Spare-Time Teaching Note Collection

Christian Clausen
Kent Grigo
Nicolaj Græsholt
Mikkel Brun Jakobsen
Mikkel Kringelbach
Richard Möhn
Mathias Pedersen
Frederik Brinck Truelsen

2015, Winter Edition

Foreword

The Spare-Time Teaching Note Collection is a bi-annual collection of notes, tutorials, exercises, and other material made or remade by students. It is not intended for research but rather useful or entertaining things that someone feels like sharing. Anyone can submit anything, however it has been peer-reviewed and the included content is deemed to be instructive or entertaining. Material from the Spare-Time Teaching Friday Talks is automatically accepted.

Apart from the Friday Talk material, this collection contains:

- Short overview reports
- Tutorials
- Interesting reports of hard problems
- New perspective on something known
- Useful lookup tables
- Generally interesting stuff

The topics of this edition range from: Lambda Calculus, computer-game making (game design, graphics programming), programming-language transformations (abstract interpretation, obfuscation, program transformations), and much more.

The content of this collection is not to be considered as research, but rather as experience reports, tutorials, or supplementing material for presentations.

There are two major categories of papers in this edition: Notes from Friday talks, which apear in order by the date of the talk, these are not nessasarily self contained. The other category is self contained entertaining or instructive papers, which apear without date.

Acknowledgments The organizers of Spare-Time Teaching would like to thank all the speakers, writers, and attendees who have donated some of their spare time to help and entertain other students. Without whom, Spare-Time Teaching wouldn't be possible.

We would also like to thank the reviewers who helped edit, correct, and prepare this edition of the Note Collection: Kent Grigo, Richard Möhn, Thor Bagge, and Christian Clausen.

Finally, from everyone who is enjoying the events of Spare-Time Teaching, we would like to thank the Department of Computer Science and IT-Vest for their support, which helps us expand and further improve the environment at the events.

Contents

A Paper on the Expectancy of Papers

The Authors Name
OhMyUnix Institute of Technology

A date of final editing

A short overview of the paper presenting the sections and results of the paper. The paper will contain a number of appropriately named sections, each section presenting an expectancy of the specific section along with an amount of relevant arguments, material, or references backing the paper up. Our study concludes that you will have expected something else from this paper[1].

1. Research

This section provides an overview of the research that's been done during the writing of this section. The main sources of research has been to read this section a number of times, while every time elaborating a tiny amount on the provided material in order to be better suited in reflecting on and providing constructive criticism for the material.

2. Data

2.1. A Graph

You may consult the graph below, you are also allowed to choose not to do so,

[1] Even though it's written in LaTeX like a proper research paper would be.

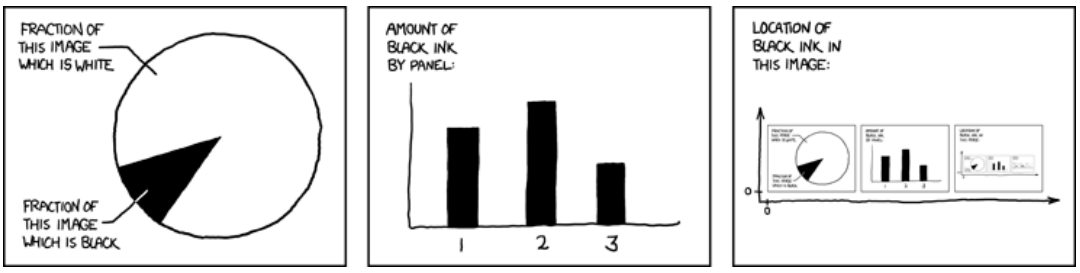


Figure 1: The contents of any one panel are dependent on the contents of every panel including itself. The graph of panel dependencies is complete and bidirectional, and each node has a loop. The mouseover text has two hundred and forty-two characters[2].

The graph is provided without the consent of the author, also as you might've expected.

2.2. A Table

The table below shows a comparison of the running times between an algorithm we worked on during the three years we had to complete the task,

Algorithm	Lines of code	Running time
Our impl.	6785 [3]	$1 - 1.4 \cdot 10^{-32}$
Naïve impl.	10	1

The running time is given relative to the running time of the naïve implementation and as can be seen above, our algorithm is clearly better.

When removing the relativity, the result becomes even more impressive as it actually took up to 9.677 seconds to solve the problem using the naïve implementation on an 2.4 GHz intel core 2 duo processor.

3. Observations

1. When the summary stated that each section would present an expectancy of the specific section, if you read it like the section would introduce itself and not like it would present an actual instance of an arbitrary expectancy from such a particularly named section; you'd be wrong, otherwise you'd be right.

2. You will have expected more from this section.

[2] `http://xkcd.com/688/`

[3] The implementation is actually 6793 with comments.

3. Notice that we can generalize notion of the graph in section 2.1 since any page actually is a graph of the distribution of content on the page. A funny observation on such a graph is that it's continuously updated to match the data supplied.

4. This bullet provides the fourth observation.

4. Proof

By imperialism,

Proof. You agree. □

The proof above builds on an expectancy of the reader agreeing with the points made by the paper. Futhermore, if the reader assumes that the proof above is also a direct proof of the Riemann hypothesis, then the proof above proves the Riemann hypothesis for the reader.

5. Conclusion

You did expect more from this section as well, if not even the entire paper. However you didn't expect a wordcount of the number of times the word "expected" has been overused in the paper; this section will not provide one.

6. References

There's a single reference, as expected, since the paper only made a single reference.

- Self-Description; `http://xkcd.com/688/`

Combinator Gymnastics

Christian Clausen

August 21, 2014

1. Reminder: Pairs and Tuples

```
mk_pair ≡ λ a b c . c a b
π¹₁ ≡ λ p . p (λ a b . a)
π²₂ ≡ λ p . p (λ a b . b)
<x, y> ⇝ mk_pair a b = λ c . c x y
mk_tuple³ ≡ λ a b c d . d a b c
π³₁ ≡ λ p . p (λ a b c . a)
π³₂ ≡ λ p . p (λ a b c . b)
π³₃ ≡ λ p . p (λ a b c . c)
mk_tuple⁴ ≡ λ a b c d e . e a b c d
π⁴₁ ≡ λ p . p (λ a b c d . a)
...
```

Pretty regular structure. Can we make the function `mk_mk_tuple` N?

2. Basis

For closed terms. By induction on t:

$$\begin{aligned}
\mathcal{A}[\![t_1\ t_2]\!] &= \mathcal{A}[\![t_1]\!]\ \mathcal{A}[\![t_2]\!] \\
\mathcal{A}[\![\lambda x.\ x]\!] &= I \\
\mathcal{A}[\![\lambda x.\ \lambda y.\ t]\!] &= \mathcal{A}[\![\lambda x.\ \mathcal{A}[\![\lambda y.\ t]\!]]\!] \\
\mathcal{A}[\![\lambda x.\ t]\!] &= K\ \mathcal{A}[\![t]\!] \quad \text{if } t \text{ doesn't use } x \\
\mathcal{A}[\![\lambda x.\ t_1\ t_2]\!] &= S\ \mathcal{A}[\![\lambda x.\ t_1]\!]\ \mathcal{A}[\![\lambda x.\ t_2]\!]
\end{aligned}$$

```
I = λ x . x
K = λ x y . x
S = λ f g x . f x (g x) = λ f g x . (f x) (g x)
```

We now have a 3-point basis for the closed lambda terms.

However, we know that:

```
S K I  →β  λ x . K x (I x)  →β  λ x . x = I
or
S K S  →β  λ x . K x (S x)  →β  λ x . x = I
...
S K M  →β λx . K x (M x)  →β λx . x = I
therefore:
S K K  →β  λ x . K x (K x)  →β  λ x . x = I
```

And thus we have a 2-point basis.

2.1. Example

```
A[λ a b c . c a b] = A[λ a . A[λ b c . c a b]]
= A[λ a . A[λ b . A[λ c . c a b]]]
= A[λ a . A[λ b . S A[λ c . c a] A[λ c . b]]]
= A[λ a . A[λ b . S (S A[λ c . c] A[λ c . a]) (K b)]]
= A[λ a . A[λ b . S (S I (K a)) (K b)]]
= A[λ a . S A[λ b . S (S I (K a))] A[λ b . K b]]
= A[λ a . S (K S (S I (K a))) (S A[λ b . K] A[λ b . b])]
= A[λ a . S (K S (S I (K a))) (S (K K) I)]
= S A[λ a . S (K S (S I (K a)))] A[λ a . (S (K K) I)]
= S (S A[λ a . S] A[λ a . K S (S I (K a))]) (K (S (K K) I))
= S (S (K S) (S A[λ a . K] A[λ a . S (S I (K a))])) (K (S (K K) I))
= S (S (K S) (S (K K) (S A[λ a . S] A[λ a . S I (K a)]))) (K (S (K K) I))
= S (S (K S) (S (K K) (S (K S) (S A[λ a . S] A[λ a . I (K a)])))) (K (S (K K) I))
= S (S (K S) (S (K K) (S (K S) (S (K S) (S A[λ a . I] A[λ a . K a])))) (K (S (K K) I))
= S (S (K S) (S (K K) (S (K S) (S (K S) (S (K I) (S A[λ a . K] A[λ a . a])))))) (K (S (K
= S (S (K S) (S (K K) (S (K S) (S (K S) (S (K I) (S (K K) I)))))) (K (S (K K) I))
```

3. More Detailed Basis

Again, for closed terms. By induction on t:

$$\begin{aligned}
\mathcal{B}[\![t_1\ t_2]\!] &= \mathcal{B}[\![t_1]\!]\ \mathcal{B}[\![t_2]\!] \\
\mathcal{B}[\![\lambda x.\ x]\!] &= I \\
\mathcal{B}[\![\lambda x.\ \lambda y.\ t]\!] &= \mathcal{B}[\![\lambda x.\ \mathcal{B}[\![\lambda y.\ t]\!]]\!] \\
\mathcal{B}[\![\lambda x.\ t]\!] &= K\ \mathcal{B}[\![t]\!] \quad \text{if } t \text{ doesn't use } x \\
\mathcal{B}[\![\lambda x.\ t_1\ t_2]\!] &= B\ \mathcal{B}[\![t_1]\!]\ \mathcal{B}[\![\lambda x.\ t_2]\!] \quad \text{if } t_1 \text{ doesn't use } x \\
\mathcal{B}[\![\lambda x.\ t_1\ t_2]\!] &= C\ \mathcal{B}[\![\lambda x.\ t_1]\!]\ \mathcal{B}[\![t_2]\!] \quad \text{if } t_2 \text{ doesn't use } x \\
\mathcal{B}[\![\lambda x.\ t_1\ t_2]\!] &= S\ \mathcal{B}[\![\lambda x.\ t_1]\!]\ \mathcal{B}[\![\lambda x.\ t_2]\!]
\end{aligned}$$

```
B = λ f g x . f (g x)
C = λ f g x . (f x) g = λ f g x . f x g
```

We now have a 5-point basis for the closed lambda terms.

We can add an η-rule before the "B-rule":

$$\mathcal{B}[\![\lambda x.\ t\ x]\!] = \mathcal{B}[\![t]\!] \quad \text{if } t \text{ doesn't use } x$$

3.1. Example

```
B[λ a b . b a] = B[λ a . B[λ b . b a]]
= B[λ a . C B[λ b . b] a]
= B[λ a . C I a]
= C I

B[λ a b c . c a b] = B[λ a . B[λ b c . c a b]]
= B[λ a . B[λ b . B[λ c . c a b]]]
= B[λ a . B[λ b . C B[λ c . c a] b]]
= B[λ a . B[λ b . C (C B[λ c . c] a) b]]
= B[λ a . B[λ b . C (C I a) b]]
= B[λ a . C (C I a)]
= B C B[λ a . C I a]
= B C (C I)

B[λ a b c d . d a b c] = B (B C) (B C (C I))
```

$$\langle n, t\rangle \xrightarrow{f} \langle S\ n, n\ B\ C\ t\rangle$$

```
f ≡ λ p . <S (π₁² p), (π₁² p) B C (π₂² p)>
mk_mk_tuple ≡ λ n . π₂² (n f <0, I>)
```

4. Exercises

Warm-up Prove that $\langle a, b\rangle$, π_1^2, and π_2^2 are mutually correct.
(Hint: $\pi_1^2\langle a, b\rangle = a$, and $\pi_2^2\langle a, b\rangle = ?$)

1. Apply the $\mathcal{A}$ algorithm to $\omega = \lambda x.\ x\ x$.
2. Apply the $\mathcal{A}$ algorithm to $\Omega = \omega\ \omega$.
3. Apply the $\mathcal{A}$ algorithm to $Y_{Turring} = A\ A$, where $A = \lambda xy.\ y\ (x\ x)$.

4. ♡ Discover a 1-point basis.
5. Apply the $\mathcal{B}$ algorithm to $\omega = \lambda x.\ x\ x$.
6. Apply the $\mathcal{B}$ algorithm to $\Omega = \omega\ \omega$.
7. Apply the $\mathcal{B}$ algorithm to $Y_{turring} = A\ A$, where $A = \lambda xy.\ y\ (x\ x)$.
8. Apply $\mathcal{B}$ to π_2^2.
9. Apply $\mathcal{B}$ to π_3^3.
10. Apply $\mathcal{B}$ to π_4^4.
11. Implement the lambda term `last N` $= \pi_N^N$.
12. Apply $\mathcal{B}$ to π_1^2.
13. Apply $\mathcal{B}$ to π_1^3.
14. Apply $\mathcal{B}$ to π_1^4.
15. Implement the lambda term `fst N` $= \pi_1^N$.
16. If you haven't done so already, implement the $\mathcal{B}$ algorithm in a programming language.
17. Run your implementation of the $\mathcal{B}$ algorithm on:
    a) π_1^3, π_2^3, π_3^3.
    b) π_1^4, π_2^4, π_3^4, π_4^4.
    c) π_1^5, π_2^5, π_3^5, π_4^5, π_5^5.
18. ♡ Implement the lambda term $\pi\ N\ n = \pi_n^N$.
19. Implement the $\mathcal{A}$ algorithm in a programming language.
20. Run your implementation of the $\mathcal{A}$ algorithm on:
    a) π_1^3, π_2^3, π_3^3.
    b) π_1^4, π_2^4, π_3^4, π_4^4.

    Compare the results with the results from exercise 17, are they as structured? As meaningful?
21. ♡ Implement a variadic version of `S N` $\mathtt{f}_1$ $\mathtt{f}_2$ `...` $\mathtt{f}_n$ `x` that will pass the last argument to all $\mathtt{f}_i$.
22. A function is called variadic when a it does not have a fixed number of arguments. Implement a variadic version of `K N x` $\mathtt{y}_1$ $\mathtt{y}_2$ `...` $\mathtt{y}_n$ that will always return the first argument.

23. ♡ Implement a variadic function `V N M` $\texttt{f}_1$ $\texttt{f}_2$ `...` $\texttt{f}_m$ `...` $\texttt{f}_n$ `x` that will pass the last argument to $\texttt{f}_m$.

24. ♡ Make a constructive proof that the 2-point basis and the 5-point basis are equivalent. (Hint: make algorithms $\mathcal{C}$ and $\mathcal{D}$ to go between the two bases)

Introduction to Abstract Interpretation

Christian Clausen, 20081015

September 4, 2014

1. Operational Semantics

$$
\begin{aligned}
var \in Var = \{&\texttt{X}, \texttt{Y}, \texttt{Z}\} \\
instr ::= \ & \texttt{inc } var \\
\mid \ & \texttt{dec } var \\
\mid \ & \texttt{zero } var \texttt{ then } m \texttt{ else } n \\
\mid \ & \texttt{push } m \\
\mid \ & \texttt{pop } var \\
\mid \ & \texttt{stop}
\end{aligned}
$$

$$\frac{P_{pc} = \texttt{inc X}}{\langle X, Y, Z, pc, stack\rangle \to \langle X+1, Y, Z, pc+1, stack\rangle}$$

$$\frac{P_{pc} = \texttt{dec X}}{\langle X, Y, Z, pc, stack\rangle \to \langle X-1, Y, Z, pc+1, stack\rangle}$$

$$\frac{P_{pc} = \texttt{zero X then } m \texttt{ else } n \quad X = 0}{\langle X, Y, Z, pc, stack\rangle \to \langle X, Y, Z, m, stack\rangle}$$

$$\frac{P_{pc} = \texttt{zero X then } m \texttt{ else } n \quad X \neq 0}{\langle X, Y, Z, pc, stack\rangle \to \langle X, Y, Z, n, stack\rangle}$$

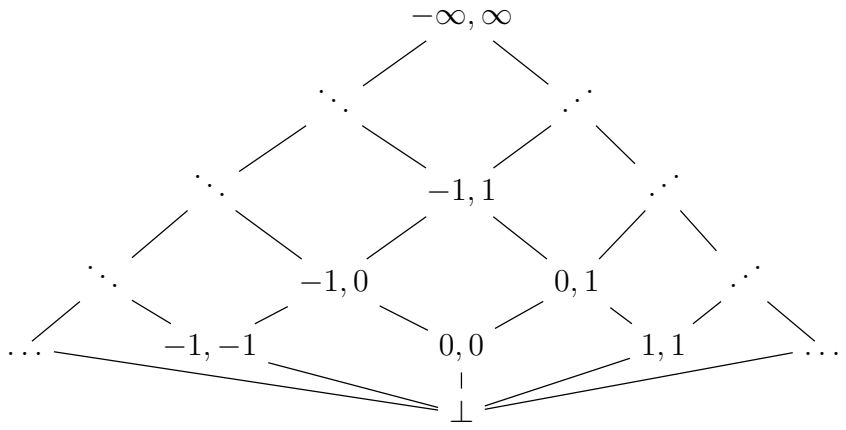
$$\frac{P_{pc} = \texttt{push X}}{\langle X, Y, Z, pc, stack\rangle \to \langle X, Y, Z, pc+1, X :: stack\rangle}$$

$$\frac{P_{pc} = \texttt{push } m}{\langle X, Y, Z, pc, stack\rangle \to \langle X, Y, Z, pc+1, m :: stack\rangle}$$

$$\frac{P_{pc} = \texttt{pop X}}{\langle X, Y, Z, pc, a :: stack'\rangle \to \langle a, Y, Z, pc+1, stack'\rangle}$$

2. Abstract Domain

$Interval :=$



2.1. Join and Meet

$$
\begin{aligned}
X \sqcup \bot &= X \\
\bot \sqcup Y &= Y \\
[a, b] \sqcup [c, d] &= [\min(a, c), \max(b, d)] \\
X \sqcap \bot &= \bot \\
\bot \sqcap Y &= \bot \\
[a, b] \sqcap [c, d] &= \begin{cases} [\max(a, c), \min(b, d)] & \text{if } \max(a, c) \leq \min(b, d) \\ \bot & \text{otherwise} \end{cases}
\end{aligned}
$$

3. Galois Connection

$$
\begin{aligned}
\wp(\mathbb{N} \times \mathbb{N} \times \mathbb{N} \times PC \times Stacks) &\leftrightarrows PC \to \wp(\mathbb{N} \times \mathbb{N} \times \mathbb{N} \times Stacks) \\
&\leftrightarrows PC \to \wp(Stacks) \\
&\underset{\alpha}{\overset{\gamma}{\leftrightarrows}} PC \to Interval
\end{aligned}
$$

where the last step is achieved by:

$$
\begin{aligned}
\alpha(\emptyset) &= \bot \\
\alpha(S) &= [\min(\text{len}(S)), \max(\text{len}(S))] \\
\gamma(\bot) &= \emptyset \\
\gamma([a, b]) &= \{stack \mid a \leq \text{len}(stack) \leq b\}
\end{aligned}
$$

4. Abstract Operators

$$
\begin{aligned}
-- &: \wp(Stacks) \to \wp(Stacks) \\
\texttt{=0} &= \lambda S.S \\
\texttt{<>0} &= \lambda S.S \\
\texttt{+1} &= \lambda S.S \\
\texttt{-1} &= \lambda S.S \\
\texttt{push} &= \lambda S.\{n :: s \mid s \in S \wedge n \in \mathbb{N}\} \\
\texttt{pop} &= \lambda S.\{s' \mid s \in S \wedge s = n :: s'\}
\end{aligned}
$$

The first four operators are identical thus we will treat them at once. We have:

$$
\begin{aligned}
\alpha \circ (\lambda S.S) \circ \gamma(\bot) &= \alpha(\emptyset) \\
&= \bot \\
\alpha \circ (\lambda S.S) \circ \gamma([a,b]) &= \alpha(\{s \mid a \leq \text{len}(s) \leq b\}) \\
&= [a,b]
\end{aligned}
$$

