that it is the result of a function, thus all occurrences of δ_6 are covariant. As is visible in this example, all occurrences of δ_6 are below a function arrow.

How are we going to infer these results? What we see here is that the analyses work by pushing some known information topdown into a definition. This is exactly what we did with the propagation of the upper bounds. Similarly, combining the results of each individual use-site to one single definition-site, is what we did with the aggregation constraint. So, by performing the analysis on the type level, instead of the expression level, on kinds instead of types, we can use the results of previous chapters to solve the problem of this chapter. For that, we only need different constraint gathering rules (although they look a lot like those for uniqueness typing) and a different interpretation in the solver.

9.3 Constraint gathering

Constraint gathering is performed on the type level. A type expression does not differ much compared to a normal expression, except that there are no lambda abstractions. Instead of lambda abstractions, there are algebraic data type definitions in the declarations of a Let.

The type of a type is called a kind, denoted by κ (see Section 2.2.2). We allow a kind to be annotated, similarly to a type. An annotated kind is denoted by $\widetilde{\kappa}$, following the same conventions as for types in Section 3.4.

Figure 9.1 lists the typing rules for constraint gathering on type expressions. We omit the cases for extensible records and expanded types, these cases clutter up the type rules, but do nothing more than collect the constraints for the type expressions occurring in them.

$$\boxed{\Gamma \vdash^{kind} \tau : \widetilde{\kappa} \rightsquigarrow \zeta}$$

$$\frac{\begin{array}{c}\vdash^{annot} \widetilde{\kappa_i} \rightsquigarrow \widetilde{\kappa} \\ \widetilde{\kappa_i};\widetilde{\kappa};pairs \vdash^{genInst} \beta \rightsquigarrow \zeta \\ \kappa_i \rightsquigarrow \kappa;pairs \\ i_\beta \mapsto \widetilde{\kappa_i} \in \Gamma\end{array}}{\Gamma \vdash^{kind} i : \widetilde{\kappa} \rightsquigarrow \zeta}\ \text{D.VAR}_{DC} \qquad \frac{\begin{array}{c}\kappa_i \rightsquigarrow \kappa;pairs \\ \widetilde{\kappa_i};\widetilde{\kappa};pairs \vdash^{genInst} \beta \rightsquigarrow \zeta \\ \vdash^{annot} \widetilde{\kappa_i} \rightsquigarrow \widetilde{\kappa} \\ i_\beta \mapsto \widetilde{\kappa_i} \in \Gamma\end{array}}{\Gamma \vdash^{kind} i_n : \widetilde{\kappa} \rightsquigarrow \zeta}\ \text{D.CON}_{DC} \qquad \frac{\begin{array}{c}\Gamma \vdash^{kind} \tau : \widetilde{\kappa} \rightsquigarrow \zeta \\ \widetilde{\kappa} \equiv \kappa^{\delta}{}_{top} \\ \zeta_{ann} \equiv \{\delta_{top} \triangleright \delta\}\end{array}}{\Gamma \vdash^{kind} \tau^{\delta} : \widetilde{\kappa} \rightsquigarrow \zeta_{ann}\zeta}\ \text{D.ANN}_{DC}$$

$$\frac{\begin{array}{c}\Gamma \vdash^{kind} a : \widetilde{\kappa_a} \rightsquigarrow \zeta_a \\ \Gamma \vdash^{kind} f : \widetilde{\kappa_f} \rightsquigarrow \zeta_f \\ \widetilde{\kappa_f} \cong \widetilde{\kappa_p} \rightarrow^{\delta_{top}}_{\beta} \widetilde{\kappa_r} \\ \widetilde{\kappa_a};\widetilde{\kappa_p};[\,] \vdash^{genInst} \beta \rightsquigarrow \zeta_{ap} \\ \vdash^{flow} \widetilde{\kappa_p} \triangleright \widetilde{\kappa_a} \rightsquigarrow \zeta_{m1} \\ \zeta_{m2} \equiv \{\delta \triangleright_s \delta_{top}\} \\ \widetilde{\kappa} \equiv \_^{\delta} \\ \vdash^{flow} \widetilde{\kappa} \triangleright \widetilde{\kappa_r} \rightsquigarrow \zeta_r \\ \kappa_p \cong \kappa_a \\ \kappa_r \cong \kappa\end{array}}{\Gamma \vdash^{kind} f\ a : \widetilde{\kappa} \rightsquigarrow \zeta_{m1}\zeta_{m2}\zeta_r\zeta_{ap}\zeta_f}\ \text{D.APP}_{DC}$$

Figure 9.1: Type type rules (DC)

Figure 4.1 has a lot of resemblance with Figure 9.1. At each use of a type constructor, we generate an *Inst* constraint. Uses of type variables do not need constraints, similar to lambda-bound identifiers for expressions. Finally, at type applications, there are two coercion constraints generated to flow results from parameter to argument, and from type expression result to function result. This way, we connect the annotations on the kinds from the root of the type expression to the leaves.

How we treat annotations that occur in the type, we make explicit with a separate case for annotations on a type constructor. This is a subtle rule in the sense that it connects two annotations that live in different worlds: one annotation lives in the kind world and one annotation lives in the type world. An annotated type constructor is encoded as the alternative *Ann* just above the alternative *Con*. For example *Ann* δ (*Con* `"Int"` [ ]) represents $Int^{\delta}_{[\,]}$. In the type rule, we connect the annotation on the type constructor to the topmost annotation on the annotated kind of the type constructor. We want to know the results of the analysis for the annotations occurring on types, but the analysis only finds results for the annotated kinds. With the constraint generated for the annotations, the results of kinds are propagated to the actual annotations. From an implementation point of view, this approach makes sure that an algebraic data type definition collects proper constraints for exposed annotations.

