Concepts of Programming Language Design

Gabriele Keller
Liam O'Connor
Tom Smeding

Utrecht University

Contents

1 Introduction

Programming languages are essential tools for software engineers and computer scientists, and there are countless different languages in existence, with new languages appearing and others falling out of favour. We have not only general-purpose languages, such as C, C++, C#, Rust, Go, Java, JavaScript, Python, Scala, Haskell, and many others, but also a vast number of domain-specific ones. Some of them, like LaTeX, SQL, and PHP, are well-known, but many are used solely within a single company or organisation, for a very specific purpose.

As this landscape is constantly changing, it makes sense to study the theory of underlying concepts instead of just focusing on the current top ten most popular languages. Most of these concepts have been around for a long time, but in some cases, they are only just beginning to be utilised in mainstream languages. Understanding them not only facilitates learning new languages but also provides the basis for deciding what constitutes a good programming language for a specific purpose, what the trade-offs are, and how to design a programming language.

Moreover, to engage in discussions about program correctness, it is essential to possess a formal understanding of the semantics of a language, enabling comprehension of specifications for both formal verification and testing.

The first aspect of a language we typically learn is its syntax. Although the syntax of a programming language strongly influences its usability, it is not the primary focus of this course. We will cover it only to the extent that it is necessary for specifying a language. Instead, we are delving into the semantics, or the meaning, of different languages and language features. When examining the semantics of programming languages, we distinguish between static and dynamic semantics. Roughly speaking, static semantics encompasses anything that can be checked and analysed without actually executing a program, while dynamic semantics pertains to the execution behavior of a program. The demarcation of this line depends on the particular language. For example, for most languages, the scope of an identifier — meaning where in the program it can be used — is part of the static semantics. However, there are also languages where this is resolved at runtime.

1.1 The implementation of programming languages

Only programs written in machine language can be directly executed on a specific architecture. Programs composed in any other language, including assembly code, must either be compiled into machine language for execution or interpreted by another program.

Assembly language is closely related to machine code. Whereas the latter is in binary format, the former incorporates human-readable names for instructions and registers. Assembly language can be directly translated into machine language. In contrast to higher-level languages, both machine code can easily be converted back to assembly code. However, it's important to note that both machine code and assembly language are specific to a particular architecture. While assembly instructions are readable, comprehending their functionality and operation can be exceptionally challenging.

In this course, we focus on high-level languages designed with human programmers in mind. There are several ways to execute programs in high-level languages: An interpreter can process the statements or expressions from the source program and executes or evaluates them sequentially. Or, a compiler handles the entire program, analysing and optimising it, and subsequently generates machine code. The resulting code is contingent on the hardware architecture of the executing machine and the operating system (OS), as it contains calls to functions provided by the latter.

Increasingly, hybrid approaches are gaining popularity. In these cases, a compiler translates the source program into a more low-level intermediate code. This intermediate code is then interpreted. The advantage of this hybrid approach lies in the fact that all compiler checks and many optimisations remain viable, while the intermediate code retains its independence from specific architectures and operating systems.

For high-level languages, the implementation process consists of several steps and components, each playing a crucial role in translating human-readable code into machine-executable instructions.

Compilation steps

- **Lexical Analysis and Parsing:** Lexical analysis, often referred to as scanning, involves breaking down the source code into a series of tokens. These tokens are the smallest units of meaning in a programming language.

  Parsing takes these tokens and arranges them into a hierarchical structure, typically represented as an abstract syntax tree (AST). The AST captures the grammatical structure of the program.

- **Semantic Analysis:** This phase checks for semantic correctness, ensuring that the program adheres to the language's rules and constraints. It covers areas such as type checking, scope resolution, and name resolution.

- **Optimisation:** This phase aims to improve the performance and efficiency of the generated code. It involves a range of techniques, including constant folding, loop unrolling, and inlining of functions.

- **Code Generation:** Here, the compiler translates the optimized intermediate code into the target machine code. This process requires deep knowledge of the target architecture, including instruction set, memory management, and calling conventions. The target machine can be a virtual machine, which we discuss in Section 1.2.

- **Linking**: In cases where multiple source files are involved, the linker combines the generated machine code into a single executable. It resolves dependencies between different parts of the program.

- **Interpreter Design:** Interpreters operate by directly executing the source code. They often include components for lexical analysis, parsing, and an execution engine that interprets and executes the AST or bytecode directly.

Our focus in this course is on semantic aspects, so we mainly look into semantic analysis, but also touch on topics like error handling and garbage collection.

1.2 Abstract and virtual machines

Abstract and virtual machines are both imaginary machines, which provide an abstraction over concrete, existing hardware and operating systems. They are often characterised by the data types or values they support natively, their machine language, and sometimes their memory model. Although sometimes these terms are used interchangeably, they do differ in some important aspects.

An abstract machine is essentially a model of computation, and we use them in this course to analyse the properties or specify the semantics of a programming language. Generally, an abstract machine is characterised by the set of states it can be in, initial and final states, and rules which describe how the machine transitions from one step to the next. The state usually entails values the machine can represent, and a program or expression, which the machine executes or evaluates, respectively, as it transitions from state to state, until it reaches a final state. Many abstract machines can not be implemented faithfully, as their state may contain components like infinite memory, or for parallel models, unlimited parallelism.

The probably most well known abstract machine, the Turing machine [1], was invented by Alan Turing in 1936 to investigate the concept of computability, to show that the Entscheidungsproblem (decision problem) posed by Hilbert is not decidable. More recent examples for abstract machines, just to name a few, are the Algol Object Code [2], an abstract machine defining the semantics of Algol60, and the G-machine, for lazy functional languages [3]. If you are interested in a more comprehensive discussion, and more background on abstract machines, [4] presents a good overview.

A virtual machine can be viewed as an abstract machine with an implementation. Some are used to abstract over concrete hardware or operating systems, and some to serve as a common compilation target for a range of languages.

The Java Virtual Machine (JVM)[5] was initially designed to serve as compilation target for Java, but other languages, such as Scala, Kotlin and Clojure, and more recently also Go, OCaml and Haskell. A Java (or JVM compatible) compiler emits Java Bytecode, with is then interpreted by the interpreter of JVM, or compiled at runtime by a JIT (just in time) compiler to machine code. OpenJDK [6] is now the official reference implementation, but there are many others.

The .NET framework[7], developed by Microsoft provides a platform that spans both architecture and source language. At its core, the framework introduces the Virtual Execution System (VES), a virtual machine designed to execute code written in the Common Intermediate Language (CIL), also known as Microsoft Intermediate Language (MSIL). This intermediary language serves as an intermediate representation of the source code, allowing it to be executed on any system equipped with the .NET runtime environment.

The strength of the .NET framework lies in its ability to unify various programming languages under a common umbrella. Through adherence to the Common Language Specification (CLS), languages like C#, F#, VB.NET, and others can seamlessly interoperate within the .NET ecosystem. This interoperability extends to the use of libraries and components written in different languages, giving developers the choice to pick the language best suited to a certain task, while working on a unified platform.

Additionally, the .NET framework provides an extensive class library, known as the Base Class Library (BCL), which offers a rich set of pre-built components and functionalities. This library encompasses a wide range of features, from data manipulation to networking, enabling developers to accelerate the development process and build robust, feature-rich applications.

LLVM [8] is another framework which is designed to provide common infrastructure for multiple programming languages and abstract over the concrete hardware and operating system, and has been used to compile many languages, such as C, C++, Rust, and Haskell. The name LLVM used to stand for Low Level Virtual Machine, but the LLVM framework now provides functionality far beyond that of a virtual machine. In contrast to JVM and .NET, its language, LLVM intermediate representation (IR) does not provide high-level features like memory safety and such, and its abstraction level is only slightly higher than that of actual assembly languages. LLVM is not tied to a specific vendor or operating system. It is a versatile compiler infrastructure that facilitates the generation of optimised machine code from an intermediate representation.

In recent years, WebAssembly (WASM) [9, 10] has emerged as a new technology. It represents a binary instruction format designed for safe and efficient execution on web browsers. Unlike traditional JavaScript, WASM allows high-performance applications to run directly in the browser, providing a significant boost to web-based software.

Both the .NET framework and LLVM have played crucial roles in the evolution of WASM. The .NET ecosystem, through projects like Blazor, has embraced WASM as a target environment, enabling developers to build web applications using C# and .NET languages. This integration leverages the strengths of .NET while extending its reach to the web, creating a seamless development experience for web and desktop applications alike.

On the other hand, LLVM has contributed to the WASM ecosystem by providing a robust compilation target. Compilers like Emscripten, which leverages LLVM, can convert code written in languages like C and C++ into WASM-compatible format. This integration enables developers to take advantage of the performance benefits offered by WASM, making it an attractive option for performance-critical web applications.

The convergence of WASM with technologies like .NET and LLVM highlights the adaptability and extensibility of modern programming ecosystems. By integrating with these established platforms, WASM has opened up new avenues for web development, enabling the creation of high-performance web applications with a wide range of programming languages. This interaction between WASM, .NET, and LLVM showcases the dynamic nature of the programming landscape and the collaborative efforts driving innovation in the field.

2 Preliminaries

In this course, we are going to discuss and reason about properties of various programming languages and language features. To do this properly, we need a formal meta-language which allows us to make statements about these properties and formally prove such statements. For example, we want to specify what constitutes a legal program in a language, and the static semantics (often in form of typing and scoping rules) and dynamic semantics of a language. Fortunately, it turns out that a single formalism, namely inductive definitions built on inference rules, is sufficient to do this.

2.1 Judgements and Inference Rules

A judgement is simply a statement that a certain property holds for a specific object[1]. For example, the following statements are judgebments:

- `3 + 4 * 5` is a valid arithmetic expression
- the string `"aba"` is a palindrome
- `0.43423` is a floating point value

Judgements are not unlike predicates you might know from predicate logic. We write

$$a\ \boldsymbol{S}$$

for a judgement of the form *The property* $\boldsymbol{S}$ *holds for object* a. In predicate logic, this is usually written differently, as $\boldsymbol{S}(a)$. However, we will see later that for our purpose, it is much more convenient to write it in the post-fix notation shown above.

Instead of viewing $\boldsymbol{S}$ as a predicate, we can also interpret it as a set of objects with a certain property, and read the judgement $a\ \boldsymbol{S}$ as: a is an element of the set $\boldsymbol{S}$. Some examples of judgements and how to read them are:

- `5` $\boldsymbol{Even}$
  - `5` is even, or
  - `5` is an element of the set of even numbers
- `3 + 4 * 5` $\boldsymbol{Expr}$
  - `3 + 4 * 5` is a syntactically correct expression, or
  - `3 + 4 * 5` an element of the set containing all syntactically correct expressions
- `0.43423` $\boldsymbol{float}$
  - `0.43423` is a floating point value

[1]More generally, a relationship between a number of objects holds. For now, we just look at statements about a single object.

– `0.43423` is an element of the set of floating points values

Judgements by themselves would be boring and fairly useless. Most interesting sets have an infinite number of elements, and to define such a set, it would obviously be impossible to explicitly list them all using simple judgements. Luckily, the sets we are interested in are also no random collections of objects, but for example programs which can be constructed according to a fixed set of rules. We define them using so-called inference rules.

Inference rules allow us to combine judgements to obtain new judgements. They have the following general form:

> If judgements J_1, and J_2, and ... and J_n are inferable, then the judgement J is inferable

and are usually given in the following standard notation:

$$\frac{J_1 \quad J_2 \quad \ldots \quad J_n}{J}$$

where the judgements J_1 to J_n are called premises, and J is called a conclusion. An inference rule does not have to have premises, it can consist of a single conclusion. Such inference rules are called axioms. But let us have a look at a concrete example now.

We start by defining the set of natural numbers (***Nat***), and some simple properties over this set. For simplicity reasons, we represent them as terms, with `0`, `(s 0)`, `(s (s 0))`, `(s (s (s 0)))` for $0, 1, 2, 3$, and so on (`s` here stands for *successor*). So, first of all, how can we define ***Nat*** itself using inference rules? Listing all the element of ***Nat*** would be equivalent to including an axiom for each number:

$$\frac{}{\texttt{0}\ \textit{\textbf{Nat}}} \qquad (\textit{Nat-1})$$

$$\frac{}{\texttt{(s 0)}\ \textit{\textbf{Nat}}} \qquad (\textit{Nat-2a})$$

$$\frac{}{\texttt{(s (s 0))}\ \textit{\textbf{Nat}}} \qquad (\textit{Nat-2b})$$

$$\vdots$$

Clearly, this is not feasible, and luckily, also not necessary, since we can easily see that, apart from the first one, all the rules have the form

$$\frac{}{\texttt{(s } x\texttt{)}\ \textit{\textbf{Nat}}}$$

where x ***Nat*** has been established by the previously listed axiom. In words, we have

1. `0` is in ***Nat***, and
2. for all x, if x is in ***Nat***, then `(s` x`)` is in ***Nat***

which can be translated directly into the following two inference rules:

$$\frac{}{\texttt{0}\ \textit{\textbf{Nat}}} \qquad (\textit{Nat-1})$$

$$\frac{x\ \textit{\textbf{Nat}}}{\texttt{(s } x\texttt{)}\ \textit{\textbf{Nat}}} \qquad (\textit{Nat-2})$$

where x can be instantiated to any term.

With Rule *Nat-1* and Rule *Nat-2*, we do not need axioms Rule *Nat-2a*, Rule *Nat-2b* and such, because we can derive the membership of each number. For example, we can prove `(s 0)` ***Nat*** by instantiating s in Rule *Nat-2* to `0`. We then get

$$\frac{\texttt{0}\ \textbf{\textit{Nat}}}{\texttt{(s 0)}\ \textbf{\textit{Nat}}}(\textit{Nat-2})$$

This leaves us with the proof obligation `0` ***Nat***, which holds because of Rule *Nat-1*. Since this is an axiom, no further proof obligations are left.

More formally, we can write the proof by stating the proof goal ($[G]$), and listing the steps and justifications which lead to the proof goal:

Proof
$[G]$ `(s 0)` ***Nat***
Begin
1. $\{$*Rule Nat-1*$\}$ `0` ***Nat***
2. $\{$1., *Rule Nat-2*$\}$ `(s 0)` ***Nat***

End

An alternative and often quite convenient way to write inference proofs is to stack the rules we apply together and draw a “proof tree” — in our example, more a proof stack, since each rule has just a single premise. We add the name of the rule we apply in each step.

$$\frac{\dfrac{}{\texttt{0}\ \textbf{\textit{Nat}}}(\textit{Nat-1})}{\texttt{(s 0)}\ \textbf{\textit{Nat}}}(\textit{Nat-2})$$

In the same way as we defined ***Nat***, we can define the sets or properties ***Even***

$$\frac{}{0\ \textbf{\textit{Even}}} \qquad (\textit{Even-1})$$

$$\frac{x\ \textbf{\textit{Even}}}{\texttt{(s (s }x\texttt{))}\ \textbf{\textit{Even}}} \qquad (\textit{Even-2})$$

and ***Odd***:

$$\frac{}{\texttt{(s 0)}\ \textbf{\textit{Odd}}} \qquad (\textit{Odd-1})$$

$$\frac{x\ \textbf{\textit{Odd}}}{\texttt{(s (s }x\texttt{))}\ \textbf{\textit{Odd}}} \qquad (\textit{Odd-2})$$

Rules do work in two ways: we can use them to define a property, but we can also use them to show that a judgement is valid. For example, assume we want to show that some judgement J is valid. We have to look for a rule which has J as a conclusion. If this rule is an axiom, we are already done. If not, we have to show that all of its premises are valid by recursively applying the same strategy to all of them. For example, we can show that `(s (s 0))` ***Even***, since

$$\frac{\texttt{0}\ \textbf{\textit{Even}}}{\texttt{(s (s 0))}\ \textbf{\textit{Even}}}(\textit{Even-2})$$

and

$$\frac{}{\texttt{0}\ \textbf{\textit{Even}}}(\textit{Even-1})$$

As the last rule is an axiom, there are no premises left to prove, and we are done.

Similarly, we can show that `(s (s (s (s 0))))` ***Even***:

$$\frac{\dfrac{\dfrac{}{\texttt{0}\ \textbf{\textit{Even}}}(\textit{Even-1})}{\texttt{(s (s 0))}\ \textbf{\textit{Even}}}(\textit{Even-2})}{\texttt{(s (s (s (s 0))))}\ \textbf{\textit{Even}}}(\textit{Even-2})$$

Note that inference works on a purely syntactic basis. Given the rules above, we are not able to prove 2 ***Even***, even though we can show that `(s (s 0))` ***Even***, and we know that `(s (s 0))` is equal to 2, we cannot apply that knowledge, since we have no rule which tells us it is ok to do so. We just mechanically manipulate terms according to the given rules.

Let us look at a slightly more interesting example: we want to define the language ***M*** which contains all expressions of properly matched parenthesis (and no other characters) (ϵ represents the empty string).

$$\boldsymbol{M} = \{\epsilon, \texttt{()}, \texttt{()()}, \texttt{()()()}, \ldots, \texttt{(())}, \texttt{((()))}, \ldots, \texttt{()(())}, \texttt{()()(())}, \ldots\}$$

Again, let us start by describing the rules in (semi-)natural language. There are basically two ways to "legally" combine parenthesis: we can either nest them, or concatenate them:

1. The empty string (denoted by ϵ) is in ***M***
2. If s_1 and s_2 are in ***M***, then $s_1 s_2$ is in ***M***
3. If s is in ***M***, then `(`x`)` is in ***M***

These rules can be directly translated into inference rules:

$$\frac{}{\epsilon\ \boldsymbol{M}} \qquad (M\text{-}1)$$

$$\frac{s_1\ \boldsymbol{M} \quad s_2\ \boldsymbol{M}}{s_1 s_2\ \boldsymbol{M}} \qquad (M\text{-}2)$$

$$\frac{s\ \boldsymbol{M}}{\texttt{(}s\texttt{)}\ \boldsymbol{M}} \qquad (M\text{-}3)$$

How can we show that `()(())` is in ***M***? As we did previously, we check if there is a rule whose conclusion matches the judgement we want to infer. If such a rule exists, we apply it and try to prove the premises in the same way, until no premises are left.

If we apply Rule (*M-2*), for, example, we have to show that both `()` and `(())` are in ***M***. Since `()`=`(`ϵ`)`, we can apply Rule (*M-3*), and only have to show that ϵ is in ***M*** (Rule (*M-1*)). By applying Rule (*M-3*) in the same way, we can show that `(())` is in ***M*** as well, and we are done.

$$\dfrac{\dfrac{\dfrac{}{\epsilon\ \boldsymbol{M}}(M\text{-}1)}{\texttt{()}\ \boldsymbol{M}}(M\text{-}3) \qquad \dfrac{\dfrac{\vdots}{\texttt{()}\ \boldsymbol{M}}(\textit{see proof on the left})}{\texttt{(())}\ \boldsymbol{M}}(M\text{-}3)}{\texttt{()(())}\ \boldsymbol{M}}(M\text{-}2\textit{, with } s_1 = \texttt{()},\ s_2 = \texttt{(())})$$

This proof worked, but only because we know how to break the string into substrings which are in ***M***. But nothing in the rules tells us which rule to apply when. For example, instead of Rule (*M-2*), we could also have applied Rule (*M-3*), to strip away the two outer brackets. It has only a single premise, so we only have to prove that `)(()` is in ***M***. The only rule that's applicable now is Rule (*M-2*), and we could apply it in several different ways resulting in different premises:

$$\frac{\texttt{)}\ \boldsymbol{M} \quad \texttt{(()}\ \boldsymbol{M}}{\texttt{)(()}\ \boldsymbol{M}}(M\text{-}2)$$

or

$$\frac{\texttt{)(}\ \boldsymbol{M} \quad \texttt{()}\ \boldsymbol{M}}{\texttt{)(()}\ \boldsymbol{M}}(M\text{-}2)$$

or

$$\frac{\texttt{)((}\ \boldsymbol{M} \quad \texttt{)}\ \boldsymbol{M}}{\texttt{)(()}\ \boldsymbol{M}}(M\text{-}2)$$

In the first application, we end up with the premise: `)` is in $\boldsymbol{M}$, but there is no rule which we can apply to get rid of it. This is not that surprising, since $\boldsymbol{M}$ should only contain expressions of properly matched parenthesis, and `)`, as well as `)(()` are not properly matched. So, by choosing the wrong rule, or applying the right rule in a wrong way – for example, splitting `()()` up into `()(` and `)` – it is easily possible to end up with premises which are not actually valid and reach a dead end. Or, we could have started with Rule *M-2*, but split the string at a different position, instantiating s_1 to `()((`, and s_2 to `))`. This would have resulted in a dead end as well, as the substrings are not in $\boldsymbol{M}$.

In our example, it was not hard to see which rule to apply, and how. It can be extremely difficult to decide which rule to apply and how, without some background knowledge about the objects and properties, as there might be an infinite number of possibilities. This is one reason why it is not possible to write a program which automatically infers judgements, and which guarantees to find such a derivation if it exists. It is, however, possible to write semi-automatic theorem provers, which come up with proofs in cases where it is fairly standard, and rely on user input otherwise.

2.1.1 Derivable and Admissible Rules

What would happen if we added the following rule:

$$\frac{s\ \boldsymbol{M}}{\texttt{((}s\texttt{))}\ \boldsymbol{M}} \qquad (\textit{Extra-1})$$

Does this change the set $\boldsymbol{M}$ in any way, that is, is there a string s for which we can infer $s\ \boldsymbol{M}$ if we use *Extra-1*, but not otherwise? This doesn't seem very likely: the rule just says that, if a string s is in $\boldsymbol{M}$ it is ok to add two pairs of matching parenthesis. Since we already had a rule which allows us to add one pair of parenthesis, we can apply this rules twice and achieve the same effect:

$$\dfrac{\dfrac{s\ \boldsymbol{M}}{\texttt{(}s\texttt{)}\ \boldsymbol{M}}(\textit{M-3})}{\texttt{((}s\texttt{))}\ \boldsymbol{M}}(\textit{M-3})$$

This means that Rule *Extra-1* is derivable from the existing rules. In contrast to our previous proofs, we didn't proof a simple judgement, since not all our proof branches end in the application of an axiom, but another rule, with an unproven premise left.

The proof-tree above corresponds to the proof with has the goal $\texttt{((}s\texttt{))}\ \boldsymbol{M}$ and the assumption $(A)\ s\ \boldsymbol{M}$:

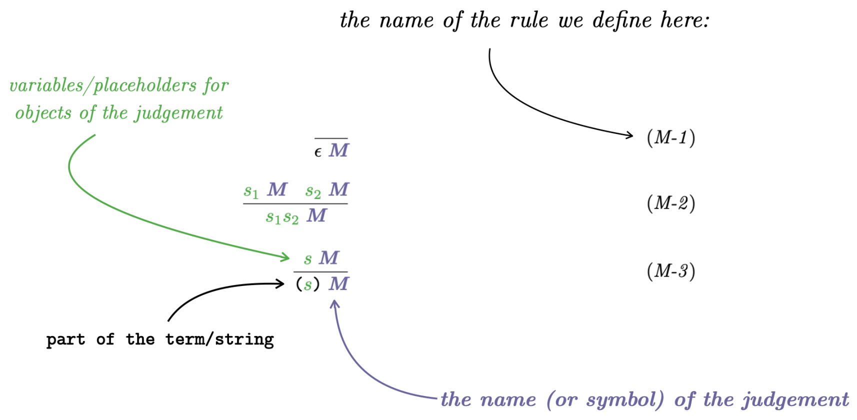
Proof
[A] $s\ \boldsymbol{M}$
[G] $\texttt{((}s\texttt{))}\ \boldsymbol{M}$
Begin
1. $\{A, \textit{Rule M-3},\}$ $\texttt{(}s\texttt{)}\ \boldsymbol{M}$
2. $\{1., \textit{Rule M-3}\}$ $\texttt{((}s\texttt{))}\ \boldsymbol{M}$

End

In all the previous rules, the strings in the premises were simpler than the string in the conclusion, and this is the case for most inference rules we will look at, but premises don't have to be simpler. For example, we might add the following rule:

$$\frac{\texttt{()}s\ \boldsymbol{M}}{s\ \boldsymbol{M}} \qquad (\textit{Extra-2})$$

Interestingly, although Rule *Extra-2* does not introduce any new strings to $\boldsymbol{M}$, the rule is also not derivable from any of the existing rules. Such a rule, which is not derivable, but does also not introduce new elements, is called admissible.



the name of the rule we define here:
variables/placeholders for objects of the judgement
$\frac{}{\epsilon\ M}$ (M-1)
$\frac{s_1\ M \quad s_2\ M}{s_1 s_2\ M}$ (M-2)
$\frac{s\ M}{(s)\ M}$ (M-3)
part of the term/string
the name (or symbol) of the judgement

Figure 2.1: Notation and fonts

2.1.2 Notation

Before we continue, a few words about the notation used in the lecture notes. To make it easier to read the inference rules and other formalisations, we choose different fonts to denote different parts of a judgement, rule or proof (see also Figure 2.1.1):

- Objects which are part of our language, such as strings or terms, are set in `black type writer font`.
- Variables which represent arbitrary objects are set in *green italics*.
- The name (or later also symbols) of properties or sets are set in ***bold, blue italics***.
- The name of a rule in *black italics*.

Whenever we define a rule, the name of the rule is given on the right. When we use a rule in a proof, we often annotate the horizontal line of the inference with the name of the rule, and sometimes additional information, like the instantiation of the variables in the rule.

2.1.3 Inductive Definitions

A set of inference rules R defining a set $\boldsymbol{A}$ is called an inductive definition of $\boldsymbol{A}$, if s $\boldsymbol{A}$ holds if and only if s $\boldsymbol{A}$ is derivable using R. All the examples we discussed are inductive definitions.

Not all sets can be defined using inductive definitions. For example, while natural numbers are one of the standard examples for such sets, floating point numbers cannot be defined in such a way.

2.1.4 Judgements and Relations

So far, we defined a judgement to be a statement about a property of an object. We can generalise this definition to a relation between a number of objects. Consider the following inductive definition of the relation "a $<$ b" on natural numbers. For convenience reasons, we choose an infix notation here:

$$\frac{n\ \boldsymbol{Nat}}{\texttt{0} < (\texttt{s}\ n)} \qquad (<\text{-}1)$$

$$\frac{n < m}{(\texttt{s}\ n) < (\texttt{s}\ m)} \qquad (<\text{-}2)$$

As before, we can also view this as an inductive definition of a set. In this case, a set of pairs, where (a, b) in $<$ if and only if a is less than b. The inductive definition of $<$ depends on the definition of ***Nat***.

2.2 Rule Induction

Natural deduction by itself is sometimes not powerful enough. For example, although we can see that the Rule *Extra-2* in Section 2.1.1 is valid for every string s in $\boldsymbol{M}$, we cannot show this by simply combining the existing rules. We will therefore introduce another proof technique here, called induction. You will probably know induction over natural numbers and structural induction from mathematics and previous courses. Both are special cases of a more general induction principle called rule induction or natural induction.

Let us go back to our previous example set $\boldsymbol{M}$ of properly matched parenthesis. The rules *M-1* to *M-3* provide an inductive definition of $\boldsymbol{M}$:

$$\frac{}{\epsilon\ \boldsymbol{M}} \tag{\textit{M-1}}$$

$$\frac{s_1\ \boldsymbol{M} \quad s_2\ \boldsymbol{M}}{s_1 s_2\ \boldsymbol{M}} \tag{\textit{M-2}}$$

$$\frac{s\ \boldsymbol{M}}{(s)\ \ \boldsymbol{M}} \tag{\textit{M-3}}$$

Now, let us assume we want to prove some property $\boldsymbol{P}$, which holds for all strings in $\boldsymbol{M}$. That is, we want to show that if $s\ \boldsymbol{M}$ then $s\ \boldsymbol{P}$.

If we can show that for every rule for $\boldsymbol{M}$, the corresponding rule for $\boldsymbol{P}$ also holds: that is, if we can prove that the following rules are valid

$$\frac{}{\epsilon\ \boldsymbol{P}} \tag{\textit{P-1}}$$

$$\frac{s_1\ \boldsymbol{P} \quad s_2\ \boldsymbol{P}}{s_1 s_2\ \boldsymbol{P}} \tag{\textit{P-2}}$$

$$\frac{s\ \boldsymbol{P}}{(s)\ \ \boldsymbol{P}} \tag{\textit{P-3}}$$

then we can conclude that, indeed, $s\ \boldsymbol{M}$ implies $s\ \boldsymbol{P}$. Why is this the case? Well, if $s\ \boldsymbol{M}$, then we know that there is at least one derivation using the rules *M-1* to *M-3*, and therefore there is a proof with exactly the same structure for $s\ \boldsymbol{P}$.

For example, for the string `()(())`, we have our proof

$$\dfrac{\dfrac{\dfrac{}{\epsilon\ \boldsymbol{M}}(\textit{M-1})}{()\ \ \boldsymbol{M}}(\textit{M-3}) \qquad \dfrac{\dfrac{\vdots}{()\ \ \boldsymbol{M}}(\textit{see proof on the left})}{(())\ \ \boldsymbol{M}}(\textit{M-3})}{()(())\ \ \boldsymbol{M}}(\textit{M-2, with } s_1 = (),\ s_2 = (()))$$

We can take this proof, and rewrite it to a proof for `()(())` $\boldsymbol{P}$, by simply replacing the property $\boldsymbol{M}$ with $\boldsymbol{P}$:

$$\dfrac{\dfrac{\dfrac{}{\epsilon\ \boldsymbol{P}}(\textit{P-1})}{()\ \ \boldsymbol{P}}(\textit{P-3}) \qquad \dfrac{\dfrac{\vdots}{()\ \ \boldsymbol{P}}(\textit{see proof on the left})}{(())\ \ \boldsymbol{P}}(\textit{P-3})}{()(())\ \ \boldsymbol{P}}(\textit{P-2, with } s_1 = (),\ s_2 = (()))$$

Since we know we can construct such a proof `()(())` $\boldsymbol{M}$ for any string s with `()(())` $\boldsymbol{M}$, we know that `()(())` $\boldsymbol{M}$ implies `()(())` $\boldsymbol{P}$. This is called the principal of rule (or structural) induction, and induction over natural numbers is just a special case of this.

Proving the axioms rules are the base cases, and as for induction over natural numbers when we have to establish that the property holds for zero, the bases cases are usually easy to prove. For the rules with premises, we assume that these hold (the premises become the induction hypotheses). For example, for Rule *P-2*, we assume s_1 $\boldsymbol{P}$ and s_2 $\boldsymbol{P}$ and induction hypotheses, and show that they imply $s_1 s_2$ $\boldsymbol{P}$.

Let's look at a concrete example: given two functions, which count the number of opening and closing brackets, respectively, we want to show as property $\boldsymbol{P}$ for all s $\boldsymbol{M}$:

$$s\ \boldsymbol{P} : \mathit{open}(s) = \mathit{close}(s)$$

The functions are defined by following set of rules:

$$\mathit{open}(\epsilon) = 0 \qquad (\mathit{open\text{-}1})$$
$$\mathit{open}(\texttt{(}s) = 1 + \mathit{open}(s) \qquad (\mathit{open\text{-}2})$$
$$\mathit{open}(\texttt{)}s) = \mathit{open}(s) \qquad (\mathit{open\text{-}3})$$
$$\mathit{open}(s_1 s_2) = \mathit{open}(s_1) + \mathit{open}(s_2) \qquad (\mathit{open\text{-}4})$$

$$\mathit{close}(\epsilon) = 0 \qquad (\mathit{close\text{-}1})$$
$$\mathit{close}(\texttt{(}s) = \mathit{close}(s) \qquad (\mathit{close\text{-}2})$$
$$\mathit{close}(\texttt{)}s) = 1 + \mathit{close}(s) \qquad (\mathit{close\text{-}3})$$
$$\mathit{close}(s_1 s_2) = \mathit{close}(s_1) + \mathit{close}(s_2) \qquad (\mathit{close\text{-}4})$$

Now, if we want to show that all s in $\boldsymbol{M}$ have the same number of opening and closing brackets, that is, if s $\boldsymbol{M}$ is derivable, then $\mathit{open}(s) = \mathit{close}(s)$, we need proofs for *P-1* to *P-2*.

- Proving *P-1*

  Proof

  [*G*] $\mathit{open}(\epsilon) = \mathit{close}(\epsilon)$

  Begin

  $$\begin{aligned} & \mathit{open}(\epsilon) \\ = & \ \{\mathit{open\text{-}1}\} \\ & 0 \\ = & \ \{\mathit{close\text{-}1}\} \\ & \mathit{close}(\epsilon) \end{aligned}$$

  End

- Proving *P-2*

  Proof

  [*IH-1*] $\mathit{open}(s_1) = \mathit{close}(s_1)$
  [*IH-2*] $\mathit{open}(s_2) = \mathit{close}(s_2)$
  [*G*] $\mathit{open}(s_1 s_2) = \mathit{close}(s_1 s_2)$

  Begin

  $$\begin{aligned} & \mathit{open}(s_1 s_2) \\ = & \ \{\mathit{open\text{-}4}\} \\ & \mathit{open}(s_1) + \mathit{open}(s_2) \\ = & \ \{\mathit{IH\text{-}1},\ \mathit{IH\text{-}2}\} \\ & \mathit{close}(s_1) + \mathit{close}(s_2) \\ = & \ \{\mathit{close\text{-}4}\} \\ & \mathit{close}(s_1 s_2) \end{aligned}$$

End

- Proving *P-3*

Proof

$[IH]\quad open(s) = close(s)$
$[G\ ]\quad open(\texttt{(}s\texttt{)}) = close(\texttt{(}s\texttt{)})$

Begin

$$
\begin{array}{ll}
 & open(\texttt{(}s\texttt{)}) \\
= & \{open\text{-}4\} \\
 & open(\texttt{(}) + open(s) + open(\texttt{)}) \\
= & \{open\text{-}3,\ open\text{-}2, open\text{-}4\} \\
 & 1 + open(s) + 0 \\
= & \{\text{IH, Arithmetic}\} \\
 & 1 + close(s) + 0 \\
= & close\text{-}3,\ close\text{-}3,\ close\text{-}4\} \\
 & close(\texttt{(}) + close(s) + close(\texttt{)}) \\
= & \{close\text{-}4\} \\
 & close(\texttt{(}s\texttt{)})
\end{array}
$$

End

In the proof, we used the rules of arithmetic, and that properties of *close* and *open*, such as $open(\texttt{(}) = close\ (\texttt{)}) = 1$, and $open(\texttt{)}) = close(\texttt{(}) = 0$.

2.2.1 Ambiguity

The definition of $\boldsymbol{M}$, although correct, has a undesirable property: for any string in $\boldsymbol{M}$, we do not have just one derivation, but an infinite number of possible derivations, since any string s can be split into ϵ and s by applying Rule *M-2*, and then Rule *M-1* to get rid of ϵ. However, this derivation step is completely unnecessary.

Fortunately, if we take a more structured view on the elements of this language, we can come up with an alternative set of rules, where we have exactly one derivation for each string in the set. We can interpret each string as a possibly empty list ($\boldsymbol{L}$) of non-empty parenthesis expressions ($\boldsymbol{N}$) according to the following inference rules:

$$\frac{}{\epsilon\ \boldsymbol{L}} \qquad (L\text{-}1)$$

$$\frac{s_1\ \boldsymbol{N} \quad s_2\ \boldsymbol{L}}{s_1 s_2\ \boldsymbol{L}} \qquad (L\text{-}2)$$

$$\frac{s\ \boldsymbol{L}}{\texttt{(}s\texttt{)}\ \boldsymbol{N}} \qquad (N\text{-}1)$$

The interesting point here is that $\boldsymbol{L}$ and $\boldsymbol{N}$ are defined in terms of each other: we have a mutually recursive definition.

Let us look at one more example of an ambiguous definition: simple arithmetic expressions, given here both in BNF[2] and as inductive definition using inference rules:

The BNF

$$\boldsymbol{Expr} \quad ::= \quad \boldsymbol{Int} \mid \texttt{(}\boldsymbol{Expr}\texttt{)} \mid \boldsymbol{Expr}\ \texttt{+}\ \boldsymbol{Expr} \mid \boldsymbol{Expr}\ \texttt{*}\ \boldsymbol{Expr}$$

[2]BNF means 'Backus-Naur form': we're using this sometimes in the course as compact description of a language, always in combination with an alternative description, so it is not a problem if you are not familiar with it – you can safely ignore it.

describes the same language as the following set of inference rules, where ***Int*** is the set of integer constants we assume defined elsewhere (from now on, we use i to denote an integer, without explicitly specifying $i \in$ ***Int*** as precondition).

$$\frac{i \in \textbf{\textit{Int}}}{i \ \textbf{\textit{Expr}}} \quad (E\text{-}1)$$

$$\frac{e \ \textbf{\textit{Expr}}}{\texttt{(}e\texttt{)} \ \textbf{\textit{Expr}}} \quad (E\text{-}2)$$

$$\frac{e_1 \ \textbf{\textit{Expr}} \quad e_2 \ \textbf{\textit{Expr}}}{e_1 \ \texttt{+} \ e_2 \ \textbf{\textit{Expr}}} \quad (E\text{-}3)$$

$$\frac{e_1 \ \textbf{\textit{Expr}} \quad e_2 \ \textbf{\textit{Expr}}}{e_1 \ \texttt{*} \ e_2 \ \textbf{\textit{Expr}}} \quad (E\text{-}4)$$

Although in this case, there are not an infinite number of possible derivations for each expression, every expression which contains more than a single arithmetic operation still can be derived in more than one way:

$$\dfrac{\dfrac{\dfrac{}{\texttt{1} \ \textbf{\textit{Expr}}}(E\text{-}1) \quad \dfrac{}{\texttt{2} \ \textbf{\textit{Expr}}}(E\text{-}1)}{\texttt{1 + 2} \ \textbf{\textit{Expr}}}(E\text{-}3) \quad \dfrac{}{\texttt{3} \ \textbf{\textit{Expr}}}(E\text{-}1)}{\texttt{1+2*3} \ \textbf{\textit{Expr}}}(E\text{-}4)$$

$$\dfrac{\dfrac{}{\texttt{1} \ \textbf{\textit{Expr}}}(E\text{-}1) \quad \dfrac{\dfrac{}{\texttt{2} \ \textbf{\textit{Expr}}}(E\text{-}1) \quad \dfrac{}{\texttt{3} \ \textbf{\textit{Expr}}}(E\text{-}1)}{\texttt{2 * 3} \ \textbf{\textit{Expr}}}}{\texttt{1+2*3} \ \textbf{\textit{Expr}}}(E\text{-}3)$$

Although both derivations are correct with respect to the rules given, the second derivation is more appropriate for an arithmetic expression, as it decomposes the expression first into two summands.

We give an alternative definition, which takes precedence and associativity of the operators into account:

$$\frac{e_1 \ \textbf{\textit{SExpr}} \quad e_2 \ \textbf{\textit{PExpr}}}{e_1 \ \texttt{+} \ e_2 \ \textbf{\textit{SExpr}}} \quad (S\text{-}1)$$

$$\frac{e \ \textbf{\textit{PExpr}}}{e \ \textbf{\textit{SExpr}}} \quad (S\text{-}2)$$

$$\frac{e_1 \ \textbf{\textit{PExpr}} \quad e_2 \ \textbf{\textit{FExpr}}}{e_1 \ \texttt{*} \ e_2 \ \textbf{\textit{PExpr}}} \quad (P\text{-}1)$$

$$\frac{e \ \textbf{\textit{FExpr}}}{e \ \textbf{\textit{PExpr}}} \quad (P\text{-}2)$$

$$\frac{e \ \textbf{\textit{SExpr}}}{\texttt{(}e\texttt{)} \ \textbf{\textit{FExpr}}} \quad (F\text{-}1)$$

$$\frac{n \in \textbf{\textit{Int}}}{n \ \textbf{\textit{FExpr}}} \quad (F\text{-}2)$$

These rules reflect the fact that, given an expression of the form e_1 `+` e_2, we only want to break it apart at the `+` operator if e_2 is a number, parenthesised expression, or a multiplication, but not if it's another addition. We use ***SExpr*** to represent all arithmetic expressions (the same set as ***Expr***), and ***PExpr*** the subset of ***SExpr*** which excludes (unparenthesised) additions. Similarly, expressions of the form e_1 `*` e_2 should only be separated if e_1 is an ***PExpr***, and e_2 either a number or a parenthesised expression (denoted by ***FExpr***).

Consider the previous expression again, to try and show it is in ***SExpr***, we can still apply the "wrong", that is the product rule, first:

$$\dfrac{\dfrac{\dfrac{??}{\texttt{1+2}\ \boldsymbol{FExpr}}\,(??)\qquad\dfrac{\dfrac{}{\texttt{3}\ \boldsymbol{FExpr}}\,(F\text{-}2)}{\texttt{3}\ \boldsymbol{PExpr}}\,(P\text{-}2)}{\texttt{1+2*3}\ \boldsymbol{PExpr}}\,(P\text{-}1)}{\texttt{1+2*3}\ \boldsymbol{SExpr}}\,(S\text{-}2)$$

However, there is not way to derive `1+2` ***FExpr***, and we are stuck. The only way to derive `1+2*3` ***SExpr*** successfully is by applying the sum-rule first:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{1}\ \boldsymbol{PExpr}}\,(F\text{-}2)}{\texttt{1}\ \boldsymbol{PExpr}}\,(P\text{-}2)}{\texttt{1}\ \boldsymbol{SExpr}}\,(S\text{-}2)\qquad\dfrac{\dfrac{\dfrac{}{\texttt{2}\ \boldsymbol{FExpr}}\,(F\text{-}2)}{\texttt{2}\ \boldsymbol{PExpr}}\,(F\text{-}2)\qquad\dfrac{}{\texttt{3}\ \boldsymbol{FExpr}}\,(F\text{-}2)}{\texttt{2*3}\ \boldsymbol{PExpr}}\,(P\text{-}1)}{\texttt{1+2*3}\ \boldsymbol{SExpr}}\,(S\text{-}1)$$

The unambiguous grammar is, again, much more complicated than the original grammar, even for such a simple language. This is not surprising, as it contains additional structural information. For programming languages, ambiguous grammars are problematic, as they may allow different interpretations of programs, and are therefore usually avoided.