For `push`, we have:

$$
\begin{aligned}
\alpha \circ (\lambda S.\{n :: s \mid s \in S \wedge n \in \mathbb{N}\}) \circ \gamma(\bot) &= \alpha(\{n :: s \mid s \in \emptyset \wedge n \in \mathbb{N}\}) \\
&= \alpha(\emptyset) \\
&= \bot \\
\alpha \circ (\lambda S.\{n :: s \mid s \in S \wedge n \in \mathbb{N}\}) \circ \gamma([a,b]) &= \alpha(\{n :: s \mid s \in \{s \mid a \leq \text{len}(s) \leq b\} \wedge n \in \mathbb{N}\}) \\
&= \alpha(\{n :: s \mid a \leq \text{len}(s) \leq b \wedge n \in \mathbb{N}\}) \\
&= [1+a, 1+b]
\end{aligned}
$$

For `pop`, we have:

$$
\begin{aligned}
\alpha \circ (\lambda S.\{s' \mid s \in S \wedge s = n :: s'\}) \circ \gamma(\bot) &= \alpha(\{s' \mid s \in \emptyset \wedge s = n :: s'\}) \\
&= \alpha(\emptyset) \\
&= \bot \\
\alpha \circ (\lambda S.\{s' \mid s \in S \wedge s = n :: s'\}) \circ \gamma([a,b]) &= \alpha(\{s' \mid s \in \{s \mid a \leq \text{len}(s) \leq b\} \wedge s = n :: s'\}) \\
&= \alpha(\{s' \mid a \leq \text{len}(s) \leq b \wedge s = n :: s'\}) \\
&= [a-1, b-1]
\end{aligned}
$$

5. Widening and Narrowing

$$
\begin{aligned}
X \bigtriangledown \bot &= X \\
\bot \bigtriangledown Y &= Y \\
[a,b] \bigtriangledown [c,d] &= \left[\begin{cases} -\infty & c < a \\ a & a \geq c \end{cases}, \begin{cases} \infty & d > b \\ b & d \leq b \end{cases} \right]
\end{aligned}
$$

$$
\begin{aligned}
X \bigtriangleup \bot &= \bot \\
\bot \bigtriangleup Y &= \bot \\
[a,b] \bigtriangleup [c,d] &= \left[\begin{cases} c & a = -\infty \\ a & \text{otherwise} \end{cases}, \begin{cases} d & b = \infty \\ b & \text{otherwise} \end{cases} \right]
\end{aligned}
$$

6. Abstract Transfer Function

$$
\begin{aligned}
T^{\#}(S) := {} & [0 \mapsto [0,0]] \\
& \cup \left(\bigcup_{\substack{pc \in dom(S) \\ P_{pc} = \texttt{push } m}} [pc+1 \mapsto push(S(pc))] \right) \\
& \cup \left(\bigcup_{\substack{pc \in dom(S) \\ P_{pc} = \texttt{pop } var}} [pc+1 \mapsto pop(S(pc))] \right) \\
& \cup \left(\bigcup_{\substack{pc \in dom(S) \\ P_{pc} = \texttt{zero } var,\ pc',\ pc''}} [pc' \mapsto S(pc)] \cup [pc'' \mapsto S(pc)] \right) \\
& \cup \left(\bigcup_{\substack{pc \in dom(S) \\ P_{pc} = \texttt{inc } var \vee P_{pc} = \texttt{dec } var}} [pc+1 \mapsto S(pc)] \right)
\end{aligned}
$$

where $f \cup g = \lambda pc. f(pc) \sqcup g(pc)$ is the pointwise join.

Coq Keywords for Propositional and Predicate Logic

Frederik Brinck Truelsen

This is an introduction to the keywords used in Coq for propositional, predicate, and classical logic.

1. Keywords

The table below shows every logical connective and their corresponding keyword in Coq for introduction or elimination. While introducing and eliminating, you work in your goals and hypotheses respectively.

Symbol	Name	Introduction	Elimination
$\top$	triviality	exact I	*
$\bot$	absurdity	*	contradiction H
$\wedge$	conjunction	split	destruct H as $[H_1 \quad H_2]$
$\vee$	disjunction	left/right	destruct H as $[H_1 \mid H_2]$
$\longrightarrow$	implication	intro H	specialize $(H_1\ H_2)$
$\forall$	for all	intro x	specialize $(H\ x)$
$\exists$	exists	exists x	destruct H as $[x\ H]$

* these do not make sense.

2. Special cases

2.1. Negation

Negation was left out of the table above as it is handled differently, by using the Coq keyword `unfold not`. This inlines the definition not, thus it becomes the unnegated proposition implies False because is syntactic sugar. This can be written:

$$\neg A \cong A \to \bot$$

```
Lemma A_impl_not_not_A :
  forall A : Prop,
    A -> ~~A.
Proof.
  intro A.
  intro H_A.
  unfold not.
  intro H_A_impl_False.
  specialize (H_A_impl_False H_A)
  contradiction H_A_impl_False.
Qed.
```

2.2. Bi-implication

Logical bi-implication is handled in the same fashion as negation and it can be reduced as two implications with a conjunction between them. This can be written:

$$A \leftrightarrow B \cong (A \to B) \land (B \to A)$$

```
Lemma conj_comm :
  forall A B : Prop,
    A /\ B <-> B /\ A.
Proof.
  intro A.
  intro B.
  unfold iff.
  split.
    intro H.
    destruct H as [H_A H_B].
    split.
      exact H_B.
    exact H_A.
  intro H.
  destruct H as [H_B H_A].
```

```
  split.
    exact H_A.
 exact H_B.
Qed.
```

2.3. Classsic

Coq gives you the opportunity to work within classical logic by adding the hypothesis `classic`.

```
Hypothesis classic : forall A : Prop, A \/ ~A.
```

By destructing `(classic A)` as shown in the example below, we get two subgoals: first with `A` as an extra assumption, followed by a subgoal with `~A` as the extra assumption.

```
Lemma not_not :
  forall A : Prop,
    A <-> ~~A.
Proof.
  intro A.
  unfold not.
  unfold iff.
  split.
    intro H_A.
    intro H_A_impl_False.
    specialize (H_A_impl_False H_A)
    contradiction H_A_impl_False.
  intro H.
  destruct (classic A) as [Ht | Hf].
    exact Ht.
  specialize (H Hf).
  contradiction H.
Qed.
```

3. Futher reading

- C. Clausen, *Logic Exercises in* Coq, Spare-Time Teaching, `http://sparetimeteaching.dk/download.php?eventId=58`

Program Transformations

Christian Clausen

September 12, 2014

Program transformations can be used for many things such as optimizations, refactorings, adding language features, enabling static analyses, or as a tool for research. In this paper, we will use transformations to demonstrate the latter.

As this paper is about program transformations, we want to show both some transformations and some programs. So, as much as often as possible, we will present both the math-y way of writing it and the implementation. We will present most examples in OCaml, but there are some limitations with OCaml, so we may do the occasional example in Scheme.

We use a functional language to implement our transformations, and the transformations work over a similar language, therefore it is very important throughout the paper to keep track of what is in the source language (implementation language) and what is in the target language (the language we are transforming).

1. Starting point

We start our journey with something familiar: the Lambda Calculus:

$$t ::= x \mid \lambda x.\ t \mid t\ t$$

```
type term =
  | Var of string
  | Abs of string * term
  | App of term * term
```

and of course we also have an interpreter. The interpreter presented here is a denotational interpreter which makes it a horrible debugging tool, therefore we recommend that the reader also have a regular interpreter standing by.

$$\begin{aligned} [\![x]\!]_\rho &= \rho(x) \\ [\![\lambda x.\ t]\!]_\rho &= \texttt{VLam}(\lambda x'.\ [\![t]\!]_{\rho[x \mapsto x']}) \\ [\![t_1\ t_2]\!]_\rho &= f\ [\![t2]\!]_\rho \\ &\qquad \text{if } \texttt{VLam} f = [\![t_1]\!]_\rho \end{aligned}$$

```
type value = VLam of (value -> value)
let rec eval t rho =
  match t with
    | Var x ->
      assoc x rho
    | Abs (x, t) ->
      VLam (fun x' -> eval t ((x, x') :: rho))
    | App (t1, t2) ->
      match eval t1 rho with
        | VLam f ->
          f (eval t2 rho)
```

2. First transformation

Now that we have a basis and a target language it is time to play. We start out with a pretty simple transformation so we can get a feeling for what it is all about: We will transform Call by Value (CBV) to Call by Name (CBN). The key idea behind this transformation is that interpreters tend to (and should) stop recursing when they reach a λ, thus we can *delay* a computation if we put a lambda around it. When we have put lambdas around everything, we need to figure out what we want to *force*. In the end, the straightforward way of doing this looks like:

$$\begin{aligned}
[\![x]\!] &= \lambda().\ x \\
[\![\lambda x.\ t]\!] &= \lambda().\ \lambda x.\ [\![t]\!]\ () \\
[\![t_1\ t_2]\!] &= \lambda().\ [\![t_1]\!]\ ()\ [\![t_2]\!]
\end{aligned}$$

```
let rec cbn t =
  match t with
    | Var s ->
      Abs ("()", Var s)
    | Abs (s, t) ->
      Abs ("()", Abs (s, App (cbn t, Var "()")))
    | App (t1, t2) ->
      Abs ("()", App (App (cbn t1, Var "()"), cbn t2))
```

You can drastically improve the readability of the result if you use $(\lambda().\ t)\ () = t$, and it will still be CBN. In this case, we call () the *force token* as it will force the computation, but will never be used.

Now, we are equipped to make one of the most common program transformations: Continuation-Passing Style (CPS) transformation. The key idea here is to pretend the transformations itself is CPS transformed, but instead of passing it a function, we actually pass it the syntax of a continuation. Thus remember that k in the following is

itself a *term* in the target language.

$$\begin{aligned}
[\![x]\!]k &= k\ x \\
[\![\lambda x.\ t]\!]k &= k\ (\lambda x.\ \lambda k'.\ [\![t]\!]k') \\
[\![t_1\ t_2]\!]k &= [\![t_1]\!](\lambda r_1.\ [\![t_2]\!](\lambda r_2.\ r_1\ r_2\ k))
\end{aligned}$$

```
let rec cps t k =
  match t with
    | Var x ->
      App (k, Var x)
    | Abs (x, t) ->
      App (k, Abs (x, Abs ("k", cps t (Var "k"))))
    | App (t1, t2) ->
      cps t1 (Abs ("r1",
      cps t2 (Abs ("r2",
      App (App (Var "r1", Var "r2"), k)))))
```

Of course we could make k in to an actual continuation, that way we get a much smaller and neater output. The algorithm would then look like:

$$\begin{aligned}
[\![x]\!]k &= k\ x \\
[\![\lambda x.\ t]\!]k &= k\ (\lambda x.\ [\![t]\!](\lambda r.\ r)) \\
[\![t_1\ t_2]\!]k &= [\![t_1]\!](\lambda r_1.\ [\![t_2]\!](\lambda r_2.\ r_1\ r_2\ (\lambda r.\ k\ r)))
\end{aligned}$$

The implementation is left as an exercise for the reader.

3. Context switch

Now we are going to switch it up a little as we are going to be looking at some SCHEME. Taking about contexts, we can think of the context as: its syntax (think: itself), and an environment with bindings of all the free variables (think: its surroundings). This is called a *Closure* and it leads us to the key idea of Closure Conversion which removes all free variables. The idea is to make every abstraction into a pair of the abstraction and the environment. Notice that for this transformation, we use multiple parameters

instead of one at a time. The straightforward way of doing this is:

$$\begin{aligned} mk\_env(\cdot) &= \texttt{env} \\ mk\_env(v :: vs) &= (\texttt{v}, v) :: mk\_env(vs) \end{aligned}$$

$$\begin{aligned} [\![x]\!]_\rho &= \rho(x) \\ &\quad \text{if } x \in \rho \\ [\![x]\!]_\rho &= \pi_2^2(\texttt{env}(x)) \\ [\![\lambda x.\ t]\!]_\rho &= (\lambda \texttt{env}\ x.\ [\![t]\!]_x; mk\_env(\rho)) \\ [\![t_1\ t_2]\!]_\rho &= t_1'\ \texttt{env}\ [\![t_2]\!]_\rho \\ &\quad \text{if } (t_1'; \texttt{env}) = [\![t_1]\!]_\rho \end{aligned}$$

```
(define (mk_env vs)
  (if (null? vs)
      '$env
      `(cons (cons ',(car vs) ,(car vs)) ,(mk_env (cdr vs)))))

(define (cc t rho)
  (cond [(and (var? t) (member (cadr t) rho))
         (cadr t)]
        [(var? t)
         `(cdr (assoc ',(cadr t) $env))]
        [(abs? t)
         `(cons
           (lambda ,(cons '$env (cadr t)) ,(cc (caddr t) (cadr t)))
           ,(mk_env rho))]
        [(app? t)
         `(let ((closure ,(cc (cadr t) rho)))
            ((car closure) (cdr closure) ,(cc (caddr t) rho)))]
        [else (error 'cc "Not supported")]))
```

4. Trying it out

Now that we have discussed a few transformations, it is time for an example. Let's first try Closure Conversion on the interpreter from earlier:

```
type closure = Clo of string * term * (string * closure) list
let rec eval t rho =
  match t with
    | Var x ->
      assoc x rho
    | Abs (x, t) ->
```

```
      Clo (x, t, rho)
    | App (t1, t2) ->
      match eval t1 rho with
        | Clo (x, t, rho') ->
          eval t ((x, eval t2 rho) :: rho')
```

After that success, we should try out CPS transforming this new interpreter:

```
let rec eval t rho k =
  match t with
    | Var x ->
      k (assoc x rho)
    | Abs (x, t) ->
      k (Clo (x, t, rho))
    | App (t1, t2) ->
      eval t1 rho (fun r1 ->
      eval t2 rho (fun r2 ->
        match r1 with
          | Clo (x, t, rho') ->
            eval t ((x, r2) :: rho') k))
```

The problem is that we get confused by all the higher-order functions, we want a way to eliminate them. The answer here is *Defunctionalization*. The idea is similar to Closure Conversion, we find all the places where higher-order functions are used and then we tag them with a data type that saves the relevant information. Then we make the top-level function `apply` that will read the tag and execute what was in the function. After Defunctionalizing, the interpreter looks like this:

```
type func =
  | Arg of term * (string * closure) list * func
  | Fun of closure * func
let rec apply f v =
  match f with
    | Arg (t2, rho, k) ->
      eval t2 rho (Fun (v, k))
    | Fun (r1, k) ->
      match r1 with
        | Clo (x, t, rho) ->
          eval t ((x, v) :: rho) k
and eval t rho k =
  match t with
    | Var x ->
      apply k (assoc x rho)
    | Abs (x, t) ->
      apply k (Clo (x, t, rho))
    | App (t1, t2) ->
```

```
eval t1 rho (Arg (t2, rho, k))
```

But now, if we try to write this in math, it is actually a transistion system also known as an abstract machine (notice the difference between 2- and 3-tuples):

$$\begin{aligned}
\langle x, \rho, k\rangle &\rightarrow_e \langle k, \rho(x)\rangle \\
\langle \lambda x.\ t, \rho, k\rangle &\rightarrow_e \langle k, \texttt{Clo}(x, t, \rho)\rangle \\
\langle t_1\ t_2, \rho, k\rangle &\rightarrow_e \langle t_1, \rho, \texttt{Arg}(t_2, \rho, k)\rangle \\
\langle \texttt{Arg}(t_2, \rho, k), v\rangle &\rightarrow_a \langle t_2, \rho, \texttt{Fun}(v, k)\rangle \\
\langle \texttt{Fun}(\texttt{Clo}(x, t, \rho), k), v\rangle &\rightarrow_a \langle t, \rho[x \mapsto v], k\rangle
\end{aligned}$$

Mind blown...

5. Conclusion

As promised we have shown a way of using common transformations for science. We have shown a connection between denotational semantics and abstract machines, meaning that if we want to make an abstract machine with some complex feature, we can just extend the denotational interpreter with that feature then apply Closure Conversion, CPS transformation, and Defunctionalization to obtain a free abstract machine.

We have also show a transformation from CBV to CBN and implemented a lot of the transformations to demonstrate how easily they translate from math to code.

6. Further reading

Mads Sig Ager, Olivier Danvy, Jan Midtgaard did most of this and much more in their paper *A Functional Correspondence between Monadic Evaluators and Abstract Machines for Languages with Computational Effects*, it's amazing, go read it... Now... What are you still doing here?

7. Exercises

1. ♡ Implement a regular big-step interpreter.
2. Modify `cbn` with the η-reduction mentioned: $(\lambda().\ t)\ ()$.
3. Implement the modified version of the CPS transformation.
4. ♡ It is possible to implement Closure Conversion on the Lambda Calculus if you implement the lambda terms for the functions we generate (`cons`, `car`, `cdr`). Do so.
5. Implement CPS transformation in Scheme.
6. Implement CBV→CBN transformation in Scheme.

A Functional Fever Fantasy

Richard Möhn

The following is based on an exercise posed in the 2014 course Machine Learning I had at Technische Universität Berlin. The original author is Pieter-Jan Kindermans and he kindly allowed me to use it. The exercise itself is far too long and requires too many prerequisites to fit the spirit of Spare-Time Teaching. Therefore I am presenting it here along with a solution, which will make your hair stand on end. Promised. There is potential for many other solutions and who knows what effects those might have. In my presentation of the problem I try to follow the guidelines from George Pólya's *How To Solve It*, which I recommend everyone read.

1. Original Exercise

Last year for Christmas, we received a robotic vacuum-cleaner that was programmed badly. Each day we switch it on in the living room. Today, seven minutes after turning it on, we found the vacuum cleaner in the office. Unlike most vacuum cleaners, ours is rather fast and spends only exactly one minute in each room before moving on to cleaning the next one (rather poorly unfortunately).

From almost a year of intense observation of robot-cleaning activities, we have learned that when the robot is in the living room, one minute later it will be in the kitchen with probability 0.5, in the office with probability 0.3, and in the bedroom with probability 0.2. If the robot is in the office, then it will get stuck under the desk for a minute with probability 0.1; with probability 0.662, it will end up in the kitchen. With a probability of just 0.008, it is rather unlikely that the robot goes from the office to the bedroom, which should come as no surprise since it has to climb a series of stairs to get to the bedroom. Finally, with probability 0.23, it will decide to clean the living room next. If the robot is in the bedroom, a minute later it will be in the living room with probability 0.3, with probability 0.6 it will have fallen down the stairs and into the office. Only in 10 % of our observations, we found the robot in the kitchen the minute after cleaning the bedroom. Surprisingly, after spending a minute in the kitchen, the robot will always move to a different room and each room is equally likely.

As said before, today we switched on the robot in the living room and found it in the office after seven minutes. What was the most likely sequence of rooms that our robot visited in between?

2. Explanation and Programming Challenge

- What is the unknown? The most likely sequence of rooms that the robot went through. We can model this as a series of states:

$$S_0 \to S_1 \to S_2 \to S_3 \to S_4 \to S_5 \to S_6 \to S_7$$

- What are the data? A, the conditional probabilities of the state transitions and B, the first and the last state, as formalized in the following three items.

- Since we switch on the robot in the living room and find it in the office after seven minutes (seven state transitions), the first and the last state are known: $S_0 = \texttt{living room}, S_7 = \texttt{office}$.

- The other states are random experiments, that have one of the outcomes `living room`, `office`, `kitchen`, `desk`, or `bedroom` each. I will abbreviate the state names to their uppercased first letter.

- The probability that the robot transitions from state $S_i = s_m$ to $S_{i+1} = s_n$ is given by the exercise text above for every possible pair of states (s_m, s_n). We formalize it as the probability of $S_{i+1} = s_n$ given $S_i = s_m$ as $P(S_{i+1} = s_n | S_i = s_m)$.

- The exercise doesn't state it explicitly, but we assume that the probability of transitioning to some state is only dependent on the state immediately preceding it. Any states before that don't matter and therefore also not the position in the chain of states. Therefore we can shorten $P(S_{i+1} = s_n | S_i = s_m)$ to $P(s_n | s_m)$.

