KTH ROYAL INSTITUTE
OF TECHNOLOGY

Degree Project in Computer Science

Second cycle, 30 credits

Sound Language Composition under Product Extension

Enabling Rapid Development of Domain-Specific Language in the Miking Framework

MARTEN VOORBERG

Stockholm, Sweden, 2024

Abstract

Domain-Specific Languages (DSLs) are programming languages that facilitate the development of programs in a *specific domain*. Their primary advantages are increased expressiveness and usability. The disadvantage of DSLs is the fact that their development is costly. The Miking framework attempts to alleviate this downside by facilitating the rapid development of DSLs.

The key insight of the Miking framework is that DSLs share many programming constructs. When developing compilers in the Miking framework, a programming language consists of many *language fragments* that each model a specific construct in the language. The Miking framework provides the notion of *language composition* that combines the aforementioned fragments into a single language which inherits functionality from all its components. Using these techniques, we do not have to create the entire compiler for a DSL from scratch, but can instead reuse existing language fragments.

The current notion of language composition in the Miking framework centers around *sum extension*: The addition of *new* constructs to a language fragment. One issue during DSL development is that the composition of two fragments through sum extension does not necessarily result in a well-defined language. The reason for this is sum extension can potentially cause *inexhaustive matches*. To address this, the Miking framework has a notion of *sound language composition* in which a type system ensures that the resulting language is in fact well-defined.

This thesis expands the work on language composition in the Miking framework by extending the notion of language composition with *product extension*: The addition of new information to *existing constructs*. Whereas language composition through sum extension is potentially unsound due to in-exhaustive matches, product extension introduces the possibility of *illegal projections*. This thesis recovers the soundness of language composition under product extension by extending the existing Miking type system with the notion of extensible product types.

Keywords

Product Extension, Language Composition, Sound Language Composition, Domain-Specific Language, Language Workbench, Miking, Type System, Record Polymorphism, Extensible Product Types

Sammanfattning

Domänspecifika språk (DSL) är programmeringsspråk som underlättar utvecklingen av program i en *specifik domän*. Deras främsta fördelar är ökad uttrycksförmåga och användbarhet. Nackdelen med DSL är det faktum att deras utveckling är kostsam. Miking-ramverket försöker lindra denna nackdel genom att underlätta den snabba utvecklingen av DSL.

Den viktigaste insikten i Miking-ramverket är att DSL:er delar många programmeringskonstruktioner. När man utvecklar kompilatorer i Miking-ramverket består ett programmeringsspråk av många *språkfragment* som varje modellerar en specifik konstruktion i språket. Miking-ramverket tillhandahåller begreppet *språkkomposition* som kombinerar de tidigare nämnda fragmenten till ett enda språk som ärver funktionalitet från alla dess komponenter. Med dessa tekniker behöver vi inte skapa hela kompilatorn för en DSL från början, utan kan istället återanvända befintliga språkfragment.

Den nuvarande uppfattningen om språkkomposition i Miking-ramverket kretsar kring *sum extension*: tillägget av *new* konstruktioner till ett språkfragment. En fråga under DSL-utveckling är att sammansättningen av två fragment genom summaförlängning inte nödvändigtvis resulterar i ett väldefinierat språk. Anledningen till detta är att summaförlängning potentiellt kan orsaka *ofullständiga matchningar*. För att ta itu med detta har Miking-ramverket en föreställning om *ljudspråkskomposition* där ett typsystem säkerställer att det resulterande språket faktiskt är väldefinierat.

Den här avhandlingen utökar arbetet med språkkomposition i Miking-ramverket genom att utöka begreppet språkkomposition med *product extension*: The addition of new information to existing constructs. Där språksammansättning genom summaförlängning är potentiellt osunda på grund av uttömmande matchningar, introducerar produkttillägg möjligheten för *olagliga projektioner*. Denna avhandling återställer sundheten i språksammansättningen under produktutvidgning genom att utöka det befintliga systemet av Miking-typ med begreppet utvidgbara produkttyper.

Acknowledgments

I was lucky enough to surrounded by the knowledgeable and supportive Miking development team throughout this thesis. Special thanks go out to my co-supervisor David Broman, who always managed to find time in his ever-busy schedule. I would like to thank my co-supervisor Viktor Palmkvist for his feedback throughout the thesis and for always being able to look 10 steps ahead. My thanks also go out to Anders Thuné from Uppsala University. The work in this thesis on the MLang type system heavily relies on his earlier work on Constructor Types and his feedback throughout this work has been very valuable. Additionally, I'd like to extend my gratitude to Benjamin Driscoll from Stanford University. He originally proposed the addition of co-syntax types and co-semantic functions to MLang. I'd like to thank Didrik Munther for helping me with the Swedish translation of my abstract. My thanks go out to the rest of the Miking team I've worked with: Lars Hummelgren, John Wikman, Gizem Caylak, Elias Castegren, and Oscar Eriksson.

Stockholm, December 2024
Marten Voorberg

Contents

Chapter 1

Introduction

In this chapter, we start by introducing Domain Specific Languages (DSLs) and their advantages and disadvantages. We provide a brief introduction to the Miking framework which enables the rapid development of DSLs through language composition. The following section gives an extensive overview of language composition in the Miking framework. After this, we state the research problem that is solved by this thesis. We proceed to briefly touch on the ethical aspects and societal impact of this work. The chapter ends with an overview of the contributions and an outline of this thesis.

1.1 Rapid DSL Development through Language Composition

Domain Specific Languages (or DSLs) are programming languages that have been created to solve problems in a specific domain [1]. Due to the fact that a DSL is heavily tailored to a single domain, programs can be created in this domain without requiring their users to have extensive background in computer science. Furthermore, DSLs can often provide greater expressivity [2]. The flip side is that the development of DSLs is known to be very time-consuming and expensive. To address this downside, the creation of tools and frameworks to facilitate easier and cheaper development of DSLs is an active area of research.

The Miking framework [3] enables the rapid development of DSLs by allowing its users to compose languages. The Miking framework recognizes the fact that many DSLs make use of the same basic building

blocks, such as lambda abstractions, records, tensors, etc. The Miking framework allows the creation of small language fragments containing the required functionality for such individual building blocks. A DSL designer can then create their DSL by picking and choosing their desired language features and combining these into their new language through language composition. Through this method, a DSL designer can focus on the unique aspects of their language and they do not have to waste any time implementing standard language constructs.

1.2 Sound Language Composition in Two Dimensions

The term *language composition* [4] is used to describe the definition of a new programming language in terms of other, typically smaller programming languages. In the Miking framework, language composition takes the form of *sum extension*: The extension of a language by adding *new* constructs. The composition of languages through sum extension turns out to be unsound. Existing work on the framework recovers the soundness of language composition under sum extension in the Miking framework through a type system over extensible sum types.

The primary focus of this thesis is to extend the notion of language composition in the Miking framework with a second dimension, called *product extension*. This second dimension allows us to extend a language by adding additional information to *existing* constructs. Similarly to sum extension, the composition of languages through product extension is also unsound. The soundness of language composition under product extension is recovered using a type system over extensible product types.

The rest of this section elaborates on the concepts of sound language composition, sum extension, and product extension in the Miking framework. We do this by implementing a small interpreter for the traditional lambda calculus and a small arithmetic language in MCore, the Miking framework's programming language. These small languages are then *composed* through sum extension to create a lambda calculus supporting arithmetic. Subsequently, we extend this new language with a type system through product extension. We exemplify the key aspects of DSL development through code snippets but omit certain details for the sake of brevity. The full example can be found in Appendix A.

1.2.1 A Basic Language Fragments

Language fragments in MLang are modeled after the formal representation of programming languages in programming language literature. In this field, a programming language can be expressed as a combination of abstract *syntax* and its *semantics*. In the context of the Miking framework, this combination is referred to as a *language fragment*. The abstract syntax describes the various constructs (e.g. types, expressions, statements, terms) that are present in the language. For example, the abstract syntax of Church's lambda calculus consists of terms, where each term is either an abstraction, application, or variable. In programming languages theory, we often specify the abstract syntax as a BNF grammar as in Equation 1.1.

The semantics of a programming language specify the "meaning" of a language. In programming language design, it is common to split the semantics of a language into *dynamic* and *static* semantics. Dynamic semantics specify the behavior of the syntax whilst static semantics describe the validation of the syntax. In programming language theory, we describe the semantics through a mathematical relation which is specified through inference rules as in Equation 1.2 or through structural induction as in Equation 1.3.

In Figure 1.1, we formally define an untyped lambda calculus with call-by-value, small-step semantics based through a combination of syntax and semantics*.

MLang, the programming language of the Miking framework, represents language fragments in terms of *syntax types* and *semantic functions*, indicated by the keywords `syn` and `sem` respectively. The MLang language fragment modeling the aforementioned lambda calculus can be found in Listing 1.1.

A syntax type represents the abstract syntax of a language as a *sum type over product types*. The term sum type comes from the fact that the set of all inhabitants of such a type is the sum of the inhabitants of its constructors. E.g. the set of all terms in the type `Term`, is the set of terms constructed through `TmVar` *plus* the those constructed through `TmAbs` *plus* those of `TmApp`. Each constructor specifies their fields through a product type, specifically a record in our example.

In MLang the semantic functions specify the behavior of a language

*We assume, for the sake of concise presentation, that all variable names are not re-used to side-step complications that arise from variable captures.

fragment. In our example, the semantic function `step` computes a single evaluation step of our lambda calculus and the function `subst` performs substation. The last argument of a semantic function is always implicit and the function performs a pattern match over this implicit argument.

```
lang LambdaCalculus
  syn Term =
  | TmApp {lhs : Term, rhs : Term}
  | TmAbs {ident : String, body : Term}
  | TmVar {ident : String}

  sem step =
  | TmApp t ->
    match t.lhs with TmAbs t2 then
      subst t2.ident t2.body t.rhs
    else if isValue t.lhs then
      TmApp {TmAppType of lhs = t.lhs, rhs = step t.rhs}
    else
      TmApp {TmAppType of lhs = step t.lhs, rhs = t.rhs}
  | TmVar _ -> error "Stuck!"
  | TmAbs _ -> error "Stuck!"

  sem subst ident term =
  | TmVar t -> if eqString ident t.ident then term
               else TmVar t
  | TmApp t ->
    TmApp {TmAppType of lhs = subst ident term t.lhs,
                        rhs = subst ident term t.rhs}
  | TmAbs t ->
    if eqString ident t.ident then TmAbs t else
    TmAbs {TmAbsType of ident = t.ident,
                        body = subst ident term t.body}

  sem isValue = ...
end
```

Listing 1.1: The implemenation of the lambda calculus in the programming language MLang. Some implementational details (e.g. the implementation of `isValue`) are omitted for the sake of brevity.

$$t ::= \lambda x.t \mid t_1\ t_2 \mid x \tag{1.1}$$
$$v ::= \lambda x.t$$

$$\text{E-AbsApp}\frac{}{\lambda x.t_1\ t_2 \to t_2[x \mapsto t_1]} \qquad \text{E-App}\frac{t_1 \to t_1'}{t_1\ t_2 \to t_1'\ t_2} \tag{1.2}$$

$$t_1\ t_2[x \mapsto t] \stackrel{\text{def}}{=} t_1[x \mapsto t]\ t_2[x \mapsto t] \tag{1.3}$$

$$x[y \mapsto t] \stackrel{\text{def}}{=} \begin{cases} t & \text{if } x = y \\ x & \text{if } x \neq y \end{cases}$$

$$\lambda x.t_1[y \mapsto t_2] \stackrel{\text{def}}{=} \begin{cases} \lambda x.t_1 & \text{if } x = y \\ \lambda x.t_1[y \mapsto t_2] & \text{if } x \neq y \end{cases}$$

Figure 1.1: A formal definition of the lambda-calculus with call-by-value, small-step evaluation semantics.

1.2.2 Language Composition

Consider now the language defined formally in Figure 1.2 and its corresponding MLang implementation in Listing 1.2. Intuitively, we should be able to combine our lambda calculus with this arithmetic language to create a more expressive programming language. This operation of combining multiple languages into a new language is called *language composition* and can be subdivided into three operations [3, 4]:

1. *Language Unification* ($L_A \uplus L_B$): Independent language L_A and L_B are unified to create new language.

2. *Language Extension* ($L \triangleleft E$): A new language is created by extending language fragment L with language fragment E. Note that E is typically only meaningful relative to L.

3. *Language Restriction*: A new language is created by removing various things from a language. Language restriction is not a part of this work and will not be discussed further.

At first glance, it seems we can create our new languages through language unification, formally $\mathcal{L}_{\lambda+} \stackrel{\text{def}}{=} \mathcal{L}_+ \uplus \mathcal{L}_\lambda$, and `lang LambdaArithLang = LambdaCalculus + ArithLang end` in MLang. However, since neither $\mathcal{L}_+$ nor $\mathcal{L}_\lambda$ specify substitution over integer constants and addition of terms, the definition of substitution by

$$t ::= n \mid t_1 + t_2$$

$$\text{E-Add1}\frac{n = n_1\ \mathcal{I}(+)\ n_2}{n_1 + n_2 \to n} \qquad \text{E-Add2}\frac{t_1 \to t_1'}{t_1 + t_2 \to t_1' + t_2}$$

$$\text{E-Add3}\frac{t_2 \to t_2'}{n + t_2 \to n + t_2'}$$

Figure 1.2: The definition of a basic arithmetic language, denoted as $\mathcal{L}_+$. In this presentation, we the meta-variable n to range over integer constants and $\mathcal{I}(+)$ to indicate mathematical addition in the integer domain.

structural induction has become ill-defined. As a result, the composition of these languages is considered *unsound*. The lack of soundness manifests itself in the resulting MLang fragment as an inexhaustive match in the `subst` semantic function.

If we want the composition of these two languages to be sound, we need to extend $\mathcal{L}_{\lambda+}$ with the behavior of substitution on integer constants and addition. Formally, we would denote this as $(\mathcal{L}_+ \uplus \mathcal{L}_\lambda) \triangleleft E$, where E denotes the extension of substitution. In MLang, we can express this as seen in Listing 1.2.

```
lang ArithLang
  syn Term =
  | TmInt {val : Int}
  | TmAdd {lhs : Term, rhs : Term}

  sem step =
  | TmAdd t ->
    match (t.lhs, t.rhs) with (TmInt t1, TmInt t2) then
      TmInt {TmIntType of val = addi t1.val t2.val}
    else match t.lhs with TmInt _ then
      TmAdd {TmAddType of lhs = t.lhs, rhs = step t.rhs}
    else
      TmAdd {TmAddType of lhs = step t.lhs, rhs = t.rhs}
  | TmInt _ -> error "Stuck!"
end

lang LambdaCalculusArith = LambdaCalculus + Arith
  sem subst ident term +=
  | TmInt t -> TmInt t
  | TmAdd t ->
    TmAdd {TmAddType of lhs = subst ident term t.lhs,
                        rhs = subst ident term t.rhs}

  sem isValue += ...
end
```

Listing 1.2: The implemenation of a basic arithmetic language MLang and its composition with LambdaCalculus.

If we look at this example from a higher level, we have a syntax type, called `Term`, defined through a sum type. We are then extending this sum type through language composition. Henceforth, we will refer to such an extension as *sum extension*. Our example clearly shows that such language composition through sum extension may be unsound due to the introduction of inexhaustive matches.

1.2.3 Product Extension

We have seen one example where we leverage language composition through sum extension to create a more powerful language. The added behavior comes by adding more constructs to the language syntax and extending the semantics to operate on those constructs. However, sum

extension does not allow us to leverage language composition to add information to *existing* constructs.

Consider the classic syntax of Simply-Typed Lambda Calculus (STLC) in Figure 1.3. Conceptually, STLC is an extension of the untyped lambda calculus defined in the previous section in which we have added a syntax for types, a type annotation to each abstraction term, and defined new dynamic semantics. The crucial difference between the extensions that we have discussed so far lies in the extension of the existing term for abstraction with a type annotation. Instead of extending the term type in the sum dimension by adding more constructors to the type, we extend it in the product dimension by adding more fields to an existing constructor. This type of extension will be called *product extension*.

Language composition under product extension is currently not supported in the Miking framework. The main focus of this thesis is to extend the MLang programming language itself to support language composition through product extension by extending the MLang compiler. In Listing 1.3, on lines 6-7, a proposed syntax for product extension is shown.

A crucial complication that is introduced by product extension, is the fact that the payload of a `TmAbs` constructor is now polymorphic as it may or may not contain a `tyAnnot` field. Certain semantic functions might require this field to be present whilst others do not. Whenever we project this field, we must be sure that it exists; otherwise our program will crash due to this *illegal projection*. For example, the `typeCheck` semantic function requires the `tyAnnot` field to be present but the function `eval` does not use it. Attempting to run `eval` on a `TmAbs` without a type annotation is therefore completely safe, but applying `typeCheck` to that same term is unsound.

$$T ::= T \to T$$

$$\Gamma ::= \epsilon \mid x : T; \Gamma$$

$$t ::= \lambda x : T.t \mid t_1\ t_2 \mid x$$

$$\textsc{T-Var}\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \textsc{T-Abs}\frac{\Gamma, x : T_1 \vdash e : T_2}{\Gamma \vdash \lambda x : T_1.t : T_1 \to T_2}$$

$$\textsc{T-App}\frac{\Gamma \vdash t_1 : T_1 \to T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2}$$

$$\textsc{E-AbsApp}\frac{}{\lambda x : T.t_1\ t_2 \to t_2[x \mapsto t_1]} \qquad \textsc{E-App}\frac{t_1 \to t_1'}{t_1\ t_2 \to t_1'\ t_2}$$

$$t_1\ t_2[x \mapsto t] \stackrel{\mathsf{def}}{=} t_1[x \mapsto t]\ t_2[x \mapsto t]$$

$$x[y \mapsto t] \stackrel{\mathsf{def}}{=} \begin{cases} t & \text{if } x = y \\ x & \text{if } x \neq y \end{cases}$$

$$\lambda x.t_1[y \mapsto t_2] \stackrel{\mathsf{def}}{=} \begin{cases} \lambda x.t_1 & \text{if } x = y \\ \lambda x.t_1[y \mapsto t_2] & \text{if } x \neq y \end{cases}$$

Figure 1.3: The abstract syntax, static semantics and dynamics semantics of STLC

```
1  lang STLC = LambdaCalculusArith
2    syn Ty =
3    | TyArrow {lhs : Ty, rhs : Ty}
4    | TyInt {}
5
6    syn Term *=
7    | TmAbs {tyAnnot : Ty}
8
9    sem eqType = ...
10   sem getFromEnv ident = ...
11
12   sem typeCheck env =
13   | TmVar t -> getFromEnv t.ident env
14   | TmAbs t -> TyArrow {TyArrowType of lhs = t.tyAnnot,
      rhs = typeCheck (cons (t.ident, t.tyAnnot) env) t.
      body}
15   | TmApp t ->
16     match typeCheck env t.lhs with TyArrow inner then
17       match inner with {lhs = lhs, rhs = rhs} in
18       if eqType (lhs, (typeCheck env t.rhs)) then rhs
19       else error "Type Mismatch in Application!"
20     else error "Expcected an Arrow Type!"
21   | TmInt _ -> TyInt {TyIntType of }
22   | TmAdd _ -> TyInt {TyIntType of }
23 end
```

Listing 1.3: The creation of STLC in MLang by extending the untyped lambda calculus through product extension. The new syntax for product extension can be seen on lines 6-7. The environment in type-checking is modeled as an associated list of pairs of strings and types. Certain implementational details are again omitted for the sake of brevity.

1.2.4 Sound Language Composition through Static Typing

We have seen that language composition through sum introduces the possibility of unsound composition due to inexhaustive matches in semantic functions. The Miking framework ensures the soundness of composition through a type system for MLang called Constructor Types (discussed in section 3.2). When we extend the notion of language

composition with product extension, we again introduce the possibility of unsound language composition due to illegal projections. To recover the soundness of composition, this thesis extends the MLang type system to prevent illegal projections.

1.3 Problem Statement

The research problem that is tacked in this thesis is to:

- Extend the notion of language composition in the Miking framework with product extension.
- Ensure the soundness of language composition under product extension through a type system over extensible product types.
- Provide expressive type annotations over syntax types under sum and product extension that remain concise and intuitive.

1.4 Research Method

In this section, we classify the work done in this thesis in terms of scientific research methods in Computer Science. This classification is done based on the work of Dodig-Crnkovic [5] and Eden [6].

The first method used is formal, mathematical reasoning about programs. We do this in Chapter 5, in which we study a type system over extensible record types by defining our language formally. We then state the formal meta-theoretical properties of our language. This approach fits neatly within the rationalist paradigm of Eden.

The rest of this thesis fits neatly into the category of experimental computer science given by Dodig-Crnkovic. We hypothesize that we can facilitate the rapid of Domain-Specific Languages by providing a language workbench based on language composition through product extension. To test this theory, we develop a prototype and perform an empirical evaluation using said prototype.

1.5 Ethics, Sustainability, and Societal Impact

Due to the nature of this thesis, the work has little bearing on sustainability, ethics, and societal impact. However, since the end goal of

this work is to facilitate more rapid development of DSLs across a wide range of domains, this work could play a role in bringing more DSLs into widespread use. The specific impact of this more widespread adoption largely depends on the DSLs that are developed.

1.6 Contributions

The contributions of this thesis are:

- An updated MLang syntax that supports more expressive forms of product extension.
- An extension of the simply typed lambda calculus in which the desired operations on extensible records are formalized.
- An implementation of a type system ensuring the soundness of language composition under product extension based on the aforementioned lambda calculus.
- Improvements to the MLang compiler infrastructure.

1.7 Thesis Outline

The structure of the thesis is as follows:

- Chapter 2 provides a comprehensive background on the research areas relevant to this thesis. Specifically, it goes into the details of the Miking framework and various relevant type systems are discussed.
- In Chapter 3, you will find an overview of the related work. It discusses various type systems in PL literature which solve similar problems to those tackled in this work.
- Chapter 4 discusses design choices and alternatives for an implementation of language composition through sum and product extension. It provides an overview of use-cases that occur naturally during DSL development that we'd like to support. Additionally, it provides a set of criteria that type system should fulfill. It also discusses various possible type systems that could meet these criteria and their trade-offs.

- In Chapter 5, we develop an extension of the lambda calculus that supports nominal, extensible, and polymorphic records.
- Chapter 6 discusses various improvements to the MLang compiler that were made as a part of this work that do not relate directly to product extension.
- The extensions of the MLang that extend the notion of language composition with product extension and a type system supporting polymorphic, extensible records is discussed in Chapter 7. It discusses updates and extensions of the MLang and MExpr syntax, the compilation of MLang constructs to MExpr, the extension of the existing MExpr type checker, and the monomorphisation transformation which converts polymorphic records into monomorphic records in MExpr.
- The evaluation of the aforementioned implementation is discussed in Chapter 8. This evaluation is performed based on the use cases and criteria outlined in Chapter 4.
- Chapter 9 discusses the limitations of the implementation.

Chapter 2

Background

This section provides the background that is necessary to understand this thesis work. We first discuss domain-specific languages, their use cases, and their limitations. Subsequently, we introduce the Miking Framework, the DSL creation workbench which is the subject of the thesis. We proceed to give a comprehensive account of the programming languages of the Miking framework: MCore, MExpr, and MLang. Lastly, we discuss the Simply-Typed Lambda Calculus and various extensions related to record polymorphism.

2.1 Domain-Specific Languages and The Miking Framework

The Miking (Meta-Viking) framework is a language framework for creating Domain-Specific Languages, tailored to the fields of probabilistic programming and mathematical modelling [3]. This section gives a brief overview of Domain-Specific Languages, the applications of the framework, and its internals.

When computer scientists think of programming languages, they typically think of *General Purpose Languages* such as C, Java, or Python. We call these languages General Purpose Languages (or GPLs) because these languages can be used to solve problems across a wide variety of domains, e.g., operating systems, control systems of automobiles, or image rendering can all be written in C.

The counterpart of a GPL is a *Domain-Specific Language* (DSL). A DSL is a language that has been created for a specific application domain

and only those language constructs that are relevant to said domain are included. A major advantage of DSLs is the fact that programs can be created in this domain without requiring users to have the extensive programming knowledge that GPLs require. The major disadvantage of DSLs is that their development is very time-consuming and expensive [1].

A lot of work has been done to facilitate faster development of DSLs which can roughly be divided into two approaches. In the first approach, the DSL is *embedded* into another language called the host language. For example, if you wanted to create a DSL targeting x86 assembly, you could compile your DSL to C and subsequently compile the resulting C into x86. By embedding your DSL in some other language, you save a lot of resources. The second line of work is to create the DSL using a *language workbench*. Such a workbench is a framework that provides many useful tools for creating DSLs such as compilers, debuggers, and parsers.

The Miking framework is such a language workbench. It provides a wide range of functionality that enables users to rapidly develop DSLs. It does this through the concept of *language composition* and *language fragments*. Language fragments might contain a specific language feature such as pattern matching or let-bindings. In the Miking framework, you can combine such language fragments to create DSLs. Instead of having to define the syntax and semantics of pattern matching for each DSL you create which has this language feature, you can just define it once in the fragment and then use this language in all those DSLs [3].

The Miking framework has been used to create DSLs in a variety of domains. In [7], Senderov et al. have created a DSL for statistical phylogenetics in the Miking framework. An intermediate language tailored to Probabilistic Programming, called CorePPL, is an intermediate language built on top of the Miking framework [8]. In a similar domain, a real-time probabilistic programming language called ProbTime was developed in the Miking framework [9].

2.2 MCore, MLang and MExpr

At the heart of the Miking framework lies its own programming language called MCore. This is a typed functional language with OCaml-like syntax. MCore consists of two programming languages: A small expression-based functional language called MExpr and a language built on the concepts of language fragments and language composition

called MLang. MLang can essentially be seen as a superset of MExpr since all expressions in MLang are MExpr expressions. This is shown schematically in Figure 2.1.

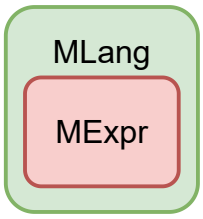

Figure 2.1: A schematic overview of MCore. We see that the our functional expression language MExpr is embedded in MLang.

2.2.1 MExpr

As the name MExpr implies, this language revolves around expressions. In fact, every construction in this language is an expression that yields a value and MExpr therefore falls into the category of expression-oriented programming languages. In contrast to most programming languages, MExpr was not designed to be a general-purpose programming language but is instead meant to be used as a core language into which other languages can be translated. To facilitate this, the language includes little to no syntactic sugar. For example, MExpr lacks infixed arithmetic operators (e.g. `3 + 4`) and replaces them with built-in functions (e.g. `addi 3 4`). We also side-step various issues related to operator overloading and ad-hoc polymorphism [10] as we simply have different built-in functions for different types (e.g. `addi` and `addf` for integer and float addition respectively).

Most of the language features in MExpr are standard for expression-based functional programming languages and will not be discussed in detail. Instead, we refer to Listing 2.1 which gives basic examples of the core functional features. For a more comprehensive overview, we defer the reader to the Miking documentation [11].

```
1 mexpr
2 -- Simple let bindings
3 let x = 10 in
4 -- Lambda abtractions can be used to create functions
5 let id = lam x. x in
6 -- Of course, higher order functions are supported
7 let twice = lam f. lam x. f (f x) in
```

```
-- Integer arithmetic is done through intrinsic
   functions
-- and not through the typical infixed operators
let meaningOfLife = addi 40 2 in
-- Partial application is also supported
let add1 = addi 1 in
let add2 = twice add1 in
-- Basic control flow can be achieved using if-then-else
let abs = lam x.
    if lti x 0
    then muli x 1
    else x
in
-- (Mutual) recursion is also supported
recursive
let odd = lam n.
    if eqi n 1 then true
    else if lti n 1 then false
    else even (subi n 1)
let even = lam n.
    if eqi n 0 then true
    else if lti n 0 then false
    else odd (subi n 1)
in
-- MExpr also supports sequences and product types
let mySequence = [1, 2, 3] in
recursive let sumSeq = lam xs.
    if null xs then 0 else
    addi (head xs) (sumSeq (tail xs))
let myName = "marten" in
let myTuple = ("marten", 1) in
let myRecord = {name = "marten", age = 23} in
-- unit value
()
```

Listing 2.1: A small MExpr program showing off some of its most essential features.

The first noteworthy feature of MExpr is its native support for unit testing. This is done through the `utest`-keyword and an example can be found in Listing 2.2. For many basic data types, the comparison functionality is automatically inferred. One can also specify a comparison

function through the `using`-keyword.

Types in MExpr

MExpr is a statically typed language. Despite this, the programmer is rarely forced to provide type annotations as they will automatically be inferred. The programmer can always choose to provide type annotations, to serve as documentation for instance. The type system of MExpr is polymorphic and based on System F, or more specifically a variant of System F called FreezeML [12, 13]. To aid readability, it is possible to define type aliases. An overview of type expressions in MExpr can be found in Listing 2.2.

```
mexpr
-- Introduction of a type alias
type IntPair = (Int, Int) in

-- Types are inferred automatically
let twice = lam f. lam x. f (f x) in

-- But can also be provided manually be the programmmer
let sumPair : IntPair -> Int = lam p. addi p.0 p.1 in
let id : all a. a -> a = lam x. x in

utest sumPair (3, 4) with 7 in
utest id 10 with 10 in

()
```

Listing 2.2: A small MExpr program showcasing some of the features of the type system.

