Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Informatics

Extending a refinement λ-calculus with polymorphism and data types

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing
UE 066 937

eingereicht von

Jakob Hoffmann
Matrikelnummer 12044470

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dr.sc. Jürgen Cito
Mitwirkung: Univ.Ass. Dipl.-Ing. Michael Schröder

Wien, 2. Dezember 2024

Jakob Hoffmann

Jürgen Cito

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

TU Bibliothek Your knowledge hub

TU WIEN Informatics

Extending a refinement λ-calculus with polymorphism and data types

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing
UE 066 937

by

Jakob Hoffmann
Registration Number 12044470

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl.-Ing. Dr.sc. Jürgen Cito
Assistance: Univ.Ass. Dipl.-Ing. Michael Schröder

Vienna, December 2, 2024

Jakob Hoffmann

Jürgen Cito

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Erklärung zur Verfassung der Arbeit

Jakob Hoffmann

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 2. Dezember 2024

Jakob Hoffmann

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Acknowledgements

I would like to express my heartfelt gratitude to several individuals who made the completion of this thesis possible.

First and foremost, my supervisor Michael Schröder, who has been instrumental throughout the research process for this work. His willingness to take the time to discuss ideas, provide feedback and help was invaluable to the progress and success of this work. I am also grateful to Prof. Jürgen Cito, for his insightful advice and for being available to assist me at key stages of my research. His expertise and direction greatly enriched this thesis and also my presentation skills.

A warm thank you also goes to my flatmates, who fostered a joyful and supportive home environment throughout this journey. Their companionship and positivity helped me stay motivated and balanced during the writing process.

I would also like to acknowledge my employer, for their flexibility and understanding that enabled me to focus on my studies, as well as the Republic of Austria for enabling me to go on educational leave, which provided me with the necessary time and financial stability to finish this thesis.

To everyone mentioned, and to those who have supported me in ways large and small, thank you.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Kurzfassung

Das Parsen von Stringeingaben ist eine häufige Programmieraufgabe, die oft ad hoc ohne Spezifikation einer formalen Grammatik implementiert wird. Die automatische Inferenz solcher Grammatiken bietet viele Vorteile und das existierende Panini-System verwendet Refinement Typen, um Eingabegrammatiken für ad hoc Parser automatisch zu generieren. Dieser Ansatz ist jedoch durch den zugrunde liegenden λ_Σ-Kalkül eingeschränkt, dem Polymorphismus und algebraische Datentypen fehlen. Das Fehlen dieser häufig verwendeten Konzepte verhindert die Inferenzfähigkeit vieler Grammatiken.

Um diese Einschränkungen zu beheben, erweitern wir den λ_Σ-Kalkül um die fehlenden Konzepte. Der resultierende λ_Σ^+-Kalkül führt außerdem eine neue Inferenzregel für die Zerlegung von Datentypen ein. Wir kategorisieren verschachtelte Grammatikstrukturen, die häufig in ad hoc Parsern auftreten, und konzentrieren uns auf Grammatiken, die durch *split*-Operationen und die dadurch erzeugten Listen und Tupel erzeugt werden. Für die Synthese solcher Grammatiken stellen wir einen Ansatz vor, der die verschachtelten Grammatiken innerhalb einer Datenstruktur zu einer Gesamtgrammatik zusammenführt.

Diese Erweiterungen integrieren wir in das Panini-System und evaluieren sie anhand realer ad hoc Parser. Dabei zeigen wir, dass wir viele der betrachteten Parser lösen und komplexe geschachtelte Grammatiken durch Inferenz automatisch erkennen können. Die vorliegende Arbeit erweitert nicht nur die Möglichkeiten eines Refinement Typsystems, sondern verbessert auch das Panini-System erheblich, sodass es eine größere Anzahl von Grammatiken generieren kann.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

TU Bibliothek Your knowledge hub WIEN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Abstract

Parsing strings is a common programming task often performed in an ad hoc manner without formally defining an input grammar. Inferring such grammars provides numerous benefits, and the existing Panini system leverages refinement types to automatically synthesize input grammars for ad hoc parsers. However, its current approach is limited by the underlying λ_Σ calculus, which lacks polymorphism and algebraic data types, restricting its ability to infer many useful grammars.

To address these limitations, we extend the λ_Σ calculus with these missing features, resulting in the λ_Σ^+ calculus, and introduce a novel inference rule for data type destruction. We categorize the nested grammar structures common in ad hoc parsers, focusing on those arising from *split* operations and the resulting list and tuple algebraic data types. To synthesize such grammars, we introduce an approach to combine the nested grammars within a data structure to an overall grammar.

We integrate these enhancements into the Panini system and evaluate them on real-world ad hoc parsers, demonstrating the ability to synthesize complex nested grammars in many cases. This work not only extends the capabilities of a refinement type system in the context of grammar synthesis, but also significantly improves the Panini system, enabling it to synthesize a broader range of grammars.

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Contents

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 1

Introduction

Refinement types extend type systems with logical constraints. These constraints are used to describe precise subsets of values of some base type as well as model relationships between values. For example, we can define the type of positive numbers as a subset of the integers,

$$\texttt{PosInt} : \{v : \mathbb{Z} \mid v > 0\}.$$

Additionally, refinement types allow us to specify relations for dependent function types, where an uninterpreted function in the refinement logic is used to model that relationship, as for example in the length function for strings:

$$\texttt{length} : (s : \mathbb{S}) \to \{v : \mathbb{Z} \mid v = \mathit{length}(s)\}$$

Suitable practical use cases for those types are avoiding division by zero errors by restricting the input of a division operation to nonzero numbers, utilizing for example the introduced `PosInt` type, or avoiding array out-of-bounds errors by restricting the input to integer values less than the array length.

First proposed by Freeman and Pfenning [12] to statically detect more kinds of runtime errors, refinement type systems have been used to ensure data validity (e.g., ensuring a list is non-empty or an integer is positive [46]), verify security properties [3], and automatically proving certain correctness properties at compile-time [8]. A well known example of a refinement type system integrated into a general-purpose programming language is the Liquid Haskell program verifier [42] which has been successfully used to, e.g., verify security policies [31] or discover bugs in data type implementations [41].

By integrating these constraints within the type system, refinement types can also improve code documentation and understanding, as the types themselves convey more information about the expected behavior of functions and data structures.

Furthermore, refinement types have been used for theorem proving [17] or to synthesize source code from just type definitions [30].

Additionally, they can be used in the background without exposing the additional complexity to the programmer. This enables advanced correctness checks or gathering more information about a program, while still allowing a programmer to write code in a general purpose programming language without an integrated refinement type system. For example, they can be used to verify database access checks, where only the access policies need to be defined according to the refinement type system [22].

One recent novel use of refinement types, that falls in this category, has been in the synthesis of string grammars, particularly as part of the Panini system [34, 35], which is the context for the present thesis.

1.1 Panini

The Panini system uses refinement types to infer input grammars for *ad hoc parsers* [35]. A grammar is a formal definition of the values an input string can assume. This definition can be achieved by using formalized parsing techniques, like combinator frameworks [23] or parser generators [28, 44], or by manually writing down the input grammar, e.g., as a regular expression. However, in real-world programs, even though parsing is a common occurrence in software engineering, this formal definition is mostly omitted. Most parsing is done in an ad hoc manner, utilizing common string operations like `split`, `trim`, and many others [36].

Having a formal grammar for a string parser offers several benefits over ad hoc parsing. A formal grammar provides a clear, structured, and well-defined set of rules. This approach reduces ambiguity, making the parser more robust and easier to debug, as the rules are explicitly stated rather than improvised. Additionally, grammars can increase code understanding as they provide an additional representation of the current functionality [13]. A formal grammar has benefits for test generation and debugging, as examples of valid inputs can be generated automatically. This approach is used in *grammar-based fuzzing* [47]. In contrast, ad hoc parsing, which relies on informal rules, can lead to errors, harder-to-maintain code and unpredictable behavior as the complexity of the input increases [24]. Being able to statically infer such a formal grammar is therefore helpful in multiple ways. Not only does it give more information to the programmer writing an ad hoc parser, but it can also be used to, for example, track changes to the input grammar over multiple versions of the source code, as well as notify the programmer of possible unforeseen changes in the grammar due to a change in the source code. Furthermore, inferred grammars can be used in software testing for white box fuzzing and generating useful example inputs. The general idea of the Panini approach is visualized in Figure 1.1 and is as follows:

1. The relevant program parts concerning the manipulation of strings are extracted and translated to an intermediate programming language called λ_Σ. This language is a small λ-calculus in A-normal form (ANF) and is only used to synthesize the (incomplete) refinement type information of the input string.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

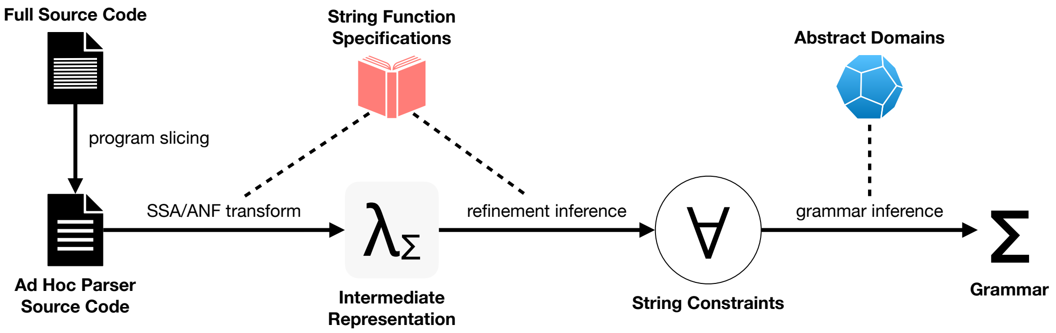

Figure 1.1: The complete process of the Panini system [34].

2. The refinement type system delivers a verification condition, which is a logical constraint generated from the λ_Σ program. If this constraint is valid, it follows that the program is well-typed [27].

3. The VC ranging over an input string is technically already a grammar for that string. To transform it into a *useful* grammar, the system uses abstract interpretation over the logical constraint (cf. section 2.2).

The Panini system can automatically infer regular grammars representing the input constraints for many real-world ad hoc parsers. However, the current implementation of the Panini system has several shortcomings, which we aim to address. Our work mainly concerns the intermediate representation λ_Σ (cf. section 2.3).

1.2 Problem Statement

One current shortcoming of the Panini approach is that the intermediate λ_Σ-calculus is not powerful enough to handle type and refinement polymorphism, nor does it support algebraic data types (ADTs) like tuples or lists. Type polymorphism refers to the classic parametric polymorphism known in many programming languages, i.e., allowing for parameterized types in functions or data types, whereas refinement polymorphism means the parameterization of the refinements by additional predicates. For example, a function that returns the maximum of two values should preserve any refinement of its input values in its output:

$$\texttt{max} : \forall\alpha.\forall p(\alpha).\{x : \alpha \mid p(x)\} \to \{y : \alpha \mid p(y)\} \to \{z : \alpha \mid p(z)\}$$

In this example, the type parameter is α and the refinement parameter is $p : \alpha \to \mathbb{B}$. Both of those parameters can be unknown when the function `max` is defined, and are either inferred or manually declared at any call-site of `max`.

Polymorphism together with support for ADTs would enable us to infer more types for more kinds of parsing programs. For example, the Python code snippet in Figure 1.2

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

```
xs = map(int, s.split(’,’))
```

```
s ∈[0-9]+(.[0-9]+)*
```

Figure 1.2: String Parser with polymorphism and algebraic data types, and a simplified input grammar.

has a given input grammar where the input string can only be integers, separated by a comma. Currently, the Panini system cannot infer this grammar.

In this example (see Figure 1.2), we have type polymorphism in the form of the map function and algebraic data types in the form of lists; neither can yet be expressed in λ_Σ. As these are common operations, especially the `split` function [36], extending λ_Σ to support those cases would enable Panini to synthesize grammars for many more real-world parsers.

Given that we already know that those extensions are vital to increase the utility of λ_Σ [35], an additional challenge arises: We need to evaluate how the extended λ_Σ interacts with real-world ad hoc parsers. The challenge is to determine the kinds of ad hoc parsers that the extension can successfully synthesize grammars from, given the many possible variations in these parsers' design. Critical factors also include the specific programming constructs ad hoc parsers may employ, such as recursion, conditionals, polymorphism, or custom data structures, and how these constructs interact within our new additions. The aim is to identify limitations imposed by the calculus on these features, as certain combinations may lead to failure or inefficiency in grammar synthesis, or may require additional type annotations by the user.

1.3 Research Questions

The aim of this work is to extend the λ_Σ-calculus of the Panini system with type and refinement polymorphism and ADTs while preserving full refinement type inference with minimal annotations and enabling grammar inference for strictly more programs. We will call this extended language λ_Σ^+. As shown in Figure 1.2, those additions are a necessary requirement to represent many common ad hoc parsers. We also aim to identify the types of parsers for which the extended λ_Σ^+ can synthesize an input string grammar.

To this end, we propose the following research questions:

RQ1 How can the λ_Σ calculus be extended to support polymorphic operations over algebraic data types, while preserving its ability to synthesize input string grammars?

RQ2 For what kind of programs can we synthesize input string grammars with λ_Σ^+?

To be able to synthesize input string grammars, the refinement λ-calculus needs to be able to infer types completely. As the Panini system is supposed to automatically translate unrefined (and possibly untyped) source code into λ_Σ, we cannot assume that any type

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

annotations are present. This is a major difference to regular applications of refinement types, where usually at least top level type annotations are required for the type system to be relatively complete [6].

Therefore, apart from the polymorphic operations that we need to add to λ_Σ, we also need to find suitable inference rules for ADTs in a refinement setting, without requiring type annotations, which to our knowledge has not been done before. Additionally, other concerns specific to grammar solving need to be discussed. Considering the example in Figure 1.2, every element of the list holding the result of `split` must conform to the same grammar constraint, i.e., each element needs to be a string of digits. This grammar is *homogenous*, as every element of the ADT has the same constraint. However, this does not need to be the general case. A list of strings could require different grammars for each of its elements. Those *heterogeneous* grammars need to be handled differently, and we expect them to be more complex.

Overall, the research questions dictate the following steps:

1. We will identify typing rules for polymorphism according to relevant literature.
2. We will construct suitable inference rules for algebraic data types without any pre-existing type annotations. Specifically, type deconstruction, i.e., accessing the values of an ADT, as construction is simply a polymorphic function returning the data type.
3. We will then add those rules into the source code of the Panini system, where we also need to check if those rules do not alter the current functionality. However, as for our changes, we only add new constructs in λ_Σ^+; we do not expect there to be any regressions regarding the current capabilities of Panini.
4. We also need to investigate out how the solving for a grammar could work for homogenous and heterogeneous grammars in ADTs. If this is possible, it will only be done exemplary, as handling this issue in a general case would be out of scope for this work. However, we still want to be able to show how a grammar can be constructed from an ADT.
5. Finally, we evaluate λ_Σ^+ with a suitable set of real-world parsers, so that we can identify common patterns and how they affect the ability of Panini with λ_Σ^+ to synthesize the string grammars.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 2

Background

To give the necessary background for this work, we will first give a deeper introduction to refinement types, liquid types and the Fusion algorithm. We explain the Panini approach to grammar solving and go into detail about the current state of the intermediate representation λ_Σ. Lastly, we give an overview of related work.

2.1 Refinement Types

As shown in chapter 1, refinement types restrict the values of a type by combining a basic type with a *logical predicate*. Every member of that type needs to satisfy this predicate. Refinement types are a restricted case of dependent types, where the logical predicates cannot be arbitrary expressions, making refinement types a subset of dependent types [38].

Refinement type checking will produce verification condition (VC) constraints [11, 15]. An example for those constraints would be those in Figure 2.2. In general, those are constructed to be in a decidable subset of SMT-solvable logic [17]. In our case, the refinement predicates are drawn from the quantifier-free theory of linear arithmetic and uninterpreted functions (QF_UFLIA), extended with a theory of string operations [34].

For simple examples, this approach is comparable to the classical Floyd-Hoare logic, where pre- and path-conditions are assumed, and the post conditions asserted [6]. The

```
abs : int → { v : int | 0 ≤ v }
abs = λ(x:int).
      let p = 0 ≤ x in
      if p then x else −x
```

Figure 2.1: Simple refinement example in λ_Σ syntax.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

$$\begin{aligned} \forall n.\ (0 \leq n) \Rightarrow \forall v.\ v = n \quad & \Rightarrow 0 \leq v \quad \wedge \\ (0 \nleq n) \Rightarrow \forall v.\ v = 0 - n \quad & \Rightarrow 0 \leq v \end{aligned}$$

Figure 2.2: VC for the example in Figure 2.1

validity of those verification conditions can then be checked by an SMT-solver and if the verification condition is valid, this implies the correctness of the checked or inferred types [27].

A simplified example of checking a refinement type can be seen in Figure 2.1. According to the type annotation, the resulting type of integer values grater or equal than zero is our post condition, which needs to be implied by all pre and path conditions. The resulting VC of a hypothetical type checking algorithm can be seen in Figure 2.2 and can be trivially checked by an SMT-solver.

SMT-solvers have become quite feasible in practice [2] and this enables refinement types to verify properties of polymorphic, higher order programs. Examples are Liquid Haskell [42], Typescript [43] or Rust [21]. Many examples specifically use the concept of liquid types, and is explained next.

2.1.1 Liquid Types and Predicate Abstraction

To be able to solve refinement types efficiently, and to be able to encode unknown relationships in not yet known refinements, a concept called liquid types (Logically Qualified Data Types) is oftentimes used [32].

This concept has been introduced to solve a fundamental issue that refinement types have, namely to enable inference, and therefore reduce the number of possibly complicated type annotations needed for a type checker to verify a program.

To the number of needed type annotations, so-called “liquid type variables” [32] are used to encode unknown refinements over a set of variables. Additional names for this concept are “Horn variables” [17] or “κ variables” [6], which is the term we use in our work.

As the typing rules for liquid type systems produce verification conditions containing κ variables, those VCs now cannot be sent directly to an SMT-solver. Those variables need to be replaced with concrete predicates. To find these predicates, an approach called predicate abstraction is used. To this end, a set of qualifiers is assumed, or in practice extracted from the given program. Those qualifiers are a set of boolean predicates over the variables defined by a κ variable, as well as fixed values. For example, $\kappa(x, y)$ can have the qualifiers $x \geq 0$ or $x = y$. A fixpoint computation is then used to identify the strongest conjunction of qualifiers that satisfy the given VC for each κ [16]. In general, there are many techniques to solve those Horn constraints [4], but for our specific case this is an implementation detail of the Panini system. And we can just assume an oracle giving us concrete refinements to try for those κ variables.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

```
      abs : {x: int} -> { v : int | κ(x,v) }
```

$$\begin{aligned} \forall n.\ (0 \leq n) &\Rightarrow \forall v.\ v = n &&\Rightarrow \kappa(n, v) \wedge \\ (0 \not\leq n) &\Rightarrow \forall v.\ v = 0 - n &&\Rightarrow \kappa(n, v) \end{aligned}$$

Figure 2.3: Liquid type and resulting VC for the example in Figure 2.1 without a given type signature.