- An example:

$$P(S_3 = \mathsf{O} | S_2 = \mathsf{K}) = P(\mathsf{O}|\mathsf{K}) = 0.662$$

- What is the condition linking the unknown to the data? We can express the unknown together with part B of the data (first and last state) thus:

$$\arg\max_{s_1,\ldots,s_6} P(S_1 = s_1, \ldots, S_6 = s_6 | S_0 = \mathsf{L}, S_7 = \mathsf{O})$$

  Now, there is some part in this term that we don't know how to deal with, namely the joint probability of the unknown states. But did we use all the data? No, the conditional probabilities are nowhere to be found in that term. What follows is a series of transformations which will bring it into a form where part A of the data can be used.

$$\arg\max_{s_1,\ldots,s_6} P(S_1 = s_1, \ldots, S_6 = s_6) | S_0 = \mathsf{L}, S_7 = \mathsf{O})$$

  (definition of conditional probability)

$$= \arg\max_{s_1,\ldots,s_6} \frac{P(S_0 = \mathsf{K}, S_1 = s_1, \ldots, S_6 = s_6, S_7 = \mathsf{O})}{P(S_0 = \mathsf{K}, S_7 = \mathsf{O})}$$

(definition of conditional probability)

$$= \arg\max_{s_1,\ldots,s_6} \frac{P(S_0 = \mathsf{K}, S_1 = s_1, \ldots, S_6 = s_6, S_7 = \mathsf{0})}{P(S_0 = \mathsf{K}|S_7 = \mathsf{0})P(S_7 = \mathsf{0})}$$

($P(S_0 = \mathsf{K}) = 1$, because we started the robot there)

$$= \arg\max_{s_1,\ldots,s_6} \frac{P(S_0 = \mathsf{K}, S_1 = s_1, \ldots, S_6 = s_6, S_7 = \mathsf{0})}{P(S_7 = \mathsf{0})}$$

($P(S_7 = \mathsf{0})$ constant in $s_1, \ldots, s_6$, calculate with law of total probability)

$$= \arg\max_{s_1,\ldots,s_6} P(S_0 = \mathsf{K}, S_1 = s_1, \ldots, S_6 = s_6, S_7 = \mathsf{0})$$

(definition of conditional probability)

$$= \arg\max_{s_1,\ldots,s_6} P(S_7 = \mathsf{0}|S_0 = \mathsf{K}, S_1 = s_1, \ldots, S_6 = s_6)P(S_0 = \mathsf{K}, \ldots)$$

(by assumption one state only dependent on the state immediately before it)

$$= \arg\max_{s_1,\ldots,s_6} P(S_7 = \mathsf{0}|S_6 = s_6)P(S_0 = \mathsf{K}, S_1 = s_1, \ldots, S_6 = s_6)$$

(repeating the last two steps, $P(S_0 = \mathsf{K}) = 1$)

$$= \arg\max_{s_1,\ldots,s_6} P(S_7 = \mathsf{0}|S_6 = s_6)P(S_6 = s_6|S_5 = s_5) \cdots P(S_1 = s_1|S_0 = \mathsf{K})$$

(dropping the big Ss, because now it's clear what follows what)

$$= \arg\max_{s_1,\ldots,s_6} P(\mathsf{0}|s_6)P(s_6|s_5) \cdots P(s_1|\mathsf{K})$$

Stated like this, we can get at our unknown solely by using the data: the conditional probabilities and the first and last state.

- Until now, everything is fairly rigorous. (Although, not in the sense of Coq, of course.) But the formula as it is can't be implemented efficiently. We'd have to test all variations of states. However, we see that none of the leftmost five terms depends on s_1, none of the leftmost four terms depends on s_2, and so on. Therefore we get handwavy, say that we can pull in the maximizations and abuse the notation for clarifying this somewhat (use your PDF viewer's zoom feature on this):

$$\arg\left(\max_{s_6}\left(P(\mathsf{0}|s_6)\max_{s_5}\left(P(s_6|s_5)\max_{s_4}\left(P(s_5|s_4)\max_{s_3}\left(P(s_4|s_3)\max_{s_2}\left(P(s_3|s_2)\max_{s_1}(P(s_2|s_1)P(s_1|\mathsf{K}))\right)\right)\right)\right)\right)$$

What this means is that the maxes do their usual thing, finding the largest possible value of the function they were passed, but also remember the argument that made the function produce that largest possible value. Then the outside arg will pull those arguments out of the maxes' memories and return them as a tuple of states.

3. Implementation

Translating this to a computer program is fairly straightforward. Below you see an implementation in Clojure. You can also download it from `https://gist.github.com/rmoehn/2bf119af3e55664d42c7`.

```
(ns user)

(def trans-probs {:living-room
                  {:living-room 0
                   :office 0.3
                   :kitchen 0.5
                   :desk 0
                   :bedroom 0.2}
                  :office
                  {:living-room 0.23
                   :office 0
                   :kitchen 0.662
                   :desk 0.1
                   :bedroom 0.008}
                  :kitchen
                  {:living-room 1/3
                   :office 1/3
                   :kitchen 0
                   :desk 0
                   :bedroom 1/3}
                  :desk
                  {:living-room 0
                   :office 1
                   :kitchen 0
                   :desk 0
                   :bedroom 0}
                  :bedroom
                  {:living-room 0.3
                   :office 0.6
                   :kitchen 0.1
                   :desk 0
                   :bedroom 0}})

(defn as-p [trans-probs]
  (fn [event _ given]
    (get-in trans-probs [given event])))

(defn foldn [e f n]
```

```
  (if (zero? n)
    e
    (recur (f e) f (dec n))))

(defn max-prob-chain [trans-probs init-state fin-state in-betw-cnt]
  (let [p (as-p trans-probs)
        possible-states (keys trans-probs)]
    ((foldn (fn [next-state] [(p next-state :| init-state) []])
            (fn [ml-states-before]
              (fn [next-state]
                (apply (partial max-key first)
                       (map (fn [s]
                              (let [[prob ml-states] (ml-states-before s)]
                                [(* (p next-state :| s) prob)
                                 (conj ml-states s)]))
                            possible-states))))
            in-betw-cnt)
     fin-state)))

(second (max-prob-chain trans-probs :living-room :office 6))
;=> [:office :kitchen :bedroom :office :kitchen :bedroom]
```

You thought I would explain this? No, no, no, I never said anything like that. I mean, I don't understand it myself. But it's got something to do with Hobbits for sure. And it is not efficient in spite of that handwavy pulling out maxes stuff. However, I do have a feeling that it could easily be made linear in the number of states by dropping in some memoization somewhere...

Logic Programming

Mathias Vorreiter Pedersen

September 19, 2014

These notes will introduce the basics of logic programming using the PROLOG programming language.

1. Datatypes

PROLOG has one datatype called a *term*. A term is either

1. An *atom*.
2. A *number*, which can either be an integer or a floating point number.
3. A *variable*.
4. A *compound term*.

In the subset of PROLOG that we will be using, an atom is a sequence of symbols starting with a lower-case letter or the empty list, which is written as [ ].

A variable is any sequence of symbols that start with an upper-case letter like A, B1, MyVariable, etc.

Finally, a compound term (or complex term) is a term of the form $\mathbf{f}(\text{arg}_1, \text{arg}_2, \ldots, \text{arg}_N)$, where $\mathbf{f}$ is an atom (usually called the *functor*) and $\text{arg}_1, \text{arg}_2, \ldots, \text{arg}_N$ are terms (usually called *arguments*). A non-empty list $[\text{arg}_1, \text{arg}_2, \ldots, \text{arg}_N]$ is also a compound term.

2. Syntax

A PROLOG program is a list of Horn clauses ending with a dot. A horn clause is either a fact that is a compound term or a rule of the form

`head` ($\texttt{arg}_1$, $\texttt{arg}_2$, ..., `arg_N`) :- $\texttt{fact}_1$, $\texttt{fact}_2$, ..., `fact_N`.

where `head` is an atom and $\texttt{arg}_1$, $\texttt{arg}_2$, ..., `arg_N`, $\texttt{fact}_2$, ..., `fact_N` are terms.

We say that the list of horn clauses defines a *knowledge base* on which we can perform queries. An example of a knowledge base is

```
append([], L, L).
append([H | T], L, [H | L2]) :- append(T, L, L2).

reverse([], L, L).
reverse([H | T], L, R) :- reverse(T, [H | L], R).

reverse(L, R) :- reverse(L, [], R).
```

3. Unification

Unification is the process of finding variable bindings such that a goal is satisfied. Unification can be viewed as a recursive procedure on two terms. Two terms `t1` and `t2` unify iff:

1. $\texttt{t}_1$ is an atom with symbol $\texttt{a}_1$, $\texttt{t}_2$ is an atom with symbol $\texttt{a}_2$ and $\texttt{a}_1 = \texttt{a}_2$.
2. $\texttt{t}_1$ is an unbound variable and $\texttt{t}_2$ is a term. The same is true if $\texttt{t}_2$ is an unbound variable and $\texttt{t}_1$ is a term.
3. $\texttt{t}_1$ is a bound variable, $\texttt{t}_2$ is a term, and the value bound to $\texttt{t}_1$ unifies with $\texttt{t}_2$. The same is true if $\texttt{t}_2$ is an unbound variable and $\texttt{t}_1$ is a term.
4. $\texttt{t}_1$ is a compound term with functor $\texttt{f}_1$ and term list $\texttt{terms}_1$, and $\texttt{t}_2$ is a compound term with functor $\texttt{f}_1$ and term list $\texttt{terms}_2$, and $\texttt{f}_1 = \texttt{f}_2$ and each pair $(\texttt{t}_1, \texttt{t}_2)$ in $\texttt{zip}(\texttt{terms}_1, \texttt{terms}_2)$ unifies. Note that this requires that $|\ \texttt{terms}_1\ | = |\ \texttt{terms}_2\ |$.

Unification in PROLOG is achieved using the `=/2` goal. An example follows:

```
1 ?- a = a.
true.

2 ?- X = a.
X = a.

3 ?- f(X, g(Y, a)) = f(h(a), g(Z, a)).
X = h(a),
Y = Z.
```

Or the last goal without syntactic sugar

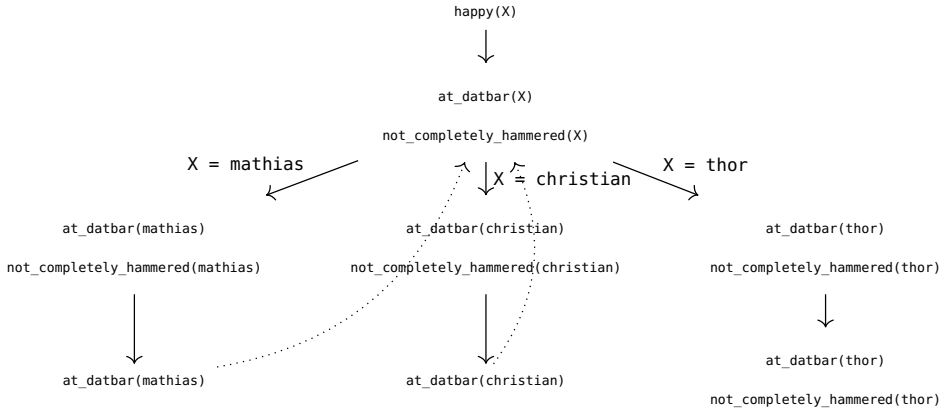
```
1 ?- =(f(X, g(Y, a)), f(h(a), g(Z, a))).
X = h(a),
Y = Z.
```

4. Backtracking

Backtracking is the process of "undoing" choices made during unification. An example follows:

```
happy(X):-
    at_datbar(X),
    not_completely_hammered(X).

at_datbar(mathias).
at_datbar(christian).

not_completely_hammered(thor).
at_datbar(thor).
```

This causes PROLOG to derive the following proof tree. First, we unify the variable X with the atom "mathias". This choice fails since we can't prove `not_completely_hammered(mathias)`. This causes PROLOG to backtrack to the point where it chose the latest binding of X. By repeating this process two more times, it succeeds with X = "thor."



This process is confirmed by SWI-PROLOG, which performs the following operations, corresponding exactly to the proof tree:

```
[trace] 1 ?- happy(X).
   Call: (6) happy(_G2852) ? creep
   Call: (7) at_datbar(_G2852) ? creep
   Exit: (7) at_datbar(mathias) ? creep
   Call: (7) not_completely_hammered(mathias) ? creep
   Fail: (7) not_completely_hammered(mathias) ? creep
   Redo: (7) at_datbar(_G2852) ? creep
   Exit: (7) at_datbar(christian) ? creep
   Call: (7) not_completely_hammered(christian) ? creep
   Fail: (7) not_completely_hammered(christian) ? creep
```

```
   Redo: (7) at_datbar(_G2852) ? creep
   Exit: (7) at_datbar(thor) ? creep
   Call: (7) not_completely_hammered(thor) ? creep
   Exit: (7) not_completely_hammered(thor) ? creep
   Exit: (6) happy(thor) ? creep
X = thor.
```

5. Cuts

Sometimes we (as logic programmers) know more than what PROLOG is able to infer. For instance, when there is only one correct binding of variables that cause a clause to succeed. Take, for instance, the following implementation of the max function.

```
max(A, B, B) :- B >= A.
max(A, B, A) :- A > B.
```

which is certainly correct, but it's inefficient. This can be seen in the context of the following knowledge base.

```
f(X) :- max(X, 20, X), p(X).
p(31).
```

That is, f(X) is true iff X is greater than or equal to 20, and p(X) is true. By tracing `f(20)`, we see

```
[trace] 1 ?- f(20).
   Call: (6) f(20) ? creep
   Call: (7) max(20, 20, 20) ? creep
   Call: (8) 20>=20 ? creep
   Exit: (8) 20>=20 ? creep
   Exit: (7) max(20, 20, 20) ? creep
   Call: (7) p(20) ? creep
   Fail: (7) p(20) ? creep
   Redo: (7) max(20, 20, 20) ? creep
   Call: (8) 20>20 ? creep
   Fail: (8) 20>20 ? creep
   Fail: (7) max(20, 20, 20) ? creep
   Fail: (6) f(20) ? creep
false.
```

Notice the `Redo`; that's PROLOG trying to find another way of satisfying `max(X, 20, X)`. by trying the other clause `A > B`. But we know that this cannot be satisfied since `B >= A` already succeeded!

Thus, we need a way to tell PROLOG that it should not try other clauses once the first succeeds. The `!/0` (cut) goal does exactly that. It always succeeds and it tells PROLOG to never try alternatives for the current goal. By rewriting max as

```
max(A, B, B) :- B >= A, !.
max(A, B, A) :- A > B.
```

and by tracing `f(20)` again, we see

```
[trace] 1 ?- f(20).
   Call: (6) f(20) ? creep
   Call: (7) max(20, 20, 20) ? creep
   Call: (8) 20>=20 ? creep
   Exit: (8) 20>=20 ? creep
   Exit: (7) max(20, 20, 20) ? creep
   Call: (7) p(20) ? creep
   Fail: (7) p(20) ? creep
   Fail: (6) f(20) ? creep
false.
```

which is a large performance gain!

6. Negation as failure

Let's say Vincent likes burgers.

```
likes(vincent,X) :- burger(X)
```

That is, vincent likes X iff X is a burger. But what if Vincent really dislikes Big Kahuna burgers? We need to specify that vincent likes burgers that are not from Big Kahuna Burgers. Let's use our new cut goal can help us here, combined with **`fail/0`**, which always fails the current goal.

```
likes(vincent,X) :- big_kahuna_burger(X),!,fail.
likes(vincent,X) :- burger(X)
```

Since PROLOG examines the rules from top to bottom, we always hit the rule containing `big_kahuna_burger(X)` first. In that case, we prevent PROLOG from trying any other choices for X and force it to fail with this choice. In other words, PROLOG will always fail `likes(vincent,X)` if `big_kahuna_burger(X)` is satisfied. Perfect!

Let's encapsulate this pattern in a rule called neg (for negation duh!)

```
neg(Goal) :- Goal, !, fail.
neg(Goal).
```

and we can now write `likes/2` as

```
likes(vincent,X) :- burger(X), neg(big_kahuna_burger(X)).
```

7. Exercises

1. How would you represent the following facts and rules in PROLOG?
   - Aarhus University is located in the city "Aarhus".
   - There is a Spare Time Teaching event if it's Friday.
   - "Mathias" will eat anything, as long as it does not contain gluten.

2. Write the predicate `my_length/2` such that if `my_length(L, N)` is `true`, then `L` is a list of length `N`. The following demonstrates how the predicate is supposed to be used

```
1 ?- my_length([], 0).
true.

2 ?- my_length([1, 3, 3, 7], s(s(s(s(0))))).
true.

3 ?- my_length([4, 2], s(0)).
false.

4 ?- my_length([1, 2, 3, 4, 5, 6, 7, 8, 9], X).
X = s(s(s(s(s(s(s(s(s(0)))))))))).
```

3. Write a predicate `map/3` such that `map(P, X, Y)` is `true` if `Y` is the list obtained by applying the predicate to each of the elements in `X` (Hint: Look up `call/3`). For instance:

```
1 ?- map(=, [1, 2, 3, 4], [1, 2, 3, 4]).
true .

2 ?- map(succ, [3, 1, 4, 1, 5, 9], X).
[4, 2, 5, 2, 6, 10] .

3 ?- map(my_length, [[], [1], [1, 2], [1, 2, 3]], X).
[0, s(0), s(s(0)), s(s(s(0)))] .
```

4. Write a predicate `foldr/4` such that `foldr(P, BC, L, R)` is true if `R` is the result of folding the predicate `P` over the list `L` using `BC` as the base case, i.e., when `L` is the empty list. An example of a use-case follows:

```
1 ?- foldr(append, [], [[1, 2, 3], [4, 5, 6], [9]], X).
X = [1, 2, 3, 4, 5, 6, 9] .
```

```
2 ?- foldr(plus, 0, [], X).
X = 0 .

3 ?- foldr(plus, 0, [1, 2, 3, 4, 5], 15).
true .
```

(Bonus: How would you write `map/3` using `foldr/4`?)

5. Solve the famous "Zebra Puzzle" using PROLOG. You're given the following information:

   - There are five houses.
   - The Englishman lives in the red house.
   - The Spaniard owns the dog.
   - Coffee is drunk in the green house.
   - The Ukrainian drinks tea.
   - The green house is immediately to the right of the ivory house.
   - The Old Gold smoker owns snails.
   - Kools are smoked in the yellow house.
   - Milk is drunk in the middle house.
   - The Norwegian lives in the first house.
   - The man who smokes Chesterfields lives in the house next to the man with the fox.
   - Kools are smoked in a house next to the house where the horse is kept.
   - The Lucky Strike smoker drinks orange juice.
   - The Japanese smokes Parliaments.
   - The Norwegian lives next to the blue house.

   And now the question is: Who owns the zebra?

Program Obfuscation

Christian Clausen

September 26, 2014

We denote programs with p, the corresponding obfuscated program p', and interpreting a program p with input d as $[\![p]\!](d)$. Further, we assume that we have a:

- self-interpreter Self satisfying:

$$[\![p]\!](d) = [\![\textsc{Self}]\!](d, p)$$

- partial evaluator (or specializer) Spec satisfying:

$$[\![p]\!](s, d) = [\![[\![\textsc{Spec}]\!](p, s)]\!](d)$$

  for static data s and dynamic data d.

1. Some basic requirements

Obfuscation has three requirements:

- It should be semantic preserving, that is: $\forall d, [\![p]\!](d) = [\![p']\!](d)$.
- p' should be hard to understand by humans but easy to understand by computers (efficient).
- Mostly be one way.

Our first attempt follows immediately from the definitions above:

$$\begin{aligned}[\![p]\!](d) &= [\![\textsc{Self}]\!](p, d) \\ &= [\![[\![\textsc{Spec}]\!](\textsc{Self}, p)]\!](d)\end{aligned}$$

Which leads us to the definition:

$$p' := [\![\textsc{Spec}]\!](\textsc{Self}, p)$$

We make the following observations about this transformation:

- it is clearly semantics preserving.
- p' inherits the algorithm of p.
- p' inherits the style of Self.

2. The self-interpreter

We now look at Self and consider which parts of it are static and which parts are dynamic. First, we notice that `p` is static and `d` is dynamic. Since `e`, `pc`, and `instr` only depend on `p`, we can declare them static. We can unfold the **`while`** and `eval` making them static as well. The variables used in `p` are also known, thus the domain of `store` is also static. We now have (dynamic data in red):

```
input p, d;
pc := 2;
store := {in -> d, out -> 0, x1 -> 0, ...};
while(pc < length(p)){
  instr := lookup(p, pc);
  switch(instr){
  skip   : pc := pc + 1;
  x := e : store[x] := eval(e, store); pc := pc + 1;
  ...
  }
}
output store[out];
eval(e, store){
  switch(e){
  const   : e;
  var     : store[e];
  e1 + e2 : eval(e1, store) + eval(e2, store);
  e1 * e2 : eval(e1, store) * eval(e2, store);
  ...
  }
}
```

After all these transformations, we have an optimal compiler:

$$p =_\alpha [\![\text{Spec}]\!](\text{Self}, p)$$

However, this is not great for obfuscation.

