Politechnika Gdańska
Wydział Fizyki Technicznej i
Matematyki Stosowanej

Katedra/Zakład:
Kierunek studiów: Matematyka
Specjalność: Biomatematyka
Rodzaj studiów: stacjonarne
Imię i nazwisko: Michał Stańczyk
Numer albumu: 116885

PRACA DYPLOMOWA INŻYNIERSKA

Temat pracy:
On mathematical methods in programming computational processes.

Zakres pracy:

Praca poświęcona jest opisywaniu obliczeń (programowaniu). Prezentuje matematyczny aparat pojęciowy, w ramach którego omówiono metody implementacji języków programowania przez *interpretację* i *kompilację*. Ostatni rozdział poświęcony jest metodom generowania programów przez *abstrakcyjną interpretację*. Pracy towarzyszy płyta CD z pełną implementacją opisanych w niej narzędzi i wyników.

Potwierdzenie przyjęcia pracy:

Opiekun pracy
dr inż. Marcin Styborski

Kierownik Katedry/Zakładu
prof. dr hab. Marek Izydorek

Gdańsk, 20.02.2013r.

OŚWIADCZENIE

Imię i nazwisko	Michał Stańczyk
Wydział	Wydział Fizyki Technicznej i Matematyki Stosowanej
Kierunek	Matematyka
Data i miejsce urodzenia	26.12.1984 Gdańsk
Adres	ul. Do Studzienki 29a/11 80-227 Gdańsk
Rodzaj studiów	stacjonarne

wyrażam/ ~~*nie wyrażam*~~ zgodę/~~y~~ * do korzystania z mojej pracy dyplomowej:
On mathematical methods in programming computational processes.
do celów naukowych lub dydaktycznych[1].

Świadomy(a) odpowiedzialności karnej z tytułu naruszenia przepisów ustawy z dnia 4 lutego 1994r. o prawie autorskim i prawach pokrewnych (Dz. U. Nr 80, poz. 904 z 2000r ze zmianami) i konsekwencji dyscyplinarnych określonych w ustawie Prawo o szkolnictwie wyższym (Dz. U. Nr 164, poz. 1365 z 2005r.) [2], a także odpowiedzialności cywilno-prawnej oświadczam, że przekładana praca dyplomowa została opracowana przeze mnie samodzielnie / ~~została opracowana w zakresie~~: [3] *
.....
.....
i nie była wcześniej podstawą żadnej innej urzędowej procedury związanej z nadaniem dyplomu szkoły wyższej uczelni lub tytułów zawodowych.
Wszystkie informacje umieszczone w pracy, uzyskane ze źródeł pisanych i elektronicznych oraz inne informacje, zostały udokumentowane w wykazie literatury odpowiednimi odnośnikami zgodnie z art. 34 ustawy o prawie autorskim i prawach pokrewnych. Jednocześnie wyrażam zgodę na dołączenie tekstu pracy do bazy prac systemu antyplagiatowego.

Gdańsk, dnia 20.02.2013 .....
podpis studenta

*) niepotrzebne skreślić

[1] Zarządzenie Rektora Politechniki Gdańskiej nr 34/2009 z 9 listopada 2009 r., załącznik nr 8 do instrukcji archiwalnej PG
[2] Ustawa z dnia 27 lipca 2005r. Prawo o szkolnictwie wyższym. Art. 214 ustęp 4. W razie podejrzenia popełnienia przez studenta czynu polegającego na przypisaniu sobie autorstwa istotnego fragmentu lub innych elementów cudzego utworu rektor niezwłocznie poleca przeprowadzenie postępowania wyjaśniającego. Art. 214 ustęp 6. jeżeli w wyniku postępowania wyjaśniającego zebrany materiał potwierdza popełnienie czynu, o którym mowa w ust.4, rektor wstrzymuje postępowanie o nadanie tytułu zawodowego do czasu wydania orzeczenia przez komisje dyscyplinarną oraz składa zawiadomienie o popełnieniu przestępstwa.
[3] dotyczy prac dyplomowych realizowanych wspólnie

Streszczenie

Niniejsza praca poświęcona jest w całości środkom wyrazu obliczeń – a więc programom i językom programowania.

Omawiamy techniki metaprogramowania: interpretację i kompilację, oraz mniej znaną metodę abstrakcyjnej interpretacji. Prezentację przeprowadzamy w ramach wypracowanego przez nas aparatu pojęciowego, opartego o terminologię teorii mnogości i algebry uniwersalnej.

Staramy się uwydatnić warstwową naturę (meta)obliczeń, mającą swój wyraz w spostrzeżeniu, że interpreter (a więc emulator pewnego modelu obliczeniowego $\mathcal{M}_2$ w pewnym modelu obliczeniowym $\mathcal{M}_1$) jest programem, generującym (w $\mathcal{M}_1$) proces obliczeniowy emulujący inny proces obliczeniowy, generowany (w $\mathcal{M}_2$) przez inny program.

Praca składa się z trzech rozdziałów:

1. "O obliczaniu i obliczeniach" – w którym proponujemy prosty model obliczeniowy $\mathfrak{S}$. Model formalizujemy jako maszynę abstrakcyjną $DRC\langle\mathcal{SE}\rangle$, wykazujemy jego pełność, wskazujemy ograniczenia i szkicujemy możliwe uogólnienia $DRC\langle\mathcal{A}\rangle$. Wreszcie w kontekście uogólnionych maszyn DRC definiujemy kluczowe w pracy pojęcia *procesu obliczeniowego*, *procedury* i *programu*, oraz dwie funkcje semantyczne: *"górną"* przyporządkowującą programowi jego funkcję wejścia-wyjścia, oraz *"dolną"*, przyporządkowującą programowi rodzinę procesów przez niego generowanych.
2. "O programowaniu i programach" – w którym wprowadzamy pojęcie *języka programowania* i omawiamy sposoby implementacji języków poprzez interpretację i kompilację. Dalsza część rozdziału, podzielona na sześć podrozdziałów, poświęcona jest dwóm przykładowym językom – $drcz_1$ będącego naszym wariantem LISPa Johna McCarthy'ego ([13]), oraz $FCL\langle\mathcal{SE}\rangle$ – prostego imperatywnego języka zaproponowanego w [10], szczegółowo omówionego w [9] i błyskotliwie użytego w [7]. Kontemplujemy naturę interpretacji i kompilacji na przykładach przejść między modelami obliczeniowymi $DRC\langle\mathcal{SE}\rangle$, $FCL\langle\mathcal{SE}\rangle$ i $drcz_0$. Za pomocą prostych środków dowodzimy poprawności jednego z kompilatorów.
3. "O generowaniu i generatorach" – w którym demonstrujemy i próbujemy wyjaśnić abstrakcyjną interpretację – metodę transformacji programów opartą o semantykę (dolną), na której oparte są techniki agresywnej propagacji stałych (*partial evaluation)* oraz nadkompilacji (*supercompilation*). Odtwarzamy wyniki [7] i prezentujemy kilka własnych. Na zakończenie szkicujemy plan dalszych badań.

Praca opatrzona jest pełną implementacją omówionych narzędzi załączoną na płycie CD. Implementacja ma formę emulatora maszyny $DRC\langle\mathcal{SE}\rangle$ (zaimplementowanego w języku C dla systemów unixowych), skompilowanego kompilatora, oraz zestawu programów udokumentowanych w załącznikach.

Abstract

This thesis investigates means of describing computations – programs and programming languages. We study the techniques of implementing languages by interpretation, compilation and abstract interpretation (driving), using simple mathematical framework. We discuss three languages: of $DRC\langle\mathcal{SE}\rangle$ abstract machine, $drcz_0$ LISP variant, and simple imperative flowchart language $FCL\langle\mathcal{SE}\rangle$. We implement each one of them in the others, and present driving-based program generation technique of online partial evaluation for $FCL\langle\mathcal{SE}\rangle$ as presented in [7]. We finish with our humble result – a non-trivial compilation with first Futamura projection (from subset of $DRC\langle\mathcal{SE}\rangle$ to $FCL\langle\mathcal{SE}\rangle$).

Contents

Preface

The art of constructing software systems seems to be majorly the art of describing their behaviour. This thesis is entirely devoted to means of expressing *computations* – single-threaded, (mostly) non-interactive input-output transformations, that constitute the building blocks of every software's behaviour.

Various methods of interpretation, compilation (translation), abstract interpretation (driving), and equivalence proofs are well known, and the variants which we propose do not pretend to originality. What we believe to be novel is the uniform conceptual framework in which we present them. We try to emphasize the "laminar nature" of computations – that metacomputations can go many levels above the straightforward interpretation, and that such liberation can be useful in practice. The framework relies on what we dubbed "mathematical methods" – using concepts from sets theory and universal algebra, arguing with proofs, and tendency towards simplicity and elegance.

Our major contribution - the $\mathfrak{S}$ model, formalized as $DRC\langle\mathcal{SE}\rangle$ machine, was strongly influenced by Landin's SECD machine (as described in [11]) – we have invented it while trying to modify original SECD design to serve as virtual machine for $drcz_0$, our experimental variant of McCarthy's LISP ([13]). The main difference is explicit management of environment, which makes implementing languages with various control mechanisms easier. The $\mathfrak{S}$ model is as (*a posteriori*) attempt at explaining $DRC\langle\mathcal{SE}\rangle$'s architecture.

The core of thesis consists of three chapters:

1. *On computing and computations*, in which we analyze computations and their procedures, introduce the basic computational model $\mathfrak{S}$ and it's formalization – abstract machine $DRC\langle\mathcal{SE}\rangle$, in context of which we define the fundamental notions of this thesis.

2. *On programming and programs*, in which we discuss means of expressing computations – programming languages, and their associated computational models. We discuss interpreters and compilers, implement two example languages, prove correctness of particular compiler, and provide example application in studies of sentential calculus.

3. *On generating and generators*, in which we try to explain the phenomena of abstract interpretation, the core of semantic-based program transformation techniques of *partial evaluation* and *supercompilation*. Results of [7] are reproduced, and some of our own provided. We conclude with challenges and further work.

Full implementation is supplied on attached CD. It centers around $DRC\langle\mathcal{SE}\rangle$ machine emulator, written in C language and reviewed in appendix A. All other tools are written in languages described in thesis, and can be executed on the emulator (*cf.* appendices B and C). Major motivation for writing these implementations was to try "mathematical methods" in practice.

The reader shall judge how promising these results are.

Acknowledgement

The author is grateful to Maciek Godek, B.Sc. for many years of stimulating discussions, and reading the drafts of this thesis, and to Kuba Dobrosielski, Eng. for careful reading, and help with my poor English. Many thanks to Paweł Pawłowski, B.Sc. and Kuba Kolecki, Eng., for their contributions (mentioned in footnotes), and to my advisor Marcin Styborski, Ph.D. for being Best Project Manager Ever.

This work is dedicated to the memory of Valentin Turchin, Peter J.Landin, and John McCarthy, the real superheroes.

1 On computing and computations

The purpose of this chapter is to introduce conceptual apparatus for further investigations.

In section 1.1 we try to capture computations in their natural environment of integer arithmetics, in order to introduce the notion of procedure – "knowledge on how to perform computations". Section 1.2 presents our simple stack-based computation model $\mathfrak{S}$, which provides unambiguous notation for descriptions of procedures. We prove its expressive power and limitations in section 1.3, and extend its domain from integers to variant of McCarthy's *S-expressions* in section 1.4. In section 1.5 a formalization of $\mathfrak{S}$ is proposed, in form of abstract $DRC\langle\mathcal{SE}\rangle$ machine, in context of which section 1.6 defines key notions of this work – of *computational process*, *procedure* and *program*.

1.1 Computations and procedures.

Assume we are able to calculate sums and products of any two numbers, and are willing to find [number being] the sum of squares of numbers 2 and 3. In other words we are willing to *compute* the sum of squares of numbers 2 and 3, or to compute *value* of "sum of squares of two numbers" *function* at *arguments* 2 and 3.

In order to do that, we could first take 2, multiply by itself obtaining 4, then take 3, multiply by itself obtaining 9, and finally add these results obtaining 13.

Notice the actions taken did not depend on the choice of numbers 2 and 3. We could use the same sequence of actions to compute sum of any two numbers:

Example 1. *To compute sum of squares of [any] two numbers:*

1. *take the first number, multiply by itself,*
2. *take the second number, multiply by itself,*
3. *add the results.*

We have described (one possible) universal method of computing "sum of squares of two numbers" function at any two arguments given - universal in the sense that anyone capable of adding, multiplying, and remembering numbers can perform it, obtaining correct result.

Which leads to the following:

Remark. *There exist effectively computable arithmetical functions,* i.e. *ones for which some universal method of computing their values at any arguments given exists.*

We will call such universal methods *procedures*.

Examples of similar functions can be provided instantly, like "the sum of squares of three numbers", "the sum of cubes of...", etc. - for each of them it's easy to propose a procedure of sequential additions and multiplications.

Such procedures seem quite simple, as they involve fixed number of actions. This does not have to be the case, as will be shown in the following two examples.

Example 2. *To compute absolute value of number given:*

1. *take the number,*
2. *if it is negative, multiply by -1,*
   otherwise leave it intact.

This procedure involves choices - under such-and-such conditions do this, otherwise do that. The number of instructions - and consequently number of actions to perform - depends here on the sign of argument[1].

Example 3. *To compute the factorial of given number:*

1. *take the number,*
2. *if it equals 0, take [constant] 1,*
   otherwise compute the factorial of this numbers predecessor,
   and multiply it by this number.

This procedure involves *i.a.* "embedded computation" - in the course of performing it, at some moment one needs to follow some other (known) procedure, *i.e.* "apply it" to some of partial results, and to use its result afterwards. Such "embeddability" of procedures enables capturing repetitions, and enables abstracting similar computations. As an example of the latter consider alternative description of the "sum of squares..." procedure:

Example 4. *To compute sum of squares of two numbers:*

1. *take the first number and compute its square,*
2. *take the second number and compute its square,*
3. *add the results;*

To compute square of given number:

1. *take the number and multiply by itself.*

In case of more complicated "sub-procedures" this could significantly simplify describing procedures. These issues are discussed in more details in chapter 2.

[1] If we could also compute the square root of number given, we could avoid that, but this is not the point. Some other (equally unsatisfactory) possibility we have discussed in section III of [20].

In summary it was shown that procedures may consist of:

1. performing "atomic" tasks, like additions, comparisons, etc.,
2. conditional choices,
3. following some (possibly other) known procedure.

1.2 Modeling procedures.

A possible way of describing procedures (of computing arithmetical functions) will now be proposed, with unambiguous rules of interpretation. These rules assume some particular setting at which the computer (a person performing computation) operates. In the following sections we will refer to these rules and setting as **the $\mathfrak{S}$ model**.

Assume[2] that the computer has unlimited amount of sheets of paper and pencils, a desk with arbitrary large amount of drawers, labeled with some identifiers (*e.g.* positive integers), and two pigeon-holes labeled "R" and "C". The computer lives as long as necessary to perform the computation, is extremely patient and does not make mistakes.

First the computer is handed a pile of sheets: one with instructions describing the procedure of computation to be performed, which she places in pigeon-hole "C", and the rest – if any – with arguments, one number per sheet, which she places in pigeon-hole "R".

The act of performing computation consists of steps. At each step the computer reads the first unstroken instruction from topmost sheet from "C", strikes it out, and follows it. If there are no unstroken instructions, the computer throws the sheet out (and continues with next step, if there are any sheets left in "C"). Instructions always have one of the following forms:

a. Write number n on new sheet and place on top of "R".
b. Write instructions $i_1, ..., i_k$ $(k \geq 0)$ on new sheet and place on top of "R".
c. Take two topmost sheets from "R", calculate sum/product/difference/...[3] of numbers they carry, and replace them with new sheet carrying the result of calculation.
d. Take topmost sheet from "R" and place it in a drawer with label l.
e. Look at topmost sheet in drawer labeled l, rewrite its content on new sheet and place it on top of "R".
f. Throw out first sheet from drawer labeled l.
g. Take topmost sheet from "R", read its content and throw it out. If the content was not number 0, then write instructions $i_1, ..., i_k$ $(k \geq 0)$ on new sheet and place on top of "C", otherwise write instructions $j_1, ..., j_m$ $(m \geq 0)$ on new sheet and place on top of "C".
h. Take topmost sheet from "R" and place it on top of "C".

[2]For justification of such physically impossible conditions *cf.* [2] ch.3 or [17].

[3]The set of basic operations is mostly matter of arbitrary choice - the only requisite is that the computer knows how to calculate their values at any valid arguments.

Computation stops if there are no more instructions in "C", or if some instruction is impossible to perform (*e.g.* topmost sheet in "R" contains instructions, while the computer is instructed to perform multiplication). In the first case we say the computation succeeded, and the content of topmost sheet in "R" is its result. In the second case we say the computation failed. Notice that not every computation has to stop after a finite number of steps (*cf.* end of section 1.5).

To instruct the computer to calculate sum of squares of numbers 2 and 3 with the procedure from example 1, the first sheet could contain the following:

1.	Take topmost sheet from "R" and place it in a drawer with label 1.
2.	Take topmost sheet from "R" and place it in a drawer with label 2.
3.	Look at topmost sheet in drawer labeled 1, rewrite its content on new sheet and place it on top of "R".
4.	Look at topmost sheet in drawer labeled 1, rewrite its content on new sheet and place it on top of "R".
5.	Take two topmost sheets from "R", calculate product of numbers they carry, and replace them with new sheet carrying the result of calculation.
6.	Look at topmost sheet in drawer labeled 2, rewrite its content on new sheet and place it on top of "R".
7.	Look at topmost sheet in drawer labeled 2, rewrite its content on new sheet and place it on top of "R".
8.	Take two topmost sheets from "R", calculate product of numbers they carry, and replace them with new sheet carrying the result of calculation.
9.	Take two topmost sheets from "R", calculate sum of numbers they carry, and replace them with new sheet carrying the result of calculation.

The second sheets would have to contain number 2, and the third number 3. Then all three sheets, in the order given, must be handed to the computer.

Throughout this section we will describe "histories of calculations" in form of tables where each row represents the next state of desk. Drawer with label l is denoted with D_l. The following sheets' contents, in left-to-right order (i.e. leftmost number is the content of topmost sheet), are separated with commas. Only the numbers of instructions are written. The result is indicated with **boldface**.

The first computation runs as follows:

"C"	"R"	D_1	D_2
1 2 3 4 5 6 7 8 9	2,3	–	–
2 3 4 5 6 7 8 9	3	2	–
3 4 5 6 7 8 9	–	2	3
4 5 6 7 8 9	2	2	3
5 6 7 8 9	2,2	2	3
6 7 8 9	4	2	3
7 8 9	3,4	2	3
8 9	3,3,4	2	3
9	9,4	2	3
–	**13**	2	3

There could also be two more instructions added at the end, to empty drawers D_1 and D_2, but in case of simple calculations this is not necessary.

Before moving on with more complicated examples, we introduce a shorthand notation for instructions:

a. CONST n,

b. PROC n,

c. ADD/MUL/SUB/...[4],

d. NAME l,

e. LOOKUP l,

f. FORGET l,

g. SELECT $\langle i_1, \ldots, i_k \rangle$ OR $\langle j_1, \ldots, j_m \rangle$,

h. APPLY.

Now, the same sequence of instructions (for computing sum of squares...) can be written[5] as:

```
1. | NAME 1,
2. | NAME 2,
3. | LOOKUP 1,
4. | LOOKUP 1,
5. | MUL,
6. | LOOKUP 2,
7. | LOOKUP 2,
8. | MUL,
9. | ADD.
```

Such shorthand descriptions will be refereed to as **$\mathfrak{S}$-descriptions**.

We will now provide $\mathfrak{S}$-descriptions of the remaining two examples from previous section.

To compute absolute value of given number x as in example 2, we could hand the computer two sheets: second one with number x, and first one with the following instructions:

```
1. | NAME 1,
2. | LOOKUP 1,
3. | CONST 0,
4. | GT,
5. | SELECT ⟨
a. |          LOOKUP 1,
b. |          CONST -1,
c. |          MUL
   | ⟩ OR ⟨
d. |          LOOKUP 1
   | ⟩.
```

[4]Some more operations are introduced in the following, described as they appear.

[5]Instruction numbers are not part of this notation, we only add them for convenience.

Here, "GT" stands for operation encoding the "greater than" relation - it takes two numbers, compares them, and produces 1 if the first is greater than the second, and 0 otherwise[6].

In case of non-commutative operations we decided to use the order of operands at which they appear in "R" pile - which implies the need for placing them in the *reverse* order[7].

If the second sheet contained number -3, the computation would run:

"C"	"R"	D_1
1 2 3 4 5	-3	–
2 3 4 5	–	-3
3 4 5	-3	-3
4 5	0, -3	-3
5	1	-3
a b c	–	-3
b c	-3	-3
c	-1, -3	-3
–	**3**	-3

The last example, describing procedure for computing factorial of given natural number (as in example 3), seems especially interesting as it includes embedded computations, and more careful management of drawer's contents.

```
1.  | PROC ⟨
a.  |         NAME 2,
b.  |         LOOKUP 2,
c.  |         CONST 0,
d.  |         EQ,
e.  |         SELECT ⟨
e1. |                 CONST 1
    |         ⟩ OR ⟨
e2. |                 CONST 1,
e3. |                 LOOKUP 2,
e4. |                 SUB,
e5. |                 LOOKUP 1,
e6. |                 APPLY,
e7. |                 LOOKUP 2,
e8. |                 MUL
    |         ⟩,
f.  |         FORGET 2.
    | ⟩,
2.  | NAME 1,
3.  | LOOKUP 1,
4.  | APPLY.
```

[6]This will be our standard way of encoding relations as operations - by picking two elements from universe of algebra we are computing in, for representing values of truth and false.

[7]*E.g.* the sequence *"CONST* 2*, CONST* 3*, SUB"* instructs the computer to calculate the expression "$3 - 2$".

Here, “EQ” stands for operation encoding the number identity relation (takes two numbers, compares them and produces 1 if they equal, and 0 otherwise).

Consider history of computation with the second sheet containing 2:

“C”	“R”	D_1	D_2
1 2 3 4	2	–	–
2 3 4	a b c d e f, 2	–	–
3 4	2	a ... f	–
4	a b c d e f, 2	a ... f	–
a b c d e f	2	a ... f	–
b c d e f	–	a ... f	2
c d e f	2	a ... f	2
d e f	0, 2	a ... f	2
e f	0	a ... f	2
e2 e3 e4 e5 e6 e7 e8, f	–	a ... f	2
e3 e4 e5 e6 e7 e8, f	1	a ... f	2
e4 e5 e6 e7 e8, f	2, 1	a ... f	2
e5 e6 e7 e8, f	1	a ... f	2
e6 e7 e8, f	a b c d e f, 1	a ... f	2
a b c d e f, e7 e8, f	1	a ... f	2
b c d e f, e7 e8, f	–	a ... f	1, 2
c d e f, e7 e8, f	1	a ... f	1, 2
d e f, e7 e8, f	0, 1	a ... f	1, 2
e f, e7 e8, f	0	a ... f	1, 2
e2 e3 e4 e5 e6 e7 e8, f, e7 e8, f	–	a ... f	1, 2
e3 e4 e5 e6 e7 e8, f, e7 e8, f	1	a ... f	1, 2
e4 e5 e6 e7 e8, f, e7 e8, f	1, 1	a ... f	1, 2
e5 e6 e7 e8, f, e7 e8, f	0	a ... f	1, 2
e6 e7 e8, f, e7 e8, f	a b c d e f, 0	a ... f	1, 2
a b c d e f, e7 e8, f, e7 e8, f	0	a ... f	1, 2
b c d e f, e7 e8, f, e7 e8, f	–	a ... f	0, 1, 2
c d e f, e7 e8, f, e7 e8, f	0	a ... f	0, 1, 2
d e f, e7 e8, f, e7 e8, f	0, 0	a ... f	0, 1, 2
e f, e7 e8, f, e7 e8, f	1	a ... f	0, 1, 2
e1, f, e7 e8, f, e7 e8, f	–	a ... f	0, 1, 2
f, e7 e8, f, e7 e8, f	1	a ... f	0, 1, 2
e7 e8, f, e7 e8, f	1	a ... f	1, 2
e8, f, e7 e8, f	1, 1	a ... f	1, 2
f, e7 e8, f	1	a ... f	1, 2
e7 e8, f	1	a ... f	2
e8, f	2, 1	a ... f	2
f	2	a ... f	2
–	**2**	a ... f	–

We see that each time the computer performs instruction (e6), she starts a sub-computation of factorial of current argument's predecessor. At every such embedded computations, she performs (a), *i.e.* places the new argument in D_2. After reverting from the sub-computation, she performs (e7) *i.e.* rewrites number from topmost D_2 sheet. She needs that number to be the argument of the computation she had reverted to, not of the one she had just left. Therefore, before leaving each sub-computation, she must "clean up" D_2, *i.e.* throw out topmost sheet from it (f).

In general, each sub-computation builds its own *context* (on tops of some of the drawers), which has to be forgotten before reverting to computation which started it.

These issues should become obvious when we present $drcz0 \to DRC\langle\mathcal{SE}\rangle$ compiler in section 2.3.

1.3 The strength and limitations of $\mathfrak{S}$.

So far the very existence of procedures was established, and we proposed unambiguous ways of describing some of them with sequences of instructions. It seems natural to ask, whether all procedures could be described this way? However, such questions cannot be answered without specifying the notion of "all procedures", which gives rise to another problem - if some rigorous, formal definition of procedure is proposed, how can one be sure its extension contains all objects there were to capture? The story of the problem is described in details in [19] and [1]. If one does not want to assume Church's thesis, a plausible solution would be to resort to weaker formulation of the question: could we describe this way all procedures for computing *(partial) recursive functions* (over *e.g.* non-negative integers)?

And this question can be answered, affirmatively.