If we go back to the `abs` example in Figure 2.1 but remove the give type signature and we therefore want to infer the concrete refinement, the instantiated liquid type with a κ variable and the therefore resulting VC can be seen in Figure 2.3. For this simple example, we can give multiple valid assignments for κ, e.g., just a simple *true* would be enough in this case. Also, $n \leq v$ would be valid. And of course the previous annotation given in Figure 2.1, i.e., $0 \leq v$ is still a valid assignment.

However, to get to a more meaningful instantiation of this refinement, i.e., not just refined to *true*, the return value needs to be used in some way. For type systems in general, function inputs need to be covariant, i.e., the actual input needs to be a subtype of the expected input. For example, if the return of `abs` is used in a function that only takes non-negative integers as an input, the type $\{v : int \mid \kappa(n, v)\}$ needs to be a subtype of the non-negative integers. Subtyping for refinement types means, apart from underlying unrefined type, that the predicate of the subtype needs to imply the predicate of the supertype. In this example, this implication is $\kappa(n, v) \implies 0 \leq v$. For the previously given valid assignments for $\kappa(n, v)$, this new overall constraint can only be valid for the assignment $0 \leq v$.

Those placeholder κ variables and the constraints they need to satisfy due to the application of subtyping rules allow the automated inference of valid concrete refinements.

The general approach of predicate abstraction is already useful in practice. However, there is an additional approach, the Fusion algorithm, which further reduces the number of needed type annotations.

2.1.2 Fusion

Another important enabler of the Panini system is the Fusion algorithm, introduced by Cosman and Jhala [6].

The idea is to be able to eliminate κ variables that can be replaced with concrete refinements, constructed using the scoped structure of the verification condition, which is otherwise lost during the predicate abstraction. Additionally, as those refinements form the "strongest" refinements, Fusion allows for type inference with less type annotations, and requires only top-level annotations or annotations for cyclic (recursive) dependencies.

This is indispensable for our case, as we want to be able to synthesize a string grammar automatically. As the idea of the Panini system is to generate code in the intermediate

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

```
assert s[0] == "a"
```

$$\begin{array}{l} \boldsymbol{\lambda}(s : \mathbb{S}). \\ \quad \textbf{let}\ x = \text{charAt}\ s\ 0\ \textbf{in} \\ \quad \textbf{let}\ p = \text{eqChar}\ x\ \text{‘a’}\ \textbf{in} \\ \quad \text{assert}\ p \end{array}$$

$$\begin{array}{l} \forall s.\ \kappa(s) \Rightarrow \\ \quad 0 < |s| \land \forall x.\ x = s[0] \Rightarrow \\ \qquad \forall p.\ (p \Leftrightarrow x = \text{‘a’}) \Rightarrow \\ \qquad\quad p \end{array}$$

Figure 2.4: A simple Python expression (left) and the equivalent λ_Σ program (middle) with an incomplete verification condition (right) for its inferred type $\{s : \mathbb{S} \mid \kappa(s)\} \to \mathbb{1}$[34].

λ_Σ language from an un-typed source language, no type annotations but the top level annotation with a hole for the string input can be assumed.

Additionally, the approach is relatively complete, meaning that without recursion, if there exist type annotations for intermediate binders, that would allow a program to be type checked, Fusion will be able to find them. As predicate abstraction is computationally expensive, being able to eliminate some κ variables significantly speeds up the VC solving. Solving those verification constraints is still in ExpTime, but Fusion is able to avoid the exponential blowup that is usually introduced by let chains.

2.2 Grammar Solving

Without loss of generality, every parsing program can be assumed to have a top-level type signature of $\{s : \mathit{string} \mid \kappa(s)\} \to \mathbb{1}$ where $\mathbb{1}$ is the unit type and $\kappa(s)$ is the unknown refinement for the parser's input string. Unfortunately, the standard approaches to find a solution for $\kappa(s)$ (see above) are insufficient if it is desired that the refinement accurately reflect the parser's input grammar.

An example of such a parser can be seen in Figure 2.4, where a simple parser is translated into λ_Σ and the resulting VC is shown. This parser expects a string of at least one character and that the first character is always "a". The constraints of this grammar `'a'.*` are clearly observable in the resulting VC

The key insight for the Panini system is that the resulting VC of such a parser will always be of the form $\forall s.\kappa(s) \implies \varphi$ and that φ is technically already a complete grammar for s. Assigning $\kappa(s) \mapsto \varphi$ also trivially validates the VC. But, impractically, φ is in size and complexity similar to the original program. To have a human-readable form, Panini uses abstract interpretation on φ to extract a quantifier-free predicate that is semantically equivalent to φ. The domain of this abstract interpretation for strings are regular expressions, and the resulting grammar is given as a *POSIX* Extended Regular Expression.

Once this grammar is extracted, we can use it to verify the VC and if it is valid, the grammar has been successfully synthesized. This approach allows Panini to practically

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

infer refinement types without any type annotations (except for recursive functions).

2.3 The λ_Σ-calculus

2.3.1 Syntax

The syntax of λ_Σ in Figure 2.5 is relatively straightforward and it is comparable to similar languages [32, 6, 17]. It is a λ-calculus in A-normal form (ANF). This restriction is necessary for liquid types, as intermediate expressions must be bound to variables, so that their inferred type information can be used [32].

However, notably missing from λ_Σ are polymorphic operations, a feature oftentimes included in the other previously mentioned examples.

In general, λ_Σ is not intended to be written and run, but used as a practical tool for static analysis. Therefore, convenience features, that might be common in other languages, are missing.

The *Terms* in Figure 2.5 contain only common operations, i.e., branching, recursion, function abstraction and application, as well as a let binding, to allow λ_Σ to be in ANF.

As there is no polymorphism in λ_Σ, the *Types* are only composed of refined base types and function types.

Figure 2.5 also shows the *Predicates* which describe the expressiveness of the refinements. As mentioned, those logical constraints on a type need to be in QF_UFLIA. Those *Predicates* are then used by the Typing Rules below, to generate the *Constraints*, which results in a verification condition for the overall inference result.

2.3.2 Typing Rules

The type/constraint synthesis rules are given as classical inference rules, with premises that need to be fulfilled and a resulting conclusion for those premises. In generall, inference in the λ_Σ context is signified by $\Gamma \vdash e \nearrow t \dashv c$, where Γ is the current context that holds existing variables and can also be filled by some rules, e.g., Syn/Lam. The term e together with the premises of a rule help to determine the resulting type t and constraint c. This constraint is oftentimes generated by an application of a subtyping rule (Figure 2.6), signified by $t_1 \leqslant t_2 \dashv c$, where the constraint c represents that t_1 is a subtype of t_2. The rules for template generation (Figure 2.6), which introduce κ variables for unknown refinements, are signified by $\Gamma \vdash t \rhd \hat{t}$, where the refinements of the type t are replaced with κ variables in $\hat{t}$.

Compared to other approaches based on liquid types, where only type checking rules or a bidirectional typing approach [9] with some synthesis rules [30] are used, the λ_Σ typing rules only utilize synthesis rules. This is due to the requirements of the Panini system, as we cannot rely on type annotation and the in section 2.2 explained approach to solving for a string grammar. The type checking step can be handled by the subtyping

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

$$
\begin{array}{rcclll}
\textbf{Values} & v & ::= & x, y, z, \ldots & & \text{variables} \\
 & & \mid & \ldots\text{varies}\ldots & & \text{constants} \\
 & & & & & \\
\textbf{Terms} & e & ::= & v & & \text{value} \\
 & & \mid & e\ v & & \text{application} \\
 & & \mid & \boldsymbol{\lambda}(x : b).\ e & & \text{abstraction} \\
 & & \mid & \textbf{let}\ x = e_1\ \textbf{in}\ e_2 & & \text{binding} \\
 & & \mid & \textbf{rec}\ x : t = e_1\ \ e_2 & & \text{recursion} \\
 & & \mid & \textbf{if}\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2 & & \text{branch} \\
 & & & & & \\
\textbf{Base Types} & b & ::= & \mathbb{1} \mid \mathbb{B} \mid \mathbb{Z} \mid \mathbb{Ch} \mid \mathbb{S} & & \\
 & & & & & \\
\textbf{Types} & t & ::= & \{x : b \mid p\} & & \text{refined base} \\
 & & \mid & (x : t_1) \to t_2 & & \text{dependent function} \\
 & & & & & \\
\textbf{Predicates} & p & ::= & \mathsf{true} \mid \mathsf{false} & & \text{Boolean constants} \\
 & & \mid & p_1 \wedge p_2 \mid p_1 \vee p_2 & & \text{connectives} \\
 & & \mid & p_1 \Rightarrow p_2 \mid p_1 \Leftrightarrow p_2 & & \text{implications} \\
 & & \mid & \neg p & & \text{negation} \\
 & & \mid & w_1 = w_2 \mid w_1 \neq w_2 & & \text{(in)equality} \\
 & & \mid & w_1 < w_2 \mid w_1 \leq w_2 & & \text{arithmetic comparison} \\
 & & \mid & w \in \mathrm{RE} & & \text{regular language membership} \\
 & & \mid & \kappa(\overline{v}) & & \kappa\ \text{application} \\
 & & \mid & \exists(x : b).\ p & & \text{existential quantification} \\
 & & & & & \\
\textbf{Expressions} & w & ::= & v & & \text{value} \\
 & & \mid & f(\overline{w}) & & \text{function} \\
 & & & & & \\
\textbf{Constraints} & c & ::= & p & & \text{predicate} \\
 & & \mid & c_1 \wedge c_2 & & \text{conjunction} \\
 & & \mid & \forall(x : b).\ p \Rightarrow c & & \text{universal implication}
\end{array}
$$

Figure 2.5: Syntax of λ_Σ terms, types, and refinements [34].

$$\boxed{t_1 \leqslant t_2 \dashv c} \quad \textbf{Subtyping}$$

$$\frac{}{\{\nu_1 : b \mid p_1\} \leqslant \{\nu_2 : b \mid p_2\} \dashv \forall(\nu_1 : b).\ p_1 \Rightarrow p_2[\nu_2 := \nu_1]}\ \textsc{Sub/Base}$$

$$\frac{s_2 \leqslant s_1 \dashv c_i \qquad t_1[x_1 := x_2] \leqslant t_2 \dashv c_o}{(x_1 : s_1) \rightarrow t_1 \leqslant (x_2 : s_2) \rightarrow t_2 \dashv c_i \wedge ((x_2 :: s_2) \Rightarrow c_o)}\ \textsc{Sub/Fun}$$

$$(x :: t) \Rightarrow c \overset{\text{def}}{=} \begin{cases} \forall(x : b).\ p[\nu := x] \Rightarrow c & \text{if } t \equiv \{\nu : b \mid p\}, \\ c & \text{otherwise.} \end{cases}$$

$$\boxed{\Gamma \vdash t \rhd \hat{t}} \quad \textbf{Template Generation}$$

$$t \rhd \hat{t} \overset{\text{def}}{=} \varnothing \vdash t \rhd \hat{t}$$

$$\frac{\kappa \text{ is a fresh variable of sort } b \times \overline{t}}{\overline{x : t} \vdash \{\nu : b \mid p\} \rhd \{\nu : b \mid \kappa(\nu, \overline{x})\}}\ \textsc{Kap/Base}$$

$$\frac{\Gamma \vdash t_1 \rhd \hat{t}_1 \qquad \Gamma[x \mapsto t_1] \vdash t_2 \rhd \hat{t}_2}{\Gamma \vdash (x : t_1) \rightarrow t_2 \rhd (x : \hat{t}_1) \rightarrow \hat{t}_2}\ \textsc{Kap/Fun}$$

Figure 2.6: Subtyping and template generation rules for λ_Σ.

rules. I.e., if we want to check type t for a term, we infer a type $\hat{t}$ with a constraint c for that term. This constraint in conjunction with the subtyping constraint $\hat{t} \leqslant t \dashv \hat{c}$ is then the overall constraint of a type checking step.

We refer the interested reader to Schröder and Cito [35] for a detailed explanation of each typing rule. To showcase their application, we now give a simple example, similar to Figure 2.4. In this previous example, we can see the resulting simplified VC of the application of the rules Syn/Lam, Syn/Let and Syn/App.

However, due to the iterative application of the typing rules, those constraints become convoluted fast. To be able to better show the interplay of the typing rules, the reduced example can be seen in Figure 2.8. Note that compared to the first example, the shown VC is actually the one produced by the λ_Σ typing rules. For this example two rules are relevant, Syn/Lam, as the overall expression is a lambda expression, and Syn/App, as *charAt* is a function that s and 0 are applied to. To discuss this example, we need the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

$$\boxed{\Gamma \vdash e \nearrow t \dashv c} \quad \textbf{Type/Constraint Synthesis}$$

$$\frac{\Gamma(x) = t}{\Gamma \vdash x \nearrow \mathsf{self}(x, t) \dashv \mathsf{true}}\ \textsc{Syn/Var} \qquad \frac{\mathsf{prim}(c) = t}{\Gamma \vdash c \nearrow t \dashv \mathsf{true}}\ \textsc{Syn/Con}$$

$$\frac{\Gamma \vdash e \nearrow (y : t_1) \to t_2 \dashv c_e \qquad \Gamma \vdash x \nearrow t_x \qquad t_x \leqslant t_1 \dashv c_x}{\Gamma \vdash e\ x \nearrow t_2[y := x] \dashv c_e \land c_x}\ \textsc{Syn/App}$$

$$\frac{\tilde{t}_1 \triangleright \hat{t}_1 \qquad \Gamma[x \mapsto \hat{t}_1] \vdash e \nearrow t_2 \dashv c_2}{\Gamma \vdash \boldsymbol{\lambda}(x : \tilde{t}_1).\ e \nearrow (x : \hat{t}_1) \to t_2 \dashv (x :: \hat{t}_1) \Rightarrow c_2}\ \textsc{Syn/Lam}$$

$$\frac{\begin{array}{l} \Gamma \vdash e_1 \nearrow t_1 \dashv c_1 \\ \Gamma[x \mapsto t_1] \vdash e_2 \nearrow t_2 \dashv c_2 \qquad t_2 \triangleright \hat{t}_2 \qquad t_2 \leqslant \hat{t}_2 \dashv \hat{c}_2 \end{array}}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \nearrow \hat{t}_2 \dashv c_1 \land ((x :: t_1) \Rightarrow c_2 \land \hat{c}_2)}\ \textsc{Syn/Let}$$

$$\frac{\begin{array}{llll} \tilde{t}_1 \triangleright \hat{t}_1 & \Gamma[x \mapsto \hat{t}_1] \vdash e_1 \nearrow t_1 \dashv c_1 & & t_1 \leqslant \hat{t}_1 \dashv \hat{c}_1 \\ & \Gamma[x \mapsto t_1] \vdash e_2 \nearrow t_2 \dashv c_2 & t_2 \triangleright \hat{t}_2 & t_2 \leqslant \hat{t}_2 \dashv \hat{c}_2 \end{array}}{\Gamma \vdash \mathbf{rec}\ x : \tilde{t}_1 = e_1\ \ e_2 \nearrow \hat{t}_2 \dashv ((x :: \hat{t}_1) \Rightarrow c_1 \land \hat{c}_1) \land ((x :: t_1) \Rightarrow c_2 \land \hat{c}_2)}\ \textsc{Syn/Rec}$$

$$\frac{\begin{array}{llll} \Gamma \vdash x \nearrow \mathbb{B} & \Gamma \vdash e_1 \nearrow t_1 \dashv c_1 & t_1 \triangleright \hat{t} & t_1 \leqslant \hat{t} \dashv \hat{c}_1 \\ & \Gamma \vdash e_2 \nearrow t_2 \dashv c_2 & & t_2 \leqslant \hat{t} \dashv \hat{c}_2 \end{array}}{\Gamma \vdash \mathbf{if}\ x\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \nearrow \hat{t} \dashv (x = \mathsf{true} \Rightarrow c_1 \land \hat{c}_1) \land (x = \mathsf{false} \Rightarrow c_2 \land \hat{c}_2)}\ \textsc{Syn/If}$$

Figure 2.7: Type synthesis rules for λ_Σ.

$$\mathit{charAt} : (s : \mathbb{S}) \to \{z : \mathbb{Z} \mid z \geq 0 \land z < |s|\} \to \{t : \mathbb{Ch} \mid t = s[z]\}$$

$$\begin{array}{ll} \boldsymbol{\lambda}(s : \mathbb{S}). & \forall s.\ \kappa(s) \Rightarrow \\ \quad \text{charAt}\ s\ 0 & \quad (\forall s_1.\ \kappa(s_1) \land s = s_1 \implies \mathit{true}) \land \\ & \quad (\forall z.\ (z = 0) \Rightarrow z \geq 0 \land z < |s|) \end{array}$$

Figure 2.8: Definition of *charAt* and reduced version of Figure 2.4, with a complete VC generated by λ_Σ

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

context

$$t_{\mathit{charAt}} := (s : \mathbb{S}) \to \{z : \mathbb{Z} \mid z \geq 0 \land z < |s|\} \to \{t : \mathbb{Ch} \mid t = s[z]\} \tag{2.1}$$

$$\Gamma := [\texttt{charAt} \mapsto t_{\mathit{charAt}}] \tag{2.2}$$

$$\Gamma_\lambda := \Gamma[s \mapsto \{s : \mathbb{S} \mid \kappa(s)\}] \tag{2.3}$$

as defined by `charAt` and initialized by Syn/Lam. As the rules are recursive, we will discuss the deepest nested rule Syn/App first. The value s is applied to *charAt* and according to Syn/App, s needs to be a subtype of the first parameter of *charAt*. This results in the constraint

$$c_s := \forall (s_1 : \mathbb{S}).\kappa(s_1) \land s = s_1 \implies \mathit{true}$$

for the subtyping relation between s and the (renamed) input parameter s_1. As the constraints generated by gathering the variables s and *charAt* using the rule Syn/Var are all present in the context (and resulting in a *true* VC), the resulting derivation tree is

$$\dfrac{\begin{array}{c}\Gamma_\lambda \vdash \text{charAt} \nearrow t_{\mathit{charAt}} \dashv \mathit{true} \quad \Gamma_\lambda \vdash s \nearrow \{s : \mathbb{S} \mid \kappa(s)\} \\ \{s : \mathbb{S} \mid \kappa(s)\} \leqslant \{s : \mathbb{S} \mid \mathit{true}\} \dashv c_s\end{array}}{\Gamma_\lambda \vdash \text{charAt s} \nearrow \{z : \mathbb{Z} \mid z \geq 0 \land z < |s|\} \to \{t : \mathbb{Ch} \mid t = s[z]\} \dashv c_s}\ \textsc{Syn/App}$$

and the resulting VC is simply c_s. In the next step, the value 0 gets applied to the remaining type of charAt, and the Syn/App is used again, with a similar derivation tree. By the definition of Syn/App, the resulting VC of the nested Syn/App rule c_s is used in conjunction with the subtyping constraint

$$c_z := \forall z.\ (z = 0) \Rightarrow z \geq 0 \land z < |s|\,,$$

for the input parameter z as the overall VC. And the overall inference for the expression charAt s 0, without giving the complete derivation tree, is:

$$\Gamma_\lambda \vdash \text{charAt } s \ 0 \nearrow \{t : \mathbb{Ch} \mid t = s[0]\} \dashv c_s \land c_z$$

Note that the resulting type already has the value 0 replaced, as defined in Syn/App. The Syn/Lam rule uses the hole instantiation rules to generate a with a κ variable refined type of its input parameter s. This type is then used to prepare the previously introduce context Γ_λ and is used to infer the nested function applications. Overall, we have the following derivation tree:

$$\dfrac{\begin{array}{c}(s : \mathbb{S}) \rhd \{s : \mathbb{S} \mid \kappa(s)\} \qquad \Gamma_\lambda := \Gamma[s \mapsto \{s : \mathbb{S} \mid \kappa(s)\}] \\ \Gamma_\lambda \vdash \text{charAt } s \ 0 \nearrow \{t : \mathbb{Ch} \mid t = s[z]\} \dashv c_s \land c_z\end{array}}{\Gamma \vdash \boldsymbol{\lambda}(s : \mathbb{S}).\text{charAt } s \ 0 \nearrow \{s : \mathbb{S} \mid \kappa(s)\} \to \{t : \mathbb{Ch} \mid t = s[z]\} \dashv \forall s.\ \kappa(s) \Rightarrow c_s \land c_z}\ \textsc{Syn/Lam}$$

The Syn/Lam rule returns the implication

$$\forall s.\ \kappa(s) \Rightarrow c_s \land c_z$$

as its VC, which can also be seen fully in Figure 2.8. This VC relates the κ variable with the input refinement of *charAt*, therefore, to validate this VC a predicate with at least $|s| > 0$ needs to be chosen for κ. In the Panini system, this grammar can be extracted from this result as explained in section 2.2.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

2.4 Related Work

In this section, we give an overview of the related work, highlighting not only the previously mentioned fundamental work, but also different applications of refinement types as well as more directly related work. Refinement types in general have been introduced by Freeman and Pfenning [12].