3. Obfuscations

In the following, we will use this example program as p:

```
input x;
y := 2;
while(x > 0){
  y := y + 2;
  x := x - 1;
}
```

```
output y;
```

Our first obfuscation is a store obfuscation; instead of storing variable X, we store $2 \cdot X$. We then have to define two functions $obf(x) = 2 \cdot x$ and $dob(x) = x/2$. With this obfuscation, p' will be:

```
input x;
x := 2 * x;
y := 2 * 2;
while(x/2 > 0){
  y := 2 * (y/2 + 2);
  x := 2 * (x/2 - 1);
}
output y/2;
```

A more complicated store obfuscation is based on weaving the values together in pairs; $obf(x, y) = (x + y, x - y)$, $dob(u, v) = ((u + v)/2, (u - v)/2)$. This will result in:

```
input x;
u := x + y;
v := x - y;
u := (u + v)/2 + 2;
v := (u - v)/2 - 2;
while((u + v)/2 > 0){
  u := u + 2; v := v - 2;
  u := u - 1; v := v + 1;
}
output (u - v)/2;
```

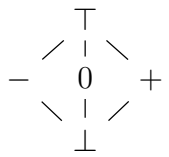
However, if we get a bit more serious. We think about what a good program is. Neil Jones once said:

> Good programs are well-structured and have concise invariants.

This quote gives the idea to obfuscate the control flow. We can do this by marking `pc` as dynamic in Spec so that the resulting program will always only have one loop.

```
input x;
pc := 2;
while(pc < 6){
  switch(pc){
  2 : y := 2; pc := 3;
  3 : if(x > 0) { pc := 4 } else { pc := 6 }
  4 : y := y + 2; pc := 5;
  5 : x := x - 1; pc := 3;
  }
}
output y;
```

Another observation that we make is that the attacker of our program is often a static analyzer. Since most static analyses are sound, they are not complete, so we can instead try to force our program into the slack. For instance, if we say say the attacker is a sign analysis, we have:



$\cdot$	-	0	+
-	+	0	-
0	0	0	0
+	-	0	+

+	-	0	+
-	-	-	$\top$
0	-	0	+
+	$\top$	+	+

So, we can implement $\cdot$ with $+$. Consider this p:

```
input x;
y := 2;
while(x > 0){
  y := y * y;
  x := x - 1;
}
output y;
```

Which will become:

```
input x;
y := 2;
while(x > 0){
  y := y * (y - 1) + y;
  x := x - 1;
}
output y;
```

4. Futamura projections

- $p' = [\![\textsc{Spec}]\!](\textsc{Self}, p)$ gives an obfuscated program.
- $comp = [\![\textsc{Spec}]\!](\textsc{Spec}, \textsc{Self})$ gives an obfuscator.
- $cogen = [\![\textsc{Spec}]\!](\textsc{Spec}, \textsc{Spec})$ gives a generator for obfuscators.

Emacs Guide

Kent Grigo

1. Introduction

This guide gives a quick overview of the hotkeys and functions that the authors use in Emacs. The guide will cover general movements, nifty tricks for language-specific functions, and ways of remembering the many hotkeys.

2. Notation

`C-y` means hold down `ctrl` (or `command` on Mac) and press y. `C-` is read "control." `M-y` means hold down `alt` (or `option` on Mac) and press y. `M-` is read "meta." In dire situations, `Esc` can be used to toggle the meta key.

Code, Emacs functions, and their arguments are highlighted by a gray background. Square brackets mean variables, e.g., `C-[number]` can be `C-5`. When possible, the letters that are used in the hotkeys are bold-faced in the description as a way of remembering the hotkeys.

Prefix keys are used to combine multiple hotkeys.[1] For instance, `C-x C-f ~/.emacs` means do `C-x`, then `C-f`, and pass `~/.emacs` as argument.[2] The most used prefix keys are `C-x`, `C-c`, and `C-[number]`. `C-x` is the prefix related to files, windows, and buffers, e.g., saving by `C-x C-s`. Whereas `C-c` is the prefix related to the current mode, e.g., process next command in Coq by `C-c C-n`. `C-[number]` can be used in two ways: (1) it repeats the following action, e.g., `C-1 C-0 )` makes 10 `)`; and (2) it modifies a function, e.g., `C-3 C-k`[3] will `kill` three lines instead of one.

[1] This has lead to the XKCD joke on keybindings in Emacs: `http://xkcd.com/378/`

[2] Modifications for Emacs are done in this file. The changes are made with programs written in Elisp.

[3] As described in section 4.

3. Initial annoyances

Emacs can be difficult because many standard hotkeys for other applications are defined as something else in Emacs.

Hotkeys	Common binding	Emacs binding
`C-z`	undo	minimizes the window
`C-x`	cut	the prefix key for file-, buffer-, and window-related functions
`C-c`	copy	the prefix key for mode-related functions
`C-v`	paste	page down
`C-s`	save	search

In case of accidentally pressing a wrong hotkey and finding yourself in an unknown and unwanted situation, the panic button `C-g` will do the trick. For instance, if you start a sequence of commands, `C-g` will cancel them. This hotkey also deactivates functions such as a search.

4. Most used, general hotkeys

Hotkeys	Description
`C-x C-s`	**s**aves the file
`C-x C-c`	**c**loses Emacs
`C-j`	makes a line break and indents
`C-s [query]`	searches forwards; used by pressing `C-s` again to search further
`C-r [query]`	searches backwards; used by pressing `C-r` again to search further
`C-w`	cut, **w**ank, or `kill-region`
`M-w`	copy, **w**ank, or `kill-ring-save`
`C-y`	paste, `yank`
`C-k`	cuts line from cursor and onwards, `kill`
`C-d`	**d**eletes character in front of the cursor
`M-d`	**d**eletes word in front of the cursor
`C-_`	undo
`M-/`	auto-complete
`C-g`	panic button; cancels prefix keys or initiated functions
`C-space`	sets mark and starts selection
`M-x [function]`	executes the given function
`C-l`	cycles between three views: show cursor at (1) center, (2) top, or (3) bottom
`M-;`	toggles between outcommenting region; by no selection, makes comment at the end of the line.

In Emacs, "mark" refers to a saved position in the buffer. The position is saved whenever a function is initiated or when `C-space` is pressed. One can jump to a mark by pressing `C-x C-x`. Some functions like `kill-region`, `C-w`, do not necessarily require a selection but selects the region between the latest mark and the cursor.

5. Movement

Notice that `M-` can be thought of as a stronger version of `C-` in most cases.

Hotkeys	Move -
`C-f`	a character **f**orward
`M-f`	a word **f**orward
`C-M-f`	a statement **f**orward
`C-b`	a character **b**ackward
`M-b`	a word **b**ackward
`C-M-b`	a statement **b**ackward
`C-p`	to the **p**revious line, i.e., one line up
`C-n`	to the **n**ext line, i.e., one line down
`C-e`	to the **e**nd of the line
`M-e`	to the **e**nd of the sentence
`C-a`	to the st**a**rt of the line
`M-a`	to the st**a**rt of the sentence
`M-{`	to the start of the section
`M-}`	to the end of the section
`C-<`	to the start of the file
`C->`	to the end of the file
`C-v`	a page down
`M-v`	a page up
`C-x C-x`	to mark
`M-g g [line]`	to specified line
`M-g M-g [line]`	to specified line

Emacs gives a way of going to a specific line by both `M-g g` and `M-g M-g`. We recommend you to rebind `M-g g` as `goto-char` by inserting `(global-set-key (kbd "M-g g") 'goto-char)` into your .emacs file. Jumping to a specific character will be useful when working with OCaml because its compiler gives error messages based on character numbers.

6. Less used, but nice to know

Hotkeys	Description
`C-x h`	selects entire buffer
`C-t`	**t**ransposes the two characters around the cursor
`M-t`	**t**ransposes the two words around the cursor
`C-M-t`	**t**ransposes the two statements around the cursor
`C-x C-t`	**t**ransposes the line at cursor with the line above
`M-l`	changes word to all lowercase
`M-u`	changes word to all uppercase
`M-c`	changes word to be capitalized
`C-x C-+`	increases font size
`C-x C-−`	decreases font size

7. LaTeX

Hotkeys	Description
`C-c C-e [environment]`	creates the given environment and prompts for optional arguments
`C-c C-f C-t`	creates `\texttt{}`
`C-c C-f C-b`	creates `\textbf{}`
`C-c C-f C-i`	creates `\textit{}`
`C-c C-f C-e`	creates `\emph{}`

Note: if you have already written some text and want to make it bold, you can mark the text and then do `C-c C-f C-b`, the text will then be wrapped in `\textbf{}`.

8. Coq

Hotkeys	Description
`C-c C-n`	processes **n**ext command
`C-c C-RET`	processes buffer up to cursor
`C-c C-b`	processes entire **b**uffer
`C-c C-a C-i`	**i**ntroduces **a**ll universal quantifiers with `intros`, even when there is only one quantifier
`C-c .`	processes buffer up to cursor whenever `.` is typed; this functionality is called "electric terminator"

9. Acknowledgements

We are grateful to Amanda T. L. Sandegaard, Christian Clausen, Lasse B. Kristensen, and Thor Bagge for commenting on preliminary versions of this guide.

Introduction to Induction

Mikkel Kringelbach

October 3, 2014

1. Induction Proof

To start of with, let's take a look at the standard example of proving something by induction. We will prove the following statement about the sum of first n natural numbers:

$$0 + 1 + \cdots + n = \frac{n(n+1)}{2}$$

Proof.

Base case ($n = 0$):

$$0 = \frac{0 \cdot (0+1)}{2}$$

Induction case:

We assume that the statement holds for the first k natural numbers and have to show that it works for the $k+1$ natural number, i.e., we assume that this is true: $0 + 1 + \cdots + k = \frac{k(k+1)}{2}$.

$$\begin{aligned}
0 + 1 + 2 + \cdots + k + (k+1) &= \frac{k(k+1)}{2} + (k+1) && \text{Induction Hypothesis} \\
&= \frac{k(k+1)}{2} + \frac{2(k+1)}{2} && \text{Extending fraction} \\
&= \frac{k(k+1) + 2(k+1)}{2} && \text{Combining fraction} \\
&= \frac{(k+1)(k+2)}{2} && \text{Moving } (k+1) \text{ out} \\
&= \frac{(k+1)((k+1)+1)}{2} && \text{Moving the plus}
\end{aligned}$$

□

Now, the induction proof is done, and it would be accepted if it was handed in.

To get an idea of why this works, I will use the induction case to construct the proofs for some small numbers. So, if we wanted to see that the property holds for $n = 1, 2, 3, 4$, we could do the following:

$$\begin{aligned}
0+1 &= \frac{0(0+1)}{2}+1 = \frac{0(1)}{2}+\frac{2(1)}{2} = \frac{0(1)+2(1)}{2} = \frac{(1)(0+2)}{2} = \frac{(1)(1+1)}{2} \\
0+1+2 &= \frac{1(1+1)}{2}+2 = \frac{1(2)}{2}+\frac{2(2)}{2} = \frac{1(2)+2(2)}{2} = \frac{(2)(1+2)}{2} = \frac{(2)(2+1)}{2} \\
0+1+2+3 &= \frac{2(2+1)}{2}+3 = \frac{2(3)}{2}+\frac{2(3)}{2} = \frac{2(3)+2(3)}{2} = \frac{(3)(2+2)}{2} = \frac{(3)(3+1)}{2} \\
0+1+2+3+4 &= \frac{3(3+1)}{2}+4 = \frac{3(4)}{2}+\frac{2(4)}{2} = \frac{3(4)+2(4)}{2} = \frac{(4)(3+2)}{2} = \frac{(4)(4+1)}{2} \\
&\vdots
\end{aligned}$$

We see that the steps that we performed in the induction case can be used to keep generating the proof for larger and larger numbers. This is why we believe that just by showing the two steps; the base case and the induction step, we can show that something is true for all the natural numbers.

In order to better understand what induction is, we need to write it down more formally.

2. Formal Induction Proof

In order to work more with induction, we need to define induction more formally. But, before we can define induction over natural numbers, we have to define the natural numbers. We typically define the natural numbers as follows:

```
nat ::= 0
      | Succ nat
```

That means a natural number is either 0 or the successor of a natural number. For example, three would be represented as `Succ (Succ (Succ 0))`.

Now, we are able to write down the induction rule for a proposition P over natural numbers:

$$\frac{P(0) \quad \forall k.P(k) \Rightarrow P(\texttt{Succ}\ k)}{\forall n.P(n)} Ind_{nat}.$$

This rule is saying that (1) if I can show that $P(0)$ is true, and that (2) if I assume $P(k)$ is true then I can show that $P(\texttt{Succ}\ k)$ is true, then I may conclude that $P(n)$ is true for all natural numbers.

In our previous example, we would have had:

$$P(n) := 0 + \texttt{Succ}\ 0 + \ldots + \text{n} = \frac{n(\texttt{Succ}\ n)}{2}$$

And the two cases we proved were exactly, $P(0)$ and $P(k) \Rightarrow P(\texttt{Succ}\ k)$

2.1. Example - Plus

Now, in order to do a proof using our formal induction, we also need some formal definitions over natural numbers to work with. The most natural thing to do is addition:

$$add(n1, n2) = \begin{cases} n2 & \text{if } n1 = 0 \\ \texttt{Succ}\ \ (add(n1', n2)) & \text{if } n1 = \texttt{Succ}\ \ n1' \end{cases}$$

This captures our normal understanding of how addition works and will correctly compute the sum of natural numbers.

Now, let's try and prove that 0 is neutral for addition, meaning that for all natural numbers n: $add(0, n) = n$ and $add(n, 0) = 0$.

The proof that 0 is the neutral element on the left follows directly prof the definition of *add*:

Proof.

$$add(0, n) = n \qquad \text{Definition of } add$$

□

When we try to add 0 from the right, we have a problem though and need to use induction to get the proof to go through:

Proof.

Base case:

$$add(0, 0) = 0 \qquad \text{Definition of } add$$

Induction step:

We assume that it hold for k, i.e., $add(k, 0) = k$ and want to show that it holds for Succ k:

$$\begin{aligned} add(\texttt{Succ}\ \ k, 0) &= \texttt{Succ}\ \ add(k, 0) && \text{Definition of } add \\ &= \texttt{Succ}\ \ k && \text{Using the assumption for } k \end{aligned}$$

□

Exercises - Induction of Natural Numbers

See Exercises 1-5.

3. Induction in Computer Science

Induction would not be very helpful if we were only able to do it over the natural numbers. As computer scientist, we do use numbers but we also make use of lots of other data types. One can generalize the notion of induction from the previous section to what is called *structural induction*. Structural induction works over any structure, and means that we will be able to prove properties for almost all of the data structures we use in Computer Science. In the following, there are a couple of examples of how the induction principle looks for different data types, and for each a comment on why it looks like that.

3.1. Lists

Lists are one of the most basic structures in computer science, they are defined by the following grammar. Similar to natural numbers they have an empty constructor, for the natural numbers it was 0 for lists it is [] the empty list. Now, when we wanted to construct the next natural number, we would apply `Succ` to the previous number, we can't do this for lists as we would like the list to contain elements. Therefore, when we extend the list, by using ::, we have to give it the element that we would like to put into the list.

Datatype

```
nat_list ::= []
           | nat :: nat_list
```

Now, the induction principle will look similar to the one for natural numbers but we have to consider how we handle the extra number we attach in front of the list. The *base case* will be similar to before: we have to show that the statement holds for the empty list. The *inductive step* will be a bit different: we still assume that it holds for a list, ns', and have to show that it works for an extension of that list, $n :: ns'$. Since we don't know what n will be, we have to show that it works for any possible n.

Induction Principle

$$\frac{P([]) \quad \forall n, ns'.P(ns') \to P(n :: ns')}{\forall ns.P(ns)} Ind_{nat\_list}.$$

Example - Append

To demonstrate the induction principle for this, we present append Append takes two lists and constructs a new list, where the elements of the new list are the elements of the first list followed by the elements of the second. So, let's define *append*:

$$append(ns1, ns2) = \begin{cases} ns2 & \text{if } ns1 = [] \\ n :: (add(ns1', ns2)) & \text{if } ns1 = n :: ns1' \end{cases}$$

The property we will prove for append is that [] is the neutral element, i.e., $append(ns, []) = ns$ and $append([], ns) = ns$.

The proof that [] is the neutral element on the left follows directly from the definition of *append*:

Proof.

$$append([], ns) = ns \qquad \text{Definition of } append$$

□

When we try to append [] from the right, we have a problem though. Like for *add*, we need to use induction to get the proof to go through:

Proof.

Base case:

$$append([], []) = [] \qquad \text{Definition of } append$$

Induction step:
We assume that it hold for ns', i.e., $append(ns', []) = ns'$ and want to show that $append(n :: ns', []) = n :: ns'$ for any n:

$$\begin{aligned} append(n :: ns', []) &= n :: append(ns', []) && \text{Definition of } append \\ &= n :: ns' && \text{Using the assumption for } ns' \end{aligned}$$

□

Exercises - Induction on Lists

See Exercises 6-7.

3.2. Booleans

Could we also consider a notion of induction on booleans? The answer is yes, booleans might not be recursively defined, but we can still state an induction principle:

Datatype

```
bool ::= true
       | false
```

Induction Principle

$$\frac{P(true) \quad P(false)}{\forall b.P(b)} Bool.Ind.$$

This captures the normal idea of how we would prove that something is correct for all booleans, we would simply show that it holds for *true* and *false*.

3.3. Trees

A structure that is also commonly used is trees, and just like before, we can also define induction over trees:

Datatype

```
nat_tree ::= Leaf nat
           | Node nat_tree nat_tree
```

Induction Principle

$$\frac{\forall n.P(\text{Leaf } n) \quad \forall nt1, nt2.P(nt1) \Rightarrow P(nt2) \Rightarrow P(\text{Node } nt1\ nt2)}{\forall nt.P(nt)} Ind_{Tree}.$$

Example - Mirror

We define the mirror of a tree as follows:

$$mirror(nt) = \begin{cases} \text{Leaf } n & \text{if } nt = \text{Leaf } n \\ \text{Node } mirror(nt2)\ \ mirror(nt1) & \text{if } nt = \text{Node } nt1\ \ nt2 \end{cases}$$

Exercises - Induction on Trees

See Exercises 8-9.

More complicated structures

Induction might still not seem as useful, but as our understanding grows, we can start applying induction to more complicated examples, like regular expressions. We can start showing equality of different structures within computer science, in particular regular expressions and finite automata. What is interesting is that we can also do induction of relations that are recursively defined, this becomes extraordinarily useful when we develop complicated type systems and we want to prove properties about them.

Exercises - More complicated types of Induction

See Exercises 10-11.

4. Programming and Induction

In this section, we will take a look at how one would code some of the functions described earlier, in particular we will define *add*, and *mult*:

```
let rec add n1 n2 =
  match n1 with
    | 0 -> n2
    | S n1' -> Succ (add n1' n2)

let rec mult n1 n2 =
  match n1 with
    | 0 -> 0
    | S n1' -> add(n2, mult n1' n2)
```

We see that they have a similar structure, and as computer scientist, we would like to eliminate the redundancy, therefore we define this function to help us more clearly express what makes us add and mult:

```
let rec traverse_nat b f n =
  match n with
    | 0 -> base
    | S n' -> f n' (traverse_nat b f n')

let add n1 n2 =
    traverse_nat n2 (fun n' H_n' -> Succ H_n') n1

let mult n1 n2 =
    traverse_nat 0 (fun n' H_n' -> add n2 H_n') n1
```

We see that the two parameters we give traverse_nat, is exactly describing the same as our definitions in the beginning of this note. Now, it seems that this function traverse_nat is useful to define functions over natural numbers. Traverse_nat has the following type

$$\texttt{traverse\_nat}: \quad \alpha \to (nat \to \alpha \to \alpha) \to nat \to \alpha$$

I will expand a bit upon this notation to show in which cases the α are used:

$$\texttt{traverse\_nat}: \quad \alpha(0) \to (\texttt{k} : nat \to \alpha(\texttt{k}) \to \alpha(\texttt{Succ k})) \to \texttt{n} : nat \to \alpha(n)$$

Does this look familiar? Let's see what the type of Ind_{nat} is, when next to this one:

$$\begin{array}{rlll} \texttt{traverse\_nat}: & \alpha(0) \to (\texttt{k} : nat \to & \alpha(\texttt{k}) \to \alpha(\texttt{Succ k})) & \to \texttt{n} : nat \to \quad \alpha(n) \\ Ind_{nat}: & P(0) \to (\forall k. & P(\texttt{k}) \Rightarrow P(\texttt{Succ } k)) & \Rightarrow \forall n. \quad P(n) \end{array}$$

We see that these types are the same! This is absolutely astonishing, and quite beautiful. This connection between our induction principle and this simple traversal function, which we created by seeing the similarities between the *add* and *mult* functions.