We omit the type rules for binding groups and algebraic data type definitions and give only a textual description instead. For an algebraic data type definition, the constraints for all the types in the constructors are collected. The environment

is enlarged with annotated kinds of the type variables, derived by means of pattern matching on the annotated kind of the data type. An aggregation constraint is generated for each type variable that combines each use-site of the type variable. These constraints are then collected in binding-groups and passed to the solver. Types of an expression outside an algebraic data type declaration, each get their own binding group.

9.4 Constraint interpretation

To solve the constraints, we refer back to Chapter 4, and especially Section 4.3. Instead of two solver phases, we only need a single solver phase.

We consider variance first. There is a problem with variance in the sense that some of the propagation constraints are not negations, and negations are not represented by the constraint gathering in Section 3.5. But, the negations only occur in the constraint set of the function arrow, and thus we only have to put this new constraint into the initial constraint set:

$$(\rightarrow) :: *^{\delta_1} \rightarrow *^{\delta_2} \rightarrow *^{\delta_3}, \delta_3 \triangleright \delta_2, \delta_3 \triangleright_{\not\equiv} \delta_1$$
$$\mathit{Int} :: *^{\delta_4}$$

This new constraint has some slight complications on the constraint graph, however, since care has to be taken that simplification of an odd number of negation edges, results still in an odd number of negation edges. However, this is not much more involving than determining soft paths, so it is only a minor complication, especially because we can eliminate hyper edges for the analysis in this chapter entirely (Section 9.5).

The infrastructure remains the same. This may raise the question how we deal with variance and below-the-function-arrow when converting the constraints of this chapter to constraint graphs. We do not need an extensive analysis for this: the kind language is simple enough that we can obtain variance information with a variation of the functions given in the introduction of this chapter. Values for belowness are fixed at *NotBelow* as the concept of belowness is not important for the analysis on algebraic data types.

So, the only big change is in the interpretation of the constraints. We interpret the constraints to find a substitution. A substitution maps an annotation to some analysis result. In case of variance, we use for this the following lattice $\sqsubseteq_V$:

$$\begin{aligned} ? &\sqsubseteq_V + \\ ? &\sqsubseteq_V - \\ + &\sqsubseteq_V +/- \\ - &\sqsubseteq_V +/- \end{aligned}$$

Then the interpretation is defined as follows:

$$\begin{aligned} &\mathit{variance}\ (a \triangleright b) \\ &\quad = a \triangleright (a\ `\sqcup_{\sqsubseteq_V}`\ b) \\ &\mathit{variance}\ (a \triangleright_{\not\equiv} b) \\ &\quad = a \triangleright (\mathit{neg}\ a\ `\sqcup_{\sqsubseteq_V}`\ b) \\ &\mathit{variance}\ (a_1 \odot a_n \leqslant a) \\ &\quad = \mathbf{let}\ z = a_1\ `\sqcup_{\sqsubseteq_V}`\ ...\ `\sqcup_{\sqsubseteq_V}`\ a_n\ `\sqcup_{\sqsubseteq_V}`\ a \\ &\qquad \mathbf{in}\ \ z \odot ... \odot z \leqslant z \\ &\mathit{neg}\ + \ = - \\ &\mathit{neg}\ - \ = + \\ &\mathit{neg}\ x \ = x \end{aligned}$$

For the below-the-function-arrow analysis, no special constraints are needed. But, the initial substitution needs to be adapted for a special annotation δ that represents that it is below the arrow. Then the initial constraint set for below-the-function-arrow analysis is:

$(\rightarrow) :: *^{\delta_1} \rightarrow *^{\delta_2} \rightarrow *^{\delta_3}, \delta \triangleright \delta_1, \delta \triangleright \delta_2$
$Int :: *^{\delta_4}$

The lattice $\sqsubseteq_{BL}$ that we use is similar to that of variance:

$$
\begin{array}{lll}
? & \sqsubseteq_{BL} & below \\
? & \sqsubseteq_{BL} & not\_below \\
below & \sqsubseteq_{BL} & both \\
not\_below & \sqsubseteq_{BL} & both
\end{array}
$$

Again, we formulate the interpretation function in a familiar way:

$$
\begin{array}{l}
belowSolveF\ (a \triangleright b) \\
\quad = a \triangleright (a\ `\sqcup_{\sqsubseteq_{BL}}`\ b) \\
belowSolveF\ (a_1 \odot ... \odot a_n \leqslant a) \\
\quad = \textbf{let}\ z = a_1\ `\sqcup_{\sqsubseteq_{BL}}`\ ...\ `\sqcup_{\sqsubseteq_{BL}}`\ a_n\ `\sqcup_{\sqsubseteq_{BL}}`\ a \\
\qquad \textbf{in}\ \ z \odot ... \odot z \leqslant z
\end{array}
$$

The result is that the constraints specify the traversal over the data types, and that the interpretation function perform the analysis that we did by hand in the introduction of this chapter.

9.5 Graph simplification

As we mentioned before, the graph simplification can remain largely unaltered. However, we can convert an aggregation constraint for these analysis into coercion constraints. This has the advantage that the graph reducer does not need to deal with aggregation constraints. A constraint of the form $a \odot b \leqslant c$ means $a == b == c$ in the interpretation, thus we can replace it with $a \triangleright b \triangleright c$ and $c \triangleright b \triangleright a$. The graphs for data types are even more sparse than those of expressions, which means that the resulting graphs are very small. This is important, because the above analysis is applied to all types occurring in constraints.

9.6 Conclusion

This chapter shows that the results of previous chapters can not only be used for uniqueness typing, but for two 'completely different' analyses as well, without much additional work. We did not compare the performance of the variance approach to a non-constraint based variance approach, but we think that due to the simplifications on the constraint graphs, that our approach is not much slower than a direct implementation.

