~min sheet~

Dynamics - Values

It's now time to learn about the dynamics of our little language -- how it runs/evaluates/reduces to a value.

To keep things simple, for now, we are going to view the goal of a program to be just computing a value.

Our programs will compute values.

Of course in reality, programs can do so much more e.g. print things, perform other IO, but for now we will ignore that to keep things simple.

For our language, we will specify what it means to be a value with the following val judgement:

$$\text{VAL-NUM}\ \frac{n \in \mathbb{N}}{\mathrm{num}[n]\ \mathbf{val}}$$

Q What does this judgement say?

We should be adept at interpreting judgements defined by rules now.

VAL-NUM says if $n \in \mathbb{N}$, then $\mathrm{num}[n]$ is a value.

The only other values in our language are strings.

Q Can you help me complete the specification of the value judgement by adding a rule for strings?

$$\text{VAL-STR}\ \frac{s \in \Sigma^*}{\mathrm{str}[s]\ \mathbf{val}}$$

Q Since we have said that only $\mathrm{num}[n]$ and $\mathrm{str}[s]$ are values, what can we say about the type of a value?

If e **val** then either $\vdash e : \mathrm{Num}$ or $\vdash e : \mathrm{Str}$

Q Who can remember the difference between closed/open expressions?

Open expressions have free variables. Closed expressions do not.

Q Are values closed or open expressions?

Closed

Dynamics — Transitions

Now we have our goal, values, we will specify our dynamics as a transition system over terms that evaluates them to values.

Note that we are using operational semantics to specify our language but other styles are available (e.g. denotational semantics, which we are not teaching because, due to its similarity to Haskell, is easier, and it gives you less control over the operation (e.g. eval order) of your language because like Haskell, it is more declarative. Feel free to take a look tho!)

Our transition system is going to consist of rules where:

Transition system:
- closed terms = states
- instruction transitions, which perform computation
- search transitions, which determine the evaluation order of the language

Rules:

$$\text{D-PLUS}\quad \frac{n_1 + n_2 = n}{\text{plus}(\text{num}[n_1]; \text{num}[n_2]) \mapsto \text{num}[n]}$$

D-PLUS is an instruction transition that states if we have two numbers that are being added, then we can evaluate the plus term to its result.

Note that we only perform this calculation on num values. This is because our evaluation order for plus is very deterministically defined, as we will see in the next rules: the first argument must be evaluated, then the second, then and only then do we perform the addition.

$$\text{D-PLUS-1}\quad \frac{e_1 \mapsto e_1'}{\text{plus}(e_1; e_2) \mapsto \text{plus}(e_1'; e_2)}$$

D-PLUS-1 says: if we can do work/reduce the first value, then do so in place.

$$\text{D-PLUS-2}\quad \frac{e_1\ \text{val} \quad e_2 \mapsto e_2'}{\text{plus}(e_1; e_2) \mapsto \text{plus}(e_1; e_2')}$$

D-PLUS-2 says: once e_1 has been fully evaluated to a value, we can do some work on e_2. If we can reduce it, we will do so in place.

Q say we wanted to be less prescriptive and deterministic (we probably don't because a deterministic evaluation order is good, but lets say we did). How would we adjust these rules so you could do work on either e_1 or e_2, and didn't have to wait for e_1 to be a val before reducing e_2

$$\text{D-PLUS-2}\quad \frac{e_2 \mapsto e_2'}{\text{plus}(e_1; e_2) \mapsto \text{plus}(e_1; e_2')}$$

The Cat and Times dynamics are very similar to Plus, just the names and the performed op change.

Q Could you help me with len though?

$$\text{D-LEN}\quad \frac{|s| = n}{\text{len}(\text{str}[s]) \mapsto \text{num}[n]}$$

Performs the length computation once the argument of len is a string value.

$$\text{D-LEN-1}\quad \frac{e \mapsto e'}{\text{len}(e) \mapsto \text{len}(e')}$$

Steps the argument of len if it is not a value.