We can do something similar for lists, and if we do we get the following function:

```
let rec traverse_list b f xs =
  match xs with
    | [] -> base
    | x :: xs' -> f x (traverse_list b f xs')
```

One might be able to recognize this function, as it is the typical `fold_right` function for lists. In other words, `fold_right` is the function corresponding to induction over lists.

The most impressive thing is that one can go directly from the `fold_right` function directly to the data type for lists or the induction principle. If you are more interested in this, I will recommend showing up to an Introduction to Lambda Calculus talk, and if you are still interested try and study up on the Curry-Howard Isomorphism.

Exercises - Programming and Induction

See Exercises 12-14. Thanks to Christian Clausen for providing some of these examples.

5. Exercises

1. Show that add is commutative, i.e., $add(n_1, n_2) = add(n_2, n_1)$.
2. Show that add is associative, i.e., $add(n_1, add(n_2, n_3)) = add(add(n_1, n_2), n_3)$.
3. Define multiplication in a similar way as addition and prove that 1 is the neutral element.
4. Show that multiplication is commutative and associative.
5. Define exponentiation on natural numbers.
6. Show that *append* is associative.
7. Define a function that returns the length of a list, $length(l) = ?$.
8. Show that $mirror(mirror(t)) = t$ using induction over trees.
9. Define the function $number\_of\_leafs$ to count the number of leafs in a tree.
10. (Assumes knowledge of regular expressions)
    Write out the induction principle for regular expressions.
11. (Assumes completion of the Regularity and Automata course at Aarhus University)

    Look at the proof of Kleene's second theorem, we can construct a finite automata with the same language as any regular expression. Try writing out all of the different hypotheses and consider what P is in this case.

12. Consider the following data type:

```
color := Red
       | Green
       | Blue
       | Grey nat
       | Transparent color
```

What is the Inductive principle, the traverse function, and the type of the traverse function?

13. Consider the following Induction principle:

$$\frac{\forall x', y' : P(x') \to P(y') \to P(\mathrm{Foo}(x', y')) \quad \forall t, x' : P(x') \to P(\mathrm{Bar}(t, x')) \quad \forall t : P(\mathrm{Baz}(t))}{\forall x : P(x)}$$

What is the data type corresponding to it? The fold? and the type of the fold?

14. If the fold of a datatype has the OCaml type:

```
fold : (nat -> 'a) -> (X -> X -> 'a -> 'a -> 'a) -> X -> 'a
```

then what is the datatype?

A Rapper Wrapper

Nicolaj Græsholt <projects@ohmyunix.com>
OhMyUnix Institute of Technology

In this paper, we'd like to introduce the notion of a *Rapper Wrapper*, present rigorous research, and provide an actual implementation of a *Rapper Wrapper*. We follow up with thoughts and preliminary research of the idea of a *Wrapper Rapper*. The paper ends with a *conclusion* that discusses the implementation and even argues that it's possible to construct the *Rapper Wrapper* for all Rappers. Finally, *further reading* contains resources about clever ways of using wrapper functions as well as some great places to obtain street knowledge about hip-hop.

1. Research

A wrapper function is a subroutine in a software library or a computer program whose main purpose is to call a second subroutine[1] or a system call with little or no additional computation.

Rapping is spoken or chanted rhymed lyrics[2].

2. Implementation

The implementation is done in a pseudo-language in order to make it non-platform specific.

[1] Reselman, Bob; Peasley, Richard; Pruchniak, Wayne (1998). Using Visual Basic 6. Que. p. 446. ISBN 9780789716330

[2] `http://dictionary.reference.com/browse/Rapping`, 2014-08-25

2.1. A Rapper function

```
function rapper(){
    print "Ice, ice baby.";*
    wait 2600;**
    print "Ice, ice baby.";
    wait 2600;
}
```

* Depending on whether the desired rapper is Vanilla ice.
** Depending on the BPM of the actual song; results may vary on different equipment.

2.2. A Rapper Wrapper function

```
function wrapper(){
    return rapper();
}
```

3. A Wrapper Rapper

On the subject of Rappers on Wrappers, we've been able to dig up a competition that the rapper called "Calvin Cordozar Broadus, Jr.", also known by his birth name "Snoop Doggy D.O. Double-G Zillaboy Lion", had created called "Wrapper for a Rapper"[3] in which he wants to procure the paramount of a blunt roller to join his entourage for a number of years.

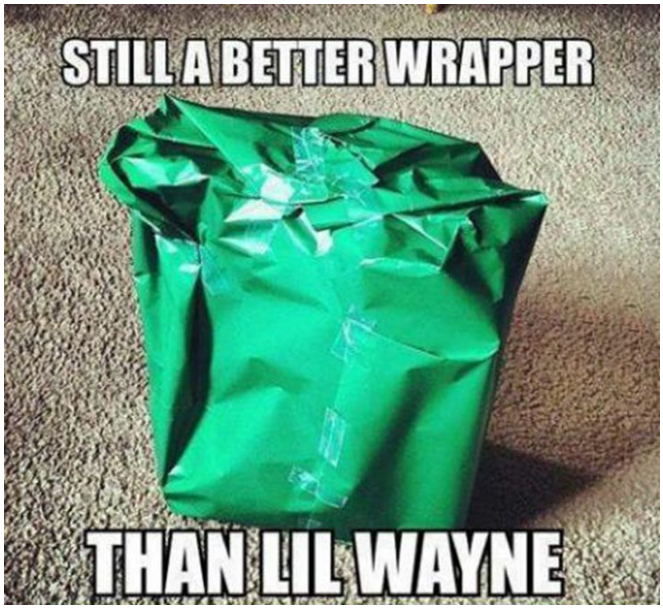


Figure 2: It's suggested that Lil' Wayne is not a good wrapper.

[3] http://newsexaminer.net/art/snoop-dogg-offers-100kyear-blunt-roller-join-entourage/, 2014-02-24

Another example of Wrapper Rappers is suggested by an unknown researcher of the internet society[4], in figure 2 wherein it's stated that Lil' Wayne should have taken a co-interest in wrapping. The statement is brought in a most up-to-date fashion as *black-bordered white text* of the *sans-serif Impact* typeface on a picture, often, relevant to the statement; the picture shows a poorly wrapped present, and it's made apparent that Lil' Wayne is at least a worse wrapper; the reader will notice that a sharp inequality is inferred in the statement.

4. Conclusion

It's possible to construct a *Rapper Wrapper* using only basic knowledge of programming and hip-hop. The code doesn't use any advanced features of a programming language, and is thus implementable in most. The lyrics are also trivial to construct as seen in this example. To satisfy the rhyming-constraint one is allowed to use the same lyric.

Furthermore, since we did not at any point make any specific assumptions about the Rapper, we've shown that it's possible to construct a *Rapper Wrapper* for all Rappers.

The notion of a *Wrapper Rapper* is a field much unstudied, and it opens up to an incredible amount of new thoughts and questions about the world we live in.

5. Futher reading

- Article: The Wolfram Alpha entry for Vanilla Ice.

  `http://www.wolframalpha.com/input/?i=Vanilla+Ice`

- Book: UNIX Network Programming - Using wrappers for testing purposes.

  Stevens, Richard; Fenner, Bill; Rudoff; Andrew M. (2003). Addison-Wesley. pp. 5–6,29. ISBN 9780131411555.

- Movie: Get Rich or Die Tryin' - A documentary starring 50 Cent.

  `http://www.imdb.com/title/tt0430308/`

[4] `http://lolsnaps.com/upload_pic/WrapperRapper-94964.jpg`

Lambda Calculus from Scratch

Christian Clausen

October 10, 2014

1. Grammar

```
t ::= v
    | λ v . t
    | t t
```

2. Reduction rules

2.1. Metarules

```
"⇝" means "syntax for"
x y z ⇝ (x y) z
λ x. t₁ t₂ ⇝ λx. (t₁ t₂)
λ x y . t ⇝ λ x . λ y . t
t[t'/x] is substitution
```

2.2. α Renaming

```
λ x . t  →α  λ y . t[y/x]
```

2.3. η Expansion

```
f  →η  λ x . f x
```

2.4. β Reduction

```
(λ x . t₁) t₂  →β  t₁[t₂/x]
```

3. The Identity Function

3.1. Function

```
id ≡ λ x . x
```

3.2. Evaluation

```
id id = (λ x . x) (λ x . x)  →α  (λ x . x) (λ y . y)
 →β  x[(λ y . y)/x] = λ y . y = id
```

```
id id = (λ x . x) id
 →β  x[id/x] = id
```

4. Some Terminology

Abstraction, normalization, free variable, name capture, combinator, weak-head normal form, closed terms, ...

5. Combinators

```
I ≡ λ x . x
K ≡ λ x y . x
B ≡ λ x y z . x (y z)
S ≡ λ f g x . f x (g x)
K* ≡ λ x y . y
```

```
K I  ↠  K*
K* I  ↠ I
S K I  ↠  I
S K K  ↠  I
S (K S) K  ↠  B

X = λ x . x S K
X X  ↠  I
X (X X)  ↠  S K
X (X (X X))  ↠  K
X (X (X (X X)))  ↠  S
```

6. Booleans

6.1. Definition

```
T ≡ λ t f . t
F ≡ λ t f . f
```

6.2. Not

```
not ≡ λ b . λ t f . b f t
not ≡ λ b . b (λ t f . f) (λ t f . t)
not ≡ λ b . b F T
¬a ⇝ not a
```

6.3. And

```
and ≡ λ b₁ b₂ . b₁ b₂ F
a & b ⇝ and a b
```

6.4. Exercise: Or

```
or ≡ λ b₁ b₂ . b₁ T b₂
a | b ⇝ and a b
```

7. Numbers

7.1. Definition

```
nat ::= 0
      | Succ nat
```

```
0 ≡ λ fₛ vᵤ . vᵤ
mk_succ ≡ λ n . λ fₛ vᵤ . fₛ (n fₛ vᵤ)
succ ≡ mk_succ
```

Church numerals

```
0 ≡ λ s z . z
1 ≡ λ s z . s z
2 ≡ λ s z . s (s z)
...
```

7.2. Is zero?

```
is_zero? ≡ λ n . n (λ α . F) T
```

7.3. Plus

```
plus ≡ λ m n . m succ n
a + b ⇝ plus a b
```

7.4. Times

```
times ≡ λ m n . m (plus n) 0
a * b ⇝ times a b
```

8. Pairs

8.1. Definition

```
pair ::= P * *
```

```
mk_pair ≡ λ a b . λ f_p . f_p a b
<a, b> ⇝ mk_pair a b
π_1 ≡ λ p . p (λ a b . a)
π_2 ≡ λ p . p (λ a b . b)
```

8.2. Swap

```
swap ≡ λ p . <π_2 p, π_1 p>
swap ≡ λ p . mk_pair (π_2 p) (π_1 p)
```

8.3. Sliding Window

$$\texttt{<a, b>} \xmapsto{f} \texttt{<b, a + b>}$$

```
fib ≡ λ n . π_1 (n f <0, 1>)
```

$$\texttt{<a, b>} \xmapsto{f} \texttt{<succ a, a * b>}$$

```
fac ≡ λ n . π2 (n f <1, 1>)
```

$$\texttt{<a, b>} \xmapsto{f} \texttt{<b, a>}$$

```
is_even? ≡ λ n . π1 (n f <T, F>)
```

9. Lists

9.1. Definition

```
list ::= Nil
       | Cons * list
```

```
mk_cons ≡ λ hd tl . λ fc vn . fc hd (tl fc vn)
hd :: tl ⇝ mk_cons hd tl
nil ≡ λ fc vn . vn
```

9.2. Sum

```
sum_list ≡ λ xs . xs plus 0
```

10. Trees

10.1. Definition

```
tree ::= Leaf *
       | Node tree tree
```

```
mk_node ≡ λ t1 t2 . λ fn fl . fn (t1 fn fl) (t2 fn fl)
mk_leaf ≡ λ n . λ fn fl . fl n
```

10.2. Sum

```
sum_tree ≡ λ t . t plus id
```

11. Unbounded Iteration

$\omega \equiv \lambda$ x . x x
$\Omega \equiv \omega\ \omega$

11.1. But Y

11.1.1. Curry

W_f = λx . f (x x)
$\mathsf{Y}_C \equiv \lambda$ f . W_f W_f

11.1.2. Turing

A $\equiv \lambda$ x y . y (x x y)
$\mathsf{Y}_T \equiv$ A A

11.2. Example: Streams

$\widehat{\mathbb{N}} \equiv \lambda$ self n . <n, self (succ n)>
$\mathbb{N} \equiv$ Y $\widehat{\mathbb{N}}$ 0

$\widehat{\text{zip}} \equiv \lambda$ self xst yst . <π_1 xst, self yst (π_2 xst)>
zip $\equiv$ Y $\widehat{\text{zip}}$

Parsing

Mathias Vorreiter Pedersen

1. Introduction

From day 1 in computer science, we've been told to love tree-like structures, but such a nice structure does not come for free.

Instead we're often just given a piece of text with a promise that this text satisfies some *grammar*, and it's up to us to unveil the structure of this input based on the grammar. These grammars are often specified using *Backus-Naur form*, which we will refer to as BNF. The following is an example of a BNF

```
<E> ::= E '+' T
      | T

<T> ::= T '*' F
      | F

<F> ::= num
      | '(' E ')'
```

This BNF specifies simple grammar for expressions containing addition, multiplication, and nested parenthesized expressions. The grammar is specified by *productions* saying, for instance, that we can replace a *nonterminal* E with the sequence E '+' T containing two nonterminals and a single *terminal* '+'.

Let's call this language `EXPR`. Now, given the string $s =$ "$1 + (2 * 3)$", how do we determine whether s is accepted by the grammar? And how do we produce the rich tree-like structure hidden in this input? Let's explore different ways of answering this question!

2. Recursive descent parsing

Recursive descent parsing is based on the simple notion that the productions can *almost* directly be translated into a set of functions, each of which handles a specific nonterminal. Thus for instance, one could (attempt) to recognized the `EXPR` as follows.

```
E() {
  if E() {
    expect('+')
    return T()
  }
  return T()
}
```

```
T() {
  if T() {
    expect('*')
    return F()
  }
  return F()
}
```

```
F() {
  if num() {
    return true
  }
  expect('(')
  if not E() {
    return false
  }
  expect(')')
  return true
}
```

Now that was easy! The three nonterminals translate almost perfectly into three functions E, T, and F, making the code very readable.

There is a fatal flaw in our program though: Since the grammar for EXPR is *left-recursive*[1], the code produced by our simple translation never terminates on any input! What a useless parser!

Let's try and rewrite the grammar such that we remove the left-recursion, but still recognize exactly the language EXPR.

2.1. Removing direct left-recursion

We'll "fix" our grammar by using the following algorithm:

Given any production of the form $A \to A\alpha_1 \mid \cdots \mid A\alpha_n \mid \beta_1 \mid \cdots \mid \beta_m$, we replace the production of A with the production $A \to \beta_1 A' \mid \cdots \mid \beta_m A'$ and create a new nonterminal A' with the production $A' \to \alpha_1 A' \mid \cdots \mid \alpha_n A' \mid \varepsilon$.

Applying this algorithm to the grammar of EXPR, we get the following

<E> ::= T E'

<E'> ::= '+' T E'
| ε

<T> ::= F T'

<T'> ::= '*' F T'
| ε

<F> ::= num
| '(' E ')'

Let's translate these productions into code

[1] We have (at least) one productions of the form $A \to A\alpha \mid \beta$, where α and β are sequences of nonterminals and terminals.

```
E() {
  if not T() {
    return false
  }
  return E'()
}

E'() {
  if not peek('+') {
    return true
  }
  consume()
  if not T() {
    return false
  }
  return E'()
}
```

```
T() {
  if not F() {
    return false
  }
  return T'()
}

T'() {
  if not peek('*') {
    return true
  }
  consume()
  if not F() {
    return false
  }
  return T'()
}
```

and F is unchanged since it's not left-recursive. We've thus removed the left-recursive grammar from our grammar, but without changing the language that we recognize. In fact, we can see that we've transformed the functions E and T into tail-recursive functions (and therefore these are effectively loops).

While we recognize the same language, we've changed an important property of the grammar: The addition and multiplication operators are no longer left-associative, but right-associative! This is illustrated in the parse trees below.

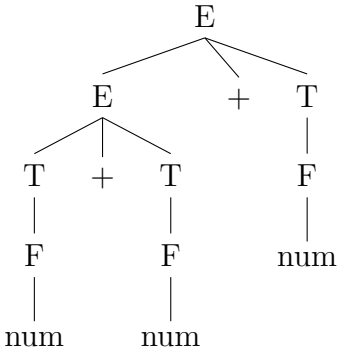


Figure 3: Parse tree of "1 + 2 + 3" *before* left-recursion removal

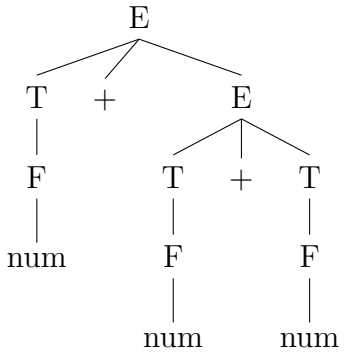


Figure 4: Parse tree of "1+2+3" *after* left-recursion removal

There is no apparent solution to this problem in the realm of recursive descent parsing. So, let's explore a very different parsing technique is widely used for real-world parsing

problems.

3. Table-based parsing

In table-based parsing, we precompute a "magic" table that will guide us in the parsing process. This magic table represents a deterministic finite automaton (DFA).

In addition to the table, we will have a pointer into the input string such that we can identify which token we're currently looking at. Lastly, we will have a stack containing both nonterminals and terminals.

There are many types of table-parsed parsing techniques, but we'll focus on one specific class: LR(1) parsing, which is shorthand for left-to-right parse, rightmost-derivation with 1-token of lookahead. The rightmost-derivation part might seem odd but, because this technique works bottom-up, we're essentially working left-to-right.

Before we discuss how to construct it, let's talk about what this table contains (that is, the states of the DFA).

Shifts, reduces and gotos

Each entry in the table will contain one of three *actions*. The `shift` (resp. `goto`) actions consists of a tag and a number $(\texttt{s}, n)$ (resp. $(\texttt{g}, n)$). The third type of entry is the `reduce` action $(\texttt{r}, A \to \alpha)$, consisting of the tag `r` and a production $A \to \alpha$ from the grammar.

- A `goto` action simply instructs the parser to go directly to the state specified by n.
- A `shift` action tells the parser to "shift" its pointer to the next token and push the token onto a separate stack. After having done this, we move to state n.
- Finally, the `reduce` action pops a number of tokens off the stack. Since this approach is bottom-up, these popped off tokens represents the right hand side of a production $A \to \alpha$. This sequence of tokens α on the stack is replaced by the corresponding nonterminal A that can derive α.

An example follows:

We'll use our original grammar for `EXPR`, and we add a start nonterminal S with the single production $S \to E\,\$$, where \$ is the end of file marker.

Let's say the stack looks like:

```
E + T *
```

and the input that has yet to be parsed looks like:

```
42 $
```

Assume that the table tells us to shift, we push 42 onto the stack, advance the input pointer, and move to the new state. Thus, our stack is now:

`E + T * 42`

and our queue is

`$`

Assuming that the new state tells us to reduce according to the production `F` → `num`, we pop one element off the stack and replace it with the nonterminal `F`. Our stack is now

`E + T * F`

and the input pointer has not changed.