Following [2] and [8] we use Kleene's definition of (partial) recursive arithmetical[8] functions (often called μ-recursive functions):

Definition 1. *We call an n-ary (partial) function* $f : \mathbb{N}^n \to \mathbb{N}$ **μ-recursive** *if we can express it as a:*

1. *constant function* $f(x_1, \ldots, x_n) = k$ *for some* $k \in \mathbb{N}$, *or*
2. *successor function* $f(x) = x + 1$, *or*
3. *projection* $f(x_1, \ldots, x_n) = x_i$ *for some* $1 \leq i \leq n$, *or*
4. *composition* $f(x_1, \ldots, x_n) = h(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))$ *for some m-ary* μ*-recursive function* h *and some* m *n-ary* $\mu-$*recursive functions* $g_1, \ldots, g_m$, *or*

[8] It seems that this definition could be generalised to wider class of structures, however we shall not be interested in generalising classical recursion theory this time.

5. *function definable by primitive recursion,* i.e. $f(x_0, x_1, \ldots, x_n)$ *such that:*
   - $n \geq 0$, *and*
   - $f(0, x_1, \ldots, x_n) = g(x_1, \ldots, x_n)$, *and*
   - $f(x_0 + 1, x_1, \ldots, x_n) = h(x_0, f(x_0, x_1, \ldots, x_n), x_1, \ldots, x_n)$

   for some n*-ary* μ*-recursive function* g *and* $(n+2)$*-ary* μ*-recursive function* h, *or*

6. *function defined by minimisation operator* $f(x_1, \ldots, x_n) = \mu(h)(x_1, \ldots, x_n)$ *for some* μ*-recursive* $(n+1)$*-ary function* h,
   where $\mu(h)(x_1, \ldots, x_n) = k$ *iff* $h(k, x_1, ..., x_n) = 0$ *and for all* $m < k$ $h(k, x_1, ..., x_n)$ *is defined and* $h(k, x_1, ..., x_n) \neq 0$.

We will now sketch the proof of the following:

Theorem 1. *For any* μ*-recursive function* f *we can provide* $\mathfrak{S}$*-description for procedure of computing value of* f *at any arguments from* $Dom(f)$*, consisting only of instructions of the following forms: CONST* n*, ADD, SUB, EQ, NAME/FORGET/LOOKUP* n*, APPLY, and SELECT* $\langle \ldots \rangle$ *OR* $\langle \ldots \rangle$.

Proof. By induction on cases:

1) Assume f can be expressed as $f(x_1, \ldots, x_n) = k$.
   Then the $\mathfrak{S}$-description for computing values of f takes the form:

   NAME 1,
   ...,
   NAME n,
   CONST k,
   FORGET 1,
   ...,
   FORGET n.

2) Assume f can be expressed as $f(x) = x + 1$.
   Then the $\mathfrak{S}$-description for f takes the form:

   CONST 1,
   ADD,

3) Assume f can be expressed as $f(x_1, \ldots, x_n) = x_i$.
   Then the $\mathfrak{S}$-description for f takes the form:

   NAME 1,
   ... ,
   NAME n,
   LOOKUP i,
   FORGET 1,
   ... ,
   FORGET n.

4) Assume f can be expressed as $f(x_1, \ldots, x_n) = h(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots x_n))$, and that symbols $\pi_h, \pi_{g_1}, \ldots, \pi_{g_m}$ stand for $\mathfrak{S}$-descriptions of procedures to calculate functions $h, g_1, \ldots, g_m$ resp. (which exist by induction hypothesis).

Then the $\mathfrak{S}$-description for f takes the form:

NAME 1,
$\ldots$,
NAME n,
PROC $\langle \pi_h \rangle$
NAME $n+1$,
PROC $\langle \pi_{g_1} \rangle$
NAME $n+2$,
$\ldots$,
PROC $\langle \pi_{g_m} \rangle$,
NAME $n+m+1$,
LOOKUP 1,
$\ldots$,
LOOKUP n,
LOOKUP $n+m+1$,
APPLY,
$\ldots$,
LOOKUP 1,
$\ldots$,
LOOKUP n,
LOOKUP $n+2$,
APPLY,
LOOKUP $n+1$,
APPLY,
FORGET 1,
$\ldots$,
FORGET n,
FORGET $n+1$,
FORGET $n+2$,
$\ldots$,
FORGET $n+m+1$.

5) Assume f satisfies:

$$f(x_0, \ldots, x_n) = \begin{cases} g(x_1, \ldots, x_n) \ , & x_0 = 0 \\ h(x_0 - 1, f(x_0 - 1, x_1, \ldots, x_n), x_1, \ldots, x_n) \ , & x_0 > 0 \end{cases}$$

and that symbols π_g and π_h stand for $\mathfrak{S}$-descriptions of procedures to calculate functions g and h resp. (which, again, exist by the induction hypothesis).

Then the $\mathfrak{S}$-description for f takes the form:

```
PROC ⟨π_g⟩,
NAME n + 2,
PROC ⟨π_h⟩,
NAME n + 3,
PROC ⟨
        NAME 1,
        ... ,
        NAME n + 1,
        LOOKUP 1,
        CONST 0,
        EQ,
        SELECT ⟨
                LOOKUP n + 1,
                ...,
                LOOKUP 2,
                LOOKUP n + 2,
                APPLY,
                ⟩ OR ⟨
                LOOKUP n + 1,
                ...,
                LOOKUP 2,
                LOOKUP n + 1,
                ...,
                LOOKUP 2,
                CONST 1,
                LOOKUP 1,
                SUB,
                LOOKUP n + 4,
                APPLY,
                LOOKUP n + 3,
                APPLY
                ⟩ ,
        FORGET 1,
        ... ,
        FORGET n + 1,
        ⟩ ,
NAME n + 4,
LOOKUP n + 4,
APPLY,
FORGET n + 2,
FORGET n + 3,
FORGET n + 4.
```

6) Assume f can be expressed as $f(x_1, \ldots, x_n) = \mu(h)(x_1, \ldots, x_n)$, and that symbol π_h stands for $\mathfrak{S}$-description of procedures to calculate function h (exists by induction hypothesis).

Then the $\mathfrak{S}$-description for f takes the form:

```
PROC ⟨π_h⟩,
NAME n + 2,
PROC ⟨
        NAME 1,
        . . . ,
        NAME n + 1,
        LOOKUP n + 1,
        . . . ,
        LOOKUP 1,
        LOOKUP n + 2,
        APPLY,
        CONST 0,
        EQ,
        SELECT ⟨
                LOOKUP 1,
                ⟩ OR ⟨
                LOOKUP n + 1,
                . . .,
                LOOKUP 2,
                CONST 1,
                LOOKUP 1,
                ADD,
                LOOKUP n + 3,
                APPLY
                ⟩,
        FORGET 1,
        . . . ,
        FORGET n + 1
        ⟩, NAME n + 3,
CONST 0,
LOOKUP n + 3,
APPLY,
FORGET n + 2,
FORGET n + 3.
```

□

Notice that not every $\mathfrak{S}$-description captures a procedure of computing any function. A trivial example would be the following "infinite loop procedure":

```
1.  | PROC ⟨
a.  |         LOOKUP 1,
b.  |         APPLY
    |         ⟩,
2.  | NAME 1,
3.  | LOOKUP 1,
4.  | APPLY.
```

This construct we will use in proof of the following theorem, which we include in this section mostly to shows that not every arithmetical function has $\mathfrak{S}$-description of computing it.

It is a consequence of well-known fact that the halting problem is not decidable – we have "translated" (from Turing machines to $\mathfrak{S}$ model) proof of Turing-uncomputability of halting function from [2] (thm. 4.2).

We need to assume that we can encode sequences of integers and sequences of instructions as a single integer – this can be easily done with some variant of gödelization, which we do not discuss here[9], as in section 1.4 we provide other solution.

For now however it would be convenient to assume that such encoding (and decoding as well) is possible. For this single section[10] we will denote encoding of sequence of instructions d as $\ulcorner \mathtt{d} \urcorner$, and of sequence of numbers $\langle x_1, \ldots, x_n \rangle$ $(n \geq 0)$ as $\ulcorner \langle \mathtt{x_1}, \ldots, \mathtt{x_n} \rangle \urcorner$.

Each computation in $\mathfrak{S}$ is determined[11] by the $\mathfrak{S}$-description and its arguments. We say that the computation [described with] d *halts on arguments* $x_1, \ldots, x_n$ if the computer after receiving sheets with d, x_1, ..., x_n performs a computation that succeeds, *i.e.* stops after finite number of steps.

Definition 2. *The* **halting function for** $\mathfrak{S}$ *is a (most often partial*[12]*) function* $h : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ *such that for all* $\mathfrak{S}$*-descriptions* d *and any numbers* $x_1, \ldots, x_n$, $n \geq 0$*:*

$$h(\ulcorner \mathtt{d} \urcorner, \ulcorner \langle \mathtt{x_1}, \ldots, \mathtt{x_n} \rangle \urcorner) = \begin{cases} 1\ , & \text{if } d \text{ halts on } x_1, \ldots, x_n \\ 0\ , & \neg \end{cases}$$

Theorem 2. *Let* h *be the halting function for* $\mathfrak{S}$*. Then there does not exist* $\mathfrak{S}$*-description of computing values of* h *at any argument from* $Dom(h)$*.*

Proof. Reductio ad absurdum.

Assume that such $\mathfrak{S}$-description exists, call it d_H. Consider then the following $\mathfrak{S}$-description d_0 – notice it does not take any input, includes d_H as it sub-procedure (3), and (which might seem hairy at first, but is not) its own Gödel number as constant (6):

```
1.  | PROC ⟨
a.  |         LOOKUP 1,
b.  |         APPLY
    |         ⟩,
2.  | NAME 1,
3.  | PROC d_H,
4.  | NAME 2,
5.  | CONST ⌜⟨⟩⌝
6.  | CONST ⌜d_0⌝,
7.  | LOOKUP 2,
8.  | APPLY,
    | (verte)
```

[9] Interested reader will find it – often under the name of *arithmetization*, or *Gödel numbers* – in *e.g.* [8] ch. III p.3, or [2] ch.15.

[10] From sec.1.4 onwards the square quotes are reserved for other, yet similar, purpose.

[11] *cf.* section 1.5.

[12] Halting function could be partial, as the encoding function does not have to be surjective. It only must be defined on all "meaningful" arguments.

```
9.  | SELECT ⟨
c.  |            LOOKUP 1,
d.  |            APPLY
    | ⟩ OR ⟨
e.  |            CONST 997
    | ⟩.
```

In short it instructs to do the following: introduce "halting procedure" and "infinite loop procedure", name their $\mathfrak{S}$-descriptions with 2 and 1 resp. (1-4), then use this "halting procedure" to decide whether d_0 stops (on empty tuple of arguments) or not (5-8). Next it comes to decide what to do (9) – if the result of "halting procedure" application to d_0 and $\langle\rangle$ was 0, then d_0 stops, with result 997[13], and otherwise it falls into infinite loop.

The contradiction follows easily: if d_H exists, then d_0 is valid $\mathfrak{S}$-description of procedure of computing some function with no arguments (a constant) – it halts (with result 997) iff the result of d_H applied to it is 0, which happens iff d_0 does not halt.

□

1.4 S-expressions.

We could stick with computations over integers and use *e.g.* gödelization to encode other entities of interest (sequences of numbers, functions, formulas, etc.) – however such an uniform treatment seems quite costly, as representing even moderately complicated objects would require using inconveniently large numbers. Therefore we take another route, moving from integers to symbolic expressions (*S-expressions*) – objects able of encoding almost every finite structure in concise and easily representable form, proposed by McCarthy in his deservedly famous [13].

We present our generalization of McCarthy's *S-expressions* as an algebra (algebraic system). The construction might seem overly general, however it is intended to exhibit the fact that *S-expressions* can be arbitrarily extended with many sorts of atomic objects, which is the case in many LISP variants.

Let $\mathcal{A} = \langle A, o_1^{(a_1)}, ..., o_n^{(a_n)} \rangle$ be an algebra[14] with n operations, of arities $a_1, \ldots, a_n$.

To encode truth-values one either picks two elements of A (if $\mathcal{A}$ was field, one could use, as before, its 1 and 0 for encoding truth and false resp.), or extends $\mathcal{A}$'s universe with two new elements $a_\top$ and $a_\bot$, different from all others, over which no operation of $\mathcal{A}$ is defined (in either case we will refer to this algebra as A, and use names $a_\top$ and $a_\bot$ in context of truth-values).

[13] We decided it would be natural for d_0 to end with *some* result, rather than ending with empty "R". This one happens to be the police emergency phone number in Poland.

[14] For general relational systems it suffices to convert them to algebras, by encoding relations as operations, as was already discussed.

Definition 3. *The* **class of S-expressions over the set** A *is the smallest class containing* A *and closed under taking ordered pairs*[15] , i.e.

$$SE(A) = \bigcap \{X : A \subseteq X \wedge \forall_{x,y \in X} \langle x, y \rangle \in X\}$$

Atoms *of* $SE(A)$ *are elements of* A *(including foremention ed* $a_\bot$ *and* $a_\top$*).*

To endow $SE(A)$ with algebraic structure $SE(\mathcal{A})$ simply treat each operation from $\mathcal{A}$ as operation on $SE(\mathcal{A})$ defined on tuples of atoms, and undefined ($\bot$) otherwise, and add five new operations:

$$cons(x, y) = \langle x, y \rangle$$

$$car(z) = \begin{cases} x\ , & z = \langle x, y \rangle \\ \bot\ , & z \in A \end{cases}$$

$$cdr(z) = \begin{cases} y\ , & z = \langle x, y \rangle \\ \bot\ , & z \in A \end{cases}$$

$$atom(x) = \begin{cases} a_\top\ , & x \in A \\ a_\bot\ , & \neg \end{cases}$$

$$eq(x, y) = \begin{cases} a_\top\ , & x, y \in A \wedge x = y \\ a_\bot\ , & \neg \end{cases}$$

Identifiers *cons*, *car* and *cdr* are sort of LISP tradition, their set-theoretic names would be *pair*, *predecessor* and *successor* respectively. Notice that *atom* and *eq* encode predicates of "being an atom", and of atoms identity, respectively.

Definition 4. *For an arbitrary algebra* $\mathcal{A} = \langle A, o_1^{(a_1)}, ..., o_n^{(a_n)} \rangle$, **an algebra of S-expressions over** $\mathcal{A}$ *is the system*

$$SE(\mathcal{A}) = \langle SE(A), o_1^{a_1}, \ldots o_n^{a_n}, cons, car, cdr, atom, eq \rangle.$$

Now an algebra $\mathcal{SE}$ of *S-expressions over numbers and identifiers* can be defined:

Definition 5. *Let* $\mathcal{S} = \langle S, +, -, \times, <, num \rangle$*, such that* S *is disjoint union of set of integers* $\mathbb{Z}$ *and enumerable set of pairwise distinct identifiers* $\{\xi_n\}_{n \in \mathbb{N}}$*.* $+, -, \times$ *are binary operations of addition, multiplication and subtraction, undefined on identifiers,* $<$ *encodes binary relation "greater than" on pairs of numbers,* num *encodes unary relation of being number, and the first two identifiers* ξ_1 *and* ξ_0 *represent values of truth and false. Then* $\mathcal{SE} = SE(\mathcal{S})$*.*

Similarly to [13], we use simplified textual representations for elements of $\mathcal{SE}$, written here `typewritten` and enclosed in square quotes $\ulcorner \ldots \urcorner$. In case of meta-expressions[16] we use Greek letters as placeholders, standing for either arbitrary, or defined elsewhere *S-expression*). Since the representation schema is

[15] In "typeless" set theories like ZF(C) one has to guarantee distinction between elements of A and their pairs, *e.g.* by representing the former as ordered pairs whose predecessor is some set not in A.

[16] Not to be confused with McCarthy's *M-expressions*, which we do not mention – meta-expressions are just a kind of regular expressions.

not unique, to express the fact that $e \in \mathcal{SE}$ is represented with ε we write "$e \sim \varepsilon$" rather than "$e = \varepsilon$".

We use Arabic numerals (with sign) for representing numbers, and literals (*i.e.* sequences of alphanumeric and special characters +, −, ∗, /, =, <, >, :, ., ,, ;, ^, @, #, !, such that the first character is not a digit) for identifiers; two distinguished literals *nil* and T represent ξ_0 and ξ_1 respectively. A string $\ulcorner(\varepsilon_1.\varepsilon_2)\urcorner$ represents an ordered pair of expressions represented with ε_1 and ε_2 (*i.e.* the result of $cons(\varepsilon_1, \varepsilon_2)$).

It is common practice to encode sequences (ordered lists) with nested pairs, *e.g.* the *S-expression* $\langle 1, \langle 2, \langle 3, \xi_0\rangle\rangle\rangle$ stands for triple $\langle 1, 2, 3\rangle$[17]. For convenient representations of such constructions, McCarthy introduced the following convention:

a) $\ulcorner()\urcorner$ stands for (the same $\mathcal{SE}$ as) $\ulcorner\mathtt{nil}\urcorner$,

b) $\ulcorner(\varepsilon)\urcorner$ stands for $\ulcorner(\varepsilon.\mathtt{nil})\urcorner$,

c) $\ulcorner(\varepsilon_1\,\varepsilon_2 \ldots \varepsilon_n)\urcorner$ stands for $\ulcorner(\varepsilon_1.(\varepsilon_2.(\ldots.(\varepsilon_n.\mathtt{nil})\ldots)))\urcorner$, and

d) $\ulcorner(\varepsilon_1\,\varepsilon_2 \ldots \varepsilon_n\,.\,\varepsilon_0)\urcorner$ stands for $\ulcorner(\varepsilon_1.(\varepsilon_2.(\ldots.(\varepsilon_n.\varepsilon_0)\ldots)))\urcorner$.

In the rest of this thesis we prefer McCarthy's notation[18].

In addition to McCarthy's conventions we sometimes use the concatenation operator[19] :: as a shorthand for otherwise complicated expressions – the two representations: $\ulcorner(\varepsilon_1 \ldots \varepsilon_n) :: (\varepsilon'_1 \ldots \varepsilon'_m)\urcorner$ and $\ulcorner(\varepsilon_1 \ldots \varepsilon_n\ \varepsilon'_1 \ldots \varepsilon'_m)\urcorner$ stand for the same *S-expression*.

To describe procedures of computing over $\mathcal{SE}$, it suffices to have instruction for each of $\mathcal{SE}$'s operations (+, −, ×, <, *num*, *atom*, *eq*, *car*, *cdr*, *cons*):

c'. ADD / SUB / MUL / GT / NUM / ATOM / EQ / CAR / CDR / CONS,

and to modify computer's interpretation of "SELECT" instruction:

g'. Take topmost sheet from "R", read its content and throw it out. If the content was not $\ulcorner()\urcorner$, write instructions $i_1, ..., i_k$ $(k \geq 0)$ on new sheet and place on top of "C", otherwise write instructions $j_1, ..., j_m$ $(m \geq 0)$ on new sheet and place on top of "C".

In the rest of thesis we will refer to this extended model as $\mathfrak{S}$ also.

As an example consider a procedure to compute length of given [$\mathcal{SE}$ encoding of] list l – if l is empty (*i.e.* encoded as $\ulcorner()\urcorner$), its length is 0, otherwise it is the length of l's tail (*i.e.* *cdr*) plus one.

[17] It seems worth mentioning that when ordered lists are encoded as nested pairs, the operations *car* and *cdr* gain obvious interpretation of *head* and *tail* (or *first* and *rest*) list operators, commonly used in various programming languages.

[18] As does our parser in system we have implemented - *cf.* `c-src/parser.c` on attached CD and in [21].

[19] We will provide its $DRC\langle\mathcal{SE}\rangle$ implementation in section 1.5.

```
PROC ⟨
        NAME 1,
        LOOKUP 1,
        CONST ⌜()⌝,
        EQ,
        SELECT ⟨
                CONST ⌜0⌝
                ⟩ OR ⟨
                LOOKUP 1,
                CDR,
                LOOKUP 2,
                APPLY,
                CONST ⌜1⌝,
                ADD
                ⟩,
        FORGET 1
⟩,
NAME 2,
LOOKUP 2,
APPLY.
```

Of course the described procedure computes only partial function; to make it total, one could replace instructions of comparing content of topmost sheet from D_1 against $\ulcorner()\urcorner$ with test for it being an atom.

1.5 Mechanization of $\mathfrak{S}$.

As elements of $\mathcal{SE}$ domain can encode sequences (and nested sequences as well), they obviously can serve as representations of $\mathfrak{S}$-descriptions, *i.e.* sequences of instructions. We propose the following:

a. $\ulcorner(\texttt{CONST}\ \varepsilon)\urcorner$,

b. $\ulcorner(\texttt{PROC}\ \pi)\urcorner$,

c. $\ulcorner(\texttt{ADD})\urcorner$ / ... / $\ulcorner(\texttt{CONS})\urcorner$,

d. $\ulcorner(\texttt{NAME}\ \eta)\urcorner$,

e. $\ulcorner(\texttt{LOOKUP}\ \eta)\urcorner$,

f. $\ulcorner(\texttt{FORGET}\ \eta)\urcorner$,

g. $\ulcorner(\texttt{SELECT}\ \pi_1\ \pi_2)\urcorner$,

h. $\ulcorner(\texttt{APPLY})\urcorner$,

where $\varepsilon \in \mathcal{SE}$, $\eta \in \mathcal{SE}$ such that $num(\eta) = \ulcorner\texttt{T}\urcorner$, and $\pi, \pi_1, \pi_2 \in \mathcal{SE}$ (already) encode some $\mathfrak{S}$-descriptions.

Moreover, as each instruction has fixed "arity" (*i.e.* is encoded as list of fixed length) we can concatenate them, instead of using list of lists.

Example 5. *The following description of procedure:*

> *NAME 1,*
> *LOOKUP 1,*
> *LOOKUP 1,*
> *MUL,*
> *FORGET 1.*

would be encoded as:

$$\ulcorner(\texttt{NAME 1 LOOKUP 1 LOOKUP 1 MUL FORGET 1})\urcorner.$$

Example 6. *The following description of procedure:*

> *ATOM,*
> *SELECT* ⟨
> *CONST* ⌜(`An atom`)⌝
> ⟩ *OR* ⟨
> *CONST* ⌜(`A cons pair`)⌝
> ⟩.

would be encoded as:

$$\ulcorner(\texttt{ATOM SELECT (CONST (An atom)) (CONST (A cons pair))})\urcorner.$$

Some more examples will be given at the end of this section.

Every computation captured in model $\mathfrak{S}$ can be identified with sequence of *states* – contents of all sheets (allowing for their order) in both pigeon-holes and all drawers. This holds because each state uniquely determines its successor, as the action the computer is going to perform is determined by the topmost sheet from "C", and its outcome by contents of sheets in "R" and drawers. In other words, each step of computation (*i.e.* each computer's action) is state transformation, determined by the preceding state only – a *function* from states to states. We will call it the *step function*, and in context of $\mathfrak{S}$ model – the $\mathfrak{S}$*-step function*.

It seems that all aspects of computations which we were interested in when introducing $\mathfrak{S}$ can be stated in terms of states and their transformations.

We will now take a closer look at the structure of states of $\mathfrak{S}$, in particular the possibility of encoding them.

Each pigeon-hole and drawer contains a pile of zero or more sheets, ordered into FIFO queue (a stack). Stacks can be canonically implemented with lists, and lists can easily be encoded with *S-expressions*, as was shown in section 1.4. Therefore stacks can be implemented in $\mathcal{SE}$ algebra, *e.g.* as follows:

Operation	**Implementation**
push - add element x on top of stack S	stack before: S, stack after: $cons(x, S)$.
pop - remove element from top of stack S	stack before: S, stack after: $cdr(S)$.
top - look up element from top of stack S	element looked up: $car(S)$.
empty? - check if stack S is empty	result: $eq(S, \ulcorner()\urcorner)$.

Each state we can identify with a sequence of stacks $\langle D_1, \ldots, D_n, R, C\rangle$ where $n \in \mathbb{N}$, as there is always only a finite number of drawers used. Such sequence can be represented as a list of lists, which in turn can be encoded as a single element of $\mathcal{SE}$ algebra.

As a consequence, the $\mathfrak{S}$-step function can be also encoded as a partial function $\Rightarrow_{\mathfrak{S}}: \mathcal{SE} \to \mathcal{SE}$. Of course $\Rightarrow_{\mathfrak{S}}$ is effectively computable – we have indicated a method of computing its values in the first list of instructions in section 1.2. It requires some work to provide a $\mathfrak{S}$-description of procedure of computing $\Rightarrow_{\mathfrak{S}}$'s values – we will *generate* such $\mathfrak{S}$-description in section 2.3, and now only define $\Rightarrow_{\mathfrak{S}}$ as a production-rule system.

We are ready to define $DRC\langle\mathcal{SE}\rangle$ machine – a dynamic system, encoding model $\mathfrak{S}$, and capable of being implemented as a computer program for existing hardware (which we did), or as a mechanical device.

For readability we write $\Rightarrow_{\mathfrak{S}}$ in infix notation[20], and assume its signature is $\mathcal{SE}^3 \to \mathcal{SE}^3$ rather than $\mathcal{SE} \to \mathcal{SE}$, so that $\Rightarrow_{\mathfrak{S}}$'s first argument encodes drawers list $(D_1, \ldots, D_n)$, and the two others encode R and C respectively. Converting from $\mathcal{SE}^3 = \mathcal{SE} \times \mathcal{SE} \times \mathcal{SE}$ to $\mathcal{SE}$ and back could get done with the following two functions:

$$\mathcal{SE} \leftarrow \mathcal{SE}^3 : c(\varepsilon_1, \varepsilon_2, \varepsilon_3) = cons(\varepsilon_1, cons(\varepsilon_2, cons(\varepsilon_3, \ulcorner()\urcorner))),$$

$$\mathcal{SE}^3 \leftarrow \mathcal{SE} : d(\varepsilon) = \langle car(\varepsilon), car(cdr(\varepsilon)), car(cdr(cdr(\varepsilon)))\rangle.$$

We also introduce the following shorthands for drawers:

a) δ stands for the sequence $\delta_1, \ldots, \delta_n$,

b) $\delta \oplus [\eta \mapsto \varepsilon]$ stands for $\delta_1, \ldots, \ulcorner(\varepsilon)\urcorner :: \delta_\eta, \ldots, \delta_n$ *(push ε onto δ_η)*,

c) $\delta \ominus [\eta]$ stands for $\delta_1, \ldots, cdr(\delta_\eta), \ldots, \delta_n$ *(pop from δ_η)*,

d) $\delta[\eta]$ stands for $car(\delta_\eta)$ *(top of δ_η)*.