Open Sum Types

Like most functional languages, MExpr supports sum types (sometimes also referred to as variant types). The specifics of defining such sum types are rather different in MExpr when compared to most other functional languages. Whilst most functional programming languages define such a data type in a single expression or statement, the definition of a sum type in MExpr is split among multiple expressions which need not be consecutive. In MExpr, one first declares the name of the variant

type through the `type` keyword. This definition is *open-ended*: The programmer is free to add constructors to this type at any later point in the program. The addition of such constructors is done through the `con` keyword. The creation of an optional integer type, called `MaybeInt`, in MExpr can be found in Listing 2.3. Note that the `con`-expressions are non-consecutive and we can in fact use the type before all constructors are defined.

```
mexpr
type MaybeInt in
con None : () -> MaybeInt in

-- We can use the type before all constructors
-- are defined!
let isEmpty : MaybeInt -> Bool = lam x.
    match x with None _ then true else false in
utest isEmpty (None ()) with true in

con Some : Int -> MaybeInt in

let partialDivision = lam x. lam y.
    if eqi y 0 then
        None ()
    else
        Some (divi x y)
in

utest partialDivision 10 2 with Some 5 in
utest partialDivision 10 0 with None ()

()
```

Listing 2.3: Defining and using sum types in MExpr.

MExpr variant types can also be parameterized on type variables. This language feature can be used to turn our `MaybeInt` type into an optional data type that we can use with any type, as seen in Listing 2.4. An unusual aspect of sum types in MExpr is that all constructors must take *exactly one* argument. This however, does not restrict the programmer since the absence of any arguments can be represented by the unit type (as done in our `MaybeInt` example) and multiple values can be grouped using tuples or records into a single value.

```
mexpr
type Maybe a in
con None : all a. () -> Maybe a in
con Some : all a. a  -> Maybe a in
()
```

Listing 2.4: An MExpr program which showcases parametrically polymorphic sum types.

2.2.2 MLang

MLang is a superset of MExpr. Whereas a MExpr contains a single expression to be evaluated, an MLang program contains a list of top-level declarations and a single expression that is evaluated relative to the definitions. Most of these top-level declarations have a 1-to-1 correspondence to an MExpr expression, such as let-bindings, unit tests, and type definitions. Each MLang program starts with a number of top-level declarations followed by the `mexpr` keyword after which the expression starts.

Language Fragments and Language Composition

The core language feature that is added in MLang is the *language fragment*. Such a fragment is a combination of syntax types and semantic functions. Multiple fragments can be composed through an operation called *language composition*. The basics of language composition are discussed in subsection 1.2.2.

Pattern Matching in Semantic Functions

Another language feature that sets MLang apart from the bulk of functional programming languages is the ordering and composition of patterns in semantic functions. Most programming languages that support pattern matching, follow a top-to-bottom semantics: When a match-statement is encountered, we first try to match the first pattern, and only if this match fails do we look at subsequent patterns. Such a top-to-bottom pattern match semantics would however not work very well for semantic functions in MLang. This is because typical MLang programs create semantic functions by composing semantic functions from many different languages. If top-to-bottom semantics were used in such cases,

the programmer would need to be extremely careful about the order in which languages are composed as a different order of composition could result in widely different behavior.

Because of this, MLang does not follow the typical top-to-bottom semantics for pattern matching in semantic functions. Instead, it follows a most-to-least specific semantics. This means that the most specific pattern, i.e., the one matching the fewest inputs, is evaluated first, and the least specific patterns, are evaluated last. Consider the semantic function in Listing 2.5. In top-to-bottom semantics, the semantic function `f` would return 0 for any input since the wildcard pattern `_` would match any input. However, in our most-to-least specific semantics, the compiler will re-order the patterns based on specificity. Clearly, the pattern `(0, 0)` is the most specific as it matches exactly one input so it will come first. When we look at the next pattern to be evaluated, things get less clear as the patterns `(0, _)` and `(1, _)` can not be ordered by specificity as they match disjoint sets of values. However, since their matching sets are disjoint, the order does not matter anyway. The wildcard pattern is obviously the least specific and will come last.

```
lang SomeLang
    sem f : (Int, Int) -> Int
    sem f =
    | _ -> 0
    | (0, 0) -> 1
    | (0, _) -> 2
    | (1, _) -> 3
end
```

Listing 2.5: A semantic function which demonstrates the specificity-based semantics of pattern matching.

Our specificity-based semantics allow language composition to be commutative. However, it does mean that semantic functions can only be composed if their patterns can be ordered by specificity or are non-overlapping. Consider the example in Listing 2.6. Since both cases in `L1` and `L2` both match `(0, 0)` but neither is strictly more specific than the other, they can not be ordered by specificity, and as such the composition of languages `L1` and `L2` is invalid. This, and other conditions for language composition validity, are more extensively discussed in subsection 6.2.4.

```
lang L1 =
    sem f =
```

```
    | (0, _) -> -1
end

lang L2
    sem f =
    | (_, 0) -> 1
end

-- This composition will not be allowed by the compiler!
lang L12 = L1 + L2
end
```

Listing 2.6: An example of two semantic funcitons which can not be composed.

MLang also allows the programmer to combine multiple patterns using the operators `|`, `&`, and `!` indicating that either pattern must be matched, both must be matched, or a pattern must not be matched. An example of this can also be found in Listing 2.7.

```
lang L1
    sem f =
    -- Match any tuple whose 1st element is a 0 or 1
    | (0, snd) | (1, snd) -> snd
    -- Match any tuple whose 1st and 2nd element is a 0
    | (0, _) & (_, 0) -> 0
    -- Match any tuple whose 1st element is not 2
    | !(2,_) -> 1
end

mexpr
use L1 in
utest f (0, 10) with 10 in
utest f (0, 0) with 0 in
utest f (4, 3) with 1 in
()
```

Listing 2.7: A semantic function in which patterns are combined using boolean opetors.

2.3 Type Systems and Polymorphism

Type systems have proven themselves to be extremely valuable tools in the programmers toolbox. A good type system can detect a wide range of program bugs without ever running the program. Type systems for programming languages have been extensively studied and this is traditionally done by creating toy languages based upon the Simply-Typed Lambda Calculus (STLC). We will assume that the reader is familiar with the basics of STLC (if not, we refer to the canonical textbook by Pierce [14]). In this section, we will discuss various extensions of the simply-typed lambda calculus:

- Record and variant primitives.
- Parametric polymorphism through System F.
- Record subtyping and its limitations.
- Various forms of row-polymorphism.
- Recursive types in the form of μ-types.

2.3.1 The Lambda Calculus and Its Extensions

The purest version of lambda calculus only contains syntax terms for variables, abstraction, and application. Despite this calculus being extremely expressive, it is not very practical to reason about or write programs in. To this end, we first extend our calculus with basic integer arithmetic by adding integer constants and addition. The extended calculus can be found in Figure 2.2.

Records

Another useful addition to our calculus is the record. Records are a fundamental datatype that allows the programmer to map labels to values. For instance, the record $\texttt{r} = \langle \texttt{x} = \texttt{10}, \texttt{y} = \lambda \texttt{x}.\texttt{x} \rangle$ maps the label `x` to the integer constant `10` and the label `y` to the identity function. Either of these fields can be extracted from the record through a so-called projection operation (e.g. `r.y`). Records have proven to be an invaluable construct and are therefore supported in one form or another by most, if not all, serious programming languages including MCore. Despite records

being a very simple datatype, a lot of interesting, non-trivial issues arise when we create a type system over records. In this section, we will review some of the problems that arise when combining types and records and discuss various type systems that attempt to remedy these problems.

Variants

Variant types, or sometimes called sum types, are closely related to the notion of records. Variant types arise naturally in programming when describing values that belong to a collection but are not the same shape. Canonical examples include a head-tail list in which a value is either the empty list or a pair consisting of a value and another list; a tree in which values are either a leaf or a node; or a user interaction event which is either a mouse click at a certain coordinate or a keypress of a certain character.

Progress, Preservation, and Type Safety

The primary benefit of static type systems for programming languages is the detection of errors at compile-time. Concretely this means that the type system rejects programs that potentially behave in erroneous ways. In formal language descriptions in programming language theory, such erroneous behavior is typically characterized as getting *stuck* during evaluation—this means that an expression was reached that can not be evaluated further but which we do not consider an end result (traditionally called a value). In formal type theory, we typically prove that a well-typed program will also be well-behaved in the sense that it does not get stuck. Type theorists do this by proving two theorems about their languages called *progress* and *preservation*. The progress theorem states that any well-typed expression is either a value or it can be evaluated to another expression. The preservation theorem states that whenever a well-typed expression is evaluated, the resulting expression will also be well-typed. When we have proven that the progress and preservation theorems hold in a certain language, we say that we have achieved *type safety* [14, Ch. 8].

Type Inference

In our discussion so far we have been talking about *type checking*. In the type-checking problem, the user provides type annotations for

some of the expressions and our goal is to determine whether those types are in accordance with our type rules. In large programs, or in programming languages with complicated type systems, the process of providing explicit types can place a large burden on the programmer. It is therefore very common for programming languages to support *type inference* [14, Ch. 22]. In the type inference problem, sometimes also called the type-reconstruction problem, the end-user provides an expression *without* explicit type annotations and it is up to the type system to find the type of that expression. Oftentimes, an expression can be assigned many types. In the type inference problem, we are typically concerned with finding the *principal type*: The most general type that can be assigned to that expression. The canonical type system supporting is called the Hindley-Milner type system. The type inference problem is typically solved through algorithm W which centers around unification [15, 16].

Kinding Systems

A natural extension of STLC is to introduce type-level expressions such as `Array Int` or `Ref Char`. At this point we are essentially binding types to other types This is typically done through so-called *type operators*. The extension of the lambda calculus with type operators is called λ_ω. In STLC, when we bind terms to other terms, we introduce the potential for ill-formed term-level expression and introduce a type system over terms to address this. Similarly, when we add type operators, we introduce the potential for ill-formed type-level expressions. To address this, we introduce what can essentially be seen as a type system over types, called a *kinding system*. We will see in further sections that whenever we introduce the potential of ill-formed type expressions, we can remedy this by introducing new kinding rules to ensure that our type expressions are well-formed.

2.3.2 Parametric Polymorphism and System F

A major limitation of the basic version of STLC we have defined so far is the restriction that each function has a single, explicitly defined type. We call terms that have a single type *monomorphic*. This severely restricts the re-usability of functions in our calculus. Consider the identity function. Since we must explicitly provide a type annotation for each abstraction, we must create a new identity function for each

Syntax

$t ::=$		terms:
	x	variable
	n	integer
	$t + t$	addition
	$\lambda x : T.t$	abstraction
	$t\ t$	application
	$\left\langle l_i = t_i^{i \in 1 \ldots n} \right\rangle$	record
	$t.l$	record projection
$v ::=$		values:
	$\lambda x : T.t$	abstraction value
	n	integer value
	$\left\langle l_i = t_i^{i \in 1 \ldots n} \right\rangle$	record value
$T ::=$		types:
	Int	integer type
	$T \to T$	function type
	$\left\langle l_i : T_i^{i \in 1 \ldots n} \right\rangle$	record type
$\Gamma ::=$		contexts:
		empty context
	$\Gamma,\ x : T$	term variable binding

Typing

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1.t_2 : T_1 \to T_2}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}}$$

$$\frac{}{\Gamma \vdash n : \mathsf{Int}}$$

$$\frac{\Gamma \vdash t_1 : \mathsf{Int} \quad \Gamma \vdash t_2 : \mathsf{Int}}{\Gamma \vdash t_1 + t_2 : \mathsf{Int}}$$

$$\frac{\text{forall } i \quad \Gamma \vdash t_i : T_i}{\Gamma \vdash \left\langle l_i = t_i^{i \in 1 \ldots n} \right\rangle : \left\langle l_i : T_i^{i \in 1 \ldots n} \right\rangle}$$

$$\frac{\Gamma \vdash t : \left\langle l_i : T_i^{i \in 1 \ldots n} \right\rangle}{\Gamma \vdash t.l_j : T_j}$$

Figure 2.2: A variant of the Simply Typed Lambda Calculus with explicit abstraction parameter type that has been extended with integers and records.

type we wish to use it with, even though the bodies of these functions is exactly the same.

$$\texttt{idInt} \stackrel{\text{def}}{=} \lambda \texttt{x} : \texttt{Int}.\texttt{x}$$
$$\texttt{idBool} \stackrel{\text{def}}{=} \lambda \texttt{x} : \texttt{Bool}.\texttt{x}$$

Another canonical example of this phenomenon is when we attempt to define a function `twice` that takes a function and an argument and applies the function to the argument twice. Here we also see that the need to explicitly type the argument of the function forces us to define identical functions for different types:

$$\texttt{twiceInt} \stackrel{\text{def}}{=} \lambda \texttt{f} : (\texttt{Int} \to \texttt{Int}).\lambda \texttt{x} : \texttt{Int}.\ \texttt{f}\ (\texttt{f}\ \texttt{x})$$
$$\texttt{twiceBool} \stackrel{\text{def}}{=} \lambda \texttt{f} : (\texttt{Bool} \to \texttt{Bool}).\lambda \texttt{x} : \texttt{Bool}.\ \texttt{f}\ (\texttt{f}\ \texttt{x})$$

To alleviate such problems, we would like to add a notion of *polymorphism* to our calculus. Essentially, we would like to have a single definition of our function `twice` that can be used with values of the type `Int`, `Bool`, and any others. Whilst polymorphism comes in many flavors, we will solve this problem through *parametric polymorphism*. At its essence, parametric polymorphism allows the type of a term to be specified not only through concrete types but also using *type variables*; these type variables can then later be instantiated to concrete types.

Parametric polymorphism is typically added to the lambda calculus in the form of *System F* which was devised by Girard in 1972 [12, 14]. The idea behind System F is simple but elegant. We reuse the abstraction and application in the lambda calculus but define these operations on types instead of terms by introducing the following terms into our syntax: $\lambda\texttt{X}.\texttt{t}$ to abstract over a type variable `X` and introduce the syntax `t` [`T`] to apply a concrete type to a type expression. Additionally, we introduce universal quantification over types into our type syntax: $\forall\texttt{X}.\texttt{t}$. We can easily and

elegantly express our `id` and `twice` functions in System F as follows:

$$
\begin{aligned}
\texttt{id} &\stackrel{\text{def}}{=} \lambda \texttt{X}.\lambda \texttt{x} : \texttt{X}.\texttt{x} \\
\texttt{id}&: \forall \texttt{X}.(\texttt{X} \to \texttt{X}) \\
\texttt{twice} &\stackrel{\text{def}}{=} \lambda \texttt{X}.\lambda \texttt{f} : (\texttt{X} \to \texttt{X}).\lambda \texttt{x} : \texttt{X}.\texttt{f}(\texttt{f}\texttt{x}) \\
\texttt{twice}&: \forall \texttt{X}.(\texttt{X} \to \texttt{X}) \to \texttt{X} \to \texttt{X} \\
\texttt{twice}&\ [\texttt{Int}]\ (\lambda \texttt{x}.\texttt{x} + 1)\ 0 \Rightarrow 2
\end{aligned}
$$

2.3.3 Nominal and Structural Typing

Broadly speaking, type systems can be split into two categories: *nominal* and *structural* [14, Ch. 19.3].

In a nominal type system, the compatibility or equivalence of two types is determined through explicit names, usually defined by the programmer. Programming languages in the Object-Oriented programming paradigm feature strongly nominal type systems: If the programmer defines two classes that have an identical structure (e.g. the same methods, fields, access modifiers, etc.), the programmer is not able to use one class where the other is expected, because they have different names unless they establish this relation through the use of interfaces or class extension. Similarly, if you define two identical structs in C with different names, the compiler will not allow you to mix and match these and they will remain incompatible.

As the name implies, in a structural type setting, type equivalence or compatibility is fully determined by looking at the structure of a type. If two types have an identical structure, they are compatible. All of the lambda calculus variants discussed in this section belong to the class of structural type systems.

2.3.4 The Limitations of Monomorphic Records

Consider again the records in the type system outlined in Figure 2.2. The type rules prevent the programmer from accessing labels on records that do not exist, e.g. if you attempt to project label `z` on the record $\langle \texttt{x} = 1, \texttt{y} = 2 \rangle$ this expression will be ill-typed. This is of course a very desirable property to have in our type system because it eliminates an entire category of bugs. However, this type system can also be needlessly restrictive in the sense that perfectly

well-behaved programs are considered ill-typed. This occurs when attempting to reuse user-defined functions. Consider the abstraction $\mathsf{incrX} \stackrel{\mathsf{def}}{=} \lambda \mathtt{r} : \langle \mathtt{x} : \mathsf{Int} \rangle \,.\, \mathtt{r.x} + 1$. If we apply the function to the record $\langle x = 20 \rangle$, our program is well-typed and well-behaved. However, when we attempt to apply `incrX` to $\langle \mathtt{x} = 10, \mathtt{y} = 20 \rangle$, our program becomes ill-typed because of the field y even though the program is well-behaved; after all, only the field x will be projected. Intuitively, programs should always stay well-behaved if we treat a record as if had fewer fields than it actually has.

If we want to have a function such as `incrX`, we have to define it for each shape of record we wish to use. The limitation that we are running here is that our function is *monomorphic*: The function only accepts parameters of one specific type. This problem is similar to the problem we encountered when defining `incr` and `twice` in the previous section. However, System F (on its own) is not expressive enough to express the kind of polymorphism we would like when working on records. For instance, we could not assign the type $\mathtt{incrX} : \forall \mathtt{X.x} \to \mathtt{Int}$ because this function is not well-behaved for any type, but only for record types in which `x` is mapped to a value of type `Int`. To express the polymorphic type of functions such as `incrX`, we need a specific notion of record polymorphism.

Type systems containing record polymorphism have been extensively studied since the 70s. In fact, it seems like every aspiring type theorist comes up with their own take on the subject. Roughly speaking, there are two approaches when it comes to record polymorphism: The first option is to define a *subtyping relation* over records and the second approach is to parameterize the definitions of our functions over so-called *rows*.

2.3.5 Record Subtyping and Bounded Quantification

One of the earliest attempts at record polymorphism, dating back to 1985, has been through subtyping [17, 14]. Recall our intuition that it is always safe to treat a record as if it had fewer rules. Subtyping is at its core a formalization of this intuition. When a record type $\mathtt{r}_1$ is a subtype of $\mathtt{r}_2$, denoted $\mathtt{r}_1 <: \mathtt{r}_2$, this means that it is safe to use $\mathtt{r}_1$ in place of $\mathtt{r}_2$. This subtyping relation comes in two flavors: *nominal* and *structural* subtyping.

In a nominal subtyping relation, the subtyping relation is explicitly declared by the programmer. This is especially common in

object-oriented languages; for instance when a programmer defines `class Dog extends Animal {...}`, they explicitly declare that `Dog` is a subtype of `Animal` and it is thus safe to provide a dog to any method that expects an animal.

In a structural subtyping system, the user does not explicitly define the relation, but it is inferred by the type system based on the structure of the type. In other words, there are explicit rules which define the subtyping relation. When defining a subtyping relation for records, we need three rules (Equation S-RcdWidth, Equation S-RcdDepth, and Equation S-RcdPerm) which state that we can safely ignore fields of a record, we can apply the subtyping relation to the fields of a record itself, and we can re-order records in any way we see fit. Now that our subtyping relation is formally defined, we can easily add subtyping to our type system by adding the rule Equation T-Sub. This rule states that any term of type `T` can be considered to also be of type `S` whenever `S` $<:$ `T`.

$$\frac{-}{\langle \mathtt{l_i} : \mathtt{T_i} \rangle^{\mathtt{i} \in 1 \ldots \mathtt{n}+\mathtt{k}} <: \langle \mathtt{l_i} : \mathtt{T_i} \rangle^{\mathtt{i} \in 1 \ldots \mathtt{n}}} \quad \text{(S-RcdWidth)}$$

$$\frac{\forall \mathtt{i}. \mathtt{S_i} <: \mathtt{T_i}}{\langle \mathtt{l_i} : \mathtt{T_i} \rangle^{\mathtt{i} \in 1 \ldots \mathtt{n}} <: \langle \mathtt{s_i} : \mathtt{S_i} \rangle^{\mathtt{i} \in 1 \ldots \mathtt{n}}} \quad \text{(S-RcdDepth)}$$

$$\frac{\langle \mathtt{l_i} : \mathtt{T_i} \rangle^{\mathtt{i} \in 1 \ldots \mathtt{n}} \text{ permutation of } \langle \mathtt{k_i} : \mathtt{K_i} \rangle^{\mathtt{i} \in 1 \ldots \mathtt{n}}}{\langle \mathtt{l_i} : \mathtt{T_i} \rangle^{\mathtt{i} \in 1 \ldots \mathtt{n}} <: \langle \mathtt{k_i} : \mathtt{K_i} \rangle^{\mathtt{i} \in 1 \ldots \mathtt{n}}} \quad \text{(S-RcdPerm)}$$

$$\frac{\Gamma \vdash t : S \quad S <: T}{\Gamma \vdash t : T} \quad \text{(T-Sub)}$$

Now let's go back to our example function incrX from subsection 2.3.4. Since $\langle \mathtt{x} : \mathtt{Int}, \mathtt{y} : \mathtt{Int} \rangle <: \langle \mathtt{x} : \mathtt{Int} \rangle$ by a trivial application of Equation S-RcdWidth, the expression $\mathtt{incrX}\, \langle \mathtt{x} : \mathtt{Int}, \mathtt{y} : \mathtt{Int} \rangle$ is now well-typed.

Whilst the addition of subtyping to our lambda calculus is a big improvement in the expressiveness of our type system, it unfortunately does have some limitations, mainly when it comes to updating records. Consider the following lambda abstraction `g` which computes a new record consisting of `x` and the passed record stored in `r`.

$$\mathtt{g} \stackrel{\text{def}}{=} \lambda \mathtt{r} : \langle \mathtt{x} : \mathtt{Int} \rangle \,.\, \langle \mathtt{newX} = \mathtt{x} + 1, \mathtt{r} = \mathtt{r} \rangle$$
$$\mathtt{g} : \langle \mathtt{x} : \mathtt{Int} \rangle \to \langle \mathtt{newX} : \mathtt{Int}, \mathtt{r} : \langle \mathtt{x} : \mathtt{Int} \rangle \rangle$$

Due to our subtyping rules, we can apply the record $\langle \mathtt{x} = 1, \mathtt{y} = 2 \rangle$ to `g`. However, our type system does not allow us to access the field

y on the r-projection of the result even though this is perfectly well-defined behavior. When we apply the subtyping relation to the record $\langle \mathtt{x} = 1, \mathtt{y} = 2 \rangle$, we lose information about the field y.

To provide more expressive types for such functions, we can combine our subtyping with System F. In this new type system, called *bounded quantification*, we provide an upper bound to the universally quantified type variables, using the syntax $\forall \mathtt{X} <: \mathtt{t}_1.\mathtt{t}_2$. Using bounded quantification, we can define and type the function g as follows:

```
1    g = λX <: <x:Int>. λr : X. <x = r.x+11, r = r>
2    g : ∀X <: <x:Int>. <x:Int, r:X>
```

Bounded quantification using a nominal subtyping relation is commonly used in object-oriented programming languages such as Java to provide expressive generics. For instance, when we create a binary search tree (BST) class in Java, that class should work with any class that implements the `Comparable` interface. To express this, we would give the class the signature seen below which essentially corresponds to the type $\forall T <: Comparable.e$.

```
public class BST<T extends Comparable<T>> {...}
```

When it comes to using bounded quantification in combination with a structural subtyping relation, things become more complicated. It actually turns out that the subtyping relation becomes undecidable under bounded quantification [18]. If we want to retrieve the decidability of the subtyping relation, we have to limit the expressiveness of our type system. Due to these issues, bounded quantification is not often used in combination with structural subtyping.

2.3.6 Row Polymorphism

Similar to record subtyping, row polymorphism allows us to define polymorphic functions over records. However, where subtyping revolves around some structural or nominal subtyping relation, row polymorphism is a combination of parametric polymorphism and structural typing. Recall that in parametrically polymorphic type systems like System F, we are able to abstract over type variables. In a row-polymorphic type system are able to abstract over *row variables*. In most modern row-polymorphic type systems, a row variable, typically denoted as ρ, is a (potentially empty) mapping from labels to types; typically defined as

in Equation 2.1. Our running example `incrX` can be defined, typed, and used in a row polymorphic type system as follows:

$$\rho ::= \cdot \mid \ell : T; \rho \tag{2.1}$$

$$\begin{aligned}
&\texttt{incrX} \stackrel{\text{def}}{=} \lambda\rho.\lambda\texttt{r}.\,\langle \texttt{x} : \texttt{Int}; \rho\rangle\,.\texttt{r.x} + 1\\
&\texttt{incrX}\colon \forall\rho.(\langle \texttt{x} : \texttt{Int}; \rho\rangle) \to \texttt{Int}\\
&\texttt{incrX}\ [\langle \texttt{y} : \texttt{Int}\rangle]\ \ \langle \texttt{x} = 1, \texttt{y} = 2\rangle\, 2
\end{aligned}$$

A major advantage of this type system over bounded quantification is that we are able to elegantly handle record updates without introducing any additional complexity. For instance, our running example g can easily be defined and typed as follows:

$$\begin{aligned}
&\texttt{g} \stackrel{\text{def}}{=} \lambda\rho.\lambda\texttt{r}\,\langle \texttt{x} : \texttt{Int}; \rho\rangle\,.\,\langle \texttt{x} = \texttt{r.x} + 1, \texttt{r} = \texttt{r}\rangle\\
&\texttt{g} : \forall\rho.(\langle \texttt{x} : \texttt{Int}; \rho\rangle) \to (\langle \texttt{x} : \texttt{Int}, \texttt{y} : \langle \texttt{x} : \texttt{Int}; \rho\rangle\rangle)
\end{aligned}$$

Row types were initially proposed by Wand in 1987 as a way to model inheritance in object-oriented languages [19]. His original rows formed mappings from labels to types. In 1989, Rémy gave an alternative account of row types in which a row maps labels to presence indicators [20]. Since then, many versions of row-polymorphic type systems have been developed. Broadly speaking, these type systems differ in their approaches to *row extension* (i.e. adding fields to a single row) and *row concatenation* (i.e. combing multiple rows). In this thesis, we will focus on row extension and omit row concatenation.

The crucial complication of row extension in row-polymorphic type systems arises when we attempt to extend a row with a label that is already contained. Consider a function `extendR` that extends a record with a field `x` as seen in Equation 2.2. In most type systems, it is illegal to extend a record with a label that it already contains. In other words, a well-formed row does not contain duplicate labels. A first approach is to keep track of the labels that are present or absent in a given record using kinds [20, 21, 22]. An alternative approach, based on qualified types, is to introduce a "lacks" predicate which ensures that a certain label may not occur [23, 24]. A different view was taken by Leijen, where it is allowed for a label to occur multiple times within the same row through the introduction of scoping over labels in rows [25, 26].

$$\begin{aligned}&\texttt{extendR} \stackrel{\text{def}}{=} \lambda\rho.\lambda\texttt{r} : \langle\rho\rangle\,.\,\langle\texttt{extend r with x} = 1\rangle \\ &\texttt{extendR} : \forall\rho.\,\langle\rho\rangle \to (\langle\texttt{x} : \texttt{Int}, \rho\rangle)\end{aligned} \tag{2.2}$$

2.3.7 Recursive Types

Recursive types allow us to assign types to potentially infinitely large expressions with a regular structure. They are highly relevant in the context of programming languages, as the abstract syntax trees are naturally recursive. The canonical example is the functional nil-cons list. Such a list is either empty, denoted by a distinguished value `nil`, or it is constructed, abbreviated to `cons`, from a value combined with another list. We can express the list of primes smaller than 8 as:

$$\texttt{Cons}(2, \texttt{Cons}(3, \texttt{Cons}(5, \texttt{Cons}(7, \texttt{nil}))))$$

We can easily assign a type, denoted by `T` to a concrete list of finite length, but it is non-trivial to come up with a type that can express a list of arbitrary length. This is because the type recursively occurs within itself:

$$\texttt{T} = [\texttt{nil} : \texttt{Unit} + \texttt{Cons} : (\texttt{Int}, \texttt{T})]$$

Of course, such a definition is not well-founded. Instead, we can treat it as an equation that our list type must satisfy. We can formalize this insight by introducing an explicit recursive operator μ. We can define the type as seen in Equation 2.3—Intuitively, this definition says that `T` is defined to be the infinitely-sized type which satisfies the equation $\mu\texttt{X}.\,[\texttt{nil} : \texttt{Unit} + \texttt{Cons}(\texttt{Int}, \texttt{X})]$.

$$\texttt{T} \stackrel{\text{def}}{=} \mu\texttt{X}.\,[\texttt{nil} : \texttt{Unit} + \texttt{Cons}(\texttt{Int}, \texttt{X})] \tag{2.3}$$

Iso-Recursive and Equi-Recursive Types

Before we considered recursive types, two types were simply equivalent if they were syntactically identical. This equivalence becomes more tricky when we introduce recursive types. Consider the types `T` and $\texttt{T}'$. It is

clear that $\mathtt{T}'$ is obtained by unrolling $\mathtt{T}$ once.

$$\mathtt{T} \stackrel{\text{def}}{=} \mu\mathtt{X}.\,[\mathtt{nil} : \mathtt{Unit} + \mathtt{Cons}(\mathtt{Int}, \mathtt{X})]$$
$$\mathtt{T}' \stackrel{\text{def}}{=} \mu\mathtt{X}.\,[\mathtt{nil} : \mathtt{Unit} + \mathtt{Cons}(\mathtt{Int}, [\mathtt{nil} : \mathtt{Unit} + \mathtt{Cons}(\mathtt{Int}, \mathtt{X})])]$$

In the literature, there are two distinct approaches to define the relation between types and their unfoldings. In the first approach, called the *equi-recursive* approach we say that recursive types are *defintionally equivalent* to their unfoldings. In the other approach, such types are not equivalent but isomorphic and this approach is therefore called the *iso-recursive* approach. In this approach, the programming language gets two new terms `fold` and `unfold` which maps recursive types to their foldings and unfoldings [14, Ch. 20].