Now, our table tells us to reduce according to the production `T` → `T` '∗' `F`. We thus replace `T` '∗' `F` with `T`. After the reduce step, our stack will look like

`E + T`

and we move to the new state. Assuming that this new state is another reduce that tells us to reduce according to `E` → `E` '+' `T`, we'll end up with the stack

`E`

and move to the new state as specified by the table entry. Let's assume that this is a shift, meaning that we shift $ onto the stack. Thus our input queue is empty and our stack is

`E $`

and we move to a new state. Let's say this state is a reduce step that tells us to reduce according to $S \to E\,\$$, thus we replace the two elements on the stack by the starting nonterminal S and the parse succeeds.

3.1. Constructing the table

We'll use the term *symbol* to denote both nonterminals and terminals. Intuitively the table is generated by identifying which symbols can follow a given sequence of symbols. In order to compute this information, we start by computing the `first` and `nullable` sets of the grammar.

- Given a symbol X, $\texttt{first}(X)$ is the set of terminals that can begin strings derived from X.
- Given a nonterminal X, $X \in \texttt{nullable}$ if X can derive the empty string ε.

The `first` set is the one we really care about, but we need `nullable` because we want to generalize the `first` set to handle a sequence of symbols. This is done by the equation

$$\texttt{first}(X\gamma) = \begin{cases} \texttt{first}(X) & X \notin \texttt{nullable} \\ \texttt{first}(X) \cup \texttt{first}(\gamma) & X \in \texttt{nullable} \end{cases}$$

The `first` and `nullable` sets are defined as the smallest sets for which the following properties hold

1. $\forall z \in \texttt{terminals} : \texttt{first}(z) = \{z\}$

2. $\forall A \to \alpha_1\alpha_2 \ldots \alpha_k \in \texttt{productions}$
   $(\forall i \in \{1, \ldots, k\} : \alpha_i \in \texttt{nullable}) \implies A \in \texttt{nullable}$
   $\forall i \in \{1, \ldots, k\} :$
   $(\forall j \in \{1, \ldots, \alpha_{i-1}\} : \alpha_j \in \texttt{nullable}) \implies \texttt{first}(\alpha_i) \subseteq \texttt{first}(A)$

(1) is simple: For each terminal z, the only symbol that can occur as the first symbol of the derivation of z is z itself.

(2) is almost as simple. Given a production $A \to \alpha_1\alpha_2 \ldots y_k$, the nonterminal A is nullable if each symbol on the left-hand side is nullable.

If only $\alpha_1\alpha_2 \ldots \alpha_n \in \texttt{nullable}$, where $n < k$, the set of first symbols that can be derived from x include the first symbols that can be derived from α_{n+1}, since all symbols preceding this can be ε.

This can easily be turned into a fixpoint algorithm that is guaranteed to terminate since there are only finitely many productions, with finitely many symbols.

The grammar for `EXPR` has very simple `first` and `nullable` sets, since no productions contains any ε-productions.

	`nullable`	`first`
E	No	{'num','('}
T	No	{'num','('}
F	No	{'num','('}
+	No	{'+'}
*	No	{'*'}
(	No	{'('}
)	No	{')'}
num	No	{ 'num' }

Having identified the `first` and `nullable` sets, the next step is to generate the DFA. We do this by generating the states and the transitions iteratively.

The states will be represented by sets of *items*, and the transitions between states is represented by *edges*. For this reason, we will use the terms "states" and "item sets" interchangeably.

We first describe what an items is. In LR(1) parsing, we have 1 token of lookahead[2], and thus each item will contain the following information

- A production $A \to \alpha_1\alpha_2 \ldots \alpha_k$ that we're currently following
- The right-hand side position n of the production, indicating how much of the right-hand side we've parsed so far.

[2]This concept of an item can be generalized to k tokens of lookahead, but results in an exponential increase in the number of states in the DFA. In practise, this is not a problem though, since most languages have an LR(1) grammar.

- A lookahead symbol z.

We write items as $(A \to \alpha_1\alpha_2 \ldots \alpha_k, n, z)$ or $(A \to \alpha.\beta, z)$ where $\lambda\alpha.\ = n$.

Correspondingly we have edges, which we write as $I \xrightarrow{X} J$, indicating that we transition from item set I to item set J when reading symbol X.

We first construct the item sets that's going to give us our `shift` and `goto` actions. This is done by the procedure `Goto`, which takes an item set I and a symbol X and advances the right-hand side of each production and constructs the set of possible items we can end up in.

```
Goto(I, X) =
  J ← ∅
  ∀(A → α . Xβ, z) ∈ I
    J ← J ∪ {(A → α X . β, z)}
  return Closure(J)
```

`Closure` is a procedure that, given an item set J, constructs the possible states that the DFA can end up in after one step.

```
Closure(I) =
  repeat
    ∀(A → α.Xβ, z) ∈ I
      ∀X → γ
        ∀w ∈ first(βz)
          I ← {(X → .γ, w)}
  until I does not change
  return I
```

That is, we find find the entire set of states reachable in one step starting at any state in I by iteratively including new states reachable by reading a terminal α.

We can now compute the entire set of states as follows:

- Find the states reachable in one step from the starting item $(S' \to .S\,\$)$ by computing
$$T = \texttt{Closure}(\{S' \to .\,S\$\})$$
- Repeat the following until E and T reaches a fix point.

  $\forall I \in T$
  $\quad\forall(A \to \alpha\,.\,X\beta, z) \in I$
  $\quad\quad J \leftarrow \texttt{Goto}(I, X)$
  $\quad\quad T \leftarrow T \cup \{J\}$
  $\quad\quad E \leftarrow E \cup \{I \xrightarrow{X} J\}$

Finally, we compute the set R of reduce actions simply by observing which items have a fully parsed right-hand side and collecting these:

$R \leftarrow \emptyset$
$\forall I \in T$
  $\forall (A \to \alpha \,.\, z) \in I$
    $R \leftarrow R \cup \{(I, z, A \to \alpha)\}$

We now have two important sets at our disposal: E containing the set of all `shift` and `goto` actions, and R containing the set of all `reduce` actions. We now fill in the table as follows:

Let I and J be states and let $I \xrightarrow{X} J \in E$ be an edge. For each item $(A \to \alpha.\beta, z) \in I$, the action performed in entry (I, z) of table T is:

$$T(I,z) = \begin{cases} (\mathbf{s}, n) & X \text{ is a terminal} \\ (\mathbf{r}, n) & X \text{ is a nonterminal} \end{cases}$$

where `n` is the index of the table corresponding to the item set J.

Similarly, the `reduce`-actions are added. For each item $(I, z, A \to \alpha)$ in R, we add the entry

$$T(I,z) = (\mathbf{r}, A \to \alpha)$$

Thus we have filled our table T with actions! Note that many of the entries do not have actions, which corresponds to an invalid input, since we encountered some input that was not part of the previous `first` set.

4. Parsing with derivatives

Now for something completely different!

Parsing with derivatives is a relatively new idea[3] based on an old concept[4]. Before discussing how we parse using derivatives, we'll look at how we recognize regular languages.

4.1. Regular languages

Let $L \subseteq \Sigma^*$ be a set of strings, which we call a *language*. We say that L is *regular* if we can construct L using the following atomic languages

- The empty language $\emptyset$ containing no strings.
- The *null* language $\{\varepsilon\}$ containing only the empty string ε.
- For any character $c \in \Sigma$, the set $\{c\}$ is a regular language

and the following operations

[3] `http://matt.might.net/papers/might2011derivatives.pdf`
[4] `https://dl.acm.org/citation.cfm?doid=321239.321249`

- Union, which we write as $L_1 \cup L_2$.
- Concatenation, written as $L_1 \circ L_2 = \{w_1\, w_2 \mid w_1 \in L_1, w_2 \in L_2\}$
- Kleene star, which is defined as

$$L^* = \bigcup_{n=0}^{\infty} L^n$$

where $L^n = \underbrace{L \circ L \circ \cdots \circ L}_{n}$.

4.2. Recognizing regular languages

We now define Brzozowski's derivative $D_c(L)$ of a regular language L with respect to a character $c \in \Sigma$:

$$D_c(L) = \{w \in \Sigma^* \mid cw \in L\} \tag{1}$$

An example is in order: Let $L = \{\texttt{spare}, \texttt{time}, \texttt{teaching}\}$. Then $D_t(L) = \{\texttt{ime}, \texttt{eaching}\}$.

This simple operation is extremely powerful. This is all due to the following simple property

$$cw \in L \iff w \in D_c(L).$$

Thus to determine if $w \in L$, we iteratively chop off each character c in w and derive L with respect to c, obtaining a new language L', which we use for the next iteration.

Finally, once we've derived with respect to each character, we check if the $\varepsilon \in L'$. By (1), this means that $w \in L$.

Now, we just need a simple way of computing $D_c(L)$ for any $c \in \Sigma$. Fortunately this derivative operation behaves nicely and satisfies

$$D_c(L_1 \cup L_2) = D_c(L_1) \cup D_c(L_2)$$
$$D_c(L^*) = D_c(L) \circ L^*.$$

Concatenation is not as straightforward since L_1 might contain the empty string ε. This is still easily handled though

$$D_c(L_1 \circ L_2) = \begin{cases} D_c(L_1) \circ L_2 & \varepsilon \notin L_1 \\ (D_c(L_1) \circ L_2) \cup D_c(L_2) & \varepsilon \in L_1 \end{cases}$$

We leave the job of defining an recursive function δ such that $\delta(L) = \texttt{true} \iff \varepsilon \in L$ as an exercise.

4.3. Modifying Brzozowski's derivative

We now move into the space of context-free languages. This means that our language L can be defined using a context-free grammar (CFG), for instance the BNF we created for `EXPR`. This presents us with a problem: CFGs can be recursive! This creates a problem when attempting to derive the language $E = (E \circ \{+\} \circ T) \cup T$ from `EXPR`

$$\begin{aligned} D_c(E) &= D_c((E \circ \{+\} \circ T) \cup T) \\ &= D_c(E \circ \{+\} \circ T) \cup D_c(T) \\ &= (D_c(E) \circ \{+\} \circ T) \cup D_c(T) \end{aligned}$$

Oops, we've reached another recursive call to $D_c(E)$. Thus a complete evaluation of $D_c(E)$ is not possible.

This brings us to the first fixup: **Laziness**. We may not even need to compute $D_c(E)$, since this depends on which character we're deriving with respect to. Thus, by computing the derivatives by need, we avoid this non-termination problem.

This fixup doesn't solve all of our problems though. Although we can now lazily compute $L' = D_c(L)$ for a context-free language L, we still need to check whether $\varepsilon \in L'$. This is done by computing $\delta(L')$, but since δ recursively traverse the structure of the L', it forces all lazy computations, once again causing non-termination. However, all of the necessary derivatives have already been computed when calculating L' lazily. Therefore the fix is simple: **Memoization**. If we memoize the derivative, the function will return the previously computed answer without diverging.

There is yet another problem with δ though, which neither laziness nor memoization seems to be able to solve. How do we check if $\varepsilon \in L'$? We cannot memoize anything since we're not constructing any object recursively, but rather we're looking for a single boolean answer. From previous courses in computer science, we know that this question is decidable, and thus we leave this as an exercise. (Hint: δ is a monotone function).

Using these simple fixups, we're now able to apply derivatives to recognize context-free languages!

4.4. Generating parse trees

Although we can now recognize context-free languages, we're still not satisfied! What we're really after is the parse tree. That is, the underlying structure of the flat text input we parsed.

To construct these parse trees, we take a quick detour and visit the concept of parser combinators. This is an extremely interesting area of parsing, but we only cover what's necessary to construct our parse trees.

4.5. Parser combinators

A *partial parser* $p : \Sigma^* \to \mathcal{P}(T \times \Sigma^*)$ is a function that, given an input string $w \in \Sigma^*$, computes partial parses of w. A partial parse is a pair (t, w') such that w' is a suffix of w and t is the parse tree corresponding to the prefix that p successfully parsed.

This is in contrast to a *full parser* $\lfloor p \rfloor(w) : \Sigma^* \to \mathcal{P}(T)$, which consumes the entire input and produces the set of trees corresponding to all successful parses of the entire string.

Note that given a partial parse $p : \Sigma^* \to \mathcal{P}(T \times \Sigma^*)$, we can construct a full parser

$$\lfloor p \rfloor(w) = \{t \in T \mid (t, \varepsilon) \in p(w)\}.$$

That is, we keep only the parse trees produced by p which were constructed from a complete parse of the entire string w.

We can now define very simple parses that handle the atomic languages described earlier:

$$\begin{aligned} c &= \lambda w. \begin{cases} \{(c, w')\} & w = cw' \\ \emptyset & w \neq cw' \end{cases} \\ \varepsilon &= \lambda w \,.\, \{(\varepsilon, w)\} \\ \emptyset &= \lambda w \,.\, \{\} \end{aligned}$$

Now, given two parsers $p, q : \Sigma^* \to \mathcal{P}(T \times \Sigma^*)$, we can construct a parse $p \cup q : \Sigma^* \to \mathcal{P}(T \times \Sigma^*)$ as follows

$$p \cup q = \lambda w \,.\, p(w) \cup q(w)$$

and similarly we can concatenate parsers to produce $p \circ q : \Sigma^* \to \mathcal{P}((T \times T) \times \Sigma^*)$

$$p \circ q = \lambda w \,.\, \{((t_1, t_2), w'') \mid (t_1, w') \in p(w), (t_2, w'') \in q(w')\}$$

The final operation we're missing is Kleene star. To define this in terms of parse combinators, we need to construct the parser that can parse the empty string. Letting

$$\varepsilon \downarrow S = \lambda w \,.\, \{(t, w) \mid t \in S\}$$

be a helper function that extract the parse trees from S, we can now define Kleene star. Given $p : \Sigma^* \to \mathcal{P}(T \times \Sigma^*)$, we define

$$\begin{aligned} p^* = (p \circ p^*) &\to \lambda(\text{hd},\ \text{tl}) \,.\, (\text{cons hd tl}) \\ \cup\, \varepsilon &\downarrow \{\text{nil}\} \end{aligned}$$

Thus p^* recursively constructs partial parses using p and concatenates these using cons. At some point, p does not succeed and p^* returns the empty parse.

4.6. Parse trees revisited

Being experts in parser combinators, we're now ready to define the derivative of a parser p.

$$D_c(p) : (\Sigma^* \to \mathcal{P}(T \times \Sigma^*)) \to (\Sigma^* \to \mathcal{P}(T \times \Sigma^*))$$

such that the full parser $\lfloor D_c(p) \rfloor(w)$ produces our parse trees along with a successful parse.

The derived language $L' = D_c(L)$ must satisfy that $w \in L' \iff cw \in L$. Thus the first attempt at a definition would be

$$D_c(p) = \lambda w \,.\, p(cw).$$

However, since $p(cw)$ contains parses that consumed no characters and returned the remaining unparsed string cw, we end up with a parser $D_c(p)(w)$ producing "prefixes" cw for the string w, which does not make sense. Thus we define $D_c(p)$ as

$$D_c(p) = \lambda w \,.\, p(cw) - (\lfloor p \rfloor(\varepsilon) \times \{cw\}).$$

Which fixes these problems. We can now apply the same iterative algorithm we used when recognizing regular languages and context-free languages, but instead of deriving the *language*, we derive the *parser*.

Getting Started with OpenGL

Mikkel Brun Jakobsen

November 31, 2014

1. Introduction

This note is a supplement to the talk "Introduction to OpenGL" since the available source code from the website might be intimidating on its own. The code snippets throughout this note are somewhat incomplete but should make sense together with the previously mentioned source code.

2. Creating a window / OpenGL context

Even through OpenGL is a cross-platform API, actually creating a window or an OpenGL context is a very non-cross platform process requiring various OS specific calls.

There are libraries that will make this process less painful. Various old OpenGL tutorials will suggest a library called *GLUT*, but it's old and unmaintained. I have listed a few "modern alternatives" below. Note that this list is far from exhaustive.

- *OpenGL Framework (GLFW)* (`http://www.glfw.org/`)
- *FreeGLUT* (`http://freeglut.sourceforge.net/`)
- *Simple Direct Media Layer (SDL)* (`http://www.libsdl.org/`)

I'm a big fan of GLFW. Here's an example of how to create a window with an OpenGL 3.3 context (*note*: no error handling, consult `http://www.glfw.org/` to create a more user-friendly application):

```
int main() {
  glfwInit();

  glfwWindowHint(GLFW_DEPTH_BITS, 32);
  glfwWindowHint(GLFW_RESIZABLE, GL_FALSE);
  glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
  glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
```

```
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

GLFWWindow* window = glfwCreateWindow(1024, 768, ‘‘Title", NULL, NULL);

glfwMakeContextCurrent(g_window);
...
```

3. Loading OpenGL functions

A window and an OpenGL context constitude a good start. But in order to be able to call OpenGL functions such as *glUseProgram*, we need to actually load these functions from the OpenGL context. This is also a platform specific task, and again we can use libraries to ease the pain. In the past, I've been using *GLEW* (`http://glew.sourceforge.net/`), but I've recently stumbled across *OpenGL Loader Generator* (`https://bitbucket.org/alfonse/glloadgen/wiki/Home`) which is also very cool.

Using GLEW and continuing the code from the previous section:

```
...
glewExperimental = GL_TRUE;
glewInit();
...
```

We should now have access to all the OpenGL functions necessary to start out with.

4. Main loop

When writing an OpenGL application, e.g., for a game, the usual approach is to have a tight loop that does something along the lines of:

- Check for and handle user input.
- Move objects based on their state and the current user input.
- Clear the back buffer.
- Draw all (visible) objects on the back buffer.
- Swap the front and the back buffer (showing the user the contents of the back buffer).

If you are unfamiliar with double-buffering, then checkout `http://en.wikipedia.org/wiki/Multiple_buffering#Double_buffering_in_computer_graphics`.

In code, this could look something like the following:

```
...
double current_time, previous_time, dt;
previous_time = glfwGetTime();
while(!glfwWindowShouldClose(window)) {
  current_time = glfwGetTime();
  dt = current_time - previous_time;
  previous_time = current_time;

  updateScene(dt);
  drawScene();

  glfwSwapBuffers(window);
  glfwPollEvents();
}
...
```

5. Rendering a square using OpenGL

5.1. 3D model terminology

A 3D model consists of *vertices* (singular: vertex) and *indices* (singular: index).

A vertex can be thought of as a position in some 3D space, but it might also contain other data such as color or texture coordinates.

The indices are used to look up vertices such that one or more OpenGL primitive can be formed. The most natural primitive when we think about computer graphics is a triangle, but OpenGL can also draw lines or points.

5.2. Specifying 3D data

We need a way to store vertices and indices such that OpenGL has access to it. OpenGL operates using a concept called "Buffer Objects". Buffer Objects can be used to store all sorts of data, including vertices and indices. We can get a buffer object handle by calling *glGenBuffers*.

Once we have a handle, we need to *bind* (using *glBindBuffer*) the object to specify its use:

- An "index buffer object" uses the `GL_ELEMENT_ARRAY_BUFFER` target.
- A "vertex buffer object" uses the `GL_ARRAY_BUFFER` target.