Finally, let

$$List_{\mathcal{SE}} = \bigcap\{X : \ulcorner()\urcorner \in X \wedge \forall_{y\in\mathcal{SE}}\forall_{x\in X} cons(y, x) \in X\},$$

and $Prog_{DRC\langle\mathcal{SE}\rangle}$ be the class of all *S-expressions* encoding $\mathfrak{S}$-descriptions.

[20] *I.e.* $\alpha \Rightarrow_{\mathfrak{S}} \beta$ stands for $\Rightarrow_{\mathfrak{S}} (\alpha) = \beta$.

Definition 6. *The system* $\langle \Sigma, \Rightarrow_{\mathfrak{S}} \rangle$ *is* $DRC\langle \mathcal{SE} \rangle$ **machine**, *with* Σ *being its* **space of all possible states** *if:*

$$\Sigma = \{\langle\langle \delta_1, \ldots, \delta_n \rangle, \rho, \kappa \rangle : \ n \in \mathbb{N} \wedge \delta_1, \ldots, \delta_n, \rho \in List_{\mathcal{SE}} \wedge \kappa \in Prog_{DRC\langle \mathcal{SE} \rangle}\},$$

and $\Rightarrow_{\mathfrak{S}} : \mathcal{SE}^3 \to \mathcal{SE}^3$ *satisfies the following production rules:*

1.	$\langle \delta, \rho, \ulcorner(\texttt{CONST}\ \varepsilon_1)\urcorner :: \kappa \rangle$	$\Rightarrow_{\mathfrak{S}}$	$\langle \delta, \ulcorner(\varepsilon_1)\urcorner :: \rho, \kappa \rangle$,	
2.	$\langle \delta, \rho, \ulcorner(\texttt{PROC}\ \pi_1)\urcorner :: \kappa \rangle$	$\Rightarrow_{\mathfrak{S}}$	$\langle \delta, \ulcorner(\pi_1)\urcorner :: \rho, \kappa \rangle$,	
3.	$\langle \delta, \ulcorner(\varepsilon)\urcorner :: \rho, \ulcorner(\texttt{NAME}\ \eta)\urcorner :: \kappa \rangle$	$\Rightarrow_{\mathfrak{S}}$	$\langle \delta \oplus [\eta \mapsto \varepsilon], \rho, \kappa \rangle$,	
4.	$\langle \delta, \rho, \ulcorner(\texttt{FORGET}\ \eta)\urcorner :: \kappa \rangle$	$\Rightarrow_{\mathfrak{S}}$	$\langle \delta \ominus [\eta], \rho, \kappa \rangle$,	
5.	$\langle \delta, \rho, \ulcorner(\texttt{LOOKUP}\ \eta)\urcorner :: \kappa \rangle$	$\Rightarrow_{\mathfrak{S}}$	$\langle \delta, \ulcorner(\varepsilon)\urcorner :: \rho, \kappa \rangle$,	*where* $\varepsilon \sim \delta[\eta]$,
6.	$\langle \delta, \ulcorner(())\urcorner :: \rho, \ulcorner(\texttt{SELECT}\ \pi_1\ \pi_2)\urcorner :: \kappa \rangle$	$\Rightarrow_{\mathfrak{S}}$	$\langle \delta, \rho, \pi_2 :: \kappa \rangle$,	
7.	$\langle \delta, \ulcorner(\varepsilon)\urcorner :: \rho, \ulcorner(\texttt{SELECT}\ \pi_1\ \pi_2)\urcorner :: \kappa \rangle$	$\Rightarrow_{\mathfrak{S}}$	$\langle \delta, \rho, \pi_1 :: \kappa \rangle$,	*where* $\varepsilon \nsim \ulcorner()\urcorner$,
8.	$\langle \delta, \ulcorner(\pi)\urcorner :: \rho, \ulcorner(\texttt{APPLY})\urcorner :: \kappa \rangle$	$\Rightarrow_{\mathfrak{S}}$	$\langle \delta, \rho, \pi :: \kappa \rangle$,	
9.	$\langle \delta, \ulcorner(\varepsilon_1\ \varepsilon_2)\urcorner :: \rho, \ulcorner(\texttt{CONS})\urcorner :: \kappa \rangle$	$\Rightarrow_{\mathfrak{S}}$	$\langle \delta, \ulcorner(\varepsilon)\urcorner :: \rho, \kappa \rangle$,	*where* $\varepsilon \sim cons(\varepsilon_1, \varepsilon_2)$,
10.	$\langle \delta, \ulcorner(\varepsilon_1)\urcorner :: \rho, \ulcorner(\texttt{CAR})\urcorner :: \kappa \rangle$	$\Rightarrow_{\mathfrak{S}}$	$\langle \delta, \ulcorner(\varepsilon)\urcorner :: \rho, \kappa \rangle$,	*where* $\varepsilon \sim car(\varepsilon_1)$,
11.	$\langle \delta, \ulcorner(\varepsilon_1)\urcorner :: \rho, \ulcorner(\texttt{CDR})\urcorner :: \kappa \rangle$	$\Rightarrow_{\mathfrak{S}}$	$\langle \delta, \ulcorner(\varepsilon)\urcorner :: \rho, \kappa \rangle$,	*where* $\varepsilon \sim cdr(\varepsilon_1)$,
12.	$\langle \delta, \ulcorner(\varepsilon_1)\urcorner :: \rho, \ulcorner(\texttt{ATOM})\urcorner :: \kappa \rangle$	$\Rightarrow_{\mathfrak{S}}$	$\langle \delta, \ulcorner(\varepsilon)\urcorner :: \rho, \kappa \rangle$,	*where* $\varepsilon \sim atom(\varepsilon_1)$,
13.	$\langle \delta, \ulcorner(\varepsilon_1)\urcorner :: \rho, \ulcorner(\texttt{NUM})\urcorner :: \kappa \rangle$	$\Rightarrow_{\mathfrak{S}}$	$\langle \delta, \ulcorner(\varepsilon)\urcorner :: \rho, \kappa \rangle$,	*where* $\varepsilon \sim num(\varepsilon_1)$,
14.	$\langle \delta, \ulcorner(\varepsilon_1\ \varepsilon_2)\urcorner :: \rho, \ulcorner(\texttt{EQ})\urcorner :: \kappa \rangle$	$\Rightarrow_{\mathfrak{S}}$	$\langle \delta, \ulcorner(\varepsilon)\urcorner :: \rho, \kappa \rangle$,	*where* $\varepsilon \sim eq(\varepsilon_1, \varepsilon_2)$,
15.	$\langle \delta, \ulcorner(\varepsilon_1\ \varepsilon_2)\urcorner :: \rho, \ulcorner(\texttt{ADD})\urcorner :: \kappa \rangle$	$\Rightarrow_{\mathfrak{S}}$	$\langle \delta, \ulcorner(\varepsilon)\urcorner :: \rho, \kappa \rangle$,	*where* $\varepsilon \sim \varepsilon_1 + \varepsilon_2$,
16.	$\langle \delta, \ulcorner(\varepsilon_1\ \varepsilon_2)\urcorner :: \rho, \ulcorner(\texttt{SUB})\urcorner :: \kappa \rangle$	$\Rightarrow_{\mathfrak{S}}$	$\langle \delta, \ulcorner(\varepsilon)\urcorner :: \rho, \kappa \rangle$,	*where* $\varepsilon \sim \varepsilon_1 - \varepsilon_2$,
17.	$\langle \delta, \ulcorner(\varepsilon_1\ \varepsilon_2)\urcorner :: \rho, \ulcorner(\texttt{MUL})\urcorner :: \kappa \rangle$	$\Rightarrow_{\mathfrak{S}}$	$\langle \delta, \ulcorner(\varepsilon)\urcorner :: \rho, \kappa \rangle$,	*where* $\varepsilon \sim \varepsilon_1 \times \varepsilon_2$,
18.	$\langle \delta, \ulcorner(\varepsilon_1\ \varepsilon_2)\urcorner :: \rho, \ulcorner(\texttt{GT})\urcorner :: \kappa \rangle$	$\Rightarrow_{\mathfrak{S}}$	$\langle \delta, \ulcorner(\varepsilon)\urcorner :: \rho, \kappa \rangle$,	*where* $\varepsilon \sim gt(\varepsilon_1, \varepsilon_2)$,

for any $\eta \in \{\varepsilon \in \mathcal{SE} : num(\varepsilon) = \ulcorner\texttt{T}\urcorner\}$, $\delta, \rho \in List_{\mathcal{SE}}$, $\kappa, \pi_1, \pi_2 \in Prog_{DRC\langle \mathcal{SE} \rangle}$, *and* $\varepsilon, \varepsilon_1, \varepsilon_2 \in \mathcal{SE}$.

We will now present small example of computation performed on $DRC\langle \mathcal{SE} \rangle$ machine; in general these do not differ from $\mathfrak{S}$'s computations.

Consider the $\mathfrak{S}$-description from example 5. To use it for computing the square of 5 one simply computes the trajectory of $\langle\langle \delta_1 \rangle, \rho, \kappa \rangle \in \Sigma$ such that $\delta_1 \sim \ulcorner()\urcorner$ (initially drawer D_1 is empty), $\rho \sim \ulcorner(\texttt{5})\urcorner$ ("R" pigeon-hole holds single sheet with $\ulcorner\texttt{5}\urcorner$) and $\kappa \sim \ulcorner(\texttt{NAME 1 LOOKUP 1 LOOKUP 1 MUL FORGET 1})\urcorner$ ("C" holds single sheet with $\mathfrak{S}$-description encoding) :

$$\begin{gathered}
\langle\langle \ulcorner()\urcorner \rangle, \ulcorner(\texttt{5})\urcorner, \ulcorner(\texttt{NAME 1 LOOKUP 1 LOOKUP 1 MUL FORGET 1})\urcorner \rangle \\
\Downarrow_{\mathfrak{S}} \textit{(rule 3.)} \\
\langle\langle \ulcorner(\texttt{5})\urcorner \rangle, \ulcorner()\urcorner, \ulcorner(\texttt{LOOKUP 1 LOOKUP 1 MUL FORGET 1})\urcorner \rangle \\
\Downarrow_{\mathfrak{S}} \textit{(rule 5.)} \\
\langle\langle \ulcorner(\texttt{5})\urcorner \rangle, \ulcorner(\texttt{5})\urcorner, \ulcorner(\texttt{LOOKUP 1 MUL FORGET 1})\urcorner \rangle \\
\Downarrow_{\mathfrak{S}} \textit{(rule 5.)} \\
\langle\langle \ulcorner(\texttt{5})\urcorner \rangle, \ulcorner(\texttt{55})\urcorner, \ulcorner(\texttt{MUL FORGET 1})\urcorner \rangle \\
\Downarrow_{\mathfrak{S}} \textit{(rule 17.)} \\
\langle\langle \ulcorner(\texttt{5})\urcorner \rangle, \ulcorner(\texttt{25})\urcorner, \ulcorner(\texttt{FORGET 1})\urcorner \rangle \\
\Downarrow_{\mathfrak{S}} \textit{(rule 4.)} \\
\langle\langle \ulcorner()\urcorner \rangle, \ulcorner(\texttt{25})\urcorner, \ulcorner()\urcorner \rangle.
\end{gathered}$$

The computation ended with result $\ulcorner\texttt{25}\urcorner$.

Of course $DRC\langle\mathcal{SE}\rangle$ machine can be generalized to family of DRC machines operating over almost any algebra $\mathcal{A}$, provided elements of $\mathcal{A}$'s universe can encode descriptions of procedures. The rules for such machine's step function would be, mutatis mutandis, rules 1-8 of definition 6, with rules 9-18 replaced with their $\mathcal{A}$ analogues.

Definition 7. *Let $\Sigma = D \times R \times C$, where:*

a) $D = \mathbb{Z}_+ \to A^*$ denotes family of (partial) functions from drawer's identifiers to sequences of their contents,

*b) $R = A^*$ sequences of intermediate results (*i.e. *"R" pigeon-hole's content), and*

c) $C \subsetneq A$ consist of elements of A encoding descriptions of procedures only.

Then Σ is a **space of all possible states for $DRC\langle\mathcal{A}\rangle$ machine**.

The attached CD contains our implementation of $DRC\langle\mathcal{SE}\rangle$ machine written in C language, with simple memory management, and extended with basic input/output mechanisms (for files and terminal access). We review it shortly in appendix A. The most recent version can be found at [21].

Appendix B describes how to use our simple $DRC\langle\mathcal{SE}\rangle$ emulator working on $DRC\langle\mathcal{SE}\rangle$ machine (its implementation is presented in section 2.3). The emulator displays whole trajectories – reader interested in experimenting with it might find the following examples[21] illuminating:

Example 7. *A procedure for computing length of given list described in section 1.4:*

```
(PROC (NAME l
       LOOKUP l
       CONST ()
       EQ
       SELECT (CONST 0)
              (LOOKUP l
               CDR
               LOOKUP length
               APPLY
               CONST 1
               ADD)
       FORGET l)
 NAME length
 LOOKUP length
 APPLY)
```

E.g. *when given list* ⌜`(a b (1 2 3) c)`⌝ *yields* ⌜`4`⌝.

[21] As the emulator is capable of using any atoms as labels for drawers, in these examples we use literals, which seem more readable. We also allowed ourselves to omit the square quotes.

Example 8. *A procedure for concatenating two given lists:*

```
(PROC (NAME a
       NAME b
       LOOKUP a
       CONST ()
       EQ
       SELECT (LOOKUP b)
              (LOOKUP b
               LOOKUP a
               CDR
               LOOKUP append
               APPLY
               LOOKUP a
               CAR
               CONS)
       FORGET b
       FORGET a)
 NAME append
 LOOKUP append
 APPLY)
```

E.g. *when given lists* ⌜(a b c)⌝ *and* ⌜(d e)⌝ *yield* ⌜(a b c d e)⌝.

And the last one being especially interesting:

Example 9. *A procedure which takes [encoding of] some [unary] procedure p and a list l yields a list of results of applying p to each element of l:*

```
(PROC (NAME p
       NAME l
       LOOKUP l
       CONST ()
       EQ
       SELECT (CONST ())
              (LOOKUP l
               CDR
               LOOKUP p
               LOOKUP map
               APPLY
               LOOKUP l
               CAR
               LOOKUP p
               APPLY
               CONS)
       FORGET l
       FORGET p)
 NAME map
 LOOKUP map
 APPLY)
```

E.g. *when given* ⌜(NAME x LOOKUP x LOOKUP x MUL FORGET x)⌝ *and* ⌜(1 2 3 4)⌝ *yields* ⌜(1 4 9 16)⌝.

1.6 Processes, procedures, and programs.

Notice that for any $DRC\langle\mathcal{A}\rangle$ machine[22], its space of all possible states can be interpreted as a directed graph $\mathcal{G}_\Sigma$, with elements of Σ being vertices, and such that it contains an edge $\langle\sigma_1,\sigma_2\rangle$ iff $\sigma_1 \Rightarrow_{\mathfrak{S}} \sigma_2$.

Definition 8. *For given computational model $\mathcal{M}$, the* **computation space of $\mathcal{M}$** *is a structure*

$$\mathfrak{C}_\mathcal{M} = \langle \mathcal{G}_{\Sigma_\mathcal{M}}, enc^\mathcal{M}(\pi,\alpha), rval^\mathcal{M}(\sigma), \Pi_C^\mathcal{M}(\sigma)\rangle$$

where $enc^\mathcal{M} : Prog_\mathcal{M} \times \mathcal{A}^* \to \Sigma_\mathcal{M}$ is the **encoding function** *for program π and inputs α, which maps π and α into state encoding machine prepared to perform π on α, $rval^\mathcal{M} : \Sigma_\mathcal{M} \to \mathcal{A}$ is the* **result value function**, *decoding the result of computation from terminal states of $\Sigma_\mathcal{M}$, and $\Pi_C^\mathcal{M} : \Sigma_\mathcal{M} \to Prog_\mathcal{M}$ is the* **commands-projection function**, *mapping states from Σ into [fragments of] programs they encode*[23].

E.g. form any $DRC\langle SE\langle\mathcal{A}\rangle\rangle$ machine we have

$$enc^{DRC\langle SE\langle\mathcal{A}\rangle\rangle}(\pi,\alpha) = \langle\langle\rangle,\alpha,\pi\rangle,$$
$$rval^{DRC\langle SE\langle\mathcal{A}\rangle\rangle}(\langle\delta,\rho,\kappa\rangle) = car(\rho),$$
$$\Pi_C^{DRC\langle SE\langle\mathcal{A}\rangle\rangle}(\langle\delta,\rho,\kappa\rangle) = \kappa.$$

Definition 9. *We say that state $\sigma_1 \in \Sigma$ is* **terminal** *iff there is no $\sigma_2 \in \Sigma$ such that $\mathcal{G}_\Sigma$ contains an edge $\langle\sigma_1,\sigma_2\rangle$* – i.e. *when some computation succeeds, its last state is terminal.*

E.g. for $DRC\langle\mathcal{SE}\rangle$ machine σ is terminal iff $\Pi_C^{DRC\langle\mathcal{SE}\rangle}(\sigma) \sim \ulcorner()\urcorner$.

Every non-terminal (and not jammed[24]) state has exactly one edge coming out of it, but can have arbitrary number of edges coming in.

Definition 10. *For any model $\mathcal{M}$, and program $\pi \in Prog_\mathcal{M}$ of n inputs, and fixed class ε^* of n-tuples of elements of $\mathcal{A}_\mathcal{M}$ (representing set of possible inputs), let*

$$\langle\pi|\varepsilon^*\rangle = \{\sigma \in \Sigma_\mathcal{M} : \exists_{\alpha\in\varepsilon^*} enc^\mathcal{M}(\pi,\alpha) \Rightarrow_\mathcal{M}^* \sigma\}$$

where $\Rightarrow_\mathcal{M}^*$ is the reflexive, transitive closure of $\Rightarrow_\mathcal{M}$[25].

We say that π **generates** *$\langle\pi|\varepsilon^*\rangle \subseteq \Sigma_\mathcal{M}$ [of $\mathfrak{C}_\mathcal{M}$]* **on** *class of arguments ε^*.*

Notice that $\sigma \in \langle\pi|\varepsilon^*\rangle$ iff there is a computation of π at some $\alpha \in \varepsilon^*$ which contains (*i.e.* "passes trough") σ. In other words, σ represents "a possible situation (in the course of computing π at some tuple of inputs from ε^*)".

[22] We suspect this is possible for any deterministic, single-threaded computational model; the cases for $drcz_0$ and $FCL^*\langle\mathcal{SE}\rangle$ will be shown in the next chapter.

[23] This can be more complicated on some computational models – however, it captures the demand that the information on the computation's "future behaviours" must be accessible.

[24] The machine "gets jammed" if performing some operation was not possible, *e.g.* when $DRC\langle\mathcal{SE}\rangle$ is asked to perform multiplication of two non-numeric $\mathcal{SE}$s, or to look up empty drawer.

[25] Notice that $\langle\Sigma,\Rightarrow_\mathcal{M}^*\rangle$ is partial order (poset).

Definition 11. *For any computation space* $\mathfrak{C}_{\mathcal{M}}$*:*

1. *A path in* $\mathcal{G}_{\Sigma_{\mathcal{M}}}$ *we call* **a (computational)** $\mathcal{M}$**-process**. *Infinite and cyclic paths we call* **infinite** $\mathcal{M}$**-processes**. *A* **process generated with** $\pi \in Prog_{\mathcal{M}}$ *(of* n *inputs)* **at inputs** $\alpha = \langle a_1, \ldots, a_n \rangle$ *is a path* $\langle \pi | \langle a_1, \ldots, a_n \rangle \rangle$.

2. *A digraph* $\mathcal{P}_\Sigma$ *whose vertices are non-empty subsets of* Σ*, we call* **an** $\mathcal{M}$**-procedure**[26] *if it satisfies the following conditions:*

   a. *If* $\mathcal{P}_\Sigma$ *contains an edge* $\langle v_1, v_2 \rangle$ *then for any* $\sigma_1 \in v_1$ *there is an edge* $\langle \sigma_1, \sigma_2 \rangle$ *in* $\mathcal{G}_\Sigma$ *for some* $\sigma_2 \in v_2$*,*
   i.e. *each edge represents some possible transitions.*

   b. *For every vertice* v_1 *in* $\mathcal{P}_\Sigma$ *if* $\sigma_1 \in v_1$ *and* $\mathcal{G}_\Sigma$ *contain an edge* $\langle \sigma_1, \sigma_2 \rangle$ *then there exists exactly one edge* $\langle v_1, v_2 \rangle$ *in* $\mathcal{P}_\Sigma$ *such that* $v_2 \ni \sigma_2$*,*
   i.e. *each possible transition is represented with some (unique) edge.*

   In case of $DRC\langle\mathcal{A}\rangle$ *machines we prefer*[27] *to require also :*

   c. *For every vertice* v *in* $\mathcal{P}_\Sigma$ *it is the case that* $\Pi_C(\sigma_1) = \Pi_C(\sigma_2)$ *for any* $\sigma_1, \sigma_2 \in v$*,*
   i.e. *each vertice represents states which are "ready to perform the same operations/choices".*

3. **An** $\mathcal{M}-$**program** *is some object of* $\mathcal{A}_{\mathcal{M}}$ *encoding description of* $\mathcal{M}-$*procedure.*

   In case of $DRC\langle\mathcal{A}\rangle$ *machines it is an element of* $\mathcal{A}$ *encoding some* $\mathfrak{S}$*-description*[28]*.*

   We say that program π **implements** n*-ary function* f *if* π *describes some procedure of computing* f*'s values (*i.e. π *takes* n *inputs* $\alpha_1, \ldots, \alpha_n$ *and returns value* $f(\alpha_1, \ldots, \alpha_n)$*).*

   The set of all $\mathcal{M}$*-programs we denote with* $Prog_{\mathcal{M}}$*.*

An abstract machine $\mathcal{M}$ can be seen as "a device which interprets its programs as descriptions of procedures". To precise this intuition we define the two "semantic functions":

Definition 12. *Let* $\mathcal{M}$ *be any abstract machine operating over algebra* $\mathcal{A}$*,* $Prog_{\mathcal{M}}$ *the set of all its possible programs, and* $Proc_{\mathcal{M}}$ *the set of all its possible processes. Then* $\mathcal{M}$ *determines:*

1. *A (total) mapping* $prc_{\mathcal{M}} : Prog_{\mathcal{M}} \to (\mathcal{A}^* \to Proc_{\mathcal{M}})$*, assigning to any program* p *the (partial) function from* p*'s arguments (inputs)* $\alpha_1, \ldots, \alpha_n$ *to process starting at state encoding* p *and* $\alpha_1, \ldots, \alpha_n$*, and*

2. *a (partial) mapping* $[\![.]\!]_{\mathcal{M}} : Prog_{\mathcal{M}} \to (\mathcal{A}^2 \to \mathcal{A})$ *assigning to program* p *its input-output function,* i.e. $f : \mathcal{A}^n \to \mathcal{A}$ *such that* p *encodes description of procedure for computing [values of]* f[29]*.*

[26] We propose definition (2) to express procedures as *generalized processes*, which will be useful in explaining phenomena occurring during abstract interpretation, including the three Futamura projections in chapter 3.

[27] In more general cases we believe this requirement could be stated by first defining certain congruence relation [.] on Σ, and then by requiring that for any $v \in \mathcal{P}_\Sigma$ and any $\sigma \in v$ $[\sigma] \subset v$.

[28] *E.g.* any $\mathcal{SE}$ which can appear on κ stack.

[29] Provided p encodes such description – *cf.* "infinite loop procedure" from section 1.3.

We call $prc_{\mathcal{M}}$ **the lower semantics for** $\mathcal{M}$, $[\![.]\!]_{\mathcal{M}}$ **the upper semantics for** $\mathcal{M}$, *and their values at given* $p \in Prog_{\mathcal{M}}$ **the upper** *and***lower semantics of** p *resp.*

E.g. in the case of $DRC\langle\mathcal{SE}\rangle$ machine, if $\pi \in Prog_{DRC\langle\mathcal{SE}\rangle}$ encodes $\mathfrak{S}$-description of procedure of computing [values of] function $f : \mathcal{SE}^n \to \mathcal{SE}$ then $[\![\pi]\!]_{DRC\langle\mathcal{SE}\rangle} = f$, while $prc_{DRC\langle\mathcal{SE}\rangle}(\pi)(\varepsilon_1, \ldots, \varepsilon_n) = \langle\sigma_1, \ldots, \sigma_k\rangle$, where $\sigma_1 = \langle\gamma, \ulcorner(\varepsilon_1 \ldots \varepsilon_{\mathtt{n}})\urcorner, \pi\rangle$, $\sigma_k = \langle\gamma', \ulcorner(\varepsilon)\urcorner, \ulcorner()\urcorner\rangle$, $\sigma_i \Rightarrow_{\mathfrak{S}} \sigma_{i+1}$ for $i = 1, \ldots, k-1$, and $f(\varepsilon_1 \ldots \varepsilon_n) = \varepsilon$.

2 On programming and programs

The previous chapter prepared the ground level for our investigations – a simple model of computations. We will now move one step up the abstraction ladder and introduce *programming languages* – technique of defining computational models, most often in the context of other computational models. Programming languages provide means of using a given computational model to emulate some other, more desired model – *e.g.* to hide some tedious details, so that the programmer is concerned less (if at all) with her ground computational model (*e.g.* hardware) and more with the problem she aims to solve.