Chapter 10

Dealing with overloading

If our analysis would be in the back-end of the compiler, overloading would already be resolved and dealing with overloading would be trivial. However, our uniqueness typing approach is situated in the front-end of the compiler. This choice has as consequence that we need to deal with overloading.

Uniqueness typing and overloading interact with each other. There are several ways to deal with this interaction. The easiest way is to assume that all cardinality annotations on an overloaded function are shared, which basically keeps the uniqueness typing out of the overloading business, but results in poor typings for expressions that use overloaded functions. There are other ways, which does involve a higher amount of interaction between both systems, that give better results.

10.1 Example

Consider the following expression:

```
class Replaceable f where
  replace :: b → f a → f b
instance Replace Maybe where
  replace _ Nothing = Nothing
  replace x (Just _)  = Just x
data Tup a = Tup a a
instance Replace Tup where
  replace x (Tup _ _) = Tup x x
instance Functor f ⇒ Replaceable f where
  replace x = fmap (const x)
```

The question is: what is the constraint set of *replace*?

10.2 Conservative approach

The approach we mentioned in the introduction of this chapter, works by generating an annotated type for *replace* with all annotations set to shared ($* \bowtie *$). This has an effect on the uses of an overloaded identifier, and for the expressions of an instance declaration.

Suppose we have the following situation:

class ... **where**
  $i^a :: \widetilde{\tau}_1$
instance I . . **where**
  $i^b = E :: \widetilde{\tau}_2$
 ... $i^c :: \widetilde{\tau}_3$ ...

At the use-site of an overloaded identifier, in this case i^c, we generate an *Inst* constraint that refers to the corresponding binding group. The interpretation of the *Inst* constraint is slightly different, in the sense that all annotations on the use-site type need to be constrained to $* \bowtie *$. In this case, the *Inst* constraint for i^c maps each annotation in $\widetilde{\tau}_3$ to $* \bowtie *$. We augment the constraint set of each binding in an instance declaration with constraints mapping the annotations on an instance declaration to $* \bowtie *$.

This way, we created a barrier of $* \bowtie *$ annotations between the use-site of an overloaded identifier and instance declarations, and ensured that dictionary passing, instance transformation, and instance selection are fully separated from uniqueness typing. We achieved this at a cost: values passed or obtained from an overloaded function cannot be unique and therefore cannot be optimised.

10.3 Improved approach

The above procedure gives poor results when using overloaded identifiers. The problem is that we are too conservative with the constraints. Instead of mapping everything to $* \bowtie *$, a constraint set that is more restrictive than any of the constraint sets of the instances, is a better approximation. Determining this constraint set is difficult, as instances can be added later, and because one instance may have a deeper structured type than another instance. Instead of calculating this constraint set, we ask the programmer to specify it, and then force it in each instance declaration.

In Section 5.3, we discussed how to deal with programmer-specified constraint sets, and what the problems are. This approach allows better results for overloaded identifiers, but is not a very good solution either.

10.4 Advanced approach

A proper way to deal with overloading is by a careful inspection of the results of overloading resolving. Since conventional type inferencing already took place, we can obtain a proof for each overloaded identifier, that specifies which dictionaries need to be inserted and how these dictionaries are obtained. For overloaded identifiers, we store this proof in the *Inst* constraint, instead of a binding-group identifier. For our explanation, we consider the proof to be the expression that is inserted by the compiler as a result of overloading resolving by the conventional type inferencer. This is a normal expression, of which the identifiers represent a dictionary or dictionary transformer. Since we know which dictionary originates from which instance declaration, we know for each of these identifiers what the corresponding binding-group is. Thus we can perform our constraint gathering on the proof and use the resulting constraint set for instantiation of an overloaded identifier.

A way to look at this approach is that we extract a piece of the back-end (the generated code for overloading resolving) and examine it in the front-end for constraint generation. This approach gives the best results, as the outcome is the same when overloading is already resolved. A downside is that this approach needs to examine the results of overloading resolving, and depending on the compiler, these results may be hard to obtain if they are produced very deep inside the compiler. This is for example the case in EHC, where overloading is resolved during type fitting.

10.5 Probing one step further

We can yet go one step further by letting the uniqueness typing influence the overloading resolving process. Unfortunately, this is not possible in our implementation, since it assumes that conventional type inferencing already took place. Integrating it into the type system is a non-trivial problem: normal typing influences the constraint sets, and if uniqueness typing influences the typing, then it becomes tricky. However, our entire approach is based upon some fixpoint improvement of the constraint sets. From this perspective, taking care of improved type information while constructing this fixpoint should be doable. But, to keep this approach manageable from an engineering point of view, a good starting point may be a type system where aspects of unification are encoded in relative isolation, such as in the Top-solver [14] for Helium [15].

The question in this case is, is it worth it? For that, we investigate what kind of influence the uniqueness typing can have upon conventional type inference. Instead of type variables, we can give a class declaration uniqueness variables. For example:

class $Array\ \alpha\ \delta$ **where**
$array :: Ix\ i \Rightarrow (i, i) \rightarrow [(i, \alpha)] \rightarrow Arr^{\delta}\ i\ \alpha$
$(!) :: Ix\ i \Rightarrow i \rightarrow Arr^{\delta}\ i\ \alpha \rightarrow \alpha$
$update :: Ix\ i \Rightarrow i \rightarrow \alpha \rightarrow Arr^{\delta}\ i\ \alpha \rightarrow Arr^{\delta}\ i\ \alpha$

At the root of the program, or earlier, δ will become known. Subsequently, an instance is selected that fits the total usage of the value (if such instance exists). For example, suppose there is an instance for any δ that just uses the default array implementation if the value is used in an arbitrary way, and uses a special instance when the value is guaranteed to be used at most once. This special instance can perform an in-place update.

With this feature we can accomplish some code-generation for uniqueness typing by using overloading. Unfortunately, the runtime overhead of dictionary passing can become more costly than the optimisation is worth. But that can partially be resolved by inlining and partial evaluation of dictionaries [16].