2.2.2 Simultaneous Induction

How can we apply the principle of rule induction to mutually recursive definitions like those of ***L*** and ***SExpr***? In most cases, we have to generalise the proof goal. For example, if we want to prove a property ***P*** for all e in ***SExpr***, that is e ***SExpr*** implies e ***PExpr***. By the principle of rule induction we have to provide the proofs for each of the two rules for ***SExpr***. That is, for Rule *S-1*, we have to show

[*A*] e_2 ***PExpr***
[*IH*] e_1 ***P***
[*G*] e_1`+`e_2 ***P***

and for Rule *S-2*

[*A*] e ***PExpr***
[*G*] e ***P***

The problem is that, since we only try to show something about ***SExpr***, the induction hypothesis does not say anything about e_2 for the first rule (we only know that e_2 ***PExpr***, and can therefore not conclude directly that the induction hypothesis holds for e_2. We also know nothing directly about e of the second rule[3]. In most cases, this is not enough to prove anything. The solution is often to try and prove a more general statement instead which leads to stronger induction hypothesis using simultaneous induction.

We try to show simultaneously that

- e ***SExpr*** implies e ***P***, and
- e ***PExpr*** implies e ***P***, and

[3]In this particular example, it is not much of a problem, since ***PExpr*** is a subset of ***SExpr***, we can easily show that anything that is true for all objects in ***SExpr*** has to be true for the objects in ***PExpr***. In general, though, this is not the case and generalising the property as suggested simplifies the proof

- e ***FExpr*** implies e ***P***

We do have more cases to cover (one for each inference rule which has ***PExpr***, ***SExpr*** or ***FExpr*** in the conclusion) as a consequence, but we do get something for our efforts – the induction hypothesis is stronger this way, as it covers all the premises in the rules:

- For Rule *S-1*

  [*IH-1*] e_1 ***P***
  [*IH-2*] e_2 ***P***
  [*G*] e_1+e_2 ***P***

- For Rule *S-2* (this is now trivial):

  [*IH*] e ***P***
  [*G*] e ***P***

- For Rule *P-1*

  [*IH-1*] e_1 ***P***
  [*IH-2*] e_2 ***P***
  [*G*] e_1*e_2 ***P***

- ...

Example 2: Proving *L* = *M*. Let us look at another example of a rule induction proof. Consider again the setof balances parentheses. Intuitively, it seems clear that the rules for ***M*** and ***L*** define the same set. However, it is not completely straight forward to prove that this is correct. As with any set-equality proof, we can split it in two parts, and show that the two sets are subsets of each other:

1. show that s ***M*** implies s ***L*** (or, in other words, ***M*** $\subseteq$ ***L***)
2. show that s ***L*** implies s ***M*** (***L*** $\subseteq$ ***M***)

We show only the first part here. It is clear that we cannot prove this by derivation, since there is no overlap in the rules of ***M*** and ***L***. We can, however, prove it using rule induction.

That is, the property ***P*** we want to prove for all s ***M*** is s ***L***. Since there are three rules for ***M***, it means we have to provide these three proofs:

- For Rule *M-1*

  Proof

  [*G*] ϵ ***L***

  Begin

  1. {*Rule L-1*} ϵ ***L***

  End

- For Rule *M-3*

  Proof

  [*IH*] s ***L***
  [*G*] (s) ***L***

  We can show this directly, using the assumption and the induction hypothesis:

  Begin

$$\dfrac{\dfrac{\dfrac{}{s\ \boldsymbol{L}}(\mathit{IH})}{(s)\ \boldsymbol{N}}(\mathit{N}) \qquad \dfrac{}{\epsilon\ \boldsymbol{L}}(\mathit{L\text{-}1})}{(s)\ \boldsymbol{L}}(\mathit{L\text{-}2})$$

End

Alternatively, we can write the proof as

Begin

1.	$\{\mathit{IH}\}$	$s\ \boldsymbol{L}$
2.	$\{1.\ \mathit{and\ Rule\ N}\}$	$(s)\ \boldsymbol{N}$
3.	$\{\mathit{Rule\ L\text{-}1}\}$	$\epsilon\ \boldsymbol{L}$
4.	$\{2.,\ 3.,\ \mathit{and\ Rule\ L\text{-}2}\}$	$(s)\ \boldsymbol{L}$

End

- For Rule *M-2*

  Proof

  [*IH-1*] $s_1\ \boldsymbol{L}$
  [*IH-2*] $s_2\ \boldsymbol{L}$
  [*G*] $s_1 s_2\ \boldsymbol{L}$

  Unfortunately, this case is more complicated. The only rule we can apply to prove $s_1 s_2\ \boldsymbol{L}$ is Rule *L-2*, but this leaves us stuck with the proof obligation $s_1\ \boldsymbol{N}$, which we can't show. All we know about s_1 is that it's in $\boldsymbol{L}$, but it might be the empty string, or `()()`, or any other string that is in $\boldsymbol{L}$, but not in $\boldsymbol{N}$.

$$\dfrac{\dfrac{?}{s_1\ \boldsymbol{N}} \qquad \dfrac{}{s_2\ \boldsymbol{L}}(\mathit{IH\text{-}2})}{s_1 s_2\ \boldsymbol{L}}(\mathit{L\text{-}2})$$

  So, we need to show that $s_1\ \boldsymbol{L}$ and $s_2\ \boldsymbol{L}$ (that is, the properties we know hold due to the induction hypotheses) imply $s_1 s_2\ \boldsymbol{L}$. That is, if we can prove that the lemma

$$\frac{s\ \boldsymbol{L} \qquad t\ \boldsymbol{L}}{st\ \boldsymbol{L}} \qquad (\mathit{Lemma\text{-}1})$$

  holds for all strings s and t, the third step of our induction proof would be valid.

Again, we can't prove the lemma just by derivation (that is what we just tried for the two strings s_1 and s_2), but we need to inspect s via rule induction. Doing induction over t wouldn't help us anything. After all, in our previous attempt, it wasn't t (i.e., the second string) which was the problem. So, the property $s\ \boldsymbol{P}$ we need to prove is

$$s\ \boldsymbol{P} : \forall t. \frac{s\ \boldsymbol{L} \quad t\ \boldsymbol{L}}{st\ \boldsymbol{L}}$$

There are two rules which may have introduced s into the set $\boldsymbol{L}$, so we have to provide two proofs, one for each rule. Note that, since the property we want to prove is a Lemma with premises itself, the premises of the Lemma become assumtions for our proof (in this case $s\ \boldsymbol{L}$ and $t\ \boldsymbol{L}$). The premise $s\ \boldsymbol{L}$ over the induction variable is not useful in itself (so we don't list it as an assumption), but the reason why $s\ \boldsymbol{L}$ holds: namely, that for the first proof, s is ϵ, and for the second $s = s_1 s_2$, with $s_1\ \boldsymbol{N}$ and $s_2\ \boldsymbol{L}$, is important and necessary for the proof.

- For Rule *L-1* (which means that $s = \epsilon$)

  Proof

  [*A*] $t\ \boldsymbol{L}$
  [*G*] $\epsilon t\ \boldsymbol{L}$

Begin

1. $\{A\}$ $\epsilon t\ \boldsymbol{L}$

End

- For Rule *L-2* (which means that $s = s_1 s_2$, for some $s_1\ \boldsymbol{N}$ and $s_2\ \boldsymbol{L}$, and since $s_2\ \boldsymbol{L}$, the induction hypothesis holds for s_2):

  Proof

  [*A-1*] $s_1\ \boldsymbol{N}$
  [*A-2*] $s_2\ \boldsymbol{L}$
  [*IH*] $\forall k.\dfrac{s_2\ \boldsymbol{L} \qquad k\ \boldsymbol{L}}{s_2 k\ \boldsymbol{L}}$

  [*G*] $\forall k.\dfrac{s_1 s_2\ \boldsymbol{L} \qquad k\ \boldsymbol{L}}{s_1 s_2 k\ \boldsymbol{L}}$

  Begin

  1. $\{\forall$*-introduction, see subproof*$\}$ $\forall k.\dfrac{s_1 s_2\ \boldsymbol{L} \qquad k\ \boldsymbol{L}}{s_1 s_2 k\ \boldsymbol{L}}$

  Subproof

  [*A-3*] $s_1 s_2\ \boldsymbol{L}$
  [*A-4*] $k\ \boldsymbol{L}$
  [*G*] $s_1 s_2 k\ \boldsymbol{L}$

  Begin (Subproof)

$$\dfrac{\dfrac{}{s_1\ \boldsymbol{N}}(A\text{-}1) \qquad \dfrac{\dfrac{}{s_2\ \boldsymbol{L}}(A\text{-}2) \qquad \dfrac{}{k\ \boldsymbol{L}}(A\text{-}4)}{s_2 k\ \boldsymbol{L}}(IH)}{s_1 s_2 k\ \boldsymbol{L}}(L\text{-}2)$$

  End (Subproof)

  End

  We have shown now by induction over the rules for $\boldsymbol{L}$ that *Lemma-1* holds for all s in $\boldsymbol{L}$.

With this lemma, we can now complete the proof for Rule *M-2* to show that $s\ \boldsymbol{M}$ implies $s\ \boldsymbol{L}$:
Proof
[*IH-1*] $s_1\ \boldsymbol{L}$
[*IH-2*] $s_2\ \boldsymbol{L}$
[*G*] $s_1 s_2\ \boldsymbol{L}$
Begin

$$\dfrac{\dfrac{}{s_1\ \boldsymbol{L}}(IH\text{-}1) \qquad \dfrac{}{s_2\ \boldsymbol{L}}(IH\text{-}2)}{s_1 s_2\ \boldsymbol{L}}(Lemma\text{-}1)$$

End
We have now shown that $s\ \boldsymbol{M}$ implies t $s\ \boldsymbol{L}$.

2.3 Examples

2.3.1 Boolean Expressions

As another example for a set defined via mutually recursive rules, consider boolean expressions. For simplicity, we only include three operators: $\wedge$, $\vee$, and $\neg$, the constants `True` and `False`, and parentheses.

Our first attempt at defining a set of inference rules to characterise boolean expressions might look as follows:

$$\frac{}{\texttt{True}\ \textbf{\textit{BExpr}}} \qquad (bool\text{-}1)$$

$$\frac{}{\texttt{False}\ \textbf{\textit{BExpr}}} \qquad (bool\text{-}2)$$

$$\frac{e\ \textbf{\textit{BExpr}}}{\neg e\ \textbf{\textit{BExpr}}} \qquad (bool\text{-}3)$$

$$\frac{e\ \textbf{\textit{BExpr}}}{\texttt{(}e\texttt{)}\ \textbf{\textit{BExpr}}} \qquad (bool\text{-}3)$$

$$\frac{e_1\ \textbf{\textit{BExpr}} \quad e_2\ \textbf{\textit{BExpr}}}{e_1 \wedge e_2\ \textbf{\textit{BExpr}}} \qquad (bool\text{-}4)$$

$$\frac{e_1\ \textbf{\textit{BExpr}} \quad e_2\ \textbf{\textit{BExpr}}}{e_1 \vee e_2\ \textbf{\textit{BExpr}}} \qquad (bool\text{-}4)$$

Unfortunately, with this set of rules, we have the same problem we had with our rules for arithmetic expressions. Even though they inductively define the set of boolean expressions, they are ambiguous and do not reflect associativity and precedence of the operators. If we want to change this, we need to come up with an alternative definition which does not have this ambiguity. The operator $\neg$ has the highest precedence, $\vee$ the lowest, and both $\wedge$ and $\vee$ are left associative.

The solution is also similar to the solution for arithmetic expressions. First, we need rules to define the subset of boolean expressions which can be arguments of the operator with the highest precedence, namely the operator $\neg$. These can only be atomic expressions (constants), any expression in parentheses, or such an expression preceded by negation.

Let's call the subset of boolean expression which can be arguments of negation ***Nexpr***, and the set of all the boolean expressions we generate with the new rules ***Bexpr***. Then the rules for ***Nexpr*** are then as follows:

$$\frac{}{\texttt{True}\ \textbf{\textit{Nexpr}}} \qquad (nexpr\text{-}1)$$

$$\frac{}{\texttt{False}\ \textbf{\textit{Nexpr}}} \qquad (nexpr\text{-}2)$$

$$\frac{e\ \textbf{\textit{Nexpr}}}{\neg e\ \textbf{\textit{Nexpr}}} \qquad (nexpr\text{-}3)$$

$$\frac{e\ \textbf{\textit{Bexpr}}}{\texttt{(}e\texttt{)}\ \textbf{\textit{Nexpr}}} \qquad (nexpr\text{-}4)$$

The rules for the operators $\wedge$ and $\vee$ correspond to those for addition and multiplication. Since the operators are left associative, the expression on the right hand side can only be an expression with stronger cohesion than the one on the left hand side. For the $\wedge$ operator, it has to be a ***Nexpr***.

$$\frac{e_1\ \textbf{\textit{ABExpr}} \quad e_2\ \textbf{\textit{Nexpr}}}{e_1 \wedge e_2\ \textbf{\textit{ABexpr}}} \qquad (abexpr\text{-}1)$$

$$\frac{e_1\ \textbf{\textit{Bexpr}} \quad e_2\ \textbf{\textit{ABexpr}}}{e_1 \vee e_2\ \textbf{\textit{Bexpr}}} \qquad (bexpr\text{-}1)$$

And finally we need rules to express the fact the $\boldsymbol{Nexpr} \subseteq \boldsymbol{ABexpr} \subseteq \boldsymbol{Bexpr}$

$$\frac{e\ \boldsymbol{Nexpr}}{e\ \boldsymbol{ABexpr}} \qquad (abexpr\text{-}2)$$

$$\frac{e\ \boldsymbol{ABexpr}}{e\ \boldsymbol{Bexpr}} \qquad (bexpr\text{-}2)$$

2.3.2 More Arithmetic

We can define addition on natural numbers inductively as relation between three numbers n,m, and k which are in $\boldsymbol{Nat}$, where $n + m = k$.

$$\frac{n\ \boldsymbol{Nat}}{\mathtt{0}{+}n{=}n} \qquad (Add\text{-}1)$$

$$\frac{n{+}m{=}k}{(\mathtt{s}\ n){+}m{=}(\mathtt{s}\ k)} \qquad (Add\text{-}2)$$

The rule

$$\frac{n\ \boldsymbol{Nat}}{n{+}\mathtt{0}{=}n} \qquad (Add\text{-}extra\text{-}1)$$

is not derivable from *Add-1* and *Add-2*, but it is admissible. You can show that this is the case using induction over natural numbers.

3 Syntax

The concrete syntax of a programming language is designed with the user/programmer in mind: it should be well structured and easy to read. The parser checks if a given program adheres to the concrete syntax and translates it into a suitable internal representation. The internal representation is usually quite different from the concrete syntax expression. To demonstrate why, let us go back to the arithmetic expressions example. Consider the following three expressions:

- `1 + 2 * 3`
- `(1) + (2) * (3)`
- `((1)) + (2 * 3)`

Syntactically, all three are different, but semantically, they denote exactly the same computation. Therefore they should ideally have the same internal representation.

If we would have chosen a prefix term representation for the arithmetic expressions instead of an infix notation, we would not have had to worry about the ambiguity of the grammar nor about superfluous parentheses, and we could have defined the language with the following three simple rules:

$$\frac{i \in \mathit{Int}}{(\texttt{Num}\ i)\ \textbf{\textit{expr}}} \tag{e-1}$$

$$\frac{t_1\ \textbf{\textit{expr}} \qquad t_2\ \textbf{\textit{expr}}}{(\texttt{Plus}\ t_1\ t_2)\ \textbf{\textit{expr}}} \tag{e-2}$$

$$\frac{t_1\ \textbf{\textit{expr}} \qquad t_2\ \textbf{\textit{expr}}}{(\texttt{Times}\ t_1\ t_2)\ \textbf{\textit{expr}}} \tag{e-3}$$

and all three arithmetic expressions we listed at the beginning of the chapter correspond to the term `Plus (Num 1) (Times (Num 2) (Num 3))`. Such a term-based syntax is obviously not well suited as concrete syntax of a practical language — it would be a nightmare to write a program in this style. However, it is the appropriate format for the internal representation. A parser, therefore, has to translate expressions of the concrete syntax into terms of the abstract syntax.

We are using the same meta-language to describe the abstract syntax for all our examples. This meta-language is very similar to Haskell data terms. An abstract syntax term t has the form

$$t = (Op\ t_1 ... t_n)$$

where t_i are again abstract syntax terms of that form, or objects of the base types of the language we're describing, for example integers, floating point values, characters, strings, or booleans. The operators *Op* depend on the language we are expressing. In general, we denote them names starting with uppercase letter, and we sometimes omit the outermost parenthesis and write `Plus (Num 1) (Num 3)` instead of `(Plus (Num 1) (Num 3))`.

For the arithmetic expression language, we have three operators: `Num`, `Plus`, and `Times`. Note that while `(Num 3)` is a valid abstract syntax term, just `3` would not be. Every value of basic type has to be wrapped in at least one operator.

These operators correspond closely to Haskell data constructors, so the type `Expr` defined in the following Haskell code snippet has exactly the same terms as elements as ***expr***.:

```
data Expr
   = Num   Int
   | Plus  Expr Expr
   | Times Expr Expr
```

3.1 From Concrete to Abstract Syntax

How can we specify the translation of expressions of concrete syntax into abstract syntax terms? As we discussed previously, inference rules and judgements cannot only be used to define simple properties of single objects, but also relationships between a number of objects. We use inference rules, therefore to define a relationship

$$e\ \textit{\textbf{SExpr}} \longleftrightarrow e'\ \textit{\textbf{expr}}$$

which holds if and only if the (concrete grammar) expression e corresponds to (abstract grammar) expression e'.

We take the inference rules of ***SExpr*** as basis, and add the translation for each case:

$$\frac{e_1\ \textit{\textbf{SExpr}} \longleftrightarrow e_1'\ \textit{\textbf{expr}} \quad e_2\ \textit{\textbf{PExpr}} \longleftrightarrow e_2\ \textit{\textbf{expr}}}{e_1\texttt{+}e_2\ \textit{\textbf{SExpr}} \longleftrightarrow (\texttt{Plus}\ e_1'\ e_2')\ \textit{\textbf{expr}}} \qquad (P\text{-}1)$$

$$\frac{e\ \textit{\textbf{PExpr}} \longleftrightarrow e'\ \textit{\textbf{expr}}}{e\ \textit{\textbf{SExpr}} \longleftrightarrow e'\ \textit{\textbf{expr}}} \qquad (P\text{-}2)$$

$$\frac{e_1\ \textit{\textbf{PExpr}} \longleftrightarrow e_1'\ \textit{\textbf{expr}} \quad e_2\ \textit{\textbf{FExpr}} \longleftrightarrow e_2\ \textit{\textbf{expr}}}{e_1\texttt{*}e_2\ \textit{\textbf{PExpr}} \longleftrightarrow (\texttt{Times}\ e_1'\ e_2')\ \textit{\textbf{expr}}} \qquad (P\text{-}3)$$

$$\frac{e\ \textit{\textbf{FExpr}} \longleftrightarrow e'\ \textit{\textbf{expr}}}{e\ \textit{\textbf{PExpr}} \longleftrightarrow e'\ \textit{\textbf{expr}}} \qquad (P\text{-}4)$$

$$\frac{e\ \textit{\textbf{SExpr}} \longleftrightarrow e'\ \textit{\textbf{expr}}}{\texttt{(}e\texttt{)}\ \textit{\textbf{FExpr}} \longleftrightarrow e'\ \textit{\textbf{expr}}} \qquad (P\text{-}5)$$

$$\frac{i \in \mathit{Int}}{i\ \textit{\textbf{FExpr}} \longleftrightarrow (\texttt{Num}\ i)\ \textit{\textbf{expr}}} \qquad (P\text{-}6)$$

Previously, we used the inference rules to prove that an object had a certain property, e.g., that `1+3` is indeed an ***SExpr***. With relations, inference rules become more interesting. For example, we can view the rules as a description of how to construct a abstract syntax term for a given arithmetic expression, say `1 + 3 * 4`. In this case, we start of with the proof/derivation goal

$$\texttt{1 + 2 * 3}\ \textit{\textbf{SExpr}} \longleftrightarrow\ ???\ \textit{\textbf{expr}}$$

that is, the right hand side of the relation is not fixed yet. To apply the addition rule, however, it has to have the form `Plus` ??. With every rule application, we know more about the exact form of this term, until we finally end up with `Plus (Num 1) (Times (Num 2) (Num 3))`:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{1}\ \textit{\textbf{FExpr}} \longleftrightarrow (\texttt{Num 1})\ \textit{\textbf{expr}}}}{\texttt{1}\ \textit{\textbf{PExpr}} \longleftrightarrow (\texttt{Num 1})\ \textit{\textbf{expr}}}}{\texttt{1}\ \textit{\textbf{SExpr}} \longleftrightarrow (\texttt{Num 1})\ \textit{\textbf{expr}}} \quad \dfrac{\dfrac{\dfrac{}{\texttt{2}\ \textit{\textbf{FExpr}} \longleftrightarrow (\texttt{Num 2})\ \textit{\textbf{expr}}}}{\texttt{2}\ \textit{\textbf{PExpr}} \longleftrightarrow (\texttt{Num 2})\ \textit{\textbf{expr}}} \quad \dfrac{}{\texttt{3}\ \textit{\textbf{FExpr}} \longleftrightarrow (\texttt{Num 3})\ \textit{\textbf{expr}}}}{\texttt{2 * 3}\ \textit{\textbf{PExpr}} \longleftrightarrow (\texttt{Times (Num 2) (Num 3)})\ \textit{\textbf{expr}}}}{\texttt{1 + 2 * 3}\ \textit{\textbf{SExpr}} \longleftrightarrow (\texttt{Plus (Num 1) (Times (Num 2) (Num 3))})\ \textit{\textbf{expr}}}$$

The process of finding, for a given s, s ***SExpr***, a term t, t ***expr***, with s ***SExpr*** $\longleftrightarrow$ t ***expr*** is called parsing. A parser has to be complete, in the sense that for every s ***SExpr*** it should find the corresponding abstract syntax term. Furthermore, it has to be unambiguous, and return for every s ***SExpr*** a unique t. The inverse process is called unparsing. Since each of the parsing rules given above directly corresponds to a production rule of ***SExpr***, it is trivial to show the completeness using rule induction.

We can interpret the inference rules given above also differently, and instead of deriving a term t ***expr*** for a given s, we can start with an abstract syntax term and derive a matching arithmetic expression. This process is called unparsing. Unparsing is, in our example and in general, ambiguous. It is also not necessarily complete, although it is for the arithmetic expression we defined. Going one step further and converting the concrete syntax token sequence back into a string is called pretty printing. Pretty printing is useful, for example, to view intermediate code in a compiler, or in tools which re-format a program. Pretty printing is even more ambiguous than unparsing, since we are free to add spaces and new lines in many places. A pretty printer aims at choosing the most readable representation, which is of course a very subjective measure.

3.2 First Order Abstract Syntax

Let's make our simple expression language slightly more interesting, by adding variable bindings. Variables can be bound in multiple ways. In Haskell, for example, they are bound, among others, in function declarations, where- and let-bindings. In our expression language, we just add the latter. In the Haskell example below, the variable `x` is bound to `3` in the expression `x + 1`:

```
let x = 3
in x + 1
```

To add let-bindings to our expression language, we need to extend the set of rules which define the concrete and abstract syntax.

Concrete Syntax: We define let-expression to act like a parenthesised expression, and to behaves therefore like a ***FExpr***. As with numbers, where we often use i, n or m to denote an integer without further specifying it, we use id from now on to represent an identifier without explicitly stating it as precondition, and without precisely specifying what a legal identifier in the language is:

$$\frac{id \in \mathit{Ident}}{id\ \boldsymbol{FExpr}} \qquad (F\text{-}3)$$

$$\frac{id \in \mathit{Ident} \quad e_1\ \boldsymbol{SExpr} \quad e_2\ \boldsymbol{SExpr}}{\texttt{let}\ id = e_1\ \texttt{in}\ e_2\ \texttt{end}\ \boldsymbol{FExpr}} \qquad (F\text{-}4)$$

We add the `end`-keyword to make nested let-expressions unambiguous and avoid having to use precedence rules, indentation, or parentheses. In the notes, we will omit the `end` keyword for brevity, unless it is necessary to parse an expression correctly.

Abstract Syntax: Variables are represented by terms, similar to numbers, with the exception that the argument of the term is an identifier id which contains the name of the variable. A let-expression is represented by a term which requires three arguments: an identifier (bound variable), a term (the term the variable is bound to), and the body of the let-expression, i.e., the term in which the variable is defined. Since the first argument has to be an identifier, we can drop the

Var operator in the argument position.

$$\frac{}{\texttt{(Var } id\texttt{)}\ \textbf{\textit{expr}}} \qquad (e\text{-}4)$$

$$\frac{t_1\ \textbf{\textit{expr}} \quad t_2\ \textbf{\textit{expr}}}{\texttt{(Let } id\ t_1\ t_2\texttt{)}\ \textbf{\textit{expr}}} \qquad (e\text{-}5)$$

This representation is called first order abstract syntax. Variables and bindings are treated just like any other language construct.

The parsing rule for let-expressions is then:

$$\frac{e_1\ \textbf{\textit{SExpr}} \longleftrightarrow e_1'\ \textbf{\textit{expr}} \quad e_2\ \textbf{\textit{SExpr}} \longleftrightarrow e_2'\ \textbf{\textit{expr}}}{\texttt{let } id = e_1 \texttt{ in } e_2 \texttt{ end}\ \ \textbf{\textit{FExpr}}\ \longleftrightarrow \texttt{(Let } id\ e_1'\ e_2'\texttt{)}\ \textbf{\textit{expr}}} \qquad (P\text{-}7)$$

3.3 Higher-Order Abstract Syntax

In the first order abstract syntax definition of arithmetic expressions, variables are treated just like numbers and represented by terms, although they play a special role.

Higher-order abstract syntax addresses this shortcoming, and provides variables and variable bindings as part of the meta-language: as we described previously, first order terms can have the form $(Op\ t_1\ \dots\ t_n)$, where t_1 to t_n are either again first order terms, or constant values of integers, strings and so on.

```
- (Num 4)

- (Plus (Num 2) (Num 1))

- (Var "x")

- (Let "x" (Num 4) (Var "x"))
```

In addition, a higher-order abstract syntax term can also

- be a variable
- have the form $(x.t)$ meaning the variable x is bound in term t
  - $(x.\texttt{(Plus } x \texttt{ (Num 1)))}$
  - $(x.(y.\texttt{(Plus } x\ y\texttt{)))}$

A term of the form $(x.t)$ is called an abstraction. It is a term whose value depends[1] on the value of the variable x. In this respect, it is similar to a function body. Since this is a different type of variable, a built-in feature of our meta language, we use a different font for it to distinguish from other variables.

An abstraction $(x.t)$ is said to bind all occurrence of x in t. All variables of a term which are not bound at the position they occur are called free variables of that term. We denote the set of free variables of a term by $FV(t)$. It is inductively defined as follows:

$$\begin{array}{ll} FV\ (i) & = \{\},\ i \in Int \\ FV\ (x) & = \{x\} \\ FV\ ((Op\ t_1\ \dots\ t_n)) & = FV(t_1) \cup \dots \cup FV(t_n),\ Op \in \{\texttt{Num}, \texttt{Plus}, \dots\} \\ FV\ ((x.t)) & = FV(t) \setminus \{x\} \end{array}$$

For example, x is in $FV(\texttt{(Plus } x \texttt{ (Let (Num 5) } (x.x)\texttt{)))}$, which corresponds to the concrete syntax expression

[1]More precisely, may depend, since it is possible that x does not actually occur in t, as in $(x.\texttt{(Num 1))}$

```
    x + (let x = 5 in x end)
```

since

$$\begin{aligned} & FV(\texttt{(Plus } x \texttt{ (Let (Num 5) (}x.x\texttt{)))}) \\ = & FV(x) \cup FV(\texttt{(Let (Num 5) (}x.x\texttt{))}) \\ = & FV(x) \cup FV(\texttt{(Num 5)}) \cup FV(\texttt{(}x.x\texttt{)}) \\ = & FV(x) \cup \{\} \cup FV(\texttt{(}x.x\texttt{)}) \\ = & \{x\} \cup \{\} \cup (FV(x) \setminus \{x\}) \\ = & \{x\} \cup \{\} \\ = & \{x\} \end{aligned}$$

If we want to use higher order syntax, we have to change the rules for variables and let-bindings in the definition of the abstract syntax:

$$\frac{}{id\ \textbf{\textit{expr}}} \quad (hos\text{-}1)$$

$$\frac{t_1\ \textbf{\textit{expr}} \quad t_2\ \textbf{\textit{expr}}}{\texttt{(Let } t_1 \texttt{ (}id.t_2\texttt{))}\ \textbf{\textit{expr}}} \quad (hos\text{-}2)$$

and adapt the the translation accordingly:

$$\frac{}{id\ \textbf{\textit{FExpr}} \longleftrightarrow id\ \textbf{\textit{expr}}} \quad (parse\text{-}1)$$

$$\frac{e_1\ \textbf{\textit{SExpr}} \longleftrightarrow t_1\ \textbf{\textit{expr}} \quad e_2\ \textbf{\textit{SExpr}} \longleftrightarrow t_2\ \textbf{\textit{expr}}}{\texttt{let } id\texttt{=}e_1 \texttt{ in } e_2 \texttt{ end}\ \textbf{\textit{FExpr}} \longleftrightarrow \texttt{(Let } t_1 \texttt{ (}id.t_2\texttt{))}\ \textbf{\textit{expr}}} \quad (parse\text{-}2)$$

Now, the operator `let` accepts only two arguments, one being the right hand side, the second the abstraction of the body. Note how a source language variable,id, which can be any string that's a legal identifier in the language, say `"x"`, is translated into x, an identifier in our meta-language. We choose to use one of the same name, so that we don't have to remember which source language identifier corresponds to which meta-language identifier.

3.4 Substitution and α-equivalence

Consider the expression

```
let x = 3 in x + 1
```

and

```
let y = 3 in y + 1
```

Both express exactly the same computation and only differ in the choice of the variable names. They are represented by the term `(Let (Num 3) (`x`.(Plus `x` (Num 1))))` and `(Let (Num 3) (`y`.(Plus `y` (Num 1))))` respectively. If two terms, as in the above example, can be made identical by renaming the variables, they are called α-equivalent, written $\equiv_\alpha$. It can be easily shown that it is indeed an equivalence relation, as it is

1. reflexive: for all terms t, $t \equiv_\alpha t$
2. symmetric: for all terms t_1, t_2, if $t_1 \equiv_\alpha t_2$ then $t_2 \equiv_\alpha t_1$
3. transitive: for all terms t_1, t_2, and t_3: if $t_1 \equiv_\alpha t_2$ and $t_2 \equiv_\alpha t_3$ then $t_1 \equiv_\alpha t_3$

If we want to determine the value of a let-expression, at some point, we have to replace the variable in the body with the value the variable is bound to. This process of replacing a variable with a value, or in general, an arbitrary term, is called substitution. We use the notation:

$$t[x := t']$$

to describe a term t where every free occurrence of x has been replaced by t'. We can rename the variables in a term now by replacing the variable at its binding occurrence, and substituting it wherever it occurs freely in the term:

$$(x.t) \equiv_\alpha (y.(t[x := y])), \quad \text{if } y \notin FV(t) \qquad (\alpha\text{-}equivalence)$$

We have to be careful about the choice of y, though. If we try to rename x to y in the term `(`x`.(Plus` x y`))` we do not want to end up with the term `(`x`.(Plus` y y`))` since the y in the original term is now captured, and the new term is not α-equivalent to the original term anymore. Therefore, we require that the new variable does not occur freely anywhere in the original term.

Let us now give the exact definition of substitution, first for variables:

$$\begin{array}{lcl}
x[x := y] & = & y \\
z[x := y] & = & z, \ \textit{if } x \neq z \\
(Op\ t_1 \dots t_n)[x := y] & = & (Op\ t_1[x := y] \ \dots \ t_n[x := y]) \\
(x.t)[x := y] & = & (x.t) \\
(z.t)[x := y] & = & (z.t[x := y]) \ \textit{if } x \neq z, \ y \neq z, \\
(y.t)[x := y] & = & \textit{undefined if } x \neq y
\end{array}$$

To avoid the problem of capturing, we require that the variable which is introduced does not occur anywhere in the binding position of a term. Similarly, if we substitute arbitrary terms, we require that none of the free variables in the new term u occurs at a binding position in the original term:

$$\begin{array}{lcl}
x[x := u] & = & u \\
z[x := u] & = & z, \ \textit{if } x \neq z \\
(Op\ t_1 \dots t_n)[x := u] & = & (Op\ t_1[x := u] \dots t_n[x := u]) \\
(x.t)[x := u] & = & (x.t) \\
(z.t)[x := u] & = & z.t[x := u], \ \textit{if } x \neq z, \ z \notin FV(u), \\
(y.t)[x := u] & = & \textit{undefined}, \ \textit{if } y \in FV(u)
\end{array}$$

Note that the rules for substituting a variable are just a special case of substitution of a general term, with $u = x$. We can rename the variable such that clashes do not occur to avoid substitution to be undefined. Many compilers will actually rename all the variables defined by the user and map them to unique names to simplify the further compilation steps. We silently assume from now on that the programs we are dealing with have unique names.

4 Semantics

After looking into ways to specify the syntax of programming languages, we now shift our focus to their meaning, or semantics, and examine how semantics can be precisely defined through the use of inference rules. In the study of programming languages, semantics allows us to understand and formally describe what programs do, beyond how they are structured syntactically.

In semantics, we make a key distinction between static semantics and dynamic semantics, both of which play crucial roles in the design and analysis of programming languages.

Static Semantics: This aspect of semantics refers to properties that can be determined at compile-time, without actually executing the program. It includes rules that govern type correctness, variable declarations, and scope management. For instance, static semantics ensures that a function expecting an integer input does not receive a string, enforcing constraints that help catch errors early.

Dynamic Semantics: In contrast, dynamic semantics deals with the behavior of programs during runtime. It defines how programs execute and produce results, detailing the sequence of computational steps that occur as the program runs. Through dynamic semantics, we specify how different constructs—like loops, conditionals, and expressions—behave and interact as they are executed.

We will look into both static and dynamic semantics, exploring how inference rules can formalize these aspects, providing a rigorous framework for understanding the meaning of programs.

4.1 Static Semantics

Static semantics includes all those semantic properties which can be checked at compile time. What these properties actually are, and to which extent a compiler (or some other tool) is able to extract information about them depends on the programming language. Often, they are either related to the scoping or the typing rules of a language. The type of an expression or value tells us something about their properties, and, most importantly, which operations can be applied to them. Essentially, types are sets of values and expressions which share these properties. In some languages, the type of an expression or the scope of a variable can only be determined during runtime, so this is not part of the language's static semantics.

In our language of arithmetic expressions, typing rules are pretty pointless, since we only have a single type, *Int*, all operators expect values of type *Int*. So the only interesting static semantic feature of this language is the scoping of the variables. The expression

```
let x = x + 1
in x * x
```

is syntactically correct, but not semantically, since `x` in the subexpression `x + 1` is not bound to a value in our language (in Haskell, which has different scoping rules, this expression is fine).

4.1.1 Scope of a Variable

In our expression language, given an expression `let x =` t_1 `in` t_2 `end`, the variable `x` is bound in t_2, but not in t_1. The part of the program or expression where a variable can be used is called the scope of a variable, that is t_2 is the scope of `x`, and `x` is bound to the value of t_1 everywhere in t_2. Scoping rules are usually part of the static semantics of a programming language, which we will discuss in Chapter 4.

When a variable with the same name is bound twice, shadowing can occur. in the example below, the outer binding of x is shadowed by the inner binding, and the value of the expression is 10.

```
let x = 3
in let x = 5
   in x + x
   end
end
```

Programming languages differ in how they determine the scope of a variable. The vast majority of languages use static scoping or lexical scoping. This means syntactic rules which can be checked by the compiler at compile-time determine whether a variable is in scope and where it is bound.

With dynamic scoping, the scope and binding is determined by the runtime context. For example, consider this pseudo-code snippet in a C-like language:

```
int x = 5;

int f () {
 return x;
}

int g () {
  int x = 10;
  return (f ());
}
```

In a statically scoped language, the variable `x` in function `f` refers to the global variable `x` which is initialised with the value 5. So the function `g` would return `5`. In a dynamically scoped language, the closest binding of the variable `x` in `f` if called via `g` would be the local variable `x` in `g`, so calling `g` would return `10`.

Note that the scope of a variable in a let-binding is defined differently in Haskell than in our toy language: in Haskell, the scope of the variable `x` in

```
let x = t₁ in t₂
```

is t_1 as well as t_2. As a consequence, the compiler accepts an expression like the one below, where the variable bound on the left hand side occurs on the right hand side:

```
let xs = 1 : xs
in take 10 xs
```

Evaluating this expression in Haskell results in a list of length ten of `1`'s, and the infinite list `xs` is never constructed.

Many languages distinguish whether a variable is visible in a block (if the language has the concept of a block), a method or function, a class (again, for languages which have this concept), or globally. Object oriented languages allow the programmer to control whether a variable or method is visible outside of a class via keywords like `private` or `public`. Similarly, languages can control which functions, variables, or data type a module exports.

Programming languages also differ in how they define the lexical scope. For example, in C, the scope of a variable in a block only starts after the variable is declared, whereas in JavaScript when using `var`, a variable is in scope in the whole function. In C, the following function is fine

```
int x = 10;
int f () {
  printf("%d\n", x); // print the value of x
  return 0;
}

int main() {
  f();
}
```

In contrast, this program

```
int f () {
  printf("%d\n", x); // print the value of x
  return 0;
}
int x = 10;
int main() {
  f();
}
```

triggers a compile time error, because the declaration comes before the use:

```
example.c:5:18: error: use of undeclared identifier 'x'
  printf("%d\n", x);
```

JavaScript, however, it is fine to declare it later, and this sample program prints `10`.

```
    function g() {
        console.log(x)
    }
    let x = 10
    g()
```

Even if the variable is declared after the function `g` is called, it does not trigger an error (but since it's not yet initialised, the output is `null`)

4.1.2 Static Semantic Rules for MinHs

For our arithmetic expression language, we would like to have a set of inference rules which define a predicate ***ok*** which holds for a given expression e if it is correct in the sense that it contains no free variables at all, therefore e ***ok***. Since we only have the type *Int*, we also know that e has this type.

As we know from the definition of the abstract syntax, the expression can either be a number, a variable, an addition, a multiplication, or a let-binding. Therefore, we need an inference rule for each of these cases stating under which conditions the expression is *ok*. For expressions representing simple numbers, it is straight forward, as they never contain a free variable, and we might write:

$$\frac{}{(\texttt{Num}\ i)\ \textbf{\textit{ok}}}$$

Let a look at let-expressions: the term $(\texttt{Let}\ e_1\ (x.e_2))$ is ***ok*** if e_1 is ***ok***. What about e_2, though? It may contain free occurrences of x. So, to determine is e_2 is ***ok***, we have to somehow remember that x is bound in e_2. It seems it is not sufficient to define ***ok*** depending solely on the expression, but we need a way to keep track of the variables which are bound in the expression.

This can be done by adding an environment Γ (in this case, a set containing just variable names) to our judgment, and write it as

$$\Gamma \vdash e\ \textbf{\textit{ok}}$$

to express e is ***ok*** under the assumption that the variables in Γ are defined.