2.1 Programming languages.

Given ground computational model $\mathcal{M}$, a programming language $\mathcal{L}$ can be defined by either a set of notations which can be unambiguously translated into $\mathcal{M}$-programs, or some computational model $\mathcal{M}_{\mathcal{L}}$, which can be emulated in $\mathcal{M}$ (with some $\mathcal{M}$-program). In the former case we speak of *compilation*, in the latter of *interpretation*.

Definition 13. *Let $\mathcal{M}_1$, $\mathcal{M}_2$ and $\mathcal{M}_3$ be any computational models operating over algebra $\mathcal{A}$*[30]*. Then:*

A translation from $\mathcal{M}_1$ to $\mathcal{M}_2$ *is any function $T : Prog_{\mathcal{M}_1} \to Prog_{\mathcal{M}_2}$ such that whenever $[\![p_1]\!]_{\mathcal{M}_1}$ is defined $T(p_1) = p_2$ iff $[\![p_1]\!]_{\mathcal{M}_1} = [\![p_2]\!]_{\mathcal{M}_2}$.*

A compiler from $\mathcal{M}_1$ to $\mathcal{M}_2$ *(in $\mathcal{M}_3$) is any $C \in Prog_{\mathcal{M}_3}$ implementing translation from $\mathcal{M}_1$ to $\mathcal{M}_2$,* i.e. $[\![C]\!]_{\mathcal{M}_3} : Prog_{\mathcal{M}_1} \to Prog_{\mathcal{M}_2}$.

An interpreter of $\mathcal{M}_1$ *(in $\mathcal{M}_2$) is any $I \in Prog_{\mathcal{M}_2}$ implementing $[\![.]\!]_{\mathcal{M}_1}$,* i.e. $[\![I]\!]_{\mathcal{M}_2} : Prog_{\mathcal{M}_1} \times \mathcal{A}^*_{(1)} \to \mathcal{A}_{(1)}$.

An interpreter of $\mathcal{M}_1$ in $\mathcal{M}_1$ we call **a metacircular $\mathcal{M}_1$ interpreter**.

Definitions trough interpreter and compiler are equivalent – having a compiler from $\mathcal{M}_1$ to $\mathcal{M}_2$, an interpreter can be obtained from composing the compiler with metacircular $\mathcal{M}_2$ interpreter (*i.e.* $I_1(p, \alpha) = I_2(C_{1\to 2}(p), \alpha)$), while having $\mathcal{M}_1$ interpreter in $\mathcal{M}_2$ one can generate $\mathcal{M}_1$ to $\mathcal{M}_2$ compiler with second Futamura projection (as will be demonstrated in chapter 3).

An $\mathcal{M}_1$ interpreter $I \in Prog_{\mathcal{M}_2}$ determines the space of all possible states of $\mathcal{M}_1$ – simply

$$\Sigma_{\mathcal{M}_1} = f(\langle I|\mathcal{A}^*\rangle \subseteq \Sigma_{\mathcal{M}_2}),$$

where $f : \langle I|\mathcal{A}^*\rangle \to \Sigma_{\mathcal{M}_1}$ is some monomorphism[31].

Notice that $\langle I|\mathcal{A}^*\rangle$ will almost always be proper subset of $\Sigma_{\mathcal{M}_2}$, even in the metacircular case $\mathcal{M}_1 = \mathcal{M}_2$ – only then it will be contractible to $\Sigma_{\mathcal{M}_2}$ as a digraph (equivalently: monotonicaly embedded in $\Sigma_{\mathcal{M}_2}$ as partial order).

[30] Or, as Kuba Kolecki pointed out, over three algebras $\mathcal{A}_1$, $\mathcal{A}_2$, $\mathcal{A}_3$, such that $\mathcal{A}_1$ is embedded in $\mathcal{A}_2$ and $\mathcal{A}_2$ in $\mathcal{A}_3$.

[31] *I.e.* monomorphism (monotonic function) of spaces of possible states as partial orders $\langle \Sigma, \Rightarrow^* \rangle$. *N.b.* this paragraph is the most important statement of this thesis.

Two languages will be presented: $drcz_0$ – our variation on McCarthy's LISP ([13]), and $FCL\langle\mathcal{SE}\rangle$ – simple imperative language used in [10], [9] and [7] to demonstrate the principles of program transformation method called *partial evaluation*. We present implementations of these languages on $DRC\langle\mathcal{SE}\rangle$ machine (as both compilers and interpreters), examine possibilities of reasoning about programs by proving that the $drcz_0$ to $DRC\langle\mathcal{SE}\rangle$ compiler is correct (*i.e.* preserves upper semantics), and present some applications in symbolic computations.

2.2 Describing procedures with $drcz_0$ and $drcz_1$.

Although $\mathfrak{S}$-descriptions (and, equivalently, their representations – $DRC\langle\mathcal{SE}\rangle$ programs) enable to describe procedures in simple and unambiguous manner, after some writing they get inconvenient, or at least over-detailed – *e.g.* in the case of procedure of computing factorial from example 3 it seems enough (with respect to unambiguity) to describe it with recursive (multicase) equation:

$$fact(n) = \begin{cases} 1\ , & n = 0 \\ n \times fact(n-1)\ , & \neg \end{cases}$$

We would like to introduce unambiguous notations for recursive equation, and rules for evaluating them at given arguments – a programming language. We decide it to capture functions over $\mathcal{SE}$ domain, some possible generalizations are obvious.

In order to construct such language, we have to investigate the (applicative) structure of recursive equations. Consider the general form of definition by recursive equation: $\varphi(\xi_1, \ldots, \xi_n) = \varepsilon$. Its LHS (*definiens*) is defined function's identifier φ, followed by list of identifiers $\xi_1, \ldots, \xi_n$ ($n \geq 0$) for variables it binds, while RHS (*definiendum*) is some expression. Expressions in recursive equations can take one of the following forms:

a. **conditional expressions**, *i.e.*

$$\begin{cases} \varepsilon_1 \text{ if } \varepsilon'_1, \\ \varepsilon_2 \text{ if } \varepsilon'_2, \\ \ldots \\ \varepsilon_m \text{ if } \varepsilon'_m. \end{cases}$$

where $m > 0$ and ε_i, ε'_i for $i = 1, \ldots, m$ are any expressions.

Since the order of "cases" (*i.e.* expressions ε'_i, $i = 1, \ldots, m$ encoding conditions) does not matter, we can rewrite such definitions (while keeping their semantics) by using *triadic conditional expression* "if α then β_t else β_f" :

$$\begin{array}{lll} \varphi(\xi_1, \ldots, \xi_n) = & \text{if } \varepsilon'_1 \text{ then } \varepsilon_1, & \\ & \text{else if } \varepsilon'_2 \text{ then } \varepsilon_2, & \\ & & \vdots \\ \ldots & \text{else if } \varepsilon'_m \text{ then } \varepsilon_m, & \\ & \text{else } \bot. & \end{array}$$

Therefore we can narrow our considerations to triadic conditional expressions.

E.g. the following Ackermann function definition:

$$A(m,n) = \begin{cases} n+1 \ , \ m = 0, \\ A(m-1,1) \ , \ n = 0, \\ A(m-1, A(m,n-1)) \ , \ \neg \end{cases}$$

would be rewritten to:

$$\begin{aligned} A(m,n) = \quad & \text{if } m = 0 \text{ then } n+1, \\ & \text{else} \quad \text{if } n = 0 \text{ then } A(m-1,1), \\ & \qquad \text{else } A(m-1, A(m,n-1)). \end{aligned}$$

b. **application forms** , *i.e.* $\varphi_i(\varepsilon_1, \ldots, \varepsilon_{n_i})$

   where φ_i is some previously defined function of arity n_i, and $\varepsilon_1, \ldots, \varepsilon_{n_i}$ are any expressions.

   Notice that primitive operations can be applied too, *e.g.* we consider expression "$x + 1$" as special case of application ("$+(x, 1)$").

c. **constants**, *e.g.* 1.

d. **variable identifiers**, *i.e.* $\xi_i, i \leq n$, *e.g.* x.

   It only makes sense to use the variables bound by the defined function, whose number we denoted with n.

E.g. the RHS of recursive equation of factorial function has form of triadic conditional expression, consisting of (1) premise-expression "$n = 0$", (2) conclusion-expression "1" and (3) alternative-expression "$n \times fact(n-1)$"; the last one being application form of multiplication operator to two expressions: variable identifier n and application form of function $fact$ to expression being application form of subtraction to two arguments, variable identifier n and constant 1.

To find value of n-ary function f defined with recursive equation

$$f(\xi_1, \ldots, \xi_n) = \varepsilon$$

at given arguments $\alpha_1, \ldots, \alpha_n$ one first creates environment – mapping from variable identifiers to [constant] expressions $\sigma : Id \to \mathcal{SE}$, and then evaluates ε in σ, *i.e.* computes value of ε in accord to the following rules:

a. if ε is a conditional expression "$if\ \alpha\ then\ \beta_\top\ else\ \beta_\bot$" then its value is:

   1.1 the value of $\beta_\top$ expression (in σ), if value of α (in σ) was not $\ulcorner()\urcorner$,

   2.2 the value of $\beta_\bot$ expression (in σ), otherwise.

b. if ε is an application form "$\varphi(\varepsilon_1, \ldots, \varepsilon_n)$", then its value is:

   b.1 if φ is primitive operator identifier, then application of its corresponding operator to values of $\varepsilon_1, \ldots, \varepsilon_n$ (in σ),

   b.2 if φ is defined function identifier, with definition "$\varphi(\xi_1, \ldots, \xi_n) = \varepsilon_\varphi$" then the value of ε_φ is σ' such that $\sigma'[\xi_i] = \varepsilon'_i$ for $i = 1, \ldots, n$, where ε'_i is the value of ε_i (in σ).

c. if ε is a constant, its value is itself (in any σ),

d. if ε is a variable identifier ξ_i, its value is $\sigma[\xi_i]$.

E.g. consider a possible subexpression ε': " $x \times x + y \times y$", in environment such that $\sigma[x] \sim \ulcorner \mathtt{2} \urcorner$ and $\sigma[y] \sim \ulcorner \mathtt{3} \urcorner$: this expression is an application form of + operator, with arguments $\alpha_1 = x \times x$ and $\alpha_2 = y \times y$. Expression α_1 is application form of × operator, with arguments $\alpha_{1,1} = x$ and $\alpha_{1,2} = x$. Since $\alpha_{1,1}$ and $\alpha_{1,2}$ are the same variable identifier, we look up its value in environment σ, receiving $\ulcorner \mathtt{2} \urcorner$; thus we get $\times(\ulcorner \mathtt{2} \urcorner, \ulcorner \mathtt{2} \urcorner)$, which is $\ulcorner \mathtt{4} \urcorner$; similarly for α_2, where we obtain $\ulcorner \mathtt{9} \urcorner$. Therefore the outermost application form is $+(\ulcorner \mathtt{4} \urcorner, \ulcorner \mathtt{9} \urcorner)$, which is $\ulcorner \mathtt{13} \urcorner$ (the value of ε' in σ).

We will now show an encoding of (applicative) expressions of recursive equations as *S-expressions*; let ε be the expression to encode:

a. conditional expression "$if\ \alpha\ then\ \beta_\top\ else\ \beta_\bot$" we encode as $\ulcorner(\mathtt{IF}\ \varepsilon_\alpha\ \varepsilon_{\beta_\top}\ \varepsilon_{\beta_\bot})\urcorner$, where ε_α, $\varepsilon_{\beta_t op}$, and $\varepsilon_{\beta_b ot}$ encode expressions α, $\beta_\top$, and $\beta_\bot$ resp.,

b. application form "$\varphi(\alpha_1, \ldots, \alpha_n)$", we encode as $\ulcorner(\varphi\ \varepsilon_{\alpha_1} \ldots \varepsilon_{\alpha_n})\urcorner$, where ε_{α_i} encode expression α_i for $i = 1, \ldots, n$.

   To shorten notations we use single-character identifiers for operations of $\mathcal{SE}$:

operation	identifier
car	$\ulcorner . \urcorner$
cdr	$\ulcorner , \urcorner$
cons	$\ulcorner ; \urcorner$
num	$\ulcorner \# \urcorner$
atom	$\ulcorner @ \urcorner$
eq	$\ulcorner = \urcorner$
+	$\ulcorner + \urcorner$
×	$\ulcorner * \urcorner$
>	$\ulcorner > \urcorner$

c. numeric constant we encode as itself, while symbolic and compound ones as *quoted expressions*, *i.e.* $\ulcorner(\mathtt{QUOTE}\ \varepsilon)\urcorner$[32],

d. variable identifiers as (arbitrary) symbols.

Moreover, to capture the whole definition we also need means of expressing:

1) *"a functional object"* – binding its arguments' values to its fixed set of names,

2) *"assignment"* – naming some functional object with given variable identifier.

Following McCarthy ([13]) we use *lambda notation* for (1) and *label operator* for (2)[33]:

e1. functional object binding n variables we encode as $\ulcorner(\hat{}\ (\xi_1 \ldots \xi_n)\ \varepsilon)\urcorner$, where ξ_i is a symbol for $i = 1, \ldots, n$ and ε is encoding given objects body,

e2. assignment operation, binding expression ε with variable's identifier ξ_j, we encode as $\ulcorner(!\ \xi_j\ \varepsilon)\urcorner$.

[32] We will write these often as shorthand $\ulcorner'\varepsilon\urcorner$.

[33] This is like paradoxical combinator Y but has uglier, non-compositional (context-dependent) semantics.

E.g. the factorial function would be encoded as

```
(! fact (^ (n)
           (IF (= n 0)
               1
               (* n (fact (- n 1)))) ))
```

We can now formalize the evaluation function of finding value of expression ε in environment σ:

Definition 14. *$drcz_0$ evaluation function is* $Eval : (\mathcal{SE} \times Env) \to (\mathcal{SE} \times Env)$ *such that:*

$$Eval(\varepsilon,\sigma) = \begin{cases} \langle \varepsilon, \sigma\rangle \ , \ num(\varepsilon) \sim \ulcorner \mathtt{T} \urcorner \vee \varepsilon \sim \ulcorner () \urcorner \\ \\ \langle \sigma[\varepsilon], \sigma\rangle \ , \ atom(\varepsilon) \sim \ulcorner \mathtt{T} \urcorner \wedge num(\varepsilon) \nsim \ulcorner \mathtt{T} \urcorner \\ \\ \langle \varepsilon_1, \sigma\rangle \ , \ \varepsilon \sim \ulcorner (\mathtt{QUOTE}\ \varepsilon) \urcorner \\ \\ \langle cons(val(Eval(\varepsilon_1,\sigma)), val(Eval(\varepsilon_2,\sigma))), \sigma\rangle \ , \ \varepsilon \sim \ulcorner (;\ \varepsilon_1\ \varepsilon_2) \urcorner \\ \langle car(val(Eval(\varepsilon_1,\sigma))), \sigma\rangle \ , \ \varepsilon \sim \ulcorner (.\ \varepsilon_1) \urcorner \\ \langle cdr(val(Eval(\varepsilon_1,\sigma))), \sigma\rangle \ , \ \varepsilon \sim \ulcorner (,\ \varepsilon_1) \urcorner \\ \langle atom(val(Eval(\varepsilon_1,\sigma))), \sigma\rangle \ , \ \varepsilon \sim \ulcorner (@\ \varepsilon_1) \urcorner \\ \langle num(val(Eval(\varepsilon_1,\sigma))), \sigma\rangle \ , \ \varepsilon \sim \ulcorner (\#\ \varepsilon_1) \urcorner \\ \langle eq(val(Eval(\varepsilon_1,\sigma)), val(Eval(\varepsilon_2,\sigma))), \sigma\rangle \ , \ \varepsilon \sim \ulcorner (=\ \varepsilon_1\ \varepsilon_2) \urcorner \\ \langle val(Eval(\varepsilon_1,\sigma)) + val(Eval(\varepsilon_2,\sigma)), \sigma\rangle \ , \ \varepsilon \sim \ulcorner (+\ \varepsilon_1\ \varepsilon_2) \urcorner \\ \langle val(Eval(\varepsilon_1,\sigma)) \times val(Eval(\varepsilon_2,\sigma)), \sigma\rangle \ , \ \varepsilon \sim \ulcorner (*\ \varepsilon_1\ \varepsilon_2) \urcorner \\ \langle gt(val(Eval(\varepsilon_1,\sigma)), val(Eval(\varepsilon_2,\sigma))), \sigma\rangle \ , \ \varepsilon \sim \ulcorner (>\ \varepsilon_1\ \varepsilon_2) \urcorner \\ \\ Eval(\beta_\top,\sigma) \ , \ \varepsilon \sim \ulcorner (\mathtt{IF}\ \alpha\ \beta_\top\ \beta_\bot) \urcorner \wedge val(Eval(\alpha,\sigma)) \nsim \ulcorner () \urcorner \\ Eval(\beta_\bot,\sigma) \ , \ \varepsilon \sim \ulcorner (\mathtt{IF}\ \alpha\ \beta_\top\ \beta_\bot) \urcorner \wedge val(Eval(\alpha,\sigma)) \sim \ulcorner () \urcorner \\ \\ Eval(\beta, \sigma \oplus [\alpha_1 \mapsto \varepsilon'_1, \ldots, \alpha_n \mapsto \varepsilon'_n]) \ , \\ \quad \varepsilon \sim \ulcorner (\varepsilon_0\ \varepsilon_1 \ldots \varepsilon_n) \urcorner \wedge \\ \quad \varepsilon'_i = Eval(\varepsilon_i,\sigma)\ i = 0,\ldots,n \wedge \\ \quad \varepsilon'_0 \sim \ulcorner (\hat{}\ (\alpha_1 \ldots \alpha_n)\ \beta) \urcorner \end{cases}$$

Where $\mathcal{SE} \leftarrow (\mathcal{SE} \times Env) : val(\varepsilon,\sigma) = \varepsilon$.

We assume that what *!* operator does is to update some "global environment", seen from the inside of any functional object[34].

We will provide now some examples of procedures which we described for $DRC\langle\mathcal{SE}\rangle$ machine in examples 7,8 and 9 respectively:

Example 10. *A procedure for computing length of given list l:*

```
(! len (^ (l)
          (IF (= l ())
              0
              (+ 1 (len (, l))))))
```

E.g. $\ulcorner$`(len (QUOTE (a b (2 3) c)))`$\urcorner$ *evaluates (in any* σ *mapping* $\ulcorner$`len`$\urcorner$ *to above definition) to* $\ulcorner$`4`$\urcorner$.

[34] *I.e.* $Eval(\ulcorner (!\ \alpha\ \beta) \urcorner, \sigma)$ pushes $\langle \alpha \mapsto val(Eval(\beta,\sigma))\rangle$ binding to the "top σ" – *cf.* `drcz1-src/drcz1-interpreter.drcz1`.

Example 11. *A procedure for concatenating two given lists a and b:*

```
(! apd (^ (a b)
          (IF (= a ())
              b
              (; (. a) (apd (, a) b)))))
```

E.g. ⌜(apd (QUOTE (a b c)) (QUOTE (d e)))⌝ *evaluates to* ⌜(a b c d e)⌝.

Example 12. *A procedure which takes [encoding of] some [unary] procedure p and a list l yields a list of results of applying p to each element of l:*

```
(! map (^ (p l)
          (IF (= l ())
              ()
              (; (p (. l)) (map p (, l))))))
```

E.g. ⌜(map fact (QUOTE (1 2 3 4)))⌝ evaluates (in σ containing definitions of *map* and *fact*) to ⌜(1 2 6 24)⌝.

E.g. ⌜(map (^ (x) (∗ x x)) (QUOTE (1 2 3 4)))⌝ evaluates (in σ containing definition of *map*) to ⌜(1 2 9 16)⌝.

Notice the unconventional way of using functional object (⌜^⌝) in the last example. We gain such freedom because the evaluation function for expressions is capable of evaluating each type of expressions (a.-d.) in *any* context. Similarly we can (and do) use conditional expression in position of procedure's operand, or even in the place of the operator, *e.g.*

```
(^ (a b c)
   ((if (= c 0)
        (^ (x y) (+ x y))
        (^ (x y) (* x y)))
    a b))
```

We call this new notation (capturing recursive equations and more) a $drcz_0$ **programming language**.

All the above examples can be experimented with on our $DRC\langle\mathcal{SE}\rangle$ machine emulator – the program `drcz1-src/drcz1-interpreter.drcz1` on attached CD contains source code of REPL (*i.e.* "Read-Eval-Print-Loop") interpreter for $drcz_1$, which repeatedly reads *S-expression*'s representation from input stream, evaluates it (in its global environment) and outputs representation of its value. It can be compiled to $DRC\langle\mathcal{SE}\rangle$ machine code, and executed in users terminal.

The details of $DRC\langle\mathcal{SE}\rangle$ emulator's usage are described in appendix C.

The freedom of composing expressions enables us *i.a.* building nested definitions, of type "$f(x,y) = sq(x) + sq(y)$ where $sq(z) = z \times z$". *I.e.* we can encode this construct as the following $drcz_0$ expression:

```
(! f (^ (x y)
        ( (^ (x y sq) (+ (sq x) (sq y)))  x y (^ (z) (* z z)) )))
```

We will now introduce a few *macros*, or *syntactic sugar* – shorthand notations for certain compositions of primitive operations: nested definitions ⌜LET⌝, McCarthy's conditionals ⌜?⌝ (*cond* in [13]), ⌜LIST⌝ constructions, and logical connectives of ⌜not⌝, ⌜or⌝, ⌜and⌝, ⌜any⌝, and ⌜all⌝.

1. expression of the form ⌜(LET ((α₁ ε₁) ... (αₙ εₙ)) β)⌝
   abbreviates [expr. of the form] ⌜((^ (α₁ ... αₙ) β) ε₁ ... εₙ)⌝,

   E.g. ⌜(LET ((a 2) (b 3)) (+ a b))⌝ evaluates to ⌜5⌝.

2. expression of the form ⌜(? (ε′₁ ε₁) ... (ε′ₙ εₙ))⌝
   abbreviates ⌜(IF ε′₁ ε₁ (IF ... (IF ε′ₙ εₙ ()) ...))⌝,

   E.g.

```
( (^ (x) (? ((= x ()) (QUOTE nil))
            ((@ x) (QUOTE atom))
            (T (QUOTE cons))))
  5 )
```

   evaluates to ⌜atom⌝.

3. expression of the form ⌜(not ε)⌝
   abbreviates ⌜(IF ε () T)⌝,

4. expression of the form ⌜(or ε₁ ε₂)⌝
   abbreviates ⌜(IF ε₁ T ε₂)⌝,

5. expression of the form ⌜(and ε₁ ε₂)⌝
   abbreviates ⌜(IF ε₁ ε₂ ())⌝,

   E.g. ⌜(IF (and (@ x) (not (# x))) (QUOTE (symbol)) (QUOTE (not symbol)))⌝
   in environment σ with *sigma*[⌜x⌝] ∼ ⌜atom⌝ evaluates to ⌜(symbol)⌝.

6. expression of the form ⌜(any ε₁ ... εₙ)⌝
   abbreviates ⌜(IF ε₁ T (IF ... (IF εₙ T ()) ...))⌝,

7. expression of the form ⌜(all ε₁ ... εₙ)⌝
   abbreviates ⌜(IF ε₁ (IF ... (IF εₙ T ()) ...)())⌝,

8. expression of the form ⌜(list ε₁ ... εₙ)⌝
   abbreviates ⌜(; ε₁ (... (; εₙ ()) ...))⌝.

   E.g. ⌜(list 1 2 (+ 3 4))⌝ evaluates to ⌜(1 2 7)⌝.

We call this notation $drcz_1$ – a *sugared* version of $drcz_0$. A simple compiler from $drcz_1$ to $drcz_0$ is implemented in file `drcz1-src/desugar.drcz1`, while (REPL) interpreter extended with these constructions can be found in `drcz1-src/drcz1-interpreter.drcz1` on the attached CD.

As an example of $drcz_1$ program consider the following[35] definition of procedure of finding values of *Eval* function at any *S-expression* ε and environment σ:

```
(! eval (^ (exp env)
           (? ((any (= exp ()) (= exp 'T) (# exp))
                {=>} (opair exp env))
              ((@ exp) {=>} (opair (lookup exp env) env))
              ((= (. exp) 'quote) {=>} (opair (second exp) env))
              ((= (. exp) ';) {=>} (opair (; (val (eval (second exp) env))
                                             (val (eval (third exp) env)))
                                          env))
              ((= (. exp) '.) {=>} (opair (. (val (eval (second exp) env)))
                                          env))
              ((= (. exp) ',) {=>} (opair (, (val (eval (second exp) env)))
                                          env))
              ((= (. exp) '@) {=>} (opair (@ (val (eval (second exp) env)))
                                          env))
              ((= (. exp) '#) {=>} (opair (# (val (eval (second exp) env))
                                          env)))
              ((= (. exp) '=) {=>} (opair (= (val (eval (second exp) env))
                                             (val (eval (third exp) env)))
                                          env))
              ((= (. exp) '+) {=>} (opair (+ (val (eval (second exp) env))
                                             (val (eval (third exp) env)))
                                           env))
              ((= (. exp) '-) {=>} (opair (- (val (eval (second exp) env))
                                             (val (eval (third exp) env)))
                                          env))
              ((= (. exp) '*) {=>} (opair (* (val (eval (second exp) env))
                                             (val (eval (third exp) env)))
                                          env))
              ((= (. exp) '>) {=>} (opair (> (val (eval (second exp) env))
                                             (val (eval (third exp) env)))
                                          env))
              ((= (. exp) '^) {=>} (opair exp env))
              ((= (. exp) 'if)
               {=>} (opair (if (val (eval (second exp) env))
                               (val (eval (third exp) env))
                               (val (eval (fourth exp) env)))
                           env))
              (T {=>} (let ((evexp {<-} (evlis exp env))
                            (env {<~} env))
                        {in} (let ((    body {<-} (third (. evexp)))
                                   (argnames {<-} (second (. evexp)))
                                   ( argvals {<-} (, evexp))
                                   (evn {<~} env))
                               {in} (opair (val (eval body
                                                      (extend argnames argvals env)))
                                           env)))))))

(! evlis (^ (l env)
            (if l
                (; (val (eval (. l) env))
                   (evlis (, l) env))
                {else} ())))

(! lookup (^ (name env)
             (? ((= env ()) {=>} ())
                ((= (. (. env)) name) {=>} (, (. env)))
                (T {=>} (lookup name (, env))))))
```

[35]The fragments delimited with {...} are comments, *i.e.* they are not part of encoded expression, and the parser will ignore them.

```
(! extend (^ (names vals env) (append (pair names vals) env)))

(! opair (^ (a b) (; a b)))
(! val (^ (op) (. op)))

(! first  (^ (l) (. l)))
(! second (^ (l) (. (, l))))
(! third  (^ (l) (. (, (, l)))))
(! fourth (^ (l) (. (, (, (, l))))))

(! append (^ (a b)
             (if a
                 (; (. a) (append (, a) b))
                 {else} b)))

(! pair (^ (a b)
           (if a
               (; (; (. a) (. b))
                  (pair (, a) (, b)))
               {else} ())))
```

E.g.
$\ulcorner$(`eval` (`QUOTE` (+ 2 3)) ())$\urcorner$ evaluates to $\ulcorner$(`5`)$\urcorner$ (*i.e.* $\ulcorner$(`5`.())$\urcorner$),
$\ulcorner$(`eval` (`QUOTE` (∗ (+23) 5)) ())$\urcorner$ evaluates to $\ulcorner$(`25`)$\urcorner$ (*i.e.*$\ulcorner$(`25`.())$\urcorner$), and
$\ulcorner$(`eval` (`QUOTE` ((^ (`x`) (+ `x` `x`)) (∗ 2 3)))())$\urcorner$ to $\ulcorner$(`12`)$\urcorner$.