Whilst the equi-recursive approach is arguably more intuitive, it also places stronger demands on the type checker, especially when we combine recursive types with other language features. A language with iso-recursive types but easier to type-check however there is some overhead due to the insertion of `fold` and `unfold` terms. Fortunately, it is often possible in practice to insert such terms automatically.

Mutual Recursion

So far, we have only considered types which are recursively defined in terms of themselves. We can of course also have *mutually recursive* types; i.e. two or more types that are defined in terms of each other.

The canonical example of a mutually recursive datatype is a tree. Each tree stores a value (an integer in our case) and a list of children. We do not have a primitive list in our language, but we know how to use recursive types to create such a list. We will call a list of trees a forest. Since our recursion talks about both types, we must inline the definition of `forest` into the definition of `tree`:

$$\mathtt{tree} \stackrel{\text{def}}{=} \mu\mathtt{t}.(\mathtt{Int}, \mu\mathtt{f}.([\mathtt{Unit} + (\mathtt{t}, \mathtt{f})])$$
$$\mathtt{forest} \stackrel{\text{def}}{=} \mu\mathtt{f}.([\mathtt{Unit} + (\mathtt{t}, \mathtt{f})]$$

Whilst such solutions do allow us to μ-types to express mutual recursion, the size of our types grows quadratically as the amount of mutually recursive types increases. A more scalable and principled approach to solve this problem is to introduce higher kinds of types into our language, specifically, we would like tuples on the type level and kind variables

(typically denoted as Ω). With such tuples over types, we can more elegantly define our `tree` type as below [27].

$$\begin{aligned} \texttt{pair} &\stackrel{\text{def}}{=} \mu \mathsf{a} : (\Omega, \Omega).\ ((\texttt{Int}, \mathsf{a}.1), [\texttt{Nil} + (\mathsf{a}.1, \mathsf{a}.2)] \\ \texttt{tree} &\stackrel{\text{def}}{=} \texttt{pair}.1 \\ \texttt{forest} &\stackrel{\text{def}}{=} \texttt{pair}.2 \end{aligned}$$

Chapter 3

Related Work

This chapter gives an overview of work that is closely related to this thesis. We first talk about Wadler's expression problem and various solutions to it across different programming paradigms. After this, we give an overview of type systems in PL literature that solve similar problems to those we tackle in this thesis.

3.1 Wadler's Expression Problem

In 1998, in a discussion about adding generics to Java, Wadler coined the term *the expression problem* [28]. It describes the problem in which we would like to create a programming language that allows the description of datatype through cases. It must also be possible to add in new cases and new functions over this datatype in such a way we maintain static type safety and without recompilation. We can state the requirements for a solution to the expression problem in more detail as follows [29]:

1. Extensibility in two directions. We must be able to easily add more cases and we must easily be able to add more operations.

2. Static type safety. We should be able to statically check that we will never apply an operation to a variant for which it is ill-defined.

3. Compilation and type-checking must be done independently on individual components, not on the result of composition.

4. No code duplication or modification.

3.1.1 Solving the expression problem in the Miking framework

The Miking framework addresses the expression problem head-on through the language features of syntax definitions, semantic functions, and language compositions. At its very core, a syntax in MLang is a datatype that can be extended through new cases through language composition and semantics functions over these syntax definitions which can also be extended through language composition. In MLang, we do consider a relaxed version of the problem in which we are willing to forego the third condition of independent type-checking and compilation.

Furthermore, the problem we are solving in this thesis is in fact an extended version of Wadler's expression problem. Not only should we be able to add new cases to our datatype, but we would also like to extend existing cases through product extension. Additionally, the fact that we are able to express the kind of data type extension that the expression problem requires is only the first piece of the puzzle. The second, and arguably more difficult, piece is the requirement that the language mechanisms that provide this extensibility must also be type-safe. With this in mind, we can view the work of this thesis as a solution to an extended version of Wadler's expression problem in a typed, functional language tailored specifically to DSL development.

3.1.2 Existing Solutions to the Expression Problem

Since the expression problem was stated by Wadler, various solutions across all of the programming paradigms have been suggested. In the object-oriented paradigm, Wadler himself suggested a solution using generics in Java. Another solution in Java was given by Torgesen which was based on a combination of subtyping and generics in Java [30]. Furthermore, Wang and Oliveira proposed an elegant solution in Scala based on "covariant type refinement of type refinement in types of positive positions"[29]. They also show how their solution can be implemented more indirectly in Java.

In the functional paradigm, many solutions have also been proposed. The problem is solved in MLPolyR through extensible row polymorphic sum types in combination with composable case expressions [31]. This approach is extensively discussed in section 3.4. Another solution, exemplified by a Haskell implementation is given by Swierstra [32].

3.2 Constructor Types in MExpr

Constructor types are a type system that was developed for the Miking framework which allows for the local extension of sum types and composition of semantic functions*. The main goal during the development of constructor types for MExpr was to eliminate inexhaustive matches in semantic functions. Constructor types achieve this by parameterizing syntax types on a type variable of a specific kind. This kind then describes a lower and upper bound on the constructor that must be present on this type. An example of this can be found in Listing 3.1. Since the type system discussed in Chapter 7 is built on top of constructor types, we defer to this section for details on the inner workings.

To secure the fundamentals of this type system, the lambda calculus extension λ_C was developed which supports local extension of sum types and semantic function composition. This lambda calculus was mechanized in Coq [33] and the standard meta-theoretic properties of progress and preservation were shown to hold. They also showed that the type system eliminates the problem of inexhaustive matching by proving that a term being well-typed implied that any contained match expressions were exhaustive.

```
1  lang SomeLang
2      syn Term =
3      | Foo ()
4      | Bar ()
5      | Baz ()
6
7      sem f : all m :: {Term [< Foo Bar]}.
8              Term{m} -> Int
9      sem f =
10     | (Foo _ | Bar _) -> 0
11 end
```

*No citation is provided as the work is unpublished at the time of writing. The work in this section has been done by Anders Thuné and the Miking development team. It can be found at https://github.com/miking-lang/miking.

Listing 3.1: A small program showing a semantic function annotated with a constructor type that indicates that this function only accepts the constructors `Foo` and `Bar`, but not `Baz`. Attempting to provide `Baz` to `f` will result in a type-error at compile-time instead of an inexhaustive match at run-time.

3.3 Row-Polymorphic Records in MExpr

As part of a research project into compilation strategies for record-polymorphic type systems, Palmkvist has added row-polymorphism to MExpr*. This implementation is based on the work of Leijen [25] and distinguishes itself from other row polymorphic type systems by allowing duplicate labels through the introduction of label scoping.

```
1 addFoo : forall r a. a -> { | r} -> {foo :: a | r}
2 addFoo a r = r + {foo = a}
```

Listing 3.2: A simple polymorphic function that takes any record and adds a field 'foo' to it.

Consider the function `addFoo` in Listing 3.2 which adds a field `foo` to any record. Unlike most type systems, the term `addFoo "abc" {foo = 10}` is actually well-typed and will evaluate to `{foo = "abc", foo = 10}` of type `{foo : String, foo : Int}`. When we project the field `foo` on this, the most recently added value will be projected. If we want to access the original `foo` of type `int`, we can do so through record constriction as follows: `{foo = "abc", foo = 10} - {foo}` which evaluates to `{foo = 10}`.

This type system is purely structural and the programmer is therefore allowed to substitute in any row whenever we quantify over rows. The polymorphic records we get through product extension behave differently. The programmer defines which fields are present in the record that is attached to a certain constructor and it should be illegal to extend such records with any other fields. Furthermore, duplicate labels are not desirable when it comes to product extension.

*No citation is provided because this work is unpublished. The source code of this implementation can be found at https://github.com/elegios/miking/tree/record-stuff.

Another reason why the work by Palmkvist differs from this work is in the compilation strategy. Due to the fact that records can take arbitrary shapes, compilation can not be done through monomorphization. Instead, a compilation strategy where record polymorphic functions are automatically augmented with all information needed for compilation based on [34] is used. Under product extension, all fields that can possibly be present on a record are defined by the programmer. Because of this, polymorphic records can easily and efficiently be monomorphized.

3.4 Extensible Programming in MLPolyR

The language MLPolyR, created by Blume et al., is a functional programming language with support for extensible records, extensible sums, and case expressions over said sum types which can be extended through a composition mechanism [31]. Furthermore, it has an expressive type system with row-polymorphic record and sum types, support for equi-recursive types, and complete type inference. Due to these extensible sum types, extensible case expressions, and its expressive type system, MLPolyR is presented as an elegant, functional solution to Wadler's expression problem.

The compilation of MLPolyR occurs in two steps. This first step heavily relies on the duality between sums and products. In this step, MLPolyR is rewritten to System F with support for records by rewriting all sums and cases to records. This system is then subsequently compiled into a small untyped lambda calculus with support for named functions and records.

Interestingly, the language features of record extension, sum extension, and case extension provide the same functionality as we get through syntax types, semantic functions, and language composition in MLang. Furthermore, we have seen that the type system of MLPolyR provides a high degree of expressivity whilst still having complete type inference. This is a strong indication that a type system similar to that of MLPolyR based on row polymorphisms over sums and products with equi-recursive types, could provide the expressivity we are looking for in MLang.

There are however some aspects of MLPolyR that make a 1-to-1 translation of its type system to MLang unsuitable. The largest one is that types in MLPolyR are purely structural whilst the MLang type

system has nominal aspects. Furthermore, type annotations in MLPolyR can be very verbose to provide this high degree of expressiveness. If we were to use a type system like it in MCore for realistic DSL development, the readability and writeability of type annotations would suffer.

3.5 Abstracting Extensible Data Types

As we've seen in subsection 2.3.6, various type systems built on row types and row polymorphism have been developed in recent decades. Whilst these row-based type systems share many aspects, they often have different conditions for row extension (i.e., the addition of new entries to a row). Furthermore, row concatenation, the combination of multiple rows into a single new row, is often unsupported. Morris and McKinna introduce the notion of *row theories* which is an abstract characterization of existing row systems [35]. Furthermore, they design a new typed functional programming language called Rose on top of their new notion of row theories. In this way, their language can be instantiated with various existing row theories.

The work of Morris and McKinna convincingly shows that various existing row-based typing systems can be seen as row theories and therefore implemented in Rose. All of these type systems are however structural in their nature. Such structural type systems will typically produce type annotations that are too large to be of practical use in the Miking framework.

3.6 Mutually Iso-Recursive Subtyping

For the work on WebAssembly [36], a type system supporting mutual recursion and subtyping of algebraic data-types was needed. Due to specific WebAssembly design decisions, the type checking operation needs to be performed very efficiently. Rossberg found that the canonical approach of providing mutually recursive types through higher level kinds, discussed in subsection 2.3.7, can not be combined with subtyping [27]. To address this shortcoming, they developed a lambda calculus fragment supporting iso-recursive types with declared subtyping and mutual recursion called λ_{misu} [27] and show that the desired meta-theoretical properties such as progress and preservation hold in their calculus.

Although the work of Rossberg takes place in a different context, the type-checking of mutually recursive types under subtyping bears a resemblance to the desired MLang type system supporting product extension. The types defined through `syn`-declarations in MLang are often naturally mutually recursive. Furthermore, we can view the product extension of a `syn`-type as the creation of a supertype. Consider the definition of the syntax type `Term` in Listing 1.1 and its extension in Listing 1.3. We can essentially interpret these definitions as the creation of two versions of the `Term` type, namely `LambdaCalculus.Term` and `STLC.Term` and model the extension in the type system through a subtyping relation `LambdaCalculus.Term <: STLC.Term`.

Chapter 4

Design

4.1 Use Case Analysis

Recall that MLang leverages sound language composition programming language to facilitate the rapid development of DSLs. This section outlines various use cases that occur in programming language development when using language composition. Since MLang is itself written in MLang, many of the use cases here are inspired by situations that have naturally arisen during the development of the MLang compiler. The use cases will serve as an inspiration for the type of system that will be created. They will also be used in Chapter 8 to evaluate the resulting implementation.

The first three use cases are already supported by the Miking framework, but they are included in this section for the sake of completeness and because it is important that the prototype implemented in this thesis still supports these use cases.

The final three use cases in this section relate to extensible product types and are unsupported by the previous version of the Miking framework. This section elaborates on the design of the language constructs that were added to MLang exemplified through specific use cases.

4.1.1 Sum Extension

Consider again our running example in Listing 4.2. If we look at the language fragment `IntArith`, we see that we define a sum type `Expr` which can take on two types of values: `IntConst` and `Add`. Our

independent language fragment `BoolArith` similarly defines a language that contains expressions for boolean constants and negations. When we compose these languages into a new language called `IntBoolArith`, we are *extending the sum type* `Expr` to be able to take on all of the values from `Bool`, `Arith`, and `IfThenElse` from the language itself.

A question that arises at that point is whether semantic function `eval`, which takes an argument of this extended sum type, can handle each of the options in this sum type. In other words, is the pattern match in our semantic function exhaustive in this newly composed language? In this specific example, our semantic function is exhaustive, but this is something that is not guaranteed to be the case when using language composition.

For example, consider the addition of the semantic function `negate` to `BoolArith` in Listing 4.1. The operation of boolean negation is only well-defined on boolean expressions, so it is natural that this semantic function only accepts `BoolConst` and `Not` expressions. If the programmer provides any other expression, say an `Add` expression, this will raise a run-time error due to an inexhaustive match. This is exactly the kind of run-time error we would like to catch during the type-checking phase.

```
lang BoolArith = Base
    ...
    sem negate =
    | BoolConst r -> Not {e = BoolConst r}
    | Not r -> r.e
end
```

Listing 4.1: The addition of a negation function to `BoolArith`.

In short, *sum extension* allows us to combine `syn` and `sem` definitions to create larger, more expressive languages. When we compose languages through sum-extension, we do create the possibility for *inexhaustive matches*. To ensure the soundness of composition, a type system for MLang must be expressive enough to detect such inexhaustive matches.

```
lang Base
    syn Expr =

    sem eval : Expr -> Expr
    sem eval =
end

```

```
lang IntArith = Base
    syn Expr =
    | IntConst {val: Int}
    | Add {lhs: Expr, rhs: Expr}

    sem eval =
    | IntConst r -> IntConst r.val
    | Add r ->
        match eval r.lhs with IntConst val1 in
        match eval r.rhs with IntConst val2 in
        IntConst (addi val1 val2)
end

lang BoolArith = Base
    syn Expr =
    | BoolConst {val: Bool}
    | Not {e: Expr}

    sem eval =
    | BoolConst r -> BoolConst r.val
    | Not r ->
        match r.val with BoolConst b in
        BoolConst (not b)
end

lang IntBoolArith = IntArith + BoolArith
    syn Expr =
    | IfThenElse {cond: Expr, thn: Expr, els: Expr}

    sem eval =
    | IfThenElse r ->
        match eval r.cond with BoolConst b in
        if b then
            eval r.thn
        else
            eval r.els
end
```

Listing 4.2: An example Mlang program in which a language fragment for integer and boolean arithmetic are defined and then composed into a new language fragment.

4.1.2 Sum Contraction

Sum contraction is the term that we will use when we have a function that takes a sum type (usually this will be a syntax type) and returns a sum type with fewer options. This use case arises naturally when we have a programming language with syntactic sugar that we would like to get rid of, often called desugaring.

When designing a programming language or DSL, we often include so-called *syntactic sugar*. Essentially, syntactic sugar is a catch-all term for expressions or syntax in a programming language that are convenient to use but could also be expressed in terms of other language constructs with relative ease. For example, consider the addition of an imperative language with a `while` loop construct. Such languages often also contain `for` loops even though all `for` loops can be rewritten into while-loops. In an untyped functional setting, let-bindings can also be considered syntactic sugar since they can be rewritten into a lambda abstraction and application. The transformation from an expression containing syntactic sugar and to an expression without sugar is often called *desugaring*.

In our running example, we could consider the addition of an `Incr` expression to our language. Such an addition would of course be syntactic sugar as this operation can also be expressed trivially by combining `Add` and `IntConst`. We have created the language `SugaredIntArith` which can be found in Listing 4.3.

```
1  lang SugaredIntArith = IntArith + Base
2      syn Expr =
3      | Incr {e: Expr}
4
5      sem desugar =
6      | IntConst r -> IntConst r
7      | Add r -> Add {r with lhs = desugar r.lhs,
8                             rhs = desugar r.rhs}
9      | Incr r -> Add {lhs = r.e,
10                      rhs = IntConst {val = 1}}
11 end
```

Listing 4.3: The creation of a language for integer arithmetic containing syntactic sugar in the form of `Incr`.

A correct implementation of the `desugar` function should ensure that *all* occurrences of `Incr` have been removed. That means that both top-level occurrences but also nested occurrences of `Incr` must be removed.

For instance, the implementation below is incorrect since it does not recursively desugar nested expressions.

```
sem desugar =
    | Incr r -> Add {lhs = r.e, rhs = IntConst {val =
   1}}
    | other -> other
```

If we consider the type of `desugar`, we would like the type of its argument to be any `IntArith` expression including `Incr`. However, the result type should contain `IntConst` and `Add` expressions but explicitly state that no more occurrences of `Incr` can occur. A type system that can express this form of *contraction* would be able to reject the program above in the type checking phase.

4.1.3 Smap and Open Types

In this section, we will motivate the usage of open types and how they relate to the commonly used shallow mapping function in MLang, typically called `smap`.

Recall our integer arithmetic language with syntactic sugar in the form of an `Incr` expression. Suppose now that this language is much bigger and contains many more integer arithmetic expressions like `Sub`, `Mul`, `Negate`, etc. If we think about how such expressions interact with our function that desugars or gets rid of, `Incr` expressions, we'll realize that they all behave identically: They simply recursively desugar their sub-expressions. The definition of this larger language can be found in Listing 4.4 and it should be immediately obvious that we have a lot of undesirable code duplication in our definition of `desugar`.

```
lang IntArith = Base
    syn Expr =
    | IntConst {val: Int}
    | Add {lhs: Expr, rhs: Expr}
    | Sub {lhs: Expr, rhs: Expr}
    | Mul {lhs: Expr, rhs: Expr}
    | Div {lhs: Expr, rhs: Expr}
end
lang SugaredIntArith = IntArith + Base
    syn Expr =
    | Incr {e: Expr}

```

```
13      sem desugar =
14      | IntConst r -> IntConst r
15      | Add r -> Add {r with lhs = desugar r.lhs,
16                             rhs = desugar r.rhs}
17      | Mul r -> Add {r with lhs = desugar r.lhs,
18                             rhs = desugar r.rhs}
19      | Sub r -> Add {r with lhs = desugar r.lhs,
20                             rhs = desugar r.rhs}
21      | Div r -> Add {r with lhs = desugar r.lhs,
22                             rhs = desugar r.rhs}
23      | Incr r -> Add {lhs = r.e, rhs = IntConst {val =
        1}}
24  end
```

Listing 4.4: A larger language for integer arithmetic containing undesirable code duplication in the defnitin of `desugar`.

The idiomatic method for getting rid of such code duplication in MLang is by defining a shallow map function, typically called `smap`. This shallow map is implemented as a higher-order function that takes an expression and a function and applies that function to the direct children of the provided expression. Listing 4.5 shows an alternative definition of `SugaredIntArith` in which we have refactored using `smap`. A critical reader might argue that we have merely moved our code duplication from our `desugar` function to our `smap` function. The key insight here is that we can use reuse `smap` to implement many other language transformations besides `desugar`. Another advantage of using `smap` in this case is that it facilitates easier extension of our language. Consider an extension that adds a constructor `Pow {base : Expr, exp : Expr}` to our language. As long as this extension also defines `smap`, our semantic function `dusugar` does not need to be extended.

```
1   lang IntArith = Base
2       syn Expr =
3       | IntConst {val: Int}
4       | Add {lhs: Expr, rhs: Expr}
5       | Sub {lhs: Expr, rhs: Expr}
6       | Mul {lhs: Expr, rhs: Expr}
7       | Div {lhs: Expr, rhs: Expr}
8
9       sem smap f =
10      | IntConst r -> IntConst r
```

```
    | Add r -> Add {lhs = smap f r.lhs,
                    rhs = smap f r.rhs}
    | Sub r -> Sub {lhs = smap f r.lhs,
                    rhs = smap f r.rhs}
    | Mul r -> Mul {lhs = smap f r.lhs,
                    rhs = smap f r.rhs}
    | Div r -> Div {lhs = smap f r.lhs,
                    rhs = smap f r.rhs}
end
lang SugaredIntArith = IntArith + Base
    syn Expr =
    | Incr {e: Expr}

    sem desugar =
    | Incr r -> Add {lhs = desugar r.e,
                     rhs = IntConst {val = 1}}
    | other = smap desugar other
end
```

Listing 4.5: A language for integer arithmetic in which we have used `smap` to elegantly implement `desugar`.

4.1.4 Product Extension

So far, we've extended our languages through sum extension (i.e. by adding new constructors to syntax types). In this section, we will highlight language composition through *product extension*, i.e., the extension of a language by adding fields to *existing* constructors. We will discuss two common use cases under product extension: Transformations that require a certain field to be present and transformations that extend terms with specific fields.

We've already seen an example of our first use case in subsection 1.2.3. In this example, we go from an untyped lambda calculus to a typed lambda calculus by adding a `tyAnnot` field to our abstraction term through product extension (see Listing 1.3). Our semantic function `typeCheck` only works on terms in which all `TmAbs` terms contain a `tyAnnot` field. On the other hand, the semantic function `eval` is well-behaved regardless of the presence of a `tyAnnot` field. We want to extend the syntax of MLang to support product extension in this manner and to extend the type system such that it prevents illegal projections under

product extension.

In our aforementioned example, the type checker returns the type of each term, but these the nodes in the AST are not annotated with their type. Since types are often used in later stages of compilation (e.g. optimization or code generation) we do not want to recompute them every single time but rather store them directly on the AST nodes themselves.

We can model this through product extension by extending each term with a `ty` field. Our type checking transformation now assigns the computed type to each term and returns this newly extended term. An example of this can be found in Listing 4.6. We would like the type checker for MLang to be powerful enough to accurately capture the type of such functions, i.e., they take a type without a specific field and return that same term, but extended with said field.

```
lang STLC = LambdaCalculus
    syn Type =
    | TyArrow (Type, Type)

    syn Term *=
    | TmVar {ty : Type}
    | TmAbs {tyAnnot : Type, ty : Type}
    | TmApp {ty : Type}

    sem projType =
    | TmVar t -> t.ty
    | TmAbs t -> t.ty
    | TmApp t -> t.ty

    sem assignTy env
    | TmVar t ->
        match mapLookup t.ident env with Some ty
        then TmVar {extend t with ty = ty}
        else error "Unbound variable!"
    | TmAbs t ->
        let body = assignTy
            (mapInsert t.ident t.tyAnnot) t.body in
        let ty = TyArrow (t.tyAnnot, projType body) in
        TmAbs {extend t with ty = ty, body = body}
    | TmApp t ->
        let lhs = assignTy env t.lhs in
        let rhs = assignTy env t.rhs in
```

```
28          match projTy lhs with TyArrow (t1, t2) then
29          if eq t1 (projTy rhs)
30             then TmApp {extend t with lhs = lhs,
31                                       rhs = rhs,
32                                       ty = t2}
33             else error "..."
34          else error "..."
35  end
```

Listing 4.6: A version of STLC in which each term stores its type by extending the untyped lambda calculus through product extension. On lines 5-8, we are extending our syntax type `Term` in the *product dimension*, indicated by the `*=` symbol. On lines 18, 24, and 30, we use a newly added `extend` keyword to add a new field to a record term.

4.1.5 Representing Environments as Extensible Record Types

In programming language theory, many semantic relations are *context-dependent*. E.g., whether the term $f\ 3$ is well-typed with the `Int` type is context-depending, specifically, it depends on the type f. Following convention, the symbol Γ denotes our type checking context, and $\Gamma \vdash f\ 3 : \texttt{Int}$ denotes that the term $f\ 3$ has the type `Int` under the context Γ. For the purposes of type checking the basic STLC, an environment is formally defined as in Equation 4.1.

$$\Gamma ::= \emptyset \mid x : T; \Gamma \tag{4.1}$$

When we go from the STLC to System F (see Chapter 2), we allow the programmer to introduce type variables. In order to determine whether the term $\lambda x : A\ .x$, is only well-typed if A is a type variable that is defined in the current context. To this end, our context must additionally store the set of defined type variables as seen in Equation 4.2.

$$\Gamma ::= \emptyset \mid x : T; \Gamma \mid A; \Gamma \tag{4.2}$$

When developing compilers for programming languages, the transformations that implement our semantic definitions must also include such a context. In compiler construction, we use the terms environment and context interchangeably. Our example on System F motivates the need for the types of our context to be extensible as well.

Although it is possible to model such a context as an extensible sum type, this is computationally inefficient and tedious from a software engineering perspective. Instead, we typically model such contexts as records (i.e. labeled product typed). Our context for type checking in System F could be modeled as a record containing two fields: `varMap: Map String Type` and `tyvars: Set String`.

Under language composition, we want such environments to be extensible. To facilitate this, we need a notion of extensible product types in MLang. We will call the syntactical construct providing extensible product types *co-syntax types*, or `cosyns`. An example of how such `cosyns` can be declared and extended is found in Listing 4.7.

```
1  lang TypeCheckBase
2      cosyn TypeCheckEnv = {}
3  end
4
5  lang STLCTypeCheck = TypeCheckBase
6      cosyn TypeCheckEnv *= {varMap : Map String Type}
7  end
8
9  lang SystemFTypeCheck = STLCTypeCheck
10     cosyn TypeCheckEnv *= {tyvars : Set String}
11 end
```

Listing 4.7: An MLang program the extension of a type checking environment through `cosyns`. On line 2, we define a co-syntax type. On lines 6 and 10, we extend this co-syntax type. We use the `*=` symbol for product extension since co-syntax types are extensible product types.

4.1.6 Extensible Construction of Extensible Product Types

Once you have extensible product types, the use case for the extensible construction of such types naturally arises. Consider our type checking environment from the previous section. When we actually use such a type checking implementation, we usually want to initialize all the fields of the environment to some default or empty value. Since our environment type is extensible, we might not know all of the fields defined on this type and, as such, cannot create it.

This motivates the introduction of a language construct that allows you to define the extensible construction of our extensible product types.

This language feature is the dual of semantic functions (or `sem`) in MLang which allows you to define extensible deconstruction of sum types. Following mathematical naming conventions, we call this language feature a *co-semantic function*, or `cosem`.

In Listing 4.8, you will find an example of the extensible creation of a type checking environment through a co-semantic function. On line 3, we declare an empty co-semantic function using the `cosem` keyword. On lines 8 and 15, we extend this co-semantic function. Lines 9 and 16 contain cases for co-semantic functions. On the left of the `<-` symbol, we have the field to be computed. On the right, we have an expression to be assigned to this field. Co-semantic function can be called just like any other functions (e.g. `let env = emptyEnv in ...`). In our example, our cosemantic function does not take any arguments, but it is also possible to define co-semantic functions that take one or more arguments (e.g. `cosem funName arg1 arg2 = ...`).

```
1  lang TypeCheckBase
2      cosyn TypeCheckEnv = {}
3      cosem emptyEnv =
4  end
5
6  lang STLCTypeCheck = TypeCheckBase
7      cosyn TypeCheckEnv *= {varMap : Map String Type}
8      cosem emptyEnv *=
9      | {TypeCheckEnv of varMap} <- mapEmpty cmpString
10
11 end
12
13 lang SystemFTypeCheck = STLCTypeCheck
14     cosyn TypeCheckEnv *= {tyvars : Set String}
15     cosem emptyEnv *=
16     | {TypeCheckEnv of tyvars} <- setEmpty cmpString
17 end
```

Listing 4.8: An MLang showcasing the creation of the declaration and extenson of extensible construction through a co-semantic function to facilitate the creation of a default, empty type checking environment.

4.2 Type System Design Requirements

In this section, we will start outlining various potential type system designs for MLang. In these designs, we will prioritize the following design goals:

1. Type inference
2. MExpr compatibility
3. Expressivity
4. Conciseness
5. Understandability

4.2.1 Type Inference

Our first requirement is that type inference is that the type-checker is able infer most of the types within MLang programs without user-provided type annotations.

Due to the undecidability of the type inference problem for certain constructs, e.g. higher-rank polymorphism, there will be certain cases where the programmer is forced to provide a type annotation. Our goal is to keep such cases to a minimum.

4.2.2 MExpr Compatibility

Secondly, our MLang type system must be compatible with our current MExpr type system. A lot of previous work has been done on the type-checking, optimization, and compilation of MExpr. The extended version of MLang and the type system that we develop for it should integrate and reuse this existing work as much as possible.

4.2.3 Expressivity

By expressivity, our third design requirement, we mean the ability of a type system to support the use cases we've outlined in section 4.1. The first part of this is that programs such as the ones we've outlined in our use cases must actually be well-typed in the system. In addition to this, we would also like the type system to aid the programmer in the

correct implementation of these use cases by eliminating the possibility of various run-time errors such as inexhaustive matching under sum extension, illegal field projection under product extension, or incorrect implementation of sum contracting transformations.