Since number expressions contain no variables, they are ***ok*** under any environment Γ:

$$\frac{}{\Gamma \vdash \texttt{(Num}\ i\texttt{)}\ \textbf{\textit{ok}}} \quad (\textit{ok-1})$$

Addition and multiplication are ***ok*** under an environment Γ if their subexpressions are ***ok*** under the same enviroment:

$$\frac{\Gamma \vdash t_1\ \textbf{\textit{ok}} \qquad \Gamma \vdash t_2\ \textbf{\textit{ok}}}{\Gamma \vdash \texttt{(Plus}\ t_1\ t_2\texttt{)}\ \textbf{\textit{ok}}} \quad (\textit{ok-2})$$

$$\frac{\Gamma \vdash t_1\ \textbf{\textit{ok}} \qquad \Gamma \vdash t_2\ \textbf{\textit{ok}}}{\Gamma \vdash \texttt{(Times}\ t_1\ t_2\texttt{)}\ \textbf{\textit{ok}}} \quad (\textit{ok-3})$$

We can now also handle the case where the expression consists of a single variable: the expression is ***ok*** only if the variable is already in the environment.

$$\frac{x \in \Gamma}{\Gamma \vdash x\ \textbf{\textit{ok}}} \quad (\textit{ok-4})$$

In case of a let-binding, we first check the right-hand side of the binding with the old environment, and then insert the new variable into the environment and check the body expression:

$$\frac{\Gamma \vdash t_1\ \textbf{\textit{ok}} \quad \Gamma \cup \{x\} \vdash t_2\ \textbf{\textit{ok}}}{\Gamma \vdash \texttt{(Let}\ t_1\ \texttt{(}x\texttt{.}t_2\texttt{))}\ \textbf{\textit{ok}}} \quad (\textit{ok-5})$$

In Haskell, where the bound variable can occur in the binding itself, the rule would be somewhat different:

$$\frac{\Gamma \cup \{x\} \vdash t_1\ \textbf{\textit{ok}} \quad \Gamma \cup \{x\} \vdash t_2\ \textbf{\textit{ok}}}{\Gamma \vdash \texttt{(Let}\ t_1\ \texttt{(}x\texttt{.}t_2\texttt{))}\ \textbf{\textit{ok}}} \quad (\textit{Haskell-scoping})$$

Initially, the environment is empty, and an arithmetic expression is ***ok*** if and only if we can derive $\{\} \vdash e\ \textbf{\textit{ok}}$ using the rules *ok-1* to *ok-5*

With these rules, we can show that, for example,

```
let x = 4 in x + x
```

is a legal arithmetic expression according to our definition:

$$\frac{\dfrac{}{\{\} \vdash \texttt{(Num 4)}\ \textbf{\textit{ok}}}(\textit{ok-1}) \qquad \dfrac{\dfrac{}{\{x\} \vdash x\ \textbf{\textit{ok}}}(\textit{ok-4}) \qquad \dfrac{}{\{x\} \vdash x\ \textbf{\textit{ok}}}(\textit{ok-4})}{\{x\} \vdash \texttt{(Plus}\ x\ x\texttt{)}\ \textbf{\textit{ok}}}(\textit{ok-2})}{\{\} \vdash \texttt{(Let (Num 4) (}x\texttt{.(Plus}\ x\ x\texttt{)))}\ \textbf{\textit{ok}}}(\textit{ok-5})$$

However, if we introduce a free variable, such as `y` in the body of the binding here:

```
let x = 4 in x + y
```

we cannot derive $\{\} \vdash \texttt{(Let (Num 4) (}x\texttt{.(Plus}\ x\ y\texttt{)))}\ \textbf{\textit{ok}}$, since we would have a proof obligation $\{x\} \vdash y\ \textbf{\textit{ok}}$, for which we have no rule.

4.2 Dynamic Semantics

The dynamic semantics of a programming language connect the syntax of the language to some kind of computational model. There are different techniques to describe the dynamic semantics of programming languages: axiomatic semantics, denotational, and operational semantics.

Axiomatic semantics specify the semantics of a language in terms of rules describing how a program affects the state, and what it computes. In Hoare-logic, for example, the semantics of the individual language constructs s is given in form of a triple

$$\{P\}s\{Q\}$$

which states that, if the preconditons P hold, and s is executed, the postconditions Q will hold. Axiomatic semantics is often used for program verification.

Denotational semantics describe the semantics of a language by mapping the language's constructs to some mathematical calculus, for example to the λ-calculus, or fix-points over complete partial orders.

In this course, we only use operational semantics, that is, we specify how programs are being executed, as operational semantics help us to understand how languages can be implemented.

Small Step Operational Semantics, also called Structured Operational Semantics (SOS) or Transitional Semantics achieves this by defining an abstract machine and step-by-step execution of a program on this abstract machine. Big Step Semantics or Natural Semantics specifies the semantics of a program in terms of the results of the complete evaluation of subprograms.

4.2.1 Small Step or Structural Operational Semantics (SOS)

Let us start by giving the SOS for the arithmetic expression example. We first have to define a transition system, which can be viewed as a abstract machine together with a set of rules defining how the state of the machine changes during the evaluation of a program. To be more precise, we need to define:

- a set of states S on an abstract computing device
- a set of initial states $I \subseteq S$
- a set of final states $F \subseteq S$
- a relation $\mapsto \subseteq S \times S$ [1] describing the effect of a single evaluation step on state s.

A machine can start up in any of the initial states, and the execution terminates if the machine is in a final state. That is, for $s_f \in F$, there is no $s \in S$ such that $s_f \mapsto s$.

According to this notation $s_1 \mapsto s_2$ can be read as state s_1 evaluates to s_2 in a single execution step. A execution sequence is simply a sequence of states s_0, s_1, s_2, ..., s_n where $s_0 \in I$ and $s_0 \mapsto s_1 \mapsto s_2 \mapsto \ldots \mapsto s_n$

We say that a execution sequence is maximal if there is no s_{n+1} such that $s_n \mapsto s_n + 1$, and complete complete, if s_n is a final state.

Note that, although every complete execution sequence is maximal, not every maximal sequence is complete, as there may be states for which no follow up state exist, but which are not in F. Such a state is called a stuck state and in some sense, it corresponds to undefined behaviour in a program. Obviously, stuck states should be avoided, and transition systems defined in a way that stuck states cannot be reached from any initial state.

How should S, I, F and $\mapsto$ be defined for our arithmetic expressions? If we evaluate arithmetic expressions, we simplify them step-wise, until we end up with the result. We can define our transition system similarly.

[1] $\times$ denotes the carthesian product of two sets: $A \times B = \{(a, b) \mid a \in A, b \in B\}$

The Set of States S: We include all syntactically correct arithmetic expressions without free variables:

$$S = \{e \mid \{\} \vdash e\ \textbf{\textit{ok}}\}$$

The Set of Initial States I: In this case, the set of initial state can be the same as the set of all states:

$$I = \{e \mid \{\} \vdash e\ \textbf{\textit{ok}}\}$$

The Set of Final States F: as every expression should be evaluated to a number eventually, we define the set of final states

$$F = \{(\texttt{Num}\ n)\}$$

Operations The next step is to determine the operations of the abstract machine. For all the arithmetic operations like addition and multiplication, we need the corresponding "machine" operation. To evaluate let-bindings, we also have to be able to replace variables in an expression by a value, so we add substitution of variables to the set of operations our abstract machine can perform. Substitution is a reasonable complex operation requiring the traversal of the whole expression, so by assuming substitution is an operation of the machine means that it is pretty far from a realistic machine. We will later look at alternative definition of abstract machines which do without substitution.

The $\mapsto$-Relation Finally, we do have to define the $\mapsto$-relation inductively over the structure. We do not have to provide any rules for terms of the form $(\texttt{Num}\ n)$, since they represent final states. We start with the evaluation of addition. Keep in mind that $\mapsto$ only describes a single evaluation step of the machine. If both arguments of `Plus` are already fully evaluated, we can simply add the two values using the "machine addition":

$$\frac{}{(\texttt{Plus}\ (\texttt{Num}\ n)\ (\texttt{Num}\ m)) \mapsto (\texttt{Num}\ (n+m))} \quad (\mapsto\text{-}1)$$

What should happen if the arguments are not yet fully evaluated? We have to decide which argument to evaluate first — it does not really matter. So, we start with the leftmost:

$$\frac{e_1 \mapsto e_1'}{(\texttt{Plus}\ e_1\ e_2) \mapsto (\texttt{Plus}\ e_1'\ e_2)} \quad (\mapsto\text{-}2)$$

This step is repeated until the first argument is fully evaluated, at which point be continue with the second argument:

$$\frac{e_2 \mapsto e_2'}{(\texttt{Plus}\ (\texttt{Num}\ n)\ e_2) \mapsto (\texttt{Plus}\ (\texttt{Num}\ n)\ e_2')} \quad (\mapsto\text{-}3)$$

Multiplication works in exactly the same way:

$$\frac{}{(\texttt{Times}\ (\texttt{Num}\ n)\ (\texttt{Num}\ m)) \mapsto (\texttt{Num}\ (n*m))} \quad (\mapsto\text{-}4)$$

$$\frac{e_1 \mapsto e_1'}{(\texttt{Times}\ e_1\ e_2) \mapsto (\texttt{Times}\ e_1'\ e_2)} \quad (\mapsto\text{-}5)$$

$$\frac{e_2 \mapsto e_2'}{(\texttt{Times}\ (\texttt{Num}\ n)\ e_2) \mapsto (\texttt{Times}\ (\texttt{Num}\ n)\ e_2')} \quad (\mapsto\text{-}6)$$

Let-bindings are slightly more interesting. Again, we have to make up our mind in which order we want to evaluate the arguments. If we evaluate the first argument (i.e., the right-hand side of the binding) first and then replace all occurrences of the variable by this value, we have the following rules:

$$\frac{e_1 \mapsto e_1'}{(\texttt{Let}\ e_1\ (x.e_2)) \mapsto (\texttt{Let}\ e_1'\ (x.e_2))} \quad (\mapsto\text{-}7)$$

$$\frac{}{(\texttt{Let}\ (\texttt{Num}\ n)\ (x.e)) \mapsto e[x := (\texttt{Num}\ n)]} \quad (\mapsto\text{-}8)$$

Alternatively, we could have decided to replace the variable immediately with the expression e_1 in e_2, and specify a lazy evaluation order.

$$\frac{}{(\texttt{Let}\ e_1\ (x.e_2)) \mapsto e_2[x := e_1]} \quad (\mapsto\text{-}lazy)$$

If x occurs in the expression e_2 multiple times, it means, however, that we copy the expression, and consequently have to evaluate it more than once[2]. Since the evaluation of arithmetic expressions always terminates (there are no loops or recursion), it does not affect the result, just the number it takes to get to the final result.

Example Let us now have a look at how the evaluation of a expression proceeds. We just write down the sequence of steps, not the full derivation for each step:

$$\begin{array}{ll} & (\texttt{Let}\ (\texttt{Plus}\ (\texttt{Num}\ 5)\ (\texttt{Num}\ 3))(x.(\texttt{Times}\ x\ (\texttt{Num}\ 4)))) \\ \mapsto & \\ & (\texttt{Let}\ (\texttt{Num}\ 8)\ (x.(\texttt{Times}\ x\ (\texttt{Num}\ 4)))) \\ \mapsto & \\ & (\texttt{Times}\ (\texttt{Num}\ 8)\ (\texttt{Num}\ 4)) \\ \mapsto & \\ & (\texttt{Num}\ 32) \end{array}$$

More Notation We use the relation $s_1 \overset{\star}{\mapsto} s_2$ to denote that a state s_1 evaluates to a state s_2 in zero or more steps. In other words, the relation $\overset{\star}{\mapsto}$ is the reflexive, transitive closure of $\mapsto$. That is

$$\frac{}{s \overset{\star}{\mapsto} s} \quad (\overset{\star}{\mapsto}\text{-}1)$$

$$\frac{s_1 \mapsto s_2 \quad s_2 \overset{\star}{\mapsto} s_3}{s_1 \overset{\star}{\mapsto} s_3} \quad (\overset{\star}{\mapsto}\text{-}2)$$

Furthermore, we write $s_1 \overset{!}{\mapsto} s_2$ to express that s_1 fully evaluates in zero or more steps to a state s_2, or more formally,

$$s \overset{!}{\mapsto} s' \equiv s \overset{\star}{\mapsto} s' \text{ and } s' \in F \quad (\overset{!}{\mapsto}\text{-}1)$$

We write $s \overset{n}{\mapsto} s'$ if s evaluates to s' in n steps:

$$\frac{}{s \overset{0}{\mapsto} s} \quad (\overset{n}{\mapsto}\text{-}1)$$

$$\frac{s_1 \mapsto s_2 \quad s_2 \overset{n}{\mapsto} s_3}{s_1 \overset{n+1}{\mapsto} s_3} \quad (\overset{n}{\mapsto}\text{-}2)$$

[2]Actual lazy languages, like Haskell, avoid this, as we will see in Chapter 10

In the previous subsection, we said that stuck states should be avoided. That is, every maximal evaluation sequence should result in a final state. Is this the case for our definitions? Well, the only two types of expressions for which there are no further evaluation rules are numbers and variables. Now, numbers are not a problem, since they are a final state. But what about variables? The answer is that during evaluation, we should never have to evaluate a variable, because all bound variables get replaced by the value they are bound to by substitution, and a statically correct program does not contain any free variables. We can show this formally by showing that evaluation of a legal expression will never lead to an illegal expression. This property is called preservation, and we will discuss it in more detail later, in the context of type safety. 1

4.2.2 Big Step Semantics

Let us now look at the big step semantics. Similar to the initial states in SOS, we have to define a set of evaluable expressions E, a set of values (corresponding to the final states in SOS), which can, but do not have to be a subset of E. Finally, we define a relation "evaluates to" $\Downarrow \subseteq E \times V$. In contrast to SOS, the relation does not say anything about the number of steps the evaluation requires.

Applied to our example, we define

- the set E of evaluable expressions to be : $\{e \mid \{\} \vdash e\ \textbf{\textit{ok}}\}$
- the set V of values: $\{(\mathtt{Num}\ n)\}$

To define $\Downarrow$, we again have to consider all possible cases for e. This time, we do need rules for fully evaluated expressions, that is, in case of our simple language, for expressions of the form $(\mathtt{Num}\ n)$, but they are simple axioms, as they fully evaluate to themselves.

$$\frac{}{(\mathtt{Num}\ n)\Downarrow(\mathtt{Num}\ n)} \qquad (\Downarrow\text{-}1)$$

On the other hand, we only need a single rule for each addition and multiplication, as big step semantics does not specify which of the arguments is evaluated first, since the evaluation of the two expressions is independent:

$$\frac{e_1\Downarrow(\mathtt{Num}\ n_1) \qquad e_1\Downarrow(\mathtt{Num}\ n_2)}{(\mathtt{Plus}\ e_1\ e_2)\Downarrow(\mathtt{Num}\ (n_1+n_2))} \qquad (\Downarrow\text{-}2)$$

$$\frac{e_1\Downarrow(\mathtt{Num}\ n_1) \qquad e_1\Downarrow(\mathtt{Num}\ n_2)}{(\mathtt{Times}\ e_1\ e_2)\Downarrow(\mathtt{Num}\ (n_1*n_2))} \qquad (\Downarrow\text{-}3)$$

The following rule for the let-binding, however, states that e_1 has to be evaluated first, because the variable is replaced by the resulting value, and therefore we have a data dependency:

$$\frac{e_1\Downarrow(\mathtt{Num}\ n_1) \quad e_2[x := (\mathtt{Num}\ n_1)]\Downarrow(\mathtt{Num}\ n_2)}{(\mathtt{Let}\ e_1\ (x.e_2))\Downarrow(\mathtt{Num}\ n_2)} \qquad (\Downarrow\text{-}4)$$

Now consider the example expression used to demonstrate evaluation using the SOS rules. Not surprisingly, the expression evaluates to the same value using big step semantics. Here is the derivation (without rule annotations for space reasons):

$$\frac{\dfrac{\dfrac{}{(\mathtt{Num}\ 5)\Downarrow(\mathtt{Num}\ 5)} \qquad \dfrac{}{(\mathtt{Num}\ 3)\Downarrow(\mathtt{Num}\ 3)}}{(\mathtt{Plus}\ (\mathtt{Num}\ 3)\ (\mathtt{Num}\ 5))\Downarrow(\mathtt{Num}\ 8)} \qquad \dfrac{\dfrac{}{(\mathtt{Num}\ 8)\Downarrow(\mathtt{Num}\ 8)} \qquad \dfrac{}{(\mathtt{Num}\ 4)\Downarrow(\mathtt{Num}\ 4)}}{(\mathtt{Times}\ (\mathtt{Num}\ 8)\ (\mathtt{Num}\ 4))\Downarrow(\mathtt{Num}\ 32)}}{(\mathtt{Let}\ (\mathtt{Plus}\ (\mathtt{Num}\ 5)\ (\mathtt{Num}\ 3)))(x.(\mathtt{Times}\ x\ (\mathtt{Num}\ 4)))\Downarrow(\mathtt{Num}\ 32)}$$

The concept of an evaluation sequence does not make sense for big step semantics, as expressions evaluate in a single "step".

4.2.3 Comparing SOS and Evaluation Semantics

We gave two different set of rules, both describing in different ways the operational semantics of the same language. SOS is more detailed, as it specifies each evaluation step, and also the evaluation order of independent subexpressions, as for example for addition and multiplication for our example language, which is often unnecessary.

Small step semantics, in contrast to big step, can tell us something about non-terminating programs. We can observe how the evaluation proceeds, and maybe how it effects some state in the process, whereas big step semantics does not give us much information about non-terminating evaluations.

In our example, we showed that, at least for one example expression, both lead to the same result. But how can we prove that the small step and the big step semantics are equivalent, that is, that for all expressions e of a language, we have $e \overset{!}{\mapsto} e'$ if and only if $e \Downarrow e'$?

We can do this in two parts, by showing that

1. $e \Downarrow$ (Num n) implies $e \overset{!}{\mapsto}$ (Num n), and
2. $e \overset{!}{\mapsto}$ (Num n) implies $e \Downarrow$ (Num n)

in two separate proofs.

Part I: $e \Downarrow$ (Num n) implies $e \overset{!}{\mapsto}$ (Num n)

We can prove this using rule induction. Since there are four rules for $\Downarrow$ and we need to consider each of these rules as a separate case.

- Rule ($\Downarrow$-*1*): $e =$ (Num n').

  Proof

  [A] (Num n')$\Downarrow$(Num n)

  [G] (Num n') $\overset{!}{\mapsto}$ (Num n)

  Begin

  1. { A, Rule ($\Downarrow$-*1*) } $n = n'$
  2. {$\overset{n}{\mapsto}$-*1*} (Num n) $\overset{0}{\mapsto}$ (Num n)
  3. {$\overset{\star}{\mapsto}$-*1*, and 2.} (Num n) $\overset{\star}{\mapsto}$ (Num n)
  4. {$\overset{!}{\mapsto}$-*1*, 3., and (Num n) $\in F$ }: (Num n) $\overset{!}{\mapsto}$ (Num n)

  End

- Rule ($\Downarrow$-*2*): $e =$ (Plus e_1 e_2)

  Proof

  [A] (Plus e_1 e_2)$\Downarrow$(Num ($n_1 + n_2$))

  [*IH-1*] $e_1 \overset{!}{\mapsto} n_1$

  [*IH-2*] $e_2 \overset{!}{\mapsto} n_2$

  [G] (Plus e_1 e_2) $\overset{!}{\mapsto}$ (Num ($n_1 + n_2$))

  Our problem here is that we cannot directly evaluate (Plus e_1 e_2) any further, since, depending on what e_1 and e_2 look like, different single step semantics rules apply. We do know what the two subexpressions evaluate to in the single step semantics, namely (Num n_1) and (Num n_2), respectively. In fact, the two lemmata below hold for our specific semantics. Even though they might seem obviously true, we cannot assume this without a proof.

$$\frac{e_1 \overset{\star}{\mapsto} e_1'}{\texttt{(Plus } e_1\ e_2\texttt{)} \overset{\star}{\mapsto} \texttt{(Plus } e_1'\ e_2\texttt{)}} \qquad (\textit{Lemma-4.3.1})$$

$$\frac{e_2 \overset{\star}{\mapsto} e_2'}{\texttt{(Plus (Num } n\texttt{) } e_2\texttt{)} \overset{\star}{\mapsto} \texttt{(Plus (Num } n\texttt{) } e_2'\texttt{)}} \qquad (\textit{Lemma-4.3.2})$$

These two lemmas can be proven correct by induction over the length of the evaluation sequence (the proofs themselves are straight forward, so we do not show them here). With the help of these two lemmata, we can apply the induction hypotheses to show that the small and big step semantics rules lead to the same result:

Begin

1. {see Subproof} $\texttt{(Plus } e_1\ e_2\texttt{)} \overset{\star}{\mapsto} \texttt{(Num } n_1 + n_2\texttt{)}$

   Begin (Subproof)

   $\texttt{(Plus } e_1\ e_2\texttt{)}$

   $\overset{\star}{\mapsto}$ {*I.H.-1, Lemma-4.3.1*}

   $\texttt{(Plus (Num } n_1\texttt{) } e_2\texttt{)}$

   $\overset{\star}{\mapsto}$ {*I.H.-2, Lemma-4.3.2*}

   $\texttt{(Plus (Num } n_1\texttt{) (Num } n_2\texttt{))}$

   $\mapsto$ {*def of* $\mapsto$}

   $\texttt{(Num } n_1 + n_2\texttt{)}$

   End (Subproof)

2.

3. $\{\overset{!}{\mapsto}$-*1*, 1., and $\texttt{(Num } n_1 + n_2\texttt{)} \in F$ $\}$ $\texttt{(Plus } e_1\ \ e_2\texttt{)} \overset{!}{\mapsto} \texttt{(Num } (n_1 + n_2)\texttt{)}$

   End

- Rule ($\Downarrow$-*3*): $e = \texttt{(Times } e_1\ \ e_2\texttt{)}$: The proof is almost the same as the proof for `Plus`, and we need the corresponding lemmata.
- Rule ($\Downarrow$-*4*): $e = \texttt{(Let } e_1\ \texttt{(}x\texttt{.}e_2\texttt{))}$: The proof is again similar to the first. This time, we only need a lemma for the first argument of `Let`.

Part II: $e \overset{!}{\mapsto} \texttt{(Num } n\texttt{)}$ implies $e \Downarrow \texttt{(Num } n\texttt{)}$: For the opposite direction, rule induction over $\mapsto$ does not work directly, but we can do induction over the length k of the evaluation sequence (which corresponds to rule induction over the $\overset{!}{\mapsto}$ and $\mapsto$ together). For these proofs, the lemmas we used in for the Part I are useful as well,

4.3 The Interaction between Static and Dynamic Semantics

The dynamic semantics of our language maps expressions in ***FExpr*** to numerical values. But it is a partial mapping, as there are some expressions e in ***FExpr*** for which the dynamic semantics does not provide a value. For example, the expression

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

contains a free variable, so while it is in ***FExpr***, we cannot derive that it is in ***ok***. The expression is grammatically correct, but it doesn't have a meaning. However, the dynamic semantics is a total mapping for all expressions for which we can derive ***ok***, and we can show by induction for the big step sematics that

$$\forall e.\ e\ \textit{\textbf{ok}} \Rightarrow \exists n.\ e \Downarrow (\texttt{Num}\ n)$$

and therefore, since they are equivalent

$$\forall e.\ e\ \textit{\textbf{ok}} \Rightarrow \exists n.\ e \overset{!}{\mapsto} (\texttt{Num}\ n)$$

In other words, every expression which is correct according to the static semantics evaluates to a value in the dynamic semantics. This is only true because our language is so simple that every computation terminates, as there are no loops and no recursion. For Turing complete languages, however, it is impossible to have a decidable static semantics which gives this guarantee, but we can still give some useful guarantees, as we will see in Chapter 8.

5 | MinHs – a functional core language

We begin our discussion of programming languages with MinHs, a stripped-down, Haskell-like version of a strict, purely functional language. But what actually characterises a purely functional language and distinguishes it from imperative, procedural, and object-oriented languages? As the name suggests, functions are the central concept.

In purely functional languages, functions are first-class citizens: they can be passed around as arguments, and functions can return other functions as results, just like any other values. Functions that return other functions as results or accept functions as arguments are called higher-order functions.

Pure functions in these languages are like mathematical functions: given a set of input values, a function returns a result that depends only on those inputs, not on any external state. Consequently, calling a function multiple times with the same arguments will always yield the same result. If we know the result of a function, we can replace any call to that function with its result throughout the program without changing the program's behavior. This property is known as referential transparency.

Variables in a functional program also resemble mathematical variables. They represent potentially unknown but fixed values. In contrast, variables in non-functional languages typically represent memory locations, whose contents may change during execution.

Referential transparency has some valuable consequences. Just as in mathematics, we can use equational reasoning with programs, showing, for example, that two different implementations of a function are equivalent. Purely functional languages also lend themselves well to parallel and concurrent programming, as shared implicit state can lead to issues like race conditions and synchronization problems.

But if functional programs cannot manipulate or observe implicit state, how can they perform useful tasks, like input/output (I/O)? By definition, I/O operations depend on and alter the state of I/O devices. To address this, purely functional languages make the state explicit by passing it as an argument. For example, consider a function that reads a line from standard input. In an impure language, this function would require no arguments. In Python, for instance, executing the `input` function with the same prompt could yield different strings each time, depending on user input.

```
name = input("Please enter your name: ");
```

The work-around in a functional language is to make the state explicit in the type. The function `getLine` takes the state of the world as input, and returns a new world, and a string. That is, conceptually, it has type

```
getLine :: World → (World, String)
```

Now, at least in theory, everyting is pure again. Given exactly the same world, with the same users and devices, all in the same state, the function returns the same new world, and string. The new problem is of course that the world cannot actually be represented as argument. Even if it (or the relevant part of it) was, we certainly don't want the user to be able to copy it and be able to observe to the previous and the new world. There are several solutions to this problem. Haskell, for example, wraps this type in an abstract data type, the `IO` type, and provides the user with a limited, controlled interface to the type, so that parts of the world can be observed and manipulated, but the world can never be accessed directly. Essentially (and somewhat simplified), `IO` is a type synonym:

```
data IO a = IO (World → (World, a))

getLine :: IO String         -- returns a new world and a string
putStr  :: String → IO ()    -- returns just a new world
```

All we can do in Haskell is plugging IO-functions together, usually via the do-notation

```
main = do
  name <- getLine
  putStr ("Your name is " ++ name ++ "\n")
```

Here, `do` takes the functions `getLine` and `putStr`, and combines them to a new function that, when applied to the world, will read a string, create a new string, and print that to standard out. When the program is run, this function is applied to the world once. Consequently, there is no pure function of type

```
IO a → a
```

because once a value depends on the state of the world, it can't be converted to a pure value anymore. This feels like cheating, because the world is of course never passed around. Internally, during execution, it uses the same non-pure IO operations all other languages use. However, as long as this is not observable in the program, it does not matter, as it doesn't break the rules of referential transparency. Purity in a functional language is just a matter of abstraction. Even evaluating a simple, seemingly pure arithmetic expression, does have side effects in reality – it will change the state of the registers and many other things.

5.1 Concrete Syntax

But now let's get started with defining our own functional language, MinHs. We give the concrete syntax of MinHs in BNF. Note that the definition is ambiguous, but the usual precedences and associativities apply for arithmetic and boolean operations (left associative) and application (left associative). The type constructor → is right associative:

Variables	***Ident***	::=	...
Integer values	***Int***	::=	...
Boolean values	***Bool***	::=	`True` | `False`
Types	***Type***	::=	`Bool` | `Int` | ***Type*** → ***Type*** | `(` ***Type*** `)`
Infix Operators	⊗	::=	`+` | `*` | `-` | `=` | `<` | `>` | `>=` | `<=`
Expressions	***Expr***	::=	***Ident*** | ***Int*** | ***Bool*** | `(` ***Expr*** `)` | ***Expr*** ⊗ ***Expr*** | ***Expr*** ***Expr***
			| `if` ***Expr*** `then` ***Expr*** `else` ***Expr***
			| `recfun` ***Ident*** `::` `(` ***Type*** → ***Type***`)` ***Ident*** `=` ***Expr***

In MinHs, we have two base types: `Int` and `Bool`, and a binary type constructor →, which denotes function types. As the function type constructor is right associative, the type `Int → Bool → Int` is the same as `Int → (Bool → Int)`: the type of a function which, given an `Int` value, returns a function from `Bool` to `Int`. The left-associativity of application means that `f 3 4` is the same as `(f 3) 4`.

Compared to our arithmetic expression language, we now have three new language constructs: if-expressions, which behave in the usual way, (recursive) function definitions and applications.

These seemingly minor additions have a significant effect, as they make the language Turing complete.

Function definitions in MinHs are slightly unusual, and closer to a compiler internal representation (which is good for our purpose).

So, let us have a more detailed look at them. The expression

$$\texttt{recfun}\ f :: (\tau_1 \rightarrow \tau_2)\ x = e$$

defines a function named f, which accepts a single argument x of type τ_1 and returns a value of type τ_2. The type of the function (and therefore, implicitly, also the type of the argument variable) has to be provided by the programmer. This makes Minhs an explicitly typed language, like for example Java or C. Haskell, like C# and Python, in contrast, is implicitly typed: the programmer doesn't have to add type annotations, but the compiler will infer the type itself. All of the latter three languages also support optional type annotations, which the compiler will check against the inferred type.

The scope of both the function name f and the variable x is e. This means that we can call the function inside of its body, but only there. If we want to call the same function at different positions in the program, we have to write down the whole function every time. This is, of course, awkward, and we could improve the situation by including let-expressions into the language. Instead of writing

```
(recfun sqr :: (Int → Int) x = x * x) 5 + (recfun sqr :: (Int → Int) x = x * x) 10
```

we could simply write

```
let square = (recfun sqr :: (Int → Int) x = x * x)
in  square 5 + square 10
```

and reuse the definition of `sqr`. Extending our language in this way would, however, not add any interesting problems, and the objective of MinHs is to keep the language as simple as possible. The language version we use for the assignment does support let-bindings, though.

Similarly, MinHs admits only a single argument for function definitions, which again is an inconvenience for the programmer, but no restriction of the expressiveness of the language. A function, like integer division, which requires two arguments

```
recfun div :: (Int → Int → Int) x y =
  if x < y
    then 0
    else div (x-y) y
```

can be defined in MinHs using nested function definitions as follows:

```
recfun div :: (Int → Int → Int) x =
  recfun div2 :: (Int → Int) y =
    if x < y
      then 0
      else div (x-y) y
```

Since the type constructor → is right associative, the type of `div`: `Int → Int → Int` can be interpreted as the type of a function which requires two integer value to return a value as result, or as `Int → (Int → Int)`, a function which, after accepting one integer value as argument, returns a new function of type `Int → Int` as result. For the same reason, if we have the application `div 14 5`, it is the same as `(div 14) 5` (as application is left associative), and can be interpreted either as the application of the binary function `div` to the two arguments `14` and `5`, or alternatively, as application of the function `div` to the number `14`, which results in a new function which divides `14` by any number it is applied to, and the subsequent application of this function to `5`.

5.2 Higher-order Abstract Syntax

We do not list all the rules for mapping the concrete syntax to the abstract syntax here – for the part of the language which is similar to the arithmetic expression language, the rules would be the same. The translation from concrete to abstract syntax consists of the following steps:

1. Integer constants are represented by terms of the form `(Num` n`)`, where n is an integer value.
2. The boolean constants are represented by the terms `(Const True)`, and `(Const False)`.
3. Every application of an infix operation is represented by a suitable term notation. E.g., `x + y` becomes `(Plus` x y`)`.
4. If-expressions are represented in prefix notation as well. E.g., `if True then 1 else 2` becomes `(If (Const True) (Num 1) (Num 2))`.
5. Application becomes explicit. E.g., `f 4` becomes `(Apply` f `(Num 4))`.
6. Function definitions of the form `recfun` f :: $(\tau_1 \to \tau_2)$ x = e become `(Recfun` τ_1 τ_2 `(`f`.(`x`.`e`))`. That is, in the body e of the function f, both the parameter variable x, as well as the variable f, the name of the function, may occur freely. It is important that f can occur freely in e, otherwise we wouldn't be able to express recursion, and the language would not be more powerful than the simple arithmetic expression language we discussed before.

The higher-order abstract syntax representation of the function `div` given above is

```
Recfun Int
       (Int → Int)
       (div.x. Recfun Int
                      Int
                      (div2.y. (If (Less x y)
                                   (Num 0)
                                   (Apply div (Sub x y)))))
```

5.3 Static Semantics

For the static semantics, we will not only check that each variable is in scope, but also make sure that the program is type correct. This means, that every operator, function or language construct is applied to values of the type that it expects. MinHs is statically typed – that is, each subexpression has a unique type which can be determined at compile time. For example, we do not allow subexpressions like the following:

```
if x < y
  then True
  else 5
```

as it may either evaluate to a value of type `Bool` or of type `Int`, and we can't know at compile time if the values of `x` and `y` are not statically known. Dynamically typed languages, however, do allow such expressions, and type errors may only show up at run time in such a language.

There is only one way for a variable to be introduced into a MinHs expression, namely via `recfun`. The programmer has to provide the types for both the function and the argument variable. In contrast to the arithmetic expression language, where we only needed to keep track of all the variable names when formalising the static semantics, we now have to associate a type with each name. Therefore, an environment now is a set of variable names bundled with their type of the form $\Gamma = \{x :: \texttt{Int}, f :: \texttt{Bool} \to \texttt{Int}, \ldots\}$.

We assume that each variable name is unique. This is not a serious restriction, because any program for which this doesn't hold can be α-renamed to an equivalent program for which it holds.

The judgement $\Gamma \vdash e : \tau$ states that the expression e has type τ under the environment Γ. The inference rules defining the typing judgment are as shown below. (Note that we only provide the typing rules for two built-in operators, `Plus` and `Less`; the rules for the others should be clear.)

Variables and constant values For variables, we have to check what the type of that variable is in the environment. If it is not in the environment, the variable occurs free in the program and the program is not type correct. Constant expressions are always type correct.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad (\textit{mhs type-1})$$

$$\frac{}{\Gamma \vdash (\texttt{Num}\ n) : \texttt{Int}} \qquad (\textit{mhs type-2})$$

$$\frac{b \in \{\texttt{True}, \texttt{False}\}}{\Gamma \vdash (\texttt{Const}\ b) : \texttt{Bool}} \qquad (\textit{mhs type-3})$$

Operations The type of the argument expressions is checked recursively. If these subexpressions have the expected type, the whole expression is type correct, and assigned a type depending on the operator.

$$\frac{\Gamma \vdash e_1 : \texttt{Int} \quad \Gamma \vdash e_2 : \texttt{Int}}{\Gamma \vdash (\texttt{Plus}\ e_1\ e_2) : \texttt{Int}} \qquad (\textit{mhs type-4})$$

$$\frac{\Gamma \vdash e_1 : \texttt{Int} \quad \Gamma \vdash e_2 : \texttt{Int}}{\Gamma \vdash (\texttt{Less}\ e_1\ e_2) : \texttt{Bool}} \qquad (\textit{mhs type-5})$$

Conditionals The first argument of a conditional has to have type `Bool`. The two branches may have any type, as long as both of them have the same type. If not, the program is not type correct in MinHs.

$$\frac{\Gamma \vdash e_1 : \texttt{Bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_1 : \tau}{\Gamma \vdash (\texttt{If}\ e_1\ e_2\ e_3) : \tau} \qquad (\textit{mhs type-6})$$

Application For application, we check that the first argument is actually a function. It can have any argument τ_1 and result type τ_2, as long as the second argument expression to `Apply` also has type τ_1. The type of the application is then τ_2.

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash (\texttt{Apply}\ e_1\ e_2) : \tau_2} \qquad (\textit{mhs type-6})$$

Functions The function rule, similar to the let-rule in the arithmetic expression language, adds variables to the environment. Here, we need the type information provided by the programmer, and stored in the first and second argument of `Recfun`. Otherwise, we would not know what exactly to put in the environment.

$$\frac{\Gamma \cup \{f : \tau_1 \to \tau_2\} \cup \{x : \tau_1\} \vdash e : \tau_2}{\Gamma \vdash (\texttt{Recfun}\ \tau_1\ \tau_2\ (f.(x.e))) : \tau_1 \to \tau_2} \qquad (\textit{mhs type-7})$$

Note again that f is put in the environment, as it might be a recursive function.

5.3.1 Example

Let's use the rule to type check the MinHs function below:

```
recfun f :: (Int → Bool) x = f (x + 1)
```

Translated into higher-order abstract syntax, it becomes the expression:

$$\texttt{(Recfun Int Bool (}f\texttt{.(}x\texttt{.(Apply }f\texttt{ (Plus }x\texttt{ (Num 1))))))}$$

The function is not very useful. It never terminates, no matter what number it is applied to, as it just calls itself with the same argument incremented by one. But we only want to check if it is correct with respect to the static semantics–that is, all variables have to be defined, and all operations and functions only applied to arguments of the correct type.

$$\dfrac{\dfrac{\dfrac{f : \texttt{Int} \to \texttt{Bool} \in \{\ldots\}}{\{\ldots\} \vdash f : \texttt{Int} \to \texttt{Bool}}\,(1) \qquad \dfrac{\dfrac{x : \texttt{Int} \in \{\ldots\}}{\{\ldots\} \vdash x : \texttt{Int}}\,(1) \quad \dfrac{}{\{\ldots\} \vdash \texttt{(Num 1)} : \texttt{Int}}\,(2)}{\{\ldots\} \vdash \texttt{(Plus }x\texttt{ (Num 1))} : \texttt{Int}}\,(4)}{\{f : \texttt{Int} \to \texttt{Bool}, x : \texttt{Int}\} \vdash \texttt{(Apply }f\texttt{ (Plus }x\texttt{ (Num 1)))} : \texttt{Bool}}\,(6)}{\{\} \vdash \texttt{(Recfun Int Bool (}f\texttt{.(}x\texttt{.(Apply }f\texttt{ (Plus }x\texttt{ (Num 1))))))} : \texttt{Bool}}\,(7)$$

After the first inference rule application, the environment does not change anymore, so we abbreviate $\{\, f : \texttt{Int} \to \texttt{Bool},\ x : \texttt{Int}\,\}$ with $\{\ldots\}$. We also just use the *minhs type* rule number, instead of the full rule name.

Also, note that the result type `Bool` is never actually used. We can replace it in the type annotation with any other type, and the type checking would work exactly the same. Contrast this with the similar function:

```
recfun f :: (Bool → Int) x = (f x) + 1
```

Here the type of the argument `x` can be replaced with any other type, since `x` is never used, but the result of the application is, as part of the sum, so it has to be an `Int`.

5.4 Dynamic Semantics

We define the dynamic semantics of MinHs in terms of a single step or structural operational semantics. Similarly to the expression language, the initial states are simply all well typed, closed (that is: they don't contain free variables) expressions. The final states are expressions of the form `Num n`, as before as well as `Const True` and `Const False`. Function expressions in MinHs are also final states, as they cannot be evaluated further by themselves.

- **States:** the set of all closed, well typed expressions

$$S = \{e \mid \exists\tau.\{\} \vdash e : \tau\}$$

- **Initial states:** the set of all closed, well typed expressions

$$I = \{e \mid \exists\tau.\{\} \vdash e : \tau\}$$

- **Final states:** the set of all constant values as well as functions

$$F = \{\texttt{(Num }i\texttt{)} \mid i \in Int\} \cup \{\texttt{(Const True)}, \texttt{(Const False)}\} \cup \{\texttt{(Recfun }\tau_1\ \tau_2\texttt{ (}f\texttt{.(}x\texttt{.}e\texttt{)))}\}$$

- **Operations of the abstract machine:** operations for all the built-in operators on integers: addition, subtraction, multiplication, and such, as well as substitution of variables by values.

Applications of built-in operators are evaluated in the same way as in the arithmetic expression language.

Conditionals are evaluated lazily in their second and third argument. That is, the then- and else-branch are, as one would expect, only evaluated once we know the condition evaluates to `True` or `False`, respectively.

$$\frac{}{\texttt{(If True } e_1\ e_2\texttt{)} \mapsto e_1} \qquad (\textit{minhs step-1})$$

$$\frac{}{\texttt{(If False } e_1\ e_2\texttt{)} \mapsto e_2} \qquad (\textit{minhs step-2})$$

$$\frac{e \mapsto e'}{\texttt{(If } e\ e_1\ e_2\texttt{)} \mapsto \texttt{(If } e'\ e_1\ e_2\texttt{)}} \qquad (\textit{minhs step-3})$$

The following rules for function application specify a strict evaluation: that is, the argument to the function is evaluated before substituting the paramater name. Note also that we replace every occurence of the function name in its body by the function itself.

$$\frac{v \in F}{\texttt{(Apply (Recfun } \tau_1\ \tau_2\ \texttt{(}f\texttt{.(}x\texttt{.}e\texttt{))) } v\texttt{)} \mapsto e[f := \texttt{(Recfun } \tau_1\ \tau_2\ \texttt{(}f\texttt{.(}x\texttt{.}e\texttt{)))}][x := v]} \qquad (\textit{4})$$

$$\frac{e_1 \mapsto e_1'}{\texttt{(Apply } e_1\ e_2\texttt{)} \mapsto \texttt{(Apply } e_1'\ e_2\texttt{)}} \qquad (\textit{minhs step-5})$$

$$\frac{e_2 \mapsto e_2'}{\texttt{(Apply (Recfun } \ldots\texttt{) } e_2\texttt{)} \mapsto \texttt{(Apply (Recfun } \ldots\texttt{) } e_2'\texttt{)}} \qquad (\textit{minhs step-6})$$

Alternatively, we could give MinHs' function application a lazy evaluation semantics, and replace every occurence of the function parameter with the unevaluated expression. For that, we must extend the machine to support the substitution of variables by arbitrary expressions e', and not just values v, as in Rule (*minhs step-4*), which we can then replace with the alternative rule:

$$\frac{}{\texttt{(Apply (Recfun } \tau_1\ \tau_2\ \texttt{(}f\texttt{.(}x\texttt{.}e\texttt{))) } e'\texttt{)} \mapsto e[f := \texttt{(Recfun } \tau_1\ \tau_2\ \texttt{(}f\texttt{.(}x\texttt{.}e\texttt{)))}][x := e']} \qquad (\textit{lazy})$$

In the arithmetic expression language, both the lazy and the strict semantics always resulted in the same value. In MinHs, some programs terminate using lazy evaluation, but not with strict.

```
(recfun foo :: (Int → Int) x = 5) ((recfun bar :: (Int → Int) x = bar x) 10)
```

Evaluating the application of `bar` does not terminate, but with the lazy semantics rules, the whole program would terminate, since `foo` throws its argument away and just returns `5`.