It is important to note that once an instance is chosen, that it cannot be replaced by another instance, because instances represent expressions, and expressions have an effect upon the usage of other values. If we would replace the instance for another instance, then we need to undo the effects of the other instance, which makes the typing non-monotone and possibly never terminating. Therefore, the selection of an instance should not influence the cardinality variable that is used for its selection.

An instance can be selected when the cardinality value corresponding to it, is fixed. Each instance constraint of an overloaded identifier checks if it can select an instance. If it can, it uses the constraint graph of that instance for instantiation. It registers the fact this instance is chosen. In a subsequent iteration the substitutions are affected by this choice and the effects propagates to other places. At the end of the inference, we check for each *Inst* constraint of an overloaded identifier that the cardinality of the substitution matches is equal to the cardinality recorded at the moment an instance is chosen. If that is not the case, then the instance influenced the value of the cardinality annotation that is used to select the instance. In that case the instance is incorrect.

This approach is probably difficult to integrate into a compiler due to the interaction with instance selection and unifications.

10.6 Conclusion

This chapter showed the complications resulting from dealing with overloading. We can approach these complications at different levels of interaction. In Section 10.2, we present an approach that keeps the overloading mechanism and the cardinality inferencer separate by means of a barrier of $* \bowtie *$ annotations. Section 10.4 shows an approach that gives better results but is (slightly) more difficult to implement. Finally, we can allow full interaction between overloading

and a cardinality analysis, which offers nice opportunities to use the overloading mechanism to select special code depending upon usage of a value. Unfortunately, this last approach complicates the compiler severely.

Chapter 11

Separate compilation

The type system we discussed is tailored to several languages of the EH family. At the time of writing, the languages of the EH family are not ready for industrial use. To make the EH language industry ready, some problems will need to be dealt with, which also influences uniqueness typing. Among these problems are a module system and separate compilation. We discuss how to deal with these problems in this chapter.

A module system and separate compilation are similar problems, except that they occur in different places within the compiler. Separate compilation is mainly a code generation issue, whereas a module system is a type system issue. Separate compilation requires a module system to split up a program in different compilation units. The uniqueness type system complicates these problems.

11.1 Module system

We take the module system of Haskell as example. A program is partitioned in modules, where a module can import toplevel bindings from another module, and export toplevel bindings. Importing a module does not complicate the uniqueness type system much: put the constraints of the importing module into a new binding group, and bring the identifiers into scope. In case of mutually recursive modules, the constraints should be put into the same scope as the toplevel scope of the current module. So, for importing a module, the existing infrastructure suffices. It is the exporting of modules that causes problems.

The problem is: how often is an exported binding used? That depends on the importer of the module, but the idea of a module is that they can be examined in isolation. That is why we make the conservative assumption that all exported bindings are used in an arbitrary way.

Annotations that occur on argument or result values do not have to be made arbitrary, since they are determined by the use-sites of the exported binding. Generating a $\tau_{any}^{*\bowtie *} \leqslant \widetilde{\tau}_{binding}$ constraint is sufficient to make the conservative assumption.

The consequences of this assumption are relatively low. Exported top-level bindings are typically functions, not values. Exported functions are intended to be shared, otherwise there would be not much use to put the function in another module. Arguments and results are not affected, which is important, otherwise almost no value can be made unique when using such an exported function.

Depending on the code generator, additional restrictions need to be specified. In case of separate compilation with a code generator, the type system can require that an exported binding has a monovariant cardinality annotation on the type. However, this leads to language design issues: is the programmer forced to specify a monovariant type, or should the compiler choose one. The first case puts a burden on the programmer, the second case leads to yet another choice:

should the compiler pick a monovariant type that is valid for all uses, or a more restricted type, with the consequence that some legal programs will not be accepted?

The above mentioned problems do not have satisfactory solutions. The problems—perhaps *limitations* is a better word for them—originate from design choices in both the language and the code generator of the compiler. A solution to these kind of problems can perhaps be found in allowing the programmer to choose certain features of the module system and the compilation process, and accept the limitations as a result from these choices.

11.2 Separate compilation

Separate compilation is difficult for a code generator that works with a specialiser. The specialiser only knows which variants of a binding are in use within the same module. The variants of a binding that are used by another module are not known by the specialiser, as these are discovered during a future compilation. So, which variants of a binding should the specialiser put into the code module? The specialiser cannot produce all combinations, since there are exponentially many variations in the number of cardinality annotations, and there are already a lot of cardinality annotations.

The other extreme is not to generate any variants for a module, but to put a description in the module of how a variant can be obtained. This description is a representation of the abstract syntax tree of the module decorated with some information needed for the specialization process. This process can hardly be called separate compilation, as the code generation process itself is effectively performed on a whole program at a time, although it depends on how short the description can be made, although it is not obvious how such a description would look like, other than just a representation of the abstract syntax tree of a function.

A way in between is to only allow variants for exported bindings within the same module, and for other modules, only allow one variant. That is, the binding of an exported identifier is polyvariant in the same module, but monovariant from other modules. The selection of a monovariant uniqueness type from the allowed types given its polyvariant type, is a problem. Choose the type too restrictive, then it's more difficult to use the binding from other modules, but choose it too general, and opportunities for optimisations are lost.

There is no problem if the code generator works with code specialisation by dictionary insertion. The dictionaries represent the variability, and the code for a module itself can stay the same. But this approach has as problems the overhead of dictionary passing (especially since dictionaries for cardinalities are likely inserted at the root of the program only), and does not support all code generation optimisations that are possible. The overhead of dictionary passing can be eliminated by specialisation of dictionaries by means of partial evaluation techniques [16], but such an approach leads to similar problems with separate compilation.

Since this is a common problem for an optimising compiler, perhaps that separate code generation is too much to ask for in the presence of code optimisations.