4.2.4 Conciseness

A crucial aspect of a type system is that its type annotations can both be easily written and easily read by a programmer. This means that type expressions can not be overly verbose, but should instead be *concise*. A concrete example of this is found when we talk about product extension: Suppose we implemented a type system supporting product extension in which type annotations involved the individual fields of each constructor. Realistic ASTs in DSLs have dozens of constructors each with half-a-dozen fields. As such, such a type annotation would have to contain more than 50 individual fields which is obviously much too verbose to be practical.

One key distinction here is that we require the *type annotations* to be concise. These are the type expressions that are exposed to the user. If the internal representation of the types within the type checker were more verbose, this would not be a problem as long as this verbosity is hidden from the end-user.

4.2.5 Understandability

Our last requirement is that the MLang type system be understandable. Our type system should not require a PhD in type theory to be understood. For this requirement, we can again raise the point that the internals of the type system can be much rather complex, as long as the part of the type system that is exposed to an end-user is easily understood.

The design of an MLang type system that meets these five requirements is further complicated by the fact that many of these requirements conflict with each other. For instance, by making a type system more expressive, we are likely to also make it more verbose and complex. This design process can therefore be thought of as a balancing act between these five requirements. A key insight is that we should tailor the design specifically to the use cases we have outlined. We prefer good support for the most

important use cases we care about at the cost of not being able to handle certain, less important use cases to a system in which we support an extremely wide range of programs, but the system is clumsy, complex, and verbose.

4.3 Type System Design Space

In this last section, we will explore the design space of type systems that are suitable for MLang. For each type system, we briefly discuss the intuitions, underlying mechanics, and trade-offs and refer to programming languages that use such type systems.

4.3.1 Mutually Recursive Algebraic Types

One of the simplest and most commonly used type systems for variant types over product types is the *Algebraic Data Type* (ADT) [37]. In this type system, a single identifier is used to range over all inhabitants of a type. An example can be found in Listing 4.9 on line 6. The first version of the MCore type system actually used this form of (mutually recursive) ADTs. They are also widely used in mainstream functional programming languages such as Haskell and OCaml [38, 39].

The major advantages of this system are its simplicity and conciseness. However, its expressivity is severely limited under extension in the product and sum dimensions. Since ADTs assign a single identifier to all inhabitants of a `syn`-type, the type system lacks the mechanics to determine whether a specific field or constructor is present. As a result, ADTs are not able to prevent inexhaustive matches and illegal projections under sum and product extension respectively. Furthermore, the type annotations in a system built on ADTs are not expressive enough to precisely describe common language transformations, e.g., sum contraction in a desugaring pass or product in a type checking pass.

In short, ADTs do provide a simple, concise, and understandable type system. The cost of this is that ADTs cannot ensure the soundness of composition. As a result, the detection of various bugs such as inexhaustive matches and illegal projections falls squarely on the programmer.

```
1 lang SomeLang
2     syn Expr =
```

```
3     | IntExpr {val : Int}
4     | VarExpr {id : String}
5
6     sem eval : Map String Expr -> Expr -> Expr
7     sem eval ctx =
8     | IntExpr r -> IntExpr r
9     | VarExpr r ->
10        match mapLookup r.id ctx with Some e then e
11        else error "Unbound variable!"
12    ...
13 end
```

Listing 4.9: A simple MLang program in which type annotations are in the form of Algebraic Data Types.

4.3.2 Structural, Mutually-Recursive Mu-Types over Row-Polymorphic Variants and Records

In the previous section, we looked at ADTs, a strongly *nominal* type system. In this section, we take a look at the opposite end of the spectrum by considering a strongly *structural* type system. In structural typing literature, the canonical approach to handle mutual recursion is to use mutually recursive μ-types (see subsection 2.3.7). There are various type systems we can turn to in order to handle polymorphism over sum and product types such as subtyping, parametric polymorphism over rows, or parametric polymorphism over presence variables (see section 2.3).

The programming language MLPolyR [31], also discussed in Chapter 3, uses such a type system. More specifically, they use equi-recursive types over row-polymorphic records and variants. Their work convincingly shows that such type systems can achieve a high degree of expressivity whilst preserving type inference. This expressively does come at the cost of conciseness and understandability. These structural type systems typically refer to individual constructors and fields. Since practical DSLs have dozens of constructors with a handful of fields each per syntax type, the type annotations become very verbose.

4.3.3 Homogeneous, Nominal Types with Parametric Polymorphism

So far, we have seen two type systems at the extreme ends of the spectrum: ADTs were very concise but lacked expressivity under extension, and structural recursive types with polymorphic variants and records which were very expressive but fell short in conciseness and understandability. In this section, we strike a balance between these two extremes. We consider a type system in which the types are nominally given (similar to ADTs) but are parameterized on some type term to allow for polymorphism in their variants and records. These parameters then apply to *all* type terms in the fields, leading to what we call *homogeneity*.

The second iteration of the MCore type system called Constructor Types, also discussed in section 3.2, takes this approach. The mechanics of constructor types allow you to specify upper and lower bounds on the constructors in a type. By taking advantage of the duality between variants and records, one can easily extend this type system to work on records. An example of a constructor type annotation can be found in Listing 7.9.

An extension to Constructor Types in which its mechanisms are extended to consider product types can provide a good balance between conciseness and expressivity. It achieves conciseness through homogeneity, i.e., types occur in the same shape throughout the type expression. It is non-obvious how to combine this with transformations that are inherently non-homogeneous such as `smap`.

Although a homogeneous type system can alleviate the verbosity of type annotations by specifying the fields and constructors of this type exactly once, the type annotations still refer to individual fields and constructors. This can be side-stepped by providing a form of syntactic sugar over these constructor-type-like annotations. These sugared type annotations could refer to groups of fields and constructors either through explicitly declared groupings or by inferring groupings from language fragments.

In the rest of this thesis, we develop a type system based on this third option. I.e., a type system based on homogeneous, nominal types built using parametric polymorphism

Chapter 5

Homogeneous, Polymorphic Records

In previous chapters, we've motivated the addition of extensible records in MLang. A direct consequence of the extensibility of records is that they become *polymorphic*. In this chapter, we develop a lambda calculus for extensible polymorphic records centered around the concept of *homogeneity*. Note that the type system developed in this chapter does not directly correspond to the type system that is implemented and discussed in Chapter 7. Instead, this chapter serves to underpin the fundamentals of type checking homogeneous, extensible record types.

5.1 Concrete MExpr Syntax

We choose to design a type system supporting extensible records in MExpr. These extensible records in MExpr then serve as a compilation target for various extensible record constructs in MLang (e.g. product extension co-syntax types). The concrete syntax is chosen to dual to the `type-con` syntax used in MCore for extensible sum types. The key point of this syntax is that the definition of the record type and the definition of its field are decoupled which allows you to extend existing record types. An example can be found in Listing 5.1.

```
1 rectype Foo
2 recfield x : Foo -> Int
3 recfield y : Foo -> Int
4 recfield n : Foo -> String
```

Listing 5.1: An example of an extensible record type.

5.2 Record Polymorphism through Presence Variables

We've seen in section 2.3 that there are many type systems supporting polymorphic records. In this formalization, we choose a form of record polymorphism based on *presence variables*, similar to the work of Rémy [20].

The motivation for choosing presence-based polymorphism is its simplicity. This simplicity allows us to focus on modeling the key concepts: homogeneity and extensibility of records. Many of the findings in this section relating to those concepts can also be applied in a polymorphic type system with different mechanics (e.g. subtyping or row variables). In fact, our implementation in Chapter 7 is not built on presence polymorphism but on bounded quantification over kinds. Despite this, we are still able to use the insights from this section to design the typing rules for extensible, homogeneous records in this different context.

In traditional presence-based row-polymorphism, a row is a collection of labels and types in which each label is also tagged with a *presence indicator*. This states whether this label is absent ($\circ$) or present ($\bullet$). The polymorphism in such type systems comes from the fact that we can also quantify over such presence indicators using type variables. The mechanics of these type variables are borrowed from System F, i.e., we obtain a term for presence variable abstraction (e.g. $\Lambda\theta.\ t$) and presence variable application (e.g. $t\ [\bullet]$). The type of a record is then given by a complete row of presence variables. The major downside of this style of polymorphism is that each row must provide a presence indicator for every single label in the program. This type system falls into the category of structural type systems.

The type system we develop in this chapter shares many of these ideas but takes a more nominal approach. The first major difference is that our rows do not need to store a type, since the type is given statically by a `recfield` expression as seen in Listing 5.1. Additionally, since each field of an extensible record is declared explicitly, our rows do not need to provide a presence annotation for each label, but only for those relevant to a specific type.

If we consider our `Foo` type from Listing 5.1, and we have the expression $e = \{x = 1, y = 2\}$ then we can see that it has the type

$e : Foo\{x^\bullet, y^\bullet, n^\circ\}$. When we define abstractions over such extensible records, we typically quantify over the presence of the fields that are not used as seen in Equation 5.1. This allows us to determine at the call site whether such fields should be present or absent as demonstrated in Equation 5.2.

$$\texttt{xPlus1} \stackrel{\text{def}}{=} \Lambda\theta_1\Lambda\theta_2.\ \lambda \texttt{r} : \texttt{Foo of } \{\texttt{x}^\bullet, \texttt{y}^{\theta_1}, \texttt{z}^{\theta_2}\}.\ \texttt{r}^{\texttt{Foo}}.\texttt{x} + 1 \tag{5.1}$$

$$\texttt{xPlus1} : \forall\theta_1, \theta_2.\ (\texttt{Foo of } \{\texttt{x}^\bullet, \texttt{y}^{\theta_1}, \texttt{z}^{\theta_2}\}). \to \texttt{Int}$$
$$(\texttt{xPlus1}\ [\bullet, \circ])\ \{\texttt{x} = 1, \texttt{y} = 2\}^{\texttt{Foo}} \tag{5.2}$$

5.3 Homogeneity

Unlike our running example, extensible data types in MExpr will typically be mutually recursive. I.e., an extensible data type may contain itself or other extensible data types. To deal with this, we parameterize extensible data types on type variables grouped into a mapping that stores all the presence information for each of the contained records. We also enforce homogeneity:

Definition 1 (Homogeneity) *Consider a type system supporting polymorphic records and variants. We call such a type system* homogeneous *if all occurrences of extensible types within a type are (implicitly) parameterized on the same construct.*

Consider the type in Listing 5.2. Suppose that we want to assign a type to the record `r`. In a structural type system, we can do this without issues. However, a homogeneous type system does not allow this. The reason here is that the homogeneity enforces all occurrences of `Foo` to be of the same type, which is clearly not the case in r.

```
1 rectype Foo
2 recfield x : Foo -> Int
3 recfield y : Foo -> Int
4
5 rectype Bar
6 recfield f1 : Bar -> Foo
7 recfield f2 : Bar -> Foo
8
9 let r = {Bar of f1 = {Foo of x = 1},
```

```
10                f2 = {Foo of y = 1}}
```

Listing 5.2: An example of an extensible record type.

The main advantage of homogeneity is its conciseness: A type annotation needs to refer to each contained record type exactly once, instead of potentially needing to refer to a different type for each field as you would in a structural setting. The rest of this chapter formalizes a homogeneous type system for extensible records based on presence indicators.

5.4 Induced Definitions and Mappings

The extensible product types are given by a mapping from the symbol of the extensible type to a presence-row. We use the Map kind to ensure that the mapping is complete, i.e., every extensible type is mapped to a row. We do this to clean up the formalization whilst not placing any restrictions because if one extensible type does not depend on another, the typing rules will never check the presence rows. We also use the syntax $M[\tau]$ to retrieve a specific presence row for the type τ in mapping M. Due to our kinding rules, $M[\tau]$ is always well-defined for any mapping of kind $\mathsf{Map}_\emptyset$.

So far, we have considered the concrete syntax as outlined in Listing 5.1. However, in our formalization, we do not consider this syntax directly. Instead, this syntax induces various definitions in our formalization.

- The set of identifiers of extensible product types and their individual identifiers are denoted as $\mathcal{T} \ni \tau$ respectively.
- The function $fields : \mathcal{T} \to \mathcal{P}(\mathcal{L})$ returns the set of fields for a given extensible product type.
- The function $ty(\tau, \ell)$ returns type of a given field in an extensible product type. The returned type is *parameterized* on a mapping M. More formally, ty is a partial mapping from $\mathcal{T} \times \mathcal{L} \hookrightarrow \mathsf{Type}$ where the kind of this type is $\mathsf{Map}_\emptyset \Rightarrow \mathsf{Ty}$.

Consider the extensible product types defined in Listing 5.3. The definitions induced by this concrete syntax can be found in Equation 5.3.

```
rectype Foo
recfield x: Foo -> Int

rectype Bar
recfield f1: Bar -> Foo
recfield f2: Bar -> Foo
```

Listing 5.3: A simple example of extensible datatypes.

$$\mathcal{T} \stackrel{\text{def}}{=} \{\texttt{Foo}, \texttt{Bar}\} \tag{5.3}$$

$$\begin{aligned} \mathsf{fields}(\texttt{Foo}) &\stackrel{\text{def}}{=} \{\texttt{x}\} \\ \mathsf{fields}(\texttt{Bar}) &\stackrel{\text{def}}{=} \{\texttt{f1}, \texttt{f2}\} \end{aligned}$$

$$\begin{aligned} \mathsf{ty}(\texttt{Foo}, \texttt{x}) &\stackrel{\text{def}}{=} \Lambda M.\ \mathsf{Int} \\ \mathsf{ty}(\texttt{Bar}, \texttt{f1}) &\stackrel{\text{def}}{=} \Lambda M.\ \mathsf{Bar\ of}\ M \\ \mathsf{ty}(\texttt{Bar}, \texttt{f2}) &\stackrel{\text{def}}{=} \Lambda M.\ \mathsf{Bar\ of}\ M \end{aligned}$$

5.5 Abstract Syntax

The abstract syntax can be found in Table 5.1. A large part of this abstract syntax is standard for row-polymorphism-based lambda calculi, but we will take a closer look at some of the non-standard definitions.

The meta-variable P ranges over presence indicators. Per the convention adopted by Remy, presence and absence are denoted by $\bullet$ and $\circ$ respectively [20]. The meta-variable θ ranges over all presence variables which are typically denoted as θ with some subscript.

We use the variable ρ to range over all rows. A row is either empty, denoted as ϵ, or it is a combination of a label ℓ with a presence P and another row. Our definition here deviates from those typically found in the literature since we do not carry an explicit type annotation. In purely structural type systems this row must carry a type for each label. However, since our nominal definition of extensible product types provides a definition for each label through the *ty* mapping, this type annotation can be omitted in rows.

Mappings from extensible product type symbols to their respective presence rows are given by the meta-variable M. Similarly to our definition of rows, a mapping is either empty (denoted as ϵ) or a combination of a type symbol and a row. In our formalization, we use the syntax $M[\tau]$ to retrieve the row for type τ in mapping M. Our kinding rules will ensure that this operation is well-defined.

Most of the syntax for expressions is rather standard. One thing to note is that the meta-variables x and n range over variable names and integers respectively. We use the syntax of System F for abstraction and application of type variables specialized to presence variables (i.e. $\Lambda\theta.\ e$ and $e\ [P]$).

```
rectype T1
field x: Foo -> Int
field y: Foo -> Int

rectype T2
field x: Foo -> Int
field z: Foo -> Int
```

Listing 5.4: Two extensible product types which share a label.

Additionally, we should point out that our record operations for creation, projection, and updating are all syntactically parameterized on the identifier of an extensible product type. This is because multiple extensible product types can use the same labels making it undecidable which type should be used. Consider the types defined in Listing 5.4. Both `T1` and `T2` contain the label `x` so when one creates the record $\{x = 1\}$, it is unclear whether it should be typed as $T_1\ of\ \{x^\bullet, y^\circ\}$ or $T_2\ of\ \{x^\bullet, z^\circ\}$. To resolve this ambiguity, our syntax for record construction contains the symbol of the extensible product type: $\{x = 1\}^{T_1}$. It is worth pointing out that this syntactic parameter happens for the sake of the formalization; in the actual implementation, we should try to infer the most likely type through context whilst giving the user the possibility to provide an explicit annotation.

$$
\begin{aligned}
\mathsf{ExtType} &\supseteq \mathcal{T} \ni \tau \\
\mathsf{Label} &\supseteq \mathcal{L} \ni \ell \\
\\
\mathsf{Presence} &\ni P ::= \theta \mid \circ \mid \bullet \\
\mathsf{Row} &\ni \rho ::= \epsilon \mid \ell^P ; \rho \\
\mathsf{Map} &\ni M ::= \epsilon \mid \tau \mapsto \rho ; M \\
\mathsf{Type} &\ni T ::= \mathsf{Int} \mid T \to T \mid \tau \text{ of } M \mid \forall \theta.\ T \\
\mathsf{Kind} &\ni K ::= \mathsf{Ty} \mid \mathsf{Pre} \mid \mathsf{Row}^{\tau}_{\mathcal{L}} \mid \mathsf{Map}_{\mathcal{T}} \mid \star \\
\\
\mathsf{Env} &\ni \Gamma ::= \epsilon \mid \Gamma, x : T \mid \Gamma, \theta \\
\mathsf{TyEnv} &\ni \Delta ::= \epsilon \mid \Delta, \theta :: \mathsf{Kind} \\
\\
\mathsf{Expr} &\ni e ::= x \\
&\quad \mid n \\
&\quad \mid \lambda x : T.e \\
&\quad \mid e_1\ e_2 \\
&\quad \mid e_1 + e_2 \\
&\quad \mid e^{\tau}.\ell \\
&\quad \mid \{\ell_1 = e_1, \ldots, \ell_n = e_n\}^{\tau}_i \\
&\quad \mid \{\mathsf{update}\ r\ \mathsf{with}\ \ell = e\}^{\tau} \\
&\quad \mid \{\mathsf{extend}\ r\ \mathsf{with}\ \ell = e\}^{\tau} \\
&\quad \mid \Lambda\theta.e \\
&\quad \mid e\ [P]
\end{aligned}
$$

Table 5.1: The abstract syntax of an extension of the simply typed lambda calculus supporting extensible record types.

5.6 Kinding Rules

The kinding rules for our STLC extension can be found in Table 5.2. The kinds $\star$ and Ty are used for well-kinded expressions and types respectively and most of their associated rules are standard and do not warrant further

discussion. We will more closely examine the rules K-EmptyRow, K-ExtendRow, K-EmptyMap, K-ExtendMap, and K-Of.

The kind for rows $\mathsf{Row}^{\tau}_{\mathcal{L}}$ denotes a row for the extensible product type τ in which the labels $\mathcal{L}$ do not occur. We can apply the rule K-EmptyRow to any empty row for any type τ to obtain the kind $\mathsf{Row}^{\tau}_{fields(\tau)}$ indicating that this row is missing presence annotations for all fields in $fields(\tau)$. We can then subsequently apply K-ExtendRow to remove labels from the set of missing labels. When we do this, we must ensure that the label that is being added does belong to τ (i.e. $\ell \in fields(\tau)$) and that the presence annotation is well-kinded under Δ. Additionally, we ensure that the label is still missing from this row. This is done by checking that the old set of missing labels is the disjoint union of the new set of missing labels and the singleton set of the added label (i.e. $\mathcal{L}' = \mathcal{L} \uplus \{\ell\}$). A row is then considered *complete for* τ if no labels are missing, or in other words, when it has the kind $\mathsf{Row}^{\tau}_{\emptyset}$.

The kind for mappings $\mathsf{Map}_{\mathcal{T}}$ indicates a mapping that is missing entries for all types in $\mathcal{T}$. Similarly to our kinding rules for products, when can assign the kind $\mathsf{Map}_{\mathcal{T}}$ to an empty mapping since $\mathcal{T}$ denotes the set of all extensible product types. Again analogously to our kinds for rows, we can apply the rule K-ExtendMap to add an entry to our mapping. We ensure that no mapping was present for τ through the third premise and its disjoint union. We also ensure that the row in our entry is complete for τ in the first premise. We say that a *mapping is complete* when no entries are missing, i.e., it has the kind $\mathsf{Map}_{\emptyset}$.

Lastly, we use the rule K-Of to say that a type expression containing an extensible product type (i.e. $\tau\ of\ M$) is well-kinded if its mapping is complete.

$$\textsc{K-EmptyRow}\ \frac{}{\Delta \vdash \epsilon :: \mathsf{Row}^{\tau}_{\mathsf{fields}(\tau)}} \qquad \textsc{K-RowExtend}\ \frac{\begin{array}{c}\Delta \vdash P :: \mathsf{Pre} \\ \ell \in \mathsf{fields}(\tau) \\ \Delta \vdash \rho :: \mathsf{Row}^{\tau}_{\mathcal{L}'} \\ \mathcal{L}' = \mathcal{L} \uplus \{\ell\}\end{array}}{\Delta \vdash \ell^{P}; \rho :: \mathsf{Row}^{\tau}_{\mathcal{L}}}$$

$$\textsc{K-EmptyMap}\ \frac{}{\Delta \vdash \epsilon :: \mathsf{Map}_{\mathcal{T}}} \qquad \textsc{K-ExtendMap}\ \frac{\begin{array}{c}\Delta \vdash \rho :: \mathsf{Row}^{\tau}_{\emptyset} \\ \Delta \vdash M :: \mathsf{Map}_{\mathcal{T}'} \\ \mathcal{T}' = \mathcal{T} \uplus \{\tau\}\end{array}}{\Delta \vdash \tau \mapsto \rho; M :: \mathsf{Map}_{\mathcal{T}}}$$

$$\textsc{K-Of}\ \frac{\Delta \vdash M :: \mathsf{Map}_{\emptyset}}{\Delta \vdash \tau\ \mathsf{of}\ M :: \mathsf{Ty}} \qquad \textsc{K-Int}\ \frac{}{\Delta \vdash \mathsf{Int} :: \mathsf{Ty}}$$

$$\textsc{K-TyArrow}\ \frac{\Delta \vdash T_1 :: \mathsf{Ty} \quad \Delta \vdash T_2 :: \mathsf{Ty}}{\Delta \vdash T_1 \to T_2 :: \mathsf{Ty}} \qquad \textsc{K-ForAll}\ \frac{\Delta, \theta \vdash T :: \mathsf{Ty}}{\Delta \vdash \forall \theta.\ T :: \mathsf{Ty}}$$

$$\textsc{K-Pre}\ \frac{}{\Delta \vdash \bullet :: \mathsf{Pre}} \qquad \textsc{K-Abs}\ \frac{}{\Delta \vdash \circ :: \mathsf{Pre}} \qquad \textsc{K-PreVar}\ \frac{\theta :: \mathsf{Pre} \in \Delta}{\Delta \vdash \theta :: \mathsf{Pre}}$$

$$\textsc{K-PreAbs}\ \frac{\Delta, \theta :: \mathsf{Pre} \vdash e :: \star}{\Delta \vdash \Lambda\theta.e :: \star} \qquad \textsc{K-PreApp}\ \frac{\Delta \vdash e :: \star \quad \Delta \vdash P :: \mathsf{Pre}}{\Delta \vdash e\ [P] :: \star}$$

$$\textsc{K-Var}\ \frac{}{\Delta \vdash x :: \star} \qquad \textsc{K-Num}\ \frac{}{\Delta \vdash n :: \star} \qquad \textsc{K-Num}\ \frac{\Delta \vdash e_1 :: \star \quad \Delta \vdash e_2 :: \star}{\Delta \vdash e_1\ e_2 :: \star}$$

$$\textsc{K-Proj}\ \frac{\Delta \vdash e :: \star}{\Delta \vdash e^{\tau}.\ell :: \star} \qquad \textsc{K-Create}\ \frac{\forall i \in 1..n.\ \Delta \vdash e_i :: \star}{\Delta \vdash \{\ell_i = e_i\}^{\tau} :: \star}$$

$$\textsc{K-Add}\ \frac{\Delta \vdash e_1 :: \star \quad \Delta \vdash e_2 :: \star}{\Delta \vdash e_1 + e_2 :: \star}$$

Table 5.2: Kinding rules for an extension of STLC with extensible record types.

5.7 Typing Rules

The typing rules can be found in Table 5.3. As is the case with our kinding rules, most of the rules are standard for STLC with System F-style presence polymorphism. We will take a closer look at the rules for record operations.

The rule T-RecordCreation assigns a type to a newly created record. The first premise ensures that the labels in the actual record match which are annotated as present in M. Because of our kinding rules for mappings, we know that $M[\tau]$ will in fact exist. Furthermore, our kinding rules for rows indicate that all labels must be members of $fields(\tau)$ and that a presence indicator is included in $M[\tau]$. Similarly, the second premise ensures that all fields that are not present in the record term, are marked as absent in the type. Our last premise simply states that each of contained expressions is well-typed in accordance with our *ty*-mapping. The homogeneity comes into play here because the same mapping M is propagated into the contained expressions.

To assign a type to record projection, we use the rule T-RecordProj. The first premise of this rule checks that the left-hand side of the projection is in fact a record type of τ with some mapping M. Our second premise ensures that the label we are projecting is annotated as present in M for τ. Lastly, the returned type is easily retrieved from the *ty*-mapping by applying M.

For our record extension rule, we define two auxiliary functions. Firstly, we define $\mathsf{diff}(M, M')$ to be the set of extensible type symbols that differ between M and M' where both are complete mappings as in Equation 5.4. Additionally, the function $\mathsf{tydeps}(\tau, \ell)$ denotes the set of extensible product types which can be contained in field ℓ in type τ. The rule T-RecordExtend has five premises. Premises (i) and (ii) are straightforward and state that the record we are updating must be well-typed with mapping M and that the sub-expressions we are extending with must be well-typed with mapping M' respectively. The fourth and fifth premises combined ensure that the inserted labels are present in the resulting mapping M' and that unaltered labels have the same presence annotations in M' as in M. The premise that is doing the heavy lifting is the third. This rule states that any label ℓ in $\mathsf{fields}(\tau)$ either does not contain any extensible product types whose mapping we are altering (i.e. $\mathsf{tydeps}(\tau, \ell) \cap \mathsf{diff}(M, M') = \emptyset$), this label is being altered by this

extension ($\exists i \in 1..n.\ \ell_i = \ell$), or this label is absent ($\ell^\circ \in M[\tau]$).

$$\mathsf{diff}(M, M') = \{\tau \in \mathcal{T} \mid M[\tau] \neq M'[\tau]\} \tag{5.4}$$

Our last rule for record updates, T-RecordUpdate, states that it is always safe to update a present field as long as the type does not change. It turns out that this rule is actually strictly less expressive than T-RecordExtend, but we have included it here as it is much more easily understood.

$$\text{T-Var}\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \text{T-Abs}\frac{\Gamma, x : T_1 \vdash e : T_2}{\Gamma \vdash \lambda x : T_1.e : T_1 \to T_2}$$

$$\text{T-App}\frac{\Gamma \vdash e_1 : T_1 \to T_2 \quad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1\ e_2 : T_2}$$

$$\text{T-Num}\frac{}{\Gamma \vdash n : \mathsf{Int}} \qquad \text{T-Add}\frac{\Gamma \vdash e_1 : \mathsf{Int} \quad \Gamma \vdash e_2 : \mathsf{Int}}{\Gamma \vdash e_1 + e_2 : \mathsf{Int}}$$

$$\text{T-TApp}\frac{\Gamma \vdash e : \forall\theta.\ T}{\Gamma \vdash e\ [P] : T[\theta \mapsto P]} \qquad \text{T-TAbs}\frac{\Gamma, \theta \vdash e : T}{\Gamma \vdash \Lambda\theta.\ e : \forall\theta.T}$$

$$\text{T-RecordProj}\frac{\begin{array}{c}\Gamma \vdash r : \tau\ \mathsf{of}\ M \\ \ell^\bullet \in M[\tau] \\ T = ty(\tau, \ell)[M]\end{array}}{\Gamma \vdash r^\tau.\ell : T}$$

$$\text{T-RecordCreation}\frac{\begin{array}{c}\{\ell_i \mid i \in 1..n\} = \{\ell \mid \ell^\bullet \in M[\tau]\} \\ \mathsf{fields}(\tau) \setminus \{\ell_i \mid i \in 1..n\} = \{\ell \mid \ell^\circ \in M[\tau]\} \\ \forall i \in 1..n.\ \Gamma \vdash e_i : ty(\tau, \ell_i)[M]\end{array}}{\Gamma \vdash \{\ell_i = e_i\}^\tau_{i \in 1..n} : \tau\ \mathsf{of}\ M}$$

$$\text{T-RecordUpdate}\frac{\begin{array}{c}\Gamma \vdash r : \tau\ \mathsf{of}\ M \\ \Gamma \vdash e : ty(\tau, \ell)[M] \\ \ell^\bullet \in M[\tau]\end{array}}{\Gamma \vdash \{r\ \mathsf{with}\ \ell = e\}^\tau : \tau\ \mathsf{of}\ M}$$

$$\text{T-RecordExtend}\frac{\begin{array}{c}\text{(i)}\ \Gamma \vdash r : \tau\ \mathsf{of}\ M \\ \text{(ii)}\ \forall i \in 1..n.\ \Gamma \vdash e_i : ty(\tau, \ell_i)[M'] \\ \text{(iii)}\ \forall \ell \in \mathsf{fields}(\tau).\ (\mathsf{tydeps}(\tau, \ell) \cap \mathsf{diff}(M, M') = \emptyset \vee \\ (\exists i \in 1..n.\ \ell_i = \ell) \vee \ell^\circ \in M') \\ \text{(iv)}\ \forall i \in 1..n.\ \ell_i^\bullet \in M'[\tau] \\ \text{(v)}\ \forall \ell \in (\mathsf{fields}(\tau) \setminus \{\ell_i \mid i \in 1..n\})\ .\ell^{M[\tau][\ell]} \in M'[\tau]\end{array}}{\Gamma \vdash \{r\ \mathsf{with}\ (\ell_i = e_i)_{i \in 1..n}\}^\tau : \tau\ \mathsf{of}\ M'}$$

Table 5.3: Typing rules for an extension of STLC with extensible record types.