6 TinyC

After examining MinHs, a purely functional language, we now turn our attention to a procedural language with side effects and non-local control flow: TinyC. In TinyC, state is central to program execution. Unlike in purely functional languages, variables in TinyC refer to specific memory locations whose contents can change over time as the program runs. This mutable state is a key characteristic of procedural languages, where operations often involve reading from and writing to memory.

In discussing the semantics of TinyC, we consider not only the results of computations but also how each step of execution impacts the program's state—specifically, the memory contents. This is in stark contrast to purely functional languages, where computations yield consistent results without altering any underlying state. Here, each function or procedure may have side effects that modify the program's state, which can subsequently influence the behavior of other parts of the program.

Additionally, TinyC features non-local control flow mechanisms, such as loops and conditional branches, which allow for more complex execution paths. These mechanisms can significantly affect program behavior depending on the current state. Thus, understanding a TinyC program requires tracking not only the logical flow of instructions but also the evolution of state across different points in the program.

6.1 Language definition

We use a small fragment of the C language for our discussion. The language, TinyC, is defined as follows:

$$\begin{array}{ll}
\mathit{prgm} & ::= \mathit{gdecs}\ \mathit{stmt} \\
\mathit{gdecs} & ::= \epsilon \mid \mathit{gdec}\ \mathit{gdecs} \\
\mathit{gdec} & ::= \mathit{fdec} \mid \mathit{vdec} \\
\mathit{vdecs} & ::= \epsilon \mid \mathit{vdec}\ \mathit{vdecs} \\
\mathit{vdec} & ::= \texttt{int}\ \mathit{Ident} \texttt{=} \mathit{Int}; \\
\mathit{fdec} & ::= \texttt{int}\ \mathit{Ident}\ \texttt{(}\mathit{arguments}\texttt{)}\ \mathit{stmt} \\
\mathit{stmt} & ::= \mathit{expr}; \mid \texttt{if}\ \mathit{expr}\ \texttt{then}\ \mathit{stmt}\ \texttt{else}\ \mathit{stmt}; \mid \texttt{return}\ \mathit{expr}; \mid \\
 & \quad \{\ \mathit{vdecs}\ \mathit{stmts}\ \} \mid \texttt{while (}\ \mathit{expr}\ \texttt{)}\ \mathit{stmt} \\
\mathit{stmts} & ::= \epsilon \mid \mathit{stmt}\ \mathit{stmts} \\
\mathit{expr} & ::= \mathit{Int} \mid \mathit{Ident} \mid \mathit{expr} \texttt{+} \mathit{expr} \mid \mathit{expr} \texttt{-} \mathit{expr} \mid \\
 & \quad \mathit{Ident} \texttt{=} \mathit{expr} \mid \mathit{Ident}\ \texttt{(}\mathit{exprs}\texttt{)} \\
\mathit{arguments} & ::= \epsilon \mid \texttt{int}\ \mathit{Ident}\ \texttt{,}\ \mathit{arguments} \\
\mathit{exprs} & ::= \epsilon \mid \mathit{expr}\ \texttt{,}\ \mathit{exprs}
\end{array}$$

Again, we only include a small set of built-in operations and only a single value type — the techniques we present can be extended to handle a more extensive set of operations (easy) and types (more tricky). The most striking omission, though, are pointers or references. We will add them later in a restricted form. Note that all variables must be initialised to a given constant value.

A program p consists of a sequence of global declaration $gdecs$ and a statement s. Global declarations can be either variable declarations or function declarations. The statement is usually a block $\{ldecs\ ss\}$, that is, a sequence of local declarations $ldecs$ followed by a sequence of statements

We distinguish between statements and functions: statements are predominantly about effect, where as expressions are about value. The distinction is not very clear, though, since expressions also can have effect on the state, and there is implicitly a value associated with each statement.

It is important to note the difference between TinyC's variables and MinHs'. MinHs variables are variables in the strict mathematical sense: they can be bound to a value, but once they are bound, their value cannot change. In contrast to this, TinyC variables represent memory locations, whose content can (and usually does) change during the execution of the program.

For example, the program listed below contains two global declarations - a variable and a function declaration, and the program statement (the `main` function of our TinyC program), simply calls this function and assigns the return value to the global variable `result`:

```
int result = 0;
int div (int x, int y) {
  int res = 0;
  while (x > y) {
    x = x - y;
    res = res + 1;
  }
  return res;
}

result = div (16, 5);
```

For TinyC, we stick to the concrete syntax, as it allows for a more compact, easier to read notation in the rules.

6.2 Static Semantics

Since we only have a single type, `int`, and the programmer has no way of defining new data types, there is not much to be done in TinyC in terms of type checking. We can check statically, though, if all the variables have been declared and initialised, and make sure that functions are always applied to the correct number of arguments.

We need to collect all the functions and variables declared globally and locally, and check the statements and expressions against these two sets:

- Variables $V = \{x_1, x_2, \ldots\}$
- Functions and their arity $F = \{f_1 : n_1, f_2 : n_2, \ldots\}$

We overload ***decs*** to denote a relation over global declarations and a pair V', F' and a relation over local declarations and a set V': ***decs*** :

- $V, F \vdash gdec$ ***decs*** V', F', if a global declaration is well-formed with respect to the given grammar and a set of previously defined functions F and variables V, and declares the set of variables V' and a set of functions F'.
- $V \vdash ldec$ ***decs*** V', if a local declaration is well-formed with respect to the given grammar and a set of previously declared variables V, and declares the set of variables V'.

The following rules define the relation judgments:

- An empty sequence of local declarations is well-formed and declares no variables:

$$\frac{}{V \vdash \circ\ \textbf{\textit{decs}}\ \emptyset} \quad (decs\text{-}1)$$

- A local declaration is well-formed if the variable has not been declared previously:

$$\frac{x \notin V \qquad V \cup \{x\} \vdash \mathit{ldecs}\ \textbf{\textit{decs}}\ V'}{V \vdash \texttt{int}\ x = v;\ \mathit{ldecs}\ \textbf{\textit{decs}}\ V' \cup \{x\}} \tag{decs-2}$$

- An empty sequence of global declarations is well-formed and declares no variables:

$$\frac{}{V, F \vdash \circ\ \textbf{\textit{decs}}\ \emptyset} \tag{decs-3}$$

- A global value declaration is well-formed if the variable has not been declared previously:

$$\frac{x \notin V \qquad V \cup \{x\}, F \vdash \mathit{gdecs}\ \textbf{\textit{decs}}\ V', F'}{V, F \vdash \texttt{int}\ x = v;\mathit{gdecs}\ \textbf{\textit{decs}}\ V' \cup \{x\}, F'} \tag{decs-4}$$

- A global function declaration is well-formed if the function has not been declared previously:

$$\frac{f \notin F \qquad V, F \cup \{f : n\} \vdash \mathit{gdecs}\ \textbf{\textit{decs}}\ V', F'}{V, F \vdash \texttt{int}\ f\texttt{(}x_1, \ldots, x_n\texttt{)} = \mathit{stmt}\texttt{;}\ \mathit{gdecs}\ \textbf{\textit{decs}}\ V', F' \cup \{f : n\}} \tag{decs-5}$$

Now we have to define a relation to check if a program is well-formed, and if an expression or statement is well-formed with respect to a given set of variable and function declarations. Again, we use a single symbol, *ok* to express the relation on expressions, statements as well as programs, and write

- prg ***ok*** if a program is well-formed
- V, F $\vdash \mathit{expr}$ ***ok***, if an expression is well-formed with respect to the current environments V and F
- V, F $\vdash \mathit{stmt}$ ***ok***, if a statement is well-formed with respect to the current variable environment V and function environment F.

The following rules define the judgement:

- Statements and expressions are well-formed, if all of their components are well-formed under the same environments. Since the rules follow the same pattern, we only list some of them here:

$$\frac{V, F \vdash \mathit{expr}\ \textbf{\textit{ok}} \qquad V, F \vdash \mathit{stmt}\ \textbf{\textit{ok}}}{V, F \vdash \texttt{while(}\mathit{expr}\texttt{)}\ \mathit{stmt}\ \textbf{\textit{ok}}} \tag{cok-1}$$

$$\frac{V, F \vdash \mathit{expr}\ \textbf{\textit{ok}} \qquad V, F \vdash \mathit{stmt}_1\ \textbf{\textit{ok}} \qquad V, F \vdash \mathit{stmt}_2\ \textbf{\textit{ok}}}{V, F \vdash \texttt{if(}\mathit{expr}\texttt{)}\ \texttt{then}\ \mathit{stmt}_1\ \texttt{else}\ \mathit{stmt}_2\ \textbf{\textit{ok}}} \tag{cok-2}$$

$$\frac{V, F \vdash \mathit{expr}_1\ \textbf{\textit{ok}} \qquad V, F \vdash \mathit{expr}_2\ \textbf{\textit{ok}}}{V, F \vdash \mathit{expr}\texttt{+}\mathit{expr}_2\ \textbf{\textit{ok}}} \tag{cok-3}$$

- A block is well-formed under F and V if the local declarations are well-formed, and declare a (possibly empty) set of new variables V', and the statements are well-formed with respect to the current declarations V and F, and block local declarations V':

$$\frac{V, F \vdash \mathit{ldecs}\ \textbf{\textit{decs}}\ V' \qquad V \cup V', F \vdash \mathit{stmts}\ \textbf{\textit{ok}}}{V, F \vdash \{\mathit{ldecs}\ \mathit{stmts}\}\ \textbf{\textit{ok}}} \tag{cok-4}$$

- A variable is well-formed if it is declared:

$$\frac{x \in V}{V, F \vdash x\ \textbf{\textit{ok}}} \tag{cok-5}$$

- An assignment is well-formed if the expression is well-formed and the variable declared:

$$\frac{V, F \vdash expr \ \textbf{ok} \quad x \in V}{V, F \vdash x\texttt{=}expr \ \textbf{ok}} \qquad (cok\text{-}6)$$

- And finally, a function call is well-formed if the function is declared and called with the correct number of arguments:

$$\frac{V, F \vdash expr_1 \ \textbf{ok} \quad V, F \vdash expr_2 \ \textbf{ok} \quad \ldots \quad f : n \in F}{V, F \vdash f\texttt{(}expr_1\texttt{,} \ldots \texttt{,}expr_n\texttt{)} \ \textbf{ok}} \qquad (cok\text{-}7)$$

6.3 Dynamic Semantics

We provide the dynamic semantics of TinyC in terms of a big-step semantics, similar to the one we used for MinHs in the assignment. However, there is one major difference between the two languages: in TinyC, variables can (and generally do) change their value during execution. It is therefore not possible to deal with variables by replacing all occurrences with the value they are assigned to, as we did in MinHs. Instead, we have to keep a value environment, which contains all the currently visible variables together with their values. Since the scope of a function is the whole program, we also have to store the function definitions in the environment, although these definitions cannot change during execution.

6.3.1 The environment g

The environment has a similar form as the global declarations in the grammar: it consists of a set of function definitions of the form:

$$\texttt{int } f\texttt{(int } x_1, \ldots\texttt{)} \ stmt$$

and a ordered sequence of variable bindings $x = v$. Note that it is important that the variable bindings are ordered, since we have to be able to identify the binding of a variable put into the environment most recently.

We define the following notation and operations on g

- **empty environment**: we write $\circ$ for the empty environment.
- **lookup:** We denote environment lookup using the infix operator @: so, $g@x = v$ means the current value of x is v. If x is not in g, then $g@x$ is undefined.
- **update:** We write $g@x \leftarrow v$ to update the value of a variable x which already is in the environment to the value v. Again, if x is not in g, then the result of the update operation is undefined.
- **extending environment:** we write $g.\texttt{int} \ x = v$ to add a new variable binding to an environment. If x is already in g, then the new binding overshadows the old binding (but does not delete it).

Examples:

- $\circ$`.int x = 5.int x = 11@x` evaluates to `11` (the first binding of `x` is not accessible anymore)
- $\circ$`.int y = 5.int x = 11@y` evaluates to `5`

Note that we do not need to define an operation to delete bindings from an environment.

6.3.2 Program Execution

A program can be executed without an environment, as all the variables and functions have to be declared in the program itself. For a program $p = \mathit{gdecs}\ \mathit{stmt}$ we write

$$p \Downarrow (g, rv)$$

if it evaluates to rv, which can either be a plain value or a return value of the form `return` (v), and an environment g.

Given an environment g which contains bindings for all free variables in stmt and all functions, we write

$$(g, \mathit{stmt}) \Downarrow (g', rv)$$

to express that stmt evaluates to the value rv (again, either a plain value or a return value) and the new environment g' under g, and similarly for expressions:

$$(g, \mathit{expr}) \Downarrow (g', v)$$

Program execution. The execution rule for a program is trivial. A program $p = g\ s$ consisting of global declarations g and a statement s is executed by executing the statement under the environment as initialised by g:

$$\frac{(g, s) \Downarrow (g', rv)}{g\ s \Downarrow (g', rv)} \qquad (\textit{tc-1})$$

Conditionals. The rules for statements are more interesting. We start with if-statements. Apart from the environment g, which is threaded through the execution of all subexpressions and statements, they look almost the same as the MinHs rules (note that, in the absence of boolean values, we use 0 to represent false, and all other integer values to represent true):

$$\frac{(g, e) \Downarrow (g', \mathtt{0}) \qquad (g', s_2) \Downarrow (g'', rv)}{(g, \mathtt{if}\,(e)\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2) \Downarrow (g'', rv)} \qquad (\textit{tc-2})$$

$$\frac{(g, e) \Downarrow (g', v), v \neq \mathtt{0} \qquad (g', s_1) \Downarrow (g'', rv)}{(g, \mathtt{if}\,(e)\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2) \Downarrow (g'', rv)} \qquad (\textit{tc-3})$$

The evaluation of e, as well as the execution of all the statements, can change the environment — it is therefore important that we use the new environment g' for the evaluation of s_1 and s_2, respectively.

While-loops. As with if-expressions, we have to evaluate the condition of the while-expression first, before we can decide how to continue. If it evaluates to `0` (which we use to represent `False` in TinyC, as it has no boolean type), the whole expression evaluates to `0`, and we are finished:

$$\frac{(g, e) \Downarrow (g', \mathtt{0})}{(g, \mathtt{while}\,(e)\ s) \Downarrow (g', \mathtt{0})} \qquad (\textit{tc-4})$$

Otherwise, we have to execute the body of the while-expression, and then jump back to the while-loop:

$$\frac{(g, e) \Downarrow (g', v), v \neq \mathtt{0} \qquad (g',\ s;\ \mathtt{while}\,(e)\ s) \Downarrow (g'', rv)}{(g, \mathtt{while}\,(e)\ s) \Downarrow (g'', rv)} \qquad (\textit{tc-5})$$

Again, it is important that the new environment is always threaded through. After all, the altered state is the only interesting part of executing a while-loop, since the result value will always be `0`.

We can also replace the rules *tc-4* and *tc-5*, and specify the semantics of a while-loop in a single rule, in terms of if-expressions:

$$\frac{(g, \mathtt{if}\,(e)\ \mathtt{then}\ \{s; \mathtt{while}\,(e)\ s\}\ \mathtt{else}\ ) \Downarrow \mathtt{0};)(g', rv)}{(g, \mathtt{while}\,(e)\ s) \Downarrow (g', rv)} \qquad (\textit{tc-while'})$$

Return-statement. If a return statement contains just a value, it is not evaluated any further:

$$\frac{(g, e)\Downarrow(g', v)}{(g, \texttt{return(}e\texttt{)})\Downarrow(g', \texttt{return(}v\texttt{)})} \qquad (tc\text{-}6)$$

Blocks. Blocks introduce a set of new variable bindings into the environment:

$$\frac{(g.l, ss)\Downarrow(g'.l', rv)}{(g, \{l\ ss\})\Downarrow(g', rv)} \qquad (tc\text{-}7)$$

Note that the statement sequence ss is executed under the old environment g plus the new local bindings. The resulting environment has the form $g'.l'$, meaning that bindings in g may have changed, and bindings in l may have changed, but no additional bindings have been introduced (it is necessary to look at all the inference rules to see why this is always the case). The resulting environment is g' — that is, we add the new bindings only temporary for the execution of ss, and remove them when leaving the block.

Sequence of statements. To evaluate a sequence of statements, the single statements are evaluated one after the other, passing on the resulting environment. If a statement evaluates to a return value, the remaining statements are ignored. Also, note how the results of non-return statements are discarded.

$$\frac{}{(g, \circ)\Downarrow(g, \texttt{0})} \qquad (tc\text{-}8)$$

$$\frac{(g, s)\Downarrow(g', \texttt{return(}v\texttt{)})}{(g, s\ ss)\Downarrow(g', \texttt{return(}v\texttt{)})} \qquad (tc\text{-}9)$$

$$\frac{(g, s)\Downarrow(g', v) \quad (g, ss)\Downarrow(g'', v')}{(g, s\ ss)\Downarrow(g'', v')} \qquad (tc\text{-}10)$$

Variable lookup and assignment. Variables are evaluated by looking up the binding in the environment, and assignment change the value of a variable in the environment:

$$\frac{g@x = v}{(g, x)\Downarrow(g, v)} \qquad (tc\text{-}11)$$

$$\frac{(g, e)\Downarrow(g', v)}{(g, x\texttt{=}e)\Downarrow(g'@x\leftarrow v, v)} \qquad (tc\text{-}12)$$

Function call. The semantics of function calls is expressed in terms of the semantics of block statements:

$$\frac{\begin{array}{c} g@f = \texttt{int}\ f\texttt{(int}\ x_1\texttt{,}\ldots\texttt{)}\ s \\ (g, (e_1, \ldots))\Downarrow(g', (v_1, \ldots)) \quad (g', \{\texttt{int}\ x_1\texttt{=}v_1\texttt{;}\ldots s\})\Downarrow(g'', \texttt{return(}v\texttt{)}) \end{array}}{(g, f\texttt{(}e_1\texttt{,}\ldots\texttt{)})\Downarrow(g'', v)} \qquad (tc\text{-}13)$$

$$\frac{\begin{array}{c} g@f = \texttt{int}\ f\texttt{(int}\ x_1\texttt{,}\ldots\texttt{)}\ s \\ (g, (e_1, \ldots))\Downarrow(g', (v_1, \ldots)) \quad (g', \{\texttt{int}\ x_1\texttt{=}v_1\texttt{;}\ldots s\})\Downarrow(g'', v\texttt{;}) \end{array}}{(g, f\texttt{(}e_1\texttt{,}\ldots\texttt{)})\Downarrow(g'', v)} \qquad (tc\text{-}14)$$

where

$$\frac{(g, e_1)\Downarrow(g_1, v_1) \quad (g_1, e_2)\Downarrow(g_2, v_2) \ \ldots \ (g_{n-1}, e_n)\Downarrow(g_n, v_n)}{(g, (e_1, \ldots, e_n))\Downarrow(g_n, (v_1, \ldots, v_n))} \qquad (tc\text{-}15)$$

Evaluation of function application is the only rule which “unwraps” a return statement. Combined with the rules for sequences of statements, this means that a return statement will effectively cause the evaluation to jump to the first statement after the innermost function call.

7 | Abstract Machines

Abstract machines are a theoretical model of a calculator, typically consisting of a set of states, including initial and final states, and a set of machine operations, which manipulate the state of the machine. They are an important concept in theoretical computer science, for example to specify the operational semantics of a programming language (as we did we MinHs and TinyC), or computational and complexity theory. Probably one of the best known examples of an abstract machine is the Turing machine, which was designed by Alan Turing in 1936 as a means to tackle the Entscheidungsproblem (Decision Problem).

Abstract Machines are closely related to Virtual Machines, which are basically Abstract Machines with an implementation. Such virtual machines can be used to achieve portability of high-level programming languages (like the Java Virtual Machine, or the Common Language Framework for the .NET framework) or complete operating systems.

In this course, we use abstract machines as a means to study the operational semantics of programming languages. We started with two simple machines, which have a very small set of operations and states. We will gradually add more details to the machine and bring it closer to a machine we could actually use as a basis for an implementation. We do this to make languages easier to reason about - we use the higher level specification (for example, big step semantics) for reasoning about higher level properties such as safety, and use the lower level specification for reasoning about machine characteristics such as performance.

Technically, you have already seen two examples of abstract machines - in the structural operational semantics of a language (the small step semantics), and in the evaluation (big step) semantics. While these technically constitute abstract machines, they are perhaps too abstract for our needs.

We seek to specify the evaluation of a language in a more low-level way. Should we start from the evaluation semantics, or the small-step semantics? Notice that the evaluation semantics do not even specify the order in which terms should be evaluated. Seeing as nondeterminism doesn't exist in real computers, we must specify such an order. Hence we could say that the small-step semantics are lower-level than the big-step. Therefore, it makes sense for us to start with the small-step semantics. We call the small-step semantics of MinHS the M-Machine.

7.1 The *C-Machine*

One thing still left rather abstract in the M-Machine is control flow, specifically, the notion of a runtime stack. When we want to evaluate a term, say `(Plus (Plus (Num 2) (Num 3)) (Num 4))`, the *M-Machine* rules tell us that we must first evaluate the inner `(Plus (Num 2) (Num 3))` and then return it to the original expression. This is akin to pushing the greater expression context onto a stack, evaluating the inner expression and then popping the context off the stack again, returning the evaluated expression to it.

We will introduce a new machine, the C-Machine, that makes this stack explicit. In order to do this, we will need to formalise it, but we will get an informal intuition for it first.

7.1.1 *C-Machine* States

In the *M-Machine*, the state of the machine merely consists of the current expression to be evaluated - the notion of the stack is hidden in the deduction tree of the inference rules. In our C-Machine, however, our state consists of three parts:

- The current expression to be evaluated
- A stack of expression frames
- A single binary flag that denotes whether the machine is currently evaluating an expression, or returning a value after evaluating.

To syntactically distinguish simple expressions from return values, we represent them differently than in Chapter 5:

$$\frac{n \in \mathit{Int}}{n\ \textbf{\textit{Value}}} \quad (\mathit{CVal\text{-}1})$$

$$\frac{b \in \{\texttt{True}, \texttt{False}\}}{b\ \textbf{\textit{Value}}} \quad (\mathit{CVal\text{-}2})$$

$$\frac{\Gamma \vdash (\texttt{Recfun}\ \tau_1\ \tau_2\ (f.(x.e)))\ \texttt{:}\ \tau_1 \rightarrow \tau_2}{\langle f.x.e\rangle\ \textbf{\textit{Value}}} \quad (\mathit{CVal\text{-}3})$$

Essentially, we just drop the `Num` and `Bool` label for simple values, and replace `Recfun` with angle brackets for any function which is well-defined for some environment Γ according to the static semantics given in Chapter 5. For the discussion of the dynamic semantics here, we also just omit the type annotations of functions completely.

Frames represent a computation which is waiting on the evaluation of a subexpression. We use $\square$ to represent the subexpression it is waiting for. For example, to represent an addition which is waiting for its first argument to be evaluated, and with expression e as second argument, we write:

$$(\texttt{Plus}\ \square\ e)$$

For our C-machine, we assume that arguments are evaluated from left to right, so a second possible frame is

$$(\texttt{Plus}\ v\ \square)$$

where v is the already fully evaluated first argument (in MinHs, it has to be a number, otherwise the original addition expression would not have been a valid expression according to the static semantics). So, for regular n-ary operators Op, we have n possible frames, with one argument replaced by $\square$, and all the arguments preceding the $\square$-marker values. `If` and `Recfun` are exceptions which we will discuss when introducing their evaluation rules.

Now, we can define what a control stack looks like: just a stack of control frames, or, more formally:

$$\frac{}{\circ\ \textbf{\textit{Stack}}} \quad (\mathit{CStack\text{-}1})$$

$$\frac{s\ \textbf{\textit{Stack}} \quad f\ \textbf{\textit{Frame}}}{f \triangleright s\ \textbf{\textit{Stack}}} \quad (\mathit{CStack\text{-}2})$$

where $\circ$ represents the empty stack, and $x \triangleright s$ is a stack with a frame f on the top.

A sketch of a *C-Machine*

Now that we have the frames and a stack, we can have a shot at representing the machine states. Suppose we want to evaluate our earlier example with the *C-Machine* (in line with the shorter notation we use here, `Num` tags are dropped):

$$\texttt{(Plus (Plus 2 3) 4)}$$

To begin, we need to come up with an initial state for our expression. So, when we start, we have the empty stack ($\circ$), and the machine starts with the flag set to evaluate the expression (denoted by $\succ$).

$$\circ \quad \succ \quad \texttt{(Plus (Plus 2 3) 4)}$$

The rules in the *M-Machine* state that in order to evaluate a `Plus` expression, first the first subexpression should be evaluated, then the second. Hence, in this case, our *C-Machine* will mirror the behaviour of *M-Machine* and therefore should evaluate `(Plus 2 3)` first. To achieve this, a stack frame is pushed for `Plus`, with a $\square$ in place of the first subexpression, and the machine is set to evaluate the expression we just replaced:

$$\texttt{(Plus}\ \square\ \texttt{4)} \triangleright \circ \quad \succ \quad \texttt{(Plus 2 3)}$$

The machine now has to evaluate another `Plus`, so another stack frame is pushed:

$$\texttt{(Plus}\ \square\ \texttt{3)} \triangleright \texttt{(Plus}\ \square\ \texttt{4)} \triangleright \circ \quad \succ \quad \texttt{2}$$

Now the machine simply has to evaluate a numeric literal. Seeing as no further evaluation need take place (the value associated with the expression can be inferred without further work), the machine switches to return (denoted by $\prec$) the value back into the awaiting stack frame:

$$\texttt{(Plus}\ \square\ \texttt{3)} \triangleright \texttt{(Plus}\ \square\ \texttt{4)} \triangleright \circ \quad \prec \quad \texttt{2}$$

Having evaluated the first argument, the machine again suspends computation of the `Plus` expression in order to evaluate the second subexpression, which proceeds similarly:

$$\begin{aligned}
\texttt{(Plus 2}\ \square\texttt{)} \triangleright \texttt{(Plus}\ \square\ \texttt{4)} \triangleright \circ \quad &\succ \quad 3\\
\texttt{(Plus 2}\ \square\texttt{)} \triangleright \texttt{(Plus}\ \square\ \texttt{4)} \triangleright \circ \quad &\prec \quad 3
\end{aligned}$$

As the machine is returning the last value necessary for the `Plus` frame, it pops the frame off the stack, performs a primitive addition operation, and returns the result 5:

$$\texttt{(Plus}\ \square\ \texttt{4)} \triangleright \circ \quad \prec \quad 5$$

The rest of the evaluation proceeds predictably:

$$\begin{aligned}
\texttt{(Plus}\ 5\ \square\texttt{)} \triangleright \circ \quad &\succ \quad 4\\
\texttt{(Plus}\ 5\ \square\texttt{)} \triangleright \circ \quad &\prec \quad 4\\
\circ \quad &\prec \quad 9
\end{aligned}$$

7.1.2 Formalising the *C-Machine*

Now that we have an informal idea of what our *C-Machine* looks like, we can begin to formalise the machine. An abstract machine in general consists of:

- A set of states Q.

- A set of initial states $I \subseteq Q$.
- A set of final states $F \subseteq Q$.
- A state transition function, $\mapsto_C \,:\, S \to S$.

You have seen this before in small-step semantics – this is because small-step semantics are a form of abstract machine.
We start by defining the legal, initial and final states of the machine:

- The set of legal states Q contains the following elements:
  - If s ***Stack*** and e ***Expr***, then $s \succ e \in Q$.
  - If s ***Stack*** and v ***Value***, then $s \prec v \in Q$.

  That is, Q is comprised of all evaluating states and all returning states.
- The initial states set I is defined as the set of all evaluating states with an empty stack:
  - If e ***Expr***, then $\circ \succ e \in I$.
- The final states F are defined as all returning states with an empty stack:
  - If v ***Value***, then $\circ \prec v \in F$.

Now we must define the state transition relation for our *C-Machine*, $\mapsto_C$.

Literals

To begin, we will start on the easy part – evaluating numeric, boolean literals, and functions:

$$\frac{}{s \succ n \quad \mapsto_C \quad s \prec n} \tag{CM-1}$$

$$\frac{}{s \succ (\texttt{Bool}\ b) \quad \mapsto_C \quad s \prec b} \tag{CM-2}$$

$$\frac{}{s \succ (\texttt{Recfun}\ (f.(x.e))) \quad \mapsto_C \quad s \prec \langle f.x.e \rangle} \tag{CM-3}$$

So, the machine simply returns the corresponding values unchanged – no further computation is necessary. For function values, we wrap them in angle brackets $\langle f.x.e \rangle$, but the principle is the same as for numbers and booleans:

Primitive Operations

Now we can specify more complicated rules, such as that for `Plus`. When faced with an unevaluated `Plus`, expression, the machine first evaluates the first subexpression, and pushes the rest on the stack:

$$\frac{}{s \succ (\texttt{Plus}\ e_1\ e_2) \quad \mapsto_C \quad (\texttt{Plus}\ \square\ e_2) \triangleright s \succ e_1} \tag{CM-4}$$

Once that subexpression is evaluated, the machine will begin evaluating the second subexpression:

$$\frac{}{(\texttt{Plus}\ \square\ e_2) \triangleright s \prec v \quad \mapsto_C \quad (\texttt{Plus}\ v\ \square) \triangleright s \succ e_2} \tag{CM-5}$$

Finally, when both subexpressions are evaluated, the machine returns the resulting sum, computed via a primitive machine operation, +:

$$\frac{}{\texttt{(Plus } v_1\ \square\texttt{)} \triangleright s \prec v_2 \quad \mapsto_C \quad s \prec (v_1 + v_2)} \qquad (\textit{CM-6})$$

The definitions for `Times`, `Greater`, `Equal`, `Not`, `And` etc. are very similar, just making use of different primitive machine operations.

Conditionals

Now, what about evaluating `If`-expressions? Recall that in the *M-Machine*, the machine first evaluates the condition to some result, and, depending on the result, would evaluate just one branch of the conditional.

Similarly, when faced with an unevaluated `If` expression, the *C-Machine* evaluates the condition first:

$$\frac{}{s \succ \texttt{(If } e_1\ e_2\ e_3\texttt{)} \quad \mapsto_C \quad \texttt{(If } \square\ e_2\ e_3\texttt{)} \triangleright s \succ e_1} \qquad (\textit{CM-7})$$

If the result is `True`, the first branch is evaluated, otherwise the second:

$$\frac{}{\texttt{(If } \square\ e_2\ e_3\texttt{)} \triangleright s \prec \texttt{True} \quad \mapsto_C \quad s \succ e_2} \qquad (\textit{CM-8})$$

$$\frac{}{\texttt{(If } \square\ e_2\ e_3\texttt{)} \triangleright s \prec \texttt{False} \quad \mapsto_C \quad s \succ e_3} \qquad (\textit{CM-9})$$

Therefore, for the `If`-operator, we only need one single frame, even though it has three arguments.

Function Application

Finally, we must deal with function application. Recall that in the *M-Machine*, first the function expression was evaluated, then the argument to the function, then finally the body of the function, substituting in the value of the argument. We employ a similar strategy here.

First evaluate the function expression:

$$\frac{}{s \succ \texttt{(App } f\ e\texttt{)} \quad \mapsto_C \quad \texttt{(App } \square\ e\texttt{)} \triangleright s \succ f} \qquad (\textit{CM-10})$$

Then, once that is fully evaluated to a function value, evaluate the argument:

$$\frac{}{\texttt{(App } \square\ e\texttt{)} \triangleright s \prec \langle f\,.\,x\,.\,e'\rangle \quad \mapsto_C \quad \texttt{(App } \langle f\,.\,x\,.\,e'\rangle\ \square\texttt{)} \triangleright s \succ e} \qquad (\textit{CM-11})$$

Then finally, we can evaluate the function body e by replacing every occurence of the parameter x with the value v, and every occurence of the function f with the function itself, in case the function is recursive:

$$\frac{}{\texttt{(App } \langle f\,.\,x\,.\,e\rangle\ \square\texttt{)} \triangleright s \prec v \quad \mapsto_C \quad s \succ e[x := v][f := \langle f\,.\,x\,.\,e\rangle]} \qquad (\textit{CM-12})$$

Note that every single rule in our system is now an axiom, as there is no hidden recursion anymore, so not surprisingly, each step in the C-machine makes much less progress than a single step in the M-Machine.

7.1.3 Example

Let us step through the evaluation of a very simple recursive function, to see how this works. Here, function `f` checks if the boolean argument if `True`, and returns `5` if that's the case. Otherwise, it calls itself recursively on the negated argument (and therefore return `5` in the next step).

```
(recfun f :: (Bool → Bool) b =
  if x then 5
       else f (not x)) False
```

The higher-order abstract syntax representation (again, using the shorter notation for values) of this program is:

$\mathtt{(Apply}\ \langle f.x.\mathtt{(If}\ x\ \mathtt{5}\ \mathtt{(Apply}\ f\ \mathtt{(Not}\ x\mathtt{)))}\rangle\ \mathtt{False)}$

To make the notation more compact, we abbreviate further, and write `F` `T`, `App` in place of `False`, `True` and `Apply`, respectively. The machine starts with the empty stack, and the expression as current expression:

$$
\begin{array}{lrl}
 & \circ & \succ \mathtt{(App}\ \langle f.x.\mathtt{(If}\ x\ \mathtt{5}\ \mathtt{(App}\ f\ \mathtt{(Not}\ x\mathtt{)))}\rangle\ \mathtt{F)} \\
\mapsto_c & \mathtt{(App}\ \square\ \mathtt{F)} \triangleright \circ & \succ \langle f.x.\mathtt{(If}\ x\ \mathtt{5}\ \mathtt{(App}\ f\ \mathtt{(Not}\ x\mathtt{)))}\rangle \\
\mapsto_c & \mathtt{(App}\ \square\ \mathtt{F)} \triangleright \circ & \prec \langle f.x.\mathtt{(If}\ x\ \mathtt{5}\ \mathtt{(App}\ f\ \mathtt{(Not}\ x\mathtt{)))}\rangle \\
\mapsto_c & \mathtt{(App}\ \langle f.x.\mathtt{(If}\ x\ \mathtt{5}\ \mathtt{(App}\ f\ \mathtt{(Not}\ x\mathtt{)))}\rangle\ \square\mathtt{)} \triangleright \circ & \succ \mathtt{F} \\
\mapsto_c & \mathtt{(App}\ \langle f.x.\mathtt{(If}\ x\ \mathtt{5}\ \mathtt{(App}\ f\ \mathtt{(Not}\ x\mathtt{)))}\rangle\ \square\mathtt{)} \triangleright \circ & \prec \mathtt{F} \\
\mapsto_c & \circ & \succ \mathtt{(If\ F\ 5\ (App}\ \langle f.x.\mathtt{(If}\ \ldots\mathtt{)}\rangle\ \mathtt{(Not\ F))} \\
\mapsto_c & \mathtt{(If}\ \square\ \mathtt{5\ (App}\ \langle f.x.\mathtt{(If}\ \ldots\mathtt{)}\rangle\ \mathtt{(Not\ F)))} \triangleright \circ & \succ \mathtt{F} \\
\mapsto_c & \mathtt{(If}\ \square\ \mathtt{5\ (App}\ \langle f.x.\mathtt{(If}\ \ldots\mathtt{)}\rangle\ \mathtt{(Not\ F)))} \triangleright \circ & \prec \mathtt{F} \\
\mapsto_c & \circ & \succ \mathtt{(App}\ \langle f.x.\mathtt{(If}\ \ldots\mathtt{)}\rangle\ \mathtt{(Not\ F))} \\
\mapsto_c & \mathtt{(App}\ \square\ \mathtt{(Not\ F))} \triangleright \circ & \succ \langle f.x.\mathtt{(If}\ \ldots\mathtt{)}\rangle \\
\mapsto_c & \mathtt{(App}\ \square\ \mathtt{(Not\ F))} \triangleright \circ & \prec \langle f.x.\mathtt{(If}\ \ldots\mathtt{)}\rangle \\
\mapsto_c & \triangleright\mathtt{(App}\ \langle f.x.\mathtt{(If}\ \ldots\mathtt{)}\rangle\ \square\mathtt{)} \triangleright \circ & \succ \mathtt{(Not\ F)} \\
\mapsto_c & \mathtt{(Not}\ \square\mathtt{)} \triangleright \mathtt{(App}\ \langle f.x.\mathtt{(If}\ \ldots\mathtt{)}\rangle\ \square\mathtt{)} \triangleright \circ & \succ \mathtt{F} \\
\mapsto_c & \mathtt{(Not}\ \square\mathtt{)} \triangleright \mathtt{(App}\ \langle f.x.\mathtt{(If}\ \ldots\mathtt{)}\rangle\ \square\mathtt{)} \triangleright \circ & \prec \mathtt{F} \\
\mapsto_c & \mathtt{(App}\ \langle f.x.\mathtt{(If}\ x\ \mathtt{5}\ \mathtt{(App}\ f\ \mathtt{(Not}\ x\mathtt{)))}\rangle\ \square\mathtt{)} \triangleright \circ & \prec \mathtt{T} \\
\mapsto_c & \circ & \succ \mathtt{(If\ T\ 5\ (App}\ \langle f.x.\mathtt{(If}\ \ldots\mathtt{)}\rangle\ \mathtt{(Not\ T))} \\
\mapsto_c & \mathtt{(If}\ \square\ \mathtt{5\ (App}\ \langle f.x.\mathtt{(If}\ \ldots\mathtt{)}\rangle\ \mathtt{(Not\ F)))} \triangleright \circ & \succ \mathtt{T} \\
\mapsto_c & \mathtt{(If}\ \square\ \mathtt{5\ (App}\ \langle f.x.\mathtt{(If}\ \ldots\mathtt{)}\rangle\ \mathtt{(Not\ F)))} \triangleright \circ & \prec \mathtt{T} \\
\mapsto_c & \circ & \succ \mathtt{5} \\
\mapsto_c & \circ & \prec \mathtt{5}
\end{array}
$$

That's a lot of work, just to compute if three is even! No wonder we prefer evaluation semantics! Computers, however, certainly would prefer the *C-Machine* - note that every state transition for the *C-Machine* is an axiom. This means we can implement it as a single tight `while` loop that moves from state to state until it reaches a state in F.

Note: In an exam situation, you may be asked to present a derivation like the above. It is not necessary to write out every single step, just those steps you believe to be most important.

7.2 The *E-Machine*

Now that we've made control flow more explicit, it becomes easier to see how we would implement the language efficiently on a real computer.

Let's take a look at our primitive machine operations so far:

- **The Numeric Operators**: $+$, $*$ ...
- **Comparison Operators**: $==$, $<$, ...
- **Logical Operators**: $\wedge$, $\vee$, $\neg$, ...
- **Substitution**: $e[x := v]$

The great thing about most of these is that in most computers they are native machine instructions, so they can be implemented very efficiently.

The one operation that *cannot* be implemented very efficiently is substitution. Substitution is of complexity $O(n)$ in the size of the expression - it would be very difficult to implement it as an efficient machine instruction!

So, we are going to extend our machine once more, to include environments in the machine state to store the value of our variables in the current context. Previously, we used environments in the static semantics to keep track of the set of defined variables, in the arithmetic expression language, and for MinHs, the set of defined variables and their types. We assumed that all the variable names where unique, so we did not have to worry about shadowing. This is a reasonable assumption, because we can always take a program and α-rename the variables to get an equivalent program with unique names. If the names were not unique, we would have to use a stack-like structure to store the variables and their type, so that we can access the innermost binding of variable.

The environments in the dynamic semantics are different, though. They map every variable to the value it is bound to. For example, consider the following application of a recursive function (in concrete syntax, and omitting the types here for readability.)

```
(recfun f x =
  if (x == 1)
    then 1
    else x * f (x - 1)) 10
```

First, the function is applied to the value `10`, so `x` is bound to `10`. In the next iteration, it is applied to `9`, so we need to bind `x` to that. The value environment therefore has to be stack as well, which lets us access the latest binding of a variable.

We will call our new machine the E-Machine, for environment machine, as it looks up variables and their value in the environment rather than rely on substitution.

Value environments are just stacks for bindings. For the empty environment stack (to distinguish it from the empty control stack), we write $\bullet$, and for an environment with the binding of the variable x to the value v as topmost entry, Γ as the remaining environment stack $x = v; \Gamma$.

Similar to our value and function environments for TinyC, we have a lookup operation $\Gamma@x$, which returns the value of variable x of the topmost binding.

$$\frac{}{\bullet\ \textit{\textbf{Env}}} \quad (\textit{Env-1})$$

$$\frac{\Gamma\ \textit{\textbf{Env}} \quad v\ \textit{\textbf{Value}}}{x = v; \Gamma\ \textit{\textbf{Env}}} \quad (\textit{Env-2})$$

The set of Machine states Q of the E-machine are now a triple of current control stack s, current environment Γ, and current expression e or current return value v:

- **Evaluation mode** $s|\Gamma \succ e$:the machine with the control stack s, environment Γ is evaluating the expression e.
- **Return mode** $s|\Gamma \prec v$: the machine with the control stack s, environment Γ is returning the value v.

Variables

In the *C-Machine*, there is no rule to evaluate a variable, since this should never happen, unless the variable was free in the initial expression, which would mean that it wasn't a legal expression according to the static semantics to begin with.

