Advanced Topics with Programming Languages

Josh Felmeden

October 19, 2021

Contents

1 Introduction

One way of looking at programming languages is to look at **types** and **type systems**. Haskell is a language that uses typing. There can be static and dynamic typing.

Types classify programs by the kind of data they compute.

2 Judgements

A **judgement** is a statement. In this topic, we will centre everything around an *evident judgement*. A judgement becomes evident when you can *prove* it. Therefore, when we sa a judgement, we need to provide evidence of proof.

Judgements come with rules. Here is an axiom:

```
zero nat
```

Zero is the object, and nat is the name. Alongside this, we can use an inference rule:

```
n nat           'premise(s)
------------- s 'name of rule
succ(n) nat     'conclusion
```

These two structures can be used in **derivation trees** which are used to prove judgements. For example, to prove that two is a natural number, we can do the following:

```

--------- axiom
zero nat
-------- s
succ(zero) nat
--------- s
succ(succ(zero)) nat
```

We can also write

```
data nat = zero | succ nat
```

2.1 Simultaneous rules

we can state proofs of rules mix and match to use a proof that proves two things at once. For example:

```

-------- ZE
zero even
```

```
n even
---------- ODD
succ(n) odd
```

```
n odd
----------- EVEN
succ(n) even
```

This proves both odd and even.

3 Induction

Every set of rules generates an *induction principle*.

Consider the claim `if succ(n)nat then n nat`. This seems obvious, but we can actually prove this.

Proof We will use induction
`P(n)`: 'If n nat and n = succ(x) for some x then x nat'
`Case zero`: Nothing to prove
`Case(succ(n) nat)` The derivation of succ(n) nat ends with

```
n nat
--------- succ
succ(n) nat
```

The D is a derivation of n nat.
succ(n) = succ(x) and therefore n = x. We can conclude that n is nat and therefore x is nat.

This statement is an **admissible rule**. A rule is admissible when we have a derivation of the premises, then we know we can construct a derivation of the conclusion. In essence, you need to *prove* this one (usually by induction).

In contrast, a rule is **derivable** if we can use a derivation of its premise as a building block in deriving its conclusion. In essence, you can *infer* this one (stitch together stuff).

3.1 Simultaneous induction

Recalling the even and odd proof, we can write these as Let P(n even) and Q(n odd). If:

- P(zero) and
- whenever n even and $\mathcal{P}(n)$ we have $\mathcal{Q}(\text{succ}(n))$ and
- whenever n odd and $\mathcal{Q}(n)$ we have $\mathcal{P}(\text{succ}(n))$

We are allowed to *invert* a judgement, and this is called an *inversion principle*.

4 Types

Term e is **well-typed** iff there is τ such that $\emptyset \vdash e : \tau$ is derivable according the the *static* rules of the language.

Say we want to prove the following:

$$\emptyset \vdash \text{ let}(\text{str}[\text{my}]; x, (\text{times}(\text{len}(x); \text{num}(0))))$$

Type systems restrict the set of allowed programs.

4.1 Basic properties of typing

Lemma (Inversion of Typing): Suppose that $\Gamma \vdash e : \tau$. If $e = \text{ plus}(e_1; e_2)$ then $\tau = \text{ num}, \Gamma \vdash e_1 : \text{ num}$ and $\Gamma \vdash e_2 : \text{ num}$ and similarly for the other constructs of the language.

Lemma (Unicity of Typing): For every typing context Γ and expression e there exists at most one τ such that Γ such that $\Gamma \vdash e : \tau$.

Lemma (Weakening): If $\Gamma \vdash e' : \tau'$ then $\Gamma, x : \tau \vdash e' : \tau'$ for any $x \notin dom(\Gamma)$ and any type τ.

Lemma (Substitution): If $\Gamma, x : \tau \vdash e' : \tau'$ and $\Gamma \vdash e : \tau$ then $\Gamma \vdash e'[e/x] : \tau'$

5 Dynamics

Now, we are going to look at the runtime *semantics*. A **value** is an atomic structure that cannot be reduced any more (like a string or a value). Once we have that as a program, we know we don't need to evaluate it any more.

Now, let's define the actual semantics of language **E**. It is defined by the form: The transition judgement between states is inductively defined by the following rules. If we have some two argument (such as plus), evaluate the left hand side into a value first, before doing the second. While this doesn't make a difference to plus, it makes a different for more complicated things.

At this point, if we have something that doesn't match the type, we end up being 'stuck'. Additionally, we introduce the symbol: $\longmapsto^*$, which means derives in multiple steps. This is transitive: $e_1 \longmapsto^* e_2$, $e_2 \longmapsto^* e_3 \rightarrow e_1 \longmapsto^* e_3$.

Propositions

If e val, then there is no e' such that $e \longmapsto e'$.
If $e \longmapsto e_1$ and $e \longmapsto e_2$ then $e \equiv e_2$.

To ensure **type safe** programming languages, we know a few things:

- Certain kinds of mismatches cannot happen at runtime (such as `"one" == 123`)
- Type safety expresses the *coherence* between statics (Types) and dynamics (semantics)
- A consequence of type safety is that evaluation cannot get stuck.

From here onwards, we write $\vdash e : \tau$ for $\emptyset \vdash e : \tau$.

Theorem (type safety)

1. If $\vdash e : \tau$ and $e \longmapsto e'$, then $\vdash e' : \tau$ (*type preservation*)
2. If $\vdash e : \tau$, then either e val, or there exists e' such that $e \longmapsto e'$ (*progress*)

6 Lambda Calculus

So, well typed programs are very cool and all. But, if we were to add a division operator, what would happen if we divide by 0? This is well typed but the program can still get stuck. We can either define the rule with a 0 divisor rule, or add a check.

We can check at runtime to return an error so that the computer still returns something (an error, which is not a type). These errors need to be differentiated:

- **Unchecked error**: ruled out by the type system. No run-time checking performed because the type system rules out the possibility of the error arising
- **Checked error**: not ruled out by the type system, hence run-time check must occur

Important to differentiate between the two because the checked error will incur significant overhead.

Error, therefore is a new type:

$$\frac{}{\texttt{error}\ \mathsf{err}}$$

6.1 Preserving Type Safety with Error

Theorem Progress with Error

If $\vdash e : \tau$ then either e err, e val or there exists e' such that $e \longmapsto e'$.

6.2 Binary Product

We will introduce a new type: binary **product**. This looks like:

$$\begin{aligned} \text{Type } \tau ::= &\ \ldots (\text{as in E}) \\ &\ \texttt{prod}(\tau_1; \tau_2) \tau_1 \times \tau_2 \text{ binary product} \end{aligned}$$

Exp e ::= … (as in E)		
$\texttt{pair}(e_1; e_2)$	$\langle e_1, e_2 \rangle$	ordered pair
$\texttt{pl}(e_1; e_2)$	$\langle e_1 \rangle$	left projection
$\texttt{pr}(e_1; e_2)$	$\langle e_2 \rangle$	right projection