Xi and Pfenning [46] use refinement types to show the absence of array out-of-bounds errors in higher order programs. Dunfield [8] and Bengtson et al. [3] respectively show that refinement types can be used to validate the correctness of data structure implementations and the correctness of cryptographic protocol implementations.

However, those approaches required a high number of annotations, upwards of 10% of all source code lines [32]. Liquid types (cf. subsection 2.1.1) aimed to partially remedy this issue [32]. Due to the reduced annotation burden, liquid types are used extensively, and most of our related work builds on this concept.

Liquid types have been implemented for many programming languages, the most extensively documented approach being Liquid Haskell [42].[1] Compared to previous implementations in *OCaml*[32], Liquid Haskell needed to adapt liquid types to the lazy evaluation nature of the Haskell language. This lazy nature can lead to unsoundness in the refinement type system, as diverging values can lead to valid VCs for unsound implementations. This problem is solved by labeling types as either divergent or finite, meaning that those values reduce. Additionally, a termination analysis is used to assign a finite label to terms, as the authors note, in practice most terms don't diverge. In their evaluation, they showed that for recursive functions, Liquid Haskell could prove termination in 96% of cases. With this extension, the authors provide a sound, precise and automated verification of the functional properties of real world Haskell code.

Different implementations of liquid types also exist for *Rust*[21], *TypeScript*[43] and *C*[33]. Those implementations also need to handle different language specific issues like mutability or function overloading.

Apart from those concrete adaptations to specific programming languages, the concept of liquid types has also been extended since its initial inception. Some of those extensions to liquid types have been summarized by Jhala and Vazou [17], which contains an introduction and exhaustive overview of liquid types.

Kawaguchi, Rondon, and Jhala [18] propose a mechanism to extend liquid types to handle complex invariants of data types. They introduce recursive and polymorphic refinements, two ideas to add refinements to either recursive data structures or to finite maps, allowing for example the refinement of the value of a map to depend on its key. This avoids using universally quantified formulas, and thus remains decidable.

A generalization of this approach is the introduction of abstract refinements by Vazou, Rondon, and Jhala [40]. This idea is also called refinement polymorphism and allows

[1] https://ucsd-progsys.github.io/liquidhaskell/

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

having abstract refinement parameters on types. This refinement parameter is a function of base types to a boolean value, and is interpreted as an uninterpreted function by the SMT solver. This allows for decidable SMT checking. Similar to the idea of recursive refinements, this parameter can be passed recursively into a data structure, enabling, e.g., a sorted list where every appended element needs to be larger than every other element in the list. Those abstract parameters can not only be explicitly defined, but also implicitly instantiated. This instantiation is again handled by the κ variables, where the implication constraints force a valid instantiation of those variables.

Another continuation to this concept of abstract refinements are bounded refinement types by Vazou, Bakst, and Jhala [39]. This allows the refinement parameters to be restricted by certain criteria, e.g., $\forall p.p(x) \Rightarrow p(x+1)$ for a parameter $p : \mathbb{Z} \to \mathbb{B}$ similar to typeclasses in Haskell. Those bounds, as they are also translated to κ variables, still are decidable by SMT solvers, and enable the specification and verification of diverse higher order abstractions.

Those extensions, while they greatly increase the capabilities of liquid types and are useful in general, seem to have limited application in our case. Complex higher order functions or data structures with non-trivial guarantees are not a relevant factor in real world ad hoc parsers. The code we observed in section 4.2 is not impacted by those features, especially as the source is mostly untyped.

Another related extension is the work by Montenegro et al. [25], and their previous work [26], where they aim to handle quantified properties on arrays. This is similar to the concept of abstract refinements, however, by allowing quantification that is restricted to a decidable subset of the theory of arrays, the values of an array can also relate to one another. This extension enables the static verification of programs that manipulate arrays by allowing type refinements to express complex invariants involving array indices, such as sortedness or uniqueness of elements. The primary contribution of their work lies in extending the expressiveness of liquid types to capture these quantified properties while still maintaining decidability. Those array properties could be useful in the context of string grammar synthesis. E.g., if the first few elements of an array need to be of some grammar g and all elements afterward need to be of another grammar g'. However, we do not jet know if this would be a common case.

This approach, similar to the original idea of liquid types, relies on a set of qualifiers, that would need to be extracted or defined beforehand. From those qualifiers suitable refinements are created, that, compared to the work by Rondon, Kawaguci, and Jhala [32], may contain some qualified assertions over arrays, while still remaining in a decidable subset of the theory of arrays. As the Fusion algorithm is used in our work to eliminate as many κ variables as possible beforehand, and we do not entirely rely on predicate abstraction with extracted qualifiers, and it is unclear if those array predicates can be combined with our approach or if it would increase the necessary annotation burden.

Independently of the verification capabilities of liquid types, there are also several concrete applications using those refinement types to achieve multiple different use cases. As our

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

work falls mainly into this category, i.e., using liquid types for a specific application, those works are also relevant for our context.

One application of liquid types is *Lifty* (liquid information flow control) by Polikarpova et al. [31], a domain specific language λ_L to enforce information flow control policies at compile time using liquid types. This calculus is significantly more limited than λ_Σ, but additionally includes specific properties to handle information flow control. A continuation of this approach, is the STORM (Security Typed Object Relation Mapper) web framework by Lehmann et al. [22]. The idea is to enrich the data model of a web application with security policies, regulating the information flow control, and then using those policies to generate a refined ORM layer. STORM verifies using Liquid Haskell, e.g., necessary access checks are made before allowing database operations according to those generated refinements. This approach is somewhat similar to our context, as the refinement typing is opaque to the user, meaning that in the regular application code those refinements are not used. For the Panini system, we also know the shape of *ad hoc* parsers and use this information to generate the refinements.

Another application of liquid types by Knoth et al. [20] is the idea to encode the resource consumption into liquid types, and use the refinement type checking to automatically verify bounds on the resource consumption of a program. As the refinements are additionally annotated with a resource potential, the proposed typing rules also need to handle this specific detail. This work is also related to the work by Polikarpova, Kuraj, and Solar-Lezama [30], with the aim to synthesize programs from refinement type specifications, as those resource constraints are used to try to find implementations satisfying those constraints in the program synthesis search space [19]. Those works are only partially related to our approach, as even though the presented typing rules are in the context of liquid types, they are highly specific to this context. Nonetheless, it is still relevant to convey the different applications of refinement types in general and specifically liquid types.

Overall, refinement types have proven to be useful in practice, especially due to the reduced annotation burden enabled by the concept of liquid types. The Panini system as explained in section 2.2 and section 2.3 also builds heavily on the concept of liquid types and in the next chapter, we will introduce the proposed extensions to λ_Σ to increase the capabilities of Panini.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 3

The λ_Σ^+-calculus

In this chapter, we extend the λ_Σ-calculus with new syntactic constructs to support Polymorphism and algebraic data types. We call this extended calculus λ_Σ^+, and we formalize additional inference rules and explore the new semantics of λ_Σ^+, illustrate the rules with examples, and discuss their implications for grammar inference. By the end of this chapter, the reader will understand how these extensions enhance the Panini system's ability to synthesize string grammars, and will gain insight into the mechanics of refinement type inference for polymorphism and algebraic data types.

To identify and create suitable rules, we have been inspired by the work of Jhala and Vazou [17], which introduces the Sprite tutorial language.[1] This language has already influenced the implementation of λ_Σ, and also guides our extension.

Type polymorphism is a standard extension of the regular typed lambda calculus [29] and commonly part of liquid type systems [32, 42, 17]. Polymorphic type inference rules for liquid type systems are often based on the original work by Rondon, Kawaguci, and Jhala [32] and this is also the case for λ_Σ^+.

In the literature we surveyed (cf. section 2.4), we could not identify many inference rules directly related to algebraic data types. This seems to have multiple reasons. Firstly, ADTs are often not relevant to the core idea of a particular system or application, and therefore omitted. Also, due to the fact that ADTs can be implemented directly into any λ calculus using the Church or Scott encoding [1], omitting ADTs is not necessarily a restriction, at least theoretically.

Overall, we have only observed specific rules for ADTs in the Sprite tutorial language. For the destruction of ADTs, those rules were checking rules and we did not identify rules to infer the destruction of ADTs. In general, the destruction of an ADT is a common part in many programming languages, e.g., *Haskell* uses the `case` keyword or pattern

[1] https://github.com/ranjitjhala/sprite-lang

TU WIEN Bibliothek Your knowledge hub

matching. For this statement, different alternative expressions are evaluated, depending on the concrete value of a data type. For example, if a list is empty or not, and each constructor has a corresponding alternative.

As we specifically aim to add ADTs to λ_{Σ}^{+}, we also need inference rules for their destruction. As mentioned in section 2.2, Panini relies on inference, so we could not use the identified checking rules. We have therefore also identified new inference rules for those cases. This also allows to potentially alter the generated VCs and have fine-grained control over how the nested values of an ADT are accessed, which is useful in the context of grammar synthesis. Those novel rules are a combination of previous work [17] as well as the existing rules in Panini, allowing refinement inference for ADT deconstruction.

To be able to explain those new additions, we start with a high level description of our extension to the λ_{Σ} calculus.

3.1 Syntax of λ_{Σ}^{+}

The syntax of λ_{Σ} has been extended in multiple places to implement our changes. Figure 3.1 contains the existing syntax of the Panini system as well as the new additions.

Terms

In total, there are four new terms that are added to the syntax. Type application and type abstraction are new terms needed for type polymorphism. Type application $e[t]v$ is applying a concrete type t to a polymorphic term e. This term can be automatically created by annotating a λ_{Σ} program with classical Hindley-Milner type inference[14, 7]. Type abstraction is similar to the λ-abstraction already present in the original syntax, and allows a term to be parameterized by an abstract type.

There are no terms for data constructors of a defined data type. In our implementation, this term is interpreted as a variable value, that is resolved to the concrete type of the data constructor, i.e., there is no difference between a (global) variable or a data constructor, as both are types that exist in the inference context. The data constructors are added to the context once a new data type is defined. This simplifies our implementation, and we can define polymorphic types as a data constructor for an ADT.

The switch statement branches a value v that needs to resolve to a data type over multiple alternatives. As ADTs are composed of alternatives, each data constructor of v has a corresponding alternative in the switch statement. Our switch statement is therefore the approach to destruct a data type and in conjunction with the data constructors allows us to create and access the added data types. This term allows us to specifically define inference rules for the ADT destruction case.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

$$
\begin{array}{rcclll}
\textbf{Terms} & e & ::= & v & & \text{value} \\
 & & \mid & e\ v & & \text{application} \\
 & & \mid & \boldsymbol{\lambda}(x : b).\ e & & \text{abstraction} \\
 & & \mid & \textbf{let } x = e_1 \textbf{ in } e_2 & & \text{binding} \\
 & & \mid & \textbf{rec } x : t = e_1 \quad e_2 & & \text{recursion} \\
 & & \mid & \textbf{if } v \textbf{ then } e_1 \textbf{ else } e_2 & & \text{branch} \\
 & & \mid & e[t]\ v & & \text{type application} \\
 & & \mid & \boldsymbol{\Lambda}\alpha.e & & \text{type abstraction} \\
 & & \mid & \textbf{switch}(v) \ \mid \ \overline{a} & & \text{switch statement} \\
 & & & & & \\
\textbf{Base Types} & b & ::= & \mathbb{1} \mid \mathbb{B} \mid \mathbb{Z} \mid \mathbb{Ch} \mid \mathbb{S} \mid \mathbf{D}(\bar{t}) & & \\
 & & & & & \\
\textbf{Types} & t & ::= & \{x : b \mid p\} & & \text{refined base} \\
 & & \mid & (x : t_1) \to t_2 & & \text{dependent function} \\
 & & \mid & \alpha & & \text{type variable} \\
 & & \mid & \forall\alpha.t & & \text{type polymorphism} \\
 & & & & & \\
\textbf{Alternative} & a & ::= & \mathbf{C}(\bar{z})\textbf{:}\ e & & \text{switch alternative} \\
 & & & & & \\
\textbf{Data Type} & \delta & ::= & \mathbf{D}, \overline{\alpha}, \overline{\mathbf{C} : t} & &
\end{array}
$$

Figure 3.1: Extended Syntax for λ_Σ^+.

Base Types

The base types are extended by data types. This base type also includes the concrete instantiation of the data type, i.e., what the concrete types of the current data type instance are. For example, for a data type list, a concrete instance would be if it is a list of integers or a list of strings.

Types

The types are extended by type variables as well as type polymorphism. This is exactly the same as in a regular polymorphic lambda calculus without refinements.

Alternatives

An alternative sets up the context for a term within a switch statement. It is composed of a data constructor, a list of variable names and a nested term. The variable names are used together with the constructor type to construct the inference context for the term

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

```
type list(#a) =
   | Nil: ∀#a.{v:list(#a) | len(v) = 0}
   | Cons: ∀#a.(x:#a) → {xs:list(#a)|true} →
         {v:list(#a) | len(v) = 1 + len(xs)}

measure len = list(#a) → int
```

Figure 3.2: A definition of a list data type and corresponding length measure in λ_Σ^+.

of the alternative, i.e., each variable corresponds to an input of the data constructor and can then be used in the nested alternative term.

Data Types

A data type is composed of a name $\mathbf{D}$, a list of type variables $\overline{\alpha}$, as well as a list of data constructor names with their corresponding types.

Figure 3.2 shows the definition of a list data type. The data type name is $\mathbf{D} :=$ list, we have a single polymorphic type parameter, and two constructors `Nil` and `Cons`. To allow for uninterpreted functions in refinements, the type for that function must be explicitly declared as a *measure*. This measure can then be used in the refinements of the returned lists. It also defines the corresponding function in the SMT solver's logic, so that the VCs can be solved.

As mentioned, the constructors `Nil` and `Cons` are variables in the inference context, and handled the same way as any other defined function. To construct a new ADT value, those constructors are then simply used like any other function. The alternative term also needs those variables, to set up the inference context for a specific constructor.

3.2 Synthesis Rules

In this section, we explain the new synthesis rules in detail. Those rules, shown in Figure 3.3, together with the subtyping rules in section 3.3 enable type synthesis for our extended λ_Σ calculus. As mentioned, the rules for polymorphism are typical and have not been altered for this work. This is not the case for the synthesis rules handling the new switch and alternative syntax for ADTs, and their inception is explained in detail.

Type Abstraction

This is the simplest new synthesis rule. If we have a term e that is wrapped in a type abstraction, and the term e synthesizes to a type t, we know that the overall type of the term synthesizes to $\forall\alpha.t$ under the constraint c that has been given by the term e.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

$$\boxed{\Gamma \vdash e \nearrow t \dashv c} \qquad \textbf{Type/Constraint Synthesis}$$

$$\frac{\Gamma \vdash e \nearrow t \dashv c \qquad \hat{t} = \forall \alpha . t}{\Gamma \vdash \mathbf{\Lambda} \alpha . e \nearrow \hat{t} \dashv c}\ \textsc{Syn/TypeAbs}$$

$$\frac{\Gamma \vdash e \nearrow \forall \alpha . t_e \dashv c \qquad z \rhd t_z}{\Gamma \vdash e[z] \ \nearrow t_e[\alpha \coloneqq t_z] \dashv c}\ \textsc{Syn/TypeApp}$$

$$\frac{\begin{gathered}\Gamma \vdash a_i \nearrow t_i \dashv (\overline{z}_i :: \overline{t^z}) \Rightarrow c_i \text{ for each } i \\ t_1 \rhd \hat{t} \qquad t_i \leqslant \hat{t} \dashv \hat{c}_i \text{ for each } i\end{gathered}}{\Gamma \vdash \mathbf{switch(}y\mathbf{)} \ \mid \ \overline{a} \nearrow \hat{t} \dashv \bigwedge (\overline{z_i} :: \overline{t_{z_i}}) \Rightarrow c_i \wedge \hat{c}_i \text{ for each } i}\ \textsc{Syn/Switch}$$

$$\frac{\begin{gathered}s = \mathit{ctor}(\Gamma, \mathbf{C(}\overline{z}\mathbf{)}, y) \qquad \Gamma' = \mathit{unapply}(\Gamma, y, \overline{z}, s) \\ \overline{t^z} = \Gamma'(\overline{z}) \qquad \Gamma' \vdash e \nearrow t \dashv c\end{gathered}}{\Gamma[y \mapsto \mathbf{D(}\overline{t}\mathbf{)}] \vdash \mathbf{C(}\overline{z}\mathbf{):} \ e \nearrow \hat{t} \dashv (\overline{z} :: \overline{t^z}) \Rightarrow c}\ \textsc{Syn/Alt}$$

Figure 3.3: Extended Type Synthesis rules for λ_Σ.

Due to the nature of type abstraction, no other constraints need to be generated. Depending on how the term is consecutively used, the new rule Syn/TypeApp in combination with the existing Syn/App and Syn/Lam rules handles the correct inference of specific refinements. This interplay is explained in section 3.4.

Type Application

The Syn/TypeApp rule is the "reverse" rule of Syn/TypeAbs, as the type abstraction gets removed from the resulting type of e. Similar to Syn/TypeAbs, the constraint does not need to be changed, as the concrete refinement inference comes into effect due to the existing Syn/App and Syn/Lam rules.