Chapter 12

Inspecting Results

Optimisations, such as in-place update, can have severe consequences on performance. In some circumstances, a programmer relies upon some optimisations being applied. As a programmer, we want to make sure that this is also the case. If the optimisation is not applied, we want to know why not. This requires a way for the compiler to output the results of the analysis, or a way for the programmer to enforce restrictions. We briefly consider both aspects in this chapter.

12.1 HTML Pretty Printing

When creating the prototype, we discovered that there are a lot of annotations occurring everywhere, and that constraints graphs tend to get confusingly big (as in not fitting on a few pages anymore). Such graphs are difficult to interpret for a human being. We discovered that the graphs are not important for a programmer. The analysis is essentially a huge propagation of reference counts with some coercions and some additions. A programmer is interested in reading the results of the analysis on the types of identifiers that occur in the program. For example, if the analysis returns that some annotation indicates that a value is used in an arbitrary way, the programmer wants to see where in the type the annotation is located, and check related annotations to investigate what is causing it.

But, there is a problem: the analysis is polyvariant. In a single picture, we can only give concrete values for cardinality annotations on the types of the outermost binding-group. For the inner binding-groups, we have no substitution, since there can be many or even no uses of a binding-group, so which one should we pick? Therefore, we propose an interactive solution.

The compiler creates an HTML file with a pretty print of the program (See Figure 12.1 for an example). Initially, all the pretty-prints of the binding-groups are inactive, indicated with a gray color. Only the outermost binding-group is active, indicated by a black color and some syntax highlighting. The identifiers of an active binding-group are clickable. If such an identifier is clicked, it activates the corresponding binding-group. For each active binding-group, we know the path from the root of the program, and the corresponding substitutions. Each identifier in an active binding group can be expanded such that the annotated type becomes visible, with the substitution applied to it.

Interactive inspection of the analysis result appears to be very useful. The outcome of the analysis is typically large, but the interactive pretty print gives tight control over what is visible and what not. For example, a typical use-case is:

> Why is this integer not unique? Oh I see, it is passed to a function that says that it is shared. Ok, lets open up the definition of the function to see why this is the case. Ah, the parameter is passed twice to some function and it appears that it uses both arguments once.

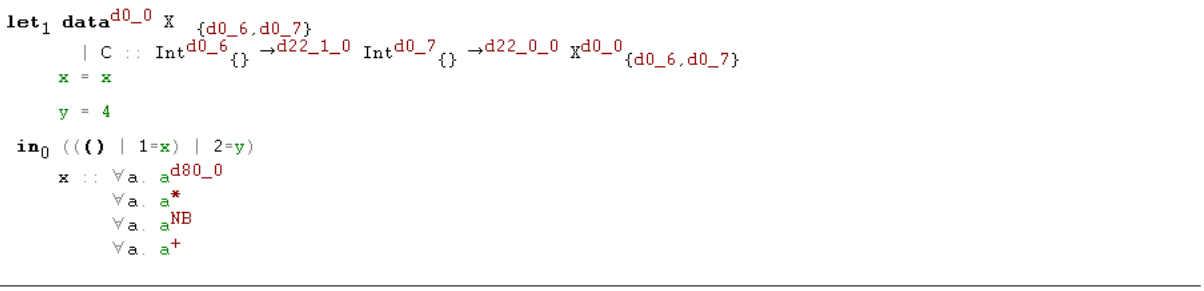


Figure 12.1: Example of an HTML pretty print

There are some issues. We just assume that clicking on a recursive definition does not change anything. But this way we cannot look into a function that has different cardinalities for a recursive call. On the other hand, this does not occur often. But even when ignoring recursive definitions, the number of possible substitutions on binding groups is large, since chunks of HTML-code need to be generated for each sequence of identifiers on an execution path. This does not scale up to large programs. For a large program, either some direct interaction with the solver is required, or if we keep using HTML pretty-prints, the compiler needs to generate some piece of JavaScript code that can calculate the substitutions. Unfortunately, this does not work well together with the fixpoint iteration on constraints (Section 5.4), since that would require most of the inferencer to be available in JavaScript as well. For the normal fixpoint iteration approach (Section 5.2), only a table representing the constraint graph and the computation of a substitution from this table, needs to be generated for each binding group.

Our experience shows that in order to understand the uniqueness typing of a program, it helps to have some interactive and visual feedback. The pretty printing approach of this section helps a lot, and it may be an interesting future work to see if there are other techniques to explain to the programmer why or why not the compiler infers that some value is unique or not.

12.2 Signatures

But inspecting a program is not enough. Sometimes, you want to enforce that some type has a certain cardinality in order to guarantee that some optimisation is made. Or, as is the case in Clean, where a unique world is required for IO. Care has to be taken that the world remains unique throughout the program. We discuss how we can use signatures to enforce certain uniqueness properties.

Section 5.3 described how signatures can be specified by the programmer. Signatures to replace the constraints for a binding group are difficult to specify since it needs to capture dependencies on other binding groups. But we can augment signatures with additional constraints given by the programmer.

We assume that the programmer specifies the constraints in the same way as in Section 5.3.1, but treat the specification differently. The programmer gives a partially annotated type, and a set of constraints. Parts of the type that are not annotated do not result in additional constraints. The solver is adapted such that it signals an error when the value of a concrete annotation is changed. For example, if the programmer supplies the constraints $\delta \triangleright 1 \bowtie 1, 1 \bowtie 1 \triangleright \delta$, but the inferencer derives that δ is $1 \bowtie *$, the substitution of δ is changed from $1 \bowtie 1$ to $1 \bowtie *$, which we consider an error since $1 \bowtie 1$ was given by the programmer. A special *top* value is used for the cardinality annotation to indicate that the annotation is in error.