2.3 $drcz_{0/1}$ implementations and example programs.

We will now provide two implementations of $drcz_0$ (and therefore $drcz_1$ as well) for $DRC\langle\mathcal{SE}\rangle$ machine. Our first approach would be to translate $drcz_0$ expressions into sequences of $DRC\langle\mathcal{SE}\rangle$ machine instructions.

Let $cmp : \mathcal{SE} \to \mathcal{SE}$ such that $Dom(cmp)$ are $drcz_0$ programs, and $Img(cmp)$ are $DRC\langle\mathcal{SE}\rangle$ machine programs. Then:

a. for a conditional expression $\varepsilon \sim \ulcorner$(`IF` α $\beta_\top$ $\beta_\bot$)$\urcorner$ its compilation image $cmp(\varepsilon)$ is $cmp(\alpha) :: \ulcorner$(`SELECT` $\varepsilon_{\beta_\top}$ $\varepsilon_{\beta_\bot}$)$\urcorner$, where $\varepsilon_{\beta_\top} = cmp(\beta_\top)$ and $\varepsilon_{\beta_\bot} = cmp(\beta_\bot)$.

b. for an application $\varepsilon \sim \ulcorner(\varepsilon_0\ \varepsilon_1 \ldots \varepsilon_n)\urcorner$ its compilation $cmp(\varepsilon)$ is $cmp(\varepsilon_n) :: \ldots :: cmp(\varepsilon_1) :: cmp(\varepsilon_0)$ if ε_0 is primitive operator's identifier, and $cmp(\varepsilon_n) :: \ldots :: cmp(\varepsilon_1) :: cmp(\varepsilon_0) :: \ulcorner$(`APPLY`)$\urcorner$ otherwise.

c. for constant ε its compilation $cmp(\varepsilon)$ is $\ulcorner$(`CONST` ε)$\urcorner$.

d. for primitive operator's identifier ε its compilation $cmp(\varepsilon)$ is $\ulcorner(\varepsilon')\urcorner$ where ε' is appropriate machine code (*e.g.* $\ulcorner$`ADD`$\urcorner$ for $\ulcorner+\urcorner$).

e. for variable ε its compilation $cmp(\varepsilon)$ is $\ulcorner$(`LOOKUP` ε)$\urcorner$.

f. for functional object $\varepsilon \sim \ulcorner$(^ $(\xi_1 \ldots \xi_n)\beta$)$\urcorner$ its compilation $cmp(\varepsilon)$ is $(NAME\ \xi_1 \ldots NAME\ \xi_n) :: cmp(\beta) :: (FORGET\ \xi_1 \ldots \xi_n)$.

These rules can be easily formalized with recursive equation:

Definition 15. *The translation* $cmp : \mathcal{SE} \to \mathcal{SE}$ *from* $drcz_0$ *to* $DRC\langle\mathcal{SE}\rangle$ *defines as follows:*

$$
Comp(\varepsilon) = \begin{cases}
\ulcorner(\texttt{CONST}\ \varepsilon)\urcorner\ ,\ \varepsilon \sim \ulcorner\texttt{T}\urcorner \vee \varepsilon \sim \ulcorner()\urcorner \vee num(\varepsilon) \sim \ulcorner\texttt{T}\urcorner \\
\\
\ulcorner(\texttt{CONS})\urcorner\ ,\ \varepsilon \sim \ulcorner;\urcorner \\
\ulcorner(\texttt{CAR})\urcorner\ ,\ \varepsilon \sim \ulcorner.\urcorner \\
\ulcorner(\texttt{CDR})\urcorner\ ,\ \varepsilon \sim \ulcorner,\urcorner \\
\ulcorner(\texttt{ATOM})\urcorner\ ,\ \varepsilon \sim \ulcorner@\urcorner \\
\ulcorner(\texttt{NUM})\urcorner\ ,\ \varepsilon \sim \ulcorner\#\urcorner \\
\ulcorner(\texttt{EQ})\urcorner\ ,\ \varepsilon \sim \ulcorner=\urcorner \\
\ulcorner(\texttt{ADD})\urcorner\ ,\ \varepsilon \sim \ulcorner+\urcorner \\
\ulcorner(\texttt{MUL})\urcorner\ ,\ \varepsilon \sim \ulcorner*\urcorner \\
\ulcorner(\texttt{GT})\urcorner\ ,\ \varepsilon \sim \ulcorner>\urcorner \\
\\
\ulcorner(\texttt{LOOKUP}\ \varepsilon)\urcorner\ ,\ atom(\varepsilon) \sim \ulcorner\texttt{T}\urcorner \wedge num(\varepsilon) \nsim \ulcorner\texttt{T}\urcorner \wedge \varepsilon \nsim \ulcorner;\urcorner \wedge \ldots \wedge \varepsilon \nsim \ulcorner>\urcorner \\
\\
Comp(\alpha) :: \ulcorner(\texttt{SELECT}\ \beta'_\top\ \beta'_\bot)\urcorner\ \textit{where} \\
\beta'_\top = Comp(\beta_\top)\ \textit{and}\ \beta'_\bot = Comp(\beta_\bot)\ , \\
\quad \varepsilon \sim \ulcorner(\texttt{IF}\ \alpha\ \beta_\top\ \beta_\bot)\urcorner \\
Comp(\varepsilon_n) :: \ldots :: Comp(\varepsilon_0)\ ,\ \varepsilon \sim \ulcorner(\varepsilon_0\ \varepsilon_1 \ldots \varepsilon_n)\urcorner \\
\\
\ulcorner(\texttt{PROC}\ \pi)\urcorner\ \textit{where} \\
\pi \sim \ulcorner(\texttt{NAME}\ \alpha_1 \ldots \texttt{NAME}\ \alpha_n)\urcorner :: Comp(\beta) :: \ulcorner(\texttt{FORGET}\ \alpha_1 \ldots \texttt{FORGET}\ \alpha_n)\urcorner\ , \\
\quad \varepsilon \sim \ulcorner(\hat{}\ (\alpha_1 \ldots \alpha_n)\ \beta)\urcorner \\
\\
Comp(\beta) :: \ulcorner(\texttt{NAME}\ \alpha)\urcorner\ ,\ \varepsilon \sim \ulcorner(!\ \alpha\ \beta)\urcorner
\end{cases}
$$

which rewrites straightforward into $drcz_1$:

```
(! compile (^ (prg)
              (if prg
                  (append (comp (. prg))
                          (compile (, prg)))
          {else} ())))

(! comp-primop (^ (e)
                  (? ((= e ';) {=>} '(CONS))
                     ((= e '.) {=>} '(CAR))
                     ((= e ',) {=>} '(CDR))
                     ((= e '#) {=>} '(NUM))
                     ((= e '@) {=>} '(ATOM))
                     ((= e '=) {=>} '(EQ))
                     ((= e '+) {=>} '(ADD))
                     ((= e '-) {=>} '(SUB))
                     ((= e '*) {=>} '(MUL))
                     ((= e '>) {=>} '(GT)))))

(! comp-app (^ (op app)
               (complis (reverse app)
                        (if (is-primop? op) () '(APPLY)))))

(! complis (^ (exprs postfix)
              (if exprs
                  (append (comp (. exprs))
                          (complis (, exprs) postfix))
          {else} postfix)))
```

```
(! comp (^ (expr)
           (? ((= expr ())
               {=>} '(CONST ()))
              ((# expr)
               {=>} (list 'CONST expr))
              ((@ expr)
               {=>} (if (is-primop? expr)
                        (comp-primop expr)
                  {else} (list 'LOOKUP expr)))
              ((= (. expr) 'quote)
               {=>} (list 'const (second expr)))
              ((= (. expr) '!)
               {=>} (append (comp (third expr)) (list 'NAME (second expr))))
              ((= (. expr) 'if)
               {=>} (append (comp (second expr))
                            (list 'SELECT
                                  (comp (third expr))
                                  (comp (fourth expr)))))
              ((= (. expr) '^)
               {=>} (list 'PROC
                          (append (name-block (second expr))
                                  (append (comp (third expr))
                                          (forget-block (reverse (second expr)))))))
              (t {=>} (comp-app (. expr) expr))
           )))

(! name-block (^ (varlist)
                 (if varlist
                     (; 'NAME
                        (; (. varlist)
                           (name-block (, varlist))))
                 {else} ())))

(! forget-block (^ (varlist)
                   (if varlist
                       (; 'FORGET
                          (; (. varlist)
                             (forget-block (, varlist))))
                 {else} ())))

(! is-primop? (^ (sym)
                 (member? sym '(; . , # @ = + - * >))))

(! reverse (^ (l) (_rev l ())))
(! _rev (^ (p r) (if p (_rev (, p) (; (. p) r)) {else} r)))

(! member? (^ (e l)
              (? ((= l ()) ())
                 ((= (. l) e) T)
                 (T (member? e (, l))))))
```

E.g. $\ulcorner$(`compile '( (! f (^ (n) (IF (= n 0) 1 (* n (f (− n 1)))))) (f (+ 2 3)) ))`$\urcorner$ evaluates to

```
⌜(PROC    (NAME n
           CONST 0
           LOOKUP n
           EQ
           SELECT     (CONST 1)
                      (CONST 1
                      LOOKUP n
                      SUB
                      LOOKUP f
                      APPLY
                      LOOKUP n
                      MUL)
           FORGET n)
NAME f
CONST 3
CONST 2
ADD
LOOKUP f
APPLY)⌝
```

With the use of *cmp* one can easily compile *eval* in $drcz_0$ into $DRC\langle\mathcal{SE}\rangle$ program, thus obtaining $drcz_0$ interpreter for $DRC\langle\mathcal{SE}\rangle$. We have done it with $drcz_1$ REPL interpreter (`drcz1-src/drcz1-interpreter.drcz1`), thus obtaining environment in which the rest of programs presented in this work were derived and tested before being compiled.

We will now demonstrate that the compiler just defined implements $drcz_0$ language properly, *i.e.* that it *preserves semantics of programs*, or in other words is *a compiler* in sense of definition 13.

Theorem 3. *Let $\varepsilon \in \mathcal{SE}$, and σ be an environment. Then:*

(1) for $\varepsilon \nsim \ulcorner(\hat{}\ (\alpha_1 \ldots \alpha_n)\ \beta)\urcorner$ if $Eval(\varepsilon, \sigma) = \langle \varepsilon', sigma' \rangle$ then $\langle \sigma, \rho, Comp(\varepsilon) \rangle \Rightarrow^*_{\mathfrak{S}} \langle \sigma', \ulcorner(\varepsilon')\urcorner :: \rho, \ulcorner()\urcorner \rangle$, and

(2) for $\varepsilon \sim \ulcorner(\hat{}\ (\alpha_1 \ldots \alpha_n)\ \beta)\urcorner$, $Comp(\varepsilon) = \varepsilon''$, such that for any $\varepsilon_1, \ldots, \varepsilon_n \in \mathcal{SE}$ if $Eval(\ulcorner(\varepsilon\ \varepsilon_1 \ldots \varepsilon_n)\urcorner, \sigma) = \langle \varepsilon', \sigma' \rangle$, then $\langle \sigma, \rho, Comp(\varepsilon_n) :: \ldots :: Comp(\varepsilon_1) :: \varepsilon'' :: \ulcorner(\texttt{APPLY})\urcorner \rangle \Rightarrow^*_{DRC\langle\mathcal{SE}\rangle}$ $\Rightarrow^*_{DRC\langle\mathcal{SE}\rangle} \langle \sigma', \ulcorner(\varepsilon')\urcorner :: \rho, \ulcorner()\urcorner \rangle$.

Proof. Part (1) by induction on ε's structure:

1) For $\varepsilon \sim \ulcorner()\urcorner$ or $\varepsilon \sim \ulcorner\texttt{T}\urcorner$ or $num(\varepsilon) \sim \ulcorner\texttt{T}\urcorner$:

   1. $Eval(\varepsilon, \sigma) = \langle \varepsilon, \sigma \rangle$, *(from def.14)*
   2. $Comp(\varepsilon) = \ulcorner(\texttt{CONST}\ \varepsilon)\urcorner$, *(from def.15)*
   3. $\langle \sigma, \rho, \ulcorner(\texttt{CONST}\ \varepsilon)\urcorner \rangle \Rightarrow_{\mathfrak{S}} \langle \sigma, \ulcorner\varepsilon\urcorner :: \rho, \ulcorner()\urcorner \rangle$, *(from def.6)*
   4. $\langle \sigma, \rho, Comp(\varepsilon) \rangle \Rightarrow^*_{\mathfrak{S}} \langle \sigma, \ulcorner\varepsilon\urcorner :: \rho, \ulcorner()\urcorner \rangle$. *(from (2), (3) and $\Rightarrow^*_{\mathfrak{S}}$ def.)*

2) For ε being symbol not representing primitive operator:

1. $Eval(\varepsilon, \sigma) = \langle \varepsilon', \sigma \rangle$, where $\varepsilon' = \sigma(\varepsilon)$, *(from def.14)*
2. $Comp(\varepsilon) = \ulcorner(\texttt{LOOKUP}\ \varepsilon)\urcorner$, *(from def.15)*
3. $\langle \sigma, \rho, \ulcorner(\texttt{LOOKUP}\ \varepsilon)\urcorner \rangle \Rightarrow_{\mathfrak{S}} \langle \sigma, \ulcorner\varepsilon'\urcorner :: \rho, \ulcorner()\urcorner \rangle$, where $\varepsilon' = \sigma(\varepsilon)$, *(from def.6)*
4. $\langle \sigma, \rho, Comp(\varepsilon) \rangle \Rightarrow^*_{\mathfrak{S}} \langle \sigma, \ulcorner\varepsilon'\urcorner :: \rho, \ulcorner()\urcorner \rangle$. *(from (2), (3) and $\Rightarrow^*_{\mathfrak{S}}$ def.)*

3) For $\varepsilon \sim \ulcorner(\texttt{QUOTE}\ \varepsilon')\urcorner$:

1. $Eval(\varepsilon, \sigma) = \langle \varepsilon', \sigma \rangle$, *(from def.14)*
2. $Comp(\varepsilon) = \ulcorner(\texttt{CONST}\ \varepsilon')\urcorner$, *(from def.15)*
3. $\langle \sigma, \rho, \ulcorner(\texttt{CONST}\ \varepsilon')\urcorner \rangle \Rightarrow_{\mathfrak{S}} \langle \sigma, \ulcorner\varepsilon'\urcorner :: \rho, \ulcorner()\urcorner \rangle$, *(from def.6)*
4. $\langle \sigma, \rho, Comp(\varepsilon) \rangle \Rightarrow^*_{\mathfrak{S}} \langle \sigma, \ulcorner\varepsilon'\urcorner :: \rho, \ulcorner()\urcorner \rangle$. *(from (2), (3) and $\Rightarrow^*_{\mathfrak{S}}$ def.)*

4) For $\varepsilon \sim \ulcorner(.\ \varepsilon_1)\urcorner$:

1. let $\varepsilon' = car(val(Eval(\varepsilon_1, \sigma)))$,
2. $Eval(\varepsilon, \sigma) = \langle \varepsilon', \sigma \rangle$,
   (from def.14 and (1))
3. $Comp(\varepsilon) = Comp(\varepsilon_1) :: Comp(\ulcorner.\urcorner) = Comp(\varepsilon_1) :: \ulcorner(\texttt{CAR})\urcorner$
   (from def.15)
4. $Eval(\varepsilon_1, \sigma) = \langle \varepsilon'_1, \sigma' \rangle$ implies $\langle \sigma, \rho, Comp(\varepsilon_1) \rangle \Rightarrow^*_{\mathfrak{S}} \langle \sigma, \ulcorner\varepsilon'_1\urcorner :: \rho, \ulcorner()\urcorner \rangle$.
   (induction hypothesis)
5. $\langle \sigma, \rho, Comp(\varepsilon_1) :: \ulcorner(\texttt{CAR})\urcorner \rangle \Rightarrow_{\mathfrak{S}} \langle \sigma', \ulcorner\varepsilon'_1\urcorner :: \rho, \ulcorner(\texttt{CAR})\urcorner \rangle$,
   (from (4) and "stack compositionality")
6. $\langle \sigma', \ulcorner(\varepsilon'_1)\urcorner :: \rho, \ulcorner(\texttt{CAR})\urcorner \rangle \Rightarrow_{\mathfrak{S}} \langle \sigma, \ulcorner\varepsilon'\urcorner :: \rho, \ulcorner()\urcorner \rangle$.
   (from (1) and def.6)
7. $\langle \sigma, \rho, Comp(\varepsilon_1) :: \ulcorner(\texttt{CAR})\urcorner \rangle \Rightarrow^*_{\mathfrak{S}} \langle \sigma', \ulcorner\varepsilon'\urcorner :: \rho, \ulcorner()\urcorner \rangle$.
   (from (5), (6), and $\Rightarrow^*_{\mathfrak{S}}$ def.)
8. $\langle \sigma, \rho, Comp(\varepsilon) \rangle \Rightarrow^*_{\mathfrak{S}} \langle \sigma', \ulcorner\varepsilon'\urcorner :: \rho, \ulcorner()\urcorner \rangle$.
   (from (3) and (7))

5) For $\varepsilon \sim \ulcorner(\texttt{IF}\ \alpha\ \beta_\top\ \beta_\bot)\urcorner$:

1. let $\alpha' = val(Eval(\alpha, \sigma))$,
2. $Eval(\varepsilon, \sigma) = \begin{cases} Eval(\beta_\top, \sigma)\ , & \alpha' \nsim \ulcorner()\urcorner \\ Eval(\beta_\bot, \sigma)\ , & \neg \end{cases}$.
   (from (1) and def.14)
3. $Comp(\varepsilon) = Comp(\alpha) :: \ulcorner(\texttt{SELECT}\ \beta'_\top\ \beta'_\bot)\urcorner$
   where $\beta'_\top = Comp(\beta_\top)$ and $\beta'_\bot = Comp(\beta_\bot)$,
   (from def.15)
4. $Eval(\alpha, \sigma) = \langle \alpha', \sigma' \rangle$ implies $\langle \sigma, \rho, Comp(\alpha) \rangle \Rightarrow^*_{\mathfrak{S}} \langle \sigma', \ulcorner(\alpha')\urcorner :: \rho, \ulcorner()\urcorner \rangle$,
   (induction hypothesis and (1))
5. $\langle \sigma, \rho, Comp(\alpha) :: \ulcorner(\texttt{SELECT}\ \beta'_\top\ \beta'_\bot)\urcorner \rangle \Rightarrow^*_{\mathfrak{S}} \langle \sigma', \ulcorner(\alpha')\urcorner :: \rho, \ulcorner(\texttt{SELECT}\ \beta'_\top\ \beta'_\bot)\urcorner \rangle$,
   (from (4) and compositionality)

6. $\langle \sigma', \ulcorner(\alpha')\urcorner :: \rho, \ulcorner(\texttt{SELECT}\ \beta'_\top\ \beta'_\bot)\urcorner\rangle \Rightarrow_{\mathfrak{S}} \left\{ \begin{array}{ll} \langle \sigma', \rho, \beta'_\top\rangle\ , & \alpha' \nsim \ulcorner()\urcorner \\ \langle \sigma', \rho, \beta'_\bot\rangle\ , & \neg \end{array} \right.$,
   (from def.6)
7. $Eval(\beta_i\, \sigma') =< \varepsilon_i, \sigma''_i >$ implies $\langle \sigma', \rho, Comp(\beta_i)\rangle \Rightarrow^*_{\mathfrak{S}} \langle \sigma''_i, \ulcorner\varepsilon_{\mathtt{i}}\urcorner :: \rho, \ulcorner()\urcorner\rangle$, for $i \in \{\top, \bot\}$, *(induction hypothesis)*
8. $\langle \sigma, \rho, Comp(\varepsilon)\rangle =$
   $= \langle \sigma, \rho, Comp(\alpha) :: \ulcorner(\texttt{SELECT}\ \beta'_\top\ \beta'_\bot)\urcorner\rangle \Rightarrow^*_{\mathfrak{S}} \left\{ \begin{array}{ll} \langle \sigma''_\top, \ulcorner(\varepsilon_\top)\urcorner :: \rho, \ulcorner()\urcorner\rangle\ , & \alpha' \nsim \ulcorner()\urcorner \\ \langle \sigma''_\bot, \ulcorner(\varepsilon_\bot)\urcorner :: \rho, \ulcorner()\urcorner\rangle\ , & \neg \end{array} \right.$.
   (from (6) and (7))

Part (2).

Let $n \geq 0$, $\varepsilon \sim \ulcorner(\hat{}\ (\alpha_1 \ldots \alpha_{\mathtt{n}})\ \beta)\urcorner$, and $\varepsilon_1, \ldots, \varepsilon_n \in \mathcal{SE}$.

1. Let $\varepsilon'_i = val(Eval(\varepsilon_i, \sigma))$, and $\varepsilon''_i = Comp(\varepsilon_i)$ for $i = 1, \ldots, n$,
2. $Eval(\ulcorner(\varepsilon\ \varepsilon_1 \ldots \varepsilon_{\mathtt{n}})\urcorner) = Eval(\beta, \sigma \oplus [\alpha_1 \mapsto \varepsilon'_1, \ldots \alpha_n \mapsto \varepsilon'_n])$, *(from (0) and def.14)*
3. $Eval(\varepsilon_i, sigma) = \langle \varepsilon'_i, sigma\rangle$ implies $\langle \sigma, \rho, \varepsilon''_i\rangle \Rightarrow^*_{\mathfrak{S}} \langle \sigma, \ulcorner\varepsilon'_{\mathtt{i}}\urcorner :: \rho, \ulcorner()\urcorner\rangle$, *(from (0) and induction hypothesis)*
4. $\langle \sigma, \rho, \varepsilon''_n :: \ldots :: \varepsilon''_1 :: \varepsilon''\rangle \Rightarrow^*_{\mathfrak{S}} \langle \sigma, \ulcorner\varepsilon'_1 \ldots \varepsilon'_{\mathtt{n}}\urcorner :: \rho, \varepsilon''\rangle$, *(from (3) and compositionality)*
5. $\varepsilon'' = Comp(\varepsilon) =$
   $= \ulcorner(\texttt{NAME}\ \alpha_1 \ldots \texttt{NAME}\ \alpha_{\mathtt{n}})\urcorner :: Comp(\beta) :: \ulcorner(\texttt{FORGET}\ \alpha_1 \ldots \texttt{FORGET}\ \alpha_{\mathtt{n}})\urcorner$, *(from def.15)*
6. $\langle \sigma, \ulcorner(\varepsilon'_1 \ldots \varepsilon'_{\mathtt{n}})\urcorner :: \rho, \ulcorner(\texttt{NAME}\ \alpha_1 \ldots \texttt{NAME}\ \alpha_{\mathtt{n}})\urcorner :: Comp(\beta) :: \ulcorner(\texttt{FORGET}\ \alpha_1 \ldots \texttt{FORGET}\ \alpha_{\mathtt{n}})\urcorner\rangle \Rightarrow^*_{\mathfrak{S}}$
   $\langle \sigma \oplus [\alpha_1 \mapsto \varepsilon'_1, \ldots \alpha_n \mapsto \varepsilon'_n], \rho, Comp(\beta) :: \ulcorner(\texttt{FORGET}\ \alpha_1 \ldots \texttt{FORGET}\ \alpha_{\mathtt{n}})\urcorner\rangle$,
   (from def.6 and compositionality)
7. $Eval(\beta, \sigma') = \langle \beta', sigma'\rangle$ implies $\langle \sigma', \rho, Comp(\beta)\rangle \Rightarrow^*_{\mathfrak{S}} \langle \sigma', \ulcorner(\beta') :: \rho, \ulcorner()\urcorner\urcorner\rangle$,
   (induction hypothesis)
8. $\langle \sigma', \rho, Comp(\beta) :: \ulcorner(\texttt{FORGET}\ \alpha_1 \ldots \texttt{FORGET}\ \alpha_{\mathtt{n}})\urcorner\rangle \Rightarrow^*_{\mathfrak{S}} \langle \sigma', \ulcorner(\beta')\urcorner :: \rho, \ulcorner(\texttt{FORGET}\ \alpha_1 \ldots \texttt{FORGET}\ \alpha_{\mathtt{n}})\urcorner\rangle$,
   (from (7) and compositionality)
9. $\langle \sigma \oplus [\alpha_1 \mapsto \varepsilon'_1, \ldots \alpha_n \mapsto \varepsilon'_n], \ulcorner(\beta')\urcorner :: \rho, \ulcorner(\texttt{FORGET}\ \alpha_1 \ldots \texttt{FORGET}\ \alpha_{\mathtt{n}})\urcorner\rangle \Rightarrow^*_{\mathfrak{S}}$
   $\langle \sigma\ulcorner(\beta')\urcorner :: \rho, \ulcorner()\urcorner\rangle$, *(from def. 6 and compositionality)*
10. $\langle \sigma, \rho, \varepsilon''_n :: \ldots :: \varepsilon''_1 :: \varepsilon''\rangle \Rightarrow^*_{\mathfrak{S}} \langle \sigma\ulcorner(\beta')\urcorner :: \rho, \ulcorner()\urcorner\rangle$. *(from (4), (6), (9) and compositionality)*

□

We will now provide two $drcz_1$ example programs.