However, one additional step needs to be taken to construct the concrete type of $e[z]$. The abstract type value α bound by the type abstraction needs to be substituted in the inferred type t_e of e. This substitution is not simply a replacement with the unrefined base type z, but with a newly generated template. This is what enables the inference of concrete refinements, and is similar to the original Syn/Lam in Figure 2.7, with the difference that the instantiated template type is not added to the context, but directly added to the resulting type.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

$$
\begin{array}{lcl}
\texttt{ctor} & : & (\Gamma \times \mathbf{C} \times X) \rightarrow T \\
\hline
\texttt{ctor}(\Gamma, \mathbf{C}, y) & = & s[\overline{\alpha} := \overline{t}] \\
\quad \text{where} & & \\
\qquad \mathbf{D}(\overline{t}) & = & \Gamma(y) \\
\qquad \overline{\forall \alpha}.s & = & \Gamma(\mathbf{C})
\end{array}
$$

Figure 3.4: Method to create a monomorphic constructor function, using the current data type instance [17].

Switch and Alternative

The synthesis rules for switch and alternative will be explained together, as those rules work in a tandem to be able to synthesize the deconstruction of a data type. Those rules are arguably more complex than the other additions. We have observed checking rules for similar syntax [17], which gave us inspiration on how to handle the necessary internal details for possible inference rules. However, as inference is required, we had to adapt the premises and inner workings of those checking rules.

Considering its syntactic meaning, a **switch** statement, with its alternatives, is a combination and generalization of the **if** and **let** statements. For a **switch** statement, we have both, a branching into the different alternatives and, for those alternatives, variables that are added to the context. Compared to the rule Syn/If, the possible amount of branches is unrestricted, and compared to the rule Syn/Let, there can also be multiple variables depending on the concrete data constructor.

We therefore decided to utilize and combine those similarities into those two new rules. Specifically, splitting a switch term into two different rules is already given by the syntax and the implied semantics, as clearly the nested alternative statements are somewhat (but not entirely) independent of the overall switch statement.

We will start with a description of the Syn/Alt rule, as the Syn/Switch rule is composed of iterative applications of that rule. This rule generates VCs similarly to Syn/Let, as all variables defined by an alternative need to imply the nested VC. To be able to infer an alternative and specifically prepare the necessary context, two necessary preconditions need to be handled.

First, the concrete instance of the data type constructor under this alternative needs to be constructed. This is done using the *ctor* method (cf. Figure 3.4). This concrete instance is created using the data constructor of the alternative and the type variables of the data type that the switch statement is referring to. The *ctor* helper method fetches the concrete instance of the data type as well as the type of the constructor method of the current constructor from the context. The returned type is then the constructor type with the type variables applied, so that all quantified variables of the data type are removed.

Secondly, we need to identify all variables that the data constructor defines and add their

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

$\texttt{unapply}$	:	$(\Gamma \times X \times \overline{X} \times T) \to \Gamma$
$\texttt{unapply}(\Gamma, y, z : \overline{zs}, (x : s) \to t)$	$=$	$\text{unapply}(\Gamma[z \mapsto s], y, \overline{zs}, t[x := z])$
$\texttt{unapply}(\Gamma, y, \varnothing, t)$	$=$	$\Gamma[y \mapsto \mathit{meet}(\Gamma(y), t)]$
$\texttt{meet}$	:	$(T \times T) \to T$
$\texttt{meet}\ (\{x : b \mid p_1\}, \{y : b \mid p_2\})$	$=$	$\{x : b \mid p_1 \wedge p_2[y := x]\}$
$\texttt{meet}\ ((x : s_1) \to t_1, (y : s_2) \to t_2)$	$=$	$(x : \texttt{meet}(s_1, s_2)) \to \texttt{meet}(t_1, t_2[y := x])$
$\texttt{meet}\ (\forall \alpha . t_1, \forall \beta . t_2)$	$=$	$\forall \alpha . \texttt{meet}(t_1, t_2[\beta := \alpha])$

Figure 3.5: Methods to assign (monomorphic) constructor function parameters and result to variables in an alternative [17].

type into a modified context. This is done with the *unapply* method (cf. Figure 3.5). This method "unrolls" the concrete constructor function created by *ctor*, so that every z value of the constructor is matched to an input of the type of that constructor. Those variables are then used in the synthesis of the term e. In general, this behavior is simply the semantic meaning of a switch statement, i.e. the term of the alternative needs to be inferred with a context containing the variables of the alternative.

Additionally, the *meet* function unifies the type of y with the type returned by s after all variables are removed of the concrete constructor function. This allows refinements that are inherent to a constructor method to then be also in the current context. For example, if the constructor *Cons* returns a list $\{x : \mathit{list}(\alpha) \mid \mathit{len}(x) > 0\}$, the variable y then also has this additional refinement.

As the context is extended by new variables, the resulting constraint is similar to the Syn/Let rule, as the new variables and their types need to imply the synthesized constraints of e.

The Syn/Switch rule now needs to combine the types and constraints of each alternative. Similarly to the Syn/If rule, the constraints need to be valid in conjunction, and we need to add a subtyping constraint. In the Syn/If case, the types of the nested terms of the branches are synthesized. Then a subtyping constraint is generated for both branches, using the instantiated template type of the second branch. This enforces that types of those branches match and consequently this template type is the overall type of the **if** term. And for the Syn/Switch rule this is equivalent, as we simply use the first branch to create the template $\hat{t}$ and the synthesized types of each branch need to be a subtype of this template.

The type of an alternative expression can rely on the variables that have been added into the context during Syn/Alt, and the subtyping constraint does as well. The implication generated by Syn/Alt handles this extension of the context. Therefore, this implication needs to include the subtyping constrain $\hat{c}_i$ generated in Syn/Switch in its consequent. Those added variables are similar to the branching condition in Syn/If in the sense that the branching condition of an **if** statement also implies the nested VC for each branch.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

$$\boxed{t_1 \leqslant t_2 \dashv c} \quad \textbf{Subtyping}$$

$$\frac{}{\alpha \leqslant \alpha \dashv \mathsf{true}}\ \textsc{Sub/Abs}$$

$$\frac{s \leqslant t \dashv c}{\forall\alpha.s \leqslant \forall\alpha.t \dashv c}\ \textsc{Sub/Poly}$$

$$\frac{\begin{gathered} c_0 = \forall(\nu_1 : b).\ p_1 \Rightarrow p_2[\nu_2 := \nu_1] \\ s_i \leqslant t_i \dashv c_{s_i} \text{ for each } i \qquad c_s = \bigwedge c_{s_i} \end{gathered}}{\{\nu_1 : \mathbf{D}(\overline{s}) \mid p_1\} \leqslant \{\nu_2 : \mathbf{D}(\overline{t}) \mid p_2\} \dashv c_0 \wedge c_s}\ \textsc{Sub/Data}$$

Figure 3.6: Extended Subtyping rules for λ_Σ.

3.3 Subtyping Rules

For the new subtyping rules, the main concern is the Sub/Data rule, which is now the most involved subtyping rule. For abstract types and polymorphic types, subtyping constraints are straightforward and, as mentioned, those rules are present in most liquid type systems. The less common Sub/Data rule is specifically taken from Jhala and Vazou [17].

Abstract Types and Polymorphic Types

The generation of subtyping constraint for abstract types is trivially true. For polymorphic types, the subtyping constraint is the constraint without the bounded quantification of the polytype. This is in accordance with the meaning of a subtype and also not specific to refinement type systems.

Data Types

This is a more complicated subtyping rule compared to the previous rules in λ_Σ. For a data type to be a subtype of another, the Sub/Base rule needs to be extended. Therefore, the refinement of the subtype also needs to imply the refinement of the supertype. This implication is the same as in the rule Sub/Base. Additionally, for the subtyping relationship to be valid, each type parameter needs to be in a valid subtyping relationship. Therefore, those constraints in conjunction with the refinement implication compose the overall subtyping constraint for Sub/Data.

To complement this brief overview of the extension in λ_Σ^+, in the next section we will go into detail for some of these rules, give examples of their application and show the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

```
listAdd1 = λ(x:list(int)). Cons[int] 1 x
```

Figure 3.7: Instantiating a list data type in λ_Σ^+.

interplay of subtyping and synthesis rules.

3.4 Inference in Depth

As we have seen an overview of the changes to the λ_Σ type system, we will now give an in depth explanation of how those rules work as well as illustrate those rules with a few examples.

Figure 3.2 shows the definition of a list data type in λ_Σ^+. Notably, the constructor methods are simply polymorphic functions that return a value of the datatype. For our concrete implementation, this means that a constructor is handled exactly like a variable in the current context.

We will give two examples where we will manually apply the defined rules of λ_Σ^+. With those examples, we will show the construction as well as the deconstruction of the given data type in Figure 3.2. As polymorphism is a prerequisite for those ADTs, this is sufficient to explain all our newly added rules in depth.

Data Type Creation

To show both, polymorphism in general and subtyping for data types, we will use the example in Figure 3.7. This example is a function that takes an integer list as a value and returns a list with the value 1 appended. As already mentioned, we assume that general type inference with Hindley-Milner has already annotated the usage sites of polymorphic functions, as is shown with $\texttt{Cons}[\mathit{int}]$. This example can also be given as the following sub expressions:

$$e_0 := \texttt{Cons}[\mathit{int}] \tag{3.1}$$

$$e_1 := e_0\ 1 \tag{3.2}$$

$$e_2 := e_1\ x \tag{3.3}$$

$$e_3 := \lambda(x : \mathit{list}(\mathit{int})).e_2 \tag{3.4}$$

We need the following context for type synthesis.

$$t_{Cons} := \forall \alpha.x : \alpha \rightarrow xs : \mathit{list}(\alpha) \rightarrow \{v : \mathit{list}(\alpha) \mid \mathit{len}(v) = 1 + \mathit{len}(xs)\} \tag{3.5}$$

$$\Gamma := [\texttt{Cons} \rightarrow t_{Cons}] \tag{3.6}$$

$$\Gamma_\lambda := \Gamma[x \mapsto \{x : \mathit{list}(\{v : \mathit{int} \mid \kappa_1(v)\}) \mid \kappa_0(x)\}] \tag{3.7}$$

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Note that the context Γ_λ has already been inferred, as generally, those rules are executed recursively on the program, and the outermost rule Syn/Lam modifies this context.

We will explain the applied rules from the innermost expression first, starting with e_0, which is synthesized using the newly added rule Syn/TypeApp. In this expression, a nested application of Syn/Var is used to extract the polymorphic type t_{Cons} from the context for the list constructor variable `Cons`. The with Hindley-Milner annotated type *int* is instantiated with a new κ variable and replaces the type parameter α in t_{Cons}, resulting in the now monomorphic type

$$\begin{aligned} t_{Cons_{int}} := \{x : int \mid \kappa_2(x)\} \rightarrow xs : list(\{v_1 : int \mid \kappa_2(v_1)\}) \rightarrow \\ \{v : list(\{v_1 : int \mid \kappa_2(v_1)\}) \mid len(v) = 1 + len(xs)\} \end{aligned}$$

leading to the overall derivation tree

$$\frac{\Gamma_\lambda \vdash \text{Cons} \nearrow t_{Cons} \dashv true \qquad int \triangleright \{v_1 : int \mid \kappa_2(v_1)\}}{\Gamma_\lambda \vdash \text{Cons[int]} \nearrow t_{Cons_{int}} \dashv true}\ \textsc{Syn/TypeApp}$$

for the Syn/TypeApp rule. With the rule Syn/App, the type $t_{Cons_{int}}$ is then used to synthesize e_1, where the value 1 is applied to this resulting function type. To construct the resulting VC the applied value needs to be a subtype of the first function parameter:

$$\{v : int \mid v = 1\} \leqslant \{x : int \mid \kappa_2(x)\} \dashv \forall(v : int).v = 1 \implies \kappa_2(v)$$

As the VC of the previous Syn/TypeApp is just *true*, this subtyping constraint is the resulting VC for the first Syn/App, together with the remaining part of the function type:

$$t_{App_1} := xs : list(\{v_1 : int \mid \kappa_2(v_1)\}) \rightarrow \{v : list(\{v_1 : int \mid \kappa_2(v_1)\}) \mid len(v) = 1{+}len(xs)\}$$

$$\frac{\begin{array}{c}\Gamma_\lambda \vdash \text{Cons[int]} \nearrow t_{Cons_{int}} \dashv true \qquad \Gamma_\lambda \vdash 1 \nearrow \{v : int \mid v = 1\} \dashv true \\ \{v : int \mid v = 1\} \leqslant \{x : int \mid \kappa_2(x)\} \dashv \forall(v : int).v = 1 \implies \kappa_2(v)\end{array}}{\Gamma_\lambda \vdash \text{Cons[int]}\ 1 \nearrow t_{App_1} \dashv \forall(v : int).v = 1 \implies \kappa_2(v)}\ \textsc{Syn/App}$$

As in this program, two values get applied, the rule Syn/App is used again, to synthesize e_2. The value x then needs to be a subtype of the second function parameter of $t_{Cons_{int}}$, which has the type

$$t_{list_{xs}} := \{xs : list(\{v_1 : int \mid \kappa_2(v_1)\}) \mid true\}$$

as seen in t_{App_1}. As x is the parameter of the overall λ function, it is in the current context Γ_λ and its type is

$$t_{list_x} := \{x : list(\{v : int \mid \kappa_1(v)\}) \mid \kappa_0(x)\}$$

resulting in the subtyping constraint:

$$t_{list_x} \leqslant t_{list_x} \dashv \forall(v : list)\ \kappa_0(v) \implies true \land \forall(v : int)\ \kappa_1(v) \implies \kappa_2(v)$$

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

This subtyping is an application of the new rule Sub/Data, where now not only the subtype itself needs to imply the refinement of the supertype, but also the nested type parameters. This constraint in conjunction with the subtyping constraint of the first function parameter forms the overall VC of the synthesis of e_2. The last remaining part of $t_{Cons_{int}}$

$$\{v : \mathit{list}(\{v_1 : \mathit{int} \mid \kappa_2(v_1)\}) \mid \mathit{len}(v) = 1 + \mathit{len}(x)\},$$

is the resulting type of Syn/App. The Syn/Lam rule is executed on e_3, the outermost expression. Here, the parameter x is added to the context, resulting in Γ_λ. This context is used for the previous synthesize rule applications. The inserted type is the parameter of the λ function instantiated with a hole (κ variable). This new type is used to construct the overall type of `listAdd1`, as well as to create the overall VC, as the for all quantified refinement of this type needs to imply the previous VC. This overall result can be seen in Figure 3.8, where the consequent of this first implication are the subtyping constraints generated by the successive application of the rule Syn/App.

$$\begin{gathered}
\Gamma \vdash \text{listAdd1} \nearrow \\
\{x : \mathit{list}(\{v : \mathit{int} \mid \kappa_1(v)\}) \mid \kappa_0(x)\} \to \{v : \mathit{list}(\{v_1 : \mathit{int} \mid \kappa_2(v_1)\}) \mid \mathit{len}(v) = 1 + \mathit{len}(x)\} \\
\Dashv \\
\forall (x : \mathit{list})\ \kappa_0(x) \implies \\
(\forall (v : \mathit{int})\ v = 1 \implies \kappa_2(v)) \wedge \\
(\forall (v : \mathit{list})\ \kappa_0(v) \implies \mathit{true} \wedge \forall (v_1 : \mathit{int})\ \kappa_2(v_1) \implies \kappa_1(v_1))
\end{gathered}$$

Figure 3.8: Inference result of Figure 3.7

This type and its VC is similar to the example in Figure 2.3 in the sense that the resulting VC only becomes interesting once this type is used. In this case, the Horn constraints could once again simply be replaced by true, which would verify this type.

The relevant constraints are usually given due to a top level type annotation, or by an application of a value to this λ function. Assume that we have a type annotation

$$\begin{aligned}
&\{x : \mathit{list}(\{v : \mathit{int} \mid \{v : \mathit{int} \mid v > 0\}\}) \mid \mathit{true}\} \to \\
&\{v : \mathit{list}(\{v_1 : \mathit{int} \mid \{v_1 : \mathit{int} \mid v_1 > 0\}\}) \mid \mathit{len}(v) = 1 + \mathit{len}(x)\}
\end{aligned}$$

for this function, with a list of positive integers, to a list still with positive integers. We now need to check the type for `listAdd1` and, as previously explained, in λ_Σ^+ this is handled as a subtyping case. The inferred type needs to be a subtype of the annotated type. For the nested values of the list, this now means, that $v > 0$ needs to imply $\kappa_1(v)$ and $\kappa_2(v)$ needs to imply $v > 0$. Those implications, in conjunction with the VC in Figure 3.8 can only be satisfied by $\kappa_1(v) := v > 0$ and $\kappa_2(v) := v > 0$.

This behavior is similar if `listAdd1` is used in a type application, meaning that the resulting type is inferred by Syn/App. The given input then also needs to be a subtype

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

```
toInt : {s: string | s ∈[0−9]+ } → int
assert : { b:bool | b = true } → unit

listHeadToInt = \x:list(string).
   switch(x)
   | Nil: assert false
   | Cons(z,zs):
      let r = toInt z in
      assert true
```

Figure 3.9: Example of list destruction, with definitions for `toInt` and `assert`

of the first input of the inferred function. And this subtyping constraint gives similiar implications as a type checking step.

Again, this example also practically explains the usefulness of those κ variables used in liquid type systems. As those subtyping constraints build such implication chains, no type annotations are needed in many cases, if those κ variables can be solved.

Data Type Destruction

For our destruction example, we also want to include a bit of grammar solving. The program in Figure 3.9, takes a list, accesses the first element and applies it to the `toInt` function. In this example, to illustrate the necessity of the *unapply* method from Figure 3.5, we fail the computation if the *Nil* branch is used. The program `listHeadToInt` can then only be verified if it is called with nonempty lists, as this information is constructed by the *unapply* method.

Furthermore, we show how the constraint imposed by `toInt` is propagated through the refinements, and how Panini is then able to synthesize a grammar for the elements of a list.

The overall top level overview is similar to the first example in Figure 3.7. We first apply the Syn/Lam rule, which again instantiates our list of strings to a type with holes. Then our newly added Syn/Switch rule is used, followed by two applications of the Syn/Alt rule. To focus this example onto the relevant parts in this example, we ignore the deeper nested Syn/Let and Syn/App rules in the alternatives and the types and VCs are asumed to be allready infered. Those given types and VCs are also simplified. The assert statement is ignored completely, and we just assume a true or false VC to be returned.