As I alluded to last time, substitution will be involved in our specification of the dynamics of the language. It appears in the dynamics of our binder Let.

$$\text{D-LET}\quad \frac{}{\text{let}(e_1; x.e_2) \mapsto e_2[e_1/x]}$$

It says: substitute in the bound expression for the bound variable.

These rules mean that we can evaluate terms of our language step by step, where each step is justified by a derivation of these dynamics.

$$\text{plus}(\text{len}(\text{str}[\text{'asdf'}]); \text{num}[1])$$
$$\mapsto$$
$$\text{plus}(\text{num}[4]; \text{num}[1])$$

For readability it is convention to highlight or underline the term that gets to transformed.

Then we justify this transition with the following derivation:

$$\cfrac{\cfrac{|\text{'asdf'}| = 4}{\text{len}(\text{str}[\text{'asdf'}]) \mapsto \text{num}[4]}\ \text{D-LEN}}{\text{plus}(\text{len}(\text{str}[\text{'asdf'}]); \text{num}[1]) \mapsto \text{plus}(\text{num}[4]; \text{num}[1])}\ \text{D-PLUS-1}$$

Dynamics - Multi-Step transitions

Of course to evaluate a program, we want to make multiple steps. In fact we want to keep stepping till we hit a value:

plus(len(str['asdf']); num[1])
$\mapsto$
plus(num[4]; num[1])
$\mapsto$ Q what next?
num[5]

Transition Sequence

Q What rule justifies the final step?
D-PLUS

To create these transition sequences, we define the

reflexive transitive closure

of our dynamics

Q Who knows what reflexivity means for a relation

every element is related to itself (represents taking no steps)

Q What about transitive?

relations can be chained together (represents adding one more step to the sequence)

Here is the reflexive transitive closure of $\mapsto$, $\mapsto^*$ in rules:

$$\text{D-MULTI-REFL}\ \frac{}{e \mapsto^* e}$$

This says that the empty sequence of transitions is a valid sequence, that any term can take 0 steps to itself.

$$\text{D-MULTI-STEP}\ \frac{e \mapsto e' \qquad e' \mapsto^* e''}{e \mapsto^* e''}$$

This rule says that if we take one step, then we can add it to our sequence.

Note that * is generally used to denote the RTC, and these rules can be reused for any RTC and transition rules by just changing the symbols. In fact they are exactly the rules for a list of transitions.

Q Say we had the transition system $\ggg$ and we wanted to define the RTC of it $\ggg^*$. How would we do that?

$$\ggg\text{REFL}\ \frac{}{e \ggg^* e}$$

$$\ggg\text{STEP}\ \frac{e \ggg e' \qquad e' \ggg^* e''}{e \ggg^* e''}$$

Dynamics - Properties

This is just highlighting some points that i've made along the way as we met our dynamics

(Finality) If e **val** then there is no e' with $e \mapsto e'$

Proof by inspection

This is the point that values were our goal, so after we hit a value there are no more transitions that are applicable and this is by design of our dynamics.

(Determinism) If $e \mapsto e_1$ and $e \mapsto e_2$ then $e_1 \equiv e_2$ (up to α-equiv)

This is a very strong statement, that we only get because we have been very careful with our rule definition. We have defined them such that only one transition is ever available, so our language is completely deterministic.

NOTE (for TLC kids): this is different to $\rightarrow_\beta$ in TLC, which was non-deterministic in its evaluation order (you could reduce anywhere in a term), but in TLC we regained determinism in our results via confluence. So same goal: one answer, different methods of achieving it.

Notation:

$$\underbrace{e \Downarrow v}_{e \text{ evaluates to}} \overset{\text{def}}{\equiv} \underbrace{e \mapsto^* v}_{\text{there is a transition sequence from } e \text{ to } v} \wedge \underbrace{v \textbf{ val}}_{v \text{ is a value}}$$

+

Due to the determinism of our dynamics, v is unique.