Example 13. *A DRC⟨$\mathcal{SE}$⟩ machine emulator (*i.e. *interpreter) in $drcz_1$ – a straightforward representation of production rules of definition 6:*

```
(! step (^ (d r c)
           (if (= c ())
               (. r)
               (let ((op {<-} (. (disp c)))
                     ( c {<-} (, c))
                     (d {<~} d)
                     (r {<~} r))
                    {in} (? ((= op 'CONST)
                             {=>} (step d (; (. c) r) (, c)))
                            ((= op 'PROC)
                             {=>} (step d (; (. c) r) (, c)))
                            ((= op 'NAME)
                             {=>} (step (name (. c) (. r) d) (, r) (, c)))
                            ((= op 'FORGET)
                             {=>} (step (forget (. c) d) r (, c)))
                            ((= op 'LOOKUP)
                             {=>} (step d (; (lookup (. c) d) r) (, c)))
                            ((= op 'SELECT)
                             {=>} (if (= (. r) ())
                                      (step d (, r) (apd (. (, c)) (, (, c))))
                                      (step d (, r) (apd (. c) (, (, c))))))
                            ((= op 'APPLY)
                             {=>} (step d (, r) (apd (. r) c)))
                            ((= op 'CONS)
                             {=>} (step d (; (; (. r) (. (, r))) (, (, r))) c))
                            ((= op 'CAR)
                             {=>} (step d (; (. (. r)) (, r)) c))
                            ((= op 'CDR)
                             {=>} (step d (; (, (. r)) (, r)) c))
                            ((= op 'NUM)
                             {=>} (step d (; (# (. r)) (, r)) c))
                            ((= op 'ATOM)
                             {=>} (step d (; (@ (. r)) (, r)) c))
                            ((= op 'EQ)
                             {=>} (step d (; (= (. r) (. (, r))) (, (, r))) c))
                            ((= op 'ADD)
                             {=>} (step d (; (+ (. r) (. (, r))) (, (, r))) c))
                            ((= op 'MUL)
                             {=>} (step d (; (* (. r) (. (, r))) (, (, r))) c))
                            ((= op 'SUB)
                             {=>} (step d (; (- (. r) (. (, r))) (, (, r))) c))
                            ((= op 'GT)
                             {=>} (step d (; (> (. r) (. (, r))) (, (, r))) c)))))))

(! apd (^ (a b) (if a (; (. a) (apd (, a) b)) b)))

(! name (^ (sym val env) (; (; sym val) env)))

(! forget (^ (sym env)
             (? ((= env ()) {=>} ())
                ((= (. (. env)) sym) {=>} (, env))
                (T {=>} (; (. env) (forget sym (, env)))))))

(! lookup (^ (sym env)
             (? ((= env ()) {=>} ())
                ((= (. (. env)) sym) {=>} (, (. env)))
                (T {=>} (lookup sym (, env))))))
```

```
(! run (^ (prog inputs) (step () inputs prog)))
```

⌜`apd`⌝ is a short version of list concatenation function, while ⌜`name`⌝, ⌜`forget`⌝, and ⌜`lookup`⌝ implement operations on drawers[36]. Notice that "desugaring" this program to $drcz_0$, and compiling to $DRC\langle\mathcal{SE}\rangle$ yields [meta-circular] $DRC\langle\mathcal{SE}\rangle$ interpreter for $DRC\langle\mathcal{SE}\rangle$.

E.g. ⌜(`run '(NAME x LOOKUP x LOOKUP x MUL FORGET x) '(5)`)⌝ evaluates to ⌜`25`⌝.

Example 14. *A program for finding the greatest consistent subset of given set of sentences*[37]*, It presents our preferred style of programming in* $drcz_1$ *– to first define types (as algebras) and then describe program as set of complex relations on that types:*

Given a set of sentences of Classical Sentential Calculus, in order to find its biggest (wrt. inclusion) consistent subsets one has to consider the following questions:

1) what the sentences of CSC *are*?

2) when is the set of sentences of CSC *consistent*?

3) when is the set of sentences biggest wrt. inclusion?

To answer (1) we revert to classical definition of sentential calculus' grammar (*e.g.* def.1 in [15]):

Definition 16. *The set of sentences of Classical Sentential Calculus is the smallest set containing sentential variables and closed under operations of creating negation and alternative.*

$$\mathcal{S} = \bigcap\{X : (\forall_{\xi_i, i\in\mathbb{N}}\ \xi_i \in X) \wedge (\forall_{\xi_i,\xi_j\in X}\ ((-\xi_i) \in X \wedge (\xi_i + \xi_j) \in X))\}.$$

We capture these definitions with the following[38]:

```
(! SENTENCE:mk-variable (^ (v) (list v)))
(! SENTENCE:mk-negation (^ (s) (list 'N s)))
(! SENTENCE:mk-alternative (^ (s1 s2) (list 'A s1 s2)))

{ the remaining connectives can get defined in terms of negation and alternative: }
(! SENTENCE:mk-conjunction (^ (s1 s2)
                              (SENTENCE:mk-negation
                                (SENTENCE:mk-alternative
                                  (SENTENCE:mk-negation s1)
                                  (SENTENCE:mk-negation s2)))))

(! SENTENCE:mk-implication (^ (s1 s2)
                              (SENTENCE:mk-alternative
                                (SENTENCE:mk-negation s1)
                                s2)))
```

[36] Or rather on their "magical" A-list representation.

[37] Big thanks to Paweł Pawłowski, B.Sc. for asking to write such program for him.

[38] Notice the naming schema `<type>:<operation>`.

```
(! SENTENCE:mk-equality (^ (s1 s2)
                           (SENTENCE:mk-conjunction
                             (SENTENCE:mk-implication s1 s2)
                             (SENTENCE:mk-implication s2 s1))))


(! SENTENCE:varname (^ (s) (first s)))
(! SENTENCE:arg1 (^ (s) (second s)))
(! SENTENCE:arg2 (^ (s) (third s)))

(! SENTENCE:variable? (^ (s) (= (length s) 1)))
(! SENTENCE:negation? (^ (s) (and (= (length s) 2) (= (first s) 'N))))
(! SENTENCE:alternative? (^ (s) (and (= (length s) 3) (= (first s) 'A))))

(! SENTENCE:vars (^ (s) (uniq (list-all-vars s))))
```

To answer (2) it is enough to notice that set of CSC sentences $\{\varphi_1, \ldots, \varphi_n\}$ is consistent iff its big conjunction $\varphi_1 \& \ldots \& \varphi_n$ is *not* an antitautology (*i.e.* a sentence false under any possible arrangement of truth-values for φ_i, $i = 1, \ldots, n$).

```
(! SENTENCE:eval
   (^ (s bind)
      (? ((SENTENCE:variable? s)
          {=>} (lookup (SENTENCE:varname s) bind))
         ((SENTENCE:negation? s)
          {=>} (not (SENTENCE:eval (SENTENCE:arg1 s) bind)))
         ((SENTENCE:alternative? s)
          {=>} (or (SENTENCE:eval (SENTENCE:arg1 s) bind)
                   (SENTENCE:eval (SENTENCE:arg2 s) bind))))))

(! lookup (^ (v bind) (AL:assoc v bind (^ (x y) (= x y)))))

{ matrix method of checking for anti-tautology: }
(! gen-all-TF-arrangements
   (^ (l)
      (if (= l 1)
          '((()) (T))
          {else} (let ((children {<-} (gen-all-TF-arrangements (- l 1))))
                   {in} (append (map (^ (b) (; T b)) children)
                                (map (^ (b) (; () b)) children)))))))

(! gen-all-bindings (^ (vars)
                       (mapg (^ (bp arg) (pair arg bp))
                             vars
                             (gen-all-TF-arrangements (length vars)))))

(! SENTENCE:antitautology?
   (^ (s)
      (antitautology-test s (gen-all-bindings (SENTENCE:vars s)))))

(! antitautology-test
   (^ (s bindings)
      (? ((empty? bindings) {=>} T)
         ((SENTENCE:eval s (first bindings)) {=>} ())
         (T {=>} (antitautology-test s (rest bindings))))))

(! truth-table (^ (s)
                  (mapg (^ (b s) (list b (SENTENCE:eval s b)))
                        s
                        (gen-all-bindings (SENTENCE:vars s)))))
```

```
(! big-conjunction
   (^ (ss)
      (if (empty? (rest ss))
          (first ss)
          {else} (SENTENCE:mk-negation
                   (SENTENCE:mk-alternative
                     (SENTENCE:mk-negation (first ss))
                     (SENTENCE:mk-negation (big-conjunction (rest ss))))))))

(! consistent? (^ (ss)
                  (not (SENTENCE:antitautology? (big-conjunction ss)))))
```

Finally, to answer (3) one can think of the problem as follows: first find list of all consistent subsets of given set (excluding some obvious ones by not checking inside already consistent subsets), and then filter this list (⌜trim − subsets⌝) by dropping every set included in any of the others (⌜subset − of − any?⌝).

```
(! gcs (^ (ss)
          (let ((ss {<-} (gather-gcss (list ss) () ())))
               {in} (trim-subsets ss ss))))

(! gather-gcss (^ (pend checked res)
                  (? ((empty? pend)
                      {=>} res)
                     ((memberg? (first pend) checked SET:equal?)
                      {=>} (gather-gcss (rest pend)
                                        checked
                                        res))
                     ((consistent? (first pend))
                      {=>} (gather-gcss (rest pend)
                                        (push (first pend) checked)
                                        (push (first pend) res)))
                     (T
                      {=>} (gather-gcss (append (incl-predecessors (first pend))
                                                (rest pend))
                                        (push (first pend) checked)
                                        res)))))

(! trim-subsets (^ (pend ss)
                   (if pend
                       (let ((   s {<-} (first pend))
                             (pend {<-} (rest pend))
                             (ss {<~} ss))
                            {in} (if (subset-of-any? s (SET:drop s ss))
                                     (trim-subsets pend ss)
                                     {else} (push s (trim-subsets pend ss))))
                       {else} ())))

(! incl-predecessors
   (^ (set)
      (mapg (^ (el set) (SET:diff set (SET:mk-singleton el))) set set)))

(! subset-of-any? (^ (s ss)
                     (? ((SET:empty? ss) {=>} ())
                        ((SET:subset? s (first ss)) {=>} T)
                        (T {=>} (subset-of-any? s (rest ss))))))

{-- main --}
(map (^ (x) (map SENTENCE:write x)) (gcs (map SENTENCE:parse (read))))
```

Where ⌜(read)⌝ stands for standard (keyboard) input operator (*cf.* appendix A). The code refers also to ⌜SENTENCE : parse⌝ and ⌜SENTENCE : write⌝, as we have also programmed a simple parser from Polish notation, not included on this listing.

In the above code the operation *mapg* is generalization of map, consuming one additional argument, for making certain variables visible (a similar solution could be seen in our *let* expressions, where each time we wanted to pass some argument we had to rewrite it explicitly). We could avoid writing these if we picked some strategy for binding free variables. These issues do not concern us now, and are discussed in details in [18], while a possible technique for compiling closures to $DRC\langle\mathcal{SE}\rangle$ machine would be Reynold's *defunctionalization* ([14]) – ingenious tool used both in compilers and equivalence proofs for functional programming languages (*cf.* also [3]).

E.g. Given ⌜((p) (q) (N p) (N q) (A p q))⌝ yields
⌜(((p) (q) (A p q)) ((p) (N q) (A p q)) ((q) (N p) (A p q)) ((N p) (N q)))⌝.

Despite its small size (somewhere around 50 definitions, including the parser), the program works quite well[39] ; it could be optimized with *e.g.* more effective definition of sets inclusion – however we are only concerned with expressive power at the moment. Notice how easily the program emerged from the problem stated.

Full implementation can be found in `drcz1-src/gcss.drcz1` file on attached CD.

2.4 Describing procedures with $FCL\langle\mathcal{SE}\rangle$ and $FCL^*\langle\mathcal{SE}\rangle$.

$FCL\langle\mathcal{SE}\rangle$ is extremely simple, imperative language of assignments and jumps. It is closely related to Register Machines – family of computational models which emerged around 1960s as replacement for Turing Machines[40].

$FCL\langle\mathcal{SE}\rangle$ differs from $DRC\langle\mathcal{SE}\rangle$ in three significant points:

1) it's environment is "flat", *i.e.* each drawer holds exactly one sheet,

2) it stores partial results by naming them in environment, not by holding them on stack,

3) it captures subcomputations as blocks and jumps, not procedural constants and applications.

These features make $FCL\langle\mathcal{SE}\rangle$ both easy to implement (on any existing hardware/software) and nice object of study, as will be seen in chapter 3.

$FCL\langle\mathcal{SE}\rangle$ programs are *S-expressions* encoding blocks of commands. To grasp the idea, consider the following:

[39] Subsets of set consisting of 8 sentences were found below 200sec. on 1GHz PC.

[40] Reader interested in deriving RMs from TMs might wish to see *abacus machine* of [2], ch.5.

Example 15. *An $FCL\langle\mathcal{SE}\rangle$ program of two inputs m and n, describing procedure of computing m^n.*

```
((m n)
 init

 (init (let r 1)
       (goto test))

 (test (if (= n 0) end loop))

 (loop (let r (* r m))
       (let n (- n 1))
       (goto test))

 (end (return r)))
```

E.g. *when given input values* `⌜2⌝` *and* `⌜3⌝` *yields* `⌜8⌝`.

A much more interesting example of $FCL\langle\mathcal{SE}\rangle$ program will be presented in section 2.5.

In general, programs in $FCL\langle\mathcal{SE}\rangle$ have form $\ulcorner(\xi\ \alpha_0)\urcorner :: \beta$, where ξ is a list of input names, α_0 starting block name, and β is A-list with block labels as keys and blocks as values. Blocks are lists of instructions, each one being either:

a. **an assignment** – instruction of assigning given expression's value to selected variable, *e.g.* `⌜(let a (+ b c))⌝`

b. **a jump directive** – instruction of changing current block to one with given label, *e.g.* `⌜(goto test2)⌝`

c. **a conditional jump directive** – instruction of changing current block: if condition expression evaluates to `⌜T⌝` then changing to first given label, else to second, *e.g.* `⌜(if (= a 0) zero nonzero)⌝`

d. **return statement** – evaluate given expression and return its value as program's result. *e.g.* `⌜(return (+ a b))⌝`

To run a program π with inputs $\varepsilon_1, \ldots, \varepsilon_n$ initialize store with input names and values, pick its initial block label and run it:

$$Run(\pi, \langle\varepsilon_1, \ldots, \varepsilon_n\rangle) = RunBlock\left(\begin{array}{c} startLabel(\pi), \\ initStore(inputNames(\pi), \langle\varepsilon_1, \ldots, \varepsilon_n\rangle), \\ \pi \end{array}\right)$$

To run given block, find it's definition and "step" it:

$$RunBlock(pp, \sigma, \pi) = Step(findBlock(pp, \pi), \sigma, \pi).$$

To "step" a sequence of instructions:

$$Step(\beta,\sigma,\pi) = \begin{cases} Step(\beta_r, \sigma \oplus [\alpha \mapsto \varepsilon'], \pi) \ , \ \beta \sim \ulcorner((\texttt{let}\ \alpha\ \varepsilon)\ .\ \beta_{\texttt{r}})\urcorner \\ RunBlock(\lambda,\sigma,\pi) \ , \ \beta \sim \ulcorner((\texttt{goto}\ \lambda))\urcorner \\ RunBlock(\lambda_1,\sigma,\pi) \ , \ \beta \sim \ulcorner((\texttt{if}\ \varepsilon\ \lambda_1\ \lambda_2))\urcorner \wedge \varepsilon' \nsim \ulcorner()\urcorner \\ RunBlock(\lambda_2,\sigma,\pi) \ , \ \beta \sim \ulcorner((\texttt{if}\ \varepsilon\ \lambda_1\ \lambda_2))\urcorner \wedge \varepsilon' \sim \ulcorner()\urcorner \\ \varepsilon' \ , \ \beta \sim \ulcorner((\texttt{return}\ \varepsilon))\urcorner \end{cases}$$

where ε' stands for $evalExpr(\varepsilon,\sigma)$, evaluation function for $\mathcal{SE}$ polynomials; we don't describe it here as it is just a restriction of $drcz_0$'s $Eval$ function. For more detailed discussion on the language refer [9].

Glück in his [7] extends $FCL\langle\mathcal{SE}\rangle$ with recursive call: in RHS of *let* assignment[41] instead of expression there can be call of the form $\ulcorner(\texttt{call}\ \alpha)\urcorner$, where α is label of some block. It executes this block in copy of it's current environment σ (with another FCL interpreter), and takes the result as its own value. We refer to this extended version as $FCL^*\langle\mathcal{SE}\rangle$. An example program – computing the Ackermann function (from [7] also):

```
((m n)
 ack

 (ack (if (= m 0) done next))
 (done (return (+ n 1)))
 (next (if (= n 0) ack0 ack1))
 (ack0 (let n 1)
       (goto ack2))
 (ack1 (let n (- n 1))
       (let n (call ack))
       (goto ack2))
 (ack2 (let m (- m 1))
       (let n (call ack))
       (return n)) )
```

2.5 $FCL^{(*)}\langle\mathcal{SE}\rangle$ implementations and examples.

To implement $FCL\langle\mathcal{SE}\rangle$ interpreter in $drcz_1$ it is enough to implement functions *Run*, *RunBlock* and *Step* from previous section:

```
(! run (^ (program inputvals)
          (run-block (PROGRAM:start-pp program)
                     (STORE:new (PROGRAM:input-names program) inputvals)
                     (PROGRAM:block-map program))))

(! run-block (^ (pp store block-map)
                (step (BLOCK-MAP:find pp block-map) store block-map)))

(! eval-expr (^ (expr store)
                (? ((= expr ()) {=>} ())
                ((# expr) {=>} expr)
                ((@ expr) {=>} (STORE:lookup expr store))
                ((= (first expr) 'quote) {=>} (second expr))
                (T {=>} (apply-op (first expr)
                                  (evlis-expr (rest expr) store))))))
```

[41] In our implementation in *return* also.

```
(! evlis-expr (^ (expr-list store)
                (if (empty? expr-list)
                    ()
                    {else} (; (eval-expr (first expr-list) store)
                              (evlis-expr (rest expr-list) store)))))

(! step { CODE-BLOCK x STORE x BLOCK-MAP -> SE }
 (^ (code-block store block-map)
    (let ((      cmd {<-} (CODE-BLOCK:first-cmd code-block))
          (  cb-rest {<-} (CODE-BLOCK:rest-cmds code-block))
          (    store {<~} store)
          (block-map {<~} block-map))
         {in} (? ((CMD:let? cmd)
                  {=>} (step cb-rest
                             (STORE:update (CMD:let-variable cmd)
                                           (let ((expr {<-} (CMD:let-expression cmd))
                                                 (    store {<~} store)
                                                 (block-map {<~} block-map))
                                                {in} (eval-expr expr store))
                                           store)
                       block-map))
                 ((CMD:goto? cmd)
                  {=>} (run-block (CMD:goto-pp cmd) store block-map))
                 ((CMD:if? cmd)
                  {=>} (if (eval-expr (CMD:if-condition cmd) store)
                           (run-block (CMD:if-then-pp cmd) store block-map)
                           {else} (run-block (CMD:if-else-pp cmd) store block-map)))
                 ((CMD:return? cmd)
                  {=>} (let ((expr {<-} (CMD:return-expression cmd))
                             (    store {<~} store)
                             (block-map {<~} block-map))
                            {in} (eval-expr expr store))))
                 (T
                  {=>} (list 'ERR 'step: 'unknown 'command cmd))))))
```

To enable $FCL^*\langle\mathcal{SE}\rangle$ calls we only have to modify *step*:

```
((CMD:let? cmd)
 {=>} (step cb-rest
            (STORE:update (CMD:let-variable cmd)
                          (let ((expr {<-} (CMD:let-expression cmd))
                                (    store {<~} store)
                                (block-map {<~} block-map))
                               {in} (if (call? expr)
                                        (run-block (call-pp expr) store block-map)
                                        {else} (eval-expr expr store)))
                          store)
            block-map))
```

Full source for the interpreter can be found on attached CD in `drcz1-src/FCL/fcl-rec-interpreter.drcz1` file (*cf.* also appendix B). Following Glück ([7]) we have embedded in this interpreter some more sophisticated $\mathcal{SE}$ operators (*i.a. append*, *drop*, *list*) These can be found in `apply-op` definition.

Now we would like to find an effective translation from $FCL\langle\mathcal{SE}\rangle$ to $drcz_0$, *i.e.* a compiler. In order to construct one we have to examine semantics of $FCL\langle\mathcal{SE}\rangle$.

First, blocks – sequences of zero or more assignments followed by return statement or (un)conditional jump – can be identified with $Env \to \mathcal{SE}$ functions they realize (where $Env = \mathcal{SE}^{Id}$, $Id = \{\varepsilon \in \mathcal{SE} : atom(\varepsilon) \sim \ulcorner \mathtt{T} \urcorner \wedge num(\varepsilon) \sim \ulcorner () \urcorner\}$). Each block as function *definition* ends with either result expression (of return statement), or tail call(s) to other function(s) (block(s)).

To express a single block of $n > 0$ instructions in $drcz_0$, we can split it into n "small transformations" and encode each one with the following rules (we assume there are $k > 0$ variable identifiers used in program $\pi \in Prog_{FCL\langle\mathcal{SE}\rangle}$):

a. An assignment $\ulcorner(\mathtt{let}\ \xi_\mathtt{i}\ \beta)\urcorner$ where ξ_i is some variable identifier ($i \leq k$) and β is [valid] expression (*i.e.* some polynomial of k variables $\xi_1, \ldots, \xi_k$ over $\mathcal{SE}$), followed by a list of one or more instructions (transformations of form (a)-(d)) τ, encode as

$$\ulcorner(\hat{}\ (\xi_1 \ldots \xi_\mathtt{k})\ (\tau'\ \xi_1 \ldots \xi_{\mathtt{i}-1}\ \beta\ \xi_{\mathtt{i}+1} \ldots \xi_\mathtt{k}))\urcorner,$$

where τ' is encoding of τ.

b. An unconditional jump $\ulcorner(\mathtt{goto}\ \alpha)\urcorner$ where α is some block label identifier encode as

$$\ulcorner(\hat{}\ (\xi_1 \ldots \xi_\mathtt{k})\ (\alpha\ \xi_1 \ldots \xi_\mathtt{k}))\urcorner.$$

c. A conditional jump $\ulcorner(\mathtt{if}\ \varepsilon\ \alpha_\top\ \alpha_\bot)\urcorner$ where ε is some expression, $\alpha_\top$, $\alpha_\bot$ are some block labels identifiers, encode as

$$\ulcorner(\hat{}\ (\xi_1 \ldots \xi_\mathtt{k})\ (\mathtt{if}\ \varepsilon'\ (\alpha_\top\ \xi_1 \ldots \xi_\mathtt{k})\ (\alpha_\bot\ \xi_1 \ldots \xi_\mathtt{k})))\urcorner,$$

where ε' is the encoding of expression ε.

d. A return statement $\ulcorner(\mathtt{return}\ \varepsilon)\urcorner$ where ε is some expression, encode as

$$\ulcorner(\hat{}\ (\xi_1 \ldots \xi_\mathtt{k})\ \varepsilon')\urcorner,$$

where ε' is the encoding of expression ε.

To name the whole encoded block β with its label α, simply represent it as $\ulcorner(!\ \alpha\ \beta)\urcorner$.