In the *E-Machine*, variables are to be expected, and their current value should be stored in the environment (assuming we checked that no variable occurs out of scope via the static semantics, and the rules for function application add the bindings correctly to the environment).

When the current expression is a variable in the E-machine, we look up its value in the environment, and return that value:

$$\frac{\Gamma @ x = v}{s|\Gamma \succ x \quad \mapsto_{\boldsymbol{E}} \quad s|\Gamma \prec v} \qquad (\textit{EM-1})$$

Function applications

In the C-machine, function application is implemented by substituting the parameter variable with the value the function is applied to, and the function name in the body with the full function. In the environment semantics, we need to add bindings for both variables to the environment. This is similar to what we did in TinyC, with the difference that there was no need to add the function to the environment, since all functions in TinyC are globally defined and already in the function environment.

Now, what happens when we call a function? Naturally, we'd want to introduce new bindings to our environment, for the argument and the recursive name. When the function returns, however, we want to remove these bindings, as they are no longer in scope. The way we achieve this may sound strange: we just store the environment active at the time we called the function on the stack. To be able to do so, we extend our stack to be able to include environments as well as frames:

$$\frac{}{\circ \ \textit{\textbf{Stack}}} \qquad (\textit{MStack-1})$$

$$\frac{s \ \textit{\textbf{Stack}} \quad f \ \textit{\textbf{Frame}}}{f \triangleright s \ \textit{\textbf{Stack}}} \qquad (\textit{MStack-2})$$

$$\frac{s \ \textit{\textbf{Stack}} \quad \Gamma \ \textit{\textbf{Env}}}{\Gamma \triangleright s \ \textit{\textbf{Stack}}} \qquad (\textit{MStack-3})$$

Now, we can define application, by evaluating the expressions with the bindings of f and x, and storing the old environment Γ on the stack:

$$\frac{}{(\texttt{Apply}\ \langle f.x.e\rangle\ \square) \triangleright s|\Gamma \prec v \quad \mapsto_{\boldsymbol{E}} \quad \Gamma \triangleright s|x = v; f = \langle f.x.e\rangle; \Gamma \succ e} \qquad (\textit{Attempt-1})$$

When the function returns, we restore the environment Γ from the stack, and discard the current environment Γ', which has the effect of removing the unwanted bindings which were only valid in the function body:

$$\frac{}{\Gamma \triangleright s|\Gamma' \prec v \quad \mapsto_{\boldsymbol{E}} \quad s|\Gamma \prec v} \qquad (\textit{EM-2})$$

All other state transition rules are identical to the *C-Machine*, preserving the environment across the transition.

The initial and final states are also unchanged, except that they now include the empty environment:

- If e ***Expr***, then $\circ \mid \bullet \succ e \ \in I$
- If v ***Value***, then $\circ \mid \bullet \prec v \ \in F$

7.2.1 Example

We use the same example as for the C-machine, but now E-machine rules, with an environment instead of substitution. The function expression

$$\langle f\texttt{.}x\texttt{.(If } x \texttt{ 5 (App } f \texttt{ (Not } x\texttt{)))}\rangle$$

never changes, so we just write

$$\langle \ldots \rangle$$

in its place.

$$\begin{array}{lrl}
 & \circ \mid & \bullet \succ \texttt{(App } \langle\ldots\rangle \texttt{ F)} \\
\mapsto_{E} & \texttt{(App } \Box \texttt{ F)} \triangleright \circ \mid & \bullet \succ \langle\ldots\rangle \\
\mapsto_{E} & \texttt{(App } \Box \texttt{ F)} \triangleright \circ \mid & \bullet \prec \langle\ldots\rangle \\
\mapsto_{E} & \texttt{(App } \langle\ldots\rangle \ \Box\texttt{)} \triangleright \circ \mid & \bullet \succ \texttt{F} \\
\mapsto_{E} & \texttt{(App } \langle\ldots\rangle \ \Box\texttt{)} \triangleright \circ \mid & \bullet \prec \texttt{F} \\
\mapsto_{E} & \bullet \triangleright \circ \mid & x = \texttt{F};\, f = \langle\ldots\rangle;\, \bullet \succ \texttt{(If } x \texttt{ 5 (App } f \texttt{ (Not } x\texttt{)))} \\
\mapsto_{E} & \texttt{(If } \Box \texttt{ 5 (App } \langle\ldots\rangle \texttt{ (Not } x\texttt{)))} \triangleright \bullet \triangleright \circ \mid & x = \texttt{F};\, f = \langle\ldots\rangle;\, \bullet \succ x \\
\mapsto_{E} & \texttt{(If } \Box \texttt{ 5 (App } \langle\ldots\rangle \texttt{ (Not } x\texttt{)))} \triangleright \bullet \triangleright \circ \mid & x = \texttt{F};\, f = \langle\ldots\rangle;\, \bullet \prec \texttt{F} \\
\mapsto_{E} & \bullet \triangleright \circ \mid & x = \texttt{F};\, f = \langle\ldots\rangle;\, \bullet \succ \texttt{(App } \langle\ldots\rangle \texttt{ (Not F))} \\
\mapsto_{E} & \texttt{(App } \Box \texttt{ (Not F))} \triangleright \bullet \triangleright \circ \mid & x = \texttt{F};\, f = \langle\ldots\rangle;\, \bullet \succ \langle\ldots\rangle \\
\mapsto_{E} & \texttt{(App } \Box \texttt{ (Not F))} \triangleright \bullet \triangleright \circ \mid & x = \texttt{F};\, f = \langle\ldots\rangle;\, \bullet \prec \langle\ldots\rangle \\
\mapsto_{E} & \texttt{(App } \langle\ldots\rangle \ \Box\texttt{)} \triangleright \bullet \triangleright \circ \mid & x = \texttt{F};\, f = \langle\ldots\rangle;\, \bullet \succ \texttt{(Not F)} \\
\mapsto_{E} & \texttt{(Not } \Box\texttt{)} \triangleright \texttt{(App } \langle\ldots\rangle \ \Box\texttt{)} \triangleright \bullet \triangleright \circ \mid & x = \texttt{F};\, f = \langle\ldots\rangle;\, \bullet \succ \texttt{F} \\
\mapsto_{E} & \texttt{(Not } \Box\texttt{)} \triangleright \texttt{(App } \langle\ldots\rangle \ \Box\texttt{)} \triangleright \bullet \triangleright \circ \mid & x = \texttt{F};\, f = \langle\ldots\rangle;\, \bullet \prec \texttt{F} \\
\mapsto_{E} & \texttt{(App } \langle\ldots\rangle \ \Box\texttt{)} \triangleright \bullet \triangleright \circ \mid & x = \texttt{F};\, f = \langle\ldots\rangle;\, \bullet \prec \texttt{T} \\
\mapsto_{E} & (x = \texttt{F};\, f = \langle\ldots\rangle;\, \bullet) \triangleright \bullet \triangleright \circ \mid & x = \texttt{T};\, f = \langle\ldots\rangle;\, \bullet \succ \texttt{(If } x \texttt{ 5 (App } \langle\ldots\rangle \texttt{ (Not } x\texttt{)))} \\
\mapsto_{E} & \texttt{(If } \Box \texttt{ 5 (App } \langle\ldots\rangle \texttt{ (Not F)))} \triangleright (x = \texttt{F};\, f = \langle\ldots\rangle;\, \bullet) \triangleright \bullet \triangleright \circ \mid & x = \texttt{T};\, f = \langle\ldots\rangle;\, \bullet \succ x \\
\mapsto_{E} & \texttt{(If } \Box \texttt{ 5 (App } \langle\ldots\rangle \texttt{ (Not F)))} \triangleright (x = \texttt{F};\, f = \langle\ldots\rangle;\, \bullet) \triangleright \bullet \triangleright \circ \mid & x = \texttt{T};\, f = \langle\ldots\rangle;\, \bullet \prec \texttt{T} \\
\mapsto_{E} & (x = \texttt{F};\, f = \langle\ldots\rangle;\, \bullet) \triangleright \bullet \triangleright \circ \mid & x = \texttt{T};\, f = \langle\ldots\rangle;\, \bullet \prec \texttt{5} \\
\mapsto_{E} & \bullet \triangleright \circ \mid & x = \texttt{F};\, f = \langle\ldots\rangle;\, \bullet \prec \texttt{5} \\
\mapsto_{E} & \circ \mid & \bullet \prec \texttt{5}
\end{array}$$

The evaluation with the E-machine returns the same result as with the C-machine (as we would hope), but it takes even more steps. In the last few steps, when `5` is returned from the function call, note how the old environments, which were saved to the stack, are restored. In this example, it does not make a difference, but in the example we looked at earlier, we need the old value of `x` when returning from the call:

```
(recfun f x =
  if (x == 0)
    then 0
    else x + f (x - 1)) 10
```

If not, we function would always evaluate to zero.

7.2.2 Closures

Unfortunately, the problem is even more complicated. Consider the following function `c`, which, if applied to two arguments, returns the first one:

```
(recfun c x =
   (recfun cx y = x)) 10 20
```

or, in higher-order abstract syntax

$$\texttt{(App (App}\ \langle c.x.\langle cx.y.x\rangle\rangle\ \texttt{10) 20)}$$

Let us first look at how the evaluation of this expression works with substitution semantics (C-machine):

$$\begin{aligned}
& \circ \succ \texttt{(App (App}\ \langle c.x.\langle cx.y.x\rangle\rangle\ \texttt{10) 20)}\\
\mapsto_C\ & \texttt{(App}\ \Box\ \texttt{20)} \triangleright \circ \succ \texttt{(App}\ \langle c.x.\langle cx.y.x\rangle\rangle\ \texttt{10)}\\
\mapsto_C\ & \texttt{(App}\ \Box\ \texttt{10)} \triangleright \texttt{(App}\ \Box\ \texttt{20)} \triangleright \circ \succ \langle c.x.\langle cx.y.x\rangle\rangle\\
\mapsto_C\ & \texttt{(App}\ \Box\ \texttt{10)} \triangleright \texttt{(App}\ \Box\ \texttt{20)} \triangleright \circ \prec \langle c.x.\langle cx.y.x\rangle\rangle\\
\mapsto_C\ & \texttt{(App}\ \langle c.x.\langle cx.y.x\rangle\rangle\ \Box\texttt{)} \triangleright \texttt{(App}\ \Box\ \texttt{20)} \triangleright \circ \succ \texttt{10}\\
\mapsto_C\ & \texttt{(App}\ \langle c.x.\langle cx.y.x\rangle\rangle\ \Box\texttt{)} \triangleright \texttt{(App}\ \Box\ \texttt{20)} \triangleright \circ \prec \texttt{10}\\
\mapsto_C\ & \texttt{(App}\ \Box\ \texttt{20)} \triangleright \circ \succ \langle cx.y.\texttt{10}\rangle\\
\mapsto_C\ & \texttt{(App}\ \Box\ \texttt{20)} \triangleright \circ \prec \langle cx.y.\texttt{10}\rangle\\
\mapsto_C\ & \texttt{(App}\ \langle cx.y.\texttt{10}\rangle\ \Box\texttt{)} \triangleright \circ \succ \texttt{20}\\
\mapsto_C\ & \texttt{(App}\ \langle cx.y.\texttt{10}\rangle\ \Box\texttt{)} \triangleright \circ \prec \texttt{20}\\
\mapsto_C\ & \circ \succ \texttt{10}\\
\mapsto_C\ & \circ \prec \texttt{10}
\end{aligned}$$

As it should, it evaluates to `10`. Note how, in the sixth step, the variable x in the function is replaced with the argument `10` in the evaluation of the application, and the result value is another function which, no matter what it is applied to, returns the value `10`.

Now, consider the evaluation using our current E-machine rules:

$$
\begin{array}{lrl}
 & \circ \mid & \bullet \succ \texttt{(App (App } \langle c.x.\langle cx.y.x\rangle\rangle \texttt{ 10) 20)} \\
\mapsto_{\boldsymbol{E}} & \texttt{(App } \Box \texttt{ 20)} \triangleright \circ \mid & \bullet \succ \texttt{(App } \langle c.x.\langle cx.y.x\rangle\rangle \texttt{ 10)} \\
\mapsto_{\boldsymbol{E}} & \texttt{(App } \Box \texttt{ 10)} \triangleright \texttt{(App } \Box \texttt{ 20)} \triangleright \circ \mid & \bullet \succ \langle c.x.\langle cx.y.x\rangle\rangle \\
\mapsto_{\boldsymbol{E}} & \texttt{(App } \Box \texttt{ 10)} \triangleright \texttt{(App } \Box \texttt{ 20)} \triangleright \circ \mid & \bullet \prec \langle c.x.\langle cx.y.x\rangle\rangle \\
\mapsto_{\boldsymbol{E}} & \texttt{(App } \langle c.x.\langle cx.y.x\rangle\rangle\ \Box\texttt{)} \triangleright \texttt{(App } \Box \texttt{ 20)} \triangleright \circ \mid & \bullet \succ \texttt{10} \\
\mapsto_{\boldsymbol{E}} & \texttt{(App } \langle c.x.\langle cx.y.x\rangle\rangle\ \Box\texttt{)} \triangleright \texttt{(App } \Box \texttt{ 20)} \triangleright \circ \mid & \bullet \prec \texttt{10} \\
\mapsto_{\boldsymbol{E}} & \bullet \triangleright \texttt{(App } \Box \texttt{ 20)} \triangleright \circ \mid & x = \texttt{10};\ c = \langle\ldots\rangle; \bullet \succ \langle cx.y.x\rangle \\
\mapsto_{\boldsymbol{E}} & \bullet \triangleright \texttt{(App } \Box \texttt{ 20)} \triangleright \circ \mid & x = \texttt{10};\ c = \langle\ldots\rangle; \bullet \prec \langle cx.y.x\rangle \\
\mapsto_{\boldsymbol{E}} & \texttt{(App } \Box \texttt{ 20)} \triangleright \circ \mid & \bullet \prec \langle cx.y.x\rangle \\
\mapsto_{\boldsymbol{E}} & \texttt{(App } \langle cx.y.x\rangle\ \Box\texttt{)} \triangleright \circ \mid & \bullet \succ \texttt{20} \\
\mapsto_{\boldsymbol{E}} & \texttt{(App } \langle cx.y.x\rangle\ \Box\texttt{)} \triangleright \circ \mid & \bullet \prec \texttt{20} \\
\mapsto_{\boldsymbol{E}} & \bullet \triangleright \circ \mid & y = \texttt{20};\ cx = \langle\ldots\rangle; \bullet \succ x \\
\mapsto_{\boldsymbol{E}} & \text{☹} &
\end{array}
$$

There's a problem! We can't evaluate x here, as x is not in our environment! What is going on here? Our program is perfectly fine according to the static semantics, and all variables are in scope, so how could it happen?

If you look at the evaluation, you can see that when the function $\langle cx.y.x\rangle$ is returned, the function local environment which contains the binding of x, gets discarded and replaced with the empty environment, which was valid at the time the function was called. Restoring the environment of the caller is necessary, as we have seen in the previous section. However, when we return the function value $\langle cx.y.x\rangle$, we also have to remember somehow the state of the environment at that point in time, such that, when we apply it to an argument which binds y, all the variables which occur in the body of the function are bound to a value.

The Solution

To fix this problem, we must revise our definition of a function value. Instead of just returning the function expression, we also return the current environment. Such a bundle of a function and its environment is called a closure, and they are necessary in all languages which allow functions as return values, or the application of functions to a subset of their arguments (partial application).

We write a closure, that is, the pair of current environment Γ and function $\langle f.x.e\rangle$ as $\langle\!\langle \Gamma, f.x.e \rangle\!\rangle$. With this, we can now define our evaluation rules for functions and applications properly as:

$$\frac{}{s \mid \Gamma \succ \langle f.x.e\rangle \quad \mapsto_{\boldsymbol{E}} \quad s \mid \Gamma \prec \langle\!\langle \Gamma, f.x.e \rangle\!\rangle} \qquad (EM\text{-}3)$$

When we apply a closure with an argument, it's very similar to our rules before, except instead of simply augmenting the current environment with the argument values and recursive name, we first set the current environment to be the environment stored in the closure:

$$\frac{}{\texttt{(Apply } \langle\!\langle \Gamma, f.x.e \rangle\!\rangle\ \Box\texttt{)} \triangleright s \mid \Gamma' \prec v \quad \mapsto_{\boldsymbol{E}} \quad \Gamma' \triangleright s \mid x = v;\ f = \langle\!\langle \Gamma, f.x.e \rangle\!\rangle; \Gamma \succ e} \qquad (EM\text{-}4)$$

With this environment capture in place, we can now evaluate the example above successfully:

$$
\begin{array}{lrl}
 & \circ \mid & \bullet \succ (\texttt{App}\ (\texttt{App}\ \langle c.x.\langle cx.y.x\rangle\rangle\ \texttt{10})\ \texttt{20}) \\
\mapsto_{E} & (\texttt{App}\ \Box\ \texttt{20}) \triangleright \circ \mid & \bullet \succ (\texttt{App}\ \langle c.x.\langle cx.y.x\rangle\rangle\ \texttt{10}) \\
\mapsto_{E} & (\texttt{App}\ \Box\ \texttt{10}) \triangleright (\texttt{App}\ \Box\ \texttt{20}) \triangleright \circ \mid & \bullet \succ \langle c.x.\langle cx.y.x\rangle\rangle \\
\mapsto_{E} & (\texttt{App}\ \Box\ \texttt{10}) \triangleright (\texttt{App}\ \Box\ \texttt{20}) \triangleright \circ \mid & \bullet \prec \langle\!\langle \bullet, c.x.\langle cx.y.x\rangle\rangle\!\rangle \\
\mapsto_{E} & (\texttt{App}\ \langle\!\langle \bullet, c.x.\langle cx.y.x\rangle\rangle\!\rangle\ \Box) \triangleright (\texttt{App}\ \Box\ \texttt{20}) \triangleright \circ \mid & \bullet \succ \texttt{10} \\
\mapsto_{E} & (\texttt{App}\ \langle\!\langle \bullet, c.x.\langle cx.y.x\rangle\rangle\!\rangle\ \Box) \triangleright (\texttt{App}\ \Box\ \texttt{20}) \triangleright \circ \mid & \bullet \prec \texttt{10} \\
\mapsto_{E} & \bullet \triangleright (\texttt{App}\ \Box\ \texttt{20}) \triangleright \circ \mid & x = \texttt{10};\ c = \langle\ldots\rangle; \bullet \succ \langle cx.y.x\rangle \\
\mapsto_{E} & \bullet \triangleright (\texttt{App}\ \Box\ \texttt{20}) \triangleright \circ \mid & x = \texttt{10};\ c = \langle\ldots\rangle; \bullet \prec \langle\!\langle x = \texttt{10};\ c = \langle\ldots\rangle; \bullet, cx.y.x\rangle\!\rangle \\
\mapsto_{E} & (\texttt{App}\ \Box\ \texttt{20}) \triangleright \circ \mid & \bullet \prec \langle\!\langle x = \texttt{10};\ c = \langle\ldots\rangle; \bullet, cx.y.x\rangle\!\rangle \\
\mapsto_{E} & (\texttt{App}\ \langle\!\langle x = \texttt{10};\ c = \langle\ldots\rangle; \bullet, cx.y.x\rangle\!\rangle\ \Box) \triangleright \circ \mid & \bullet \succ \texttt{20} \\
\mapsto_{E} & (\texttt{App}\ \langle\!\langle x = \texttt{10};\ c = \langle\ldots\rangle; \bullet, cx.y.x\rangle\!\rangle\ \Box) \triangleright \circ \mid & \bullet \prec \texttt{20} \\
\mapsto_{E} & \bullet \triangleright \circ \mid & y = \texttt{20};\ cx = \ldots;\ x = \texttt{10};\ c = \ldots; \bullet \succ x \\
\mapsto_{E} & \bullet \triangleright \circ \mid & y = \texttt{20};\ cx = \ldots;\ x = \texttt{10};\ c = \ldots; \bullet \prec \texttt{10} \\
\mapsto_{E} & \bullet \triangleright \circ \mid & y = \texttt{20};\ cx = \ldots;\ x = \texttt{10};\ c = \ldots; \bullet \prec \texttt{10}
\end{array}
$$

8 Type Safety, Errors and Exceptions

In programming, types are sets that classify values and expressions according to their properties and the operations that can be performed on them. Types define constraints on how data is represented and manipulated, providing a structure that ensures certain kinds of correctness in programs.

A type system of a programming language is a framework, often considered a form of semantics, that assigns a type to every value, expression, or statement within the language. This assignment can occur at compile time (in languages with static type systems) or at runtime (in languages with dynamic type systems). A well-designed type system plays a critical role in improving software quality and reliability by identifying and preventing errors where operations are applied to incompatible types, which could otherwise lead to undefined behavior and software bugs.

Static type systems offer the advantage of identifying type-related issues at compile time. By detecting these issues early, static type systems help developers address problems before the program is even run, which can lead to safer and more efficient code. For example, if a programmer tries to divide a string by an integer, a static type system will flag this error before the code is executed, preventing unexpected behavior and potentially reducing debugging time.

In contrast, dynamic type systems perform type checks at runtime, which allows for greater flexibility, especially in languages that emphasize rapid development and runtime adaptability, such as Python or JavaScript. In these languages, type checking occurs only when the relevant operation is executed. Although dynamic type systems cannot prevent all type-related errors before the program runs, they often allow the program to terminate gracefully when a type error occurs, providing feedback that can help diagnose the issue.

The choice between static and dynamic type systems affects how developers interact with the language. Static typing tends to lead to more predictable and optimized performance because types are verified and enforced by the compiler, which can then optimize code accordingly. Dynamic typing, however, promotes flexibility and ease of use, as developers are not required to declare types explicitly, allowing for rapid prototyping and more fluid code changes.

In recent years, many languages have adopted gradual typing, allowing developers to benefit from the strengths of both static and dynamic typing. For example, gradual typing enables developers to specify types where desired.

Programming languages are often categorised by the sort of type system they provide. Popular statically typed languages (i.e languages which perform type checking as part of static semantics) include Java, C# and C, whereas dynamically typed languages include Ruby, Perl and Python.

An example of a gradually typed programming language is TypeScript [11]. TypeScript extends JavaScript by adding optional static typing, allowing developers to specify types where desired but without requiring them. This means that developers can write TypeScript code without specifying types, and TypeScript will function similarly to JavaScript (a dynamically typed language). However, they also have the option to add type annotations, enabling type checks and other benefits typically associated with statically typed languages.

In addition to the static-dynamic distinction, languages can be categorised by how type-safe or strongly typed they are. These terms are often used to mean various things by different authors and in different books, so we will define them formally in a moment. Nevertheless it is commonly accepted that C is an unsafe language, and that Java or Haskell have substantially more type-safety.

8.1 Type Safety

Robin Milner, one of the inventors of Standard ML (the language, not machine learning!), famously stated that "well typed programs cannot go wrong" [12]. Obviously (and unfortunately), this doesn't mean that well typed programs always do what they are supposed to do. It means that they cannot get stuck, or show undefined behaviour in a type safe language: the execution of a program that is deemed correct according to the static semantics will result in a legal final state if the execution terminates according to the dynamic semantics. Type safety therefore tells us something about the relation between the static and dynamic semantics.

Formally, we use the following common definition for type safety:

Type Safety. A language with small-step states Q, final states $F \subseteq Q$, state transition relation $\mapsto$, and typing rules is type safe if it has two properties:

- Progress - If a program can be typed, it is either a final state or can progress to another state. That is, if $\vdash e : \tau$ then $e \in F$ or $\exists\ e'\ .\ e \mapsto e'$.

- Preservation - If a program has type, evaluation will not change that type. That is, if $\vdash e : \tau$ and $e \mapsto e'$ then $\vdash e' : \tau$.

It can be seen from the above definition that well typed programs will not reach a stuck state. If the program is a final state, then it is by definition not stuck. If not, we know from the progress property that the program must move to a new state. We know from preservation that this new state is also typed, which means (from progress) that it must either be a final state or progress to a new state. Similar reasoning applies until the program terminates (or loops).

It therefore follows that languages such as C, which are unsafe, could reach a stuck state. In such a situation, the program doesn't simply halt (or at least, it's not obliged to). What happens is left undefined. For example, there is no telling what this C program will do without knowledge of the memory layout used by the particular compiler in combination with the used operating system. If the function pointer `f` happens to point to a valid location in the code section, this program might not even trigger an error:

```
 void boom () {
  void (f*)(void) = 0xdeadbeef;
  f ();
}
```

Clearly, speaking of type safety is only applicable in the context of formal treatment of programming languages. Determining exactly what guarantees a type system gives you requires these techniques.

In general, the more expressive the type system is, the more information can be inferred by the compiler. Therefore, for practical reasons we typically want type checking to be decidable. If it was not, our compiler may not terminate. There are languages such as C++, however, where type checking may not terminate.

The remainder of this document will focus on three extensions to the MinHS semantics. One dynamic extension, to ensure progress in the face of partial functions, and two static types extensions, which make MinHS type system more expressive.

8.2 Dealing with Partiality

MinHs, as defined in Chapter 5 is type safe (though we didn't provide a proof). But let's suppose we add a partial operation, such as division, typed as follows:

$$\frac{\Gamma \vdash e_1 : \texttt{Int} \quad \Gamma \vdash e_2 : \texttt{Int}}{\Gamma \vdash (\texttt{Div}\ e_1\ e_2) : \texttt{Int}} \quad (\textit{mhs type-8})$$

The expression (Div 5 0) is well-typed, but what happens in the dynamic semantics? Division by zero is not defined, so we cannot evaluate it, which would violate the progress property. There are essentially two ways to fix this:

- Change the static semantics - That is, disallow divisions by zero statically. There are techniques to approximate this for Turing-complete languages, however in general this is undecidable. For those that are interested, the proof is roughly as follows: If we had the ability to decide statically if we are dividing by zero in all cases, we could write a program that solves the halting problem. Given a program X as input, which may or may not terminate, produce a program X' which just runs X and then returns 0. Then insert X' into the divisor position of some division, and statically check if this division would divide by zero. If yes, the program terminates, otherwise it does not. Seeing as the halting problem is known undecidable, we therefore know that our assumption (that we can decide if we are dividing by zero statically) is false. Hence determining if we can divide by zero statically is undecidable.
- Change the dynamic semantics - That is, for partial functions, make sure that it ends up in a defined (error) state. This approach is used by most languages.

Seeing as MinHS is Turing complete, we are unable to statically analyse if the program divides by zero (at least not if we want the static analysis to terminate for every program). Hence, we shall extend the dynamic semantics of the language to handle the situation at runtime.

The simplest fix is to make partial functions yield some new state Error $\in F$ for undefined cases:

$$\frac{}{(\texttt{Div}\ v\ (\texttt{Num}\ 0)) \mapsto \texttt{Error}} \quad (\textit{minhs-step } 8)$$

Furthermore, we would define Error to interrupt any nested computation and produce Error.

$$\frac{}{(\texttt{Plus Error}\ e) \mapsto \texttt{Error}} \quad (\textit{minhs-step } 7)$$

$$\frac{}{(\texttt{Plus}\ e\ \texttt{Error}) \mapsto \texttt{Error}} \quad (\textit{minhs-step } 8)$$

$$\frac{}{(\texttt{If Error}\ e_1\ e_2) \mapsto \texttt{Error}} \quad (\textit{minhs-step } 9)$$

There are, of course, a very large number of additional Error propagation rules. Here, our abstract machines actually buy us some brevity. We simply state that partial functions result in Error, and completely annihilate the stack (e.g in the C Machine):

$$\frac{}{(\texttt{Div}\ v\ \square) \triangleright s \prec 0 \quad \mapsto_C \quad \circ \prec \texttt{Error}} \quad (\textit{CM-13})$$

This guarantees progress - partial functions will evaluate to Error where they are not defined, meaning that the evaluation will not hit a stuck state.

We have yet to ensure preservation, however. Preservation says that type is preserved across evaluation. Seeing as any partial function application (of any type) could evaluate to Error, the only way to make Error respect preservation is to make it a member of every type:

$$\frac{}{\Gamma \vdash \texttt{Error} : \tau} \quad (\textit{mhs type-9})$$

8.2.1 Exceptions

Adding a `Error` state seems well and good for ensuring type safety, but many real-world languages have more robust, fine-grained error handling techniques, namely exceptions.

Exceptions are a means for a function to exit without returning. Instead, the function may raise an exception, which is caught by an exception handler somewhere further up the runtime stack. Most of you would have seen exceptions from languages such as Java, Python, or C++.

We will extend MinHS to include exceptions by adding two pieces of abstract syntax:

- `(Try` e_1 `(`x`.`e_2`))` evaluates e_1, and if
- `(Raise` τ v`)` is ever encountered while evaluating e_1, it will stop evaluating e_1, and start evaluating e_2 where x is bound to the value v.

These `Try` expressions can of course be nested, and exceptions can be re-`Raise`d within an exception handler.

An example in concerte syntax might look like the following pseudo-code snippet:

```
recfun f (Int -> Int) x =
  try ... -- some code which may contain a division by zero, or other exceptions
  catch y with   -- if an exception is raised, then it's bound to y
    if (y == DivByZero)
      then 0
      else raise y  -- if it's another exception, this function doesn't know what to do
```

Exception values (Such as y in the above example), are made to be of a fixed type, τ_{exc}. It is not relevant what type this is exactly for our discussion. It could be a special `Exception` type (e.g., some enumeration type which, in the above example, contains a value `DivByZero`), it could be an `Int` error code, or just a `String` message.

We type these new expressions as follows. `Try` expressions take the type of their subexpressions, and `raise` expressions are of any type specified in the expression (for a similar reason to the typing of `Error`):

$$\frac{\Gamma \vdash e : \tau_{exc}}{\Gamma \vdash (\texttt{Raise}\ \tau\ e) : \tau} \quad (\textit{minhs type exc-1})$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \cup \{x : \tau_{exc}\} \vdash e_2 : \tau}{\Gamma \vdash (\texttt{Try}\ e_1\ (x.e_2)) : \tau} \quad (\textit{minhs type exc-2})$$

Dynamic Semantics for the C-Machine

Let us now have a look at the dynamic semantics of exceptions. We use the simpler substitution semantics of the C-machine for now, where we do not have to worry about the environment.

To evaluate `Try`, we evaluate its first argument, and push a `Try` frame on the stack, just with any other operator:

$$\frac{}{s \succ (\texttt{Try}\ e_1\ (x.e_2)) \quad \mapsto_C \quad (\texttt{Try}\ \square\ (x.e_2)) \triangleright s \succ e_1} \quad (\textit{CM exc-1})$$

Then, if the evaluation returns a regular value, we simply discard the `try` stack frame:

$$\frac{}{(\texttt{Try}\ \square\ (x.e_2)) \triangleright s \prec v \quad \mapsto_C \quad s \prec v} \quad (\textit{CM exc-2})$$

But what happens when an exception is raised? We cannot just return the value like a regular value, and we also cannot just delete the stack, as we did with `Error`. Instead, we have to search through the stack to find the top-most handler (i.e., `Try`-frame), by popping all other frames from the stack. One option to model this is to introduce a new kind of machine mode, in addition to the return mode $\prec$ and evaluation mode $\succ$, used to signify that an exception is being raised, written $\curlyeqprec$.

If we encounter a `Raise` expression, we first evaluate the exception value being raised:

$$\frac{}{s \succ (\texttt{Raise}\ e) \quad \mapsto_C \quad (\texttt{Raise}\ \square) \triangleright s \succ e} \tag{\textit{CM exc-3}}$$

Once it returns, we enter the new exception handling mode, $\curlyeqprec$:

$$\frac{}{(\texttt{Raise}\ \square) \triangleright s \prec v \quad \mapsto_C \quad s \curlyeqprec v} \tag{\textit{CM exc-4}}$$

This mode continuously pops frames off the stack:

$$\frac{f \neq (\texttt{Try}\ \square\ (x\texttt{.}e))}{f \triangleright s \curlyeqprec v \quad \mapsto_C \quad s \curlyeqprec v} \tag{\textit{CM exc-5}}$$

Until at last we encounter a `Try` expression, and the machine switches back to regular evaluation mode, with the handler-expression as current expression:

$$\frac{}{(\texttt{Try}\ \square\ (x\texttt{.}e)) \triangleright s \curlyeqprec v \quad \mapsto_C \quad s \succ e[x := v]} \tag{\textit{CM exc-6}}$$

9 | Composite Data Types

Up to now, with the exception of function types, we have only discussed primitive data types, such as integers and boolean values. The absence of more complex types is not merely an inconvenience—it significantly restricts the expressiveness of the language. For instance, we cannot define a function that returns multiple values at once, or a function that, depending on the input, returns either a boolean or an integer value, or even a (statically) unknown number of result values.

To overcome these limitations, general-purpose programming languages provide mechanisms that allow programmers to define their own composite data types. These mechanisms enable more expressive and versatile code by supporting structured and complex data representations. The way that languages support these types, and how they associate operations with them, varies significantly. Broadly, we can classify the approaches into three categories:

1. Low-level languages, like C, which offer minimal abstraction and closely reflect the representation of data in memory.

2. Object-oriented languages (or languages with object-oriented features), where classes are the default mechanism for defining complex types.

3. Declarative languages, which often utilize algebraic data types (ADTs)—not to be confused with abstract data types, which share the same acronym.

In this chapter, we will explore all three approaches. We will look more closely on the semantics of algebraic data types, while the semantics of class-based approaches will be covered in Chapter **??**, where we look at a minimal object-oriented language.

9.1 User-defined Data Types in Programming Languages

9.1.1 Type Synonyms

A type synonym is a feature in some programming languages that allows developers to create an alias for an existing type, essentially giving a new name to a type without creating a new, distinct type.This new name, or synonym, can then be used interchangeably with the original type throughout the program. Type synonyms are common in languages like C#, Haskell, Rust (with the type keyword), and some other statically typed languages.

They are not really composite types as such and also don't add anything in terms of expressivity, but they do provide several other advantages, so it's worthwhile to briefly look at them:

Improving Code Readability: Type synonyms can make code more readable by giving more descriptive names to existing types. For instance, if we have a `String` type that represents a Username, creating a type synonym type `Name = String` makes the code more expressive. Whenever `Name` appears, it's clear that this particular String represents a username rather than a generic string.

Simplifying Complex Types: In languages that support more complex or nested types, type synonyms can simplify code by creating aliases for these compound types. For example, if a type involves nested lists or tuples, it may be hard to read or write consistently. A synonym like type `Matrix = [[Int]]` in Haskell makes it easier to work with matrices, allowing you to use Matrix directly instead of `[[Int]]`.

Supporting Domain-Specific Abstractions: In many cases, applications require types that correspond to specific concepts in a domain, such as `AccountId`, `ProductId`, or `Coordinates`. Type synonyms allow programmers to name these types in a way that aligns with the application's domain, helping to avoid confusion and making the codebase easier to understand and maintain.

Assisting with Type Consistency: Although type synonyms don't add any compile-time constraints themselves (unlike distinct types), they can still improve type consistency by reminding developers to use specific names in certain contexts. For example, even though `Name` and `Password` might both be synonyms for `String`, using the specific names can help catch unintentional misuse or misinterpretation during code review.

Type synonyms are a lightweight way to improve code expressiveness without altering type behavior, enhancing readability, and supporting domain-specific clarity in code.

Example In C#, for example, we can define a type synonym via `using`

```
using UserName = System.String;

  Name firstName = "John";
  Name lastName = "Doe";
```

This provides a new name for an existing type, and the two types can now be used interchangeably. In C, the same can be achieved using `typedef` (note that `String` is not a primitive type in C, and is already a type synonym):

```
typedef String Name;
```

and similarly in Haskell via `type`:

```
type String = Name
```

In all these example, the type checker does not distinguish between the types `String` and `Name`, so it just adds convenience and can contribute to the readability of a program, but does not enable any additional checks.

We can also define a new type name for strings, which is viewed by the type checker as a different type. In C and C#, there is no light-weight mechanism to do so, we have to go via classes or structs. In Go, like in Haskell, both aliases and new types are possible, the difference is easy to miss:

```
// Alias for string
type Name = string

// New type based on string
type Name string
```

In Haskell, we have the `newtype` keyword, but it requires to wrap the string value in a data constructor, a

9.1.2 Enumeration Types

Next, let's look at language support for defining a new type with a (typically small) finite number of elements. Typically examples are modelling the days of the week, months, a set of colours, or states an object can be in. This can be done using so-called enumeration types.

The syntax for defining such a type in C and C# is almost the same (apart from the semicolon at the end), but what happens behind the scene, and the usage, is quite different:

```
// enum example in C
enum Colour { Red, Blue, Yellow };

enum Colour favoriteColour = RED;

    if (favoriteColour == RED) {
        printf("Favorite color is red.");
    } else ...
```

In C, enums are just syntactic sugar for integer constants, starting by default at zero, so `Red` is zero, `Blue` is one, and so one, and we can do exactly the same things with `Red` is zero, `Blue` and `Yellow` as we can do with the corresponding integers. However, we get some type safety, as C allows only these values to be assigned to a variable of `Colour` type.

In contrast, in C#, it is treated by the type checker as a different type, although they are also represented as integer constants.

```
// enum example in C#

enum Colour { Red, Blue, Yellow }
class Program
{
    static void Main()
    {
        Colour favoriteColour = Colour.Red;

        if (favoriteColour == Colour.Red)
        {
            Console.WriteLine("Favourite colour is red.");
        }
```

In Haskell, there is no mechanism for enum types specifically, we can simply use a `data` with only nullary constructors:

```
-- enum example in Haskell
data Colour = Red | Green | Blue
```

and inspect them using pattern matching.

9.1.3 Records and Structs

Structs and records are data structures used in programming languages to group multiple fields or properties into a single, composite type. They allow for more complex and organized data management by enabling programmers to combine related data elements under a single name. Both are widely used to improve code readability, encapsulate data logically, and facilitate efficient memory usage.They have the following main uses:

Grouping Related Data: Structs and records bundle together fields that represent a single concept or entity, making it easier to pass around and manipulate related data as a cohesive unit. For instance, a `Person` struct might group a person's name, age, and address into one entity, rather than managing them as separate variables.

Improving Code Readability and Organization: By using structs and records, programmers can give meaningful names to related groups of data, making code more understandable. For example, a `Rectangle` struct with width and height fields clearly represents a rectangle's dimensions, rather than requiring the developer to remember which variable represents each dimension.

Encapsulating Data: Structs and records often serve as a foundation for simple data encapsulation. In many languages, structs and records can include methods (functions associated with the data), allowing developers to associate specific behaviors with the data. This way, they provide a lightweight form of encapsulation without requiring a full class structure.

Optimising Memory Usage: Structs, especially in systems programming languages like C, are typically stored in a contiguous block of memory. This allows them to be more memory-efficient than other data structures, as they avoid the overhead associated with references or pointers to multiple separate objects. Because of this, structs are often favored in performance-sensitive applications.

C. In C, we can bundle different types in a struct, and name the different fields. The following code snippet defines a struct type (`Point`), declares a variable `zeroPnt` of this type, and sets both fields to zero.

```
// declaring the struct
struct Point {
  float x;
  float y;
} Point;

struct Point zeroPnt;

 zeroPnt.x = 0.0;
 zeroPnt.y = 0.0;
```

Accessing a field of the struct type in C (and in many other languages) is done using the notation *varName*.*fieldName*. When declaring a variable of struct type, the C runtime system allocates a contiguous block of memory large enough to fit all the fields of the struct (in addition to some padding, in some circumstances).

A class based approach. In most object oriented languages, composite types are defined as classes. Some, however, such as C# and C++, often for historical reasons, offer C-like structs as well. In C#, for example, we can define a `Point` type either as a struct

```
struct Point {
    public float X;
    public float Y;

    public Point (float x, float y) {
         X = x;
         Y = y;
    }
}
```

or as a class,

```
class Point {
    public float X { get; set; }
    public float Y { get; set; }

    public Point (float X, float Y) {
        X = x;
        Y = y;
    }
}
```

These two definitions both model a point type, but they are not exactly semantically equivalent. Structs in C# are so-called value types, and classes reference types. We will discuss the differences between the two in Chapter 10 in detail.

Algebraic data types. In Haskell (and in many functional languages), there are essentially two ways to model a point type. Values can just be bundled using built-in tuple types (conbined with type synonyms, if desired):

```
type Point =  (Float, Float)

zeroPoint :: Point
zeroPoint = (0, 0)
```

However, we can also define a new type

```
data PointT = PointC Float Float
```

or (almost equivalently)

```
newtype PointT = PointT Float Float
```

This defines a new type, `PointT`, and a new data constructor, `PointC`, which is essentially a function which takes two floating point arguments, and creates a new value of type `PointT`.[1]

```
PointC :: Float  -> Float -> PointT
```

The fields can be accessed using pattern matching:

```
isZeroPnt :: Point -> Bool
isZeroPnt (PointC x y) = (x == 0) && (y == 0)
```

Note that the x- and the y-coordinate are here not identified by name, but by their argument position. That might be fine if, like in this example, there are only few arguments to a constructor. However, if a constructor has many arguments, maybe of the same type, it's error prone and inconvenient to not give the different fields names. Haskell offers names fields (or records) to address this

```
data PointT = PointC {x :: Float, y ::  Float}
```

The names can just be ignored, and we can use exactly the same pattern matching syntax as above, but by naming the fields, we implicitly also define access functions of that name:

```
x :: Point -> Float
y :: Point -> Float
```

[1]Usually in Haskell, when defining a type with just one data constructor, the type constructor and the data constructor are given the same name. We choose two different names here just to be able to refer to them more easily.

so an alternative way to define the `isZeroPnt` function would be:

```
isZeroPnt :: Point -> Bool
isZeroPnt pnt = (x pnt == 0) && (y pnt == 0)
```

So, applying the access function `x` to `pnt` is similar to the dot-notation expression `pnt.x` we would use in many other languages.