To improve the quality of the error message, the programmer can attach an error message to a constraint. This requires another usage property above $*$ represeting a conflict with an error message. For example:

> **data** $FileHandle^{\delta} = ..., \delta \equiv 1^{-} \bowtie 1^{-}\{$*A file handle must remain unique*$\}$
> $inplaceSort :: Array\ Int^{\delta} \rightarrow Array\ Int, \delta \equiv 1^{-} \bowtie 1^{-}\{$*Only arrays with unique elements can be sorted* **in** $-$ *place*$\}$

There are two ways to proceed from here. The HTML file can be used to investigate a trail of *top* values through the program and indicate the sites that cause trouble. Or, we can generate an error message if we encode some context information in the constraints, such as the file location of the expression that generated the constraint. Care needs to be taken that this context information is preserved when reducing the graph (or, replay the substitution on the unreduced graph to obtain context information).

Chapter 13

Conclusion

Chapter 3 defines a constraint-based type system for uniqueness typing. The language of this chapter does not even come close to that of a real function programming language, but provides sufficient features to show which constraints are generated and how they are interpreted to check a uniqueness typing of a program. Chapter 4 extends on this approach by showing how uniqueness *inference* can be performed on the constraints. In the chapters that follow, we gradually showed how language features interact with the type system. We did not shy away from implementation-level problems (such as supporting data types) and present problems that we encountered and discuss solutions. Finally, in Chapter 8 we ended up with a uniqueness type inferencer that supports sufficient language features to use in the front-end of a compiler. Chapter 12 shows that the analysis of the inferencer in the front-end of the compiler, is important to communicate demands and results with the programmer.

Key advantages of the type system are that it is constraint-based, fully polyvariant, accurate and scalable. It is constraint-based so the implementation is split up in multiple phases, separating concerns and isolating changes. It is fully polyvariant (Chapter 4), meaning that each use-site can have different uniqueness types (if the constraints allow it), also in the presence of recursion (Chapter 5). By means of special β annotations, we accomplish a form of higher-rankness in uniqueness types. We show in Section 5.4 that polyvariant recursion for uniqueness types is decidable, if polymorphic recursion is taken care of. The type system is accurate in the sense that it allows components of a value to be typed differently from the spine of a value, such that functions that only touch the spine, do not influence the components. This is our main difference with other approaches, as it requires us to assume that the presence of some identifier does not guarantee that the identifier is actually used. Finally, with scalable, we mean that there are several possibilities to improve the performance drastically by giving up some accuracy. Performance is scaled by influencing two factors, the number of annotations (Section 6.2, Section 8.2) and the number of instantiation-operations on the constraint graph (Section 5.4).

Unfortunately, there are some disadvantages. During prototyping, we discovered that despite our ways to isolate changes, it still complicates the compiler from an engineering point of view. Since uniqueness types can occur everywhere, any change to the compiler requires a verification that the uniqueness types are preserved by the change. Since Haskell does not rely on uniqueness typing in order to perform IO [28], and the rise of smart data-type implementations, such as versioned arrays (`Data.Array.Diff`), it remains a question if the additional complications of uniqueness typing are worth it.

13.1 Related work

Philip Wadler [37], presented a linear type system for functional languages. This system assumes that each occurrence of an identifier occurs sequential and is always used. As a consequence, the type system is virtually the same as a conventional type system, except with restrictions on the environment.

The Clean compiler [2] analyzes the abstract syntax tree how different uses of an identifier occur, and marks a type of an identifier shared if it occurs more than once in sequence. The type system then propagates these values accordingly. The type system of Clean gives a polyvariant uniqueness typing. There is support for data types, but data types cannot be parameterized over uniqueness annotations occurring in types of a data-type definition (Section 8.2).

Clean has a restriction that if a component of a value is unique, then the spine must be unique as well. Or in other words, that the components are assumed to be used at least as much as the spine. This ensures that Clean does not have the graph-duplication problems for instantiation in the presence of polymorphism (Section 6.4). A consequence is that in curry-style function application, the order in which the parameters are defined matters. There can be no unique parameter to the left of a shared parameter.

Another restriction is that Clean assumes that each occurrence of an identifier means that it is also used. The elements of the list *xs* cannot be marked as unique in Clean, despite the fact that the elements are used only once, since the *length* function does not touch the elements:

$$(\lambda xs \rightarrow map\ (+length\ xs_1)\ xs_2)\ [1,2,3,4]$$

Our analysis discovers that length does not use values of *xs*; it derives $xs_1 :: List^{1 \bowtie * }\ Int^{0 \bowtie 1}$ and $xs_2 :: List^{1 \bowtie * }\ Int^{1 \bowtie 1}$, which results in a linear type for the integers in the list.

A consequence of these restrictions is that the type system of Clean performs less work. Uniqueness signatures are rather trivial to implement in the system of Clean, as well as the interaction with a class system, contrary to signatures in our system (Section 5.3) and (Section 10.4).

In a recent PhD thesis, Keith Wansbrough [39], gives an overview of different usability-analysis, and discusses an implementation of usage-analysis in the back-end of the GHC compiler. His implementation is polyvariant, and uses a similar way of dealing with data types as our exposed annotations in Section 8.2. However, this type system also makes the assumption that spine of a value is as unique as the elements.

13.2 Complexity comparison to Clean

Using information about expressions that are not used, has consequences. Uniqueness typing for Clean, for instance, assumes that all expressions are used at most once. The reference counting phase then amounts to marking identifiers unique if there is only one occurrence in the abstract syntax tree or only parallel occurrences, and shared if that is not the case. The uniqueness propagation phase then ensures that the expressions that were incorrectly assumed to be used at most once (these are the values that are passed as argument to a shared parameter of a function), get a shared type and propagates this sharing to all places where the value occurs. So, the Clean compiler performs a local reference count analysis and a global uniqueness propagation.