5.8 Evaluation Rules

The call-by-value semantics of our lambda calculus are standard for a simply typed lambda calculus with System F style presence polymorphism and can be found in Table 5.4.

$$v ::= \lambda x : T.e \mid \Lambda X.e \mid n \mid \{\ell_i = v_i\}_{i \in 1..n}$$

$$\text{E-App1}\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad \text{E-App2}\frac{e_2 \to e_2'}{v_1\ e_2 \to v_1\ e_2'}$$

$$\text{E-AppAbs}\frac{}{(\lambda x : T.\ e)\ v \to [x \mapsto v]e} \qquad \text{E-Add1}\frac{e_1 \to e_1'}{e_1 + e_2 \to e_1' + e_2}$$

$$\text{E-Add2}\frac{e_2 \to e_2'}{n + e_2 \to n + e_2'} \qquad \text{E-Add3}\frac{n_1\ \mathcal{I}(+)\ n_2 = n}{n_1 + n_2 \to n}$$

$$\text{E-TApp}\frac{e \to e'}{e\ [P] \to e'\ [P]} \qquad \text{E-TAppAbs}\frac{}{(\Lambda\theta.\ e)\ [P] \to [\theta \mapsto P]e}$$

$$\text{E-Proj1}\frac{e \to e'}{e^\tau.\ell \to e'^\tau.\ell} \qquad \text{E-Proj2}\frac{}{\{\ell_i = e_i\}^\tau_{i \in 1..n}.\ell_j \to e_j}$$

$$\text{E-Record}\frac{e_j \to e_j'}{\{\ell_i = e_i\}^\tau_{i \in 1..n} \to \{(\ell_i = e_i)_{i \in 1..n, i \neq j}\ \ \ell_j = e_j'\}^\tau}$$

$$\text{E-Update1}\frac{e \to e'}{\{e\ \mathsf{with}\ (\ell_i = e_i)_{i \in 1..n}\} \to \{e'\ \mathsf{with}\ (\ell_i = e_i)_{i \in 1..n}\}}$$

$$\text{E-Update2}\frac{e_j \to e_j'}{\{e\ \mathsf{with}\ (\ell_i = e_i)_{i \in 1..n}\} \to \{e\ \mathsf{with}\ (\ell_i = e_i)_{i \neq j}\ \ \ell_j = e_j\}}$$

$$\text{E-Update3}\frac{}{\{\{\ell_i = v_i\}_{i \notin L}\ \mathsf{with}\ (\ell_i = v_i')_{i \in L}\} \to \{(\ell_i = v_i')_{i \in L}\ \ (\ell_i = v_i)_{i \notin L}\}}$$

Table 5.4: Evaluation rules for an extension of STLC with extensible record types.

5.9 Metatheoretical Properties

This section states the theorems of progress and preservation. However, since no proofs are given, they are stated as conjectures.

Conjecture 1 (Progress) *For all expressions e that are well-kinded (i.e. $\epsilon \vdash e :: \star$) and for which there exists a type T such that e is of type T under an empty context (i.e. $\epsilon \vdash e : t$), either e is a value or there exists some other expression e' such that e evaluates to e' (i.e. $e \to e'$)*

Conjecture 2 (Preservation) *For all expressions e and e' and for all types T, if e is well-kinded and of type T under an empty context and e evalutes to e', then e' must also be well-kinded of type T under the empty context.*

Chapter 6

MLang Compiler Infrastructure Improvements

This chapter gives an overview of the Miking compiler infrastructure and the changes that were made as a part of this thesis. We first talk about the current version of the compiler and highlight various issues with its infrastructure. After this, we discuss how the compiler was updated to alleviate these issues as a part of this thesis work. The latter section also gives an overview of crucial implementation details of various MCore compiler passes that were implemented or ported.

When writing compilers, it is common practice to *bootstrap* them. This means that the compiler for a certain source language is written in that source language itself. One of the major advantages of bootstrapping is that the compiler of a language serves as a major test case for itself. This is especially applicable to MLang as the language's primary use case is the construction of compilers. The MLang compiler in the Miking project is *partially bootstrapped*: A large part of the MLang compiler is in fact written in MLang itself but a part of it is written in OCaml. As a result, it is possible to bootstrap the compiler when targeting OCaml code, but it is not yet possible to bootstrap when targeting other compiler back-ends. Henceforth, we will refer to the part of the compiler written in MLang as `mi` and the part of the compiler written in OCaml as `boot`.

The MLang compiler was split into these two different components in the initial stage of development for a pragmatic reason. When development on the Miking framework started, an interpreter for MLang was written in `boot`. As a part of this interpreter, a lexer and parser were implemented. Since the original Miking compiler primarily targets

OCaml, it made sense to reuse the lexer and parser.

In addition to parsing, various other transformations are also done in `boot` such as the compilation from MLang to MExpr. This split of the MLang compiler into `boot` and `mi` has two major downsides. Firstly, we can only bootstrap the compiler when we are targeting OCaml and not any of the other back-ends since `boot` was written in OCaml. Secondly, various transformations must be written in both `boot` and `mi`. For these reasons, the Miking compiler should move away from this split architecture of the MLang compiler into `boot` and `mi` towards a fully bootstrapped compiler.

6.1 Overview of Current Pipeline

A schematic overview of the current pipeline can be found in Figure 6.1. At a high level, the MLang compiler can be described as follows: Firstly, `mi` parses the command line arguments and determines which MLang file should be compiled. This file path is then passed from `mi` to `boot` which subsequently parses this MLang file and compiles it down to MExpr. The compiled MExpr is then returned by `boot` to `mi` and `mi` then performs various transformations and optimizations before this MExpr is translated into the desired target by one of the compiler backends. From our schematic overview, we can conclude that the responsibilities of `mi` are to provide a command-line interface, interact with `boot`, and perform all transformations related to MExpr. The responsibilities of `boot` are to parse MLang files into MLang ASTs and to compile MLang down to MExpr.

One of the major downsides of this pipeline architecture is the duplication of certain transformations as we will demonstrate through two examples. Consider symbolization (or name resolution): we perform a symbolization pass on the MLang AST in `boot` and on the MExpr AST in `mi`. The architecture also causes the need for pattern analysis to be implemented in both `boot` and `mi`. Recall from subsection 2.2.2 that MLang has an unusual language feature in which the patterns in a semantic function are ordered during compilation. We call the logic that underpins this ordering pattern analysis. Pattern analysis is done in `boot` to perform language composition checks and the compilation from MLang to MExpr, but is also used in `mi` during type checking and the pattern lowering pass. The fact that pattern analysis needs to be implemented

in `boot` and in `mi` leads to various maintainability problems.

6.2 Overview of New Pipeline

Now that we have a solid overview of the existing pipeline and its shortcomings, we can design and implement a new pipeline. The difference between the current and new pipeline can most easily be described through the responsibilities of `boot`. In the new pipeline, the only responsibility that `boot` has is to parse an MLang file into an undecorated MLang AST. All of the functionality related to compiling MLang and MExpr has been handed over to `mi`. A schematic overview of this new pipeline architecture can be found in Figure 6.2.

In the best-case scenario, all of the responsibilities of `boot` would be transferred to `mi` after bootstrapping. In this scenario, the sole purpose of `boot` would be to kickstart the bootstrapping process through its interpreter. Since the only remaining responsibility of `boot` after bootstrapping in the new pipeline is to parse MLang files into MExpr, this can be done by implementing an MLang parser in `mi`. Furthermore, various optimizations currently implemented in `boot` would have to be re-implemented in `mi`*. Whilst both are certainly achievable, they are not a part of this thesis work due to time constraints.

6.2.1 Parsing

Once the command-line arguments have been parsed, we need to parse an MLang file into an MLang AST. The actual parsing is done by `boot` through the Menhir parser generator [40]. Unlike the current version boot, no transformations are performed on the parsed MLang AST. Instead, the untouched AST is transferred directly to `mi`.

One aspect of the miking compiler we have not yet touched on is the communication between `boot` and `mi`. Since `boot` and `mi` are different written in different programming languages, communication can not occur directly. However, since `boot` is written in OCaml and `mi` is usually compiled to OCaml, `boot` can be embedded in the `mi` executable as an OCaml library. This `boot` library then exposes various functions and datatypes in the form of MLang *intrinsics*: These are functions and

*We chose not to port these existing optimizations to the new pipeline due to time constraints.

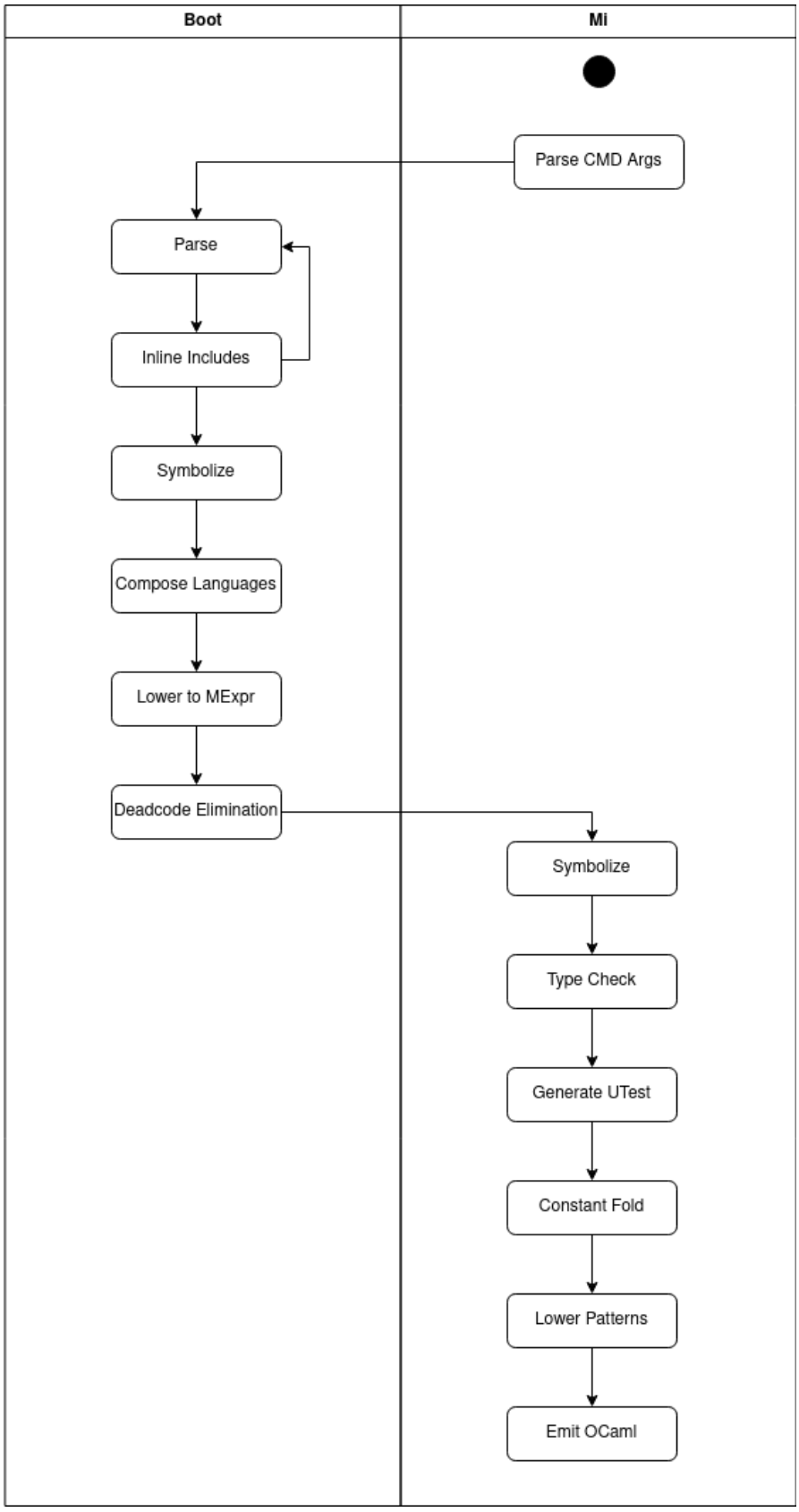

Figure 6.1: A schematic overview of the current architecture of the MLang compiler.

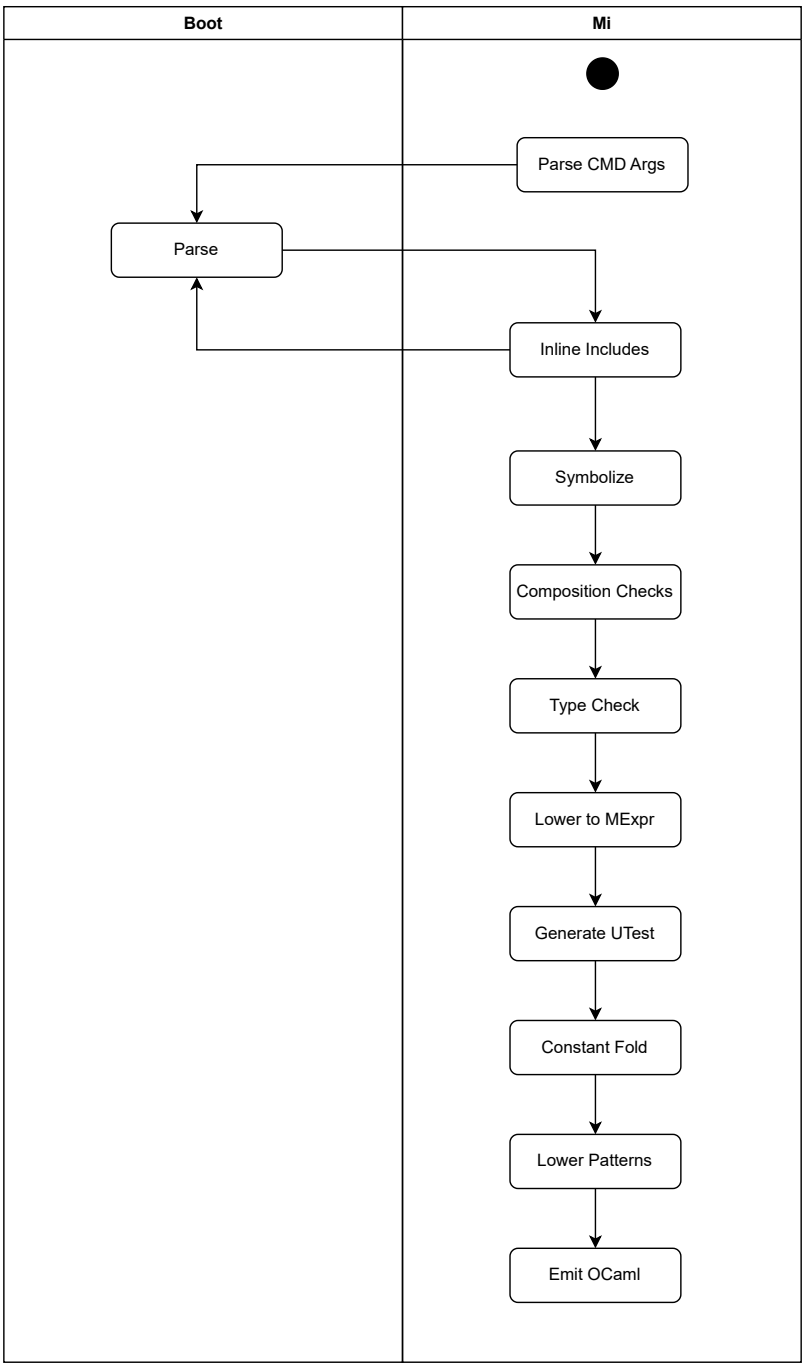

Figure 6.2: A schematic overview of the new architecture of the MLang compiler in which all responsibilities of `boot` except parsing have been transferred to `mi`.

types whose definitions are not given by an MLang definition but are linked to the correct behavior during compilation.

For the purposes of parsing, intrinsics provide a mechanism for querying `boot` from `mi`. The first important intrinsics are called `bootParserParseMLangString` and `bootParserParseMLangFile` which parse an MLang file or string into an AST. This AST is then returned to `mi`. The datatype of such an AST is an intrinsic type called `BootParserTree`. This type is *opaque*, i.e., unlike traditional MLang types it can not be deconstructed through pattern matching. Instead, we provide intrinsics that operate on this `BootParserTree` type for deconstruction such as `bootParserGetString` or `bootParserGetInt`.

In the current version of the pipeline, it was the responsibility of `boot` to lower all MLang constructs into MExpr constructs. Because of this, there was only a mechanism for transferring MExpr ASTs from `boot` to `mi`, and no mechanism for transferring an MLang AST from `boot` to `mi`. In the new pipeline, the boot parser was extended to support the transfer of MLang ASTs from `boot` to `mi` by adding various intrinsics. This extension can be found in the file `stdlib/mlang/boot-parser.mc`.

6.2.2 Include Handling

When an AST is parsed by `boot` and transferred to `mi`, it might still contain include statements. Includes are handled rather crudely in MLang as the top-level declarations of the included file are simply inlined. Since the included file itself can contain includes itself, this process occurs transitively. If this were done naively, this would cause a lot of code duplication since certain files, such as commonly used library files, may be included dozens of times through different include statements. The current, crude solution for this is to only include each file once. More specifically, when an include is encountered, we check whether this file has been included already and if so, we simply do nothing, and otherwise, we parse the file and inline its top-level declarations.

Whilst this does solve the problems related to code duplication, this solution is far from perfect in situations where identifiers are re-used. The problem that we run into, is that the order of imports matters. Consider the code in listing Listing 6.1. Here we have a main file which includes two files `foo.mc` and `bar.mc` both of which define a variable called `f`. Since `foo.mc` also includes `bar.mc`, the include in `main` is ignored causing `f` to point to the definition in `bar`.

These issues can be solved by adding a more sophisticated system that handles the inclusion of functionality from other files, such as OCaml modules [41]. However, the design space of module systems is sizeable, and the interactions between language composition and modules are non-obvious. Because of this, the Miking development team opted to go with the basic include-handling described in this section and to replace it with an improved system down the line.

```
-- START foo.mc --
let f = 10
-- END foo.mc

-- START bar.mc
include "foo.mc"

-- The f on the right-handside refers to the
-- one in foo.mc.
-- Thus the value of f in bar.mc is 11
let f = f + 1
-- END bar.mc

-- START main.mc
include "bar.mc"

-- Since bar.mc already includes foo.mc, we
-- ignore this include
include "foo.mc"

-- This test will fail even though the f we included
-- last in this file should be the one from foo.mc.
-- Since this foo.mc was already included by bar.mc,
-- this include is ignored.
mexpr
utest f with 10 in ()
-- END main.mc
```

Listing 6.1: An example of an MLang program which has unintuitive include-related behaviour.

6.2.3 Language Inclusion Generation and Symbolization

In MLang, it is possible to reuse the same string identifier to refer to multiple terms as illustrated in Listing 6.2. For the rest of the compilation pipeline, it is however easier if each identifier refers to a unique definition. In the MLang compiler, this is done by defining identifiers in terms of two things: A string and a symbol. The string is simply the identifier that is given by the programmer whilst the symbol can be thought of as a token that signifies a unique definition. When an AST is parsed, all identifiers are represented as strings without symbols. The *symbolization pass* assigns a unique symbol to each definition and ensures that the usage of an identifier is assigned the proper symbol. The symbolization pass is also sometimes referred to as name resolution.

```
1 let f = lam f. addi f 1
2 utest f 10 with 11
```

Listing 6.2: An example of an MLang program which re-uses an identifier.

Symbolization for MExpr is straightforward since all definitions are explicit. It boils down to a traversal of the AST in which we generate symbols at definitions and lookup symbols at the usages of identifiers. We distinguish four kinds of symbols: variables, type variables, constructors, and type constructors.

The symbolization of MLang code can be built on top of the existing symbolization of MExpr code, but there are some complications we need to take care of: Firstly, MLang introduces namespacing based on language fragments through `use` expressions. Secondly, unlike MExpr, MLang has implicit definitions of semantic functions under language composition.

Creating Explicit Declarations for Implicits

When we use language composition in MLang, we end up with many different language fragments that have their own semantic functions. These semantic functions will share the same string identifier but might have different behavior. As such, during symbolization, each of those semantic functions should be assigned a unique symbol. To complicate matters further, the composition of languages can implicitly define semantic functions with unique behavior. In order to generate names for these implicit declarations, we perform a transformation, located in `stdlib/mlang/language-composer.mc` which generates explicit

declarations for such implicit declarations in the AST. An example of this can be found in Listing 6.3. In addition to generating such declarations, this transformation also establishes the inclusion relationships between various semantic functions and syntax definitions.

```
-- In the program below, the  transformation will
-- generate a semantic function declaration called
-- f in L12 that includes  L1.f and L2.f.
lang L0 =
  sem f = ...
end
lang L1 = L0
  sem f = ...
end
lang L2 = L0
  sem f = ...
end
lang L12 = L1 + L2
end
```

Listing 6.3: An example of an MLang program in which an implicit declararion will be made implicit.

Symbolization of Language Fragment

Once the implicit declarations of each language fragment have been made explicit, we must perform symbolization of the language fragment. As you may recall from Chapter 2, a language fragment contains type alias definitions, semantic function definitions or extensions, and syntax definitions or extensions. The order of operations in this symbolization pass is very important since these declarations are very rarely isolated and usually depend on one another. Symbolization of a language fragment therefore occurs in the following steps:

1. Symbolize the identifier of syntax declarations in the language fragment. Note that we only symbolize declarations, and not extensions to generate a single unique name for a syntax declaration that is shared among fragments.

2. Symbolize the type alias definitions. This must take place after the previous step since the type aliases might make reference to syntax type definitions.

3. Symbolize the syntax constructors defined in the fragment. Again, this can only take place after the previous step as the type constructors might include newly defined type aliases.

4. Symbolize the names of newly defined semantic functions.

5. Symbolize the bodies of those same newly defined semantic functions. Since MLang supports mutual recursion among semantic functions, we must first perform the fourth step in which we generate names for all semantic functions before we can symbolize any function bodies.

Resolving `use`

The definition of a language fragment defines various names that are only available within that language fragment or when the fragment has been explicitly used through a `use` expression as in Listing 6.4. During symbolization, we create an environment storing each of the names defined in each language fragment. When a `use`-expression is encountered, we simply bring the names of that environment into scope and get rid of the `use`-expression.

```
lang MyLang
    type IntAlias = Int
    sem f : IntAlias -> IntAlias
    sem f =
    | x -> addi x 1
end
mexpr
let res : (use MyLang in IntAlias) = use MyLang in f 1
   in
utest res with 2 in ()
```

Listing 6.4: An example of an MLang program with language fragment namespacing.

6.2.4 Composition Checks

Language composition, the key abstraction in MLang, allows the programmer to compose various language fragments into a new and bigger language fragment. It is syntactically possible to compose any language fragments with each other, however the resulting, composed

fragment might not be well-defined. In order for the composition of languages to be valid (i.e. the result of the composition is well-defined and well-behaved), the following conditions must be met:

1. For each semantic function and syntax declaration, the base of the included declarations must be the same.

2. The number of arguments to a semantic function must be the same as that of the functions it includes. It is worth pointing out that the used string identifiers are allowed to differ as semantic function arguments are positional.

3. The number of type parameters that a syntax definition has must be the same as that of the declarations it includes.

4. For each semantic function, consider the full set of case patterns which also contains the patterns from included definitions. Each of the par of patterns in this set must either be disjoint or orderable by strict subset — See subsection 2.2.2.

Each of these conditions is checked by the compiler, specifically in `stdlib/mlang/composition-check.mc`.

6.2.5 MLang to MExpr Compilation

The Miking compiler does not compile MLang directly to the target language (typically OCaml). Instead, it first compiles MLang into MExpr which is then subsequently compiled to the desired target language. The compilation of MLang to MExpr historically occurred in `boot`, but it has been ported to `mi` as a part of this thesis work.

Since MLang is an extension of MExpr a large part of the translation is trivial. Many top-level declarations in MLang have a 1-to-1 counterpart in MExpr and can therefore be translated very directly (e.g. let bindings, recursive let bindings, type aliases). Only the compilation of language fragments poses any noteworthy problems. Recall that a language fragment contains type alias definitions, syntax definitions and extensions, and semantic function definitions and extensions. A type alias definition can trivially be compiled as they are also supported in MExpr.

An MLang syntax definition is compiled into a single, open-type definition. Each constructor for this type of definition is then compiled

into a single `con` definition. Concretely, the compilation pass creates a `type` definition when a syntax is declared and several `con` definitions whenever this syntax is extended.

An MLang semantic function is compiled into a combination of let-bindings, lambda abstractions, and match statements. Whilst the generation of the let-bindings and lambda abstractions are trivial and intuitive, we must consider the order of the patterns when we generate the match statement.

Recall that as a part of our language composition checks, we ensured that each pair of patterns in semantic functions was either disjoint or orderable by strict subset. When we compile MLang to MExpr, we create a directed graph based on this strict subset relation for the patterns in a semantic function. We then compute a *topological ordering* of this directed graph using Kahn's algorithm [42]. A topological ordering is an ordering such that for every directed edge (u, v) in our graph, u occurs before v in our ordering [43, Section 2.2.3]. Note that such an ordering is only guaranteed to exist since our composition checks ensure that this directed graph is acyclic. Based on this topological ordering, we produce a chain of match expressions ending in a fallback case to catch inexhaustive matches.

6.3 Summary

In this chapter, we have looked at various improvements to the Miking compiler infrastructure made as a part of this thesis work. Although these problems do not directly relate to the research problem tackled in this thesis, they were necessary to get the Miking compiler into a state in which we can extend MLang with additional constructs (e.g. product extension, co-syntax types, co-semantic functions). The changes made to the compiler relating to these MLang extensions are discussed in the next chapter.

Chapter 7

Implementation of MLang Extensions

This section discusses the implementation of the MLang extensions implemented in this thesis. These extensions include product extension, explicit sum extension, co-syntax types, co-semantic functions, and a type system supporting extensible records.

We first give an overview of the syntax that was added to MLang and MExpr. After this, we describe the compilation from MLang to MExpr with a focus on the compilation of the extensible sum and product types in MLang to their MExpr equivalents. The type checking of MExpr programs with extensible product and sum types is discussed next. This section also discusses the internals of our Constructor Type extension and its `Data` kind. Lastly, we discuss the monomorphisation transformation which transforms extensible MExpr records into monomorphic MExpr records.

7.1 Syntax

This section gives an overview of the syntax that was added to MCore to support product extension. The syntax is demonstrated through short programs. We opt to not provide a formal grammar since it would have to consider many irrelevant details (e.g. the grammar of types, expressions, patterns, etc.). Instead, we refer the reader to the grammar that is used by `menhir` [40] to parse MLang. It can be found in the artifact at `/boot/src/lib/parser.mly` [44].

```
lang L0
    syn Expr =
    | TmInt Int

    sem eval =
    | TmInt v -> v
end
lang L1 = L0
    syn Expr =
    | TmAdd (Expr, Expr)
    sem eval =
    | TmAdd (lhs, rhs) ->
        addi (eval lhs)
             (eval rhs)
end
```

```
lang L0
    syn Expr =
    | TmInt Int

    sem eval =
    | TmInt v -> v
end
lang L1 = L0
    syn Expr +=
    | TmAdd (Expr, Expr)
    sem eval +=
    | TmAdd (lhs, rhs) ->
        addi (eval lhs)
             (eval rhs)
end
```

Figure 7.1: On the left, we see a program using old MLang syntax in which the syntax for base declaration and sum extension is identical. On the right, we see the new syntax which differentiates between base declarations and sum extension.

7.1.1 MLang Syntax

Explicit Declaration and Sum Extension

On a semantic level, MLang distinguishes between *base declarations* and *extension declarations* of syntax and semantic declarations. A base declaration of a syntax type or semantic function is a standalone declaration that does not include behavior from any earlier declarations. A sum extension of a syntax type or semantic function extends an existing type with additional constructors or cases respectively.

In the old version of MLang, base and extension declarations share the same syntax. Since their behavior is different, especially w.r.t. the conditions for language composition discussed in subsection 6.2.4, we choose to distinguish between base declarations and sum extensions at a syntactic level. A base declaration is declared using the `=` symbol whilst sum extension uses a `+=` symbol. This is illustrated in Figure 7.1

Product Extension

To add support for language composition through product extension, we must introduce new syntax for this language feature. The syntax we have chosen is similar to that of sum extension: We indicate the extension with the *= symbol and list the extension for each constructor. We also allow the programmer to specify an extension that applies to each constructor. The new syntax for product extension is illustrated in Listing 7.1.

```
1  lang LC
2      syn Term =
3      | TmVar {ident : String}
4      | TmApp {lhs : Term, rhs : Term}
5      | TmAbs {ident : String, body : Term}
6  end
7  lang STLC = LC
8      syn Type = ...
9      syn Term *=
10     | TmAbs {ty : Type}
11 end
12 lang InfoSTLC = STLC
13     type Info = {line : Int, col : Int, file: String}
14     syn Term *= {info : Info}
15 end
```

Listing 7.1: An MLang program showcasing the product extension syntax in which we create a basic lambda calculus fragment, extend this to STLC, and then add info fields. On line 9-10, we extend the abstraction constructor with a type field and we extend all constructors with an info field on line 14.

Cosyn Definition and Extension

The syntax for co-syntax type declarations (or cosyns) allows the programmer to define extensible product types scoped and for these to be extended in future language fragments. The syntax is very similar to that of syntax types and an example can be found in Listing 7.2.

```
1  lang L0
2      cosyn SomeEnv = {x : Int, y : Int}
3  end L0
4  lang L1 = L0
5      cosyn SomeEnv *= {z : String}
```

```
6 end
```

Listing 7.2: An MLang program which showcases the syntax for declaring and extending cosyn types.