Finally, to encode the argument identifiers $(\xi_1, \ldots, \xi_{l \leq k})$ and initial block label α_0 encode the whole program π as

$$\ulcorner\beta^* :: (\alpha_0\ \xi'_1 \ldots \xi'_\mathtt{k})\urcorner,$$

where β^* is list of $m > 0$ block definitions $\ulcorner(!\ \alpha_\mathtt{m}\ \beta_\mathtt{m})\urcorner$, and $\xi'_i = \begin{cases} \ulcorner(\mathtt{read})\urcorner, & \text{if } i \leq l \\ \ulcorner()\urcorner, & \neg \end{cases}$ for $i = 1, \ldots, k$.

```
(! compile (^ (prog)
              (let ((vars {<-} (PROGRAM:input-names prog))
                    (allvars {<-} (disp (PROGRAM:all-vars prog)))
                    (main {<-} (PROGRAM:start-pp prog))
                    (defs {<-} (PROGRAM:block-map prog)))
                   {in} (append (comp-defs defs allvars)
                                (comp-target main vars allvars)))))
```

```
(! comp-defs (^ (defs vars)
                (if defs
                    (let ((name {<-} (. (first defs)))
                          ( val {<-} (, (first defs)))
                          (defs {<-} (rest defs))
                          (vars {<~} vars))
                         {in} (; (list '! name (comp-block val vars))
                                 (comp-defs defs vars)))
                    {else} ())))

(! comp-target (^ (label inputs vars) (list (; label (mk-vars inputs vars)))))

(! mk-vars (^ (inputs vars)
              (? ((= vars ())
                  {=>} ())
                 ((member? (. vars) inputs)
                  {=>} (; (list 'read) (mk-vars inputs (, vars))))
                 (T
                  {=>} (; T (mk-vars inputs (, vars)))))))


(! comp-block (^ (b vars)
                 (let ((  op {<-} (. (. b)))
                       (args {<-} (, (. b)))
                       (   b {<-} (, b))
                       (vars {<~} vars))
                      {in} (? ((= op 'let)
                               {=>} (list '^
                                          vars
                                          (; (comp-block b vars)
                                             (mk-asgn (. args)
                                                      (. (, args))
                                                      vars))))
                              ((= op 'return)
                               {=>} (mk-rtn (. args) vars))
                              ((= op 'goto)
                               {=>} (mk-jmp (. args) vars))
                              ((= op 'if)
                               {=>} (mk-if (. args)
                                           (. (, args))
                                           (. (, (, args)))
                                           vars))
                              (T {=>} 'err))))))

(! mk-asgn (^ (var expr vars)
              (? ((= vars ())
                  {=>} ())
                 ((= (. vars) var)
                  {=>} (; expr (, vars)))
                 (T
                  {=>} (; (. vars) (mk-asgn var expr (, vars)))))))

(! mk-rtn (^ (expr vars)
             (list '^ vars expr)))

(! mk-jmp (^ (label vars)
             (list '^ vars (; label vars))))

(! mk-if (^ (perm concl alt vars)
            (list '^ vars (list 'if perm (; concl vars) (; alt vars)))))
```

E.g. An $FCL\langle\mathcal{SE}\rangle$ program

```
((n m)
 init

 (init (let r 1)
       (goto test))

 (test (if (= m 0) end {else} loop))

 (loop (let r (* r n))
       (let m (- m 1))
       (goto test))

 ( end (return r)))
```

compiles to

```
(
 (! init (^ (n m r) ((^ (n m r) (test n m r)) n m 1)))
 (! test (^ (n m r) (if (= m 0) (end n m r) (loop n m r))))
 (! loop (^ (n m r) ((^ (n m r)
                        ((^ (n m r) (test n m r)) n
                                                  (- m 1)
                                                  r)) n
                                                      m
                                                      (* r n))))
 (!  end (^ (n m r) r))

 (init (read) (read) ())
)
```

Similarly we have implemented an $FCL\langle\mathcal{SE}\rangle$ to $DRC\langle\mathcal{SE}\rangle$ compiler; the idea behind it is exactly the same – with the only significant detail is encoding assignments $\ulcorner(!\ \alpha\ \beta)\urcorner$ as $\beta' :: \ulcorner(\texttt{FORGET}\ \alpha\ \texttt{NAME}\alpha)\urcorner$ (for keeping the environment "flat").

```
(! compile
 (^ (prog)
    (let ((used-vars {<-} (list-all-vars prog))
          (input-vars {<-} (PROGRAM:input-names prog))
          (start-pp {<-} (PROGRAM:start-pp prog))
          (block-map {<-} (PROGRAM:block-map prog)))
         {in} (append (initialize-variables used-vars input-vars)
                      (append (compile-blockmap block-map)
                              (mk-start-block start-pp))))))

(! initialize-variables
 (^ (vars input)
    (if vars
        (append (if (member? (first vars) input)
                    (list 'READ 'NAME (first vars))
                    {else} (list 'CONST () 'NAME (first vars)))
                (initialize-variables (rest vars) input))
        {else} ())))

(! mk-start-block (^ (pp) (list 'LOOKUP pp 'TRAPPLY)))
```

```
(! compile-blockmap (^ (bm)
                       (if (empty? bm)
                           ()
                           {else} (append (compile-block (first bm))
                                          (compile-blockmap (rest bm))))))

(! compile-block { : CONS(LABEL,CODE) -> DRC<SE>CODE }
 (^ (block)
    (let ((label {<-} (. block))
          (code {<-} (, block)))
         {in} (list 'PROC (compile-code code) 'NAME label))))

(! compile-code
 (^ (code)
    (if code
        (let ((      cmd {<-} (first code))
              (rest-cmds {<-} (rest code)))
             {in} (append (? ((CMD:let? cmd)
                              {=>} (compile-assignment (CMD:let-variable cmd)
                                                       (CMD:let-expression cmd)))
                             ((CMD:goto? cmd)
                              {=>} (compile-jump (CMD:goto-pp cmd)))
                             ((CMD:if? cmd)
                              {=>} (compile-conditional (CMD:if-condition cmd)
                                                        (CMD:if-then-pp cmd)
                                                        (CMD:if-else-pp cmd)))
                             ((CMD:return? cmd)
                              {=>} (compile-return (CMD:return-expression cmd)))
                             (T
                              {=>} (list 'err! 'unknown 'command cmd)))
                          (compile-code rest-cmds)))
        {else} ())))

(! compile-assignment (^ (var expr)
                         (append (compile-expression expr)
                                 (list 'FORGET var 'NAME var))))

(! compile-jump (^ (pp) (list 'LOOKUP pp 'TRAPPLY)))

(! compile-conditional (^ (cnd then-pp else-pp)
                          (append (compile-expression cnd)
                                  (list 'SELECT (list 'LOOKUP then-pp)
                                                (list 'LOOKUP else-pp)
                                        'TRAPPLY))))

(! compile-return (^ (expr) (compile-expression expr))) {a halt?!}

(! compile-expression
 (^ (expr)
    (? ((or (= expr ()) (# expr)) {=>} (list 'CONST expr))
       ((@ expr) {=>} (if (is-primop? expr)
                          (comp-primop expr)
                          {else} (list 'LOOKUP expr)))
       (T {=>} (compile-list (reverse expr))))))

(! compile-list
 (^ (exprs)
    (if exprs
        (append (compile-expression (first exprs))
                (compile-list (rest exprs)))
        {else} ())))
```

Full source can be found in `drcz1-src/FCL/fcl2drc.drcz1` file.

E.g. the same $FCL\langle\mathcal{SE}\rangle$ program

```
((n m)
 init

 (init (let r 1)
       (goto test))

 (test (if (= m 0) end {else} loop))

 (loop (let r (* r n))
       (let m (- m 1))
       (goto test))

 ( end (return r)))
```

compiles to

```
(READ
 NAME n
 CONST ()
 NAME r
 READ
 NAME m
 PROC (CONST 1
       FORGET r
       NAME r
       LOOKUP test
       TRAPPLY)
 NAME init
 PROC (CONST 0
       LOOKUP m
       EQ
       SELECT (LOOKUP end)
              (LOOKUP loop)
       TRAPPLY)
 NAME test
 PROC (LOOKUP n
       LOOKUP r
       MUL
       FORGET r
       NAME r
       CONST 1
       LOOKUP m
       SUB
       FORGET m
       NAME m
       LOOKUP test
       TRAPPLY)
 NAME test
 PROC (LOOKUP r)
 NAME end
 LOOKUP init
 TRAPPLY)
```

We will now provide two example FCL programs.

Example 16. *$DRC\langle\mathcal{SE}\rangle$ machine emulator in $FCL\langle\mathcal{SE}\rangle$ – inputs two lists: program and initial encoding of R stack.*

```
((prog inputs)
 init

 (init (let D ())
       (let R inputs)
       (let C prog)
       (goto step))

 (step (if (= C ()) end step2))

 (end (return (. R)))

 (step2 (let cmd (. C))
        (let C (, C))
        (goto check-const))

 (check-const  (if (= cmd 'CONST) do-const check-proc))
 (check-proc   (if (= cmd 'PROC) do-const {sic!} check-lookup))
 (check-lookup (if (= cmd 'LOOKUP) do-lookup check-name))
 (check-name   (if (= cmd 'NAME) do-name check-forget))
 (check-forget (if (= cmd 'FORGET) do-forget check-select))
 (check-select (if (= cmd 'SELECT) do-select check-apply))
 (check-apply  (if (= cmd 'APPLY) do-apply check-cons))
 (check-cons (if (= cmd 'CONS) do-cons check-car))
 (check-car  (if (= cmd 'CAR) do-car check-cdr))
 (check-cdr  (if (= cmd 'CDR) do-cdr check-eq))
 (check-eq   (if (= cmd 'EQ) do-eq check-num))
 (check-num  (if (= cmd 'NUM) do-num check-atom))
 (check-atom (if (= cmd 'ATOM) do-atom check-add))
 (check-add  (if (= cmd 'ADD) do-add check-sub))
 (check-sub  (if (= cmd 'SUB) do-sub check-mul))
 (check-mul  (if (= cmd 'MUL) do-mul check-gt))
 (check-gt   (if (= cmd 'GT) do-gt err))

 (err (return (list 'ERROR 'UNKNOWN 'COMMAND cmd)))

 (do-const (let exp (. C))
           (let C (, C))
           (let R (; exp R))
           (goto step))

 (do-lookup (let var (. C))
            (let C (, C))
            (let R (; (. (AL:lookup var D)) R))
            (goto step))

 (do-name (let var (. C))
          (let val (. R))
          (let C (, C))
          (let R (, R))
          (let D (AL:update var (; val (AL:lookup var D)) D))
          (goto step))

 (do-forget (let var (. C))
            (let C (, C))
            (let D (AL:update var (, (AL:lookup var D)) D))
            (goto step))
```

```
(do-select (let perm (. R))
           (let R (, R))
           (let concl (. C))
           (let alt (. (, C)))
           (let C (, (, C)))
           (if perm do-concl do-alt))

(do-concl (let C (append concl C))
          (goto step))

(do-alt (let C (append alt C))
        (goto step))

(do-apply (let proc (. R))
          (let R (, R))
          (let C (append proc C))
          (goto step))

(do-cons (let arg1 (. R))
         (let arg2 (. (, R)))
         (let R (, (, R)))
         (let res (; arg1 arg2))
         (let R (; res R))
         (goto step))

(do-car (let arg1 (. R))
        (let R (, R))
        (let res (. arg))
        (let R (; res R))
        (goto step))

(do-cdr (let arg1 (. R))
        (let R (, R))
        (let res (, arg))
        (let R (; res R))
        (goto step))

(do-eq (let arg1 (. R))
       (let arg2 (. (, R)))
       (let R (, (, R)))
       (let res (= arg1 arg2))
       (let R (; res R))
       (goto step))

(do-add (let arg1 (. R))
        (let arg2 (. (, R)))
        (let R (, (, R)))
        (let res (+ arg1 arg2))
        (let R (; res R))
        (goto step))

(do-sub (let arg1 (. R))
        (let arg2 (. (, R)))
        (let R (, (, R)))
        (let res (- arg1 arg2))
        (let R (; res R))
        (goto step))

(do-mul (let arg1 (. R))
        (let arg2 (. (, R)))
        (let R (, (, R)))
        (let res (* arg1 arg2))
```

```
         (let R (; res R))
         (goto step))

 (do-gt (let arg1 (. R))
        (let arg2 (. (, R)))
        (let R (, (, R)))
        (let res (> arg1 arg2))
        (let R (; res R))
        (goto step))
)
```

E.g. when given arguments ⌜(PROC (NAME x LOOKUP x LOOKUP x MUL FORGET x) APPLY)⌝ and ⌜(5)⌝ yields ⌜25⌝.

Example 17. *A bath interpreter for $drcz_0$ in $FCL^*\langle\mathcal{SE}\rangle$ – given program, list of variable identifiers, and inputs produces output (result). A program is represented as a list, whose head (car) is the target call ("main") $\ulcorner(\varphi\ \xi_1 \ldots \xi_n)\urcorner$, $n \geq 0$, and whose tail (cdr) is list of definitions of the form $\ulcorner(!\ \alpha\ \beta)\urcorner$, where α is some variable identifier (symbol) and β is any* S-expression *(most often a lambda form).*

```
((program init-vars init-vals)
 start

 (start (let env (AL:new init-vars init-vals))
        (let expr (. program))
        (let defs (, program))
        (let env (call process-defs defs env))
        (let res (call eval expr env))
        (let r-expr (. res))
        (let r-env (, res))
        (return r-expr))

{- interpreting a sequence of definitions: -}
 (process-defs (if (= defs ()) defs-finished process-def))

 (process-def (let expr (. defs))
              (let defs (, defs))
              (let res (call eval expr env))
              (let env (, res))
              (goto process-defs))

 (defs-finished (return env))

{- evaluator -}
 (eval (goto check-nil))

 (check-nil    (if (= expr ()) do-nil check-num))
 (check-num    (if (# expr) do-num check-sym))
 (check-sym    (if (@ expr) do-sym check-quote))
 (check-quote  (if (= (. expr) 'quote) do-quote check-lambda))
 (check-lambda (if (= (. expr) '^) do-lambda check-eq))
 (check-eq     (if (= (. expr) '=) do-eq check-cons))
 (check-cons   (if (= (. expr) ';) do-cons check-car))
 (check-car    (if (= (. expr) '.) do-car check-cdr))
 (check-cdr    (if (= (. expr) ',) do-cdr check-atom))
{...}
```

```
(check-atom   (if (= (. expr) '@) do-atom check-if))
(check-if     (if (= (. expr) 'if) do-if check-label))
(check-label  (if (= (. expr) '!) do-label check-application))
(check-application (goto do-application))

(err (return (list 'error 'evaluating expr 'in env)))

(do-nil (return (; () env)))

(do-num (return (; expr env)))

(do-sym (return (; (AL:lookup expr env) env)))

(do-quote (return (; (. (, expr)) env)))

(do-lambda (return (; expr env)))

(do-eq (let a1 (. (, expr)))
         (let a2 (. (, (, expr))))
         (let expr a1)
         (let res (call eval expr env))
         (let a1 (. res))
         (let expr a2)
         (let res (call eval expr env))
         (let a2 (. res))
         (return (; (= a1 a2) env)))

(do-cons (let a1 (. (, expr)))
         (let a2 (. (, (, expr))))
         (let expr a1)
         (let res (call eval expr env))
         (let a1 (. res))
         (let expr a2)
         (let res (call eval expr env))
         (let a2 (. res))
         (return (; (; a1 a2) env)))

(do-car (let a1 (. (, expr)))
        (let expr a1)
        (let res (call eval expr env))
        (let a1 (. res))
        (return (; (. a1) env)))

(do-cdr (let a1 (. (, expr)))
        (let expr a1)
        (let res (call eval expr env))
        (let a1 (. res))
        (return (; (, a1) env)))

(do-atom (let a1 (. (, expr)))
         (let expr a1)
         (let res (call eval expr env))
         (let a1 (. res))
         (return (; (@ a1) env)))

(do-if (let perm (. (, expr)))
       (let concl (. (, (, expr))))
       (let alt (. (, (, (, expr)))))
       (let expr perm)
       (let res (call eval expr env))
       (let perm (. res))
       (if perm do-concl do-alt))
```

```
 (do-concl (let expr concl)
           (let res (call eval expr env))
           (return res))

 (do-alt (let expr alt)
         (let res (call eval expr env))
         (return res))

 (do-label (let var (. (, expr)))
           (let val (. (, (, expr))))
           (let expr val)
           (let res (call eval expr env))
           (let val (. res))
           (let env (AL:update {?!} var val env))
           (return (; val env)))

 (do-application (let rator (. expr))
                 (let rands (, expr))
                 (let expr rator)
                 (let res (call eval expr env))
                 (let rator (. res))
                 (let exprs rands)
                 (let argvals (call evlis exprs env))
                 (let argnames (. (, rator)))
                 (let body (. (, (, rator))))
                 (let expr body)
                 (let env (append (AL:new argnames argvals) env))
                 (let res (call eval expr env))
                 (return res))

 (evlis (let evexprs ())
        (goto evlis-loop))

 (evlis-loop (if (= exprs ()) end-evlis evlis-step))

 (evlis-step (let expr (. exprs))
             (let exprs (, exprs))
             (let res (call eval expr env))
             (let evexprs (append evexprs (list (. res))))
             (goto evlis-loop))

 (end-evlis (return evexprs))
)
```

E.g. the following program

```
 (
  (apd (rev a) (rev b))
  (! apd (^ (x y) (if x (; (. x) (apd (, x) y)) y)))
  (! rev (^ (x) (if (@ x) x (apd (rev (, x)) (; (. x) ())))))
 )
```

with variable names list $\ulcorner$`(a b)`$\urcorner$ and inputs list $\ulcorner$`((q w e) (1; 2 3))`$\urcorner$ yield $\ulcorner$`(e w q 3 2 1)`$\urcorner$.

Some more $FCL\langle\mathcal{SE}\rangle$ programs will appear in chapter 3.

2.6 Overview.

So far we have presented three programming languages: of $DRC\langle\mathcal{SE}\rangle$ machine, $drcz_{0/1}$ and $FCL^{(*)}\langle\mathcal{SE}\rangle$. Each of them enables describing procedures, *i.e.* provides mechanisms of capturing: (1) performing "atomic" tasks, (2) conditional choices, and (3) following some (sub)procedure, as was listed in section 1.1.

We have contrasted these mechanisms in the following table:

$DRC\langle\mathcal{SE}\rangle$	$drcz_0$	$FCL\langle\mathcal{SE}\rangle$
constants, primitive operators, lookups	constants, primitive operators, variables	constants, primitive operators, and variables (and return statements)
name, forget	functional objects, and their applications	assignments
apply	application	unconditional jump
select	conditional expression	conditional jump

All three languages can be considered as computation spaces – the one for $DRC\langle\mathcal{SE}\rangle$ was already described in section 1.6. Computation space of $FCL\langle\mathcal{SE}\rangle$ is quite similar:

$$\Sigma_{FCL} = \left\{\langle\beta,\sigma,\pi\rangle : \begin{array}{c}\exists_{\lambda,\tau} findBlock(\lambda,\pi) = \tau :: \beta \\ \wedge\ \sigma \in Env = \mathcal{SE}^{Id} \\ \wedge\ \pi \in Prog_{FCL}\end{array}\right\},$$

$$\Rightarrow_{FCL} = Step,$$

$$enc^{FCL}(\pi,\alpha) = \left\langle \begin{array}{c} lookup(startLabel(\pi),\pi), \\ initStore(inputNames(\pi),\alpha), \\ \pi \end{array} \right\rangle,$$

$$\Pi^{FCL}(\langle\beta,\sigma,\pi\rangle) = \beta.$$

The computation space for $drcz_0$ might seem a bit odd, and to describe it one would probably have to use some variant of Turchin's *activation brackets* (*cf.* [16]).

However all computation spaces for $drcz_{0/1}$ and $FCL^{(*)}$ can be derived easily from $\Sigma_{DRC\langle\mathcal{SE}\rangle}$, with the use of our interpreters $int_{FCL^{(*)}/drcz_{0/1}}$:

$$\Sigma_l = f_l(\langle\pi_{int\langle l\rangle}|Prog_l \times \mathcal{SE}^*\rangle \subsetneq \Sigma_{DRC\langle\mathcal{SE}\rangle})$$

for $l \in \{FCL\langle\mathcal{SE}\rangle, FCL^*\langle\mathcal{SE}\rangle, drcz_0, drcz_1\}$ and some monomorphisms f_l to contract, *i.e.* monotonically embed these spaces in $\Sigma_{DRC\langle\mathcal{SE}\rangle}$. One would often want f_l to be non-trivial, because a single step $\Rightarrow_l$ can be emulated with a [longer] sequence of $\Rightarrow_{\mathfrak{S}}$ steps.

3 On generating and generators

The interpreters of chapter 2 enabled emulating different computational models (languages) on $DRC\langle\mathcal{SE}\rangle$ machine. However, they always introduce some overhead, called in [10], [7] *the interpretive overhead*[42]. It is caused *i.a.* by the need to analyze the structure of program being interpreted *at runtime*[43]. This problem was mostly solved by writing compilers "by hand". In this chapter we will show method capable of automatic interpretive overhead reduction, and automatic compilation (and compiler generation). The method is called *online partial evaluation* and is described in detail in many papers (*e.g.* [9], [7], [6], or [5]). The variants we present are $drcz_1$ implementation of online partial evaluator from [9], and $FCL^*\langle\mathcal{SE}\rangle$ implementation being *mutatis mutandis* the program from [7]. As they are very well documented in the mentioned papers, we will only sketch their implementations, referring interested reader to source codes on attached CD[44].

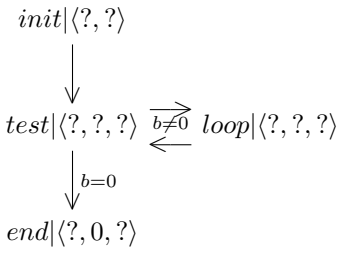
3.1 Abstract interpretation for $FCL\langle\mathcal{SE}\rangle$.

Consider an $FCL\langle\mathcal{SE}\rangle$ program π

```
((a b)
 init

 (init (let res 1)
       (goto test))

 (test (if (= b 0) stop loop))

 (loop (let res (* a res))
       (let b (- b 1))
       (goto test))

 (stop (return res)) )
```

It describes a procedure spanning the subspace $\langle\pi|Num^2\rangle$, such that for any $\langle\varepsilon_1,\varepsilon_2\rangle \in Num^2$ the computation $prc_{FCL}(\pi)(\varepsilon_1,\varepsilon_2) = \langle\pi|(\varepsilon_1,\varepsilon_2)\rangle$ is [embedded] in $\langle\pi|Num^2\rangle$.

The following diagram is simplified representation[45] of $\langle\pi|Num^2\rangle$:



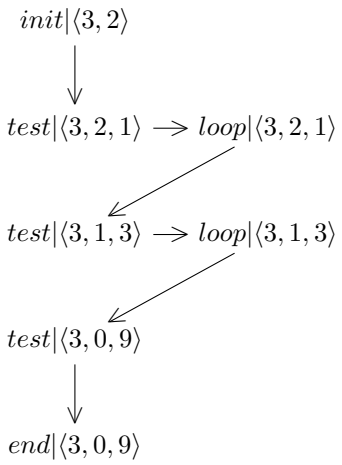
[42] *Cf.* `drcz1-src/FCL/futamura/jones-suboptimality.drcz1`.

[43] Reader interested in checking is advised to run some interpreter in other interpreter, with some simple (single-expression) program as input.

[44] `drcz1_src/FCL/onmix.drcz1` and `drcz1_src/FCL/FCL_src/onmix.fcl` files respectively.

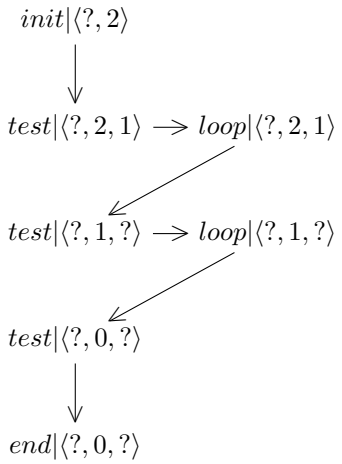
[45] All these diagrams are *mutatis mutandis* ones from [9].

The next diagram represents $\langle \pi | (\ulcorner \mathtt{3} \urcorner, \ulcorner \mathtt{2} \urcorner) \rangle$ *process* – we have derived it by following the procedure's diagram while keeping account of the environment (*i.e.* variable bindings), and by selecting the *test* branch with satisfied condition in label. This procedure is exactly what the $FCL\langle \mathcal{SE} \rangle$ interpreter does.

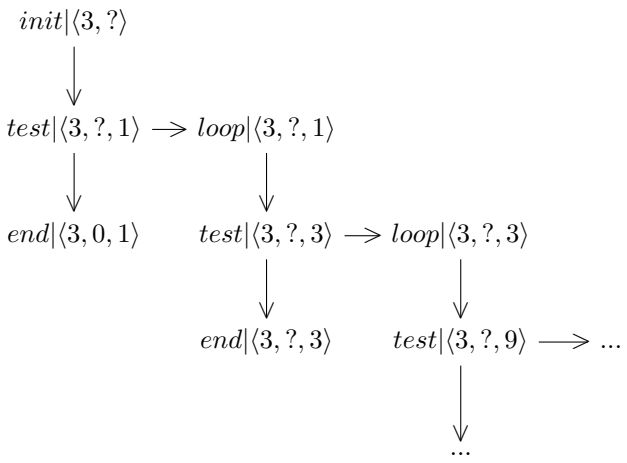


Consider now the subspace $\langle \pi | Num \times \{\ulcorner \mathtt{2} \urcorner\} \rangle$. It can be described with some procedure $\pi_{[.,2]}$, whose upper semantics is projection of π wrt. its first argument, *i.e.* function $f(n) = n^2$.

It can be represented with the following diagram – we have derived it following the procedure's diagram while keeping account of the partial [knowledge of the] environment, and by selecting these of each *test* branches whose condition is either satisfied or of unknown value. Notice we have kept each unique label-environment pair as different block. This procedure, called **driving** is what *i.a.* supercompilers ([16],[5]) and partial evaluators ([10], [9], [7], [5]) do.



Its counterpart, subspace $\langle \pi | \{ \ulcorner \mathtt{3} \urcorner \times Num \} \rangle$, seems more complicated – it turns out its description derived by the *driving* strategy we have picked is not finite.



Therefore a program implementing [this strategy of] driving would not stop (as is the case with our *onmix*). Some interesting techniques were developed for ensuring termination of driving-based programs ([16]) under some additional circumstances[46].

Most often the driving-based programs are code-transformations, *i.e.* code-generating program (as is the case with *onmix*). In other words, the main purpose of driving-based programs is to transform other programs. Various strategies of driving can devise interesting procedures out of a given procedure and some (sub)set of it's possible inputs; these include significant code optimizations (*cf.* [10]).

3.2 $FCL^*\langle \mathcal{SE} \rangle$ online partial evaluator.

We will now show a simple driving-based program in $drcz_1$, implementing online partial evaluation algorithm described in [9]. The fragments for $FCL^*\langle \mathcal{SE} \rangle$ calls are implemented as in [7].