In general, in the Panini system, the assert statement returns a *unit* type that signifies a successful program exit. The type system should be able to infer conditions so that assert false is never called. For synthesizing a string grammar, this step then enforces the grammar to be constructed in a way so that it does not lead to a negative assertion. For example, if a parser checks that a string contains only lowercase characters and aborts otherwise, this check is translated into an assert call by the Panini frontend.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

We again have a similiar context as in the first example:

$$t_{Nil} := \forall\alpha.\{v : \mathit{list}(\alpha) \mid \mathit{len}(v) = 0\} \tag{3.8}$$

$$t_{Cons} := \forall\alpha.x : \alpha \rightarrow xs : \mathit{list}(\alpha) \rightarrow \{v : \mathit{list}(\alpha) \mid \mathit{len}(v) = 1 + \mathit{len}(xs)\} \tag{3.9}$$

$$\Gamma := [\texttt{Cons} \rightarrow t_{Cons}, \texttt{Nil} \rightarrow t_{Nil}] \tag{3.10}$$

$$\Gamma_\lambda := \Gamma[x \mapsto \{x : \mathit{list}(\{v : \mathit{int} \mid \kappa_1(v)\}) \mid \kappa_0(x)\}] \tag{3.11}$$

And we also can give the expressions in a similar nested manner:

$$e_{Nil} := \texttt{Nil:}\ \ \texttt{assert false} \tag{3.12}$$

$$e_{Cons} := \texttt{Cons(z, zs):}\ \ \texttt{let r = toInt z in assert true} \tag{3.13}$$

$$e_{Switch} := \texttt{switch}(x) \mid e_{Nil} \mid e_{Cons} \tag{3.14}$$

$$e_\lambda := \lambda(x : \texttt{list(string)}.e_{Switch}) \tag{3.15}$$

We once again explain those rule application in a bottom up approach, starting with the Syn/Alt rule for the *Nil* case. As we do not explain the nested `assert` and `let` statements, this is the deepest nested rule.

To be able to infer the type of the nested expression, the functions introduced in Figure 3.4 and Figure 3.5 are used.

As mentioned, *ctor* takes a constructor method and fills the potential type parameters from the current context. For *Nil*, given the polymorphic constructor type, the string parameter refined with κ_1, is extracted from the context value of the list x. The resulting monomorphic type is therefore

$$t_{Nil_{Ctor}} := \{v : \mathit{list}(\{v_1 : \mathit{string} \mid \kappa_1(v_1)\}) \mid \mathit{len}(v) = 0\}$$

and now contains the string parameter type from the context. *Unapply* now uses this monomorphic constructor, to extract the new context variables given by the alternative. For the *Nil* case, there is no additional context variable, as it is a constructor without any additional inputs. However, the *meet* helper, called in *unapply*, combines the known refinement for the variable x, which is κ_0, with the refinement $\mathit{len}(x) = 0$ of the *Nil* constructor, to the type

$$t_{Nil_x} := \{v : \mathit{list}(\{v_1 : \mathit{string} \mid \kappa_1(v_1)\}) \mid \kappa_0(v) \wedge \mathit{len}(v) = 0\}$$

which replaces the previous x variable in the context for the inference of the nested expression of the alternative.

For this example, we assume that inferring *assert false* with this context returns a VC of false and the *unit* type. This VC needs to be implied by the all types and their refinements added to the context in *unapply*. Only x has been replaced by the alternative, and its universally quantified refinement needs to imply *false*, resulting in the overall VC

$$c_{Alt_{Nil}} := \forall(x : \mathit{list})\ \kappa_0(x) \wedge \mathit{len}(x) = 0 \implies \mathit{false}$$

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

and the complete derivation tree

$$\dfrac{\begin{array}{c} ctor(\Gamma, Nil, x) = t_{Nil_{Ctor}} \qquad unapply(\Gamma, x, \varnothing, t_{Nil_{Ctor}}) = \Gamma[x \mapsto t_{Nil_x}] \\ \Gamma[x \mapsto t_{Nil_x}] \vdash \text{assert false} \nearrow unit \dashv false \end{array}}{\Gamma \vdash \text{Nil: assert false} \nearrow unit \dashv c_{Alt_{Nil}}} \ \textsc{Syn/Alt (Nil)}$$

for the application of Syn/Alt in the *Nil* case.

The second alternative, the *Cons* constructor, is a bit more interesting, as we have two additional variables in the context generated by *unapply*. The *ctor* helper now returns the monomorphic function type for this constructor:

$$\begin{aligned} t_{Cons_{Ctor}} := \{x : string \mid \kappa_1(x)\} \rightarrow xs : list(\{v_1 : string \mid \kappa_1(v_1)\}) \rightarrow \\ \{v : list(\{v_1 : string \mid \kappa_1(v_1)\}) \mid len(v) = 1 + len(xs)\} \end{aligned} \tag{3.16}$$

The variables z and zs are assigned the first and second parameter of this function type $t_{Cons_{Ctor}}$ using the *unapply* helper. The new type of the variable x is once again a result of the *meet* helper, and unifies the refinements κ_0 and $len(x) = len(zs) + 1$ to the type

$$t_{Cons_x} := \{v : list(\{v1 : string \mid \kappa_1(v1)\}) \mid \kappa_0(v) \wedge len(v) = 1 + len(zs)\}$$

and the overall extended context in this Syn/Alt case is therefore:

$$\Gamma_{Cons} := \Gamma_{\lambda}[z \mapsto \{z : string \mid \kappa_1(z)\},\ zs \mapsto list(\{v1 : string \mid \kappa_1(v1)\}),\ x \mapsto t_{Cons_x}]$$

This context is then used to synthesize the nested expression containing the `toInt` function application, which we will not explain for this example, but results in the following VC:

$$\begin{aligned} c_{toInt} := \forall(s : string)\ \kappa_1(s) \wedge s = z \implies \\ s \in \texttt{[0-9]+} \ \wedge \forall(r : int)\ r = strToInt(s) \implies true \end{aligned}$$

This VC now restricts the string variable z, according to the `toInt` function. As the input string s needs to be a number value and z needs to be a subtype of s, the refinement κ_1 needs to imply this constraint. This property allows us to later construct a grammar.

Additionally, the newly added or altered variables z, zs and x are used to create the resulting VC

$$\begin{aligned} c_{Alt_{Cons}} := \forall(z : string)\ \kappa_1(z) \implies \forall(zs : list)\ true \implies \\ \forall(x : list)\ \kappa_0(x) \wedge len(x) = len(zs) + 1 \implies c_{toInt} \end{aligned}$$

as the variables need to imply the nested VC c_{toInt}. Note that the ordering of those variables is important. As the refinements of x can relate to the values of z or zs, the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

ordering is according to the scopes of the constructor *Cons*. The resulting nested type for this alternative is also *unit*, resulting in the derivation tree

$$\dfrac{\begin{array}{c} ctor(\Gamma, Cons, x) = t_{Cons_{ctor}} \qquad unapply(\Gamma, x, [z, zs], t_{Cons_{ctor}}) = \Gamma_{Cons} \\ \Gamma_{Cons} \vdash \text{let t = toInt z in assert true} \nearrow unit \dashv c_{toInt} \end{array}}{\Gamma \vdash \text{Cons(z, zs): ...} \nearrow unit \dashv c_{Alt_{Cons}}} \ \textsc{Syn/Alt (Cons)}$$

for Syn/Alt in the *Cons* case.

For Syn/Switch, we now simply need to combine the results of the Syn/Alt rules. The subtyping constraint is the same for both alternatives in this example. The *unit* type is instantiated with a new hole, which then needs to be implied by *true*:

$$c_{Unit} := \forall (u : unit).\ true \implies \kappa_2(u)$$

This subtyping constraint is then inserted into the constructed implications. This is necessary, as these implications contain all variables of an alternative and the refinement of an inferred alternative type, and consequently the subtyping VC can rely on this context. Ultimately, for each alternative, the antecedents of the implication chain are unchanged, but the final consequent now also contains those subtyping constraints. This overall resulting derivation tree for Syn/Switch is therefore:

$$\dfrac{\begin{array}{c} unit \rhd \{u : unit \mid \kappa_2(u)\} \\ \Gamma \vdash Nil \ldots \nearrow unit \dashv c_{Alt_{Nil}} \qquad unit \leqslant \{u : unit \mid \kappa_2(u)\} \dashv c_{Unit} \\ \Gamma \vdash Cons(z, zs) \ldots \nearrow unit \dashv c_{Alt_{Nil}} \quad unit \leqslant \{u : unit \mid \kappa_2(u)\} \dashv c_{Unit} \end{array}}{\Gamma \vdash \text{switch(x) | } \ldots \nearrow unit \dashv \begin{array}{l} (\forall (x : list)\ \kappa_0(x) \wedge len(x) = 0 \implies c_{Unit} \wedge false) \wedge \\ (\forall (z : string)\ \kappa_1(z) \implies \forall (zs : list)\ true \implies \\ \ \forall (x : list)\ \kappa_0(x) \wedge len(x) = len(zs) + 1 \implies \\ \ c_{Unit} \wedge c_{toInt}) \end{array}} \ \textsc{Syn/Switch}$$

The final Syn/Lam application is nearly identical to the first example. This extended context Γ_λ is used in the preceding synthesize rules, and the of Syn/Switch resulting *unit* type and resulting VC are used to construct the final type and VC of `listHeadToInt`.

The resulting simplified VC can be seen in Figure 3.10. We ignore the *unit* subtyping constraint c_{Unit} given by Syn/Switch, as it can be simplified to true.

To show the grammar synthesis idea, we assume a signature

$$\{x : list(\{s : string \mid ?\}) \mid len(x) > 0\} \to unit$$

We can now use the grammar synthesis approach by the Panini system to solve for the unknown string refinement of the value of the list. As $len(x) > 0$ now needs to imply κ_0, this can trivially be replaced by $len(x) > 0$. This makes the implication of the *Nil* alternative valid, as $len(x) > 0 \wedge len(x) = 0$ is simply false.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

$$
\begin{gathered}
\Gamma \vdash \text{listHeadToInt} \nearrow \\
\{x : \mathit{list}(\{s : \mathit{string} \mid \kappa_1(s)\}) \mid \kappa_0(x)\} \to \mathit{unit} \\
\dashv \\
\forall (x : \mathit{list})\ \kappa_0(x) \implies \\
\forall (x : \mathit{list})\ \kappa_0(x) \wedge \mathit{len}(x) = 0 \implies \mathit{false} \\
\wedge \\
\forall (z : \mathit{string})\ \kappa_1(z) \implies \\
\forall (zs : \mathit{list})\ \mathit{true} \implies \\
\forall (x : \mathit{list})\ \kappa_0(x) \wedge \mathit{len}(x) = \mathit{len}(zs) + 1 \implies \\
\forall (s : \mathit{string})\ \kappa_1(s) \wedge s = z \implies s \in \texttt{[0-9]+} \wedge \\
\forall (r : \mathit{int})\ r = \mathit{strToInt}(z) \implies \mathit{true}
\end{gathered}
$$

Figure 3.10: Simplified Inference result of Figure 3.9

The remaining κ variables are also eliminated, either by the Fusion algorithm or via predicate abstraction. As those approaches generally try to eliminate all κ variables, Panini needs to restore the identified string grammar variable κ_1 and uses abstract interpretation to infer a suitable grammar

$$s \in [0-9]+$$

This inferred grammar is then used instead of κ_1 to make the overall VC valid. In this simple example, the grammar just needs to imply the precondition of the `toInt` function, which therefore is exactly this condition.

Note that for this example, even though we have a list of strings, the grammar solving here is still not nested. How an ADT can be used in nested grammar solving is explained in section 3.5.

3.4.1 Formal Correctness

Proofs for refinement type systems are inherently complex, due to the interplay between subtyping rules, verification conditions and type synthesis rules [5]. A formal correctness proof of our novel data type synthesis rules is out of scope for this thesis.

However, for the type safety of those rules, we can still outline a few arguments. Type safety requires two properties, progress and preservation [45]. Progress means that if a term e is well typed, we can reduce e further, i.e., execute a computation step, or e is already a value. In our case, regarding the Syn/Switch rule, a switch term is not a value, but is reduced to its alternatives. For Syn/Alt, an alternative is composed of a nested expression, to which it is reduced to. We assume that the other rules guarantee progress, therefore progress for those two rules can also be presumed. Preservation means

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

that for each computation step evaluating a term $e \hookrightarrow e'$, the type of the term e' does not change, i.e., if the expression e has some type t, we can guarantee that this same type t is returned by that expression. From the syntactic meaning of a switch expression, this means that one of the alternatives will be evaluated. The type of a switch expression is composed of the type of the first alternative instantiated with a kappa template. As every alternative needs to be a subtype of that template type, this guarantees firstly that the unrefined types need to match, and secondly, that the template κ variable is implied by (for covariance) or implies (for contravariance) the refinement predicates of the type of an alternative. It guarantees that the refinement of an alternative is a subtype of the overall switch type. Therefore, the type does not change, only the refinement might be stronger.

For example, the subtyping constraint for two alternatives, if the alternatives return the types $t_{a_1} := \{v : int \mid v = 1\}$ and $t_{a_2} := \{v : int \mid v = 2\}$, respectively, might lead to a κ variable solved to $\kappa(v) := v = 1 \lor v = 2$. This fulfills the subtyping constraint for the first alternative $t_{a_1} \leqslant \hat{t} \dashv v = 1 \implies \kappa(v)$. If we execute a step into the first alternative, the concrete type after this step remains unchanged in the idea of a refinement type. Even though the type of an alternative might have a stronger refinement, the overall type remains the same.

Furthermore, we have also observed that the generated VCs when using data deconstruction, compared to using a polymorphic access method (e.g. *head* for a `list` type), only differ marginally. As type safety has been shown for those polymorphic rules in similar cases [5], this also supports the argument that those new rules for deconstruction are type safe.

3.5 Grammar Solving

As we now have seen and explained the new additions, the question remains, how can we utilize nested data structures, to solve complex grammars, and how do those grammars look like? As seen in the introductory example, we want to handle the case where a string is translated into an ADT. Those ADTs then contain nested string values, that can be subject to further parsing operations, which influence the overall grammar of the original input. One example can be seen in Figure 3.11, where a string is split into a tuple ADT, and the nested values of that tuple are then itself parsed either by the integer constructor or string equality. And we want to use those nested grammars of such an ADT to construct an overall grammar.

```
def parser(self, arg):
    first, second = arg.split('/', 1)
    firstValue = int(first)
    assert second == "Test"
```

Figure 3.11: Example parser for a Tuple ADT.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

We mainly focus on the observation that `split` is one of the most common string operations in ad hoc parsers, and that it is often followed by a tuple assignment [36].

Additionally, relevant ADTs for common ad hoc parsers are mostly lists and tuples. To directly transform a string to an ADT, `split` or functions similar to `split` are common options. This is the case in the Python standard library[2] and for many other languages.

Therefore, we focus on grammars that are created through possibly infinite lists or n-tuples. A list data type only has a single type parameter and once instantiated, its refinements describe each element of the list. This also means in the case where the parameter type of a list is string, we can only have a grammar where the grammar every nested element is homogeneous. Our motivating example represents such a homogenous grammar (cf. Figure 1.2). This is not the case for tuples, as we have a type parameter for each element, which can hold different string type instances. Each string instance can therefore represent different grammars, and an example would be Figure 3.11. According to this observation, for those nested grammars, we have identified three possible cases:

1. Homogeneous lists, where all nested elements in a data structure have the same grammar.

2. Heterogeneous n-tuples, with a fixed number of elements.

3. Heterogeneous lists, with a possibly infinite number of elements and with possibly dynamically changing grammars.

In theory, we are able to deal with the first two cases straightforward, as we can use suitable data structures. If we have a list data structure, instantiated with a string, this string value and its corresponding predicate, can be used to construct the grammar of the first case. In the second case, we can simply use an n-tuple instantiated with n string types. Each of those types can have different predicates, capturing the different nested grammars of those strings, which overall results in a heterogeneous grammar.

The third case, however, is complicated, as it also includes non-regular grammars. It is possible that the grammar of a list element depends on some other computation, or relates to other elements of the list. Panini can only synthesize regular grammars, and even if all nested elements individually are regular grammars, the overall resulting nested grammar might not be. However, there are still many regular grammars, that fall into the third category. One simple example, where such a heterogeneous grammar could in general be synthesized, is if there is a concrete index up to which all elements are a specific grammar, and all remaining elements are of another grammar. E.g., a string of comma separated values, where the first element is a keyword and all other elements numbers. This is still a regular grammar and could be solved by identifying those indexes, for example in the translation layer, and translating those cases into, e.g.,

[2] https://docs.python.org/3.12/library/stdtypes.html

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

two separate lists. Another approach would be utilizing qualified predicates over arrays, as discussed in section 2.4. As the purpose of this work is mainly the extension to λ_Σ, identifying relevant solvable cases, as well as implementing additional functionality for those grammars, is out of scope for this thesis (cf. section 5.1).

Therefore, we focus on the first two cases. Additionally, two other relevant pieces of information are ignored. Firstly, for homogenous lists, if there is a refinement restricting the length of a list, e.g., due to access of a specific index or other list operations, this would change the relevant grammar. We disregard this specific case due to time constraints, but our proposed solution to handle combine nested grammars should be able to handle this case as well. And secondly, the separator is excluded from the nested grammar, by the semantic definition of `split`. This was also ignored, again due to time constraints and because we would need to unify potential regular expressions.

The proposed idea on how to combine the grammars of those two cases, which is subsequently explained, should also just be seen as a first solution sketch. Other approaches might be superior, or could be necessitated by attempting to solve those outlined complex cases.

3.5.1 Handling grammars in VCs

We encountered several issues when trying to solve a grammar arising due to application of the `split` function. The primary problem is that information cannot flow backward in the system. Specifically, when an output element of the `split` function is further constrained for a nested grammar, the solver fails to leverage this information for grammar solving of the input s. This deficiency stems from the inability of the solver to recognize and utilize these additional constraints effectively.

Problem

We initially thought that a (simplified) `split` function definition might be enough to gather the necessary constraints for the original input string:

```
split:{s:string|true} → {v:list({s1:string | s∈re_star(s1 ++ ',')})|true}
```

However, this was not the case. The core issue in the VC generation can be summarized as follows: as the input is unconstrained, the solver defaults to

$$\forall s \in \mathbb{S}.\, \kappa(s) \Rightarrow \text{true},$$

allowing it to trivially conclude $\kappa_1 = \text{true}$. Additionally, the output now contains the specific refinement $s \in \texttt{re\_star}(s1 \texttt{ ++ } ',')$. If this output is now constraint further, e.g., in a `toInt` function, the derived implication of a subtyping constraint looks as follows:

$$\forall s \in \mathbb{S}.\, s \in \texttt{re\_star}(s1 \texttt{ ++ } ',') \Rightarrow s \in \texttt{[0-9]+}$$

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

The solver is unable to satisfy this constraint. Overall, the resulting string grammar is either any string, as the input κ variable is set to true, or the solver fails, as the resulting constraint of split does not imply a further constraint.

As Panini relies on simplifying those verification conditions using the *Fusion* algorithm followed by predicate abstraction before trying to solve for string grammars, at least the output constraint is already handled. The grammar solving approach reevaluates all κ variables with a string type. However, in this case, there is no direct logical connection from the input κ to the constraints generated by the output, so those are not considered at this step. This would be solvable by altering the Fusion algorithm to not discard those additional constraints, and by extending the abstract interpretation approach to be able to use those additional constraints. However, another approach is simpler and, as we do specifically know the concrete structure of a nested grammar, more consistent.

Solution

In general, the problem lies within the information flow through the refinements, which are scoped from outermost to innermost, e.g., the output refinements of a function can rely on the input values, but not the other way around. However, for the input string to be constrained according to a nested grammar, we now need some kind of backward information flow, the refinements of the output now dictate the input refinement.