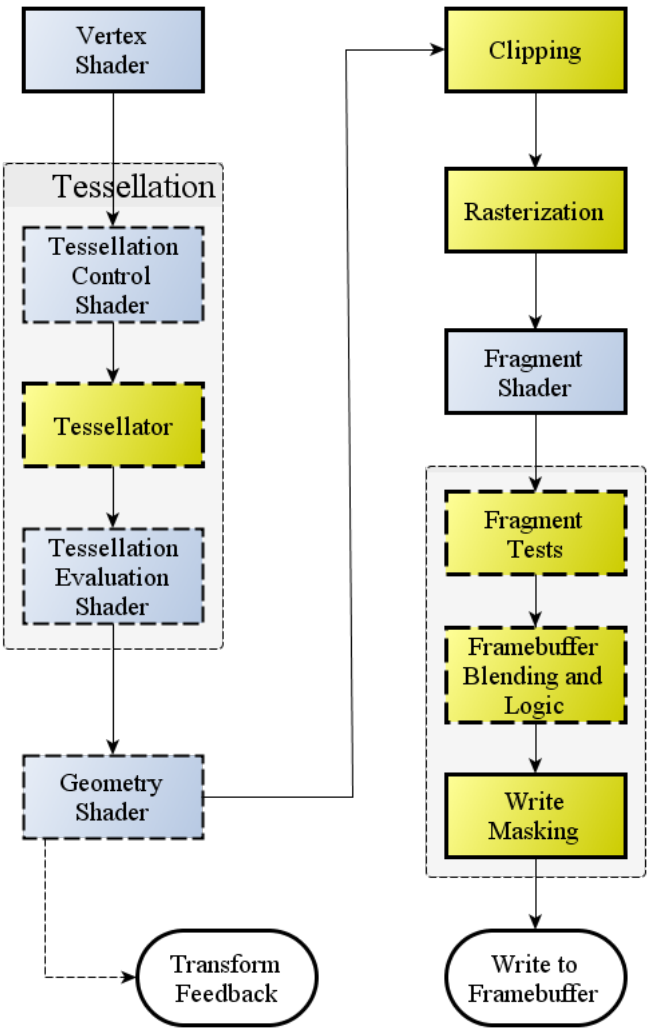
The following code creates a simple square in 3D, stores it in appropriate buffer objects, and sets up an attribute pointer to specify how OpenGL should interpret the data:

```
glGenBuffers(1, &ibo);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo);
{
  GLuint indices[6] = {
      0, 1, 2,
      0, 2, 3
  };
  glBufferData(GL_ELEMENT_ARRAY_BUFFER, 6 * sizeof(GLuint), indices, GL_STATIC_DRAW);
}

glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
{
  Vertex vertices[4] = {
      {{-0.5f, -0.5f, 0.0f}},
      {{ 0.5f, -0.5f, 0.0f}},
      {{ 0.5f,  0.5f, 0.0f}},
      {{-0.5f,  0.5f, 0.0f}},
  };
  glBufferData(GL_ARRAY_BUFFER, 4 * sizeof(Vertex), vertices, GL_STATIC_DRAW);
}

glEnableVertexAttribArray(0);
glVertexAttribPointer(
    0,
    3,
    GL_FLOAT,
    GL_FALSE,
    sizeof(Vertex),
    (GLvoid*) offsetof(Vertex, position)
);
```

In order for OpenGL to remember that the buffer objects above "belongs together" as well as how the vertex attributes are specified, the code above should be "wrapped" in a "Vertex Array Object" (*glGenVertexArrays* and *glBindVertexArray*).

5.3. The rendering pipeline

When rendering an object using OpenGL, the object goes through the OpenGL rendering pipeline, as illustrated by the following diagram:



Here is a brief description of what is going on:

- Each vertex of the 3D model is send through the active Vertex Shader which performs some operation on them. Since the vertices are independent, this process can be parallelized.
- Once all the vertices has been processed, are used to assemble primitives (e.g. triangles).
- Each primitive is *rasterized* to yield fragments (something that might become a pixel in the final image). Fragments consists of interpolated vertex data, i.e., a weighted average that depends on the fragments distance to each vertex.
- Each fragment is send through the active Fragment Shader to determine its color. Again, since fragments are independent, this process can be parallelized.

In legacy OpenGL, there was a static or a fixed pipeline where your data was expected to have a certain layout, and there were only a few ways to influence how the specified data should be rendered (by specifying lights etc.).

In modern OpenGL, you have to specify your *own* Vertex Shader and Fragment Shader, thus giving you complete control over how you want to render and, to some extent, specify your data.

This might seem a bit intimidating, but a working Vertex/Fragment-shader pair does not have to be too complicated as we shall look at in the following section.

5.4. Vertex Shader

```
#version 330

layout(location = 0) in vec3 att_position;

void main(void) {
  gl_Position = vec4(att_position, 1.0f);
}
```

The Vertex Shader is only required to write to the special variable *gl_Position*. The data written to *gl_Position* should be in so called *clip space*. The easiest way to think about clip space is as a cube with origin (0, 0, 0) and size 2 (extending from -1 to 1). Something that is inside this cube will be displayed on the screen, everything else will be culled. In a 3D application, the vertex shader would perform a perspective projection transformation such that the rendered model looks 3D. Here we simply pass on the position of the vertex, but we are actually performing what is known as an orthographic projection.

5.5. Fragment Shader

```
#version 330

out vec4 fragment_color;

void main(void) {
  fragment_color = vec4(0.0f, 1.0f, 0.0f, 1.0f);
}
```

The Fragment Shader is required to declare an output variable (a 4D vector when rendering to the default framebuffer) as well as to write a value to it. Here we simply output the color green.

5.6. Combining everything

When we have a 3D mesh stored in buffer objects and a Shader Program (*glCreateProgram()*) with a vertex and a fragment shader attached, rendering the 3D mesh is as simple as:

```
glUseProgram(g_program);
{
  glBindVertexArray(vao);
  {
    int triangle_count = 2;
    glDrawElements(GL_TRIANGLES, triangle_count * 3, GL_UNSIGNED_BYTE, (GLvoid*) 0);
  }
  glBindVertexArray(0);
}
glUseProgram(0);
```

Congratulations! You should now have a green square on your screen. To see how this can be extended to render something a bit more interesting, take a look at the material for "Introduction to OpenGL" or "Terrain and World Generation."

Starting Game Design

Christian Clausen

November 28, 2014

1. Introduction

What I want to talk about is a mix of science, experience, and observations. There is so much to cover in this topic so I have chosen to start way out from the top and then slowly zoom in on one specific genre. I hope that you can apply some of this and get some inspiration for what you can make tomorrow.

The format of the talk will be me talking, but I feel a lot of this material is learned through discussion and by saying the words. Therefore, I'm going to go through everything kind of fast. After that we will choose some games and analyze them in this framework. While I talk, there is a bunch of images going on, on the screen. They usually support what I am talking about, and they're generally recent, 2D, indie games. If you feel confident, I have set up a Socrative quiz where you can type in which game is on the slide.

[World of goo] This talk aims at giving you a scaffolding for talking about and thinking about game design. It is a "soft" topic, and there are lots of opinions, so if you don't agree with something I say, or have something to add, please, let's take a 5-minute discussion about it.

Let's start at the beginning: how to start designing a game. There are basically three ways, and you will notice how the "best" games tend to cover all three;

- [Starseed Pilgrim] Choose a feeling
  - Fear
  - Stress
  - Amazement
- [Thomas was Alone] Write a story
  - Love story
  - Gamification of Shakespeare
  - Or like here, how it may feel like becoming self-aware

- [Braid+Bridge] Choose a core mechanic
  - Add an extra dimension
  - Break and place blocks

Let's talk a little about each.

2. Choose a feeling

This is the most difficult way to start as it requires a deep understanding of that feeling and the situations that provoke it for different people, and sometimes across cultures. [Neverending Nightmare]

Here, we have a game that portraits how it is to live with OCD and depression in a semi-abstract manner. You wake up from a nightmare, only to find out you are in another nightmare. Most of the nightmares end in self-harm or suicide.

As of now, we (game designers) don't really have the understanding we need about most feelings. Actually, the only genre that we have a pretty good grasp on is horror and fear. Mainly because of the designer behind Amnesia who figured out:

> [Five Nights at Freddies] "If you want to create unease with the player, just give him nothing to do."

Let's look at why this makes sense; most fear lives in our fantasies, that is actually the distinguishing factor between horror and thriller. So, when we want to provoke it, we need to give the player a lot of time to think about all the things that could happen. Give him a lot of "hints" about the danger, then give his fantasies free run.

This game has four buttons and your only task is: don't die. But the constraint is that you can't just close the doors because you will run out of power. So, 95% of the time you have "nothing to do", except think about all the danger. It is pretty clear with the modern horror games that this is very effective.

3. Write a story

You can also decide that you have a great story that you wish to tell people [Papers Please]. Games are a great media for that since it can be so immersive. However, that immersiveness is also the danger because if you break the illusion... story's over.

There are many ways to break that illusion but the most common ones are lag, invisible walls, or bad controls. These are all *very* important to avoid, but they are more about engineering than design, so I will not elaborate further.

"Papers please" tells a story about a border-control point into an isolated, suppressive country. I imagine it much like North Korea or the old west Germany, but I am no expert here. The narrative is first person with a stressful puzzle mechanic, because you struggle to support your family when you can't solve the puzzles fast enough. This leads you to corruption, terrorism, and violence.

If you choose to design your game in this way, you have to remember that you are not authors nor directors who can decide what the protagonist does or thinks. Because the protagonist is the player, and he is going to play the way he wants to. That is why:

[Loved] Designing gameplay = Designing options for the player.

He may then chose whatever he wants, and it may not even matter, but by presenting the choice to him, he will have an easier time immersing.

In "Loved", you are often told what to do, but you can always choose to do something different. By putting the commands on the screen, the game makes you feel like you make more choices than if you had just played however you wanted to.

Another great example of this is games like Bioshock, where there is clearly only one way through the game, but the player can always just wander off and explore, maybe do some side quests, and the player is rewarded with more story. That brings me to another point:

Reward the player for playing.

Many new game designers think it is all about punishing the player for being bad [Super Meat Boy], however surprisingly (ironic) that isn't very effective. This game may seem like an anti-example, but in fact it isn't. Clearly, punishment is a big part of why this game is fun, and the way they make that possible is by having really small levels, so you don't get punished too hard. This is also a good example, because they really hit the balance on not making it too easy.

4. Choose a core mechanic

Notice that I said "choose" and not "invent" a core mechanic. That is because:

[Eets] The first step to becoming a great inventor is to steal a great invention.

"Eets" is a combination of Lemmings and the Incredible Machine.

If you look around the game market, you'll find multiple games with the same core mechanics. [Eufloria+Galcon] This is because it is easier and less risky to use something that has already been tested, and that you know works. It is also a lot easier to implement an idea that has already been "figured out", and then play with it until you have a new game. But it is also because (like with music);

[Knights of Pen and Paper] People like familiar things.

Here is an example of implementing the familiar tabletop role-playing game using familiar turn-based fighting mechanics. Choosing a familiar mechanic also has another advantage: you don't have to teach the player as much. This is important because of the key observation:

Nobody likes learning, everybody likes knowing.

A corollary of that is that you should avoid tutorials in your games. There are many ways this can be seen in the industry such as integrating them into the game, most

importantly to avoid the player feeling like he is being schooled. [Closure] In this game, for instance, the first puzzle has a box with the letter "D" on it. The latter is the key you use to walk and push the box which completes the level. So, we can make a small law of that:

> Tutorials suck...

Maybe I should clarify that by tutorial, I mean a dedicated tutorial, where the normal rules of the game don't apply. One way to address this problem, as I said, is to integrate it into the normal game. Another way is to use realism. Realism is a good way to avoid a tutorial, because if you have a mechanic that resembles something from the real world, people will already have an intuition about that, and you won't have to teach them. So, we can add to the last law:

> [Puddle] ... use realism to avoid tutorials.

In "Puddle", "you" are some liquid, and you can tilt the world to either side. First, everybody knows how liquids behave, and B it also means that when they switch to different liquids, the player immediately knows how this liquid will be different.

There is however something to keep in mind; gamers don't like the real world. Often, they play games to escape the real world, usually because it is too complex; too many factors, with too many connections to other stuff, and too many impressions. There is a pretty simple (and pleasing to computer scientists) way of avoiding this;

> [Anodyne] Use discrete representations.

Here, I am thinking about things such as health, keys, or the likes. I'm not saying that they can't be floating behind the scenes, but players will be much faster at making decisions based on discrete measures. If gameplay is when the player makes decisions, he can spend less time decoding values and more time playing.

When someone is playing your game, they are making a mental model of the universe your game is set in, so you should make the transition as seamless as possible. That means:

> [Starbound+Terraria] Your game should be transparent.

When you look at these games, you immediately think of tiles, and you know how tiles are supposed to behave, and that's how they behave. Notice how it is not because of realism this time.

It also means that if you track something about the player make him aware of it. [Mark of the Ninja] For instance, he needs to know whether the AI will react to sound. In "Mark of the Ninja" they actually show important sounds graphically, because not every player will play with sound enabled. Another reason for this is that a lot of games have sound but don't actively use it, so showing it graphically will make the player aware that this game does something different with sound.

You also need to be consistent about it: if you track something in one place, you need to track it "everywhere". If you have showed the player that the game uses sound as a mechanic, he will be thinking he can use it to his advantage too, like throwing a rock to

distract some guards. This, however, actually helps with an earlier point, that tutorials suck. In theater, there is a saying:

> "If you bring something on stage, use it!"

That applies to everything, if your actor has red hair, use that! If an actor is overweight use that! If you bring an umbrella, it can be a cane, a gun, a zip-line, anything! If you spent bits on something, use it as much as you can!

5. Go Simpler

Ok, that was a lot, let's take a little breather; [Cookie Clicker]

This may seem like an overly simple game, but there are actually a few points to be made here, the first is:

> Once you have found out what is fun about your game, do more of that.

This game is sort of the extreme example of that, but it worked. They figured out that people like getting achievements and points, so that is literally the only thing there is in this game. This guideline works better the earlier you find out if the mechanic works or not, so here is a quote that help with that:

> [Osmos] "If it isn't fun with boxes, it won't be fun with fancy boxes."

In "Osmos", you are a thing, and when you collide with something smaller than you, you absorb it and grow bigger. It is pretty clear that this game would be just as fun with plain circles, so you could get a prototype of a game like this up-and-running in 20 minutes and start testing fun.

These are archtypical examples of what we call mini-games, but another place where mini-games are more subtle is when they are embedded into other games, for instance, the inventory in Diablo 2 combined with items of different sizes, or the potion-system from the same game. The mini-game technically has nothing to do with what is fun in the main part of the game, which is collecting loot and killing stuff. So, they changed those mechanics in Diablo 3. The general point here is:

> Pop-up interfaces, and mini-games break the flow (fun) of the game.

It can be viewed as a context switch: you have to switch between just throwing spells and slashing enemies to solving the knapsack problem. The reason they implemented these mini-games was to make it more realistic, you can't carry the same number of broad-axes as gems. The problem is that it breaks another rule which relates to something we talked about earlier:

> [Plants vs. Zombies] Don't let realism stand in the way of fun.

In "Plants vs. Zombies", they clearly didn't do that. They did implement a more realistic mini-game where you just water plants to make them grow, but the two games are not intertwined with each other.

It is always dangerous when you add some realism, because you get a whole bunch of connotation that you might not want. Like, if the player is a human, we may expect that we can walk, jump, talk, and so on. But sometimes, we just have to take a step back;

> If it doesn't work, throw it out.

If you don't feel like stealing your mechanic, another way of finding one is to ask questions. Here are some examples of really interesting games that came from simple questions:

- What if there was no jump? [vvvvvv]
- What if you could only direct an AI? [Dwarfs]
- What if the music was tied to the level? [140]

I would like now to take some time to discuss some games, and then if there is time afterwards talk about game design from a more business point of view.

Let's start with some amazing games:

- [Don't Starve] - Distilled fun, no interfaces or mini-games, easy to build model of, freedom, good controls
- [Limbo] - Good story, consistent, good controls, no interfaces
- [Reus] - Discrete, no interfaces, good controls, freedom, time limit
- [Guacamelee] - Distilled, discrete, freedom
- [Death Rally] - Distilled, freedom

6. Online Games

Ok, so down to business. If we pick up where we left off, at the questions, you could ask:

- What if you just flip a switch? [avgn]

But wait, we already talked about that. This is pretty much the same (as cookie clicker), but in black and white. And these aren't the only ones [Fishville], here you just have more types of cookies.

Facebook and app games have gotten very popular and profitable, and it is because they are designed around some scientific reasons.

First, there is the biology:

- Dopamine is released when you receive acknowledgment (bonuses, achievements, stars, beat levels)

- As more and more dopamine is flooding your brain, it becomes desensitized (it gets used to the signal, so you need more to reach the same level of "feel good")
- Queue the Variable-Ratio Schedule of Reinforcement (a fancy way of saying: when the levels get harder, you feel like you have earned rewards more)
- And in the end, it would be a problem if you stay in the "high" state long enough to get completely desensitized, so they limit the length of your sessions.

But we can also look at this from a psychological point of view, in which we like getting validated. So, if a Facebook game can tell me something that I already know; like [Candy Crush], I am better than my brother. Great!

However, it isn't always the case, sometimes because it isn't, sometimes because I don't have the time to prove it. So, luckily the game provides a way for me to remedy this: buy points. Here, another interesting phenomenon kicks in:

"This must be worth money because look how much time I have spent on it"

And then I put in a little bit of money, like 1€. Almost nothing, but then another psychological trick kicks in:

"Now, it MUST be valuable, because only an idiot would spent money on it if it wasn't"

So now, I actually have to spent more and more money to stay on top. But every payment I make, gives the game a little more value. Of course desensitization also applies here, so the guilt (or whatever I'm feeling when I buy points) gets less and less severe, and since purchases aren't limited you can get completely desensitized to it. That's how you go broke, or if you're on the other side, become a millionaire.

A Formal Treatment of $^{k}/_{n}$ Power-Hours

Christian Clausen

1. Related Work

Power Hour is the name of a drinking game in which you take one shot of beer every minute, and while this is fun, sometimes you may prefer to drink more slowly. This requires a generalization of the Power Hour, however, sophisticated algorithms for this are prone to memory leaks. This problem is addressed in [1] with a category of algorithms called $^{k}/_{n}$ Power-Hours.

Further studies of the Power Hour were made in [2] where the author correctly points out that more games were possible than reported in [1]. He further studies the effects of adding multiple players.

We present a pictorial representation to address further memory leaks, a generalization over states, along with a series of propositions that suggest a possible game not considered earlier in the $^{k}/_{n}$ Power-Hours. We also include an appendix containing graphs over all the possible fair four state, one person games.

2. Introduction

Let's start with some notation. We have adapted the notation of [1] where $\uplus$ means full, $\cup$ means empty, and $\cap$ means flipped cup. Further, we have extended it with the symbol $+$ to shorten: fill and drink.

The rules of a $^{k}/_{n}$ Power-Hour are as follows: every minute you have to perform one or more actions ending in a (possibly new) state. Due to memory concerns, the actions should be uniquely determined based on the current state of your glass. If you have a full glass, you have to drink it before executing the actions. You will end up drinking k shots in n minutes.

Now, we are equipped to take a first look at our pictorial representation of a configuration, specifically the $^{2}/_{60}$ and $^{30}/_{60}$ as described by [1]. [1]

[1] We usually omit identity arrows, as here from $\cap$ to $\cap$, unless we only use one state.

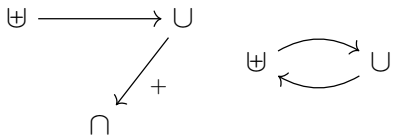


Figure 5: ${}^{2}/_{60}, {}^{30}/_{60}$

It reads:

- On ⊎ empty it and leave ∪. On ∪, fill and drink, then leave ∩ (on ∩, leave ∩).
- On ⊎ empty it and leave ∪. On ∪, leave it ⊎.

3. A Multitude of States

In this section, we present our generalized lemmas along with examples.

The first generalization we need to discuss is the number of states. While the conventional three are neat [2] suggested the fourth state ⊂ (lying) that can be used without conflict with the others. However, we advise against using it together with ⊃, as this will cause confusion since glasses can roll.

Having offered an extra state, we can only assume other people will do so as well. Therefore we will write ${}^{k}/^{s}_{n}$ for a ${}^{k}/_{n}$ game with s states. If nothing is indicated, we assume a 3-state game.

Definition 1. A fractional game is a ${}^{\frac{x}{d}}/^{s}_{n}$ where $x = a \cdot n + b$ for some a and b. A *pure* fractional game has $s = d$.

Specifically, this means a game with a graph where there is a cycle, the longest circle is d steps, and ⊎ is in it.