9.1.4 Variants

Let's say we want a function which, depending on the context, returns either an integer value or a floating point value. What should be its return type? We could use a struct of integer and floating point value, with a flag telling us whether the former or the latter are relevant, but this inefficient – certainly, if we have more then two possible return types. So, while structs allow us to return an integer **and** an floating point value, we need something to express that the value is of integer **or** float.

C. In C, the syntax for expressing a type which is either an `int` or a `float` is almost the same as that for for structs, but we use the `union` keyword instead:

```
union floatOrInt {
  float f;
  int i;
} fOrI;
```

We set the value of the variable either to a float value

```
  fOrI.f = 5.0;
```

or an integer

```
  fOrI.i = 5;
```

Even though the syntax for unions and structs in C look very similar, what happens behind the scenes is quite different. For a variable of struct type, C allocates (at least) space for every field. However, for a value of union type, it only allocates the space it needs for the biggest alternative. When we access the memory either for reading or writing, it accesses the same location. For example, in the following code snippet

```
fOrI.i = 0;
fOrI.f = 1.23456;
printf ("%d", fOrI.i);
```

when we print `fOrI.i`, it will interpret the floating point encoding of `1.23456` as integer, and print that value (`1067320848`, depending on exact representation).

The problem is now that, if a function returns a value of type `fOrI`, we don't know whether we should interpret it as integer value or as floating point value. The solution for this is to use a so-called tagged union: using a struct, we store a tag which contains this information, together with the union type:

```
typedef enum {
    INT,
    FLOAT,
} TypeTag;

typedef struct {
    TypeTag type;
    union {
        int i;
        float f;
    } data;
} TaggedVariant;
```

We can then use values of this type by first inspecting the tag, and then switching to the appropriate code:

```
void print_variant (TaggedVariant v) {
    switch (v.type) {
        case INT:
            printf("Integer: %d\n", v.data.i);
            break;
        case FLOAT:
            printf("Float: %f\n", v.data.f);
            break;
    }
}

int main() {
    TaggedVariant v1 = { .type = INT, .data.i = 42 };
    TaggedVariant v2 = { .type = FLOAT, .data.f = 7.25 };

    print_variant(v1);  // Output: Integer: 42
    print_variant(v2);  // Output: Float: 7.25
}
```

This can still go wrong, because a programmer might accidentally attach the wrong tag, and there will be no compiler warning or error message. To make it somewhat safer, the construction can be wrapped into an abstraction, so that the tag isn't set manually. Modern programming languages represent variant types behind the scenes often in a similar manner, but provide safe abstractions on the language level.

A class based approach. In object oriented languages, express variants using classes by defining the variant type as superclass. For example, if we want to have something similar to the `Variant` type above in C, we can define it as abstract class (since we do not want to define instances of the class):

```
public abstract class Variant
{
    public abstract void ShowValue();
}

public class IntVariant : Variant
{
    public int Value { get; set;}

    public IntVariant(int value)
    {
        Value = value;
    }

    public override void ShowValue()
    {
        Console.WriteLine("Int Variant: {0}",Value);
    }
}

public class FloatVariant : Variant
{
  ...
}

public class Program
{
    public static void  Main()
    {
        Variant intVariant = new IntVariant(42);
        Variant floatVariant = new FloatVariant(3.14f);

        intVariant.ShowValue();         // Output: Int Variant: 42
        floatVariant.ShowValue();       // Output: Float Variant: 3.14
    }
}
```

Haskell also offers classes, so-called type classes, although Haskell's type classes work differently and they have a different implementation and serve a somewhat different purpose compared to classes in object oriented languages. They are typically used to group a set of types which share a set of operations. The code below approximates the behaviour of the C# definition:

```
class Variant a where
  showValue :: a -> String

instance Variant Int where
  showValue n     = "Int: " ++ show n

instance Variant Int where ...
```

We will discuss the implementation of both OO classes and Haskell-like type classes later.

Using algebraic data types. An alternative, arguably more common, way to implement such a variant type is by using `data` again, but this time, the data constructors are parametrised with the value type:

```
data Variant = IntVariant Int | FloatVariant Float

showValue :: Variant -> String
showValue (IntVariant n)     = "Int: " ++ show n
showValue (FloatVariant f) = "Float: " ++ show f
```

C# doesn't have fully fledged algebraic data types, but abstract records can be used to achieve similar behaviour. In particular, we also have some form of pattern matching available:

```
public abstract record Variant;

public record IntVariant (int Value) : Variant;

public record FloatVariant (double Value) : Variant;

Shape shape = new Circle(5);
```

```
string showValue = shape switch
{
    IntVariant n    => ("Int Variant: {0}", n);
    FloatVariant f => ("Flat Variant: {0}", f);
    _ => "Unknown shape"
};
```

One major difference between the ADT approach and the class approach is how easy it is to add alternatives to the variants, and how easy it is to add functionality. In the ADT approach, we can add new functions which operate on `Variant` easily, but if we add a new variant, we have to edit every function on `Variant` anywhere in the program to add an extra case. In the class approach, it is the opposite: adding a new variant type is easy: we define a new subclass, and implement all the methods of that class. However, adding a new method means we have to add the implementation of the new method to any subclass, everywhere in the program.

9.1.5 Recursive Types

Up to now, all the types we defined had a fixed size, or at least, in the case of variant types, a fixed maximum size. For many applications, we also need to be able to define structures which can grow dynamically, and whose size is therefore not known statically. Typical examples of such structures are lists and trees: recursive structures, who can contain as a component an item of the same type.

Again, we start by looking at how such structures can be expressed in C, and then at class and ADT abstractions, using linked lists of integers as example.

C A linked list is either an empty list, or it is a list element, which contains data (an integer value in our example), and the rest of the list. So we use a `struct` to model a list element. The data field is just an integer, and the rest of the list, `next`, is a pointer to such a struct. If the tail is empty, then this pointer will be `NULL`, and otherwise it contains the address of the head of the tail. In C, we can use a type which has not been defined yet (here `struct IntNode`) in a declaration when we refer to a pointer to the not-yet-declared type. This is possible, because the amount of memory needed for the struct does not depend on the actual type of the pointer.

```
/ Define the structure for a node in the linked list
struct IntNode {
    int data;
    struct IntNode* next;
};

// type synonym to refer to the struct
typedef struct IntNode*  IntList

// constant for the empty list
const IntList emptyList = NULL;
```

In C, we have to allocate memory for the list nodes explicitly. For example, consider the following function, which, given an integer and a list, inserts an item with the integer at the head of the list. It first allocates enough space on the heap for the struct, then initialises the struct fields to the appropriate values, and returns the address of the struct (i.e., a pointer to the head of the list).

```
// Function to add a new item at the head of a list
IntList cons (int data, IntList tail) {
    IntList newList = (IntList) malloc (sizeof (struct IntNode));
    newList->data = data;
    newList->next = tail;
    return newList;
}
```

Traversing the list is straight forward (the `->` infix operator in C is an abbreviation, which dereferences a pointer to a struct, and accesses the given field)

```
// Function to traverse the linked list and print the elements
void traverseList (IntList list) {
    IntList current = list;
    while (current != emptyList) {
        printf("%d\n", current->data);
        current = current->next; // Move to the next node
    }
}

int main() {
    // Create the linked list with the element 10, 20, 30
   IntList list = cons (10, cons (20, cons (30, emptyList)));

    // Traverse the list and print the values
    printf("Linked list elements:\n");
    traverseList(list);

    // Free the allocated memory
    IntList temp;
    while (list != emptyList) {
        temp = list;
        list = list->next;
        free (temp);
    }

    return 0;
}
```

Note that when we don't need a data structure anymore which has been allocated on the heap, we need to explicitly free it again, otherwise we might have a memory leak if the program continues (in this case, it wouldn't matter, since the program terminates anyway). We need the `temp` variable here, because we have to dereference the `list` pointer before freeing the memory.

A class based approach. Now let us have a look at how we can define such a list in C#, although in practice, there is usually no need to define a list from scratch, as linked lists are provided in the standard library.

Not surprisingly, we define a class `Node`, which, similar to the C struct, recursively, contains a node as field. In constrast to C, there is no need to specify explicitly that this is a reference, because classes in C# are reference types. An empty list is, as in the C solution, represented by a null reference.

```
public class Node
{
    public int Data { get; set; }
    public Node Next { get; set; }

    // Constructor to initialize a new node
    public Node(int data)
    {
        Data = data;
        Next = null; // The next pointer is initially null
    }
}

public class LinkedList
{
    private Node head; // The head (first node) of the linked list

    // Constructor to initialize an empty linked list
    public LinkedList()
    {
        head = null;
    }

   // Method to add a new node at the head of the linked list
    public void Add (int data)1
    {
        Node newNode = new Node (data);
        newNode.Next = head;
                newNode.Data = data;
                head = newNode;
    }


    // Method to traverse the linked list and print each node's data
    public void Traverse()
    {
        Node current = head;
        while (current != null) // Traverse the list until the end
        {
            Console.WriteLine (current.Data); // Print the data of the current node
            current = current.Next;                // Move to the next node
        }
    }
}
```

And a small example on how to use such a list:

```
class Program
{
    static void Main()
    {
        // Create a new linked list
        LinkedList list = new LinkedList();

        // Add some elements to the list
        list.Add(30);
        list.Add(20);
        list.Add(10);

        // Traverse and print the list elements
        Console.WriteLine("Traversing the linked list:");
        list.Traverse();
    }
}
```

In contrast to C, we do not need to free the memory allocated for the nodes by calling the constructor, because C# is a garbage collected language. That is, unused data on the heap is freed automatically. Though there are no explicit pointers in the code, the actual representation in memory is not much different than for the C version. The major difference is that a node in C# also contains a reference to its class methods via a mechanism we will discuss in Chapter **??**.

Using algebraic data types. In Haskell, lists are provided by the `Prelude` standard library, so one would not typically define them from scratch, and usually not limited to integers, but for the sake of the example, let us assume we want to implement a program with the same behaviour as the above C# program.

We again use`data` to define our type. We can use the type we are defining, `IntList`, recursively as argument to one of the data constructors of our new type:

```
data IntList
  = EmptyList
  | Cons Int IntList

traverseList :: IntList -> IO ()
traverseList EmptyList = return
traverseList (Cons data restList) = do
  putStrLn (show data)
  traverseList restList

main = do
  putStringLn "Traversing the linked list: "
  traverseList (Cons 30 (Cons 20 (Cons 10 EmptyList)))
```

There is no need to define a function corresponding to `Add`, as the construtor `Cons` has a similar functionality (of course, it does not actually alter the list it gets as arguments, as we're in a purely functional language, but instead returns a new list with the new element as head. In this respect, it's closer to the `cons` function of our C version. Just like in C#, there is no need to free a data structure, as Haskell, like almost all functional language, is also garbage collected.

9.2 The Essence of Algebraic Data Types

Algebraic data types are a form of composite data types, originally introduced in the seventies by the functional language Hope and quickly picked up by other functional languages. They have now found their way into many modern non-functional languages, such as C#, Swift, and Rust. The main reason algebraic data types became popular beyond the FP community is that pattern matching provides an extremely convenient way to inspect, traverse and decompose ADT values.

To investigate algebraic data types, we extend our MinHs language to support them in a very basic fashion. In particular, we do not add full pattern matching to MinHs as it is essentially syntactic sugar. A more user friendly language with pattern matching can be translated to the MinHs language. In fact, in many aspects, MinHs is very similar to the internal core language used by the Glasgow Haskell Compiler, for example.

What we need in the language are products to bundle values of different type together, sums to express variants, recursive types, and finally a unit type. We will show that these are sufficient to express types corresponding to all the composite types discussed.

9.2.1 Products

In Haskell, we have tuples of different arity, to bundle values of many different type in one value. However, if we do not care about convenience, just expressiveness, then simple (binary) pairs are sufficient: if we want to combine more than two types, we can just have nested pairs. As we want to keep our MinHs extension as small as possible, we also only introduce a pair constructor.

The type of pairs is called a product type, for reasons that will become clear later. The type of the pair of τ_1 and τ_2 is therefore written `(`τ_1 `*` τ_2`)`[2], in contrast to Haskell.

We also introduce a data constructor, `( , )` to MinHS' concrete syntax, and the corresponding binary abstract syntax operator `Pair` , which produces values of a product type.

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (\texttt{Pair}\ e_1\ e_2) : (\tau_1 * \tau_2)} \qquad (\textit{pair type-1})$$

We can then combine more than two types simply by using nested products: a type `(Int * (Int * Int))` bundling three integer values can be constructed with `(3, (4, 5))`.

Once we have a pair value, how can we retrieve the individual components in the absence of pattern matching? To this end, we introduce a bit more syntax for some built-in functions, `fst` and `snd` (`Fst` and `Snd` in abstract syntax), which, given a pair, return the first or second component of the pair respectively:

$$\frac{\Gamma \vdash e : (\tau_1 * \tau_2)}{\Gamma \vdash (\texttt{Fst}\ e) : \tau_1} \qquad (\textit{pair type-3})$$

$$\frac{\Gamma \vdash e : (\tau_1 * \tau_2)}{\Gamma \vdash (\texttt{Snd}\ e) : \tau_2} \qquad (\textit{pair type-3})$$

The evaluation rules for a strict semantics of pairs, given as big step semantics here, are pretty straightforward. First, we extend the set of values with pairs. That is, if v_1 is a value and v_2 is a value, then `(Pair` v_1 v_2`)` is a fully evaluated value as well. Data constructors, like `Pair`, differ from regular operators in that they just hold on to their argument values, and do not compute any new values.

A pair is therefore fully evaluated if its components are fully evaluated:

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{(\texttt{Pair}\ e_1\ e_2) \Downarrow (\texttt{Pair}\ v_1\ v_2)} \qquad (\textit{pair bstep-1})$$

and `fst` and `snd` extract the components of a pair:

$$\frac{e \Downarrow (\texttt{Pair}\ v_1\ v_2)}{(\texttt{Fst}\ e) \Downarrow v_1} \qquad (\textit{pair bstep-2})$$

$$\frac{e \Downarrow (\texttt{Pair}\ v_1\ v_2)}{(\texttt{Snd}\ e) \Downarrow v_2} \qquad (\textit{pair bstep-3})$$

Note that again, if a MinHs expression is well-typed according to our static semantics rules, the arguments to `fst` and `snd` will always evaluate to a pair (if the evaluation terminates).

9.2.2 Sums

As with pairs, variant types in actual programming languages allow to express the union between many types, but support for binary variants is sufficient, because they can be nested too. The Haskell datatype `Either`, defined in the Prelude, is such a type constructor which offers a binary variant:

[2]Product types are analogous to cartesian products in set theory, for those that are familiar with sets.

```
  data Either a b = Left a | Right b
```

We could have defined the `Variant` type in terms of `Either`:

```
type Variant = Either Int Float
```

Although, of course, our original `Variant` type has more meaningful data constructor names. We add a binary sum type, denoted by the infix type constructor `+`, which corresponds to `Either`, and introduce similar data constructors as built-ins, namely `Inl` and `Inr`.

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash (\texttt{Inl}\ e) : (\tau_1 + \tau_2)} \qquad (\textit{sum type-1})$$

$$\frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash (\texttt{Inr}\ e) : (\tau_1 + \tau_2)} \qquad (\textit{sum type-2})$$

Once again, we can combine multiple types to give n-ary sums via nesting.

Instead of function-like destructors for sum types, we will introduce a very limited kind of pattern matching in the form of a restricted `Case` expression, with exactly two alternatives: one for `Inl`, one for `Inr`. For example

```
  recfun sumToProd :: (Int + Int -> Bool * Int) e
      = case e of Inl u  -> (False, u)
                  Inr v  -> (True,  v);
```

The expression `sumToProd (Inl 5)` evaluates to `(False, 5)`, and `sumToProd (Inr 5)` to `True, 5)` The abstract syntax of a case expression is as follows: $(\texttt{Case}\ \tau_1\ \tau_2\ e\ (x.e_1)\ (y.e_2))$, where the input sum value e is of type $(\ \tau_1 + \tau_2\ )$, and e_1 which may contain a free occurence of the variable x ,is the alternative for `Inl`, and e_2 which may contain a free occurence of the variable y, is the alternative for `Inr`.
Typing rules:

$$\frac{\Gamma \vdash e : (\tau_1 + \tau_2) \quad \Gamma \cup \{x : \tau_1\} \vdash e_1 : \tau \quad \Gamma \cup \{y : \tau_2\} \vdash e_2 : \tau}{\Gamma \vdash (\texttt{Case}\ \tau_1\ \tau_2\ e\ (x.e_1)\ (y.e_2)) : \tau} \qquad (\textit{sum type-3})$$

For the dynamic semantics of values of sum type, we also need to extend the set of values. This time, we add terms of the form $(\texttt{Inl}\ v)$ and $(\texttt{Inr}\ v)$ to the set of values, where v is also a value.

$$\frac{e \Downarrow v}{(\texttt{Inl}\ e) \Downarrow (\texttt{Inl}\ v)} \qquad (\textit{sum bstep-1})$$

$$\frac{e \Downarrow v}{(\texttt{Inr}\ e) \Downarrow (\texttt{Inr}\ v)} \qquad (\textit{sum bstep-2})$$

Case expressions evaluate their first argument, check if the result has the form `Inl` or `Inr`, take the argument of the term, and bind that value to the variable of the second or third argument, respectively.

$$\frac{e \Downarrow (\texttt{Inl}\ v) \quad e_1[x := v] \Downarrow r}{(\texttt{Case}\ \tau_1\ \tau_2\ e\ (x.e_1)\ (y.e_2)) \Downarrow r} \qquad (\textit{sum bstep-3})$$

$$\frac{e \Downarrow (\texttt{Inr}\ v) \quad e_2[y := v] \Downarrow r}{(\texttt{Case}\ \tau_1\ \tau_2\ e\ (x.e_1)\ (y.e_2)) \Downarrow r} \qquad (\textit{sum bstep-4})$$

9.2.3 Unit Type

Could we define a type with only three values? At the moment, the type with the fewest elements we've got is `Bool`, which has two values, and both products and sums of `Bool` give us a type too large with four values.

The way we resolve this is to introduce a new type, called `Unit`, sometimes written ⊤, which has exactly one value - `()`. It can be thought of as the "empty" tuple, and corresponds to a `void` type in C.

Theer is only a single typing rule:

$$\frac{}{\Gamma \vdash \texttt{()} : \texttt{Unit}} \quad (\textit{unit type})$$

The dynamic semantics is trivial as well: we just add `()` to the set of values, and the only evaluation rule maps it to itself:

$$\frac{}{\texttt{()} \Downarrow \texttt{()}} \quad (\textit{unit bstep})$$

With this type, we can construct a types with a finite number of values, corresponding to unnamed enumeration types. For example, `( Unit + Unit )` has the elements `Inl ()` and `Inr ()`, and `( Unit + ( Unit + Unit ) )` has exactly three values: `(Inl ())`, `(Inr (Inl ()))`, `(Inr (Inr ()))`.

If we did not already have the type *Bool* in our language, we could model it with `( Unit + Unit )`, and interpret `(Inl ())` as `True`, `(Inr ())` as `False` (or the other way around). Then *if*-expressions of the form

```
if e
   then e1
   else e2
```

could be replaced by:

```
case e of
  Inl v -> e1
  Inr v -> e2
```

For a fresh variable name v which does not occur elsewhere in the expression. Here, the actual argument of the data constructors is not interesting. In both cases, `v` will be `UnitEl`, and since it's fresh, it is also not used in the two branches.

9.2.4 Recursive Types

With sum, product and the unit type, we can define any kind of data type with a finite number of elements. But what about defining recursive types, like lists or tree structures? In Haskell, we defined the type

```
data IntList
  = EmptyList
  | Cons Int IntList
```

where `IntList` occured recursively in its own definition. In MinHs, we also need a way to recursively refer to the type we are currently defining. We cannot do that in MinHs yet, though, since we do not have names for types: all composite types in MinHS are anonymous! But we can extend MinHs by something similar to `Recfun`, but which works on the type level. We call this type system feature `Rec` (in type theory, this is sometimes written as μ), which binds a type name (just as `Recfun` binds a function name) locally, to be used inside its body. So the type below corresponds to the Haskell `ListOfInt` type:

```
  Rec IntList. ((Int * IntList) + ())
```

It is a recursive type either containing a unit value, or a pair of `Int` and a second value of list type. The abstract syntax for recursive types with bind the type variable t in the type expression τ is written:

$$\texttt{(Rec (}t.\tau\texttt{))}$$

But how do we represent values of recursive type? For example, the value that corresponds to the empty list? The value `Inr ()` is just of type (τ + ()), for any type τ. It is not of type `Rec`. So we need a data constructor to inject it into a recursive type. We call this constructor `Roll`. Its typing rule is as follows:

$$\frac{\Gamma \vdash e : \tau[t := \texttt{Rec}(t.\tau)]}{\Gamma \vdash \texttt{(Roll } e\texttt{)} : \texttt{(Rec (}t.\tau\texttt{))}} \qquad (\textit{rec type-1})$$

For example, since we can derive that

$$\{\} \vdash \texttt{Inr ()} : \texttt{(}\tau\texttt{+())}$$

for any type τ, we can derive

$$\{\} \vdash \texttt{Inr ()} : \texttt{((Rec (}\mathit{listOfInt}\texttt{.((Int*}\mathit{listofInt}\texttt{)+())))+())}$$

and therefore, according to Rule (*rec type-1*),

$$\{\} \vdash \texttt{Roll (Inr ())} : \texttt{(Rec (}\mathit{listOfInt}\texttt{.((Int*}\mathit{listofInt}\texttt{)+())))}$$

The operator `Unroll` does exactly the opposite to `Roll`: it unpacks a recursive value. Therefore, the typing rule is also basically the inverse:

$$\frac{\Gamma \vdash e : \texttt{(Rec (}t.\tau\texttt{))}}{\Gamma \vdash \texttt{(Unroll } e\texttt{)} : \tau[t := \texttt{Rec}(t.\tau)]} \qquad (\textit{rec type-2})$$

That is, when we `Unroll` a recursive type, the recursive name variable (t) is replaced with the entire recursive type. Each `Unroll` eliminates one level of recursion.

We can define a value of our linked list type, written in Haskell as `[3, 4]`, as follows:

$$\texttt{(Roll (Inl 3 (Roll (Inl 4 (Roll (Inr ()))))))}$$

Not very pretty, but isomorphic to the Haskell list. In contrast MinHs' to sum-types, with are similar to the Haskell predefined `Either` type constructor, or the `Sum` type constructor we defined, there is nothing in Haskell which corresponds to the `Rec` type constructor and it cannot be defined.

The big step semantics rules are (as with constructors in general), not very exciting:

$$\frac{e \Downarrow v}{\texttt{(Roll } e\texttt{)} \Downarrow \texttt{(Roll } v\texttt{)}} \qquad (\textit{rec bstep-1})$$

$$\frac{e \Downarrow \texttt{(Roll } v\texttt{)}}{\texttt{(Unroll } e\texttt{)} \Downarrow v} \qquad (\textit{rec bstep-2})$$

Consider the following MinHs implementation of a the function `length`, which takes a list, represented by the type discussed previously, and calculates its length:

```
recfun length :: ((Rec l. Unit + (Int * l)) -> Int)  xs =
  case (Unroll xs) of
    Inl unit -> 0
    Inr tail -> 1 + length (snd tail)
```

Here, `Unroll` is applied to `xs`, to get to the inner sum type. Then `case` checks which of the two alternatives it is, and, if necessary, calls `length` recursively.

9.3 Haskell Datatypes

As basic as MinHs still is, we can now define many of the data types we can define in Haskell. To show this, we will demonstrate a technique which gives an isomorphic type in MinHS to any (non-polymorphic) type in Haskell (ignoring the fact that Haskell structures are lazy for now).

Suppose we have the following Haskell type:

```
data Foo = Bar Int Bool | Baz | Herp Foo
```

First, replace all data constructors with `Unit`:

```
data Foo = Unit Int Bool | Unit | Unit Foo
```

Then, multiply the constructors with all of their arguments:

```
data Foo = Unit * Int * Bool | Unit | Unit * Foo
```

Then, replace all the alternatives with sums:

```
data Foo = Unit * Int * Bool + Unit + Unit * Foo
```

Apply algebraic simplification (usually just `Unit`$*x \Leftrightarrow x$):

```
data Foo = Int * Bool + Unit + Foo
```

Then, replace the recursive references (if any) with a `Rec` construct:

```
Rec t. (Int * Bool + Unit + t)
```

As can be seen from the above example, this technique will convert non polymorphic, non parameterised Haskell types into isomorphic MinHS ones.

10 References

Most programming languages support so-called reference types, which do not directly contain a value, but rather point to a location where a value is stored. They typically offer at least the following functionality:

- Create a new reference which points to a value of type a: allocates the space for a value of type a in memory, and return a reference to that location. In some languages, the allocated memory has to be initialised with a default or provided value of type a.
- Retrieve the value a reference is pointing to by dereferencing it.
- Update the value a reference is pointing to.

References are useful to allow sharing of large objects without needing to copy them, or to implement dynamic data structures, such are trees and lists, if the language does not already provide other means to do so. References are often implemented as pointers: a memory address, together with some meta information, for example the size of the data it points to.

10.1 References in functional languages

In Haskell, we don't have built-in explicit reference types, but `Data.IORef` provides the type `IORef a` as an abstract data type, with the following operations (among a few others) defined on it:

```
newIORef   :: a -> IO (IORef a)        -- create new reference, initialise value
readIORef  :: IORef a -> IO a          -- retrieve the value pointed to by the ref.
writeIORef :: IORef a -> a -> IO ()    -- update the value pointed to by the ref.
```

The type of the operations in Haskell makes it clear that these operations have a side effect on the world, or depend on the state of the world.

```
main = do
  xRef <- newIORef 5
  x1   <- readIORef xRef
  writeIORef xRef 10
  x2   <- readIORef xRef
  putStrLn ("x1 : " ++ show x1 ++ " x2: " ++ show x2)
```

In the program above, the value of `x1` will be `5`, and of `x2` it will be `10`, just as in the example below, as `xRef` and `yRef` refer to the same location:

```
main = do
  xRef <- newIORef 5
  x1   <- readIORef xRef
  let yRef = xRef
  writeIORef yRef 10
  x2   <- readIORef xRef
  putStrLn ("x1 : " ++ show x1 ++ " x2: " ++ show x2)
```

Behind the scenes, Haskell is using references as default representations for basic values, data structures, and function closures. For example, if we add an element to a list, as in the following code snippet:

```
let
  xs = [1,2,3]
  ys = 0 : xs
in ...
```

then `xs` is not copied to create `ys`. However, this is not observable from Haskell, as we cannot destructively update the lists. Even numbers and other values of basic type have a boxed representation by default in Haskell: an `Int` is actually by default represented as reference to a machine integer. The internal representation of the lists `xs` and `ys` in the example therefore is as shown schematically in Figure 10.1.

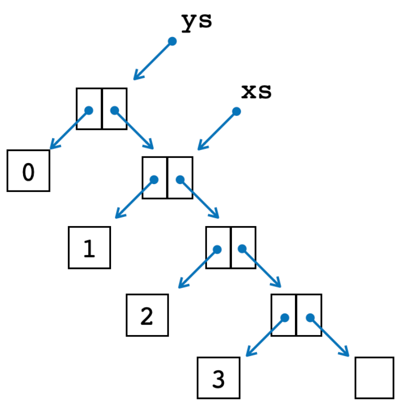


Figure 10.1: Internal representation of a shared list in Haskell

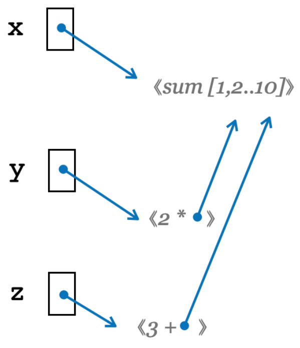


Figure 10.2: Shared lazily evaluated values

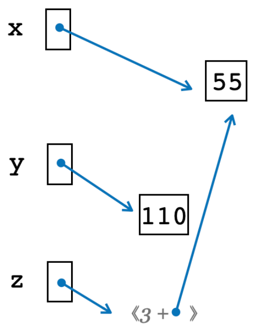


Figure 10.3: Shared lazily evaluated values

Computations in Haskell are evaluated lazily. A naive implementation of laziness could result in recomputing values multiple times, for each usage. With the boxed representation, however, such unevaluated computations are also shared, and evaluating it once makes the value available to all usages. For example, in the code snippet below, if `y` is evaluated, it triggers the evaluation of `x`:

```
let x = sum [1,2..10]
    y = 2 * x
    z = 3 + x
    ...
```

After that, the reference of `x` in the expression bound to `z` points to the evaluated result (see Figure 10.2), so when `z` is evaluated, there is no need to calculate the sum again (Figure 10.3).

10.2 References in stateful languages

In a language without implicit side effects, the programmar doesn't need to know whether data is represented internally via a reference, or directly as value, at least not with respect to the semantics of a program. Since dereferencing is expensive, it does, however, affect the performance of a program! In a language with destructive updates, however, it is important to know, as this affects the semantics of assignments and function calls.

Value and reference types. Languages, even closely related ones, differ in which types are reference types, and which are value types. Assignments of reference types only copy the reference, not the value it points to. In C#, and in Java, classes, for example, are reference types, but in C++, they are value types. Now consider the following C# code snippet:

```
public class MyClass
{
  public int value;
}

public class Program
{
  public static void Main()
  {
    MyClass ob1 = new MyClass();
    ob1.value   = 20;
    MyClass ob2 = ob1;
    ob1.value   = 10;
    Console.WriteLine("ob2.value = {0}", ob2.value);
  }
}
```

Any changes to `obj2` affect `obj1` and vice versa, as both variables refer to the same object. The output of the program in C# (and the corresponding program in Java) is therefore

```
obj2.value = 10
```

In C++, in contrast, given the code below

```
class MyClass
{
  public: int value;
};

int main() {
    MyClass ob1;
    ob1.value   = 20;
    MyClass ob2 = ob1;
    ob1.value   = 10;
    std::cout << "obj2.value = "<< ob2.value;
    return 0;
}
```

the object is copied on assignment (as it is a value type), so the output would is

```
obj2.value = 20
```

In Swift, an object oriented language with many functional features, value types include structs, enums and tuples, as well as types like arrays, strings, ints, and such, which are implemented in terms of these. Classes are reference types in Swift, just like in C# and Java.

Pointers in C. In C, there are no implicit reference types. Instead we have explicit pointers, denoted by `*`, preceded by the type it points to. For example,

```
int * x_ptr;  // declare a pointer to int
```

declares a variable `x_ptr`, which points to an `int` value. Since C doesn't enforce the initialisation of a variable value, the value of `x_ptr` is at that point undefined, so may not contain a valid address. The allocation of memory has to be done explicitly in C:

```
x_ptr  = (int*) malloc (sizeof (int));   // allocate space for an int
```

The `*` operator is also used to dereference a pointer in C:

```
*x_ptr = 6;             // dereference x_ptr, and write 6 into the location
*x_ptr = 2 * (*x_ptr); // double the value x_ptr points to
```

This calls `malloc` to allocate enough space for an `int` value, and cast the return type to an `int` pointer. The allocation may fail, so in general, the programmer has to check if the allocation was successful before continuing. Once the allocated memory is not needed anymore, the programmer has to call `free (x_ptr)` to release the memory (but that does not change the value of `x_ptr` itself). Accessing memory which has been freed results in undefined behaviour. Failing to release memory may lead to a so-called memory leak: a long running application may allocate more and more memory, even though it only needs to use a fraction of it.

Programming languages which put the burden of explicitly allocating and freeing memory on the programmer are said to have manual memory management. If done correctly, it is possible to write code which is very memory efficient. However, keeping track of the first and last use of each object and ensuring that no invalid pointer is dereferenced anywhere in the program is difficult and error prone. Therefore, most modern programming languages automatic memory management, where the runtime system automatically takes care of the allocation and release of memory.

C allows pointer arithmetic: we can add an offset to an address and dereference the resulting address. If the resulting address is outside of the allocated block, the result of dereferencing the address is undefined. In C, we can also get the address of a variable using the `&` operator. For example, after executing the following program snippet in C

```
int * x_ptr;
int x = 5;
x_ptr = &x;
*x_ptr = 7;
```

the variable `x` has the value `7`, as `x_ptr` points to `x`.

Pass-by-reference versus pass-by-value. In TinyC, like in many other languages, we defined function calls as pass-by-value (or call-by-value). In the example below, only the value of `y` is passed to `inc` and then bound to the new stack variable `x`. Incrementing the value of `x` does not affect the value of `y`

```
int inc (int x) {
  x = x + 1;
}

int y = 10;
inc (y);
```

C++, Java and C# are also pass-by-value. However, in the latter two languages, since classes are reference types, so if we pass an object to a function or method, we pass the value of the reference, and any update of the object in the function alters the object we passed. In C++, classes are value types, so an object gets copied when we pass it.

Fortran is one of the few languages which uses pass-by-reference including for values of basic types, and even for constant values!

10.3 References for TinyC

To investigate some of the complexities references introduce, let us add them in a restricted way to our language. Reference types are abstract, that is, TinyC supports the basic functionality of references: creating a reference, reading the value it points to, and updating the value, but it does not support pointer arithmetic. This way, the language is still type safe with no undefined behaviour. Nevertheless, adding them in this way to the language already demonstrates how they complicates our execution model.

The BNF below gives the definition of TinyC with references, which is mostly identical to TinyC, with the differences *highlighted*:

$$
\begin{array}{lcl}
\textit{prgm} & ::= & \textit{gdecs}\ \textit{rdecs}\ \textit{stmt} \\
\textit{gdecs} & ::= & \epsilon \mid \textit{gdec}\ \textit{gdecs} \\
\textit{gdec} & ::= & \textit{fdec} \mid \textit{vdec} \\
\textit{vdecs} & ::= & \epsilon \mid \textit{vdec}\ \textit{vdecs} \\
\textit{type} & ::= & \texttt{int} \mid \texttt{int}\,* \\
\textit{vdec} & ::= & \texttt{int}\ \textit{Ident} = \textit{Int}; \\
\textit{rdecs} & ::= & \epsilon \mid \textit{rdec}\ \textit{rdecs} \\
\textit{rdec} & ::= & \texttt{int}*\ \textit{Ident} = \texttt{alloc}(\textit{Int}); \\
\textit{fdec} & ::= & \texttt{int}\ \textit{Ident}\ (\textit{arguments})\ \textit{stmt} \\
\textit{stmt} & ::= & \textit{expr}; \mid \texttt{if}\ \textit{expr}\ \texttt{then}\ \textit{stmt}\ \texttt{else}\ \textit{stmt}; \mid \texttt{return}\ \textit{expr}; \mid \\
 & & \{\ \textit{vdecs}\ \textit{rdecs}\ \textit{stmts}\ \} \mid \texttt{while}\ (\ \textit{expr}\ )\ \textit{stmt} \\
\textit{stmts} & ::= & \epsilon \mid \textit{stmt}\ \textit{stmts} \\
\textit{expr} & ::= & \textit{Int} \mid \textit{Ident} \mid *\,\textit{Ident} \mid \textit{expr} + \textit{expr} \mid \textit{expr} - \textit{expr} \mid \\
 & & \textit{Ident} = \textit{expr} \mid *\,\textit{Ident} = \textit{expr} \mid \textit{Ident}\ (\textit{exprs}) \\
\textit{arguments} & ::= & \epsilon \mid \texttt{int}\ \textit{Ident}\ ,\ \textit{arguments} \\
\textit{exprs} & ::= & \epsilon \mid \textit{expr}\ ,\ \textit{exprs}
\end{array}
$$

We extend the set of legal types with references to `int` values, denoted `int*`. When declaring a variable of type `int*`, it has to be initialised by calling `alloc`, which always allocates memory for one `int`, and initialises the memory cell with the `int` passed in as argument. For example,

```
int * xRef = alloc (5);  // xRef now points to a location with content '5'
```

reserves memory for the variable `xRef`, reserves memory for an `int`, initialises the content of the latter memory location with the value `5`, and sets `xRef` to point to that location.

We also add one new operation to the language, namely dereferencing (`*`, similar to C), which can only be applied to identifiers of reference type, and which retrieves the `int` value of the location it points to. So, after executing the code below

```
int * xRef = alloc (5);
int   y    = 7;

y = *xRef + y;
```

The variable `y` has the value `12`.

10.3.1 Static semantics of references

Adding references to the language obviously makes the type checking more complex. With just a single type, all we had to do for variables was to check if they were in scope, and for procedure calls to check if the number of actual parameters was the same as the number of formal parameters in the declaration. Now, we have to check that the type of each variable is correct and for every procedure call if each argument is also type correct. The sets V and F where we keep track of this information then have the following form (where *type* is either `int` or `int*`):

- Variables and their type $V = \{x_1 : \textit{type}_1\ ,\ x_2 : \textit{type}_2, \ldots\}$
- Functions and their argument and return type $F = \{f_1\colon (\textit{type}_1, \ldots, \textit{type}_n) \to \textit{type},\ f_2\colon \ldots\}$

We also have to make sure that we only apply arithmetic operations to `int` values, and only dereference actual reference variables. For this to work, we have to replace the *ok* judgment with an actual typing judgement, just as we did when moving from the arithmetic expression language to MinHs. As this not much different to what we did in MinHs, we don't list the new rules for the static semantics here, but we assume for the remainder of the chapter that the program we're looking at is type correct.

As mentioned before, in C it is possible do pointer arithmetic. While this provides a great deal of flexibility in the language, and is sometimes necessary for low-level code, it is also problematic as it can introduce undefined behaviour. In TinyC, our reference type is abstract, as we cannot inspect or alter a reference directly.

10.3.2 Dynamic semantics of references

For the dynamic semantics, even simple references of the form we introduced lead to some interesting problems, and force us to think about the way we modelled memory in our formalisation of TinyC.

Essentially, we modelled the memory as a stack. Consider the block-rule:

$$\frac{(g.l, ss) \Downarrow (g'.l', rv)}{(g, \{l\ ss\}) \Downarrow (g', rv)} \qquad (tc\text{-}7)$$

We push the new variables l on the stack g, execute the statements in the block, and pop the bindings l' off again when we return from the block. We can view this as temporarily allocating memory for the local variables, and freeing it again. That was safe, because there is no way to reference the memory for the variables in the block from outside the block. Now consider the following code:

```
int *xRef = alloc (10);
int z = 3;
{
  int *yRef = alloc (5);
  xRef = yRef;            // xRef and yRef now both point to the same location
}
z = *xRef;
```

If we allocate the memory for both the variable `yRef` and the location it points to our stack, and free that stack frame when exiting the block, we have a problem, since `xRef` still points to it. We therefore need to model memory which persists after returning from a procedure call. To this end, we extend the state of the machine with a third component, h_k, which is a map from addresses from 0 to $k-1$ to integer values. Just as g is a simple model of a runtime stack, h can be viewed as a (again very simple) heap.

We represent the state now as a triple:

$$(g \blacklozenge h_k, s)$$

On the persistent store h_k we have two operations, lookup of a value and extension of the store, just like on g. In contrast to g, however, lookup does not work on variable names, but on addresses, or keys, for which we use integer values.

Part of the state we store is the next available key k. Initially, the first available address is 0, and we increment it whenever `alloc` is called.

For TinyC-programs without references, we defined three judgements – one for program execution, one for statement execution, and expression evaluation:

$$p \Downarrow (g, rv)$$

$$(g, stmt) \Downarrow (g', rv)$$

$$(g, expr) \Downarrow (g', v)$$

We extend each of these judgements with a persistent store h_k

$$p \Downarrow (g \blacklozenge h_k, rv)$$

$$(g \blacklozenge h_k, stmt) \Downarrow (g \blacklozenge h'_{k'}, rv)$$

$$(g \blacklozenge h_k, expr) \Downarrow (g' \blacklozenge h'_{k'}, v)$$

The bigstep semantics rules for all the language components we introduced in Chapter 6 stay almost as they are, with the difference that we thread the persistent store through. For example, the rule

$$\frac{(g, s) \Downarrow (g', rv)}{g\ s \Downarrow (g', rv)} \qquad (tc\text{-}1)$$

becomes

$$\frac{(g \blacklozenge \{\}_0, s) \Downarrow (g' \blacklozenge h_k, rv)}{g\ s \Downarrow (g' \blacklozenge h_k, rv)} \qquad (tcRef\text{-}1)$$

where $\{\}_0$ denotes the empty persistent store.