The key idea is that grammar solving already does this in a way and proceeds from the output backwards to the input. At the top level, the system can handle these backward inferences due to the introduction of a new κ variable, replacing unknown constraints (denoted by `?` in the surface syntax). This unknown constraint in the top level signature represents a point where the implication chain for a string grammar starts. The new κ variable implies all constraints which determine the resulting string grammar.

An initial idea was to generalize this mechanism by introducing additional placeholders ("holes") in the program, which represent grammars that are yet to be determined. These placeholders now allow the encoding of multiple grammars that may still be unknown, which could help bridge the gap between the input and its corresponding constraints. Previously, this approach was only employed at the top level string input and has now been generalized.

The challenge we encountered here is constraining the input such that it relates to an unknown grammar. We can solve this by utilizing variables that are filled by those unknown grammar holes. Therefore, we now need two new concepts: *GrammarHoles* and *GrammarLinks*. The former represents unknown constraints on the output, therefore introducing a new unknown grammar, and the latter relates this unknown grammar to the corresponding input constraints.

This concept extends the original idea of Panini by executing the original grammar solving process iteratively, with the possibility of utilizing intermediate results. The κ variables are resolved in a nested fashion, and we can assume that the grammar hole κ variable is solved before it is required in the link variable.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Practical Example

To demonstrate the concept, consider the example `parser` function:

```
toInt: {s:string | s ∈`[0−9]+`} → {v:int | v = strToInt(s)}

parser: {a:string|?} → int
parser = λ(s:string).
         let tmp = strip s in
         toInt tmp
```

We chose `strip` as the example, as it is much simpler than the `split` function, while still having an alteration of a nested grammar. This example of `strip` is analogous to data structure constraints, and was previously not considered in the PANINI system. The method outlined above now represents a possible approach to resolving these constraints. With the approach outlined above and the following function definition

```
strip: {s: string | gl(s)(r)(re_star(' ') ++ r ++ re_star(' '))}
         → {r: string | gh(r)}
```

we can now solve a grammar containing a nested grammar. We need three pieces of information, for a *GrammarLink*. First, a name, which is s in this example, relating the variable name to the *GrammarLink*. Second, a list of related *GrammarLink* names that each represent a nested grammar. In this example, this is only r, as only a single nested grammar is present. And finally, a definition of a resulting grammar, that now can contain the related *GrammarLinks* as variables. We opted to just use parenthesis to separate each part of a *GrammarLink*, but it is not a function.

The *GrammarHole* $gh(r)$ is simply instantiated to a new κ variable once the `strip` method is fetched from the context via SYN/VAR. And this κ variable is then added to the *GrammarLink*, replacing the r variable. The initial solving, using the Fusion algorithm and predicate abstraction are not influenced by this, as all string-related κ variables are handled by the grammar solving approach. And the grammar solving approach first solves the nested κ, which in this simple example is the *GrammarHole*. The resulting condition is

$$\forall r \in \mathbb{S}.\, \kappa_r(r) \Rightarrow r \in \texttt{[0−9]+},$$

and κ_r will be solved to the regular expression `[0−9]+`. In the next iteration, the remaining top level κ variable is solved, and we insert the previous solutions into the *GrammarLink*. We then translate this into a regular expression, if all variables are filled, which in turn is then used for further solving. In this example, the resulting regular expression is: `[ ]*([0−9]+)[ ]*`, which is the overall grammar of this ad hoc parser, as the remaining condition is:

$$\forall s \in \mathbb{S}.\, \kappa_s(s) \Rightarrow s \in \texttt{[ ]*([0−9]+)[ ]*}$$

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

In section 4.1, we also give the concrete implementations for our `split` function in λ_Σ^+.

Outlook and Concerns

This approach consistently works for many examples. Currently, we rely on some assumptions regarding the correctness of the approach, and there are parts of the implementation that are not yet fully refined. Those assumptions are: (1) The nested *GrammarHoles* always resolve to a valid regular expression, (2) if the *GrammarHole* is not constrained, any word in the regular language is allowed and (3) the *GrammarHole* variables are always used nested and not in an intertwined manner. In practice, for ad hoc parsers, we expect those assumptions to always hold.

Another idea would have been to trace those nested grammars and iteratively solve them as independent top-level parsers with the Panini system. In this case we would not have needed to extend the predicate logic with the specific handling of nested grammars. However, as the function definitions for those cases are not intended to be written by an end user, this is not an issue. Additionally, this would have been significantly more implementation effort.

Another benefit of this solution is that more information about our nested grammar could be included. In the `split` case, we could for example also relate to the refinement of a resulting list. This refinement could capture length constraints naturally in the refinement logic and that would allow us to include this information in the overall resulting grammar. This case would probably be easy to handle as we have fine-grained control over how the nested grammars and or refinements are combined.

Overall, we have introduced the extended λ_Σ^+ language as well as explained how those extensions relate to the grammar synthesis approach of Panini. This allows us to answer our first research question:

RQ1 How can the λ_Σ calculus be extended to support polymorphic operations over algebraic data types, while preserving its ability to synthesize input string grammars?

We have introduced λ_Σ^+ which adds polymorphism and algebraic data types to λ_Σ. While some of the new inference rules we have added in λ_Σ^+ are common in many Liquid type systems, we also have introduced a novel rule to infer the destruction of algebraic data types.

Furthermore, we have discussed what forms grammars in algebraic data types can take and how nested grammar constraints can be used to construct an overall grammar, for common cases. Not only does λ_Σ^+ preserve the ability of Panini to synthesize input string grammars, it significantly expands the range of programs for which grammars can be inferred.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 4

Implementation and Evaluation

In this chapter, we explain implementation details as well as outline some challanges that we encountered during our implementation. Subsequently, to answer RQ2, we need to evaluate those additions on real-world ad hoc parsers.

4.1 Implementation

PANINI is written in *Haskell* and to this date contains approximately 14000 lines of code. As introduced in chapter 3 the changes to the syntax and typing rules have been added to the code base of PANINI.[1] To this end, some parts of the previous implementation had to be altered.

We needed to add a representation for the newly introduced syntax. This step also necessitated adjusting the code responsible for parsing the file input. Once this had been done, the code handling the inference and subtyping had to be extended. As this code closely follows the recursive nature of the inference and subtyping rules, we introduced additional cases, as well as added minor changes to some existing cases. For example, in the subtyping for ADTs. The relevant rules are implemented according to the definitions outlined in chapter 3.

To enable nested grammar solving (cf. section 3.5), we also modified the code responsible for the final constraint solving. To this end, before solving for a specific κ variable, a sanitizing step inserts previously solved κ variables into any present *GrammarLinks*. Once all κ variables of the *GrammarLink* is solved, it is replaced by the resulting regular expression, which allows continuing the solving step.

By the definition of our typing rules, polymorphic type application needs explicit type annotations. We expect the annotation of polymorphic functions usages to happen during

[1] https://github.com/JakobHoffmann/panini

the automatic translation from parser source code to lambda sigma plus by the Panini frontend, using standard Hindley-Milner type inference.

We encountered some smaller challenges as some necessary steps were still missing for this work. We also added basic support for parsing POSIX extended regular expressions within type signatures, necessary for the previously given `toInt` function (cf. Figure 3.9).

For ADTs, both the variable name of the data type and measures, which needed to be defined in the queries sent to the SMT solver. Those additions lead to some changes regarding the transformation of VCs into SMT-solvable constraints. With this transformation, we encountered a significant challenge in translating VCs into SMT solver queries, which we will explain in detail in the following section.

4.1.1 SMT Solving with Uninterpreted Functions

The previous implementation of the Panini system simply used the generated VC as an input into the SMT solver without significant transformations. Those conditions contain multiple universally quantified implications ($\forall x.P(x) \implies Q(x)$) as defined by the typing rules in Figure 2.7 and Figure 2.6.

When defining ADTs, uninterpreted functions are used as measures for properties of those data types, e.g., the length of a list. During our implementation, it became evident that there was an issue when using uninterpreted functions that allowed nonsensical values for those measures to be type checked. For example, the system would incorrectly verify that an empty list could be of the type $w := \{v : \mathit{list} \mid \mathit{len}(v) = 1\}$. An empty list has the type $t := \{v : \mathit{list} \mid \mathit{len}(v) = 0\}$. The empty list can be of type w if $t \leqslant w$ and, due to the subtyping judgment, this is the case iff

$$\forall (v : \mathit{list}). \quad \mathit{len}(v) = 0 \Rightarrow \mathit{len}(v) = 1.$$

The SMT solver verified this implication, if an uninterpreted function was used for *len*. The behavior of the SMT solver undermined the verification process in such cases.

The crux of the problem is that Z3, when confronted with implications of the form $p \Rightarrow q$, tended to trivially satisfy the implication by assigning values to the uninterpreted functions such that p would be false.

To address this issue, we restructured our approach to avoid the trivial satisfaction of implications by the SMT solver. Instead of directly feeding the implication $p \Rightarrow q$ to Z3, we transformed the problem making the solver work towards proving the unsatisfiability of $p \land \neg q$. By expecting the solver to return `unsat` for the input $p \land \neg q$, we effectively force it to evaluate the actual conditions under which the uninterpreted functions in p hold true and q fails. This change compelled Z3 to thoroughly check the consistency and correctness of verification conditions containing uninterpreted functions, ensuring that the original implication $p \implies q$ could not be trivially satisfied. Reformulating those implications preserved the logical relationship between p and q and those implications were then correctly verified by the solver.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

This approach is also mentioned by Jhala and Vazou [17, p. 8], as well as implemented in *liquid-fixpoint*,[2] the Horn clause solving backend for Liquid Haskell.

4.2 Evaluation

For the evaluation, we have three goals: (1) We want to verify that the refinement inference for polymorphism and ADTs works in theory. To this end, a set of test programs was created. This approach should aim to show the new features and give a baseline for the capabilities of the extended λ_{Σ}^{+}.

(2) We want to show that the changes in λ_{Σ}^{+}, have not caused any regressions in the ability of Panini to synthesize string grammar. There is already a sizable number of evaluation parsers for Panini, on which we reevaluate our extensions as well.

(3) We show the usefulness of our extensions by testing them on real-world programs. To this end, we use a dataset, of ad hoc parsers mined from open source repositories [36, 37].

4.2.1 Data Collection and Methodology

Our dataset of real-world ad hoc parsers was provided by Schröder, Olschnögger, Goritschnig, and Cito [37], and is based on the Boa [10] 2022 February/Python dataset,[3] which contains 102,424 *GitHub* repositories that have Python as their main programming language. Using program slicing techniques, ad hoc parser cores containing only those parts of programs that are involved in string parsing were extracted from the original Python programs. We were given access to a preliminary version of this dataset, which contains 1574 Python ad hoc parsers.

As the abilities of the Panini system to translate those parsers automatically into λ_{Σ}^{+} are still limited, we needed to translate them manually. This manual translation step necessitated that we restrict the number of evaluated parsers, to 100 randomly selected from the dataset. This should give enough parsers for our evaluation and restricts the manual translation effort. We want to avoid translating parsers that can be solved by the current state of the Panini system, as this evaluation has already been done. Therefore, the selected set is pruned further to remove any examples that do not contain polymorphism or ADTs.

Of the 100 randomly selected parsers, we removed 37 as invalid samples. They were either not a parser or incomplete, where some parts of the extracted code are missing. From the set of valid parsers, we removed an additional five parsers, as they contain certain regex operations that are not yet supported by Panini Of the remaining 58 parsers that are in theory suitable for the Panini system, 40 parsers do not make use of ADTs or polymorphism. This means 18 parsers are remaining for our evaluation, all of which require our new additions. The selection process is also visualized in Figure 4.1.

[2] https://github.com/ucsd-progsys/liquid-fixpoint
[3] https://boa.cs.iastate.edu/stats/index.php

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

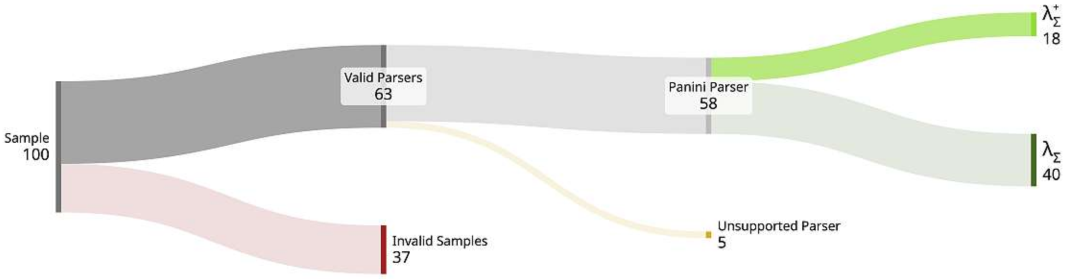

Figure 4.1: Selection of manually translated parsers from the sample set.

```
splitList:
   {s: string | gl(s)(s1, sep)(s1 ++ re_star(sep ++ s1))} →
   {sep: string | gh(sep)} →
   {v: list({s1: string | gh(s1)}) | true}
```

Figure 4.2: Definition of split for lists with homogenous grammar.

We manually translated the 18 selected Python parsers into λ^+_Σ. This translation also involved defining specific functions found in the Python code. In our observed examples, the relevant functions were `split` and the integer constructor `toInt`. The `split` function occurred either as a direct list assignment or as a tuple assignment up to a 3-tuple. As mentioned, those cases are handled differently, and correspond to different classes of grammars. For our purposes, we decided to implement those functions in a simple manner. The definition of `split` for homogenous lists can be seen in Figure 4.2. The relevant part is the definition of the *GrammarLink* for the first string input. Apart from the name s, we define two variables: $s1$ the nested grammar determined by the returned list, and the separator input *sep*. The grammar uses the operators ++ and *re_star*, which are string concatenation and Kleene star, respectively. The resulting grammar is therefore a concatenation of the variable $s1$ with an optional repetition of the *sep* variable concatenated with $s1$. This grammar is constructed once the solving process of the nested *GrammarHoles* is complete.

For the tuple assignment, we give the 2-tuple example in Figure 4.3. This figure also includes the tuple data type definition. As the 2-tuple has a different grammar for each value, we have three variables, s_1 and s_2 representing the tuple grammars and once again the separator value *sep*. This *GrammarLink* uses the same regular expression operators as the list definition, and the resulting grammar is each tuple element concatenated with the separator and an optional catch-all grammar at the end. This allows each element of an n-tuple to have a separate grammar and can thus handle the case of a finite heterogeneous grammar. For arbitrary n-tuples, the function definition is analogous.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
type tuple(#a, #b) =
| Tuple: ∀#a.∀#b.x:#a → y:#b → {v:tuple(#a, #b) | true}

splitTuple:
   {s: string | gl(s)(s_1, s_2, sep)
      (s_1 ++ sep ++ s_2 ++ re_star(sep ++`.*`))
   } →
   {sep: string | gh(sep)} →
   {v: tuple(
         {s_1: string | gh(s_1)},
         {s_2: string | gh(s_2)}
      ) | true}
```

Figure 4.3: Data type and split definition for 2-tuples in λ_Σ^+.

4.2.2 Results

To discuss our results, we first want to give an overview over all executed tests in Table 4.1. As mentioned, there already exists a large corpus of test programs for the original Panini evaluation. For those tests, the new version of Panini with λ_Σ^+ behaves exactly the same way, and no regression of its capabilities can be observed. As our additions only concern new language features, this is as expected. As those test programs can also be quite complex, there are also failing tests, but this is of no concern for this work.

During our implementation, we also created an additional test corpus for λ_Σ^+. Those unit tests contain only new syntax and were used to validate the correctness of our new inference rules. Those test are often quite simple, and are, for example, used to validate that the empty list has a length of zero:

```
y : { v : list(int) | len(v) = 0 }
y = Nil[int]
```

Those tests also contain the examples used in this work and are contained in the Panini source code, with their expected output.

And the last set of test programs, and the main focus of our evaluation, are the real-world

Test Corpus	Number of Tests	Successful λ_Σ	λ_Σ^+
Original Panini evaluation	653	452	452
λ_Σ^+ unit tests	44	-	44
Real-world evaluation	18	-	14

Table 4.1: Results of different evaluation types.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Parser	Success	Runtime	Grammar	Employed string operations
Split Tuple Unconstrained				
1613_194.py	Yes	150ms	`(#|[^ ]* ).*`	`startsWith`, `split`, `strip`
6770_1464.py	Yes	115ms	`[^/]*/.*`	`split`
26554_733.py	Yes	115ms	`[^/]*/.*`	`split`
Split List Unconstrained				
6770_1391.py	Yes	115ms	`.*`	`split` and list access
6770_1236.py	Yes	190ms	`[^ ]* .*`	`split`, list access, and `split`
11701_35.py	Yes	119ms	`.*`	`split` and *foreach*
12530_175.py	Yes	120ms	`[^:]*:.*`	`split` and list access
23916_453.py	Yes	115ms	`.*`	`split` and *foreach*
24871_516.py	Yes	122ms	`.*`	`split`, list access, and *foreach*
26554_673.py	Yes	116ms	`.*`	`strip`, and `split`
26554_696.py	Yes	114ms	`.*`	`split`
Split Tuple Constrained				
SANITIZE_STYLE	Yes	470ms	cf. Figure 4.6	
PARSE_DATE_TIME	No	∞	None	cf. Figure 4.4.
Split List Constrained				
VERSION_TO_LIST	Yes	277ms	cf. Figure 4.5	
NOT_EXCLAMATION	Yes	1368ms	cf. Figure 4.7	
Unsolvable				
6770_1339.py	No		None	`split`, *foreach* with custom range, and `strip`
26554_610.py	No		None	`split`, list comprehension, and list length comparison
20996_251.py	No		None	`split` with dynamic separator, list length check, list access, `strip`

Table 4.2: Real-world results grouped by Type.

ad hoc parsers. After a first inspection, we categorized those selected parsers into five categories, according to the three categories already defined in section 3.5: homogenous nested grammars, heterogeneous nested grammars with a fixed length and heterogeneous grammars with a dynamic length. As outlined, the last category is currently unsolvable. For the two solvable categories, we added another distinction whether the nested grammar itself imposes constraints on the input strings, or not.

Table 4.2 shows the results of our evaluation. In this evaluation table, we have our 18 samples by category and their inferred grammars. Additionally, the used string operations and runtime are given. The runtime is the average of 10 runs, and we can clearly see that for our examples the runtime is always negligible.[4] We can solve a large portion of

[4] We executed Panini on an Apple M1 Pro chip with 32GB of Memory.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
def parse_date_time(val):
  ymd, time = val.split("T")
  hms, tz_str = time[0:8], time[8]
  year, month, day = ymd.split("-")
  hour, minute, second = hms.split(":")
  t = mktime((int(year), int(month), int(day), int(hour), int(minute), int(second), 0, 0, 0))
```

Python code for PARSE_DATE_TIME.

```
parser : {a:string|?} -> unit
parser = λ(s:string). let t = splitTuple s
    "T" in
 switch(t) | Tuple(ymd, time):
  let triple = splitTriple ymd "-" in
  let timeTriple = splitTriple time "+" in
  switch(triple) | Tuple(year, month, day):
   switch(timeTriple) | Tuple(hour, minute,
       second):
    let yearInt = toInt year in
    let monthInt = toInt month in
    let dayInt = toInt day in
    let hourInt = toInt hour in
    let minuteInt = toInt minute in
    let secondInt = toInt second in
    assert true
```

Translation without *GrammarLink* issue.

```
parser : {a:string|?} -> unit
parser = λ(s:string). let t = splitTuple s
    "T" in
 switch(t) | Tuple(ymd, time):
  let triple = splitTriple ymd "-" in
  switch(triple) | Tuple(year, month, day):
   let timeTriple = splitTriple time "+" in
   switch(timeTriple) | Tuple(hour, minute,
       second):
    let yearInt = toInt year in
    let monthInt = toInt month in
    let dayInt = toInt day in
    let hourInt = toInt hour in
    let minuteInt = toInt minute in
    let secondInt = toInt second in
    assert true
```

Translation with *GrammarLink* issue.