Cosem Definition and Extension

Co-semantic functions, or cosems, are the dual of semantic functions. They allow the programmers to define extensible construction of terms. The syntax has been chosen to highlight the duality between semantic and co-semantic functions.

We use symbols `=` and `*=` to denote the declaration and extension of a co-semantic function. Such a function is made up of cases, with a *co-pattern* on the left and its value on the right. In the current syntax, the only co-pattern allows you to specify the fields of a record that are set by this case. The programmer must also specify the extensible record type to which these fields belong to resolve any possible ambiguity.

An example can be found in Listing 7.3. One line 4, we declare a co-semantic function called `defaultEnv` that takes two arguments called `param` and `param2`. Each co-semantic function consists of any number of cases. The declaration of our function `defaultEnv` has exactly one case, found on lines 5 and 6. On line 5, the co-pattern `{MyEnv of x, y}` indicates that this co-semantic function will construct a term of type `MyEnv` and that this specific case will compute values for the fields `x` and `y` of this type. The body of this case is provided on the right-hand side of the `<-` on line 6. The body will always be a record expression that maps the labels specified by the co-pattern, in this case `x` and `y`. On lines 12-14, we see the extension of this co-semantic function. On line 19, we see that we can call our co-semantic function like any other function.

```
1  lang CosemExample
2    cosyn MyEnv = {x : Int, y : Int}
3
4    cosem defaultEnv param param2 =
5    | {MyEnv of x, y} <-
6      {x = subi 100 param, y = param}
7  end
8
9  lang SomeExtensionLang = CosemExample
10   cosyn MyEnv *= {abcd : String}
11
```

```
12   cosem defaultEnv param param2 *=
13   | {MyEnv of abcd} <-
14     {abcd = "something else"}
15 end
16
17 mexpr
18 use SomeExtensionLang in
19 let env = defaultEnv 10 20 in
20 utest env.x with 90 using eqi in
21 utest env.y with 10 using eqi in
22 utest env.abcd with "abcd" using stringEq in
23 ()
```

Listing 7.3: The syntax of a basic co-semantic function in MLang.

Type Annotations

We extend the syntax of both MLang and MExpr to facilitate type annotations that reason about our newly added extensible product type and our existing extensible sum types. At the heart of these type annotations lie the quantification of type variables of a certain kind, i.e., bounded quantitation over kinds. These kinds allow us to specify an upper and lower bound on the constructors and fields in a term for extensible sum and product types respectively.

An example of this syntax can be found in Listing 7.4 on lines 9-11. We define the type variable `m` to range over all types of kind `{Expr [< TmInt TmAdd TmVar]}`, i.e., those types in which *at most* (indicated by `<`) the constructors `TmInt`, `TmAdd` and `TmVar` are present. The type variable `n` is defined to range over all types in which *at least* (indicated by `>`) the field `varVals` is defined on our extensible type `EvalCtx`. The resulting type is clearly polymorphic.

In previous versions of MLang, it was possible to provide these kinds of type annotations for extensible sum types (e.g. `Expr{m}`). The ability to provide upper and lower bounds on the fields of an extensible product type was added in this thesis (e.g. `EvalCtx{n}`, `n :: {EvalCtx [> varVals]}`|).

```
1  lang L0
2      cosyn EvalCtx = {varVals : Map String Int}
3
4      syn Expr =
5      | TmInt Int
6      | TmAdd (Expr, Expr)
7      | TmVar String
8
9      sem eval : all m :: {Expr [< TmInt TmAdd TmVar]}.
10                all n :: {EvalCtx [> varVals]}.
11                EvalCtx{n} -> Expr{m} -> Int
12     sem eval ctx =
13     | TmInt i -> i
14     | TmAdd (lhs, rhs) ->
15         addi (eval ctx lhs) (eval ctx rhs)
16     | TmVar id -> match mapLookup id ctx.varVals
17                   with Some v then v
18                   else error "..."
19 end
```

Listing 7.4: An MLang program showing how we can use type annotations on extensible product types.

The fact that these type annotations refer to individual fields and constructors gives them great expressivity. However, this does come at a cost: In large programs, types might have dozens of constructors and fields leading to overly verbose type annotations. To combat this, we extend MLang with syntactic sugar over type annotations. Programmers can write `< <Lang>::<Type>` or `> <Lang>::<Type>`. E.g. `< L0::Type` is syntactic sugar for any type which is *more restrictive* than `Type` in `L0`. Conversely, `> L0::Type` denotes any type that is less restrictive. In our example from Listing 7.4, we could replace the type annotation on lines 9-11 with `sem eval : < L0::Expr -> < L0::EvalCtx -> Int`.

7.1.2 MExpr Syntax

Definition of Extensible Records in MLang

We extend the MLang syntax to support the definition of extensible record types and the definition of fields belonging to these types. The syntax is designed to dual the existing syntax for the creation of open

sum types through the `type` expressions and `con` expressions. This new extensible record syntax is not designed to be used directly, but to serve as a compilation target for the extensible product types of MLang. The syntax can be found in Listing 7.5.

Record Operations

Extensible records support four primary operations: Creation of an extensible record, projection of a field, updating an existing field, and extending a type. We re-use the syntax of standard records in MCore for projection and updating. The re-use of record creation syntax is not possible because it would be ambiguous whether the user meant to create a normal record or an extensible record. Instead, we introduce a syntax in which the creation of an extensible record is tagged with the type being created. To differentiate between updating and extending an extensible record, we introduce the `extend` keyword. An example program with this new syntax can be found in Listing 7.5.

```
mexpr
rectype SomeEnv in
recfield x : all m. SomeEnv -> Int in
recfield y : all m. SomeEnv -> Int in
let rec = {SomeEnv of x = 1} in
let rec = {extend rec with y = 2} in
utest addi rec.x rec.y with 3 in ()
```

Listing 7.5: A small MExpr program showcasing the new syntax for definition, creation and extension of extensible recorsd types.

7.2 Compilation from MLang to MExpr

This section gives a high-level overview of the compilation of the added MLang constructs to MExpr.

7.2.1 Compilation of `cosyn` definitions

Recall that a `cosyn` in MLang facilitates the definition of an *extensible product type* that is scoped to a specific language. Our newly added extensible record types in MExpr in the form of `rectype` and `recfield` expressions, make a suitable compilation target for `cosyn` definitions.

The declaration of a `cosyn` is compiled to a `rectype`-expression. Each of the fields that are added to the `cosyn` either through a declaration or a product extension can then simply be converted to `recfield`-expressions.

Recall that our type system enforces the conciseness of type annotations through homogeneity. As such, any extensible type must take the same shape throughout all fields in a `cosyn` definition. This homogeneity is implicit in the MLang definition. To ensure this homogeneity in the compiled MExpr, we generate a type variable that is propagated to all extensible product types. Two examples of `cosyn` compilation can be found in Figure 7.2 and Figure 7.3.

Let's first look at our first example in Figure 7.2. The declaration of a co-syntax type called `SomeEnv` on line 2 in MLang gets translated to the `rectype` expression on line 1 in MExpr. This `cosyn` declaration in MLang defines two fields (line 3). Each of these fields gets converted into its own `recfield` expression in MExpr (lines 2-5). The extension of this co-syntax type in MLang on lines 6 and 7 adds a single field to this type. This field is added in MExpr on lines 6-7 through a `recfield` expression.

In our second example (see Figure 7.3), we have a dependency of one co-syntax type on another. Just as before, the declaration of our types in MLang again results in `rectype` expressions in MExpr (lines 2-3). The compilation of the `varEnv` results in a `recfield` expression on lines 4-5. Since the types of the fields `currentEnv` and `namespaces` depend on a co-syntax type, we must ensure homogeneity. We do this through an explicit recursion variable `m` on lines 6 and 8. This recursion variable is implicitly present on the left-hand side of the `->` symbol on lines 7 and 9 (i.e. `SymEnv{m}`). This recursion variable ensures that any type of occurrences of `NameEnv` in our `SymEnv` are parameterized on the same recursion variable, leading to *homogeneity*.

7.2.2 Compilation of `syn` definitions

A `syn` type in MLang allows the programmer to specify *a sum type over product types*. Furthermore, this type is extensible in both the sum and product dimensions. The result of compiling a `syn` type to MExpr will be a combination of `type` and `con` expressions to model the sum type and of `rectype` and `recfield` expressions to handle the product extension.

When a `syn` type is declared, a `type` expression is generated. When a constructor is defined in the base declaration or through sum extension,

```
lang L0
    cosyn SomeEnv =
    {x : Int, y : Int}
end L0
lang L1 = L0
    cosyn SomeEnv *=
    {z : String}
end

```

```
rectype SomeEnv in
recfield x : all m.
    SomeEnv -> Int in
recfield y : all m.
    SomeEnv -> Int in
recfield z : all m.
    SomeEnv -> String in
()

```

Figure 7.2: On the left, we see an MLang program which defines and extends a simply `cosyn`. On the right, we see the result of compiling this MLang to MExpr.

```
lang L0
    cosyn NameEnv = {varEnv : Map String Name}

    cosyn SymEnv = {currentEnv : NameEnv,
                    namespaces : Map String NameEnv}
end L0

```

```
mexpr
rectype NameEnv in
rectype SymEnv in
recfield varEnv : all m.
    NameEnv -> Map String Name
recfield currentEnv : all m.
    SymEnv -> NameEnv{m}  in
recfield namespaces : all m.
    SymEnv -> Map String (NameEnv{m}) in
()

```

Figure 7.3: On top, we see an MLang program which defines two environments for a symbolization pass. Note how the environment `SymEnv` contains `NameEnv` in both of its fields. Below, we see the resulting MExpr program. The homogeneity of the occurrences of `NameEnv` within `SymEnv` is ensured through the introduction of the type variable `m` which is propagated to both occurrences.

we create an extensible record type for its payload (i.e. the type associated with the constructor) by creating a `rectype` expression. The identifier of this type is the identifier of the constructor with the suffix `"Type"`. We then generate a `con` expression which takes the extensible record for this constructor and maps it to the extensible sum-type. Any fields added to the payload of the constructor through its declaration or through product extension simply get turned into `recfield` expressions.

Similarly to `cosyn` compilation, homogeneity is ensured by introducing a type parameter that is automatically propagated to all extensible types. An example of the compilation of a `syn` type can be found in Figure 7.4.

The declaration of a syntax type called `Term` on line 2 leads to a `type` expressions on line 3 in MExpr. For each constructor that is added to this syntax-type, we do three things. Firstly, we generate an extensible product type for its payload. This is done for the constructors `TmVar`, `TmAbs`, and `TmApp` one lines 5, 7, and 10 in MExpr respectively. For each of the fields in the payload types, we define a field for the relevant extensible product type. This is done on lines 6, 8-9, and 11-12. Lastly, we define the constructors for our new type through `con` expressions on lines 14-16. The compilation of our syntax type `Type` on lines 9-10 happens in the same way. The product extension on line 13 results in the definition of another field on the relevant extensible product type on line 25.

7.2.3 Compilation of `cosem` definitions

Co-semantic functions provide an extensible construction mechanism for extensible product types. The behavior of a co-semantic function differs based on the used language fragment. Recall our example in Listing 7.3. In this example, the co-semantic function `defaultEnv` behaves differently in the fragments `CosemExample` and `SomeExtensionLang`. We essentially have different *instances* of the co-semantic function. Because these different instances have different behaviors, we generate a unique function for each instance of a co-semantic when we compile to MExpr.

The compilation of a co-semantic function can roughly be divided into two steps. In the first, we determine which cases belong to a specific instance of a co-semantic function by looking transitively at the included language fragments. Once we know which cases are included, we compute the values for each of the cases. Note that the order in which

```
lang LC
    syn Term =
    | TmVar {ident : String}
    | TmAbs {ident : String, body : Term}
    | TmApp {lhs : Term, rhs : Term}
end

lang STLC = LC
    syn Type =
    | TyArrow {lhs : Type, rhs : Type}

    syn Term *=
    | TmAbs {tyAnnot : Type}
end

```

```
mexpr
-- Compilation of L0
type Term m in

rectype TmVarType in
recfield ident : all m. TmVarType{m} -> String
rectype TmAbsType
recfield ident : all m. TmAbsType{m} -> String
recfield body : all m. TmAbsType{m} -> Term{m}
rectype TmAppType in
recfield lhs : all m. TmAppType{m} -> Term{m}
recfield rhs : all m. TmAppType{m} -> Term{m}

con TmVar : all m. TmVarType{m} -> Term{m}
con TmAbs : all m. TmAbsType{m} -> Term{m}
con TmApp : all m. TmAppType{m} -> Term{m}

-- Compilation of L1
type Type in
rectype TyArrowType in
recfield lhs : all m. TyArrowType{m} -> Type{m}
recfield rhs : all m. TyArrowType{m} -> Type{m}
con TyArrow : all m. TyArrowType{m} -> Type{m}

recfield tyAnnot : all m. TmAbsType{m} -> Type{m}
```

Figure 7.4: The compilation of a language fragments in MLang (seen on top) to an MExpr program (below).

```
lang MyLang
  cosyn MyEnv = {x : Int, y : Int}

  cosem defaultEnv param param2 =
  | {MyEnv of x} <-
    {x = subi 100 param}
  | {MyEnv of y} <-
    {y = param2}
end
mexpr use MyLang in
let env = defaultEnv 10 20 in
utest addi env.x env.y with 110 in ()

```

```
let defaultEnv = lam param. lam param2.
    let r1 = {x = subi 100 param1} in
    let r2 = {y = param2} in
    {MyEnv of x = r1.x, y y = r2.y}
in
let env = defaultEnv 10 20 in
utest addi env.x env.y with 110 in ()

```

Figure 7.5: On top, we see an MLang program with a co-semantic function. The MExpr code that this co-semantic function is compiled to is found below. The compilation of the co-syntax type is omitted for the sake of brevity.

these values are computed is non-deterministic. The computed values are then grouped into a single record that is returned by the MExpr function modeling our co-semantic function. An example can be found in Figure 7.5.

7.2.4 Desugaring Type Annotations

Recall that we have added the syntax `< SomeLang::SomeType` and `> SomeLang::SomeType` to type annotations in MLang. This addition is purely syntactic sugar and can be expressed by the bounded quantification over kinds in MExpr. We briefly describe the transformation which replaces the syntactic sugar with equivalent type annotations.

The first complication that arises is that a type annotation such as `< MyLang::Type` should not only reason about the type on the right-hand side but also provide information about all extensible types that can occur within this type. To address this, for each extensible type, we compute the set of extensible types on which this type can depend. To compute these type dependencies, we first establish a dependency graph between extensible types. We construct this dependency graph by traversing over the field types or constructor payloads of each extensible type T_1 and add the edge (T_1, T_2) for any extensible type T_2 encountered during this traversal. Once this graph has been established, we can simply perform depth-first search to find all dependencies of a given type.

Secondly, we need a mapping for each language from extensible types to their fields and constructors. This is trivially established by traversing all languages from top to bottom and looking at the addition of newly defined fields and constructors to those that are transitively included in by included languages.

Once we have computed these type dependencies and a mapping for each language fragment from extensible types to fields or constructors, we are ready to desugar our type annotations. To do this, we construct a kind based on the annotation that provides a lower or upper bound for each of the type dependencies based on the fields and constructors defined in that language. We then introduce a type variable with this kind and replace the type annotation with the application of this type annotation to the type. An example can be found in 7.3.2.

```
lang BaseArith
  syn Expr =
  | TmInt {val : Int}
  | TmAdd {lhs : Expr, rhs : Expr}

  sem eval : < BaseArith::Expr -> Int
  sem eval =
  | TmInt t -> t.val
  | TmAdd t -> addi (eval t.lhs) (eval t.rhs)
end

```

```
lang BaseArith
  syn Expr =
  | TmInt {val : Int}
  | TmAdd {lhs : Expr, rhs : Expr}

  sem eval : all m :: {Expr [< TmInt TmAdd],
                        TmIntType [> val],
                        TmAddType[> lhs rhs] }.
           Expr{m} -> Int
  sem eval =
  | TmInt t -> t.val
  | TmAdd t -> addi (eval t.lhs) (eval t.rhs)
end

```

Figure 7.6: On top, you see an MLang program in which we have provided a type annotation using syntactic sugar. Below, we see the resulting type annotation after our desugaring transformation.

7.2.5 Type Annotations under Language Composition

The programmer can provide explicit type annotations for (co-)semantic functions and compose language fragments to combine the behavior of multiple functions in MLang. This raises the question of what happens to type annotations under composition. It is not possible to copy the type annotations from included functions when using the newly implemented constructor type extension as the type will likely change due to composition. The solution here is actually extremely simple: We do not automatically provide type annotations when composing languages

but let the type checker infer the type instead. The programmer does always have the option of providing a manual type annotation.

7.3 Type Checking

7.3.1 Dependency Graph

Before we can type-check a MExpr program, we first perform a traversal that computes the dependency graph encoding the dependencies between various extensible types. More precisely, this traversal establishes a directed graph in which there is a vertex for each extensible type and the edge (v, w) encodes that the extensible product type v depends on type w. By performing depth- or breadth-first search on this graph, we can compute the type dependencies of each product type. In addition to computing all type dependencies for a certain extensible product type, we also compute the type dependencies *per label*.

7.3.2 The `Data` kind

At the heart of the type checking operations on extensible sum and product types lies the `Data` kind. This kind consists of multiple entries. Each entry gives a lower and upper bound on the constructors and fields on an extensible lower and upper bound respectively. By taking advantage of the duality between sum and product types, one instance of the `Data` kind can reason about both sum and product types simultaneously. To get an intuition for these kinds, we will look at some basic examples.

Consider first the program in Listing 7.6. In this program, we have created a function that expects the fields `x` and `y` to be present on the extensible record type `Foo`. This is captured in the type annotation by creating a type variable of the kind `{Foo [> x y]}`, indicating that *at least* these fields must be present. The program also shows that the resulting function is polymorphic since we can apply this function to both `rXYZ` and `rXY`.

```
20 mexpr
21 rectype Foo in
22 recfield x : all m. Foo -> Int in
23 recfield y : all m. Foo -> Int in
24 recfield z : all m. Foo -> Int in
```

```
25
26 let addXY : all m :: {Foo [> x y]}.
27             Foo{m} -> Int =
28   lam r. addi r.x r.y in
29
30 let rXYZ = {Foo of x = 1, y = 2, z = 3} in
31 utest addXY rXYZ with 3 in
32
33 let rXY = {Foo of x = 20, y = 3} in
34 utest addXY rXY with 23 in ()
```

Listing 7.6: An MExpr which creates a polymorphic function on records and provides a type annotation through the `Data` kind.

We can also specify an upper bound on the fields of a record through the `Data` kind. An example of this can be found in Listing 7.7. The result of `f` has *at most* the fields `x`, `y` and `z` which is expressed as `{Foo [< x y z]}`.

```
35 mexpr
36 rectype Foo in
37 recfield x : all m. Foo -> Int in
38 recfield y : all m. Foo -> Int in
39 recfield z : all m. Foo -> Int in
40
41 let f : all m :: {Foo [< x y z]}.
42         Int -> Foo{m} =
43   lam arg. {Foo of x = 0, y = arg, z = subi 100 arg} in
44
45 let r = f 10 in
46 utest r.x with 0 in
47 utest r.y with 10 in
48 utest r.z with 90 in ()
```

Listing 7.7: An MExpr which creates a polymorphic function on records and provides a type annotation through the `Data` kind.

So far we've seen how to use the `Data` kind to provide type annotations for product types. As shown in Listing 7.8, we can use the exact same syntax when reasoning about extensible sum types. Notice how the duality between sum and product types presents itself in our type annotations: When we take a function that operates on records, we provide a *lower bound* (as in Listing 7.6), but when a function operates on variant types this become an *upper bound* (as in Listing 7.8).

```
mexpr
type Expr in
con TmInt : Int -> Expr in
con TmAdd : (Expr, Expr) -> Expr in

recursive let eval : all m :: {Expr [< TmInt TmAdd]}.
                     Expr{m} -> Int =
  lam expr.
    switch expr
      case TmInt val then val
      case TmAdd (l, r) then addi (eval l) (eval r)
    end
in
utest eval (TmAdd (TmInt 5, TmInt -1)) with 4 in ()
```

Listing 7.8: An MExpr which creates a polymorphic function on an extensible sum type and provides a type annotation through the `Data` kind.

It is also possible for a single `Data` kind to reason about extensible sum types and extensible product types at the same time. This is actually extremely prevalent in MLang because syntax types are extensible sum types over extensible product types. An example can be found in Listing 7.9.

```
lang ArithLang
  syn Expr =
  | TmInt {val : Int}
  | TmAdd {lhs: Expr, rhs: Expr}

  sem eval : all m::{Expr [< TmAdd TmInt],
                      TmIntType [> val],
                      TmAddType [> lhs rhs]}.
             Expr{m} -> Int
  sem eval =
  | TmInt t -> t.val
  | TmAdd t -> addi (eval t.lhs) (eval t.rhs)
end

mexpr use ArithLang in
let int_ = lam v. TmInt {TmIntType of val = v} in
utest eval (TmAdd {TmAddType of lhs = int_ 1,
```

```
80                                        rhs = int_ 2})
81 with 3 in ()
```

Listing 7.9: An MLang program which contains a `Data` kind that reasons about both sum and product types.

Homogeneity, Nominal Types and the Open World Assumption

The implementation of our extended type systems in MLang is a continuation of the work on Constructor Types (extensively discussed in section 3.2). This type system is built upon three fundamental concepts: Homogeneity, Nominal Typing, and the Open World Assumption.

Recall from Chapter 5 that the notion of *homogeneity on a type* is defined to mean that all occurrences of an extensible type throughout any type must be the same. The main advantage of homogeneous types is that they allow for compact type expressions at the cost of expressivity. Our type system for extensible records is *nominal*, i.e., names play a crucial role (see subsection 2.3.3). Two type expressions with different names can never be interchanged, even if they are structurally compatible.

The crucial difference between this implementation and many of the systems discussed in Chapter 3 is the fact that our sum and product types are *extensible*: The programmer is free at any point in the program to define additional fields or constructors. When we type check a term using such extensible types, the type system must take care to ensure that the assigned type is still valid even if the fields or constructors are added at a later stage. We refer to this as the *Open World Principle*.

Kind Inference

The mechanism that is central to the type inference implementation is *kind inference*. This mechanism can be seen as a collection of constraints on the kind that computes the most generic instance of a *Data* that covers all of the relevant constraints.

Consider the function `addXY` from the program in Listing 7.10 which projects the fields `x` and `y`. When the type checker looks at the first projection `r.x`, it infers that the type of `r` must be `Foo{m}` where `m :: {Foo [> x]}` and `m`. Note that the type variable `m` is generated by the type-checker and can have any unique name. Similarly, for the projection `r.y`, the type `Foo{n}` where `n :: {Foo [> y]}` is inferred. The type checker then attempts to unify `Foo{m}` and `Foo{n}` which

results in the unification of `m` and `n`. At this point, the *kind inference mechanism* takes over and realizes that the constraints of both `m` and `n` can be captured by the kind `{Foo [> x y]}`. The inferred type is therefore `Foo{o}` where `o :: {Foo [> x y]}`.

```
82 mexpr
83 lang L
84     cosyn Foo = {x : Int, y : Int}
85 end
86 mexpr use L in
87 let addXY = lam r : Foo. addi r.x r.y in
```

Listing 7.10: An example program in which the full type of `addXY` must be inferred.

The exact mechanics of kind inference can be summarized as follows. Consider the unification of two `Data` kinds, d_1 and d_2. Each kind is simply a partial function from extensible types T to lower and upper bounds of labels $\mathcal{L}$.

$$d_1, d_2 : T \hookrightarrow \mathcal{L} \times \mathcal{L}$$

Let $d_1 \sim d_2$ denote the type obtained by unifying d_1 with d_2. We can define this for each extensible type $t \in T$ by cases as follows:

$$(d_1 \sim d_2)(t) \stackrel{\text{def}}{=} \begin{cases} (l, u) & \text{if } d_1(t) = (l, u) \wedge d_2(t) = \mathsf{undef} \\ (l, u) & \text{if } d_2(t) = (l, u) \wedge d_1(t) = \mathsf{undef} \\ (l_1 \cup l_2, u_1 \cap u_2) & \text{if } d_1(t) = (l_1, u_1) \wedge d_2(t) = (l_2, u_2) \\ \mathsf{undef} & \text{otherwise} \end{cases}$$

In addition, we must ensure that the resulting bounds are compatible, i.e. the lower bound must always be a subset of the upper bound.

7.3.3 Type Checking Record Operations

The type checking of the four record operations (creation, projection, extension, and updating) happens according to the rules laid out in Table 5.3. However, the rules outlined in this table are in terms of presence indicators and mappings. In the actual type-checker we actually do not have presence variables. Instead, the data-kind takes the place of the presence indicators and mapping.

Consider the premise $\ell^\bullet \in M(t)$ rule for record projection, T-RecordProj. To check this constraint in the type checker, we

create a meta-variable with the appropriate kind, i.e. `{T [> l]}`. Subsequently, we unify the type of this type variable with the actual type of the expression, deferring the inference of the complete type to the aforementioned kind unification mechanism.

One noteworthy complication that arises has to do with our operation for record extension. Consider the third premise of the T-RecordExtension rule in Table 5.3 which looks at the type dependencies of all labels. This conflicts with our open-world assumption, because this assumption implies that these type dependencies can change at any point. The implementation of the type-checker does not fully adhere to this principle due to this.

7.4 Monomorphisation

The compilation of MLang to the typical MLang compilation targets is made more complicated by the presence of extensible records. Because of this, we get rid of the extensible records through a process of *monomorphisation*. In this monomorphisation transformation, we transform a program with extensible, polymorphic records into a program with traditional, closed records.

The core idea behind this transformation is to insert a default value for all absent fields. This is possible since all fields that can be present on a record of a certain type are exhaustively defined by all `recfield` declarations in a program.

Since the type system ensures that these absent fields are never accessed, these values will never be read. If they are accessed, this is a bug in the type system. This is reflected in the translation by choosing the `never` as the default term since the evaluation of the `never` term will cause the program to terminate. Furthermore, this term can take on any type during type checking.

Conveniently, MCore has a special `never` term, representing an expression that must *never* be evaluated. Attempting to evaluate such an expression will result in a run-time error. However, it is not possible to insert such `never` terms directly into our monomorphised records as these terms would then be evaluated causing the program to terminate. Because of this, we add a layer of indirection to all our fields. We insert the term `lam. never` for any field that is absent and `lam. v` for any field which is present with the value `v`.

```
1 mexpr
2 rectype SomeEnv in
3 recfield x : all m. SomeEnv -> Int in
4 recfield y : all m. SomeEnv -> Int in
5 let rec = {SomeEnv of x = 1} in
6 let rec = {extend rec with y = 2} in
7 utest addi rec.x rec.y with 3 in ()
```

```
1 mexpr
2 type SomeEnv = {x : Int, y : Int} in
3 let rec = {x = lam. 1, y = lam. never} in
4 let rec = {rec with y = lam. 2} in
5 utest addi (rec.x ()) (rec.y ()) with 3 in ()
```

Figure 7.7: A concrete example of the monomorphisation transformation. The lower program is the result of performing the monomorphisation transformation on the upper program.

A concrete example of this monomorphisation transformation can be found in Figure 7.7. In the upper listing, we see a program containing extensible records. Performing our monomorphisation transformation on this program results in the lower listing. We see that the `rectype` and `recfield` expressions on lines 2-4 in the upper listing are transformed into a single type alias definition on line 2 of the lower listing. The construction of a record on line 5 in the upper listing, is converted to the construction on line 3 of the lower listing. Notice how the value for `x` is wrapped in a lambda abstraction and the missing label `y` is mapped to the placeholder `lam. never`. The extension operation on line 6 in the upper program is transformed into a regular update on line 4 in the lower program in which we again wrap the value in a lambda abstraction. The projection operations on line 7 of the upper listing are still present in the lower listing on line 5. However, since the values are now wrapped in lambda abstractions, we apply unit values to the projection results to unwrap their values.

Chapter 8

Evaluation

This chapter evaluates the extensions of MLang that are discussed in the previous chapter. Additionally, the first section evaluates the updated MLang pipeline as discussed in Chapter 6. The next section discusses the evaluation of the implementation of the MLang extension from Chapter 7, including the new type system, based on the criteria outlined in Chapter 4. In the last section, we provide some brief instructions on how to reproduce the results from both sections.

The evaluation is done on two separate artifacts. The evaluation of the updated pipeline uses the artifact which we call the *pipeline artifact* [45]. The evaluation of the MLang language extensions in the second section is done using a different artifact called the *MLang extension artifact* [44].

8.1 Evaluation of Updated MLang Pipeline

We describe the evaluation of the updated MLang pipeline from Chapter 6 in this section. Recall that this pipeline allows the compilation of MLang files with minimal reliance on `boot`. It is important to note that this pipeline compiles the *original version* of MLang without any of the features added by this thesis (e.g. product extension, cosems, cosyns).

Since the MCore compiler is itself written in MLang, it can be used as a test suite. We choose a selection of non-trivial files from the standard library and do not evaluate the entire standard library as this would lead to unacceptably high run times for our test-suite. The evaluated files and the results of running them can be found in Table 8.1.