We do not make the assumption during reference counting that expressions are used at most once. We just assume that an expression is used δ times, where δ is determined at another time. This means that in situations where there are multiple (sequential) occurrences of an identifier, we do not yet know if the identifier is shared because the occurrences may be unused. So, our reference count analysis is a global analysis and we pay for that in complexity. Consequences are that we need constraints that combine multiple occurrences (the aggregation constraint), which makes constraint sets complicated. For example, the notion of entailment is difficult to express. As a result, dealing with constraint duplication in the presence of polyvariance is hard (graphs with hyper edges, graph reduction). On the other hand, our uniqueness propagation phase is rather trivial, because all the work is done by the reference counting phase.

13.3 Future work

There are fourth directions of future work. In the first place, the lessons learned by the prototype and this thesis, can be used to create a faster performing prototype, where more care is taken to speed up the graph algorithms, and has more options to choose between accuracy and performance.

Secondly, the relation between uniqueness typing and overloading requires more thorough investigation. For example, is it possible to implement overloading resolving that take cardinality variables in account.

Thirdly, the question is if the interaction with the user can be improved, by means of signatures, the identification of problem sites, or other mechanisms to inspect the result of the analysis.

Finally, there is the topic of using the uniqueness types to improve code. At the time of writing, a code generator is added to EHC [6], and other approaches of code generation for EHC are under investigation. There is already a lot of work done in this field, but it remains a question which of that work is applicable for EHC, and how to treat the problem of polyvariance.

So, this master's thesis opens up several topics of research for other master's thesis projects.

Appendix A

Implementation

This appendix provides some information concerning the implementation of the uniqueness typing as discussed in this master's thesis. Actually, we made two prototype implementations:

- A version for EH 7 or 8. This is an extensive version that goes into detail in annotating data types, enriching the type language with exposed annotations and expanded types. It deals with records, pattern matches, higher-ranked types, existential types, and impredicative types. It is created to support all aspects of the EH 7 language, since one of our research questions is how to deal with the language constructs of Haskell. The implementation uses a general architecture that can be instantiated for a whole range of analyses, such as polarity, uniqueness, and strictness inference. We used this version as research vehicle for the later chapters of this master's thesis.

- A version for EH 2. Since EH 2 is a simple language, with less complications around each corner. This version allows us to focus and experiment with the main concepts of the uniqueness inference. It is easier to experiment with. For example, the concept of β annotations was discovered near the end of the project. Adding β annotations to EH 7 or EH 9 is rather hard, since it interferes with unification. It requires changes in the unification function of EH, called *fitsIn*, and in the construction of types for data types. Given our experience with annotating types with δ annotations, we expect integrating β annotations in EH 7 to take a couple of weeks, instead of the two days for EH 2. We used this version for the construction of the Chapter 3 and Chapter 4. This version also uses a whole different graph representation.

The architecture of the EH 2 version is basically the same as the EH 7 version, so we ignore the differences for the remainder of this appendix.

A.1 Installation

The implementation is currently on a separate branch (named `uniqueness`) of the EH project. Checkout the repository using:

```
svn checkout https://svn.cs.uu.nl:12443/repos/EHC/branches/uniqueness/EHC/
```

Build the compiler by running `make bin/7_2/ehc`. After compilation, there is an executable named `ehc` available in the `bin/7_2` directory. For example, create a file named `Test.eh`:

let $id = \lambda x \rightarrow x$
in $id\ 3$

Running the compiler over `Test.eh` using `ehc Test.eh` results into an `.html` file being generated named `Test.eh.html`. This `.html` file contains a pretty print of the above program, displaying annotated types that indicate that 3 is used at most once.

A.2 Files and architecture

The uniqueness typing implementation consists of several components. There is one distinct component which is the constraint solving backend. All other components are modifications and additions to existing components of EHC. All backend related code is located in the folder `src/ehc/Annotations/`. The following files are of interest:

- The file `Annotations.chs` contains all utility code related to annotation trees, graph rewriting and fixpoint iterations.
- The file `ConstraintSolver.chs` provides utility functions to construct constraint solvers parameterized by solve functions for a specific analysis.
- The solve functions for different analyses can be found by their corresponding name in this folder.
- The file `AnnSolvers.cag` instantiates all solvers and pushes a uniqueness substitution throughout the abstract syntax tree.

Consult the implementation and corresponding inline documentation for more information regarding these topics.

The other components are situated throughout the EHC source. For example, a large amount of annotation related utility functions can be found in the `src/ehc/Ty` folder. These include functions for annotating types, expanding types (for type constructors for example), kind inferencing on annotated types, and pretty printing.

The code for inferencing constraint set is located in the `src/ehc/EH` folder. Part of the code is generated by `ruler`, but a large portion is just `AG` code. Of particular interest in this folder are `ConstrInferData.cag`, `ConstrInferExpr.cag`, and `ConstrInferTy.cag`. These are the files that generate constraints for expressions and types. The parts generated by ruler are located in `ehc/rules/EhcRulesOrig.rul`.

Bibliography

[1] P. Achten and M. J. Plasmeijer.The ins and outs of clean i/o.*Journal of Functional Programming*, 5(1):81–110, 1995.

[2] E. Barendsen and S. Smetsers.Uniqueness typing for functional languages with graph rewriting semantics.*Mathematical Structures in Computer Science*, 6(6):579–612, 1996.

[3] B. De Schutter and T. van den Boom.Model predictive control for max-min-plus-scaling systems — Efficient implementation. In M. Silva, A. Giua, and J. Colom, editors, *Proceedings of the 6th International Workshop on Discrete Event Systems (WODES'02)*, Zaragoza, Spain, Oct. 2002.

[4] A. Dijkstra.EHC Web.`http://www.cs.uu.nl/groups/ST/Ehc/WebHome`, 2004.

[5] A. Dijkstra and S. D. Swierstra.Typing Haskell with an Attribute Grammar (Part I).Technical Report UU-CS-2004-037, Department of Computer Science, Utrecht University, 2004.