Figure 4.4: Python code and translations for PARSE_DATE_TIME.

the sampled real-world parsers. For the unconstrained cases, we can solve all of them without any issues. Some of them are still complex parsers, as this category includes parsers that constraint the input before a split, and we also have observed a chaining of multiple `split` applications with a combined list and tuple assignment. Those cases can easily be identified by the resulting grammar.

By analyzing the code of our sample parsers, we determined that three parsers belong to the unsolvable category. We will explain those examples and the coding constructs that lead to this category later. For the unconstrained grammars, 11 parsers fell into this category. Some of those were quite simple, while others also had additional restrictions. And 4 parsers had constraints on the nested grammars. Those are the more interesting cases and they contain both tuple assignments and list of strings. Coincidentally, we also observed our motivating example nearly identical in this category, where the *integer* constructor is applied to each element of a list (cf. Figure 1.2). Those more complex cases are explained in depth later on. To increase the readablity, we named the complex parsers according to their function name or functionality. The original filenames are given in their detailed descriptions.

Apart from the parsers, already known to be unsolvable, we also have some issues with one of the parsers that has a complex nested grammar. We will firstly discuss this parser that was unsuccessful, meaning that we could not synthesize a grammar, but the parser itself could be solvable according to our definition. In the case of the sample PARSE_DATE_TIME (6770_1425.py), seen in Figure 4.4, we encountered two issues. This sample manually parses a ISO time string and is therefore somewhat of a textbook

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

```
def version_to_list(value):
  for p in value.split('.'):
    try:
      n = int(p)
    except ValueError:
      pass
```

Python code for VERSION_TO_LIST.

```
parser = \(s: string).
   let p = splitList s "." in
   rec map : ∀#a.∀#b.f:(#a->#b) ->
        x:list(#a) -> {y:list(#b)| len(y) =
        len(x)} =
      Λ#a.Λ#b.\f:(#a->#b).\x:list(#a).
         switch(x)
         | Nil: Nil[#b]
         | Cons(z,zs):
            let zz = f z in
            let zzs = map[#a,#b] f zs in
            Cons[#b] zz zzs
   in
   let r = map[string,int] toInt p in
   assert true
```

λ_Σ^+ translation.

```
{s:string | s ∈[0−9][0−9]⋆(.?[0−9][0−9]⋆)⋆} → unit
```

Figure 4.5: Python code, λ_Σ^+ translation and inferred grammar for VERSION_TO_LIST.

example of an ad hoc parser. Translating those steps into λ_Σ^+ leads to multiple nested `split` applications. Firstly, Panini is not terminating due to a blowup of the resulting regular expressions during the abstract interpretation step. There is an issue with the simplification of regular expressions, which leads to this blowup. We tested this example with an improved regex simplification, that is not yet included in our version of Panini, and are able to infer this grammar correctly. Unfortunately, this version has some issues with other tests, so for this work, we are unable to infer this grammar, but we are confident that this issue does not persist. While this issue is unrelated to our work, we also encountered another issue, specifically with the introduced *GrammarLinks*. As we have multiple nested `split` applications, their translation order matters. Those two translations can also be seen in Figure 4.4. The nesting of those various `split` operations and the resulting *GrammarLinks* seem to be a problem. As the various κ variables all occur in circular dependencies to each other, in one of the shown cases, we do not encounter the case that Panini is not terminating. If the last `split` operation occurs before the *switch* statement relating to the second `split`, this does not happen. Unfortunately, due to time constraints, we were unable to figure out a solution to this problem. One solution might be to include those *GrammarLinks* in the ordering of the κ variables to be solved, or figure out a way to handle those circular dependencies differently. However, this issue does not affect λ_Σ^+, and we are confident that it can be solved.

To demonstrate the capabilities of Panini with our new extensions, we also show the successful examples. We start with the parser VERSION_TO_LIST (23916_465.py), in Figure 4.5, which is very similar to our motivating example, as it maps the integer constructor over a list. Here we also see that we can infer the grammar successfully for a higher order function, which is necessary to be able to handle ADTs successfully. In this case, we specifically wrote out the higher order *map* function, whereas in the example parser NOT_EXCLAMATIONMARK in Figure 4.7, a *forEach* function is simply defined as a

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

```
def sanitize_style(self, prop, value):
  if prop.lower() in
      self.acceptable_css_properties:
    pass
  elif prop.split('-')[0].lower() in
      ['background', 'border', 'margin',
      'padding']:
    for keyword in value.split():
      if (not keyword in
          self.acceptable_css_keywords and
          not
          self.valid_css_values.match(keyword)):
        break
```

Python code for SANITIZE_STYLE.

```
parser : {prop: string | ?} → unit
parser = \(prop: string).
  let isAcceptableCssProperty =
      isAcceptableCssProperty prop in
  if isAcceptableCssProperty then
    assert true
  else
    let split = splitTuple prop "-" in
    let first = fstTup[string, string]
        split in
    let isContainedIn = isContainedIn
        first in
    assert true
```

λ_Σ^+ translation.

```
parser : {prop:string | prop ∈
   azimuth|c(lea|(ol|urs)o)r|di(rection|splay)|elevation
   |f(loat|ont(-(family|s(iz|tyl)e|(varian|weigh)t))?)|height
   |overflow|p(ause(-(after|before))?|itch(-range)?)|richness
   |s(pe(ak(-(header|numeral|punctuation))?|ech-rate)|tress)
   |v(ertical-align|o(ice-family|lume))|w(hite-space|idth)
   |[^-]*-.*
   } → unit
```

Figure 4.6: Python code, λ_Σ^+ translation and inferred grammar for SANITIZE_STYLE.

function signature. Both cases work equally and can be inferred. This nested grammar resulting of the integer constructor is simply inserted into the *GrammarLink* to generate a valid grammar. The `split` operation is the first operation that is executed in this parser and therefore this result is not further constraint, which is not the case for the next example.

The second complex example for the parser SANITIZE_STYLE (36394_1046.py) is seen in Figure 4.6. This parser combines a nested grammar and a non nested grammar, depending on an if else branch. This branching is created using a list contains check. In general, this would concern ADTs, as we have a list of hard-coded values. Solving such a contains check, while using, e.g., a tuple with containing those hard-coded values, is not straightforward. The subtyping constraints in combination with the polymorphic nature of those ADTs leads to a loss of the relevant information. However, there is also a much simpler solution. If this contains check is hard-coded, we can simply create a boolean function that gets the input string and returns a boolean value which is refined by the condition if the input string is one of the given hard-coded values. This translation step is much simpler as creating an explicit data structure in λ_Σ^+. And we have used this technique for our translation, the `isAcceptableCssProperty` method encodes this contains check. This example shows that once a partial nested grammar is solved, the regular grammar solving idea of PANINI can continue regularly and PANINI is able to unify the grammars of those two branches. This is also the case for the parser 1613_194.py, which is one of those parses without nested constraints and shows that once a `GrammarLink` is resolved, PANINI treats it as any other grammar constraint that

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

```
def __init__(self, item):
    chunks = item.split(":")
    if len(chunks) < 2:
      raise ValueError("...")
    for c in chunks:
      if c == '!profile':
        pass
      elif c.startswith("!"):
        raise ValueError("...")
```

Python code for NOT_EXCLAMATION.

```
parser : {s: string | ?} —> unit
parser = \(s: string).
let chunks = splitList s ":" in
   let tmpFun = \x:string.
      let isProfile = match x "!profile" in
      if isProfile then
         assert true
      else
         let isStartsWith = startsWithC x
              '!' in
         if isStartsWith then
            assert false
         else
            assert true
   in
   forEach[string] chunks tmpFun
```

λ_{Σ}^{+} translation.

```
{s:string | s ∈
 ([^!].*|!profile(:([^!].*|!profi(le:!profi)*le(:([^!].*)?)?)?)?)?
} → unit
```

Figure 4.7: Python code, λ_{Σ}^{+} translation and inferred grammar for NOT_EXCLAMATION.

is incorporated into the abstract interpretation. Another translation detail for this parser is that the first element is directly accessed after a split. This has also been present in the sample 6770_1236.py, and we interpreted it as a split with direct tuple assignment (cf. section 4.3).

The parser NOT_EXCLAMATION (26554_645.py), iterates over a list of string, and checks if the string is either "!profile" or does not start with an exclamation mark. The abstract interpretation of this grammar results in a small blowup of the generated constraint that is sent to the SMT solver. Nonetheless, while this grammar is not as readable, it is still a valid result and the issues encountered for this parser, are not related to λ_{Σ}^{+} or the nested grammar solving approach. We have observed similar behavior for other examples and suspect that there is an issue with the regex simplification during the abstract interpretation approach, particularly with regular expressions that have quantification, similar to the example in Figure 4.4. However, for this example, this does not lead to an infinite runtime. Due to this issue, this parser also has a longer runtime than the other examples. The branching constraint leads to a more complex nested grammar, however, this complexity does not impact our approach, as this grammar is solved by Panini beforehand and simply used by the *GrammarLink*. Those successful examples represent typical cases for ad hoc parsers, and with λ_{Σ}^{+} and our nested grammar solving approach we are able to infer their grammars.

We also want to outline for the three unsolvable cases what the specific parsing operations were that led to their classification. This is just to give an idea of what types of parsing steps are difficult to infer. However, due to the limited sample size, we cannot give any numbers on how common these cases are, and which case should be investigated further to identify potential improvements for Panini.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
def addheader(self, key,
  value, prefix = 0,
  add_to_http_hdrs = 0):
 lines = value.split("\r\n")
 if add_to_http_hdrs:
  pass
 else:
  for i in range(1,
    len(lines)):
   lines[i] = " " +
     lines[i].strip()
```

6770_1339.py

```
def _is_hostmask(self,
  ip_str):
 bits = ip_str.split(’.’)
 try:
  parts = [int(x) for x in
    bits if int(x) in
    self._valid_mask_octets]
 except ValueError:
  return False
 if len(parts) != len(bits):
  return False
```

26554_610.py