We also need to add some new rules for the new language components. We have to specify how to handle declarations and initialisations of references of the form

```
int * xRef = alloc (5);
```

via judgements of the form

$$(g \blacklozenge h_k, rdec) \Downarrow (g' \blacklozenge h'_{k'}, v)$$

and for sequences of reference declarations:

$$(g \blacklozenge h_k, rdecs) \Downarrow (g' \blacklozenge h'_{k'}, v)$$

Declaration of reference variables. For a single declaration and initialisation, we set the reference variable to the current key of the store k, allocate space by incrementing that key, and store that key in g and the initial value v at the index k in h:

$$\frac{}{(g \blacklozenge h_k, \texttt{int} * x = \texttt{alloc}(\ v);) \Downarrow ((g.\texttt{int} * x = k) \blacklozenge (h.k = v)_{k+1}, v)} \qquad (tcRef\text{-}16)$$

Sequences of declarations are just handled as usual, by threading the state of g and h:

$$\frac{}{(g \blacklozenge h_k, \circ) \Downarrow (g \blacklozenge h_k, 0)} \qquad (tcRef\text{-}17)$$

$$\frac{(g \blacklozenge h_k, r) \Downarrow (g' \blacklozenge h'_{k'}, v) \quad (g' \blacklozenge h'_{k'}, rs) \Downarrow (g'' \blacklozenge h''_{k''}, v')}{(g \blacklozenge h_k, r\ rs) \Downarrow (g'' \blacklozenge h''_{k''}, v')} \qquad (tcRef\text{-}18)$$

Operations on references. The two operation we allow on references (apart from copying) is to dereference it, that is, to retrieve the value the reference points to, and to update the value it points to.

Dereferencing has no side effect on the state of the stores, and just retrieves the key value j from g, and then looks up the corresponding value in h:

$$\frac{g@x = j \quad h@j = v}{(g \blacklozenge h_k, *x) \Downarrow (g \blacklozenge h_k, v)} \qquad (tcRef\text{-}19)$$

$$\frac{g@x = j \quad (g \blacklozenge h_k, e) \Downarrow (g' \blacklozenge h'_{k'}, v)}{(g \blacklozenge h_k, *x{=}e) \Downarrow (g' \blacklozenge (h'@j \leftarrow v)_{k'}, v)} \qquad (tcRef\text{-}20)$$

Updating the contents can have an indirect effect as side effect of evaluating the expression e, both on the content and the size of h.

10.4 Memory management

We introduced references into TinyC, but restricted their usage such that the resulting language is still type safe and does not have undefined behaviour as every reference points to a valid location, but it comes at a cost. In our semantics, the persistent memory h keeps growing in size. We could introduce a `free` function to let the programmer take care of it, but then we have a similar problem as C: we have to check every reference if it is still valid, or the language is not type safe anymore. Furthermore, we don't have a guarantee that the programmer actually does free the memory.

If we want to avoid manual memory management for language safety reasons, we can alternatively implement automatic memory management. In our execution model for TinyC, we know that for a state $g \blacklozenge h_k$, for any address i with $0 \leq i < k$, if $\texttt{int} * x = i \not\in g$ for some variable x, then i is not reachable anymore, and the memory can be reused. The question is how to identify those unreachable addresses in the heap?

One way to implement garbage collection is via reference counting, where the runtime system keeps track of the number of references pointing to a location at any point during execution. So whenever a reference variable gets popped from the state, the counter to its location is adjusted, as well as when the number of references change due to assignment. As soon as a reference counter goes to zero, its address becomes available again.

An alternative option is tracing garbage collection. There are many ways to do this, but generally it means that the execution is stopped at certain intervals and then all the references from the stack g into h are traced, and the objects they point to copied to a different part of the memory. Once that's done for the whole stack g, the part of h where the objects used to be can be freed. On our simple memory model, we can define tracing garbage collection as a mapping from a state $g \blacklozenge h_k$ to a new, corresponding state $g' \blacklozenge h'_{k'}$ such that looking up any integer variable on the stack in g will yield the same result as in g', and any reference x stored in g will point to the same value in h and the reference x stored in g' in h'.

We denote this correspondence of two states as $g \blacklozenge h_k \approx g' \blacklozenge h'_{k'}$, where the relation $\approx$ is formally defined via the following inference rules:

$$\frac{}{\circ \blacklozenge h_k \approx \circ \blacklozenge h'_{k'}} \qquad (\textit{tcEq-1})$$

$$\frac{g \blacklozenge h_k \approx g' \blacklozenge h'_{k'}}{g\,\texttt{.int}\ x\ \texttt{=}\ v \blacklozenge h_k \approx g'\,\texttt{.int}\ x\ \texttt{=}\ v \blacklozenge h'_{k'}} \qquad (\textit{tcEq-2})$$

$$\frac{h_k@k = h_{k'}@k' \qquad g \blacklozenge h_k \approx g' \blacklozenge h'_{k'}}{g\,\texttt{.int}\ \texttt{*}x\ \texttt{=}\ k \blacklozenge h_k \approx g'\,\texttt{.int}\ x\ \texttt{=}\ k' \blacklozenge h'_{k'}} \qquad (\textit{tcEq-3})$$

Note that the rules actually require that the order of variables in g and g' are exactly the same.

In a fully fledged language the size of the memory cells allocated is not uniform, as they support more than a single type, as well as compound types like structures and records. So when freeing memory in h, the memory can get fragmented over time, such that there may still be plenty of available space, but fragmented into too small chunks. Tracing garbage collection addresses this problem, but pure reference counting approaches don't. We also typically have recursive data structures. That is, some of the values in h are references themselves, so when tracing a reference from the stack, we have to recursively trace all the references it points to, and so on.

Note that the problem of automatic memory management is also present in languages without explicit references. For example, the Haskell runtime system cannot release the memory referenced by a variable (such as `xs` or `ys` in Figure 10.1) directly or indirectly when the variable is deleted from the stack, as there might be other variables still on the stack which refer to some or all of that memory. Programs written in languages with automatic memory management can exhibit memory leaks as well. In Haskell, this is sometimes the case due to interaction with laziness: say we calculate the sum of a large list. This computation does not get fully evaluated until we actually look at the result of the sum, so the list is kept around until then, even if there is no other reference to any part of it.

11 | Polymorphism

Polymorphism is the ability to exist in different forms. In the context of programming languages, polymorphism refers to a function or value symbol representing items of different type. There are various forms of polymorphism, and as with many concepts in programming languages, the meaning of a term differs in different communities.

11.1 Parametric Polymorphism

Polymorphism allows some form of generic programming, where values of *different types* can be manipulated by the *same function*. Parametric polymorphism, sometimes called generics in OO languages, is the most straightforward form of polymorphism where a function can be declared to operate over any type at all. Other flavours of polymorphism include subtyping, overloading and method overriding.

Coming back to our discussion of parametric polymorphism, consider the swap function (using Haskell syntax, with pattern matching here):

```
swap (x,y) = (y, x)
```

What would the type of this function be? In a monomorphic language like MinHS, we couldn't write this function generically. We would have to have a variety of functions, $\texttt{swapBI} : \texttt{Bool} * \texttt{Int} \to \texttt{Int} * \texttt{Bool}$, $\texttt{swapIB} : \texttt{Int} * \texttt{Bool} \to \texttt{Bool} * \texttt{Int}$, and so on - a total of T^2 functions where T is the number of types in the language[1]. This is obviously highly impractical, seeing as all these functions have the same implementation. What we want is to express a type $\texttt{swap} : \alpha * \beta \to \beta * \alpha$[2] all types α and β. That is what parametric polymorphism gives us.

Polymorphism is a prominent part of most modern programming languages. In C#, a generic swap operation, which swaps the elements in place, can be defined by declaring the type parameter (T) over which the operation is parametrised in angle brackets:

```
static void Swap<T>(ref T a, ref T b) {
  T temp;
  temp = a;
  a = b;
  b = temp;
```

When applying the generic operation, as in the code snippet the user has to instantiate the type parameter, as in the two examples below:

```
  int a, b;
  char c, d;
  a = 5;
  b = 10;
```

[1]Since we have products and sums, $T = \infty$

[2]We use greek letters for type variables to distinguish them from value variables whenever we refer to them, apart from when we give examples in the concrete syntax of a particular programming language.

```
  c = 'X';
  d = 'Y';

  Swap<int>(ref a, ref b);
  Swap<char>(ref c, ref d);
```

The declaration of a the generic swap operation in Swift looks quite similar:

```
func swap<T>(_ a: inout T, _ b: inout T) {
  let tmp = a
  a = b
  b = tmp
}
```

However, when applying a generic function in Swift, as the language is implicitly typed, it is not necessary to explicitly instantiate the type:

```
  var x = 3
  var y = 107

  swap(&x, &y)

  var str1 = "hello"
  var str2 = "world"
  swap(&str1, &str2)
```

The designers of the Go language initially chose not to support generics, stating that they wanted to keep the language simple – a decision which was very controversial. In 2018, the principal contributors of the language published a proposal to add generics as a new feature, and generics are part of Go since release v1.18 [13] in 2022.

11.1.1 Explicitly typed polymorphic MinHS

Currently, all functions in MinHS take some values of a concrete type, and return a value of a concrete type. A function `mkPair` which constructs a pair of two integers can be written as follows:

```
  mkPair = recfun f :: (Int -> Int -> (Int * Int)) x =
                    recfun g :: (Int -> (Int * Int)) y = (x, y)
```

To support parametric polymorphism in MinHs, we extend the syntax for functions slightly. In addition to having functions from values to values, like above, we include functions from types to values.

We introduce two new forms of expressions for the concrete syntax:

$$\texttt{Type}\ \alpha\ \texttt{in}\ e$$

introduces the type variable α, which can then be used in place of a regular type for a type annotation in the expression e. This is analogous to the `<T>` notation in Swift and C#, which introduces a type variable `T`. The correspondinghigher-order abstract syntax term is

$$(\texttt{Type}\ (\alpha.e))$$

The second new expression has the following concrete syntax

$$\texttt{Inst}\ e\ \tau$$

which instantiates the type variable which is bound in e to the type τ, similar to the instantiation of a type variable in C#. It's abstract syntax form is

$$\texttt{(Inst } e\ \tau\texttt{)}$$

For example, the polymorphic identity function in explicitly typed polymorphic MinHs can be written as

```
Type a in
   recfun id :: (a -> a) x = x
```

When we want to apply the function to, say, the boolean value `True`, we first have to instantiate the type variable to `Bool`, and then apply the resulting function to the value:

```
(Inst (Type a in
   recfun id :: (a -> a) x = x) Bool) True
```

Polymorphic Types

The identity function we defined above can take a value of any type as argument, and returns a value of that type. We therefore say that it has the type $\forall \alpha.\alpha \to \alpha$ (written `forall a.  a -> a` in concrete syntax and $\texttt{(}\forall\ \texttt{(}\alpha\texttt{.(}\alpha \to \alpha\texttt{)))}$ in higher-order abstract syntax).

So our types, in addition to base types and type constructors, such as function, pair and sum, can now also contain $\forall$-quantifiers and variables bound by such a quantifier. They are getting complex enough that it is worthwhile to define a judgement ***EPType***, formalising what exactly a legal type is in our explicitly typed polymorphic MinHs, and have a closer look at what a polymorphic type actually means.

We allow only type variables which are bound by a $\forall$-quantifier, just as we only allow value term variables which are bound. To check this, we need an environment to keep track of the variables which are bound in type terms.

We assume that we have just two base types, `Bool` and `Int` in our language, but it can be extended with any base type we want to have. The environment, denoted Δ in the rules below, is just a set of type variables:

$$\frac{\alpha \in \Delta}{\Delta \vdash \alpha\ \textit{\textbf{EPType}}} \qquad (\textit{EPType-1})$$

$$\frac{}{\Delta \vdash \texttt{Int}\ \textit{\textbf{EPType}}} \qquad (\textit{EPType-2})$$

$$\frac{}{\Delta \vdash \texttt{Bool}\ \textit{\textbf{EPType}}} \qquad (\textit{EPType-3})$$

$$\frac{\Delta \vdash \tau_1\ \textit{\textbf{EPType}} \quad \Delta \vdash \tau_2\ \textit{\textbf{EPType}}}{\Delta \vdash (\tau_1 \to \tau_2)\ \textit{\textbf{EPType}}} \qquad (\textit{EPType-4})$$

$$\frac{\Delta \cup \{\alpha\} \vdash \tau\ \textit{\textbf{EPType}}}{\Delta \vdash \texttt{(Rec (}\alpha\texttt{.}\tau\texttt{))}\ \textit{\textbf{EPType}}} \qquad (\textit{EPType-5})$$

$$\frac{\Delta \cup \{\alpha\} \vdash \tau\ \textit{\textbf{EPType}}}{\Delta \vdash \texttt{(}\forall\texttt{(}\alpha\texttt{.}\tau\texttt{))}\ \textit{\textbf{EPType}}} \qquad (\textit{EPType-6})$$

We don't list the rules for sum, product and unit type constructors here. The former two are the same as for the function type constructor, and for the unit type, it is just an axiom, as for other basic types.

Typing Rules

We only need two new typing rules for explicitly typed, polymorphic MinHS, namely one for `Type`, which just checks if the expression is well-typed, and the quantified type variable does not already occur free in the type environment Γ:

$$\frac{\Gamma \vdash e : \tau \quad \alpha \notin FV(\Gamma)}{\Gamma \vdash \texttt{(Type } (\alpha.e)\texttt{)} : (\forall(\alpha.\tau))} \quad (\textit{epMinHS type-1})$$

The rule for $\texttt{(Inst } e)\tau$ checks if e has the polymorphic type of the form $(\forall\ (\alpha.\tau'))$, in which case the instantiated expression has the type τ', where all free occurrences of α have been replaced with the type τ:

$$\frac{\Gamma \vdash e : (\forall(\alpha.\tau'))}{\Gamma \vdash \texttt{(Inst } e\ \tau\texttt{)} : \tau'[\alpha := \tau]} \quad (\textit{EPMinHS type-2})$$

Let us check how these rules work, and see if we can indeed derive the type of the identity function. In the higher-order abstract syntax term for the function

$$\texttt{(Type } (\alpha.\texttt{(Recfun } \alpha\ \alpha\ (id.(x.x)))))$$

we can see that `Type` binds the type variable α, which then appears as argument and result type of `Recfun`. The function and variable typing rules are the typing rules from monomorphic MinHs, just that now, we also have type variables in the environment:

$$\frac{\dfrac{\dfrac{x : \alpha \in \{x : \alpha, id : (\alpha \to \alpha)\}}{\{x : \alpha, id : (\alpha \to \alpha)\} \vdash x : \alpha}(\textit{mhs type-1})}{\{\} \vdash \texttt{(Recfun } \alpha\ \alpha\ (id.(x.x))) : (\alpha \to \alpha)}(\textit{mhs type-7}) \quad \alpha \notin FV(\{\})}{\{\} \vdash \texttt{(Type } (\alpha.\texttt{(Recfun } \alpha\ \alpha\ (id.(x.x))))) : (\forall(\alpha.(\alpha \to \alpha)))}(\textit{EPMinHS type-1})$$

The instantiation of the polymorphic function

$$\texttt{(Inst (Type } (\alpha.\texttt{(Recfun } \alpha\ \alpha\ (id.(x.x))))))\texttt{Int}$$

then has type

$$(\texttt{Int} \to \texttt{Int})$$

according to Rule (*EPMinHS type-2*).

Dynamic Semantics and Implementation

The dynamic semantics of our new language constructs is pretty boring. Not surprisingly, since this extension is really about the static semantics of the language. Terms of the form $\texttt{(Type } (\alpha.e))$ are final (i.e. do not evaluate further), and instantiation simply replaces any occurrence of the bound type α by the type τ it is instantiated with. Since the type of variables is ignored in the dynamic semantics, this has no consequences for the evaluation, it simply ensures that the terms stay well-typed (preservation).

The rules for instantiation in the structural operational semantics (see the MinHS notes) are as follows:

$$\frac{e \mapsto e'}{\texttt{(Inst } e\ \tau\texttt{)} \mapsto \texttt{(Inst } e'\ \tau\texttt{)}}(\textit{EPMinHS step-1})$$

$$\frac{}{\texttt{(Inst (Type } (\alpha.e)\ \tau)) \mapsto e[\alpha := \tau]}(\textit{EPMinHS step-1})$$

While it is easy to specify how the evaluation should behave, at least on this very abstract level, it is a different matter to actually implement polymorphic functions on a concrete machine. Consider even the simple example from the beginning of this chapter:

```
(Type a in
  (Type b in
    recfun swap :: ((a, b) -> (b, a)) xy =
      (snd xy, fst xy))
```

What should the code a compiler produces for this function look like? It has to allocate space for the returned pair, and copy the components of the arguments to the correct positions in the allocated memory location. Both allocation and copying depend on the actual content types of the pair. The types could be anything, from character, floating point values, to complex tree structures or even functions.

Haskell, as well as many other languages (in particular functional and OO languages), solve this by using a so-called boxed representation by default. Instead of passing around values directly, it passes references (or pointers) to these values. Since the size of a reference does not depend on what it points to, but only on the size of an address in the given system, the code is exactly the same. It just allocates space for two addresses, and copies the addresses to the appropriate position.

Using a boxed representation, on one hand, solves the problem in a simple and elegant manner. However, it does come at a cost. Whenever a value is used at runtime, the pointer has to be de-referenced, which is a relatively expensive operation. To alleviate this problem, optimising compilers try to specialise and automatically unbox values when possible. Haskell also allows the programmer to explicitly specify in the type that a value should not be boxed, if the code is performance sensitive.

A second option is to have to compiler produce specialised code for every use of a polymorphic function. (This is done in C++.) In general, this leads to code with better performance. However, code size can become a problem, if a complex polymorphic function is applied to many different types.

11.1.2 First-class polymorphism versus rank-1 polymorphism

Consider the following two MinHs types of two functions:

$$\texttt{((}\forall\texttt{(}\alpha\texttt{.(}\alpha \to \alpha\texttt{)))} \to \texttt{Int)}$$

and

$$\texttt{(}\forall\texttt{(}\alpha\texttt{.((}\alpha \to \alpha\texttt{)} \to \texttt{Int)))}$$

Both are types of higher-order functions. However, they differ in a subtle, but important way. The first requires as argument a polymorphic function of type

$$\texttt{(}\forall\texttt{(}\alpha\texttt{.(}\alpha \to \alpha\texttt{)))}$$

and returns an `Int`. The second is itself a polymorphic function, which accepts any function whose input type is the same as the output type, for example, also a function from `Bool` to `Bool`.

A well-typed definition for the first type could be:

```
recfun f :: ((forall a. a -> a) -> Int)  fn = (Inst fn Int) 5
```

So, it takes the polymorphic function, instantiates it to the type `Int`, and then applies it to `5`.

Since the type of the second function is polymorphic, it has to be constructed using a `Type` abstraction:

```
Type a in recfun g  :: ((a -> a) -> Int)  fn = 5
```

In the body of `g`, we don't know what the type `a` has been instantiated to, so we cannot do anything meaningful with it. In particular, we can't apply it to `5`, since `a` might be instantiated to some other type, such as `Bool`, or a function type.

Most languages, like Java, Swift or C# with parametric polymorphism restrict types such that ∀-quantors can only appear at the outermost position (as in the type of the second function). Polymorphic functions and values in these languages not first class citizens. That is, we can define polymorphic functions, but it is not possible to express a type where the argument is required to be polymorphic, or the result. Therefore, explicit ∀-quantors are not required in the type – all the type variables are implicitly ∀-quantified over the whole type. This is also true for (vanilla) Haskell.

If a language places no restrictions on where it allow polymorphic types to appear, and type variables can also be instantiated with polymorphic types (as in our MinHs variant), the language is said to support first-class polymorphism. If quantors are restricted to the outermost position, it is called rank-1 polymorphism.

11.1.3 Implicitly typed polymorphic MinHs

It is somewhat cumbersome to use a language which requires that the programmer explicitly provides types for all variables and functions. Even more so for a language with parametric polymorphism, where we then have to explicitly introduce and instantiate type variables, as we have seen in the previous section. It is much more convenient to instead to have the compiler automatically derive what the most general type of a program and its components are. That is, we just want to write the `swap` function, without type annotations, as

```
recfun swap xy = (snd xy, fst xy)
```

and let the compiler figure out that the most general type of `swap` is `forall a. forall b. (a*b) -> (b * a)`. Let us look at MinHs, to investigate how a compiler can derive this information. We leave the algebraic data types (pairs and sums) in the language, with the exception of recursive types. We also restrict the language to rank-1 polymorphism. We will discuss the problems introduced by recursive types and higher ranked polymorphism later in this section.

Formally, the restriction to rank-1 polymorphism means that we also have to change the rules which define what constitutes a legal type and distinguish between monorphic types (which do not contain ∀-quantifiers, ***MType***), and polymorphic types ***PType***, which may contain quantifiers, but only at the outermost position:

$$\frac{\alpha \in \Delta}{\Delta \vdash \alpha \ \boldsymbol{MType}} \qquad (\textit{IMType-1})$$

$$\frac{}{\Delta \vdash \texttt{Int} \ \boldsymbol{MType}} \qquad (\textit{IMType-2})$$

$$\frac{}{\Delta \vdash \texttt{Bool} \ \boldsymbol{MType}} \qquad (\textit{IMType-3})$$

$$\frac{\Delta \vdash \tau_1 \ \boldsymbol{MType} \quad \Delta \vdash \tau_2 \ \boldsymbol{MType}}{\Delta \vdash (\tau_1 \to \tau_2) \ \boldsymbol{MType}} \qquad (\textit{IMType-4})$$

$$\frac{\Delta \cup \{\alpha\} \vdash \tau \ \boldsymbol{PType}}{\Delta \vdash (\forall (\alpha . \tau)) \ \boldsymbol{PType}} \qquad (\textit{IPType-1})$$

$$\frac{\Delta \vdash \tau \ \boldsymbol{MType}}{\Delta \vdash \tau \ \boldsymbol{PType}} \qquad (\textit{IPType-2})$$

Since we do not have type annotations anymore, the minimum we have to do is to change the typing rules for functions and `Inl` and `Inr` as they relied on these annotations. For our first attempt, we just remove the type annotation from the rule for explicitly typed MinHs

$$\frac{\Gamma \cup \{f : \tau_1 \to \tau_2\} \cup \{x : \tau_1\} \vdash e : \tau_2}{\Gamma \vdash (\texttt{Recfun}\ \tau_1\ \tau_2\ (f.(x.e))) : \tau_1 \to \tau_2} \qquad (\textit{mhs type-7})$$

and get the new, almost identical rule:

$$\frac{\Gamma \cup \{f : \tau_1 \to \tau_2\} \cup \{x : \tau_1\} \vdash e : \tau_2}{\Gamma \vdash (\texttt{Recfun}\ (f.(x.e))) : \tau_1 \to \tau_2}$$

However, now the types τ_1 or τ_2 have to be guessed, because they can no longer be determined by looking at the environment, the expression, or the type derived for the body expression (note that for recursive functions, we need the type τ_2 in the environment to type the body expression). We have a similar situation for `Inl` and `Inr`. If we just drop the type annotations there, the resulting rules do not give us any information on how we can determine the type deterministically, since they allow us to choose any type for τ_2 in the first rule, and τ_1 in the second':

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash (\texttt{Inl}\ e) : (\tau_1 + \tau_2)}$$

$$\frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash (\texttt{Inr}\ e) : (\tau_1 + \tau_2)}$$

And what should we do to replace the explicit introduction of type variables via `Type` and their instantiation via `Inst`? We could do something similar to the ∀-elimination and introduction rules in predicate logic and ∀-quantify any type variables $\boldsymbol{\alpha}$ which does not occur elsewhere in the environment:

$$\frac{\Gamma \vdash e : \tau \qquad \boldsymbol{\alpha} \notin FV(\Gamma)}{\Gamma \vdash e : (\forall(\boldsymbol{\alpha}.\tau))}$$

and instantiate a polymorphic type variable with any type τ' we like:

$$\frac{e : (\forall(\boldsymbol{\alpha}.\tau)) \in \Gamma}{\Gamma \vdash e : \tau[\boldsymbol{\alpha} := \tau']}$$

With these rules, we can indeed derive that the type $(\forall\ (\boldsymbol{\alpha}.(\forall\ (\boldsymbol{\beta}.((\boldsymbol{\alpha} * \boldsymbol{\beta}) \rightarrow (\boldsymbol{\beta} * \boldsymbol{\alpha}))))))$ is a possible type for `swap`. But we can, with the same rules, also derive many other types for `swap`, such as every monomorphic instance of it.

The problem is that the rules we introduced above do not describe an algorithm to deterministically derive a type for a given expression. For example, for the function rule, we have to make the correct guesses for τ_1 and τ_2. Moreover, the ∀-elimination rule can always be applied on an expression of polymorphic type, so we have to know beforehand when to apply it (if at all).

What we need is a set of syntax-directed rules, where we can interpret the environment and the expression as input, and the type of the expression as unique output (modulo choice of the bound type variable names).

Hindley-Milner Type Inference

To eliminate the need to guess the correct choice of a type for an expression at a point when we don't have suffient information about the context the expression appears in, we instead instantiate the type with fresh type variables[3]. We then instantiate these free type variables when we learn more about how the expression is used. Going back to the `swap`-example

```
    recfun swap xy = (snd xy, fst xy)
```

We initially assign a type α to `xy`, where α is a type variable that is not used anywhere else. After traversing the body of the function, we see that the `fst` and `snd` functions are applied to `xy`, so type α has to have the form $(\beta * \gamma)$, for some unknown types β and γ, and that the type of the body expression is then $(\gamma * \beta)$. By inferring the type of an expression, we therefore also obtain more information about the type variables (in this case of α). As part of the output, our type inference algorithm has to return this information, because . We can represent it as substitution which instantiates free type variables with a more specific type. In our example, the substitution would be $[\alpha := (\beta * \gamma])$.

So, our type inference algorithm should take the current environment as well as the expression we want to type as input, and return as output the type of the expression, and a substitution

[3]A type variable is fresh if it does not already occur in the environment

which may provide more information about some of the free type variables in the environment. For example, for typing the subexpression `snd xy`, the environment would be $\{texy : \alpha\}$, the return type β, and the substitution $(\beta * \gamma)$.

How can we determine the appropriate substitution? Essentially, we have to unify the current type τ_1 of a MinHS variable or expression with the required type τ_2. That is, we look for a substitution U such that U applied to τ_1 is equal to U applied to τ_2:

$$U\tau_1 = U\tau_2$$

For example, if $\tau_1 = (\alpha*\beta)$ and $\tau_2 = (\gamma*\texttt{Int})$, then $[\alpha := \gamma, \beta := \texttt{Int}]$ would be a unifier, as would $= [\alpha := \texttt{Int}, \beta := \texttt{Int}, \gamma := \texttt{Int}]$. Since we do not want to restrict a type unnecessarily, we look for the most general unifier or MGU — that is, a unifier that does not instantiate a type variable to a more specific type than necessary. In the previous example, thie first substitution is an MGU, but the second one is more specific than necessary.

We write the following:

$$\tau_1 \overset{U}{\sim} \tau_2$$

to denote that U is the MGU of τ_1 and τ_2. The MGU of two types is not, in general, unique, but if there are two MGUs, they can only differ in the names of their free variables.

Some examples:

- $(\alpha*\beta)$ and $(\beta*\beta)$ have the MGU $[\alpha := \beta]$ (as well as $[\beta := \alpha]$). Another unifier is $[\alpha := \texttt{Int}, \beta = \texttt{Int}]$, but it's not an MGU.
- $(\alpha*\beta)$ and $(\alpha+\beta)$ have no unifier, since the top-level constructor differs. There is no way to instantiate the variables to make these two type terms to be the same.
- $(\alpha*\beta)$ and α have no unifier, as there is no (finite) substitution which, applied to both terms, makes them equal.
- $(\alpha*\texttt{Int})$ and $(\texttt{Bool}*\alpha)$ have no unifier, since the first component of the pair implies that α should be repleaced with `Bool`, the second with `Int`.

We can calculate the MGU of two types as follows:

- Input: two types τ_1 and τ_2, where $\forall$-quantified variables have been replaced by fresh, unique variables
- Output: the most general unifier of τ_1 and τ_2 (if it exists).

Unification: Given two type terms τ_1 and τ_2, if

1. both are the same type variable then return the empty substitution
2. they are different variables α and β, return $[\alpha := \beta]$
3. both are primitive types
   - if they are the same, return the empty substitution
   - otherwise, there is no unifier
4. both are product types, with $\tau_1 = (\tau_{11}*\tau_{12})$, $\tau_2 = (\tau_{21}*\tau_{22})$
   - compute the unifier S of τ_{11} and τ_{21}
   - compute the unifier S' of $S\,\tau_{12}$ and $S\,\tau_{22}$
   - return $S \cup S'$
5. both are function types or both are sum types proceed as for as for product types

6. only one is a type variable α, the other an arbitrary type term τ
    - if α occurs in τ, there is no unifier
    - otherwise, return $[\alpha := \tau]$
7. otherwise, there is no unifier

Type Inference Rules

Finally, we have everything in place to write down the algorithmic type inference rules. Our judgement has the form:

$$S\Gamma \vdash e : \tau$$

where Γ is the (input) environment, e the (input) expression we are typing, and S the (output) substitution applied to the environment necessary to derive the type (output)τ. The output in every judgement in this section is *highlighted*.

Constants When typing constant values, such as integers, boolean and so on, we just return the type and an empty substitution ($[]$). The rule for integer constants, for example, is

$$\frac{}{[]\Gamma \vdash n : \texttt{Int}} \quad (HM\text{-}1)$$

Variables Variables are more interesting. If it has a polymorphic type, we drop the quantifiers from the type and replace the quantified variables with fresh ones:

$$\frac{x : (\forall(\alpha_1 . \ldots (\forall(\alpha_n . \tau)))) \in \Gamma}{[]\Gamma \vdash x : [\alpha_1 := \beta_1 \ldots \alpha_n := \beta_n]\tau} \; \beta_i \textit{ fresh} \quad (HM\text{-}2)$$

Again, the substitution is empty.

Pairs

$$\frac{T\Gamma \vdash e_1 : \tau_1 \quad T'T\Gamma \vdash e_2 : \tau_2}{TT'\Gamma \vdash (\texttt{Pair}\; e_1\; e_2) : (T'\tau_1 * \tau_2)} \quad (HM\text{-}3)$$

Algorithmic interpretation of the pair-rule:

1. Input: Γ and (Pair e_1 e_2)
2. Infer type of e_1 under Γ, obtain substitution T and type τ_1 as result.
3. Infer type of e_2 under $T\Gamma$ (that is, substitution T applied to Γ), obtain substitution T' and type τ_2 as result.
4. Return as result the combined substitutions T and T', and the type $(T'\tau_1 * \tau_2)$. (Note that we still have to apply the substitution T' to τ_1.)

$$\frac{T\Gamma \vdash e : \tau \quad \tau \overset{U}{\sim} (\alpha * \beta)}{TU\Gamma \vdash (\texttt{Fst}\; e) : U\alpha} \quad (HM\text{-}4)$$

$$\frac{T\Gamma \vdash e : \tau \quad \tau \overset{U}{\sim} (\alpha * \beta)}{TU\Gamma \vdash (\texttt{Snd}\; e) : U\beta} \quad (HM\text{-}5)$$

Sum types

$$\frac{T\Gamma \vdash e : \tau}{T\Gamma \vdash (\texttt{Inl}\ e) : (\tau + \alpha)}\ \alpha\ \textit{fresh} \tag{HM-6}$$

$$\frac{T\Gamma \vdash e : \tau}{T\Gamma \vdash (\texttt{Inl}\ e) : (\alpha + \tau)}\ \alpha\ \textit{fresh} \tag{HMI-7}$$

$$\frac{T\Gamma \vdash e : \tau \quad \begin{array}{c} T_1T(\Gamma \cup \{x : \alpha_l\}) \vdash e_1 : \tau_1 \\ T_2T_1T(\Gamma \cup \{y : \alpha_r\}) \vdash e_2 : \tau_2 \end{array} \quad \begin{array}{c} T_2T_1T(\alpha_l + \alpha_r) \overset{U}{\sim} T_2T_1\tau \\ UT_2\tau_l \overset{U'}{\sim} U\tau_r \end{array}}{U'UT_2T_1T\Gamma \vdash (\texttt{Case}\ e\ (x.e_1)\ (y.e_2)) : U'U\tau_r}\ \alpha_l,\ \alpha_r\ \textit{fresh} \tag{HMI-8}$$

Note that we have to apply substitutions like T_2 only to types that have been inferred before those substitutions were produced; for example, in the `Case` rule, τ_l was inferred before T_2 was produced, so in the rightmost unification (producing U') of that rule, τ_l needs T_2 (and U) applied to it. However, τ_l was produced together with T_1 and after T (indeed, T was applied to the environment that was given as input to the inference of e_1, producing τ_l), so in a sense, T and T_1 have already been applied to something to produce τ_l. Applying them again would be useless work.

Application

$$\frac{T\Gamma \vdash e_1 : \tau_1 \quad T'T\Gamma \vdash e_2 : \tau_2 \quad T'\tau_1 \overset{U}{\sim} (\tau_2 \to \alpha)}{UT'T\Gamma \vdash (\texttt{Apply}\ e_1\ e_2) : U\alpha}\quad \alpha\ \textit{fresh} \tag{HMI-9}$$

Instead of giving rules for every built-in operator, such as `Plus`, `Times` and so on, we can just put the type of these operators in the initial environment, and rewrite them to regular applications. For `Plus`, we add `Plus` :: `Int` → `Int` → `Int` in the environment, and replace terms of the form

$$(\texttt{Plus}\ e_1\ e_2)$$

with

$$(\texttt{App}\ (\texttt{App}\ \texttt{Plus}\ e_1)\ e_2)$$

in the program. We can do the same with `Inl`, `Inr`, `Pair`, `Fst` and `Snd`.

If-Then-Else

$$\frac{T\Gamma \vdash e : \tau \quad \tau \overset{U}{\sim} \texttt{Bool} \quad T_1UT\Gamma \vdash e_1 : \tau_1 \quad T_2T_1UT\Gamma \vdash e_2 : \tau_2 \quad T_2\tau_1 \overset{U'}{\sim} \tau_2}{U'T_2T_1UT\Gamma \vdash (\texttt{If}\ e\ e_1\ e_2) : U'\tau_2} \tag{HMI-10}$$

Recursive Functions

$$\frac{T(\Gamma \cup \{x : \alpha_1\} \cup \{f : \alpha_2\}) \vdash e : \tau \quad T\alpha_2 \overset{U}{\sim} (T\alpha_1 \to \tau)}{UT\Gamma \vdash (\texttt{Recfun}\ (f.(x.e))) : U(T\alpha_1 \to \tau)}\ \alpha_1, \alpha_2\ \textit{fresh} \tag{HMI-11}$$

Let Bindings & Top-Level Program

$$\frac{T\Gamma \vdash e_1 : \tau \qquad T'(T\Gamma \cup \{x : \textit{Generalise}(T\Gamma, \tau)\}) \vdash e_2 : \tau'}{T'T\Gamma \vdash (\texttt{Let}\ e_1\ (x.e_2)) : \tau'} \tag{HMI-12}$$

$$\frac{T\Gamma \vdash e : \tau}{T\Gamma \vdash (\texttt{Main}\ e) : \textit{Generalise}(T\Gamma, \tau)} \tag{HMI-13}$$

where:[4]

$$Generalise(\Gamma, \tau) = \forall (FV(\tau) \setminus FV(\Gamma)).\ \tau$$

11.1.4 Principal Types

The algorithm specified by the rules above is a formalisation of the Hindley-Milner type inference algorithm. For a given parametric polymorphic program, it infers the most general type or principal type. More precisely, we say a type $\forall \alpha_i . \tau_1$ is less general than a type $\forall \beta_j . \tau_2$ if there is a substitution S, such that $S\tau_2 = \tau_1$, and S is not just a renaming of variables, that is, $\forall \alpha_i . \tau_1$ and $\forall \beta_j . \tau_2$ are not α-equivalent.

In MinHs, any type correct expression has such a principal type which is more general or α-equivalent to any derivable type for that expression. However, if we add recursive types back into the language, this is no longer true. The expression

```
Roll (Inl ())
```

can be of type Rec t. (Unit + Int) or Rec t. (Unit + t), for example, but these two types have no common, more general type of which they are both instances of (note that substitution would not affect t, as this is a bound type variable). This means that the expression has no principal type.

How is it then possible that we have recursive types in an implicitly typed polymorphic language such as Haskell? While we have a pair constructor and a sum constructor (`Either`) in Haskell, which behave exactly like MinHs' pair and sum type constructor, there is nothing which corresponds to `Rec`. In fact, if we want recursive types in Haskell, we do it via a `data` declaration, with which we implicitly provide the types of each of its constructors

```
  data Tree a
    = Leaf                           -- Leaf :: (Tree a)
    | Node a (Tree a) (Tree a)       -- Node :: a -> Tree a -> Tree a -> Tree a
```

The prenex restriction is in our language for a similar reason, and it limits the functions we can express in the language. For example, the following function cannot be typed in our system

```
   foo f = (f False, f 1))
```

since `f` is applied to `False`, a boolean value, and `1`, an integer value. A possible type for `foo` would be $(\forall \mathtt{a}.\mathtt{a} \to \mathtt{a}) \to (\mathtt{Bool}, \mathtt{Int})$, which requires `f` to be polymorphic. This type is however not legal in our system, because polymorphic functions are not first class citizens, and the $\forall$-quantifier can only occur at the outmost position. If we move the quantifier outside, however, the resulting type $\forall \mathtt{a}.((\mathtt{a} \to \mathtt{a}) \to (\mathtt{Bool}, \mathtt{Int}))$ is very different and not a possible type for `foo`, as it accepts any function from any type `a` to `a` (for example, `not` $::$ `Bool` $\to$ `Bool`), but `not` cannot be applied to an integer value.

11.2 Overloading

So far, all the operations we have in MinHs are either monomorphic and work only on a specific type, as for example addition

```
(+) :: Int -> Int -> Int
```

or they are (parametric) polymorphic, like the swap function. In practice, this is not sufficient for a general purpose language. If we add, for example, floating point numbers to MinHs, we want at least all the operations we have on integers to be available on floats as well, so we need antother operation

$(+_{Float})$ `:: Float -> Float -> Float`

[4]Note that $S \setminus S'$ means the set S without all elements in S'.

to add two floats, and the same for multiplication and so on.

However, having a differently named function for each type would make the language pretty annoying to use, in particular for a language with a more realistic set of types which includes integers and floats of different size. The situation is even worse when we consider an operation like checking two values for equality. This should, ideally, not only work on basic types, but on also on compound types, like pairs, sum and recursive types.

A common solution to this problem is to overload the name of a function, method or operation to work on different types. Either at compile time or at run time, overloading has to be resolved: that is, the overloaded operation has to be replaced with the operation which works on the type of values it is applied to.

We will be looking at two different approaches to overloading.

11.2.1 Compile-time Resolution of Overloading

In Java, as well as in C# and C++, methods can be overloaded. In contrast to the latter two languages, Java does, however, not allow for user-defined operator overloading. Overloaded methods can differ also differ in the number and order of the arguments. Like in most object oriented languages, the compiler will resolve the overloading, by looking at the types of the arguments and choosing the correct implementation.

Resolving overloading it straight forward if there is a precise match of the types of the formal parameters of an overloaded method or operator and the types of the actual parameters at a call site. However, in the presence of subtyping or subclassing, the compiler may look for the closest match. The strategy to find this differs between languages, which can lead to unexpected behaviour.

For example, consider the following two programs, one in C#, the other in Java. Both overload a method to work on the object class and `int` arrays. In the main method, the print method is called with an `int` argument.

```
// C#
public class Program {
    public static void Main() {
        PrintSomething(42);
    }

    public static void PrintSomething(object obj) {
        Console.WriteLine("I'm an object");
    }

    public static void PrintSomething(params int[] arr) {
        Console.WriteLine("I'm an array");
    }
}
```

Running the C# program will print `I'm an array`, whereas Java resolves the overloading differently and, for essentially the same code, will print `I'm an object`.

```
/* Java */
public class Main {
    public static void main(String[] args) {
      printSomething (42);
    }

  public static void printSomething(Object obj) {
    System.out.println("I'm an object");
  }
  public static void printSomething (int[] arr) {
    System.out.println("I'm an array");
  }
}
```

Therefore, although overloading is very convenient, it is important to be aware of subtle differences between languages, and use it carefully.

11.2.2 Type Classes in Haskell

Haskell also supports overloading, but chooses a very different approach, namely via type classes. The idea behind type classes is to group a set of types together if they have several, conceptually similar operations in common. For example, in Haskell, the type class on which the operations addition, multiplication, subtraction and such are defined is called `Num`, and it contains the types `Int`, `Integer` (not fixed length), `Float` and `Double`.

The type for addition is

```
(+) :: forall a. Num a  => a -> a -> a
```

which can be read as: for all types `a` in type class `Num`, `(+)` has the type `a -> a -> a` . The type constraint `Num` restricts the types which addition can be applied to.

The `Eq` type class in Haskell contains all types whose elements can be tested for pairwise equality. When you type `:info Eq` in ghci, the interpreter lists all its methods

```
(==) :: forall a. Eq a => a -> a -> Bool
(/=) :: forall a. Eq a => a -> a -> Bool
```

as well as the types which are in this type class. Apart from all the basic types

```
instance Eq Int
instance Eq Float
instance Eq Double
instance Eq Char
instance Eq Bool
```

it also lists rules of the form

```
instance Eq a ⇒ Eq [a]
instance Eq a ⇒ Eq (Maybe a)
instance (Eq a, Eq b) ⇒ Eq (a, b)
```

These are rules about type class membership: if a class `a` is in `Eq`, then lists of `a` are also in `Eq`, as well as `Maybe a`. If two types `a` and `b` are both in `Eq`, so are pairs of `a` and `b`. Operations on these compound types are implemented in terms of the operations of the argument types. So, two pairs of values are considered to be equal if both of their components are equal:

```
instance (Eq a, Eq b) ⇒ Eq (a, b) where
  (==) (a1, b1) (a2, b2) = (a1 == a2) && (b1 == b2)
  ...
```

And equality of lists can be defined as follows (sin

```
instance Eq a ⇒ Eq [a] where
  (==) [] []          = True
  (==) (a:as) (b:bs) = (a  == b) && (as == bs)
  (==) _      _       = False
```

Other examples of predefined type classes are `Show` and `Read`. A user can extend these type classes and define new classes.