Type Saftey

This is probably the most important lecture of the course. This is where we will prove the type saftey of our little language. The little lang is now minimally complete, so this is the big boy important proof we must do over it to ensure it is good.

The rest of the course will be adding features to the language, and extending this proof of type saftey.

So pay attention. I can't emphasise the importance of ~~this~~ theorem and being able to prove it enough.

Hopefully, if you are sitting in this room having chosen this course I don't need to convince you that types and typesaftey is important, but lets define specifically what we mean by type saftey and what it brings to the table

(Type Saftey)

1. (Preservation) (saftey) If $\vdash e : \tau$ and $e \mapsto e'$ then $\vdash e' : \tau$

2. (Progress) (liveness) If $\vdash e : \tau$ then either e val or $e \mapsto e'$ for some e'

Type saftey states that well-typed and closed programs don't go wrong, and it breaks not going down into two criteria:

1. types are preserved under evaluation

2. if we are not done, if we don't have a value, we can and will continue evaluating.

We will now look at each part and how to prove it (induction) in detail

Preservation

I will now start the proof of preservation for you, doing the hardest case.

It is proven by induction and during the proof, we will need to use the lemmata we met last week.

You will complete the proof in this week's problem sheet.

(Preservation) If $\vdash e : \tau$ and $e \mapsto e'$ then $\vdash e' : \tau$

Proof by induction on $e \mapsto e'$

because this is where the action is literally happening.

[case: D-LET]

$$\text{GOAL} = \text{if } \vdash \text{let}(e_1; x.e_2) : \tau \text{ and } \text{let}(e_1; x.e_2) \mapsto e_2[e_1/x] \text{ then } e_2[e_1/x] : \tau$$

$$P1 = \vdash \text{let}(e_1; x.e_2) : \tau$$
$$P2 = \text{let}(e_1; x.e_2) \mapsto e_2[e_1/x]$$

$$\text{GOAL}' = e_2[e_1/x] : \tau$$

In this case, we have no IH, so we need to consider our helpful lemmata and how they will help us solve this.

Since we are trying to prove something about the type of a let expression, it makes sense to use the inversion lemma to learn more about the types.

By inversion on P1, we know that there must exist a type σ such that

- $\vdash e_1 : \sigma$ (INV1)
- $x : \sigma \vdash e_2 : \tau$ (INV2)

Since our goal is to assert something about the type of a substitution, we will want the substitution lemma:

(Subst) If $\Gamma \vdash e : \tau$ and $\Gamma, x : \tau \vdash u : \sigma$ then $\underbrace{\Gamma \vdash u[e/x] : \sigma}_{\text{this is exactly what we want to prove!}}$

In our case:

$e = e_1$
$\tau = \sigma$
$\sigma = \tau$
$\Gamma = \{\}$
$u = e_2$

Specialised subst:

If $\underbrace{\vdash e_1 : \sigma}_{\text{INV1}}$ and $\underbrace{x : \sigma \vdash e_2 : \tau}_{\text{INV2}}$ then $\underbrace{\vdash e_2[e_1/x] : \tau}_{\text{GOAL}}$

We can conclude our goal by applying the substitution lemma to INV1 and INV2

$\square$ D-LET

NOTE:- I of course knew the proof so this seemed easy

But proofs are not easy. For tips, see Proofs.pdf

(show them P5 of the note + go through)

+ the more proofs you do the better you'll be because you know what generally happens

e.g. if we use subst in a statement that needs proven, we'll probably want the subst lemma

Highlight key lemmata + theorems pdf + how they will get this in exam

Canonical Forms

Before we can prove progress, we need one more lemma.

Similarly to Inversion, this lemma is rule set specific and proven by inspection.

For our language, it states:

(Canonical Forms) Suppose e val.

1. If $\vdash e : \text{Num}$ then $e = \text{num}[n]$ for some $n \in \mathbb{N}$

Q help me with str

2. If $\vdash e : \text{Str}$ then $e = \text{str}[s]$ for some $s \in \Sigma^*$

Proof by inspection

So you can see that this is a formalisation of the nice property of our rules that if a term is a value and a Num then it must be num[n], likewise for Str.