```
def parse(fromstring):
 for op in XNSelector.BASIC_OPERATORS:
  operands = fromstring.split(op)
  if len(operands) > 1:
   for field, selector in XNSelector.SPECIAL_SELECTORS.items():
    if operands[0].strip() == field:
     return globals()(op, map(str.strip, operands))
   return XNSelector(op, map(str.strip, operands))
```

20996_251.py

Figure 4.8: Sample parsers classified as unsolvable.

The parser 6770_1339.py is one of the cases that in theory could be solvable. The nested grammar is simply a call to *strip*, so the overall grammar is still unconstrained, but it was classified as unsolvable, as there is an iteration starting from the second list element. As mentioned ind section 3.5, those cases where different nested grammars occur at fixed indices are theoretically solvable, but not by our current implementation. More challenging is the sample parser 26554_610.py. It is similar to the motivating example, as a `toInt` constructor is mapped over a list of strings. However, this list is then filtered by an integer condition. The parser fails if the resulting filtered list has a smaller length than the original list. Apart from the fact that the integer conditions do not (yet) influence the input restriction of the `toInt` constructor, it is also unclear if the relationship between the length of those lists, and the filtering can be represented by the refinement logic. This grammar is still a regular grammar, but we expect the Panini approach to not be sufficient for such complex cases. The last unsolvable example, 20996_251.py, also has two issues. Firstly, there is a length check of the list, which could be handled in theory as previously outlined. Secondly, the input string is split multiple times by a list of different separators, and we do not know how to handle and or unify this case.

In general, we have shown that we can infer many nested grammars as well as complex nested grammars using our approach. If those grammars are solvable as defined, only one case had either a problem with the outlined nested grammar solving approach or

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

an issue with simplifying the resulting regular expressions. In the other case we had a correct grammar, but a timeout in the final SMT solver step. Overall, the issues for complex nested grammars seem to be due to both, simplifying regular expressions and their handling in the SMT solver.

4.3 Threats to Validity

The most relevant threat to the validity of our findings, is the size of our evaluation dataset. The amount of parsers we were able to evaluate was limited by the effort imposed by the manual translation from Python source code to λ_Σ^+. Overall, we evaluated 18 real-world ad hoc parsers, which we think is suitable to observe the usability of λ_Σ^+ in practice. However, this is a small sample size and only 4 of the examined parsers utilized a complex grammar with nested constraints, which especially restricts concrete statements about the behavior of λ_Σ^+ for those complex cases. However, those examples, combined with the synthetic examples, show that we are able to infer nested grammars that arise due to `split` operations in practice.

Another threat is that the dataset might not be representative. However, the overall idea is to show λ_Σ^+ for real-world ad hoc parsers, and we clearly show that we can infer complex nested grammars for many of those parsers. Determining how those parsers behave in general, and to give an exact understanding for how many of them we can infer a grammar, is part of our future work. For this work, we first and foremost wanted to introduce the underlying λ_Σ^+ language. This extension is necessary, and based on the observed real-world parsers, the current state is powerful enough to significantly infer more grammars than before.

And finally, we also need to discuss the manual translation. We took some liberties when translating the selected parsers. This entails the definitions of `split`, and some additionally limits of the manual translation.

We kept the *GrammarLink* behavior simple, as we first wanted to show the feasibility of this approach. Currently, the *GrammarLink* can only combine simple regular expressions. However, for the `split` method, additional information is available based on the length of the returned list and the separator value. From the semantic of `split`, we know that the nested grammar cannot contain the separator, and the length of a resulting list influences the quantification of the nested grammar. The length information of a list is sometimes known, such as when a length check is performed or an index is accessed. Additionally, for each supported programming language in Panini, these definitions would also need adjustments to handle edge cases specific to that language, which we have not addressed here. Therefore, the given definitions can be considered as simplified examples, and necessary, as the current implementation of the nested grammar solving approach in Panini is still limited. However, once the nested grammars are solved, there is full control over how the overall grammar is constructed, and adding those features should be straight forward. Since these restrictions do not affect the solvability of nested grammars, we chose to ignore them from our evaluation.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Parser	Issue
6770_1391.py	Index access with unknown indices
24871_516.py	Unknown function *generate_parent_dirs* in extra branch and index access
26554_645.py	Length check with exception
36394_1046.py	Call to `lower` function

Table 4.3: Translation steps that have been ignored.

Apart from list `length` and index checks, we also ignored some additional steps in our translation effort. Firstly, we encountered one call to the `lower` function. This was ignored, as Panini can not yet handle this case, and this is also not related to our additions. However, as mentioned section 3.5, the *GrammarLink* approach could also be applicable for this case. And secondly, we encountered a function call to an unknown local function in an extra branch of one parser. As this information has not been extracted from the original data and is not relevant for this evaluation, this was also ignored. Overall, for the translated parsers, those additional restrictions to the translation can be seen in Table 4.3.

As a result, some parsers were not translated as fully as they could have been. This aligns with the goal of our evaluation, which is to demonstrate how real-world ad hoc parsers and parsing steps could appear in λ_Σ^+ and to assess grammar inference for these cases, rather than to achieve complete translation. We mainly care about the employed parsing constructs, and not (yet) about the concrete definitions for split or the translation step. This is enough to show the viability of this approach and leads us to the answer for our second research question:

RQ2 For what kind of programs can we synthesize input string grammars with λ_Σ^+?

Based on a dataset of 18 parsers, we have shown that we can infer grammars for many real-world ad hoc parsers, even in complex cases. Our expectations, outlined in section 3.5, were correct in the sense that we can categorize the observed ad hoc parsers into three categories. This is trivially true by design, but we have seen that it is also relevant in practice.

We categorized parsers into five classes: (1) homogenous lists without constraints, (2) heterogeneous lists without constraints, (3) homogenous lists with constraints, (4) heterogeneous lists with constraints, and (5) unsolvable nested grammars.

Of these, we could easily infer grammars for the first two categories. For parsers with nested constraints, we only encountered one parser for which we could not synthesize a grammar. However, the encountered issues are not related to the extended λ_Σ^+ language. Those issues arise either due to shortcomings in the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

TU WIEN Bibliothek Your knowledge hub

abstract interpretation of regular expressions or due to an issue with the introduced *GrammarLinks*.

We mostly encountered simple examples for parsers utilizing our extensions, which limits our ability to be able to identify further shortcomings of the Panini system. However, this would require an extensive as well as automated analysis of real-world ad hoc parsers, which we will discuss in our future work.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 5

Future Work and Conclusions

Overall, our research has shown that λ_{Σ}^{+} in combination with the proposed solution for nested grammars is able to synthesize string grammars not only for synthetic examples, but also for real-world parsers.

However, our enhancements do not change the fact that Panini remains a prototype implementation and many necessary steps still need to be taken to enhance its functionality. We will outline those steps and discuss the limitations of our work here, as this future work aims to address these shortcomings.

5.1 Future Work and Limitiations

In general, the majority of our future work does not relate to the extended λ_{Σ}^{+} as we expect this part of our work to be complete. Considering the introduced inference rules, it could still be possible that generating equivalent, but different, VCs is necessary for future extensions. However, we expect the extended λ_{Σ}^{+} to be suitable as is while further improvements are necessary to Panini and the nested grammar solving approach. We are clearly bound by the capabilities of refinement type systems as this is the underlying restriction by Panini. The changes we have implemented are all in line with similar refinement type systems, and we have identified no issues specific to the added language features. We also specifically did not encounter any issues regarding our novel inference rule for ADT destruction. However, this novel inference rule has not been formally proven to be correct. Therefore, the next step is a full correctness proof of the underlying λ_{Σ}^{+} calculus, to increase the confidence in our theoretical foundations.

Regarding the capabilities of Panini, we need to add the λ_{Σ}^{+} extensions to the automatic translation layer, as they are not yet supported. This is definitely an area of Panini that needs to be implemented, before the expected use case becomes viable. To this end, multiple steps need to be taken. As explained in section 3.5, we need to automatically translate `split` function calls into n-tuples or list data types, depending on the access. In Python, this is less challenging in the observed cases, but it might be difficult in other languages. Additionally, as λ_Σ^+ currently relies on generated Hindley-Miller type annotation for polymorphism, those also need to be generated automatically during the translation step. In combination with the currently ongoing work to automatically extract real-world ad hoc parsers, this allows us to automatically evaluate the current state of Panini with our new additions. This is an important part of our future work, as this information is vital to be able to identify further improvements, especially for the outlined currently unsolvable parsers. Additionally, having more information about the nature of ad hoc parsers enables us to verify the well-founded assumptions we made for this work, such as the behavior of the introduced *GrammarLinks* or the proposed categorizations.

Our solution to solve nested grammars remains a proof of concept and needs to be extended. Most importantly, we need to figure out if and how circular dependencies between κ variables can be handled for nested *GrammarLinks*. As we have seen in our evaluation, we observed one case (cf. Figure 4.4) where the ordering of programming statements made a difference to solvability during the grammar solving step. In order to identify a solution for highly nested *GrammarLinks*, this issue needs to be investigated further.

Apart from this apparent limitation, there are also further necessary improvements to the nested grammar solving approach, as we need to evaluate how to include additional constraints into this approach. For this work, we did implement the definitions for *split* in a basic manner, as this was enough to show the feasibility of our approach. As future work, we now need to include further information in the generated regular expressions like list length and excluding the separator values in the nested grammars. To this end, the *GrammarLinks* should be extended to include the length of a list. For simple cases, like a simple length check, this might not be a problem, but there are also complex cases that would need to be handled. Additionally, when constructing the resulting grammar in a *GrammarLink*, the separator needs to be excluded from the nested regular expression. We also need to include programming language specific information in those definitions and for other language features like the integer constructor. As those definitions are simple type definitions in λ_Σ^+, they can easily be adapted.

Finally, we classified a number of parsers as unsolvable due to the outlined nested grammar solving approach. Due to the underlying refinement type system, there are grammars we can not infer, and this first classification might be too broad, it is suitable as it enables us to build upon this work. Further additions to array handling could specifically aid with more complex heterogeneous nested grammars, as previously mentioned (cf. section 2.4). For this work, this was not the main objective and this requires extensive and concrete information about the nature of real-world ad hoc parsers, to identify the needed necessary capabilities. The specific additions to λ_Σ^+ are necessary nonetheless, as this was the step that enables us to regard those complex cases in the first place.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

5.2 Conclusions

In this work, we have introduced the extended λ_Σ^+ and its rules. This lays the foundation to synthesize more string grammars than previously possible.

We have implemented type inference and subtyping rules for Polymorphism and ADTs, and introduced inference rules for data type destruction. To our knowledge, those inference rules for data type destruction are novel, and we have shown their practical application and how the generated VC can be used for grammar synthesis. To explain our newly added typing rules, we provided an in-depth explanation of verification condition generation for liquid type systems for these rules. This also allowed us to explain how λ_Σ^+ integrates into Panini in the context of grammar synthesis.

To increase those grammar synthesis capabilities, we have discussed the types of nested grammars that can arise using these new language features and introduced an approach to infer those nested grammars. This allowed us to evaluate our work on real-world ad hoc parsers, and we demonstrated which kind of nested grammars we can synthesize. We outlined the theoretical limitations of this approach, as well as shown the concrete limitations that are still present after adding λ_Σ^+ to Panini. Those limitations include regular grammars that could in theory be solved by the Panini approach and are also present in the observed real-world ad hoc parsers. Despite those challenges, we have shown that we can successfully synthesize either homogenous nested grammars or fixed length heterogeneous nested grammars, which was not possible before. As this represents a significant extension, we have discussed multiple useful further improvements to Panini and this work lays important and necessary groundwork for enabling those changes.

In addition to the fundamental new capabilities of λ_Σ^+, we also expect the majority of parsers with nested grammars to fall into the outlined solvable categories. Therefore, while potential future work is necessary, our contributions are relevant for the majority of parsers that previously could not have been handled by Panini, but where the Panini approach is applicable in general. In sum, our work is a significant extension to the grammar solving capabilities that were previously implemented into Panini. It enables us to synthesize many more relevant real-world ad hoc parsers.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Figures

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Tables

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Acronyms

ADT algebraic data type. xi, 3–5, 19, 20, 22, 27, 34–36, 41–43, 48, 49, 55, 57, 59

ANF A-normal form. 2, 11

QF_UFLIA quantifier-free theory of linear arithmetic and uninterpreted functions. 7, 11

SMT Satisfiability Modulo Theories. 7, 8, 17, 42, 50, 52

VC verification condition. 7–11, 13–16, 20, 22, 24, 25, 28–35, 37, 42, 55, 57, 59

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Bibliography

[1] Peter Achten and Pieter Koopman, eds. *The Beauty of Functional Code: Essays Dedicated to Rinus Plasmeijer on the Occasion of His 61st Birthday*. en. Vol. 8106. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013. ISBN: 978-3-642-40354-5 978-3-642-40355-2. DOI: 10.1007/978-3-642-40355-2. URL: http://link.springer.com/10.1007/978-3-642-40355-2 (visited on 10/31/2024).

[2] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. Published: \tt www.SMT-LIB.org. 2016.

[3] Jesper Bengtson et al. “Refinement types for secure implementations”. In: *ACM Trans. Program. Lang. Syst.* 33.2 (Feb. 2011), 8:1–8:45. ISSN: 0164-0925. DOI: 10.1145/1890028.1890031. URL: https://dl.acm.org/doi/10.1145/1890028.1890031 (visited on 09/17/2024).

[4] Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan, and Andrey Rybalchenko. “Horn Clause Solvers for Program Verification”. en. In: *Fields of Logic and Computation II: Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*. Ed. by Lev D. Beklemishev et al. Cham: Springer International Publishing, 2015, pp. 24–51. ISBN: 978-3-319-23534-9. DOI: 10.1007/978-3-319-23534-9_2. URL: https://doi.org/10.1007/978-3-319-23534-9_2 (visited on 09/19/2024).

[5] Michael H. Borkowski, Niki Vazou, and Ranjit Jhala. “Mechanizing Refinement Types”. In: *Artifact for "Mechanizing Refinement Types"* 8.POPL (Jan. 2024), 70:2099–70:2128. DOI: 10.1145/3632912. URL: https://dl.acm.org/doi/10.1145/3632912 (visited on 08/14/2024).

[6] Benjamin Cosman and Ranjit Jhala. “Local refinement typing”. In: *Proceedings of the ACM on Programming Languages* 1.ICFP (Aug. 2017), 26:1–26:27. DOI: 10.1145/3110270. URL: https://dl.acm.org/doi/10.1145/3110270 (visited on 12/05/2023).

[7] Luis Damas and Robin Milner. "Principal type-schemes for functional programs". In: *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL '82. New York, NY, USA: Association for Computing Machinery, Jan. 1982, pp. 207–212. ISBN: 978-0-89791-065-1. DOI: 10.1145/582153.582176. URL: https://dl.acm.org/doi/10.1145/582153.582176 (visited on 10/15/2024).

[8] Jana Dunfield. "Refined typechecking with Stardust". In: *Proceedings of the 2007 workshop on Programming languages meets program verification*. PLPV '07. New York, NY, USA: Association for Computing Machinery, Oct. 2007, pp. 21–32. ISBN: 978-1-59593-677-6. DOI: 10.1145/1292597.1292602. URL: https://dl.acm.org/doi/10.1145/1292597.1292602 (visited on 09/17/2024).

[9] Jana Dunfield and Neel Krishnaswami. "Bidirectional Typing". In: *ACM Computing Surveys* 54.5 (May 2021), 98:1–98:38. ISSN: 0360-0300. DOI: 10.1145/3450952. URL: https://dl.acm.org/doi/10.1145/3450952 (visited on 02/01/2024).

[10] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. "Boa: a language and infrastructure for analyzing ultra-large-scale software repositories". In: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE '13. San Francisco, CA, USA: IEEE Press, May 2013, pp. 422–431. ISBN: 978-1-4673-3076-3. (Visited on 11/21/2024).

[11] Robert W. Floyd. "Assigning Meanings to Programs". en. In: *Program Verification: Fundamental Issues in Computer Science*. Ed. by Timothy R. Colburn, James H. Fetzer, and Terry L. Rankin. Dordrecht: Springer Netherlands, 1993, pp. 65–81. ISBN: 978-94-011-1793-7. DOI: 10.1007/978-94-011-1793-7_4. URL: https://doi.org/10.1007/978-94-011-1793-7_4 (visited on 09/10/2024).

[12] Tim Freeman and Frank Pfenning. "Refinement types for ML". en. In: *ACM SIGPLAN Notices* 26.6 (June 1991), pp. 268–277. ISSN: 0362-1340, 1558-1160. DOI: 10.1145/113446.113468. URL: https://dl.acm.org/doi/10.1145/113446.113468 (visited on 11/23/2023).

[13] D. J. Gilmore and T. R. G. Green. "Comprehension and recall of miniature programs". In: *International Journal of Man-Machine Studies* 21.1 (July 1984), pp. 31–48. ISSN: 0020-7373. DOI: 10.1016/S0020-7373(84)80037-1. URL: https://www.sciencedirect.com/science/article/pii/S0020737384800371 (visited on 09/13/2024).

[14] R. Hindley. "The principle type-scheme of an object in combinatory logic". en. In: *Transactions of the American Mathematical Society* 146 (1969), pp. 29–60. ISSN: 0002-9947, 1088-6850. DOI: 10.1090/S0002-9947-1969-0253905-6. URL: https://www.ams.org/tran/1969-146-00/S0002-9947-1969-0253905-6/ (visited on 10/15/2024).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

[15] C. A. R. Hoare. “Procedures and parameters: An axiomatic approach”. en. In: *Symposium on Semantics of Algorithmic Languages*. Ed. by E. Engeler. Berlin, Heidelberg: Springer, 1971, pp. 102–116. ISBN: 978-3-540-36499-3. DOI: 10.1007/BFb0059696.

[16] Ranjit Jhala, Andreas Podelski, and Andrey Rybalchenko. “Predicate Abstraction for Program Verification”. en. In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. Cham: Springer International Publishing, 2018, pp. 447–491. ISBN: 978-3-319-10575-8. DOI: 10.1007/978-3-319-10575-8_15. URL: https://doi.org/10.1007/978-3-319-10575-8_15 (visited on 09/19/2024).

[17] Ranjit Jhala and Niki Vazou. *Refinement Types: A Tutorial*. en. arXiv:2010.07763 [cs]. Oct. 2020. URL: http://arxiv.org/abs/2010.07763 (visited on 11/07/2023).

[18] Ming Kawaguchi, Patrick Rondon, and Ranjit Jhala. “Type-based data structure verification”. In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’09. New York, NY, USA: Association for Computing Machinery, June 2009, pp. 304–315. ISBN: 978-1-60558-392-1. DOI: 10.1145/1542476.1542510. URL: https://dl.acm.org/doi/10.1145/1542476.1542510 (visited on 08/14/2024).

[19] Tristan Knoth, Di Wang, Nadia Polikarpova, and Jan Hoffmann. “Resource-guided program synthesis”. In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. New York, NY, USA: Association for Computing Machinery, June 2019, pp. 253–268. ISBN: 978-1-4503-6712-7. DOI: 10.1145/3314221.3314602. URL: https://dl.acm.org/doi/10.1145/3314221.3314602 (visited on 09/18/2024).

[20] Tristan Knoth et al. “Liquid resource types”. In: *Artifact for Liquid Resource Types* 4.ICFP (Aug. 2020), 106:1–106:29. DOI: 10.1145/3408988. URL: https://dl.acm.org/doi/10.1145/3408988 (visited on 08/30/2024).

[21] Nico Lehmann, Adam Geller, Niki Vazou, and Ranjit Jhala. *Flux: Liquid Types for Rust*. arXiv:2207.04034 [cs]. Nov. 2022. DOI: 10.48550/arXiv.2207.04034. URL: http://arxiv.org/abs/2207.04034 (visited on 11/28/2023).

[22] Nico Lehmann et al. “{STORM}: Refinement Types for Secure Web Applications”. en. In: 2021, pp. 441–459. ISBN: 978-1-939133-22-9. URL: https://www.usenix.org/conference/osdi21/presentation/lehmann (visited on 08/30/2024).

[23] Daan Leijen and Erik Meijer. “Parsec: Direct Style Monadic Parser Combinators For The Real World”. In: (Dec. 2001).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[24] Falcon Momot, Sergey Bratus, Sven M. Hallberg, and Meredith L. Patterson. “The Seven Turrets of Babel: A Taxonomy of LangSec Errors and How to Expunge Them”. In: *2016 IEEE Cybersecurity Development (SecDev)*. Nov. 2016, pp. 45–52. DOI: 10.1109/SecDev.2016.019. URL: https://ieeexplore.ieee.org/document/7839788 (visited on 09/27/2024).

[25] Manuel Montenegro, Susana Nieva, Ricardo Peña, and Clara Segura. “Extending Liquid Types to Arrays”. In: *ACM Transactions on Computational Logic* 21.2 (Jan. 2020), 13:1–13:41. ISSN: 1529-3785. DOI: 10.1145/3362740. URL: https://dl.acm.org/doi/10.1145/3362740 (visited on 02/01/2024).

[26] Manuel Montenegro, Susana Nieva, Ricardo Peña, and Clara Segura. “Liquid Types for Array Invariant Synthesis”. en. In: *Automated Technology for Verification and Analysis*. Ed. by Deepak D’Souza and K. Narayan Kumar. Cham: Springer International Publishing, 2017, pp. 289–306. ISBN: 978-3-319-68167-2. DOI: 10.1007/978-3-319-68167-2_20.

[27] Charles Gregory Nelson. “Techniques for Program Verification”. PhD Thesis. Stanford University, 1980.

[28] T. J. Parr and R. W. Quong. “ANTLR: A predicated-LL(k) parser generator”. en. In: *Software: Practice and Experience* 25.7 (1995). _eprint: https://onlinelibrary.wiley.com/doi/pdf/10. pp. 789–810. ISSN: 1097-024X. DOI: 10.1002/spe.4380250705. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380250705 (visited on 09/13/2024).

[29] Benjamin C. Pierce. *Types and programming languages*. en. Cambridge, Massachusetts London, England: The MIT Press, 2002. ISBN: 978-0-262-16209-8.

[30] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. *Program Synthesis from Polymorphic Refinement Types*. arXiv:1510.08419 [cs]. Apr. 2016. DOI: 10.48550/arXiv.1510.08419. URL: http://arxiv.org/abs/1510.08419 (visited on 08/26/2024).

[31] Nadia Polikarpova et al. “Liquid information flow control”. In: *Proceedings of the ACM on Programming Languages* 4.ICFP (Aug. 2020), 105:1–105:30. DOI: 10.1145/3408987. URL: https://dl.acm.org/doi/10.1145/3408987 (visited on 01/16/2024).

[32] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. “Liquid types”. In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’08. New York, NY, USA: Association for Computing Machinery, June 2008, pp. 159–169. ISBN: 978-1-59593-860-2. DOI: 10.1145/1375581.1375602. URL: https://dl.acm.org/doi/10.1145/1375581.1375602 (visited on 11/07/2023).

[33] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. “Low-level liquid types”. In: *SIGPLAN Not.* 45.1 (Jan. 2010), pp. 131–144. ISSN: 0362-1340. DOI: 10.1145/1707801.1706316. URL: https://dl.acm.org/doi/10.1145/1707801.1706316 (visited on 09/17/2024).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[34] Michael Schröder. “Grammar Inference for Ad Hoc Parsers”. In: *Companion Proceedings of the 2022 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. SPLASH Companion 2022. New York, NY, USA: Association for Computing Machinery, Dec. 2022, pp. 38–42. ISBN: 978-1-4503-9901-2. DOI: 10.1145/3563768.3565550. URL: https://dl.acm.org/doi/10.1145/3563768.3565550 (visited on 01/01/2024).

[35] Michael Schröder and Jürgen Cito. *Static Inference of Regular Grammars for Ad Hoc Parsers*. Under submission. 2024. URL: https://mcschroeder.github.io/files/panini_preprint.pdf.

[36] Michael Schröder, Marc Goritschnig, and Jürgen Cito. *An Exploratory Study of Ad Hoc Parsers in Python*. Registered Report. arXiv:2304.09733 [cs]. Apr. 2023. DOI: 10.48550/arXiv.2304.09733. URL: http://arxiv.org/abs/2304.09733 (visited on 12/10/2023).

[37] Michael Schröder, Andreas Olschnögger, Marc Goritschnig, and Jürgen Cito. *An Exploratory Study of Ad Hoc Parsers in Python*. In progress.

[38] Niki Vazou. *A Gentle Introduction to Liquid Types*. Sept. 2015. URL: https://goto.ucsd.edu/~ucsdpl-blog/liquidtypes/2015/09/19/liquid-types/ (visited on 08/29/2024).

[39] Niki Vazou, Alexander Bakst, and Ranjit Jhala. *Bounded Refinement Types*. arXiv:1507.00385 [cs]. July 2015. DOI: 10.48550/arXiv.1507.00385. URL: http://arxiv.org/abs/1507.00385 (visited on 09/03/2024).

[40] Niki Vazou, Patrick M. Rondon, and Ranjit Jhala. “Abstract Refinement Types”. en. In: *Programming Languages and Systems*. Ed. by Matthias Felleisen and Philippa Gardner. Berlin, Heidelberg: Springer, 2013, pp. 209–228. ISBN: 978-3-642-37036-6. DOI: 10.1007/978-3-642-37036-6_13.

[41] Niki Vazou, Eric L. Seidel, and Ranjit Jhala. “LiquidHaskell: experience with refinement types in the real world”. In: *SIGPLAN Not.* 49.12 (Sept. 2014), pp. 39–51. ISSN: 0362-1340. DOI: 10.1145/2775050.2633366. URL: https://dl.acm.org/doi/10.1145/2775050.2633366 (visited on 09/27/2024).

[42] Niki Vazou et al. “Refinement types for Haskell”. en. In: *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*. Gothenburg Sweden: ACM, Aug. 2014, pp. 269–282. ISBN: 978-1-4503-2873-9. DOI: 10.1145/2628136.2628161. URL: https://dl.acm.org/doi/10.1145/2628136.2628161 (visited on 11/28/2023).

[43] Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. *Refinement Types for TypeScript*. arXiv:1604.02480 [cs]. Apr. 2016. DOI: 10.48550/arXiv.1604.02480. URL: http://arxiv.org/abs/1604.02480 (visited on 09/17/2024).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

[44] Alessandro Warth and Ian Piumarta. “OMeta: an object-oriented language for pattern matching”. In: *Proceedings of the 2007 symposium on Dynamic languages*. DLS ’07. New York, NY, USA: Association for Computing Machinery, Oct. 2007, pp. 11–19. ISBN: 978-1-59593-868-8. DOI: 10.1145/1297081.1297086. URL: https://dl.acm.org/doi/10.1145/1297081.1297086 (visited on 09/13/2024).

[45] A. K. Wright and M. Felleisen. “A Syntactic Approach to Type Soundness”. In: *Information and Computation* 115.1 (Nov. 1994), pp. 38–94. ISSN: 0890-5401. DOI: 10.1006/inco.1994.1093. URL: https://www.sciencedirect.com/science/article/pii/S0890540184710935 (visited on 10/11/2024).

[46] Hongwei Xi and Frank Pfenning. “Eliminating array bound checking through dependent types”. In: *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*. PLDI ’98. New York, NY, USA: Association for Computing Machinery, May 1998, pp. 249–257. ISBN: 978-0-89791-987-6. DOI: 10.1145/277650.277732. URL: https://dl.acm.org/doi/10.1145/277650.277732 (visited on 09/17/2024).

[47] Andreas Zeller et al. “Fuzzing with Grammars”. In: *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2024. URL: https://www.fuzzingbook.org/html/Grammars.html (visited on 09/10/2024).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.