[6] C. Douma.Exceptional GRIN.Master's thesis, Utrecht University, Institute of Information and Computing Sciences, 2006.

[7] D. Duggan and J. Ophel.Type-Checking Multi-Parameter Type Classes.*Journal of Functional Programming*, 2002.

[8] D. Dussart, F. Henglein, and C. Mossin.Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time.In *SAS*, pages 118–135, 1995.

[9] D. Dussart, F. Henglein, and C. Mossin.Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time.In A. Mycroft, editor, *Proc. 2nd Int'l Static Analysis Symposium (SAS), Glasgow, Scotland*, volume 983 of *Lecture Notes in Computer Science*, pages 118–135. Springer-Verlag, September 1995.

[10] A. J. Gill and S. L. Peyton Jones.Cheap deforestation in practice: An optimizer for haskell.In *IFIP Congress (1)*, pages 581–586, 1994.

[11] J.-Y. Girard.Linear logic: Its syntax and semantics.In J.-Y. Girard, Y. Lafont, and L. Regnier, editors, *Advances in Linear Logic (Proc. of the Workshop on Linear Logic , Cornell University, June 1993)*, number 222. Cambridge University Press, 1995.

[12] C. V. Hall, K. Hammond, W. Partain, S. L. Peyton Jones, and P. Wadler.The glasgow haskell compiler: A retrospective.In *Functional Programming*, pages 62–71, 1992.

[13] D. Harrington.Uniqueness logic.*Theoretical Computer Science*, 354(1):24–41, 2006.

[14] B. Heeren.*Top Quality Type Error Messages*.PhD thesis, Utrecht University, Institute of Information and Computing Sciences, 2005.

[15] B. Heeren and A. v. IJzendoorn.Helium, for learning Haskell.`http://www.cs.uu.nl/helium/`, 2005.

[16] M. P. Jones.Dictionary-free overloading by partial evaluation.In *Partial Evaluation and Semantics-Based Program Manipulation, Orlando, Florida, June 1994 (Technical Report 94/9, Department of Computer Science, University of Melbourne)*, pages 107–117, 1994.

[17] M. P. Jones.Typing Haskell in Haskell.In *Haskell Workshop*. Utrecht University, Institute of Information and Computing Sciences, 1999.

[18] M. P. Jones.Type Classes with Functional Dependencies.In *Proceedings of the 9th European Symposium on Programming, ESOP 2000,*, March 2000.

[19] M. P. Jones and S. L. Peyton Jones.Lightweight Extensible Records for Haskell.In *Haskell Workshop*. Utrecht University, Institute of Information and Computing Sciences, 1999.

[20] A. Kfoury and J. Wells.Principality and Decidable Type Inference for Finite-Rank Intersection Types.In *Principles of Programming Languages*, pages 161–174, 1999.

[21] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn.Type reconstruction in the presence of polymorphic recursion.*ACM Transactions on Programming Languages and Systems*, 15(2):290–311, 1993.

[22] J. Launchbury and S. L. P. Jones.Lazy functional state threads.In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 24–35, 1994.

[23] F. Nielson, H. R. Nielson, and C. Hankin.*Principles of Program Analysis*.Springer, 1999.

[24] S. Peyton Jones, M. Jones, and E. Meijer.Type classes: an exploration of the design space.In *Haskell Workshop*, 1997.

[25] S. L. Peyton Jones.Implementing lazy functional languages on stock hardware: The spineless tagless g-machine.*J. Funct. Program.*, 2(2):127–202, 1992.

[26] S. L. Peyton Jones.Compiling haskell by program transformation: A report from the trenches.In *ESOP*, pages 18–44, 1996.

[27] S. L. Peyton Jones.*Haskell 98, Language and Libraries, The Revised Report*.Cambridge University Press, 2003.

[28] S. L. Peyton Jones.Haskell 98 libraries: Input/output.*Journal of Functional Programming*, 13(1):205–218, 2003.

[29] S. L. Peyton Jones.Haskell 98: Predefined types and classes.*Journal of Functional Programming*, 13(1):81–96, 2003.

[30] S. L. Peyton Jones, W. Partain, and A. Santos.Let-floating: Moving bindings to give faster programs.In *ICFP*, pages 1–12, 1996.

[31] S. L. Peyton Jones and M. Shields.Practical type inference for arbitrary-rank types. `http://research.microsoft.com/Users/simonpj/papers/putting/index.htm`, 2004.

[32] S. L. Peyton Jones and M. Shields.Practical type inference for arbitrary-rank types, Mar. 2004.

[33] B. C. Pierce.*Types and Programming Languages*.MIT Press, 2002.

[34] R. Plasmeijer and M. v. Eekelen.*The Concurrent Clean Language Report (draft)*.Department of Software Technology, University of Nijmegen, 2001.

[35] A. Santos.Efficient compilation of functional languages by program transformation.*Electronic Notes in Theoretical Computer Science*, 14, 1998.

[36] P. Wadler.A New Array Operation.In J. H. Fasel and R. M. Keller, editors, *Graph Reduction, Proceedings of a Workshop*, volume 274, pages 328–335, Santa Fe, New Mexico, September 29 – October 1, 1986. Springer, Berlin.

[37] P. Wadler.Linear types can change the world!In M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, pages 347–359. North Holland, 1990.

[38] P. Wadler and R. J. M. Hughes.Projections for Strictness Analysis.In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274, pages 385–407, Portland, Oregon, USA, September 14–16, 1987. Springer, Berlin.

[39] K. Wansbrough.*Simple Polymorphic Usage Analysis*.PhD thesis, University of Cambridge, March 2002.

[40] S. C. Wray and J. Fairbairn.Non-strict languages — programming and implementation.*The Computer Journal*, 32(2):142–151, 1989.