Progress

For our intro to progress, again I will start the proof and you will complete it in the sheet.

(Progress) If $\vdash e : T$ then either e val or $e \mapsto e'$ for some e'.

Proof by induction on $\vdash e : T$.

only premise, so no choice.

[Case: PLUS]

GOAL = If $\text{plus}(e_1; e_2) : T$ then either e val or $\text{plus}(e_1; e_2) \mapsto e'$ for some e'

$P1 = \text{plus}(e_1; e_2) : T$

GOAL' = either e val or $\text{plus}(e_1; e_2 \mapsto e'$ for some e'

aka, we need to produce a judgement e val or a term e' that plus reduces to.

Looking at our VAL judgement, we will need to do the latter

We will show the latter (since PLUS is not a value.

Since we are working with PLUS, we will have two IHs:

IH1 = if $\vdash e_1 : T_1$ then either e_1 val or $e_1 \mapsto e_1'$ for some e_1'

@what is IH2?

IH2 = if $\vdash e_2 : T_2$ then either e_2 val or $e_2 \mapsto e_2'$ for some e_2'

Looking at these, we can see that we need a judgement about the type of e_1 and e_2 to use the conclusions of the IHs.

We can get this by inversion

By inversion on P1 we can conclude:

- $T = \text{Num}$ (INV1)
- $e_1 : \text{Num}$ (INV2)
- $e_2 : \text{Num}$ (INV3)

By applying IH1 to INV2, we can conclude:

UIH1 = either e_1 val or $e_1 \mapsto e_1'$ for some e_1'

@help me unlock IH2

By applying IH2 to INV3, we can conclude:

UIH2 = either e_2 val or $e_2 \mapsto e_2'$ for some e_2'

Because what we have here is two eithers, we need to proceed by case analysis

We proceed by case analysis on UIH1 and UIH2

[Subcase: e_1 val]

[Subsubcase: e_2 val]

Now we need to use canonical forms

By canonical forms, we have

$e_1 = \text{num}[n_1]$
$e_2 = \text{num}[n_2]$

This means that we can apply the D-PLUS rule to get our e' and achieve our GOAL' in this case

The nice thing about specific cases and the determinism of our language is that we only have one choice, making proofs easier.

$$\text{DPLUS}\ \frac{n_1 + n_2 = n}{\text{plus}(n_1; n_2) \mapsto \text{num}[n]}$$

Thus $e' = \text{num}[n]$ and we are done in this case $\square_{\text{D-PLUS},\, e_1 \underline{val},\, e_2 \underline{val}}$

[Subsubcase: $e_2 \mapsto e_2'$ for some e_2']

This and our case means that we can achieve our goal via the D-PLUS-2 rule:

$$\text{D-PLUS-2}\ \frac{e_1\ \text{val} \qquad e_2 \mapsto e_2'}{\text{plus}(e_1; e_2) \mapsto \text{plus}(e_1; e_2')}$$

Thus $e' = \text{plus}(e_1; e_2')$ and we are done in this case $\square_{\text{D-PLUS},\, e_1 \underline{val},\, e_2 \mapsto e_2'}$

[Subcase: $e_1 \mapsto e_1'$ for some e_1']

Actually in this subcase, it doesn't matter whether or not e_2 is a value or not

Regardless of what e_2 is, we can produce e' using the D-PLUS-1.

$$\text{D-PLUS-1}\ \frac{e_1 \mapsto e_1'}{\text{plus}(e_1; e_2) \mapsto \text{plus}(e_1'; e_2)}$$

Thus we have achieved our goal with $e' = \text{plus}(e_1'; e_2)$

$\square_{\text{D-PLUS},\, e_1 \mapsto e_1'}$

We have exhaustively covered all the cases. Thus we are done for D-PLUS

$\square$ D-PLUS