11.2.3 Type Classes in MinHs

To investigate how introducing type classes and overloading affect the type system, and how they can be implemented, we follow our strategy of adding a minimal version of the feature to MinHs to serve as a case study.

Coming back to our initial motivating example, we add the type *Float* to the language, as well as two type classes *Eq*. and *Num*. Polymorphic types can now contain type constraints:

```
(+)  :: ∀ a. Num a ⇒ a → a → a
(*)  :: ∀ a. Num a ⇒ a → a → a
(==) :: ∀ a. Num a ⇒ a → a → Bool
```

Our new set of inference rules for *OkP* reflects this:

$$\frac{t \in \Delta}{\Delta \vdash t\ \textit{Ok}} \quad \frac{}{\Delta \vdash \texttt{Int}\ \textit{Ok}} \quad \frac{}{\Delta \vdash \texttt{Bool}\ \textit{Ok}} \quad \frac{}{\Delta \vdash \texttt{Float}\ \textit{Ok}} \quad \frac{\Delta \vdash \alpha\ \textit{Ok} \quad \Delta \vdash \beta\ \textit{Ok}}{\Delta \vdash \alpha \rightarrow \beta\ \textit{Ok}}$$

$$\frac{\Delta \vdash \sigma\ \textit{OkC} \quad \Delta \vdash \tau\ \textit{Ok}}{\Delta \vdash C\ \tau \Rightarrow \sigma\ \textit{OkC}}, \textit{where}\ C \in \{Num, Eq\} \qquad \frac{\Delta \vdash \tau\ \textit{Ok}}{\Delta \vdash \tau\ \textit{OkC}}$$

$$\frac{\Delta \cup \{\texttt{a}\} \vdash \tau\ \textit{OkP}}{\Delta \vdash \forall\ \texttt{a}.\ \tau\ \textit{OkP}} \quad \frac{\Delta \vdash \tau\ \textit{OkC}}{\Delta \vdash \tau\ \textit{OkP}}$$

Predicate Entailment

To infer the type of an expression or program, it is now not sufficient anymore to know the type of every MinHs builtin constant and function. In addition, we need to know which types are in which type classes. This information is encoded as a set of predicates over types. For our MinHs example, we have the following predicates for *Num*:

Num Int
Num Float

For *Eq*, the predicates are they are a bit more interesting, as there are an infinite number of types in *Eq*:

Eq Int
Eq Float
$\forall\, a. \forall\, b. Eq\ a\ \Rightarrow\ Eq\ b\ \Rightarrow\ Eq\,(a\ *\ b)$
$\forall\, a. \forall\, b. Eq\ a\ \Rightarrow\ Eq\ b\ \Rightarrow\ Eq\,(a\ +\ b)$

As mentioned when we discussed Haskell's type classes, the latter two rules can be read as:

> For all types a and b, if a is in Eq and b is in Eq, then $(a\ *\ b)$ $((a\ +\ b)$ respectively, are in Eq as well.

Note that the third and fourth rules look slightly different from the one corresponding one given in Haskell $((Eq\ a,\ Eq\ b)\ \Rightarrow\ Eq\,(a,\ b))$, where the class constraints on a and b were listed as pair. Logically, however, they are equivalent, just as $A \wedge B \Rightarrow C$ and $A \Rightarrow (B \Rightarrow C)$ are equivalent.

We call the set of all predicates for MinHs we just listed P_{MinHS}.

Let us now formalise how, given a set of type class predicates P, we can derive type class membership (also called type constraints) for concrete types. We say a predicate set P entails a constraint c, or $P \Vdash c$ if

1. $c \in P$, or
2. $P \Vdash \forall a.c'$ and $c = c'[a := t]$ for any type t, or
3. $P \Vdash \pi \Rightarrow c$ and $P \Vdash \pi$ (π might be a constraint c' or another implication $\pi' \Rightarrow c'$).

The first rule is simple: if a constraint c is in P, then P entails c, and it is the only rule relevant for the Num type class in MinHs. The second rule just says we can instantiate polymorphic contraints with any type and Rule 3 is what would correspond to modus ponens in logic. Rules 1-3 entail $Eq(Int * Bool)$, for example, in MinHs, since

1. $P_{MinHs} \Vdash \forall a.\forall b.Eq\ a \Rightarrow Eq\ b \Rightarrow Eq\ (a * b)$
2. $P_{MinHs} \Vdash \forall b.Eq\ Int \Rightarrow Eq\ b \Rightarrow Eq(Int * b)$ (1. and Rule 2)
3. $P_{MinHs} \Vdash Eq\ Int \Rightarrow Eq\ Bool \Rightarrow Eq(Int * Bool)$ (2. and Rule 2)
4. $P_{MinHs} \Vdash Eq\ Int$ (Rule 1)
5. $P_{MinHs} \Vdash Eq\ Bool \Rightarrow Eq\ (Int * Bool)$ (3., 4., and Rule 3)
6. $P_{MinHs} \Vdash Eq\ Bool$ (Rule 1)
7. $P_{MinHs} \Vdash Eq\ (Int * Bool)$ (5., 6., and Rule 3)

The typing rules have to be adjusted to include the set of constraints P. Most rules are not affected by this change and simply pass P along:

$$\frac{x : \tau \in \Gamma}{P \mid \Gamma \vdash x : \tau} \qquad \frac{}{P \mid \Gamma \vdash \texttt{(Int}\ n\texttt{)} : \texttt{Int}} \qquad \frac{}{P \mid \Gamma \vdash \texttt{(Float}\ f\texttt{)} : \texttt{Float}}$$

$$\frac{}{P \mid \Gamma \vdash \texttt{(Const}\ b\texttt{)} : \texttt{Bool}} \quad b \in \{\texttt{True}, \texttt{False}\}$$

$$\frac{P \mid \Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{P \mid \Gamma \vdash \texttt{(Apply}\ e_1\ e_2\texttt{)} : \tau_2} \qquad \frac{P \mid \Gamma \cup \{f : \tau_1 \to \tau_2\} \cup \{x : \tau_1\} \vdash e : \tau_2}{P \mid \Gamma \vdash \texttt{(Letfun}\ f.x.e\texttt{)} : \tau_1 \to \tau_2}$$

$$\frac{P \mid \Gamma \vdash e : \forall t.\tau}{P \mid \Gamma \vdash e : \tau[t := \tau']} \qquad \frac{P \mid \Gamma \vdash e : t \quad t \notin TypeVars(\Gamma)}{P \mid \Gamma \vdash e : \forall t.\tau}$$

Only the following two new rules actually use P:

$$\frac{P \mid \Gamma \vdash e : \pi \Rightarrow \rho \ \ P \Vdash \pi}{P \mid \Gamma \vdash e : \rho} (\Rightarrow -elimination)$$

$$\frac{P, \pi \mid \Gamma \vdash e : \rho}{P \mid \Gamma \vdash e : \pi \Rightarrow \rho} (\Rightarrow -introduction)$$

The elimination allows us to instantiate an overloaded function with a specific type, as long as P entails the corresponding constraints. For example, we can derive $P_{MinHS} \mid \mathit{\Gamma}_{MinHS} \vdash (+) : Int \to Int \to Int$ using the elimination-rule since

- $P_{MinHS} \mid \mathit{\Gamma}_{MinHS} \vdash (+) : Num\ Int \Rightarrow Int \to Int$ (which we can prove using the $\forall$-elimination rule) and

- $P_{MinHs} \Vdash Num\ Int$, which holds (entailment rule 1)

The introduction rule says we can move a constraint from the predicate set to the type. That is, if we can show a type ρ is derivable with a predicate set P plus a constraint π, then the type $\pi \Rightarrow \rho$ is derivable from P alone. For example, if we want to show that

- $P_{MinHS} \mid \Gamma_{MinHS} \vdash letfun\, f\ x\ =\ x\ +\ 1\ :\ \forall\, a.\, Num\ a \Rightarrow\ a\ \rightarrow\ a$

is derivable, we first apply the $\forall$-introduction rule, which leaves us with the proof obligation

- $P_{MinHS} \mid \Gamma_{MinHS} \vdash letfun\, f\ x\ =\ x\ +\ 1\ :\ Num\ a \Rightarrow\ a\ \rightarrow\ a$

Now, we apply the $\Rightarrow$-introduction rule, resulting in the proof obligation

- $P_{MinHS},\ Num\ a \mid \Gamma_{MinHS} \vdash letfun\, f\ x\ =\ x\ +\ 1\ : \Rightarrow\ a\ \rightarrow\ a$

and then the letfun rule yielding

- $P_{MinHS},\ Num\ a \mid \Gamma_{MinHS} \cup \{f\ :\ a\ \rightarrow\ a,\ x\ :\ a\} \vdash x\ +\ 1\ :\ a$

and so on. To derive $(+)\ :\ a\ \rightarrow\ a\ \rightarrow\ a$ we need the elimination rule one more time, as well as the $Num\ a$ constraint in the new predicate set.

When we introduced the first set typing rules for polymorphic, implicitely typed MinHs, we observed that they do not describe a deterministic algorithm to compute the type of a given expression. They can only be used to prove that a given expression indeed has a certain type, and to do the proof successfully, we have to guess which rule to apply or how to instantiate some of the type variables.

Overloading Resolved

Up to this point, we only discussed the effect overloading has on the static semantics. But how about the dynamic semantics? At some point, the overloaded function has to be instantiated to the correct concrete operation. This could happen at run time, but that doesn't seem to be the best way.

In object oriented language, objects typically know what to do - that is, an object is associated with a so-call virtual method or dispatch table, which contains all the methods of the object's class. This approach isn't applicable to a language like MinHs, where arguments of an overloaded function aren't objects and can be of basic type, but we can use a similar idea: we pass the table of all the methods of a class to the overloaded function, and replace the overloaded function with one that simply picks the appropriate method from the table. We call this table a dictionary, and in MinHs, we represent a table with n functions as n-tuple. For simplicity, we assume for now that MinHs supports n-tuples and projection functions $proj^k_n$ to extract the kth element from an n-tuple. So, $proj^2_1$ is the same as the function fst on pairs, $proj^2_2$ is the same as snd.

This way, we can transform every MinHs expression containing overloaded functions to a regular MinHs expression.

The dictionary for the Eq class, for example, is just a pair of the function $==$ and $/=$ of the type in question. The overloaded equality function picks the first item in the table, the $/=$ function the second.

So, the overloaded MinHs expression

$$(==)\ 1.7\ \ 2.75$$

can be transformed into regular MinHs:

$$\begin{aligned}&(proj^2_1\ ((==)_{Float},\ (/=)_{Float}))\ 1.7\ \ 2.75\\ =&--\ evaluates\ to\\ &(==)_{Float}\end{aligned}$$

11.3 Subtyping

With type classes, as implemented in Haskell, for example, the programmer can use the same overloaded function symbol both for addition of floating point values and integer values, and the compiler will figure out which to use. However, the following expression would nevertheless be rejected by the Haskell compiler:

```
let x = 1    :: Int
    y = 1.75 :: Float
in x + y
```

as addition can be applied to two `Int` or two `Float`, but not a combination of both. In Haskell, we explicitly have to convert the `Int` value `x` to `Float` to add the two values

```
let
    x = 1    :: Int
    y = 1.75 :: Float
 in (fromIntegral x) + y
```

or round `y` to an `Int` value

```
let
    x = 1    :: Int
    y = 1.75 :: Float
in x + (round y)
```

C solves this problem using something called integer promotion: the basic types are ordered and if operations like `+` or `==` are applied to mixed operands, the one which is the lowest in the hierarchy is automatically cast to the higher type. This is quite convenient, but can easily lead to unexpected behaviour and subtle bugs, in particular with respect to signed/unsigned types.

The idea behind subtyping is similar to the approach in C in that types can be partially ordered in a subtype relationship

$$\tau \leq \sigma$$

such that, whenever a value of some type σ is required, it is also fine to provide a value of type τ, as long as τ is a subtype of σ. For example, we can define the following subtype relationship:

$$\texttt{Int} \leq \texttt{Float} \leq \texttt{Double}$$

With subtyping, it is then be ok to have

$$1 +_{\texttt{Float}} 1.75$$

as floating point addition requires two floating point values, but also accepts `Int`, as they are a subtype of `Float`.

11.3.1 Subset and Coercion Interpretation

There are different ways to interpret the subtype relationship: one is to define τ to be a subtype of σ if it is an actual subset. For example, in the mathematical sense, integral numbers are a subset of rational numbers, even integral numbers of integral numbers and so on. However, this interpretation is quite restrictive for a programming language: `Int` is not actually a subset of `Float`, as they have very different internal representations. However, there is an obvious coercion from `Int` to `Float`. For our study of subtyping, we will focus on this so-called coercion interpretation of subtyping: τ is a subtype of σ, if there is a sound [5] coercion from values of type τ to values of type σ.

[5] More about that later

As another example, consider a graph and tree type in Haskell parametrized over the type of information a stored in each node. Since trees are a special case of graphs, trees can be converted into a graph and we can view the tree type as subtype of the graph type in the coercion interpretation.

```
type NodeId = Int
type Edge   = (NodeId, NodeId)

data Graph a = Graph Int [(NodeId, a)] [Edge]  -- number of nodes, properties
                                               -- of nodes,  list of edges
data Tree a = Leaf | Node a (Tree a) (Tree a)

treeToGraph :: Tree a -> (Graph a, NodeId)
treeToGraph tree = ...
```

11.3.2 Properties

For a subtyping relationship to be soundnesssound, it has to be reflexive and transitive. This is the case for both the subset as well as the coercion interpretation. For the subset interpretation, all three properties follow directly from the properties of the subset relation. In the coercion interpretation, reflexivity holds because the identity function is a coercion from $\tau \rightarrow \tau$. Transitivity holds since, given a coercion function from $f : \tau_1 \rightarrow \tau_2$ and $g : \tau_2 \rightarrow \tau_3$, the composition of f and g result in a coercion function from $\tau_1 \rightarrow \tau_3$.

$$\frac{}{\tau \leq \tau}(reflexivity)$$

$$\frac{\tau_1 \leq \tau_2 \qquad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3}(transitivity)$$

11.3.3 Coherency of Coercion

The coercion of values should also be coherent. This means that, if there are two ways to coerce a value to a value of a supertype, both coercions have to yield the same result.

For example, let us assume we define `Int` to be a subtype of `Float`, and both to be subtypes of `String`, with coercion functions

```
intToFloat :: Int -> Float
intToFloat = fromIntegral

intToString :: Int -> String
intToString = show

floatToString :: Float -> String
floatToString = show
```

On first sight, this looks like a reasonable definition. It is not coherent, however, because there are two coercion function from `Int` to `String`: the provided function `intToString`, but also `intToFloat` composed with `floatToString`. Unfortunately, applied to the number `3`, for example, one would result in the string `"3"`, the other in `"3.0"`

One reason why type promotion in C can be so tricky is exactly that it is not coherent. Consider, for example the following code snippet:

```
  short int i = -1;
  short unsigned int j = 0;
  float x = 1.0;
  short unsigned int k = i + j;
  printf \mathtt{("test1: %f ", x + k)};
  printf ("test2: %f ", x + i + j);
```

Since k is the sum of i and j, one might expect both lines to print the same result. However, this is not the case:

```
> ./coerce
test1: 65536.000000
test2: 0.000000
```

as in the first case i is first cast to unsigned short, then to a float, and in the second case it is cast to float directly.

11.3.4 Variance

If we add subtyping to MinHs, one question that arises is how the subtyping relationship interacts with our type constructors. For example, if `Int` $\leq$ `Float`, what about pairs, sums and function over these types? How do they relate to each other?

For pairs and sums, the answer is quite straight forward. Obviously, given a coercion function `intToFloat`, we can easily define coercion functions on pairs and sums:

```
p1 :: (Int * Int) -> (Float * Float)

p2 :: (Int * Float) -> (Float * Float)
p2 (x, y) = (intToFloat x,  y)
...


s1 :: (Int + Int) -> (Float + Float)
s1 x = case x of
         InL v -> intToFloat v
         InR v -> intToFloat v
...
```

This means that, if two types τ_1 and τ_2 are subtypes of σ_1 and σ_2, respectively, then pairs of τ_1 and τ_2 are also subtypes of pairs of σ_1 and σ_2 and the same is true for sums. More formally, we have:

$$\frac{\tau_1 \leq \sigma_1 \qquad \tau_2 \leq \sigma_2}{(\tau_1 * \tau_2) \leq (\sigma_1 * \sigma_2)}$$

$$\frac{\tau_1 \leq \sigma_1 \qquad \tau_2 \leq \sigma_2}{(\tau_1 + \tau_2) \leq (\sigma_1 + \sigma_2)}$$

The following diagrams also shows the subtype relationship between pair types and sum types, respectively:

$$\begin{tikzcd}
(\texttt{Int} * \texttt{Int}) \arrow[r, "\leq"] \arrow[d, "\leq"] & (\texttt{Float} * \texttt{Int}) \arrow[d, "\leq"] \\
(\texttt{Int} * \texttt{Float}) \arrow[r, "\leq"] & (\texttt{Float} * \texttt{Float})
\end{tikzcd}$$

$$\begin{tikzcd}
(\texttt{Int} + \texttt{Int}) \arrow[r, "\leq"] \arrow[d, "\leq"] & (\texttt{Float} + \texttt{Int}) \arrow[d, "\leq"] \\
(\texttt{Int} + \texttt{Float}) \arrow[r, "\leq"] & (\texttt{Float} + \texttt{Float})
\end{tikzcd}$$

Given that the pair as well as the sum type constructor interacts with subtyping in such an obvious way, it is easy to be tricked into thinking this applied to all type constructors. Unfortunately, this is not the case.

Consider function types: is `Int -> Int` as subtype of `Float -> Int`? That is, if a function of type `Float -> Int` is required, would it be ok to provide a function of type `Int -> Int` instead? Considering that the type `Int` is more restricted than the type `Float`, this means that a function which only works on the "smaller" type `Int` is also, in some sense, less powerful. Or, coming back to our second example, if we need a function to process any graph, then a function which only works on trees (and maybe relies on the fact that there are no cycles in a tree) is clearly not sufficient. We are also not able to define a coercion function

```
c :: (Int -> Float) -> (Float -> Float)
```

in terms of our coercion function `intToFloat`. The other direction, however, is actually quite easy:

```
c' :: (Float -> Float) -> (Int -> Float)
c' f = let g x = f (intToFloat x)
       in g
```

Or equivalently, as types

```
(Float -> Float) -> (Int -> Float)
```

and

```
(Float -> Float) -> Int -> Float
```

are exactly the same:

```
c' :: (Float -> Float) -> (Int -> Float)
c' f x = f (intToFloat x)
```

Therefore, somewhat surprisingly, we have `(Float -> Float)` $\leq$ `(Int -> Float)`.

So, what about the result type of a function: is `Int -> Int` a subtype of `Int -> Float` , vice versa, or are these types not in a subtype relationship at all? If we need a function which returns a `Float` and get one that returns an `Int`, it is not a problem, since we can easily convert that `Int` to a `Float`. Similarly, if we need a function which returns a graph, and we get a tree, it is ok as a tree is a special case of a graph and can be converted to the graph representation.

```
c'' :: (Int -> Int) -> (Int -> Float)
c'' f = let g x = intToFloat (f x)
        in g
```

To summarise, the subtype relationship on functions over `Int` and `Float` is as follows (of course, you can substitute any type τ for `Int`, σ for `Float` here, as long as $\tau \leq \sigma$):

$$\begin{tikzcd}
(\texttt{Float -> Int}) \arrow[r, "\leq"] \arrow[d, "\leq"] & (\texttt{Int -> Int}) \arrow[d, "\leq"] \\
(\texttt{Float -> Float}) \arrow[r, "\leq"] & (\texttt{Int -> Float})
\end{tikzcd}$$

The subtype propagation rule for function types expresses exactly the same relationship:

$$\frac{\tau_1 \leq \sigma_1 \qquad \tau_2 \leq \sigma_2}{(\sigma_1 \to \tau_2) \leq (\tau_1 \to \sigma_2)}$$

Another example of a type which interacts with subtyping in a non-obvious manner are updateable arrays and reference types. To understand what is happening, let us have a look at Haskell-style updatable references. We have the following basic operations on this type:

```
newIORef   :: a -> IO (IORef a)      -- Returns an initialised  reference
writeIORef :: a -> IORef a -> IO () -- Updates the value of a  reference
readIORef  :: IORef a -> IO a        -- Returns the current value
```

All other operations can be expressed in terms of these three operations.

The question now is: if $\tau \leq \sigma$, what is the subtype relationship between IORef τ and IORef σ? To check whether, for example, IORef Int $\leq$ IORef Float, let us have a look at what happens if we apply writeIORef (1.5::Float) to an IORef Int instead of an IORef Float. Clearly, this would not work, as the floating point value can't be stored in an Int reference. It would be ok the other way around: if we writeIORef (1::Int) apply to an IORef Float, it would be fine, since we could first coerce the value to a Float and store the result in the Float reference. This seems to suggest that IORef Float $\leq$ IORef Int.

However, if we assume that IORef Float $\leq$ IORef Int, we run into trouble with readIORef. If readIORef requires an IORef Int, because it should return an Int value as result, and we apply it to an IORef Float instead, readIORef will return a floating point value which we cannot convert into an Int. In this case, the other direction would be fine: if it expects an IORef Float, we could apply it to an IORef Int and then cast the resulting Int value to Float.

This means that $\tau \leq \sigma$ implies no subtype relationship between IORef τ and IORef σ at all: when a reference of a certain type is required, we cannot substitute the reference for a sub- or supertype.

We encounter exactly the same situation with updatable arrays. However, both Java and C# allow arrays to be used as co-variant type constructors. As a consequence, the runtime system in both languages has to perform a runtime check for array stores to preserve type safety.

Type constructors like product and sum, which leave the subtype relationship intact, as called co-variant, type constructors which reverse the relationship, as the function type constructor in its first argument, are called contra-variant, and type constructors like IORef, which do not imply a subtype relationship at all are called invariant.

12 Featherweight Java

We begin our discussion of inheritance, method overriding and subclasses by defining a minimal object oriented language, Featherweight Java ([14]). The language, as we define it here, misses many features which would make the language useful. It doesn't even have arithmetic expressions, or destructive updates! However, we investigated all these things already, so here, we just look at language features which characterise an object oriented language.

12.1 Syntax

As with TinyC, we will just work with the concrete syntax, as this is more convenient. All the symbols in the grammar ending in N, such as $varN$, $classN$, $fieldN$ represent legal names for variables, classes, fields, and so on.

Underlined symbols, such as $\underline{varN}$ or $\underline{classN\ varN}$, denote sequences of the form $varN$, $varN$, ..., and $classN\ varN$, $classN\ varN$, ..., respectively.

$$
\begin{array}{lcl}
prgm & ::= & cdecs\ expr \\
cdecls & ::= & cdecl\ \ cdecls \\
cdecl & ::= & \texttt{class}\ classN\ \texttt{extends}\ classN\ \{\underline{classN\ \ fieldN}\ ;\ cons\ \underline{method}\} \\
cons & ::= & classN\ (\underline{classN\ \ varN}\ )\{\texttt{super(}\underline{varN}\texttt{)};\ \underline{\texttt{this.}fieldN\texttt{=}varN};\} \\
method & ::= & classN\ methodN\ (\ \underline{classN\ \ varN})\ \{\texttt{return}\ e\ \};\\
\tau & ::= & classN \\
e & ::= & varN\ \mid\ e.fieldN\ \mid\ e.methodN(e)\ \mid \texttt{new}\ classN\ (\underline{e})\ \mid\ (classN)\ e
\end{array}
$$

12.1.1 Class declarations.

Class declarations extend an existing class (or, alternatively, they inherit from that class). This means that the new class inherits all the fields and methods from the class it extends. The class `Object` is a predefined, empty class without any fields or methods. The declaration first lists all the names and types (i.e. classes) of the fields it extends in superclass with, followed by a single constructor declaration and its method declarations. Note that a class can only extend a single class. That is, it has only one parent. Some object oriented languages such as C++ or Eiffel do allow the extension of multiple classes, called multiple inheritance, but others (Java, C#, Swift) do not, or only in a restricted form.

12.1.2 Constructor declarations.

Constructor declarations of a class start with that class name, a sequence of its arguments and their types, followed by a call to the superclass' constructor. and initialisation of all its new fields. There are no classifiers, such as `private`, `static`. The type of the constructor of a class is determined by the arguments of the constructor of the class it extends, and the new fields of the class.

12.1.3 Method declarations.

A method declaration is syntacticall similar to a function declaration in TinyC, but only contains a single return statement. We can extend this to add TinyC-like statements, but it would not add any interesting issues we haven't yet looked at in the TinyC context.

12.1.4 Types.

Types are just class names, although to be able to write some more interesting examples, we assume that basic types, such as `int` are present as well.

12.1.5 Expressions.

We include variables x, field selection of the form $e\texttt{.}f$, method ($e.m\texttt{(}e\texttt{)}$) and constructor (`new` $c\texttt{(}e\texttt{)}$) invocation and casts ($\texttt{(}c\texttt{)}e$) in the language. Again, we can add TinyC-like expression, to make the language useful. The identifier `this` is a special variable, whichrefers to the instance itself.

The following code snippet contains examples of class declarations, where we assume that the base type `int` is supported, as well as the modulo operation `%`.

```
class Colour extends Object {
  int red;
  int green;
  int blue;

  Colour (int r, int g, int b) {
    super ();
    this.red   = r % 256;
    this.green = g % 256;
    this.blue  = b % 256;
  }
}

class Point extends Object {
  int x;
  int y;

  Point (int x, int y) {
    super ();
    this.x = x;
    this.y = y;
  }
}
class ColourPoint extends Point {
  Colour c;

  ColourPoint (int x, int y, Colour c) {
   super (x, y);
   this.c = c;
  }
  Colour getc () {return this.c;}
}
```

12.2 Static Semantics of Featherweight Java

What makes the static semantics of Featherweight Java interesting, compared to that of the other languages we looked at so far, is the presence of extendable classes, method overloading, as well as the ability to cast the type (i.e., class) of an expression to a different type. This is new, as we viewed types so far as something purely static. Indeed, we defined type preservation, and important part of type safety, as a necessary condition for a language to be type safe. So, under which conditions should cast be defined? When we investigated subtyping, we found that an item of a given type can always be cast to a more general type, that is, a type which has at least all the properties of the original type. In the context of classes, it means that we allow casts to superclasses of the actual class.

Let us start by formally defining a partial ordering on classes. The `extends` declarations imply a subclass relationship $<:$ between the classes of a program (the transitive closure of the `extend` relation)

- For every class c, we have $c <: c$.
- If c extends c', then $c <: c'$.

- If c $<:$ c' and c' $<:$ c", then $c <: c''$

Or, written as inference rules

$$\frac{}{c <: c} \tag{Subclass-1}$$

$$\frac{c <: c' \qquad c' <: c''}{c <: c''} \tag{Subclass-2}$$

$$\frac{T(c) \texttt{ = class } c \texttt{ extends } c'\{\ldots\}}{c <: c'} \tag{Subclass-3}$$

Every class is a subclass of `Object`, and we can cast an object of class c to an object of class c' if $c <: c'$. That is, varc' has at least as many fields and methods as c. Casts will not be checked statically, only at run-time.

We define a number of judgements to specify statcially valid Featherweight Java programs:

- $\Gamma \vdash e : c$: expression e has type c under the environment Γ.
- m ***ok in*** c: method declaration m is wellformed in the context of class c.
- C ***ok***: the class C is well formed
- T ***ok***: the class table T is well formed
- ***fields*** $(c) = \{c_1\ f_1,\ c_2\ f_2, \ldots\}$: the class c has the fields f_i of type c_i (in the class T) table
- ***type***$(m, c) = \underline{c} \to c'$: the method m of class c returns a value of class c' if invoked on arguments of type $\underline{c}$ (in the class table T)

The class table T is where the type information about all the classes in a program is stored. In some of our judgments, we cheat slightly by using the implicit parameter of class table T.

We start with the ***fields*** function, which returns the names and types of all the fields of a class. The class `Object` as no fields':

$$\frac{}{\textit{fields}(\texttt{Object}) = \bullet} \tag{FJ-static-1}$$

and a class c with extends a class c' has all the fields of c' in addition to the new fields it declares:

$$\frac{T(c) = \texttt{class } c \texttt{ extends } c'\ \{\underline{c\ f}; \ldots\} \qquad \textit{fields}(c') = \underline{\underline{c'\ f'}}}{\textit{fields}(c) = \underline{c'\ f'},\ \underline{c\ f}}$$

Next, we define the typing judgement of the familiar form $\Gamma \vdash e : \tau$, where the type τ is a class name c.

We look up the type of a variable in the environment. For field selectors, we check what the class of the expression is, and return the class of the corresponding field as result type:

$$\frac{x : c \in \Gamma}{\Gamma \vdash x : c} \tag{FJ-static-3}$$

$$\frac{\Gamma \vdash e_0 : c_0 \quad \textit{fields}(c_0) = c_1\ f_1 \ldots}{\Gamma \vdash e_0 . f_i : c_i} \tag{FJ-static-4}$$

To type method invocation m of the class c_0 of an object e_0, we check that the types $c'_1 \ldots c'_n$ of the formal arguments of that method (obtained via ***type***, defined later) are supertypes of the types $c_1 \ldots c_n$ the actual arguments $e_1 \ldots e_n$. If so, the method invocation is well typed, and the result type is the return type of that method.

$$\frac{\Gamma \vdash e_0 : c_0 \quad \Gamma \vdash \underline{e} : \underline{c} \quad \textit{type}(m,\, c_0) = \underline{c'} \to c_1 \quad \underline{c} <: \underline{c'}}{\Gamma \vdash e_0\,.\,m(\underline{e}) : c_1} \tag{\textit{FJ-static-5}}$$

When we instantiate an object with `new`, we also check that the arguments provided to the constructor have the correct type (again, the arguments can be subtypes of the required type):

$$\frac{\Gamma \vdash \underline{e} : \underline{c} \quad \underline{c} <: \underline{c'} \quad \textit{fields}(c) = \underline{c'\ f}}{\Gamma \vdash \texttt{new}\ c(\underline{e}) : c} \tag{\textit{FJ-static-6}}$$

In Featherweight Java, casts are always statically valid. We only check that the expression has some valid type c'. Note that this means we have to ensure in the dynamic semantics that this does not lead to stuck or undefined states.

$$\frac{\Gamma \vdash e_0 : c'}{\Gamma \vdash (c)\,e_0 : c} \tag{\textit{FJ-static-7}}$$

Well-formedness of class declarations:

$$\frac{k = c\ (\underline{c'\ x'},\ \underline{c\ x})\{\texttt{super}(x')\texttt{;this.}f_1\texttt{=}x_1\texttt{;}\ldots\}\ \textit{ok} \qquad \textit{fields}(c') = \underline{c'\ f'} \qquad m_i\ \textit{ok in}\ c}{\texttt{class}\ c\ \texttt{extends}\ c'\ \{\underline{c\ f}\texttt{;}k\ \underline{m}\}\ \textit{ok}} \tag{\textit{FJ-static-8}}$$

To determine the type of a method of a class c, we check if the method is defined in c. If so, we return the type as declared, otherwise, we look for it in its parent class c', until we find it. If its neither in the current class nor any of its superclasses, we assign the dummy type **noType** to it.

$$\frac{T(c) = \texttt{class}\ c\ \texttt{extends}\ c'\ \{\ldots\texttt{;}\ldots\underline{m''}\} \quad m''_i = c_i\ m(\underline{c_i\ x_i})\{\texttt{return}\ e\}}{\textit{type}(m,\, c) = \underline{c_i} \to c_i} \tag{\textit{FJ-static-9}}$$

$$\frac{T(c) = \texttt{class}\ c\ \texttt{extends}\ c'\ \{\ldots\texttt{;}\ldots\underline{m''}\} \qquad m \notin m'' \quad \textit{type}(m,\, c') = \underline{c_i} \to c_i}{\textit{type}(m,\, c) = \underline{c_i} \to c_i} \tag{\textit{FJ-static-10}}$$

$$\frac{}{\textit{type}(m,\, \texttt{Object}) = \textbf{noType}} \tag{\textit{FJ-static-11}}$$

Well-formedness of method declaration, we check whether the method has been declared already in a superclass. If so, the method can be overridden with a method of exactly the same type. If not, then the *type* function returns **noType**, we just check whether the expression e_0 is well-typed, assuming the arguments of the methods have the declared type, and `this` has the type of the current class.

$$\frac{T(c) = \texttt{class}\ c\ \texttt{extends}\ c'\ \{\ldots\} \quad \textit{type}(m,\, c') = \textbf{noType} \quad \underline{x : c}; \texttt{this} : c \vdash e_0 : c_0}{c_0\ m(\underline{c\ x})\{\texttt{return}\ e_0\};\ \textit{ok in}\ c} \tag{\textit{FJ-static-12}}$$

$$\frac{T(c) = \texttt{class}\ c\ \texttt{extends}\ c'\ \{\ldots\} \quad \textit{type}(m,\, c') = \underline{c} \to c_0 \quad \underline{x :\ c}; \texttt{this} : c \vdash e_0 : c_0}{c_0\ m(\underline{c\ x})\{\texttt{return}\ e_0\};\ \textit{ok in}\ c} \tag{\textit{FJ-static-13}}$$

12.3 Dynamic Semantics of Featherweight Java

We now have to specify how expressions in Featherweigth Java should be evaluated. We only hav five different types of expressions: variables, field selection, method invocation, constructors and casts. We don't even have destrutive assignments, so variables are, like in MinHs, variables in the mathematical sense and only get bound to a value once, when they are initialised as fields of a class, or bound via method or constructor invocation. This means we can get away with using a substitution semantics, and do not have to model a state, as we did in TinyC. Of course, almost all real OO languages do have destructive updates, but we know how to handle them, and they are orthogonal to the OO-features we are investigating here.

The final states v in our small steps semantics are constructor invocations, where are arguments are also final states:

$$\frac{\underline{v}\ \textbf{\textit{value}}}{\texttt{new}\ c\ (\underline{v})\ \textbf{\textit{value}}} \qquad (\textit{FJ-value})$$

For all our expressions, if the subexpressions are not yet fully evaluated, we proceed from the left to the right (not that order matters, since we have no side effects!), until all the subexpressions are evaluated. We omit the other rules for the evaluation of the subexpressions, and only list the rules which describe what happens once the subexpressions are evaluated to values.

$$\frac{e \mapsto e'}{\texttt{new}\ c\ (\underline{v}, e, \underline{e}) \mapsto \texttt{new}\ c\ (\underline{v}, e', \underline{e})} \qquad (\textit{FJ-dynamic-1})$$

To evaluate field selectors, we choose the appropriate field from either the class c or one of its superclasses. Since the values of the fields do not change after initialisation, we can just pick the value the fields of the object was initialised with. If we had destructive updates which can alter the contents of the fields of an object, we would need to model this bt threading through a state, like in TinyC.

$$\frac{\textbf{\textit{fields}}(c) = \underline{c'}\ \underline{f'};\underline{c}\ \underline{f};}{\texttt{new}\ c(\underline{v'}, \underline{v}).f'_i \mapsto v'_i} \qquad (\textit{FJ-dynamic-2})$$

$$\frac{\textbf{\textit{fields}}(c) = \underline{c'\ f'};\underline{c\ f};}{\texttt{new}\ c(\underline{v'}, \underline{v})\texttt{.}f_j \mapsto v_j} \qquad (\textit{FJ-dynamic-3})$$

For method invocation, we first define a function ***body***, which, given a method name m, a class c, (and class table as implicit parameter), returns the formal parameters $\underline{x}$ of the method, and its body e. If an object invokes a method, we search upwards in the class hierachy of the dynamic type for the first definition of that method. Since we allow method overriding, there might be other definitions further up in the hierachy. Statically, we don't know which method in the hierachy will be invoked, but since our static semantics rules guarantee that they all have the same type, this is not a problem.

$$\frac{T(c) = \texttt{class}\ c\ \texttt{extends}\ c'\ \{\ldots\underline{m}\} \quad m_i = c_i\ m(\underline{c_i}\ \underline{x})\{\texttt{return}(e)\}}{\textbf{\textit{body}}(m,\ c) = \underline{x} \rightarrow e} \qquad (\textit{FJ-dynamic-3})$$

$$\frac{T(c) = \texttt{class}\ c\ \texttt{extends}\ c'\ \{\ldots\underline{m}\} \quad m \notin \underline{m} \quad \textbf{\textit{body}}(m,\ c') = \underline{x} \rightarrow e}{\textbf{\textit{body}}(m,\ c) = \underline{x} \rightarrow e} \qquad (\textit{FJ-dynamic-4})$$

We again make use of the absence of destructive updates, and simply replace any occurence of a formal parameter name x_i by the value of the actual argument v_i, and every occurence of `this` by the object itself.

$$\frac{\boldsymbol{body}(m,\ c) = \underline{x}{\to}e_0}{\texttt{new}\ c\ (\underline{v})\texttt{.}m(\underline{v'})\ \mapsto e_0[\underline{x} := \underline{v'}][\texttt{this} := \texttt{new}\ c\ (\underline{v})]} \qquad (\textit{FJ-dynamic-5})$$

$$\frac{c <: c'}{(c')\texttt{new}\ c(e)\ \mapsto \texttt{new}\ c(e)} \qquad (\textit{FJ-dynamic-6})$$

In our static semantics, we always accepted a cast as correct, and assigned the cast class as new static type of the object. Thus we need to check in the dynamic semantics if that actual, dynamic type c of the object is a superclass of the cast class c'. If not, the cast evaluates to `error`. This way, we know that if a field of an object is selected, or a method invoked, the field or method is indeed defined for the object. If the cast succeeds, nothing happens to the object. The additional fields and methods are just ignored.

Error values propagate through the evaluation, just like in MinHs. That is, in every dynamic semantics rule, when a subexpression evaluates to `error`. the whole expression does. We do not provide the rules.

$$\frac{c' \not<: c}{(c')\texttt{new}\ c(e)\ \mapsto \texttt{error}} \qquad (\textit{FJ-dynamic-7})$$

This distinction between static and dynamic type is new: both in MinHs and in TinyC, the type of an expression we derived at compile time was the same as the type of that object at any point in time during the execution. In Featherweight Java, the static type might not be the same as the dynamic type. If an object of class c is passed to a method which expects an argument of class c', with $c <: c'$, then the objects static type in the body of the method is c', while its dynamic type is c. This is never a problem, because c' has all the fields and methods of c (if not more).

If an object invokes a method, we search upwards in the class hierachy of the dynamic type for the first definition of that method. Since we allow method overriding, there might be other definitions further up in the hierachy. Since we do not know what the dynamic type of an object is during static checking, we also do not know at that point which method will be invoked. Since we require overriding methods have the same type as the original method, we still can check that the method is invoked with the correct arguments.

$$\frac{T(c) = \texttt{class}\ c\ \texttt{extends}\ c'\ \{\ldots\underline{m}\}\quad m_i = c_i\ m(\underline{c_i}\ \underline{x})\{\texttt{return}(e)\}}{\boldsymbol{body}(m,\ c) = \underline{x}{\to}e} \qquad (\textit{FJ-dynamic-8})$$

$$\frac{T(c) = \texttt{class}\ c\ \texttt{extends}\ c'\ \{\ldots\underline{m}\}\quad m \notin \underline{m}\quad \boldsymbol{body}(m,\ c') = \underline{x}{\to}e}{\boldsymbol{body}(m,\ c) = \underline{x}{\to}e} \qquad (\textit{FJ-dynamic-8})$$

Index

Acknowledgements These lecture notes draw on ideas and material from the draft book on *Programming Languages: Theory and Practice*, Robert Harper and many discussions. They also contain examples adapted from Frank Pfenning's lecture notes for the CMU course *Foundations of Programming Languages*. The notes are adapted from notes I wrote for the Concepts of Programming Languages course I taught at UNSW, Sydney together with Liam O'Connor Davies.

Bibliography

[1] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937.

[2] B. Randell and E. W. Russell, L. J.and Dijkstra. Algol 60 implementation: The translation and use of algol 60 programs on a computer. 1964.

[3] Lennart Augustsson. A compiler for lazy ML. In *LFP '84*, 1984.

[4] Stephan Diehl, Pieter Hartel, and Peter Sestoft. Abstract machines for programming language implementation. *Future Generation Computer Systems*, 16(7):739–751, 2000.

[5] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification, Java SE 8 Edition.* Addison-Wesley Professional, 1st edition, 2014.

[6] Oracle Corporation. https://openjdk.org, 2023.

[7] Microsoft. https://learn.microsoft.com/en-gb/dotnet/, 2023.

[8] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[9] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. *SIGPLAN Not.*, 52(6):185–200, jun 2017.

[10] WebAssembly. https://learn.microsoft.com/en-gb/dotnet/, 2023.

[11] Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding typescript. In *European Conference on Object-Oriented Programming*, pages 257–281. Springer, 2014.

[12] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[13] The Go Team. https://tip.golang.org/doc/go1.18, 2022.

[14] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: A minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, may 2001.