Filename	Compilation	Evaluation
`stdlib/option.mc`	✓	✓
`stdlib/option.mc`	✓	✓
`stdlib/char.mc`	✓	✓
`stdlib/seq.mc`	✓	✓
`stdlib/map.mc`	✓	✓
`stdlib/mexpr/symbolize.mc`	✓	✓

Table 8.1: Evaluation results of running the updated MLang pipeline on a selection MLang programs from the standard library. The files have been compiled using the `--mlang-pipeline` flag.

8.2 Evaluation of Extensible Record Types

In this section, we evaluate the implemented type system implementation by analyzing how it relates to each of the criteria outlined in section 4.2. We also evaluate the correctness of the implementation through a variety of test cases.

8.2.1 Correctness

We evaluate the correctness of the MLang compiler we have implemented that supports extensible record types, product extension, cosyns, and cosems. We use *correctness* as an all-encompassing term covering the entire compilation pipeline including parsing, type checking, name resolution, and code generation. The evaluation is done at the hands of two test suites containing many programs.

The first test suite contains correct MLang programs containing unit tests. For each of these programs, we check whether the program can be compiled and that the tests pass in the resulting executable. The results can be found in Table 8.2

To ensure that the implementation of the type checker is able to correctly detect ill-typed programs, the second test suite contains a collection of ill-typed MLang and MExpr programs. For each program, we check that the compilation fails during the type checking phase. In Table 8.3, you will find the results.

Filename	Compilation Success	Evaluation Success
`assign-ty.mc`	✓	✓
`cosem.mc`	✓	✓
`cosyn.mc`	✓	✓
`data-example-1.mc`	✓	✓
`data-example-2.mc`	✓	✓
`data-example-3.mc`	✓	✓
`data-example-4.mc`	✓	✓
`data-example-5.mc`	✓	✓
`data-example-6.mc`	✓	✓
`data-example-7.mc`	✓	✓
`dependent-cosyn.mc`	✓	✓
`extension.mc`	✓	✓
`extrec.mc`	✓	✓
`simple-sym.mc`	✓	✓
`smap-desugar.mc`	✗	?
`smap-workaround.mc`	✓	✓
`stlc.mc`	✓	✓
`contract-inferred.mc`	✓	✓
`sum-contraction.mc`	✓	✓
`sum-extension.mc`	✓	✓
`system-f.mc`	✓	✓
`update.mc`	✓	✓

Table 8.2: Evaluation results of the test suite for extensible records that tests whether well-typed test-cases programs can be compiled and that their tests pass. The "compilation" column denotes whether the compilation was successful and the "evaluation" column denotes whether the unit tests of the compiled file pass. All files can be found in the directory `test/extrec/` in the root-directory of the extended MLang artifact [44].

Filename	Type Error Detected
`illegal-extension.mc`	✓
`illegal-label.mc`	✓
`illegal-projection.mc`	✓
`illegal-update.mc`	✓
`inexhaustive-match-nested.mc`	✓
`inexhaustive-match-toplevel.mc`	✓
`underapprox.mc`	✓

Table 8.3: Evaluation results of the test suite for extensible records which tests that type errors are detected in ill-typed programs.All files can be found in the directory `test/extrec/ill-typed/` in the root-directory of the extended MLang artifact [44].

Use Case	Filename	Compilation and Evaluation
Sum Extension	`sum-extension.mc`	✓
Sum Contraction	`sum-contraction.mc`	✓
Smap and Open Types	`smap-desugar.mc`	✗
Product Extension	`assign-ty.mc`	✓
Extensible Product Types	`system-f.mc`	✓
Extensible Construction	`system-f.mc`	✓

Table 8.4: Evaluation results of MLang programs based on the use cases outlined in section 4.1. The third column denotes whether compilation was successful and the fourth column indicates if all tests were passed.All files can be found in the directory `test/extrec/` in the root-directory of the extended MLang artifact [44].

8.2.2 Expressivity

We evaluate the expressivity of the implementation by implementing, compiling, and running each of the use cases outlined in section 4.1. The results can be found in Table 8.4. The results clearly show that the type system is expressive enough for all but one of the outlined use cases: The `smap` use case. We elaborate extensively on the reasons for this in Chapter 9.

8.2.3 Type Inference

One of the design goals of the type system was to support type inference. Although most of the type information can successfully be inferred, there is one complication that arises due to the choice to make our extensible records *nominal*.

Recall from subsection 2.3.3 that names play a central role in nominal type systems. Consider the extensible record types `Foo` and `Bar` from Listing 8.1 and the expression `{x = 1}`. What should be the type of the record under these definitions? The type of this record construction expression is actually *ambiguous*: It can be a normal record of type `{x : Int}`, the type of kind `{Foo [< x]}`, or the type of kind `{Bar [< x]}`. Due this ambiguity, the type checker can not assign a type to such expressions.

We place the responsibility of disambiguation on the programmer by forcing them to provide the identifier of the desired extensible record type to be created, e.g., `{Foo of x = 1}`. If no identifier is provided, a standard, monomorphic record is created.

```
88 mexpr
89 rectype Foo in
90 recfield x : all m. Foo -> Int in
91
92 rectype Bar in
93 recfield x : all m. Foo -> Int in
94 ()
```

Listing 8.1: An MExpr program in which two extensible types `Foo` and `Bar` are defined that both have a label `x` of type `Int`

Similar problems arise when type-checking record update, extension, and projection expressions. In these cases, it is often possible for the type checker to infer whether a monomorphic record or which extensible product type is used. If the type checker can not infer this information, a nominal type is used as a default fallback. In such cases, the programmer can provide explicit type annotations to indicate the desired extensible record type.

Outside of this ambiguity arising from the nominal aspect of our type system, the programmer can rely on the type checker to infer the types. This has been evaluated in our test suites by including various non-trivial test cases without type annotations.

8.2.4 MExpr Compatibility

The requirement of MExpr compatibility has been well met by re-using large chunks of the original type checker and implementing the compilation of extensible records through monomorphisation.

The type checking of extensible records relies heavily on the existing MExpr checker. The `Data` kind which underpinned the existing MExpr type checking for extensible sum types in the existing MExpr type-checker. By taking advantage of the duality between product and sum types, we were able to re-use this kind and all of its logic related to unification and kind inference.

As discussed in section 7.4, the extensible records in MExpr are converted to standard monomorphic records. Because of this, the compilation phases such as pattern lowering, constant folding and OCaml code generation can remain untouched.

8.2.5 Conciseness

We evaluate the conciseness of the type annotations by looking at two criteria: The size of the inferred types and the size of programmer-provided type annotations through syntactic sugar. We evaluate both of these criteria for the programs we have written based on the use cases outlined in section 4.1. For each of the use cases, we have written a program with manually written type annotations to evaluate the size of those. We have taken those same programs and removed these type annotations and used the `typeof` function to look at the inferred type on the important semantic functions.

The manually written type and inferred type annotation for the sum extension program can be found in Figure 8.1. The manually written type annotation is very compact and succinctly captures the type of the `eval` transformation. The inferred type is quite a bit larger but still manageable. When we look at a more complicated program, namely `assign-ty.mc` in Listing B.1, we see that the manually written type annotations remain concise but that the inferred type becomes rather complicated. The results for other programs can be found in Appendix B.

8.2.6 Understandability

The notion of understandability is rather subjective and hard to evaluate. We will attempt to gain some insight by looking into two important

```
lang ArithLang
    sem eval : < ArithLang::Expr -> Int
end

lang ConditionalLang
    sem eval : < ConditionalLang::Expr -> Int
end

```

```
ArithLang::eval : Expr{_r} _r -> Int where
  r :: {Expr[< TmInt TmAdd],
           TmIntType[> val],
           TmAddType[> lhs rhs]}
ConditionalLang::eval : Expr{_r} _r -> Int where
  r :: {Expr[< TmInt TmAdd TmIfThenElse],
           TmIntType[> val],
           TmAddType[> lhs rhs],
           TmIfThenElseType[> els, cond, thn]}

```

Figure 8.1: The program below shows the inferred type for a program containing sum-extension. Above, you see programmer-provided type annotations based on the syntactic sugar.

aspects of the type system: The type annotations themselves and the error reporting of type errors.

We have seen the inferred and manually-written type annotations in the previous section. The syntactic sugar for type annotations stay compact and intuitive, but we see that the inferred type annotations become rather large when the size of our program increases (See Appendix B).

```
lang LC
  syn Ty =

  syn Term =
  | TmApp {lhs : Term, rhs : Term}
  | TmAbs {ident : String, body : Term}
  | TmVar {ident : String}
end

lang IntArith = LC
  syn Term +=
  | TmAdd {lhs : Term, rhs : Term}
  | TmInt {val : Int}
end

lang STLC = LC + IntArith
  syn Ty +=
  | TyArrow {lhs : Ty, rhs : Ty}
  | TyInt {dummy : ()}

  syn Term *=
  | TmAbs {tyAnnot : Ty}

  sem typeCheck env =
  | TmVar t -> getFromEnv t.ident env
  | TmAbs t ->
    TyArrow {TyArrowType of lhs = t.tyAnnot,
        rhs = typeCheck
            (cons (t.ident, t.tyAnnot) env)
            t.body}
  | TmApp t ->
    match typeCheck env t.lhs with TyArrow inner then
      match inner with {lhs = lhs, rhs = rhs} in
      if eqType (lhs, (typeCheck env t.rhs)) then rhs
```

```
35        else error "..."
36      else error "..."
37    | TmInt _ -> TyInt {TyIntType of dummy = ()}
38    | TmAdd _ -> TyInt {TyIntType of dummy = ()}
39  end
40
41  mexpr
42  use STLC in
43  let add = TmAdd {TmAddType of
44      lhs = TmVar {TmVarType of ident = "x"},
45      rhs = TmInt {TmIntType of val = 1}} in
46  let add1 = TmAbs {TmAbsType of ident = "x",
47                                  body = add} in
48
49  typeCheck [] add1 ;
50  ()
```

Listing 8.2: An ill-typed program in MLang. The semantic function `typeCheck`, defined on line 24, requires the constructor `TmAbs` to contain a `tyAnnot` field as it is projected on line 29. However, when we attempt to call this semantic function on line 49, the abstraction term created on lines 46-48 does not contain a `tyAnnot` field.

To look at the error reporting, we look at Listing 8.2. In this program, on line 49, we are attempting to run the semantic `typeCheck` on a term in which `TmAbs` does not contain the field `tyAnnot` which is required by the semantic function. The resulting error can be found in Listing 8.3. Although the resulting type error is rather large and contains a lot of information, it does immediately point the programmer in the right direction by highlighting that the `tyAnnot` field is the only difference.

```
ERROR <program-with-error.mc 93:13-93:17>:
* Expected an expression of type: Term{_x} _x
*    Found an expression of type: Term{_x1} _m
* where
*   x :: {Ty[> TyArrow TyInt],
          Term[> ],
          TmAppType[> lhs rhs],
          TmAbsType[> body ident tyAnnot],
          TmVarType[> ident],
          TyArrowType[| lhs rhs],
          TyIntType[< dummy]}
*   x1 :: {Term[> TmAbs]}
```

```
*   m :: {Term[> TmVar TmAdd TmInt],
          TmAbsType[< body ident],
          TmVarType[< ident],
          TmAddType[< lhs rhs],
          TmIntType[< val]}
* (errors: these constructors required by one kind
   but not allowed in the other: 'tyAnnot')
* When type checking the expression
typeCheck [] add1 ;
```

Listing 8.3: The type error that is given by type-checker when attempting to compile the program in Listing 8.2.

8.3 Reproduction

This section gives instructions on how to reproduce the results presented in this chapter.

8.3.1 Reproducing MLang Pipeline Evaluation

In order to reproduce the evaluation results of section 8.1, you must download the version of the MLang pipeline artifact [45]. Once downloaded, the results can be reproduced by running `make test-mlang-pipeline` in the root directory.

8.3.2 Reproducing Extensible Record Evaluation

To reproduce the evaluation results in section 8.2, you will need to download and install the MLang extension artifact [44]. Once installed, you will need to bootstrap the Miking project by running the following commands:

```
$ make boot
$ make install-boot
$ make build
$ make build-mi
```

Once you have installed Miking and successfully built the most recent version from the source code using the commands above, you can

reproduce the results using the bash script `test/extrec/main.sh`*. All test files can be found in the same directory (i.e. `test/extrec/`).

You can reproduce the results from Table 8.2 by running the `run-all` command for human-readable format or `all-csv` to get a CSV file. The test cases of Table 8.4 are a subset of the test cases run by this command. The results from Table 8.3 can be obtained by running `run-all-type` or `type-csv`. The conciseness results, found in Appendix B, can be reproduced by running the `type-log` command.

*This script is meant to be run from the root directory. Running it from any location might cause issues, e.g. `./test/extrec/main.sh run-all`

Chapter 9

Discussion

In this chapter, we discuss the limitations of the thesis work. More specifically, we reflect on the shortcomings of the developed type system and the evaluation methodology.

9.1 Type System

9.1.1 Over-Approximation and Under-Approximation

The type system for extensible records enforces *homogeneity*. Even if an expression is not homogeneous, the type system still treats it as such through *over-approximation* or *under-approximation*: When two expressions of a certain extensible type with incompatible bounds, we replace the bounds by the most or least restrictive bounds that encompass the original two bounds. Consider the program in Listing 9.1.

Due to over-approximation or under-approximation, situations can occur in which the programmer is forced to deal with certain cases which can not actually occur. For example, even if a certain constructor can only occur at the top level of an expression, the homogeneity constraints still force the programmer to consider this constructor at the other levels of this expression.

```
1 mexpr
2 rectype Foo in
3 recfield x : all m. Foo -> Int in
4 recfield y : all m. Foo -> Int in
5
6 rectype Bar in
```

```
7  recfield f1 : all m. Bar -> Foo{m} in
8  recfield f2 : all m. Bar -> Foo{m} in
9
10 let f : all m :: {Bar [< f1 f2], Foo [< x y]}.
11         Bar{m} -> Int
12 let fun = lam bar : Bar.
13   let f1 = bar.f1 in
14   let f2 = bar.f2 in
15   addi f1.x f2.y in
16
17 fun {Bar of f1 = {Foo of x = 1}, f2 = {Foo of y = -2}}
```

Listing 9.1: A program showcasing the overapproximation of required fields on the function `fun`. Even though it is easy to see that the application on line 17 is well-behaved, it is not allowed by the type system. This is because the homogeneity constraint forces both `x` and `y` to be present on all occurrences of `Foo`. Note that type annotation is given for the sake of clarity, but it equivalent to the inferred type.

9.1.2 Homogeneity and the Open World Assumption

Our type system for extensible records is built on two principles: Homogeneity and the Open World Assumption. Homogeneity states that all occurrences of an extensible type throughout a type expression must be the same. By the open world principle, we mean the fact that any extensible sum or product type can be extended at any point in the future and that previously computed types should remain valid. When laid side by side, it is easy to see how these two notions can interfere with each other. How can we ensure that all occurrences of some extensible type are equivalent if we have to consider fields or constructors yet to be defined? This fundamental issue is not solved in this work and is in fact responsible for the failing test cases in Table 8.2.

Problems surrounding `assign-ty`.
One of our test cases from Table 8.2 is a test case called `assign-ty`, based on subsection 4.1.4, in which we want to assign a `ty` field to each term in a DSL modeling the Simply Typed Lambda Calculus. Recall that the abstraction term in STLC always has a type annotation. The expected inferred type of the `assignTy` function that performs this transformation is that it takes a term where none of the terms have a `ty` field and

produces a term in which each term `ty` field. However, this is not the inferred type. Instead, the inferred type requires all input terms to already have a `ty` field.

This occurs due to the conflict between homogeneity and the open-world principle. On lines 94 and 104 in Listing C.1, we can see that the `tyAnnot` field on the `TmAbs` term is copied from the input to the output. This seemingly innocent operation is actually what causes this unexpected type to be inferred. This is because the open world principle forces the type checker to consider cases in which the `tyAnnot` is extended in such a way that it now contains a field of type `Term`. In other words, the inferred type of `tyAnnot` actually has a `Data` kind which reasons both about `Term` and `Ty`. As such, when this variable is inserted into the returned record, homogeneity requires that its type is equal to that of the other inserted terms.

To circumvent this problem, we have to strip away all of the extensions yet to be defined. This is done in `assign-ty-workaround` and can be found in Listing C.2. On line 101, by calling `stripTy`, we ensure that the resulting `tyAnnot` does not contain any `Terms` regardless of how it's extended. When we do this, the type inference mechanism is able to infer the expected type.

Problems surrounding `smap`.
In the evaluation, we've seen that transformations that have been written using `smap` can not be assigned their proper types. The reason here is again the conflict between the open world assumption and homogeneity. When we implement `smap`, we reason about a limited number of fields and constructors. However, the open-world principle forces us to consider any possible extensions too. Therefore the type checker can not actually change the extensible types within terms using `smap`.

9.1.3 Type Inference

As discussed in subsection 7.3.3, we rely on type inference to determine whether record operations should be treated as operations on standard monomorphic records or on extensible records. Recall that the type checker defaults to monomorphic records in situations in which the type is unknown. In situations where an extensible record is used, but the type inference mechanism is not powerful enough to realize this, the programmer is forced to assist the type checker by providing additional

type annotations, e.g., see Listing 9.2. In our test suite, we found that the programmer had to provide type annotations even in situations where it seems obvious. From our test suite, we can conclude that the weakness of our type inference mechanism places a needlessly high burden on the programmer to provide type annotations to disambiguate between extensible records and standard records.

```
1  lang L0
2    cosyn FooType = {a : Int}
3
4    cosyn BarType = {a : Int,
5                     foos : [Foo{m}]}
6
7    sem sum : Bar -> Int
8    sem sum =
9    | b ->
10     let foos = b.foos in
11     let as = map
12         (lam f : Foo. f.a)
13         foos in
14     addi (foldl addi 0 as) b.a
15 end
```

Listing 9.2: A program showcasing a situation in which type inference is not strong enough to detect all occurrences of extensible record types. The type annotation on lines 7 and 11 must be provided by the programmer if compilation is to be successful.

9.2 Test Suite and Backwards Comparability

One of the major limitations of the evaluation in section 8.2 is the size of the programs in our test suite. The largest program contains around 230 lines of code, whilst the size of realistic DSLs typically contains thousands of lines. Furthermore, the complexity of these test cases is also limited. They typically contain one or two program transformations whilst actual compilers contain dozens. We must therefore be hesitant in concluding that our extended MLang version is practical for large-scale DSL development, even though our initial results on small programs do look promising.

A keen observer might wonder why we do not evaluate our newly

extended MLang compiler by using it to compile itself. After all, the large parts of the MLang compiler are written in MLang itself. The reason that this is not as easy as it may first appear is that our new version of MLang is not backward compatible. I.e., various programs that could be compiled and executed using the old version of MLang are no longer valid MLang programs for a variety of reasons.

One of these reasons is the syntactical restriction that the payload is now always a record although such payloads can easily be rewritten to take the shape of a singleton record. A more fundamental reason is that our new language now performs exhaustiveness checks when applying semantic functions. It turns out that the type system often over-approximates the types, leading to the detection of spurious inexhaustive matches. Furthermore, shallow mapping is very prevalent in the MLang compiler and we've seen in the evaluation that the type system struggles to assign principal types to transformations written using `smap`.

Chapter 10

Conclusion

This final chapter provides a summary of the thesis work and an overview of future work.

10.1 Summary

This thesis studies the notion of language composition: The creation of programming languages in terms of a number of other languages and extensions. For the purposes of this work, language composition can be subdivided into two operations: *sum extension* and *product extension*. The former refers to the addition of *additional constructs* to a programming language whilst the latter refers to the extension of *existing constructs* with additional information.

In this work, we consider language composition in a programmatic context by studying and extending the MCore programming language. The MCore language is centered around the notion of *language fragments* which are modeled after the formal definitions of programming languages in programming language (PL) theory. A language fragment can contain two types of constructs: The definition of *syntax types* and *semantic functions*. Syntax types, inspired by the representations of abstract syntax in PL theory, are sum types over product types (or variants over records). Semantic functions, typically used to model the static and dynamic semantics in a formal setting, define programmatic transformations that deconstruct on the aforementioned syntax types through *pattern matching*.

Language fragments can be composed in MCore to create a new programming language. The old version of MCore supports language

composition through sum extension by allowing the programming to extend the variant in an existing syntax type. One issue that arises is that the result of language composition can become unsound due to the possibility of *inexhaustive matches* in semantic functions. To prevent this, previous work has developed a type system for MCore that is able to detect such errors, and as such, ensure the soundness of composition.

The major focus of this thesis lies on the extension of programmatic language composition with *product extension*. To this end, we extend the syntax and compilation of MCore programs. However, the extension of syntax types in the product dimension can result in unsound composition due to the possibility of introducing *illegal projections*. The thesis therefore extends the type system to prevent such illegal projections.

In addition to extending the notion of language composition to contain product extension of syntax types, we also introduce the notion of co-syntax types and co-semantic functions. The primary motivation for these new language constructs is to facilitate the definition of extensible environment types (see also section 4.1). As the name of these features implies, they are designed to dual the existing syntax types and semantic functions. Since a syntax type is an extensible sum type and the dual of a sum type is a product type, a co-syntax type is an extensible product type. Since semantic functions provide the extensible *deconstruction* of *syntax types* through *patterns*, its dual, the co-semantic function provides extensible *construction* of *co-syntax types* through *co-patterns*.

As discussed, this thesis extends the existing MCore type system with support for extensible record types. By leveraging the *duality between sum types and product types*, we are able to use a single mechanism to model both sum and product types. This mechanism can best be summarized over a form of bounded quantification over kinds in which each kind gives a lower and upper bound on relevant extensible types. More details are provided in Chapter 7.

This existing type system aims to keep type annotations concise through *homogeneity*: Each occurrence of an extensible type within a type expression is given the exact same type. We maintain the notion of *homogeneity* as we extend the type system to support product extension. Chapter 5 of this work studies the notion of homogeneity on extensible record types. We also introduce syntactic sugar over type annotations to facilitate the concise representation of expressive type annotations.

In short, this thesis has extended the MCore programming language with product extension, co-syntax definitions, co-semantic functions,

and concise type annotations. The addition of these constructs has been motivated by various real-world use cases that occur during the development of compilers and domain-specific languages. These are extensively discussed in Chapter 4. In order to extend MCore in this way, various changes needed to be made to the MCore compilation pipeline. We discuss these changes in Chapter 6. Additionally, this thesis has extended the existing type system to ensure the soundness of composition. We have studied the interplay between extensibility and homogeneity of records in Chapter 5. The implementation of the additional constructs and the type system is covered in Chapter 7.

We evaluated this newly implemented version of MCore in Chapter 8, mainly based on the aforementioned use cases. In Chapter 9, we discuss various limitations of our approach. Although we concluded that a large number of our use cases can be effectively covered, there are some limitations. The most important is the inherent conflict between the so-called *open world principle* and *homogeneity*. The former states that our type checking should happen under the assumption that any sum or product type may be extended at any point in the program. The latter states that our types should reason about all occurrences of a type at once. When laid side by side, the conflict between these becomes clear: How can we say that a certain function transforms *all* occurrences of some type, if we must operate under the assumption that any constructors or types can be extended at a later point? This remains an open problem.

10.2 Future Work

This section gives an overview of future work.

10.2.1 Improvements to the MLang Pipeline

In Chapter 6, we described the work done on the MLang pipeline in which we transferred a large part of the responsibilities of `mi` to `boot`. The end goal is to get rid of `boot` completely and to be able to bootstrap the entire compiler only using `mi`. Although the thesis work has made notable progress towards this goal, some work still remains.

The first remaining task is to implement an MCore parser in MCore in itself. Another concern is the size of the compiled programs using the new pipeline. Under language composition, large amounts of *unused* semantic functions are automatically generated and included in the resulting

OCaml back-end. This needs to be addressed by either eliminating dead code (i.e. these unused semantic functions) or, preferably, by only generating those definitions that are used in the first place.

In addition to the improvements to be made to the compiler, there are also some parts of the MCore language itself that need to be improved. One such improvement would be to improve the include-handling by introducing a proper notion of namespaces to solve the issues discussed in subsection 6.2.2.

10.2.2 Mechanization of Homogeneous, Polymorphic Records

Although we do state the critical theorems of progress and preservation in Chapter 5, they are stated as conjectures without proofs. A logical next step would be to actually prove these theorems. Although a pen-and-paper proof could provide be valuable, it would be preferable to create a mechanization in a tool such as Coq [33]. Such proofs could confirm or disprove our intuition that the type system provided in said chapter is actually sound.

10.2.3 Improved Co-Patterns for Co-Semantic Functions

In this work, we've added the construct of co-semantic functions to MLang to allow the extensible construction of (extensible) types. At its core, a co-semantic function allows the programmer to specify the construction of a value through co-patterns in an extensible matter.

The co-patterns supported in this work are however severely limited. In fact, we only have a single co-pattern that allows us to construct a record by parts by defining certain fields. Recall that our semantic functions have nested patterns not only for records but also for sequences, tuples, constructor applications, and more. It seems logical to extend our notion of co-semantic functions to support these same constructs but then for co-patterns.

10.2.4 Additional Syntactic Sugar for MLang Type Annotations

To allow the programmer to add concise type annotations to MLang programs, we've added a layer of syntactic sugar to our type annotation. This allows the programmer to specify a type by referring to a language fragment.

These sugared annotations do greatly limit the size of type annotations, but they can only be used if the transformation you are providing a type annotation to, corresponds naturally to a transformation from one fragment to another. If the structure of a transformation does not adhere to the structure of the fragments, these sugared type annotations can not be used.

One possible remedy for this would be to provide some basic algebra on top of these sugared type annotations. For instance, this would allow us to express a sum-contraction transformation as seen in Listing 10.1.

```
1  lang ArithLang
2      syn Expr =
3      | TmAdd {lhs : Expr, rhs : Expr}
4      | TmInt {val : Int}
5      | TmIncr {expr : Expr}
6
7      sem desugar : < ArithLang::Expr ->
8                    > (ArithLang::Expr - TmIncr)
9      sem desugar =
10     | TmInt t -> TmInt t
11     | TmAdd t ->
12         TmAdd  {TmAddType of lhs = desugar t.lhs,
13                               lhs = desugar t.lhs}
14     | TmIncr t ->
15         TmAdd {TmAddType of lhs = dugar t.expr,
16             rhs = TmInt {TmIntType of val = 1}}
17 end
```

Listing 10.1: A small program showcasing how a simple algebra over type annotations could be used to concisely express type annotations on line 7-8.

10.2.5 Generating `smap` Automatically

When developing domain-specific languages, it is typical to implement various versions of shallow mapping (or `smap`) for every syntax type you create. This process is both cumbersome and error-prone. Furthermore, such manual definitions might become "outdated" under product extension. Future work should investigate ways of automatically generating various `smap` definitions on-demand. This work could also investigate how such shallow maps should behave under product extension.

10.2.6 Composition of Types and Type Annotations on Semantic Functions

When multiple semantic functions are composed, the resulting semantic function is type-checked fully independently of its components. This is ultimately undesirable as the type checkers spend a lot of time type-checking multiple times. From a theoretical perspective, this is also elegant as it essentially violates the requirement in Wadler's expression problem that individual components must be type-checked individually (see section 3.1). Future work should investigate whether the type system on semantic functions can be changed in such a way that each individual case is type-checked only type-checked once and the types of semantic functions consisting of multiple cases are then computed through composition.

In the newly developed version of MLang type annotations belong to a single instance of a semantic function in a specific language fragment. When creating a new instance of a semantic function through composition, the type annotations of the components are completely ignored, although the programmer does have the option of manually providing a type annotation for the composed function. Future work should investigate whether the type annotations themselves could also somehow be composed.

10.2.7 Improvements to the Type System

In Chapter 8, we've seen that the new type system struggles to assign proper types to transformations written using shallow mapping functions (i.e. `smap`). Since these functions are very prevalent in idiomatic DSL development in Miking, this is a major roadblock to the adoption of the

newly developed type system into practical, real-world usage. Future work should therefore investigate how the type system could be altered to assign proper types to such transformations. One approach worth investigating is whether built-in shallow mapping functions with special type rules could be used.

We've also seen that the open-world assumption can conflict with our notion of homogeneity, especially when extending records. This limitation currently prevents us from assigning proper types to transformations that extend records, as seen in subsection 4.1.4. We determine whether it's possible to reconcile the conflicts between the open-world assumption and homogeneity such that these kinds of transformations can be assigned proper types. If this turns out to not be possible, we will likely need to choose between homogeneity and the open-world assumption. Since the open-world assumption is more fundamental to MLang, it seems likely that we would give up the property of homogeneity.

Appendix A

Complete Introduction Example

In this section, we give the source code for the complete interpreter that was developed in Chapter 1. The full source code can be found in Listing A.1 and under the file path `test/extrec/introduction.mc` in MLang extension artifact [44].