Lemma 2. *For a ${}^{k}/^{s}_{n}$ Power-Hour game there are s trivial fractional configurations; ${}^{\frac{n}{1}}/_{n}, {}^{\frac{n}{2}}/_{n}, \ldots, {}^{\frac{n}{s}}/_{n}$.*

Proof. by induction on the number of states. □

With the conventional three states, they look like this:

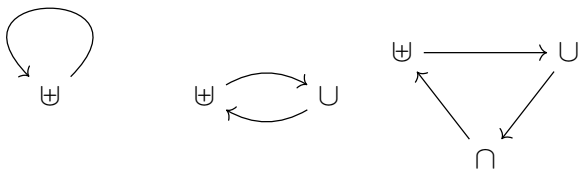


Figure 6: ${}^{\frac{n}{1}}/_{n}, {}^{\frac{n}{2}}/_{n}, {}^{\frac{n}{3}}/_{n}$

An interesting aspect is that we can add + to any of these arrows, the effect of which is are presented in the following lemma:

Lemma 3. *For a fractional ${}^{\frac{kn}{d}}/_{n}^{s}$ game with $k \leq d$, can make it into a ${}^{\frac{(k+1)n}{d}}/_{n}^{s}$ game.*

Proof. we add one drink pr. d steps in a k-step game, thus we have $\frac{1}{d}n + \frac{kn}{d} = \frac{n}{d} + \frac{kn}{d} = \frac{n+kn}{d} = \frac{(k+1)n}{d}$. □

We can now make the more interesting ${}^{2n}/_{n}$ and ${}^{\frac{2n}{3}}/_{n}$ games;

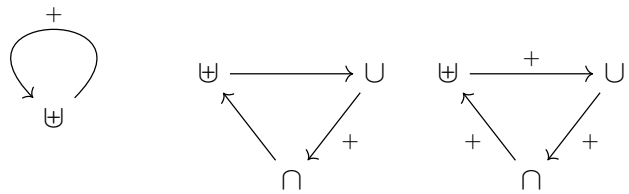


Figure 7: ${}^{2n}/_{n}$, ${}^{\frac{2n}{3}}/_{n}$, ${}^{\frac{4n}{3}}/_{n}$

In fact;

Theorem 4. *All possible pure fractional games ${}^{\frac{kn}{d}}/_{n}^{s}$ can be constructed from the previous two lemmas.*

Proof. left as an exercise to the reader. □

Corollary 5. All possible pure fractional games can be written on the form ${}^{\frac{kn}{s}}/_{n}^{s}$ with $k \leq s + 1$.

4. Constant Amount Games

In this section, we look at ${}^{c}/_{n}^{s}$ games where c is not affected by n. Intuitively, this means that we cannot have (non-identity) cycles in the graphs. We already saw the ${}^{2}/_{n}$ game. However which games are possible? In [1], the authors claim that a ${}^{3}/_{60}$ game is impossible[2], however we claim that it is possible, and even stronger:

Lemma 6. *With s states and $n \geq s$, there are s+1 lower constant games: ${}^{0}/_{n}^{s}, {}^{1}/_{n}^{s}, \ldots, {}^{s}/_{n}^{s}$.*

Proof. by induction on the number of states. □

Continuing with the classical examples, they are: [3]

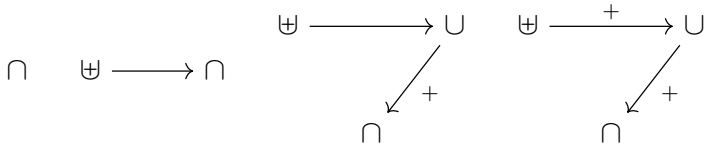


Figure 8: ${}^{0}/_{n}$, ${}^{1}/_{n}$, ${}^{2}/_{n}$, ${}^{3}/_{n}$

[2] they were conducting relevant research while writing it, so it is understandable.
[3] notice that we end on a ∩ to indicate that it does not need refilling.

At this point, you may think that we are done with constant games, however as noted by [1] there is some symmetry in this game, and we can take advantage of this here;

Lemma 7. *With s states and $n \geq s$, there are s upper constant games: ${}^{n-s+1}/_{n}^{s}, {}^{n-s+2}/_{n}^{s}, \ldots, {}^{n}/_{n}^{s}$.*

Proof. by induction on the number of states. □

You may have already figured out how they look, but here they are;

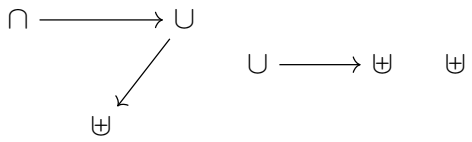


Figure 9: ${}^{n-2}/_{n}, {}^{n-1}/_{n}, {}^{n}/_{n}$

5. Having a Party

Definition 8. A game ${}^{k}/_{n}^{s}$ with $k \in \mathbb{N}$ is called fair.

Now, it is time to mix it all together:

Theorem 9 (Soundness)**.** *There is a fractional ${}^{\frac{kn-c}{d}}/_{n}^{s}$ game if $k \leq d+1, d+|c| \leq s$ and $kn - c \geq 0$.*

The equation may look complicated but it is quite logical, when you look at a picture:

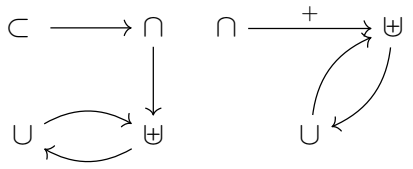


Figure 10: ${}^{29}/_{60}^{4}$, ${}^{\frac{61}{2}}/_{60} \cong {}^{31}/_{60}[2]$

And finally, we are ready to express the master lemma:

Theorem 10 (Completeness)**.** *All possible fractional games can be written as ${}^{\frac{kn-c}{d}}/_{n}^{s}$ with $k \leq d+1, d+c \leq s$, and $kn - c \geq 0$.*

which captures all the intuitions we have built.
Now that we have that in order, we should invite some friends:

Lemma 11. *If $d-1$ players take turns playing with the same cup and there is a fair game ${}^{\frac{kn-c}{d}}/_{n}^{s}$ with $p \mid c$, then there is a fair ${}^{\frac{kn-c}{pd}}/_{n}^{s}$ game.*

This means that with 2 girls, 1 cup, and an hour, the possible games are:

$$\frac{60k}{2\cdot 3}/_{60}^{3} = {}^{10k}/_{60}^{3}$$

for $k \leq 4$ (due to corollary 5), giving the new configuration ${}^{10}/_{60}$. If we also have a die, we could even get: ${}^{10k-1}/_{60}^{9}$ for $k \leq 4$. And with 3 people, we could play ${}^{5k}/_{60}^{4}$ for $k \leq 5$, giving the new (fair) games: ${}^{5}/_{60}^{4}$ and ${}^{25}/_{60}^{4}$.

6. Bibliography

[1] Ben Blum, Dr. William Locas, Chris Martens, Dr. Tom Murphy VII, *Algorithms for k/n Power-Hours*, SIGBOVIK 2012
[2] Dr. Tom Murphy VII, *New results in ${}^{k}/_{n}$ Power-Hours*, SIGBOVIK 2014

A. 4-State Games

A.1. Trivial Fractional Games

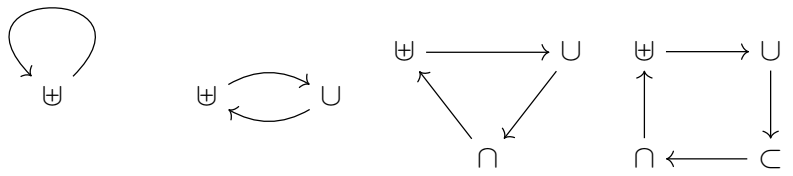


Figure 11: ${}^{n}/_{n}^{4}$, ${}^{n}/_{n}^{4}$, ${}^{n}/_{n}^{4}$, ${}^{n}/_{n}^{4}$

A.2. Other Fractional Games

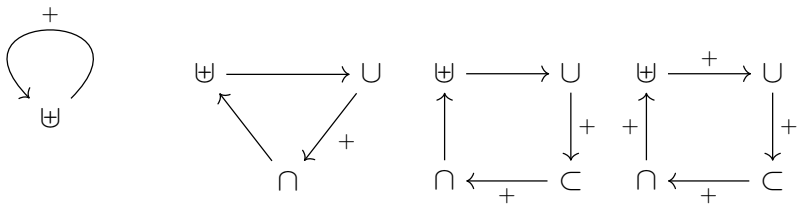


Figure 12: ${}^{2n}/_{n}^{4}$, ${}^{2n}/_{n}^{4}$, ${}^{3n}/_{n}^{4}$, ${}^{5n}/_{n}^{4}$

A.3. Lower-Constant Games

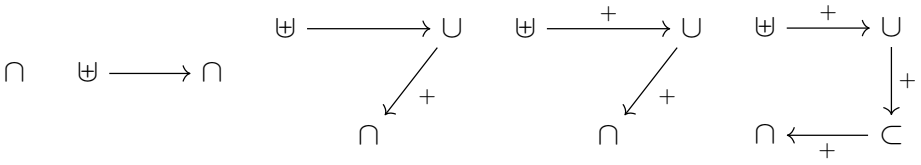


Figure 13: ${}^{0}/_{n}^{4}, {}^{1}/_{n}^{4}, {}^{2}/_{n}^{4}, {}^{3}/_{n}^{4}, {}^{4}/_{n}^{4}$

A.4. Upper-Constant Games

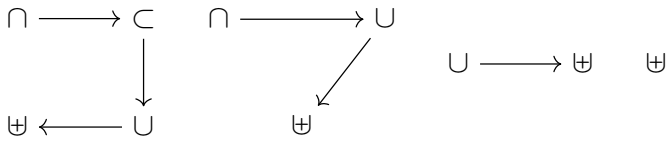


Figure 14: ${}^{n-3}/_{n}, {}^{n-2}/_{n}, {}^{n-1}/_{n}, {}^{n}/_{n}$

A.5. Mixed Games

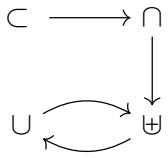


Figure 15: ${}^{\frac{n-2}{2}}/_{n}^{4}$

Robust Random Permutations with Constraints of Groups of Students over Time and How to Qualify for the Paper with the Longest Title in This Years Compendium of Spare-Time Teaching Notes

Nicolaj Græsholt <projects@ohmyunix.com>
OhMyUnix Institute of Technology

This paper presents a specific application of the Bin-Packing problem wherein we work with a varying amount of objects and bins. The objects are all dynamically weighted corresponding to which objects they've shared a bin with before. We want to be able to redistribute the objects such that each object will share a bin with the same objects a minimum of times with objects dis- and reappearing in between distributions.

A more thorough analysis is provided along with a proposed solution that's a naive implementation of how one would do it by hand in small examples. Further research includes actually proving whether it says *this can be proved on occasion* as well as testing other algorithms; maybe proving $\mathbf{P} = \mathbf{NP}$; and if we have the time, provide an efficient solution to the problem afterwards.

1. Analysis of the problem

The problem is a variant of the NP-Hard problem Bin Packing which consists of packing a number of objects in a number of bins most efficiently.

The *weighted bin-packing problem* issues weights to the different objects and must pack the objects in the smallest number of bins or into bins of a specific size.

In this particular problem, we have n students distributed in m super-groups of size p. We have to distribute these students into a minimum number of bins/subgroups of size q such that each subgroup only has one member from each distinct super-group while that particular super-groups material is presented; thus providing coverage of the studied material but still only allowing the presenting student to know the presented material beforehand. The super-groups can also have s relations to other super-groups, this means they've studied the same material and that these students shouldn't be in the

same subgroups either; two super-groups are distinct if they're not related. Furthermore, we have to do this r times while minimizing the number of times a student is put into a subgroup with a student that he/she has been in a subgroup with in a previous distribution and thus the input also consist of a possible $O(n^2)$ relations between the students that represent whether they're in a super-group or have been in a subgroup before.

The solution for this problem must be robust and account for students being removed temporarily from super-groups between distributions.

2. A proposed solution

To solve the problem, we provide a randomized dynamically weighted bin-packing algorithm wherein the weight of each student is computed relative to the students already in the subgroup.

2.1. Dynamical weighting

The weight is calculated dynamically for each student as follows

- if the subgroup is empty or there are only other students that the student has never seen before, the cost is 0;
- for each student in the subgroup that the student has been in a subgroup with before, the cost is 1; and
- for each student in the subgroup that are in the students super-group or a related super-group, the cost is 100[1].

The size of a bin (subgroup) is 100 such that a student can only be added to the subgroup if no other student in the students super-group (or a related super-group) is already in the group – if there are no bins/subgroups left with a cost under 100, a new one is created.

We always assign a student to the bin/subgroup with the lowest cost and we always assign the students with the lowest costs first. This strategy of using the optimum solution each time is believed to lead to the best optimal solution - this can be proved on occasion.

2.2. Randomization

The solution provided for this problem uses randomization to shuffle the order of the groups and students. This is done to lower

[1]This should just be a large number. A number is large if it's not possible to reach it by the previous weight rule. The size of the number has no effect on the run time of the algorithm, it only increases the memory needed to run the algorithm as it has to store the number.

- the chance of "unlucky" inputs, this can be proved on occasion; and
- the probability that specific students are always the last to be treated and thus end up being the ones having to make compromises when creating groups, this increases student satisfaction which can be proved on occasion.

2.3. A good or best-available solution

Since the problem doesn't state the chance that a certain student might be missing we're not able to make intelligent guesses about the next distribution and is thus left to greedily take a good or the best-available distribution we can come up with at the moment.

- A *good* distribution has 0 students in a subgroup with students they've seen before; and
- A *best-available* distribution at a certain time has as little students in subgroups with students they've seen before as possible.

In order to solve the problem, we keep track of the sum of the costs of the bins we've had to choose for the students, i.e., if a student is added to a subgroup with n students they've been in subgroups with before, the added cost to the solution is n. The final cost is the price of the solution.

By randomizing the input it's possible to run the algorithm several times on the same input and get results with different costs. In practice, we run it 10 times and select the solution with the lowest cost, the number of runs needed to reach an estimated lowest cost can be proved on occasion. Attempting on a computer to choose the best solution out of 10 tries and 100 tries, and doing both 1000 times, we found that with 24 students distributed in super-groups of 3 that each were paired with a relation, the average cost didn't change notably, i.e., more than a few decimals.

3. Lacking combinations

It's possible that the algorithm creates a subgroup that does not contain a sufficient amount of students from different subgroups such that all the material is covered in such a subgroup; remember that unrelated super-groups cover distinct material. In our specific application, we only want the presenting student to know the material beforehand, thus it leaves us with a number of ways to mitigate this problem. Below is a few solutions to different combinations of lacking combinations.

3.1. A full group, one group lacking one member

Let a_1, a_2, b_1, b_2, and c_1 be members of the super-groups A, B, and C respectively and let the algorithm distribute them into the subgroups (a_1, b_1, c_1) and (a_2, b_2). The process is:

1. create the union of the groups $(a_1, b_1, c_1, a_2, b_2)$;
2. ask the student c_1 to present his material; and
3. split the groups into (a_1, b_1, c_1) and (a_2, b_2) again.

Thus every student is able to get a presentation of each distinct piece of material and no student is in a group where his material is presented.

3.2. A full group, one group lacking all but one member

Let a_1, a_2, b_1, and c_1 be members of the super-groups A, B, and C respectively and let the algorithm distribute them into the following subgroups, (a_1, b_1, c_1) and (a_2). The process is:

1. create the union of the groups (a_1, b_1, c_1, a_2);
2. ask the student c_1 to present his material;
3. ask the student b_1 to present his material;
4. split the groups into (a_1, b_1) and (a_2, c_1); and
5. ask the students a_1 and a_2 to present their material.

Thus every student is able to get a presentation of each distinct piece of material and no student is in a group where his material is presented.

3.3. Two groups lacking one distinct member

Let a_1, b_1 and a_2, c_1 be members of the super-groups A, B, and C respectively and let the algorithm distribute them into the following subgroups, (a_1, b_1) and (a_2, c_1).

This solution is left as an exercise to the reader. Hint: It's noticed that this combination of groups actually occurs in the solution above.

4. Conclusion

The algorithm solves the problem because no two students are added into groups with other members of their (or their related) super-group. The algorithm is robust as it's no prerequisite that all students are available, and even in scenarios with enough missing students such that lacking-groups are created, it's possible to rearrange the groups in trivial ways to maintain the invariant such that no student has to give a presentation while another student from a related super-group is present.

We did not have time to prove $\mathbf{P} = \mathbf{NP}$, which is why we'd also take the last chance to provide this as an exercise to the reader.

Qualifying for "The Paper with the Longest Title" may seem trivial, but is actually reducible to the financial problem *Blind Bidding if You Are Scrooge McDuck and Have a Near Infinite Amount of Resources But Would Like To Use as Small an Amount as Possible*, which also qualified for the *problem* with the longest name that year.

Functions then Proofs, and back again

Christian Clausen

In this paper, we will play around with Coq.

1. List

Let's start with a simple and familiar datatype and a function to go along with it:

```
Inductive list (A : Type) : Type :=
| nil : list A
| cons : A -> list A -> list A.

Definition test1 := nil nat.
Definition test2 := cons nat 42 (nil nat).
Definition test3 := cons nat 3 (cons nat 2 (cons nat 1 (nil nat))).

Fixpoint append1 (A : Type) (xs ys : list A) :=
  match xs with
    | nil => ys
    | cons x xs' => cons A x (append1 A xs' ys)
  end.
```

Here, we have lists and a polymorphic append function for them. The Curry-Howard correspondence tells us that we can write proofs as programs, which begs the question: how about the other way? Can we write programs as proofs?

2. Curry-Howard

As it turns out, Coq provides a way for us to play with that. Instead of writing `:=` and then the implementation, we just write `:` and the type. If we were to write `append` this way it would look like:

```
Definition append2 :
  forall A : Type, list A -> list A -> list A.
```

Now, the fun part comes because we now have to make a proof of `forall A : Type, list A -> list A -> list A`. This seems fairly easy, here is our first attempt:

```
intros A xs ys.
exact xs.
```

Now, we can say **Qed** and try to **Compute** `append2 nat test1 test1`, but to our surprise Coq says that the result is `append2 nat test1 test1 : list nat`. This is because Coq thinks `append2` is a lemma, not a function. Because we ended with **Qed** `append2` is opaque, meaning we can't compute it. We can change that by writing **Defined** instead of **Qed**.

Now, **Compute** `append' nat test1 test1` gives `nil nat`. Great! High on our success we continue our systematic testing; **Compute** `append2 nat test1 test2` but unfortunately this also gives `nil nat`.

We now realize that the lazy proof we did might have been important, we therefore go back and revisit it:

```
intros A xs ys.
induction xs as [ | x xs' IHxs'].
exact ys.
apply cons.
exact x.
exact IHxs'.
```

After some testing, we conclude that this "implementation" seems to work. I guess induction really is recursion.

3. Vector

At this point, we randomly remember that we have dependent types so we decide to go on a small adventure and invent a new type of list that knows its own length:

```
Inductive vector (A : Type) : nat -> Type :=
| Nil : vector A 0
| Cons : forall n : nat, A -> vector A n -> vector A (S n).

Definition test1' := Nil nat.
Definition test2' := Cons nat 0 42 (Nil nat).
Definition test3' := Cons nat 2 3 (Cons nat 1 2 (Cons nat 0 1 (Nil nat))).
```

Of course, we immediately decide to try and define `append` with our new method:

```
Definition append4 :
  forall A : Type, forall m n : nat,
    vector A m -> vector A n -> vector A (m + n).
intros A m n xs ys.
induction xs as [ | m' x xs' IHxs'].
rewrite plus_0_l.
```

```
exact ys.
rewrite plus_Sn_m.
apply Cons.
exact x.
exact IHxs'.
```

Unfortunately, when we test this we get a lot of unreadable stuff. This is because our proof uses `plus_0_l` and `plus_Sn_m`, both of which are opaque. There are two obvious solutions to this problem: we could make custom versions of these lemmas and end with **`Defined`**, or we could use something different, we chose to just use `simpl` both places. Behold, it works perfectly.

4. Conclusion

Now that we have mastered programs-as-proofs, we decide to go back to the nice world of Gallina to implement the vector append in the good old way.

This turns out to be difficult, very difficult in fact. A solution exists, however how and why it works is out of the scope of this paper. Here it is:

```
Fixpoint append6 (A : Type) (m n : nat) (xs : vector A m) (ys : vector A n
   ) : vector A (m + n) :=
  match xs in vector _ m' return vector A (m' + n) with
    | Nil => ys
    | Cons m' x xs' => Cons A (m' + n) x (append6 A m' n xs' ys)
  end.
```

The point is that we have actually implemented a function as a proof, that we could not have implemented in the usual way.

In the file[1], you will also find implementations of length on both lists and vectors, and of course the one on vectors is rather short. While it may seem like you are not getting anything from using vectors, because you have to put their lengths everywhere, you should remember that in a practical setting you would use implicit parameters for both the polymorphic type parameter and the lengths of the vectors, thus we can call **`Compute`** `length3 test2'`.

5. Exercises

1. Implement `plus` as a proof.
2. Implement `fac` as a proof.
3. Prove associativity for all `append`s.
4. Prove distributivity of all `length`s over `append`.

[1] `daimi.au.dk/~christia/STTNC.v`

5. Implement `reverse` for lists and vectors as a proof.
6. Define a type of natural numbers that keeps track of their parity.
7. Implement plus on your new natural numbers as a proof.
8. Implement plus on your new natural numbers as a function.
9. ♡ Implement `ackermann` as a proof.
10. ♡ Implement `fib`.