To run the example, you must first install the Miking framework. Instructions can be found at `https://miking.org/installation`. Once installed, download the MLang extension artifact. In the root-directory of the artifact, rebuild the compiler by running `$make build$` and `make build-mi`. Once this is done, you can compile the example by running `./build/mi-tmp compile --test --experimental-records test/extrec/introduction.mc`. This will produce an executable called `introduction` which you can then run.

```
let and = lam b1. lam b2.
  if b1 then b2 else false

recursive let eqString = lam s1. lam s2.
  match (s1, s2) with ([], []) then
    true
  else match (s1, s2) with ([h1] ++ t1, [h2] ++ t2) then
    and (eqc h1 h2) (eqString t1 t2)
  else
    false
end

lang Base
  syn Ty =

```

```
  syn Term =

  sem isValue =
  | _ -> false

  sem step =

  sem subst ident term =
end

lang LambdaCalculus = Base
  syn Term +=
  | TmApp {lhs : Term, rhs : Term}
  | TmAbs {ident : String, body : Term}
  | TmVar {ident : String}

  sem step +=
  | TmApp t ->
    match t.lhs with TmAbs t2 then
      subst t2.ident t2.body t.rhs
    else if isValue t.lhs then
      TmApp {TmAppType of lhs = t.lhs, rhs = step t.rhs}
    else
      TmApp {TmAppType of lhs = step t.lhs, rhs = t.rhs}
  | TmVar _ -> error "Stuck!"
  | TmAbs _ -> error "Stuck!"

  sem subst ident term +=
  | TmVar t -> if eqString ident t.ident then term
               else TmVar t
  | TmApp t -> TmApp {TmAppType of lhs = subst ident
   term t.lhs,
                                     rhs = subst ident
   term t.rhs}
  | TmAbs t -> if eqString ident t.ident then TmAbs t
   else
              TmAbs {TmAbsType of ident = t.ident,
                                  body = subst ident
   term t.body}

  sem isValue +=
  | TmAbs _ -> true
```

```
54 end
55
56 lang Arith = Base
57   syn Term +=
58   | TmInt {val : Int}
59   | TmAdd {lhs : Term, rhs : Term}
60
61   sem step +=
62   | TmAdd t ->
63     match (t.lhs, t.rhs) with (TmInt t1, TmInt t2) then
64       TmInt {TmIntType of val = addi t1.val t2.val}
65     else match t.lhs with TmInt _ then
66       TmAdd {TmAddType of lhs = t.lhs, rhs = step t.rhs}
67     else
68       TmAdd {TmAddType of lhs = step t.lhs, rhs = t.rhs}
69   | TmInt _ -> error "Stuck!"
70
71   sem isValue +=
72   | TmInt _ -> true
73 end
74
75 lang LambdaCalculusArith = LambdaCalculus + Arith
76   sem subst ident term +=
77   | TmInt t -> TmInt t
78   | TmAdd t -> TmAdd {TmAddType of lhs = subst ident
     term t.lhs,
79                                      rhs = subst ident
     term t.rhs}
80 end
81
82 lang STLC = LambdaCalculusArith
83
84   syn Ty +=
85   | TyArrow {lhs : Ty, rhs : Ty}
86   | TyInt {dummy : ()}
87
88   syn Term *=
89   | TmAbs {tyAnnot : Ty}
90
91   sem eqType =
92   | (TyInt _, TyInt _) -> true
93   | (TyArrow t1, TyArrow t2) -> and (eqType (t1.lhs, t2.
```

```
    lhs)) (eqType (t1.rhs, t2.rhs))
94  | _ -> false
95
96  sem getFromEnv ident =
97  | [(h, ty)] ++ t ->
98    if eqString h ident then
99      ty
100   else
101     getFromEnv ident t
102 | [] -> error "Ident not found in env!"
103
104 sem typeCheck env =
105 | TmVar t -> getFromEnv t.ident env
106 | TmAbs t -> TyArrow {TyArrowType of lhs = t.tyAnnot,
    rhs = typeCheck (cons (t.ident, t.tyAnnot) env) t.
    body}
107 | TmApp t ->
108   match typeCheck env t.lhs with TyArrow inner then
109     match inner with {lhs = lhs, rhs = rhs} in
110     if eqType (lhs, (typeCheck env t.rhs)) then rhs
111     else error "..."
112   else error "..."
113 | TmInt _ -> TyInt {TyIntType of dummy = ()}
114 | TmAdd _ -> TyInt {TyIntType of dummy = ()}
115 end
116
117 mexpr
118 use STLC in
119 let tyInt = TyInt {TyIntType of dummy = ()} in
120 let add = TmAdd {TmAddType of lhs = TmVar {TmVarType of
    ident = "x"},
121                                 rhs = TmInt {TmIntType of
    val = 1}} in
122 let add1 = TmAbs {TmAbsType of ident = "x", tyAnnot =
    tyInt, body = add} in
123
124 let actualTy = typeCheck [] add1 in
125 let expectedType = TyArrow {TyArrowType of lhs = tyInt,
    rhs = tyInt} in
126 utest actualTy with expectedType using (lam l. lam r.
    eqType (l, r)) in
127
```

```
128 let expr = TmAdd {TmAddType of lhs = TmInt {TmIntType of
        val = 2},
129                                 rhs = TmInt {TmIntType of
        val = 1}}  in
130 let result = step expr in
131 (match result with TmInt t then
132   (utest t.val with 3 in ())
133 else
134   error "Test failed, result is not int!");
135 ()
```

Listing A.1: The complete source code of the interpreter used as an example in Chapter 1.

Appendix B

More Conciseness Results

This section provides additional results for the evaluation of the conciseness criterion discussed in subsection 8.2.5.

```
=== test/extrec/typelog/assign-ty.mc ===
assign-ty : [([Char],
    Ty{_x} _x)] -> Term{_r} _r -> Term{_x} _x where
  x :: {Ty[> TyArrow TyInt],
    Term[| TmApp TmAbs TmVar TmAdd TmInt],
    TmAppType[< lhs rhs | ty],
    TmAbsType[< body ident tyAnnot | ty],
    TmVarType[< ident | ty],
    TmAddType[< lhs rhs | ty],
    TmIntType[< val | ty],
    TyArrowType[| lhs rhs],
    TyIntType[< dummy]}
  r :: {Ty[< TyArrow TyInt],
    Term[< TmApp TmAbs TmVar TmAdd TmInt],
    TmAppType[> lhs rhs],
    TmAbsType[> body ident tyAnnot],
    TmVarType[> ident],
    TmAddType[> lhs rhs],
    TmIntType[> val],
    TyArrowType[> lhs rhs]}
=== test/extrec/typelog/sum-contraction.mc ===
eval : < BaseArith::Expr -> Int where
  ss :: {Expr[< TmInt TmAdd],
    TmIntType[> val],
    TmAddType[> lhs rhs]}

```

```
desugar : < SugarArith::Expr -> > BaseArith::Expr where
  ss :: {Expr[> TmInt TmAdd],
    TmIntType[< val],
    TmAddType[< lhs rhs]}
  ss :: {Expr[> TmInt TmAdd],
    TmIntType[< val],TmAddType[< lhs rhs]}
  ss :: {Expr[< TmInt TmAdd TmIncr],
    TmIntType[> val],
    TmAddType[> lhs rhs],
    TmIncrType[> e]}

=== test/extrec/typelog/sum-extension.mc ===
ArithLang::eval : < ArithLang::Expr -> Int where
  ss :: {Expr[< TmInt TmAdd],
    TmIntType[> val],
    TmAddType[> lhs rhs]}
  ss :: {Expr[< TmInt TmAdd],
    TmIntType[> val],
    TmAddType[> lhs rhs]}
ConditionalLang::eval : < ConditionalLang::Expr -> Int
   where
  ss :: {Expr[< TmInt TmAdd TmIfThenElse],
    TmIntType[> val],
    TmAddType[> lhs rhs],
    TmIfThenElseType[> els thn cond]}
  ss :: {Expr[< TmInt TmAdd TmIfThenElse],
    TmIntType[> val],
    TmAddType[> lhs rhs],
    TmIfThenElseType[> els thn cond]}
=== test/extrec/typelog/system-f.mc ===
typeCheck : extrec {Env of _x} -> Term{_x} _x -> Ty{_x}
   _x where
  x :: {Ty[| TyArrow TyInt TyVar TyForall],
    Term[< TmApp TmAbs TmVar TmAdd TmInt TmTypeAbs
   TmTypeApp],
    TmAppType[> lhs rhs],
    TmAbsType[> body ident tyAnnot],
    TmVarType[> ident],
    Env[> tyvars varMap],
    TyArrowType[| lhs rhs],
    TyIntType[< dummy],
    TyVarType[> ident],
```

```
66      TyForallType[| ty ident],
67      TmTypeAbsType[> body ident],
68      TmTypeAppType[> lhs rhs]}
```

Listing B.1: The results of the conciseness evaluation. For each file, we have printed the inferred type of the *crucial* transformation in the example.

Appendix C

Additional Example Programs

Additional example programs are provided in this section.

```
include "stdlib/string.mc"

let and = lam b1. lam b2.
  if b1 then b2 else false

lang LC
  syn Ty =

  syn Term =
  | TmApp {lhs : Term, rhs : Term}
  | TmAbs {ident : String, body : Term, tyAnnot : Ty}
  | TmVar {ident : String}

  sem eval =
  | (TmVar _ | TmAbs _) & tm -> tm
  | TmApp outer ->
    match outer.lhs with TmAbs t then
      eval (subst t.ident outer.rhs t.body)
    else
      eval (TmApp {TmAppType of lhs = eval outer.lhs,
   rhs = outer.rhs})

  sem subst ident term =
  | TmVar t -> if eqString ident t.ident then term else
   TmVar t
  | TmApp t -> TmApp {TmAppType of lhs = subst ident
   term t.lhs,
```

```
                                        rhs = subst ident
    term t.rhs}
  | TmAbs t -> if eqString ident t.ident then TmAbs t
               else TmAbs {TmAbsType of ident = t.ident,
                                          body = subst
    ident term t.body}
end

lang IntArith = LC
  syn Term +=
  | TmAdd {lhs : Term, rhs : Term}
  | TmInt {val : Int}

  sem eval +=
  | TmInt t -> TmInt t
  | TmAdd t ->
    match eval t.lhs with TmInt l then
      match eval t.rhs with TmInt r then
        TmInt {TmIntType of val = addi l.val r.val}
      else
        error "..."
    else
      error "..."

  sem subst ident tm +=
  | TmInt t -> TmInt t
  | TmAdd t -> TmAdd {TmAddType of lhs = subst ident tm
    t.lhs,
                                    rhs = subst ident tm
    t.rhs}
end

lang LCArith = LC + IntArith
end

lang AssignTy = LCArith
  syn Ty +=
  | TyArrow {lhs : Ty, rhs : Ty}
  | TyInt {dummy : ()}

  syn Term *=
  | TmAbs {ty : Ty}
```

```
63    | TmAdd {ty : Ty}
64    | TmInt {ty : Ty}
65    | TmApp {ty : Ty}
66    | TmVar {ty : Ty}
67
68    sem tyTm =
69    | TmAbs t -> t.ty
70    | TmAdd t -> t.ty
71    | TmInt t -> t.ty
72    | TmApp t -> t.ty
73    | TmVar t -> t.ty
74
75
76    sem eqType =
77    | (TyInt _, TyInt _) -> true
78    | (TyArrow t1, TyArrow t2) -> and (eqType (t1.lhs, t2.
       lhs)) (eqType (t1.rhs, t2.rhs))
79    | _ -> false
80
81    sem getFromEnv ident =
82    | [(h, ty)] ++ t ->
83      if eqString h ident then
84        ty
85      else
86        getFromEnv ident t
87    | [] -> error "Ident not found in env!"
88
89    sem assignTy env =
90    | TmVar t ->
91      let foundType = getFromEnv t.ident env in
92      TmVar {TmVarType of ident = t.ident, ty = foundType}
93    | TmAbs t ->
94      let tyAnnot = t.tyAnnot in
95
96      let body = t.body in
97      let ident = t.ident in
98
99      let newEnv = cons (t.ident, tyAnnot) env in
100     let newBody = assignTy newEnv body in
101     let tyBody = tyTm newBody in
102
103     let ty = TyArrow {TyArrowType of lhs = tyAnnot, rhs
```

```
    = tyBody} in
104     TmAbs {TmAbsType of tyAnnot = tyAnnot,
105                         ty = ty,
106                         body = newBody,
107                         ident = ident}
108   | TmApp t ->
109     let lhs = t.lhs in
110     let rhs = t.rhs in
111
112     let newLhs = assignTy env lhs in
113     let newRhs = assignTy env rhs in
114
115     let leftTy = tyTm newLhs in
116     let rightTy = tyTm newRhs in
117
118     let ty = TyArrow {TyArrowType of lhs = leftTy, rhs =
    rightTy} in
119     TmApp {TmAppType of lhs = newLhs, rhs = newRhs, ty =
    ty}
120   | TmInt t ->
121     let ty = TyInt {TyIntType of dummy = ()} in
122     TmInt {TmIntType of ty = ty, val = t.val}
123   | TmAdd t ->
124     let lhs = t.lhs in
125     let rhs = t.rhs in
126
127     let newLhs = assignTy env lhs in
128     let newRhs = assignTy env rhs in
129
130     let leftTy = tyTm newLhs in
131     let rightTy = tyTm newRhs in
132
133     let tyInt = TyInt {TyIntType of dummy = ()} in
134
135     if and (eqType (tyInt, leftTy)) (eqType (tyInt,
    rightTy)) then
136       TmAdd {TmAddType of lhs = newLhs, rhs = newRhs, ty
     = tyInt}
137     else
138       error "LHS and RHS of TmAdd must be of integer
    type!"
139 end
```

```
140
141 mexpr
142 print "\n\n\nTESTS\n\n\n";
143 use AssignTy in
144 let tyInt = TyInt {TyIntType of dummy = ()} in
145 let one = TmInt {TmIntType of val = 1} in
146 let five = TmInt {TmIntType of val = 5} in
147
148 let typedOne = assignTy [] one in
149 let typedFive = assignTy [] five in
150 utest tyTm typedOne with tyInt using (lam l. lam r.
    eqType (l, r)) in
151 utest tyTm typedFive with tyInt using (lam l. lam r.
    eqType (l, r)) in
152
153
154 let addOneFive = TmAdd {TmAddType of lhs = five,
155                                      rhs = one} in
156 let typedAdd = assignTy [] addOneFive in
157 utest tyTm typedAdd with tyInt using (lam l. lam r.
    eqType (l, r)) in
158
159 let add = TmAdd {TmAddType of lhs = TmVar {TmVarType of
    ident = "x"},
160                                 rhs = TmInt {TmIntType of
    val = 1}} in
161 let add1 = TmAbs {TmAbsType of ident = "x", tyAnnot =
    tyInt, body = add} in
162
163 let add1Ty = assignTy [] add1 in
164 let actualTy = tyTm add1Ty in
165 let expectedType = TyArrow {TyArrowType of lhs = tyInt,
    rhs = tyInt} in
166 utest actualTy with expectedType using (lam l. lam r.
    eqType (l, r)) in
167
168 ()
```

Listing C.1: This program is ill-typed because of a conflict between homogeneity and the open world principle.

```
1 include "stdlib/string.mc"
```

```
2
3  let and = lam b1. lam b2.
4    if b1 then b2 else false
5
6  lang LC
7    syn Ty =
8
9    syn Term =
10   | TmApp {lhs : Term, rhs : Term}
11   | TmAbs {ident : String, body : Term, tyAnnot : Ty}
12   | TmVar {ident : String}
13
14   sem eval =
15   | (TmVar _ | TmAbs _) & tm -> tm
16   | TmApp outer ->
17     match outer.lhs with TmAbs t then
18       eval (subst t.ident outer.rhs t.body)
19     else
20       eval (TmApp {TmAppType of lhs = eval outer.lhs,
     rhs = outer.rhs})
21
22   sem subst ident term =
23   | TmVar t -> if eqString ident t.ident then term else
     TmVar t
24   | TmApp t -> TmApp {TmAppType of lhs = subst ident
     term t.lhs,
25                                       rhs = subst ident
     term t.rhs}
26   | TmAbs t -> if eqString ident t.ident then TmAbs t
27                else TmAbs {TmAbsType of ident = t.ident,
28                                         body = subst
     ident term t.body}
29 end
30
31 lang IntArith = LC
32   syn Term +=
33   | TmAdd {lhs : Term, rhs : Term}
34   | TmInt {val : Int}
35
36   sem eval +=
37   | TmInt t -> TmInt t
38   | TmAdd t ->
```

```
39      match eval t.lhs with TmInt l then
40        match eval t.rhs with TmInt r then
41          TmInt {TmIntType of val = addi l.val r.val}
42        else
43          error "..."
44      else
45        error "..."
46
47    sem subst ident tm +=
48    | TmInt t -> TmInt t
49    | TmAdd t -> TmAdd {TmAddType of lhs = subst ident tm
      t.lhs,
50                                      rhs = subst ident tm
      t.rhs}
51  end
52
53  lang LCArith = LC + IntArith
54  end
55
56  lang AssignTy = LCArith
57    syn Ty +=
58    | TyArrow {lhs : Ty, rhs : Ty}
59    | TyInt {dummy : ()}
60
61    syn Term *=
62    | TmAbs {ty : Ty}
63    | TmAdd {ty : Ty}
64    | TmInt {ty : Ty}
65    | TmApp {ty : Ty}
66    | TmVar {ty : Ty}
67
68    sem tyTm =
69    | TmAbs t -> t.ty
70    | TmAdd t -> t.ty
71    | TmInt t -> t.ty
72    | TmApp t -> t.ty
73    | TmVar t -> t.ty
74
75
76    sem eqType =
77    | (TyInt _, TyInt _) -> true
78    | (TyArrow t1, TyArrow t2) -> and (eqType (t1.lhs, t2.
```

```
   lhs)) (eqType (t1.rhs, t2.rhs))
79 | _ -> false
80
81 sem getFromEnv ident =
82 | [(h, ty)] ++ t ->
83   if eqString h ident then
84     ty
85   else
86     getFromEnv ident t
87 | [] -> error "Ident not found in env!"
88
89 sem stripTyAnnot =
90 | TyInt _ -> TyInt {TyIntType of dummy = ()}
91 | TyArrow t ->
92   let lhs = stripTyAnnot t.lhs in
93   let rhs = stripTyAnnot t.rhs in
94   TyArrow {TyArrowType of lhs = lhs, rhs = rhs}
95
96 sem assignTy env =
97 | TmVar t ->
98   let foundType = getFromEnv t.ident env in
99   TmVar {TmVarType of ident = t.ident, ty = foundType}
100 | TmAbs t ->
101   let tyAnnot = stripTyAnnot t.tyAnnot in
102
103   let body = t.body in
104   let ident = t.ident in
105
106   let newEnv = cons (t.ident, tyAnnot) env in
107   let newBody = assignTy newEnv body in
108   let tyBody = tyTm newBody in
109
110   let ty = TyArrow {TyArrowType of lhs = tyAnnot, rhs
    = tyBody} in
111   TmAbs {TmAbsType of tyAnnot = tyAnnot,
112                       ty = ty,
113                       body = newBody,
114                       ident = ident}
115 | TmApp t ->
116   let lhs = t.lhs in
117   let rhs = t.rhs in
118
```

```
119     let newLhs = assignTy env lhs in
120     let newRhs = assignTy env rhs in
121 
122     let leftTy = tyTm newLhs in
123     let rightTy = tyTm newRhs in
124 
125     let ty = TyArrow {TyArrowType of lhs = leftTy, rhs =
        rightTy} in
126     TmApp {TmAppType of lhs = newLhs, rhs = newRhs, ty =
        ty}
127   | TmInt t ->
128     let ty = TyInt {TyIntType of dummy = ()} in
129     TmInt {TmIntType of ty = ty, val = t.val}
130   | TmAdd t ->
131     let lhs = t.lhs in
132     let rhs = t.rhs in
133 
134     let newLhs = assignTy env lhs in
135     let newRhs = assignTy env rhs in
136 
137     let leftTy = tyTm newLhs in
138     let rightTy = tyTm newRhs in
139 
140     let tyInt = TyInt {TyIntType of dummy = ()} in
141 
142     if and (eqType (tyInt, leftTy)) (eqType (tyInt,
    rightTy)) then
143       TmAdd {TmAddType of lhs = newLhs, rhs = newRhs, ty
      = tyInt}
144     else
145       error "LHS and RHS of TmAdd must be of integer
    type!"
146 end
147 
148 mexpr
149 print "\n\n\nTESTS\n\n\n";
150 use AssignTy in
151 let tyInt = TyInt {TyIntType of dummy = ()} in
152 let one = TmInt {TmIntType of val = 1} in
153 let five = TmInt {TmIntType of val = 5} in
154 
155 let typedOne = assignTy [] one in
```

```
156 let typedFive = assignTy [] five in
157 utest tyTm typedOne with tyInt using (lam l. lam r.
       eqType (l, r)) in
158 utest tyTm typedFive with tyInt using (lam l. lam r.
       eqType (l, r)) in
159
160
161 let addOneFive = TmAdd {TmAddType of lhs = five,
162                                      rhs = one} in
163 let typedAdd = assignTy [] addOneFive in
164 utest tyTm typedAdd with tyInt using (lam l. lam r.
       eqType (l, r)) in
165
166 let add = TmAdd {TmAddType of lhs = TmVar {TmVarType of
       ident = "x"},
167                                rhs = TmInt {TmIntType of
       val = 1}} in
168 let add1 = TmAbs {TmAbsType of ident = "x", tyAnnot =
       tyInt, body = add} in
169
170 let add1Ty = assignTy [] add1 in
171 let actualTy = tyTm add1Ty in
172 let expectedType = TyArrow {TyArrowType of lhs = tyInt,
       rhs = tyInt} in
173 utest actualTy with expectedType using (lam l. lam r.
       eqType (l, r)) in
174
175 ()
```

Listing C.2: This program is well-typed due to the `stripTyAnnot` function.

Bibliography

[1] T. Kosar, P. E. Martínez López, P. A. Barrientos, and M. Mernik, "A preliminary study on various implementation approaches of domain-specific language," *Information and Software Technology*, vol. 50, no. 5, pp. 390–405, 2008. doi: https://doi.org/10.1016/j.infsof.2007.04.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584907000419 [Pages 1 and 15.]

[2] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM computing surveys (CSUR)*, vol. 37, no. 4, pp. 316–344, 2005. [Page 1.]

[3] D. Broman, "A vision of miking: Interactive programmatic modeling, sound language composition, and self-learning compilation," in *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2019. New York, NY, USA: Association for Computing Machinery, 2019. doi: 10.1145/3357766.3359531. ISBN 9781450369817 p. 55–60. [Online]. Available: https://doi.org/10.1145/3357766.3359531 [Pages 1, 5, 14, and 15.]

[4] S. Erdweg, P. G. Giarrusso, and T. Rendel, "Language composition untangled," in *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications*, 2012, pp. 1–8. [Pages 2 and 5.]

[5] G. Dodig-Crnkovic, "Scientific methods in computer science," in *Proceedings of the Conference for the Promotion of Research in IT at New Universities and at University Colleges in Sweden, Skövde, Suecia.* sn, 2002, pp. 126–130. [Page 11.]

[6] A. H. Eden, "Three paradigms of computer science," *Minds and machines*, vol. 17, pp. 135–167, 2007. [Page 11.]

[7] V. Senderov, J. Kudlicka, D. Lundén, V. Palmkvist, M. P. Braga, E. Granqvist, D. Broman, and F. Ronquist, "Treeppl: A universal probabilistic programming language for phylogenetics," *bioRxiv*, 2023. doi: 10.1101/2023.10.10.561673. [Online]. Available: https://www.biorxiv.org/content/early/2023/10/13/2023.10.10.561673 [Page 15.]

[8] D. Lundén, J. Öhman, J. Kudlicka, V. Senderov, F. Ronquist, and D. Broman, "Compiling universal probabilistic programming languages with efficient parallel sequential monte carlo inference," in *Programming Languages and Systems*, I. Sergey, Ed. Cham: Springer International Publishing, 2022. ISBN 978-3-030-99336-8 pp. 29–56. [Page 15.]

[9] L. Hummelgren, M. Becker, and D. Broman, "Real-time probabilistic programming," 2023. [Page 15.]

[10] P. Wadler and S. Blott, "How to make ad-hoc polymorphism less ad hoc," in *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1989, pp. 60–76. [Page 16.]

[11] "Miking Documentation | Miking — miking.org," https://miking.org/docs, [Accessed 01-10-2024]. [Page 16.]

[12] J.-Y. Girard, "Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur," Ph.D. dissertation, Éditeur inconnu, 1972. [Pages 18 and 27.]

[13] F. Emrich, S. Lindley, J. Stolarek, J. Cheney, and J. Coates, "Freezeml: Complete and easy type inference for first-class polymorphism," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 423–437. [Page 18.]

[14] B. C. Pierce, *Types and programming languages*. MIT press, 2002. [Pages 23, 24, 25, 27, 28, 29, and 34.]

[15] L. Damas and R. Milner, "Principal type-schemes for functional programs," in *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1982, pp. 207–212. [Page 25.]

[16] R. Milner, "A theory of type polymorphism in programming," *Journal of computer and system sciences*, vol. 17, no. 3, pp. 348–375, 1978. [Page 25.]

[17] L. Cardelli and P. Wegner, "On understanding types, data abstraction, and polymorphism," *ACM Computing Surveys (CSUR)*, vol. 17, no. 4, pp. 471–523, 1985. [Page 29.]

[18] B. C. Pierce, "Bounded quantification is undecidable," in *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1992, pp. 305–315. [Page 31.]

[19] M. Wand, "Type inference for simple objects," in *Proc., IEEE Symposium on Logic in Computer Science*, 1987, pp. 37–44. [Page 32.]

[20] D. Rémy, "Type checking records and variants in a natural extension of ml," in *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1989, pp. 77–88. [Pages 32, 60, and 63.]

[21] B. C. Pierce, *Advanced topics in types and programming languages*, 12 2004. [Online]. Available: https://doi.org/10.7551/mitpress/1104.001.0001 [Page 32.]

[22] W. Tang, D. Hillerström, J. McKinna, M. Steuwer, O. Dardha, R. Fu, and S. Lindley, "Structural subtyping as parametric polymorphism," *arXiv preprint arXiv:2304.08267*, 2023. [Page 32.]

[23] B. R. Gaster and M. P. Jones, "A polymorphic type system for extensible records and variants," Technical Report NOTTCS-TR-96-3, Department of Computer Science, University …, Tech. Rep., 1996. [Page 32.]

[24] M. P. Jones, "A theory of qualified types," in *European symposium on programming*. Springer, 1992, pp. 287–306. [Page 32.]

[25] D. Leijen, "Extensible records with scoped labels," in *Proceedings of the 2005 Symposium on Trends in Functional Programming (TFP'05), Tallin, Estonia*, 2005. [Pages 32 and 39.]

[26] ——, "Koka: Programming with row polymorphic effect types," *arXiv preprint arXiv:1406.2061*, 2014. [Page 32.]

[27] A. Rossberg, "Mutually iso-recursive subtyping," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA2, pp. 347–373, 2023. [Pages 35 and 41.]

[28] P. Wadler *et al.*, "The expression problem," *Posted on the Java Genericity mailing list*, vol. 7, 1998. [Page 36.]

[29] Y. Wang and B. C. d. S. Oliveira, "The expression problem, trivially!" in *Proceedings of the 15th International Conference on Modularity*, 2016, pp. 37–41. [Pages 36 and 37.]

[30] M. Torgersen, "The expression problem revisited: Four new solutions using generics," in *European Conference on Object-Oriented Programming*. Springer, 2004, pp. 123–146. [Page 37.]

[31] M. Blume, U. A. Acar, and W. Chae, "Extensible programming with first-class cases," in *Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, 2006, pp. 239–250. [Pages 37, 40, and 57.]

[32] W. Swiersta, "Data types à la carte," *Journal of Functional Programming*, vol. 18, no. 4, p. 423–436, 2008. doi: 10.1017/S0956796808006758 [Page 37.]

[33] C. Paulin-Mohring, "Inductive definitions in the system coq rules and properties," in *Typed Lambda Calculi and Applications*, M. Bezem and J. F. Groote, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993. ISBN 978-3-540-47586-6 pp. 328–345. [Pages 38 and 126.]

[34] A. Ohori, "A polymorphic record calculus and its compilation," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 17, no. 6, pp. 844–895, 1995. [Page 40.]

[35] J. G. Morris and J. McKinna, "Abstracting extensible data types: or, rows by any other name," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, jan 2019. doi: 10.1145/3290325. [Online]. Available: https://doi.org/10.1145/3290325 [Page 41.]

[36] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with webassembly," *SIGPLAN Not.*, vol. 52, no. 6,

p. 185–200, jun 2017. doi: 10.1145/3140587.3062363. [Online]. Available: https://doi.org/10.1145/3140587.3062363 [Page 41.]

[37] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler, "A history of haskell: being lazy with class," in *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, ser. HOPL III. New York, NY, USA: Association for Computing Machinery, 2007. doi: 10.1145/1238844.1238856. ISBN 9781595937667 p. 12–1–12–55. [Online]. Available: https://doi.org/10.1145/1238844.1238856 [Page 56.]

[38] "Ocaml," https://ocaml.org/, 2024. [Page 56.]

[39] P. Hudak and J. H. Fasel, "A gentle introduction to haskell," *ACM Sigplan Notices*, vol. 27, no. 5, pp. 1–52, 1992. [Page 56.]

[40] F. P. Y. Régis-Gianas, "Menhir reference manual," 2016. [Pages 76 and 86.]

[41] "Modules · OCaml Documentation — ocaml.org," https://ocaml.org/docs/modules, [Accessed 24-10-2024]. [Page 80.]

[42] A. B. Kahn, "Topological sorting of large networks," *Communications of the ACM*, vol. 5, no. 11, pp. 558–562, 1962. [Page 85.]

[43] D. E. Knuth, *The Art of Computer Programming: Fundamental Algorithms, Volume 1.* Addison-Wesley Professional, 1997. [Page 85.]

[44] M. Voorberg, "Extended MLang Compiler," https://github.com/marten-voorberg/miking/releases/tag/mlang-extension-v2, 2024, [Accessed 24-10-2024]. [Pages 86, 107, 109, 110, 116, and 130.]

[45] M. Voorberg, "Extended MLang Pipeline," https://github.com/marten-voorberg/miking/releases/tag/mlang-pipeline, 2024, [Accessed 03-10-2024]. [Pages 107 and 116.]