The algorithm is simple: it assumes having two lists – *pend* with states to examine and *code* with resulting block-map. It starts with *pend* containing only one state – initial block with (partial) initial environment, and stops when *pend* is empty. At each step a state from *pend* is taken under consideration (and removed from *pend*). If it has not been seen before, one takes it's label, looks-up it's corresponding block and then drives this block, in accord with the following rules:

[46] Mostly that the processes generated by the driven program would actually halt on all inputs considered.

a. For an assignment $\ulcorner(\texttt{let}\ \alpha\ \varepsilon)\urcorner$:

   If the value ε' of expression ε can be established in current (partial) environment σ, then update it with $\alpha \mapsto \varepsilon'$.

   Otherwise, let $\varepsilon' = \mathit{reduce}(\varepsilon, \sigma)$ where *reduce* is simple partial evaluator for expressions. Then add to the generated block an instruction of form $\ulcorner(\texttt{let}\ \alpha\ \varepsilon)\urcorner$.

   Then proceed with next instruction.

b. For a jump directive $\ulcorner(\texttt{goto}\ \alpha)\urcorner$ proceed[47]. with driving block $\mathit{lookup}(\alpha, \pi)$.

c. for a conditional jump directive $\ulcorner(\texttt{if}\ \varepsilon\ \beta_\top\ \beta_\bot)\urcorner$:

   If the value ε' of expression ε can be established in current (partial) environment σ, then if $\varepsilon' \sim \ulcorner()\urcorner$ proceed as with $(\mathit{goto}\ \beta_\bot)$, otherwise proceed as with $(\mathit{goto}\ \beta_\top)$.

   If the value of ε can not be established, add to the generated block an instruction of the form $\ulcorner(\texttt{if}\ \varepsilon'\ \langle\beta_\top, \sigma\rangle\ \langle\beta_\bot, \sigma\rangle)\urcorner$, where $\varepsilon' = \mathit{reduce}(\varepsilon, \sigma)$.

   Then add the generated block b to *code* under $\langle\lambda, \sigma\rangle$ label (where λ is the label of block from driving of which b has resulted.

d. for return statement $\ulcorner(\texttt{return}\ \varepsilon)\urcorner$ add to the generated block an instruction of the form $\ulcorner(\texttt{return}\ \varepsilon')\urcorner$, where $\varepsilon' = \mathit{reduce}(\varepsilon, \sigma)$. Then add the generated block b to *code* under $\langle\lambda, \sigma\rangle$ label (where λ is the label of block from driving of which b has resulted.

```
(! specialize
   (^ (program div init-vals live-vars-map)
      (PROGRAM:new (diff (PROGRAM:input-names program) div)
                   (LABEL:new2 (PROGRAM:start-pp program)
                               (AL:new div init-vals)
                               live-vars-map)
                   (drive (list (LABEL:new2 (PROGRAM:start-pp program)
                                            (AL:new div init-vals)
                                            live-vars-map))
                          ()
                          (PROGRAM:block-map program)
                          live-vars-map))))

(! successors
 (^ (cb)
    (if cb
        (? ((and (CMD:let? (first cb))
                 (call? (CMD:let-expression (first cb))))
            {=>} (push (call-pp (CMD:let-expression (first cb)))
                       (successors (rest cb))))
           ((CMD:if? (first cb))
            {=>} (list (CMD:if-then-pp (first cb)) (CMD:if-else-pp (first cb))))
           ((CMD:goto? (first cb))
            {=>} (list (CMD:goto-pp (first cb))))
           (T {=>} (successors (rest cb))))
        {else} ()))))
```

[47]This is part of "compress transitions on-the-fly" strategy. We have experimented with variant of *onmix* with no call unfolding on-the-fly, with post-processing compressing, however working with bigger programs seemed to be impossible because of the memory usage.

```
(! drive
   (^ (pend res program live-vars-map)
      (if (empty? pend)
          res
          {else} (let ((  pp {<-} (LABEL:pp (first pend)))
                       (  vs {<-} (LABEL:vs (first pend)) {!})
                       (pend {<-} (rest pend))
                       (          res {<~} res)
                       (      program {<~} program)
                       (live-vars-map {<~} live-vars-map))
                      {in} (if (AL:has-key? (LABEL:new pp vs)
                                            res
                                            LABEL:eq?) { already driven? }
                               (drive pend res program live-vars-map)
                               {else} (let ((new-block
                                             {<-} (drive-block (AL:assoc pp
                                                                         program
                                                                         PP:eq?)
                                                               vs
                                                               program
                                                               live-vars-map))
                                            (    label {<-} (LABEL:new pp vs))
                                            (           vs {<~} vs)
                                            (         pend {<~} pend)
                                            (          res {<~} res)
                                            (      program {<~} program)
                                            (live-vars-map {<~} live-vars-map))
                                           {in} (drive (append (successors new-block)
                                                               pend)
                                                       (AL:add label new-block res)
                                                       program
                                                       live-vars-map)))))))


(! drive-block
   (^ (code vs program live-vars-map)
      (let ((  op {<-} (first (first  code)))
            (args {<-} (rest (first code)))
            (code {<-} (rest code))
            (           vs {<~} vs)
            (      program {<~} program)
            (live-vars-map {<~} live-vars-map))
           {in} (? ((= op 'goto)
                    {=>} (drive-block (AL:assoc (first args) program PP:eq?)
                                      (normalize vs (AL:assoc (first args)
                                                              live-vars-map
                                                              PP:eq?))
                                      program
                                      live-vars-map))
                   ((= op 'return)
                    {=>} (list (CMD:mk-return (reduce (first args) vs))))
                   ((= op 'if)
                    {=>} (drive-if (first args)
                                   (second args)
                                   (third args)
                                   vs program live-vars-map))
                   ((= op 'let)
                    {=>} (drive-let (first args)
                                    (second args)
                                    vs code program live-vars-map))))))
```

```
(! drive-if
   (^ (permise pp1 pp2 vs program live-vars-map)
      (if (static? permise (AL:keys vs))
          (if (eval-expr permise vs) { jedziemy z c.tr.otf! }
              (drive-block (AL:assoc pp1 program PP:eq?)
                           (normalize vs (AL:assoc pp1 live-vars-map PP:eq?))
                           program
                           live-vars-map)
              {else} (drive-block (AL:assoc pp2 program PP:eq?)
                                  (normalize vs (AL:assoc pp2 live-vars-map PP:eq?))
                                  program
                                  live-vars-map))
          {else} (list (CMD:mk-if (reduce permise vs)
                                  (LABEL:new2 pp1 vs live-vars-map)
                                  (LABEL:new2 pp2 vs live-vars-map))))))

(! drive-let
   (^ (var expr vs code program live-vars-map)
      (if (static? expr (AL:keys vs))
          (if (call? expr)
              (drive-block code
                           (STORE:update var
                                         (run-block (call-pp expr)
                                                    vs
                                                    program)
                                         vs)
                                         program
                                         live-vars-map)
              {else} (drive-block code (STORE:update var
                                                     (eval-expr expr vs)
                                                     vs)
                                     program
                                     live-vars-map))
          {else} (if (call? expr)
                     (push (CMD:mk-let var
                                       (list 'call (LABEL:new2 (call-pp expr)
                                                               vs
                                                               live-vars-map)))
                           (drive-block code
                                        (STORE:drop var vs)
                                        program
                                        live-vars-map))
                     {else} (push (CMD:mk-let var (reduce expr vs))
                                  (drive-block code
                                               (STORE:drop var vs)
                                               program
                                               live-vars-map))))))
```

The *onmix* implementation in $FCL^*\langle\mathcal{SE}\rangle$ is *mutatis mutandis* the one from [7]. The only difference is that we require to add, along with the program p to be driven, p's "live variables map" – A-list from block labels to lists of variable identifiers which are significant to that block (*i.e.* which are used in that block before/instead being overwritten). *Cf.* appendix C.

3.3 Why abstract interpretation matters.

The exciting thing about driving is that it can get self-applied – *i.e.* one can drive the driving of some other program. Consider the Ackermann function implementation in $FCL^*\langle\mathcal{SE}\rangle$ from section 2.4:

```
((m n)
 ack
 (ack (if (= m 0) done next))
 (done (return (+ n 1)))
 (next (if (= n 0) ack0 ack1))
 (ack0 (let n 1)
       (goto ack2))
 (ack1 (let n (- n 1))
       (let n (call ack))
       (goto ack2))
 (ack2 (let m (- m 1))
       (let n (call ack))
       (return n)) )
```

We could specialize it *e.g.* with respect to some given value of m, or even better: use the *onmix* in $drcz_1$ to specialize *onmix* in $FCL^*\langle\mathcal{SE}\rangle$ with respect to known program of Ackermann function, and known division[48] $\ulcorner(\mathtt{m})\urcorner$.

```
((init-vals)
21
(21 (let vs (al:new (quote (m)) init-vals))
    (let vs (al:filter-by-keys (quote (m)) vs))
    (let code (call 20))
    (return (; (quote (n)) (; (; (quote ack) vs) code))))

(20 (if (al:has-key? (; (quote ack) vs) (quote ())) 19 18))

(19 (return (quote ())))

(18 (let new-label (; (quote ack) vs))
    (if (fcl:evalexpr (quote (= m 0)) vs) 17 16))

(17 (let new-block (append (quote ())
                           (list (list (quote return)
                                       (fcl:reduce (quote (+ n 1)) vs)))))
    (let code (al:update new-label new-block (quote ())))
    (return code))

(16 (let vs1 (al:filter-by-keys (quote (m)) vs))
    (let vs2 (al:filter-by-keys (quote (m)) vs))
    (let new-block (append (quote ()) (list (list (quote if)
                                                  (fcl:reduce (quote (= n 0)) vs)
                                                  (; (quote ack0) vs1)
                                                  (; (quote ack1) vs2)))))
    (let code (al:update new-label new-block (quote ())))
    (let vs vs1)
    (let code (call 15))
    (let vs vs2)
    (let code (call 7))
    (return code))
```

[48] Division in terminology of [10], [9] and [7] is a list of identifiers of variables which will be known at specialization (*i.e.* driving) time.

```
(15 (if (al:has-key? (; (quote ack0) vs) code) 14 13))

(14 (return code))

(13 (let new-label (; (quote ack0) vs))
    (let vs (al:update (quote n) (fcl:evalexpr (quote 1) vs) vs))
    (let vs (al:update (quote m) (fcl:evalexpr (quote (- m 1)) vs) vs))
    (let value (call 12))
    (let vs (al:update (quote n) value vs))
    (let new-block (append (quote ())
                           (list (list (quote return) (fcl:reduce (quote n) vs)))))
    (let code (al:update new-label new-block code))
    (return code))

(12 (if (fcl:evalexpr (quote (= m 0)) vs) 11 10))

(11 (return (fcl:evalexpr (quote (+ n 1)) vs)))

(10 (if (fcl:evalexpr (quote (= n 0)) vs) 9 8))

(9 (let vs (al:update (quote n) (fcl:evalexpr (quote 1) vs) vs))
   (let vs (al:update (quote m) (fcl:evalexpr (quote (- m 1)) vs) vs))
   (let value (call 12))
   (let vs (al:update (quote n) value vs))
   (return (fcl:evalexpr (quote n) vs)))

(8 (let vs (al:update (quote n) (fcl:evalexpr (quote (- n 1)) vs) vs))
   (let value (call 12))
   (let vs (al:update (quote n) value vs))
   (let vs (al:update (quote m) (fcl:evalexpr (quote (- m 1)) vs) vs))
   (let value (call 12))
   (let vs (al:update (quote n) value vs))
   (return (fcl:evalexpr (quote n) vs)))

(7 (if (al:has-key? (; (quote ack1) vs) code) 14 6))

(6 (let new-label (; (quote ack1) vs))
   (let new-block (append (quote ())
                          (list (list (quote let)
                                      (quote n)
                                      (fcl:reduce (quote (- n 1)) vs)))))
   (let vs (al:drop (quote n) vs))
   (let vs1 vs)
   (let vs (al:filter-by-keys (quote (m)) vs))
   (let new-block (append new-block
                          (list (list (quote let)
                                      (quote n)
                                      (; (quote call)
                                         (; (; (quote ack) vs)
                                               (quote (n))))))))
   (let code (call 5))
   (let vs (al:drop (quote n) vs1))
   (let vs (al:update (quote m) (fcl:evalexpr (quote (- m 1)) vs) vs))
   (let vs1 vs)
   (let vs (al:filter-by-keys (quote (m m)) vs))
   (let new-block (append new-block
                          (list (list (quote let)
                                      (quote n)
                                      (; (quote call)
                                         (; (; (quote ack) vs)
                                               (quote (n))))))))
   (let code (call 1))
```

```
   (let vs (al:drop (quote n) vs1))
   (let new-block (append new-block
                          (list (list (quote return)
                                      (fcl:reduce (quote n) vs)))))
   (let code (al:update new-label new-block code))
   (return code))

(5 (if (al:has-key? (; (quote ack) vs) code) 14 4))

(4 (let new-label (; (quote ack) vs))
   (if (fcl:evalexpr (quote (= m 0)) vs) 3 2))

(3 (let new-block (append (quote ())
                          (list (list (quote return)
                                      (fcl:reduce (quote (+ n 1)) vs)))))
   (let code (al:update new-label new-block code))
   (return code))

(2 (let vs1 (al:filter-by-keys (quote (m)) vs))
   (let vs2 (al:filter-by-keys (quote (m)) vs))
   (let new-block (append (quote ())
                          (list (list (quote if)
                                      (fcl:reduce (quote (= n 0)) vs)
                                      (; (quote ack0) vs1)
                                      (; (quote ack1) vs2)))))
   (let code (al:update new-label new-block code))
   (let vs vs1)
   (let code (call 15))
   (let vs vs2)
   (let code (call 7))
   (return code))

(1 (if (al:has-key? (; (quote ack) vs) code) 14 0))

(0 (let new-label (; (quote ack) vs))
   (if (fcl:evalexpr (quote (= m 0)) vs) 3 2))
)
```

This new program given some particular value for m generates the Ackermann function projection wrt m.

E.g. When given input $\ulcorner((2))\urcorner$ yields:

```
( (n)
  (ack (m . 2))

  ((ack (m . 2)) (if (= n (quote 0)) (ack0 (m . 2)) (ack1 (m . 2))))
  ((ack0 (m . 2)) (return (quote 3)))
  ((ack (m . 1)) (if (= n (quote 0)) (ack0 (m . 1)) (ack1 (m . 1))))
  ((ack0 (m . 1)) (return (quote 2)))
  ((ack (m . 0)) (return (+ n (quote 1))))
  ((ack1 (m . 1)) (let n (- n (quote 1)))
                  (let n (call (ack (m . 1)) n))
                  (let n (call (ack (m . 0)) n))
                  (return n))
  ((ack1 (m . 2)) (let n (- n (quote 1)))
                  (let n (call (ack (m . 2)) n))
                  (let n (call (ack (m . 1)) n))
                  (return n)) )
```

This was an example of *generating extension* – a kind of meta-program, which takes some particular form only after some additional parameters are given.

The special case of generating extension is when the meta-program captures interpreter. Consider an interpreter $i \in Prog_{FCL\langle\mathcal{SE}\rangle}$ for language $\mathcal{L}$ – it takes two inputs: an $\mathcal{L}$-program *prog* and its arguments *args*, and returns the value which *prog* would return in $\mathcal{L}$'s computational model on inputs *args*. If we ask *onmix* in $drcz_1$ to specialize *onmix* in FCL wrt. i and division $\ulcorner(\texttt{prog})\urcorner$, we would get a generating extension of i, *i.e.* a program which given some $\mathcal{L}$-program π yields a projection of i capable of interpreting (executing) program π only. In other words we would get a ($\mathcal{L}$ to $FCL\langle\mathcal{SE}\rangle$) compiler.

This phenomena is known as **the second Futamura projection**.The two remaining projections state:

a. **first Futamura projection**:
   An $\mathcal{L}$ interpreter specialized wrt. to $\mathcal{L}$-program π is compiled version of π.

b. **third Futamura projection**:
   onmix specialized wrt. to *onmix* and division $\ulcorner(\texttt{prog})\urcorner$ is *a compilers generator*, *i.e.* a program which given an interpreter yields a compiler for it.

These are described in detail in *i.a.* [10] and the original [4].

An example – result reproduced from [7] – of generating compiler for Post-machine language is in `drcz1-src/FCL/futamura/post-2-fcl-I-projection.drcz1` file on attached CD.

Even a moderately aggressive technique of (online) partial evaluation can give significant benefits, of which as the most promising we consider:

1. automated "pushing" configuration (*i.e.* constants) into code,
2. possibility to build scalable libraries, which accommodate in accord to their particular use,
3. freedom in constructing elaborate data objects, as all the interpretive overhead should be removed by partial evaluator.

3.4 Specialization examples.

We will now consider a simple example to demonstrate the removal of interpretational overhead. Consider a simple language $\mathfrak{F}$ of arithmetic expressions – it uses reverse polish notation to encode expressions of $n \geq 0$ variables.

operator	stack transformation
$+$	$A, B, C, \ldots \to A + B, C, \ldots$
$*$	$A, B, C, \ldots \to A \times B, C, \ldots$
D	$A, B, \ldots \to A, A, B, \ldots$
R	$A, B, C, \ldots \to B, A, C, \ldots$

```
( (prog args)
  start

  (start (if (= prog ()) end checks))

  (checks (let op (. prog))
          (let prog (, prog))
          (goto check-sum))

  (check-sum (if (= op '+) do-sum check-mul))
  (check-mul (if (= op '*) do-mul check-dup))
  (check-dup (if (= op 'D) do-dup check-rot))
  (check-rot (if (= op 'R) do-rot err))

  (err (return (list 'error 'unknonw 'operator op)))

  (do-sum (let args (; (+ (. args) (. (, args))) (, (, args))))
          (goto start))

  (do-mul (let args (; (* (. args) (. (, args))) (, (, args))))
          (goto start))

  (do-dup (let args (; (. args) (; (. args) (, args))))
          (goto start))

  (do-rot (let args (; (. (, args)) (; (. args) (, (, args)))))
          (goto start))

  (end (return args)))
```

E.g. given program $\ulcorner$`(D * R D * +)`$\urcorner$ and arguments $\ulcorner$`(2 3)`$\urcorner$ yields $\ulcorner$`13`$\urcorner$.

The following listing presents the compilation of $\ulcorner$`(D * R D * +)`$\urcorner$ into $FCL\langle\mathcal{SE}\rangle$ by the first Futamura projection:

```
((args)
 0

 (0 (let args (; (. args) (; (. args) (, args))))
    (let args (; (* (. args) (. (, args))) (, (, args))))
    (let args (; (. (, args)) (; (. args) (, (, args)))))
    (let args (; (. args) (; (. args) (, args))))
    (let args (; (* (. args) (. (, args))) (, (, args))))
    (let args (; (+ (. args) (. (, args))) (, (, args))))
    (return args)))
```

Notice how the whole process of interpreting $\ulcorner$`(D * R D * +)`$\urcorner$ vanished, leaving only a few residual assignments.

The next program is $\mathfrak{F}$ to $FCL\langle\mathcal{SE}\rangle$ compiler generated with second Futamura projection.

```
((init-vals)
 23

 (23 (let vs (al:new (quote (prog)) init-vals))
     (let vs (al:filter-by-keys (quote (prog)) vs))
     (let code (call 22))
     (return (; (quote (args)) (; (; (quote start) vs) code))))

 (22 (if (al:has-key? (; (quote start) vs) (quote ())) 21 20))

 (21 (return (quote ())))

 (20 (let new-label (; (quote start) vs))
     (if (fcl:evalexpr (quote (= prog ())) vs) 19 18))

 (19 (let new-block (append (quote ())
                            (list (list (quote return)
                                        (fcl:reduce (quote args) vs)))))
     (let code (al:update new-label new-block (quote ())))
     (return code))

 (18 (let vs (al:update (quote op) (fcl:evalexpr (quote (. prog)) vs) vs))
     (let vs (al:update (quote prog) (fcl:evalexpr (quote (, prog)) vs) vs))
     (if (fcl:evalexpr (quote (= op (quote +))) vs) 17 6))

 (17 (let new-block (append (quote ())
                            (list (list (quote let)
                                        (quote args)
                                        (fcl:reduce
                                          (quote (; (+ (. args) (. (, args)))
                                                    (, (, args))))
                                          vs)))))
     (let vs (al:drop (quote args) vs))
     (if (fcl:evalexpr (quote (= prog ())) vs) 16 15))

 (16 (let new-block (append new-block
                            (list (list (quote return)
                                        (fcl:reduce (quote args) vs)))))
     (let code (al:update new-label new-block (quote ())))
     (return code))

 (15 (let vs (al:update (quote op) (fcl:evalexpr (quote (. prog)) vs) vs))
     (let vs (al:update (quote prog) (fcl:evalexpr (quote (, prog)) vs) vs))
     (if (fcl:evalexpr (quote (= op (quote +))) vs) 14 13))

 (14 (let new-block (append new-block
                           (list (list (quote let)
                                       (quote args)
                                       (fcl:reduce
                                         (quote (; (+ (. args) (. (, args)))
                                                   (, (, args))))
                                         vs)))))
     (let vs (al:drop (quote args) vs))
     (if (fcl:evalexpr (quote (= prog ())) vs) 16 15))
```

```
(13 (if (fcl:evalexpr (quote (= op (quote *))) vs) 12 11))

(12 (let new-block (append new-block
                           (list (list (quote let)
                                       (quote args)
                                       (fcl:reduce
                                         (quote (; (* (. args) (. (, args)))
                                                   (, (, args))))
                                         vs)))))
    (let vs (al:drop (quote args) vs))
    (if (fcl:evalexpr (quote (= prog ())) vs) 16 15))

(11 (if (fcl:evalexpr (quote (= op (quote d))) vs) 10 9))

(10 (let new-block (append new-block
                  (list (list (quote let)
                              (quote args)
                              (fcl:reduce
                                (quote (; (. args) (; (. args) (, args))))
                                vs)))))
    (let vs (al:drop (quote args) vs))
    (if (fcl:evalexpr (quote (= prog ())) vs) 16 15))

(9 (if (fcl:evalexpr (quote (= op (quote r))) vs) 8 7))

(8 (let new-block (append new-block
                          (list (list (quote let)
                                      (quote args)
                                      (fcl:reduce (quote (; (. (, args))
                                                            (; (. args)
                                                               (, (, args)))))
                                                  vs)))))
   (let vs (al:drop (quote args) vs))
   (if (fcl:evalexpr (quote (= prog ())) vs) 16 15))

(7 (let new-block (append new-block
                          (list (list (quote return)
                                      (fcl:reduce (quote (list (quote error)
                                                               (quote unknown)
                                                               (quote operator)
                                                               op))
                                                  vs)))))
   (let code (al:update new-label new-block (quote ())))
   (return code))

(6 (if (fcl:evalexpr (quote (= op (quote *))) vs) 5 4))

(5 (let new-block (append (quote ())
                          (list (list (quote let)
                                      (quote args)
                                      (fcl:reduce
                                        (quote (; (* (. args) (. (, args)))
                                                  (, (, args))))
                                        vs)))))
   (let vs (al:drop (quote args) vs))
   (if (fcl:evalexpr (quote (= prog ())) vs) 16 15))

(4 (if (fcl:evalexpr (quote (= op (quote d))) vs) 3 2))
```

```
 (3 (let new-block (append (quote ())
                           (list (list (quote let)
                                       (quote args)
                                       (fcl:reduce
                                         (quote (; (. args) (; (. args) (, args))))
                                         vs)))))
    (let vs (al:drop (quote args) vs))
    (if (fcl:evalexpr (quote (= prog ())) vs) 16 15))

 (2 (if (fcl:evalexpr (quote (= op (quote r))) vs) 1 0))

 (1 (let new-block (append (quote ())
                           (list (list (quote let)
                                       (quote args)
                                       (fcl:reduce
                                         (quote (; (. (, args))
                                                   (; (. args)
                                                      (, (, args)))))
                                         vs)))))
    (let vs (al:drop (quote args) vs))
    (if (fcl:evalexpr (quote (= prog ())) vs) 16 15))

 (0 (let new-block (append (quote ())
                           (list (list (quote return)
                                       (fcl:reduce (quote (list (quote error)
                                                                (quote unknonw)
                                                                (quote operator)
                                                                op))
                                                   vs)))))
    (let code (al:update new-label new-block (quote ())))
    (return code))
)
```

Some more examples can be found on the attached CD in `drcz1-src/FCL` directory.

3.5 On compiling $DRC\langle\mathcal{SE}\rangle$ machine code to $FCL^*\langle SE\rangle$.

By the first Futamura projection, given an $DRC\langle\mathcal{SE}\rangle$ interpreter in $FCL^*\langle SE\rangle$ it is possible to compile $DRC\langle\mathcal{SE}\rangle$ programs into $FCL^*\langle SE\rangle$. We have presented such interpreter in example 16 (section 2.4). However it is not very useful in compiling $DRC\langle\mathcal{SE}\rangle$ programs – the resulting procedures contain the whole interpreter as its part. Therefore we have devised a modified version, which turned out to enable compiling subset of $DRC\langle\mathcal{SE}\rangle$ (restricted to first-class [sub]procedure calls). *E.g.*

```
  (PROC (NAME a
         NAME b
         LOOKUP a
         CONST ()
         EQ
         SELECT (LOOKUP b)
                (LOOKUP b
                 LOOKUP a
                 CDR
                 PLOOKUP apd
                 APPLY
                 LOOKUP a
                 CAR
                 CONS)
        FORGET a
        FORGET b)
   PNAME apd
   PLOOKUP apd
   APPLY)
```

compiles to

```
((inputs)
 4

(4 (let d (gen (quote ())))
   (let r inputs)
   (let drc (call 3))
   (return (. (. (, drc)))))

(3 (let drc (call 2))
   (let d (. drc))
   (let r (. (, drc)))
   (let pr (. (, (, drc))))
   (let pd (. (, (, (, drc)))))
   (return (list d r (quote ()) pr pd)))

(2 (let val (. r))
   (let r (, r))
   (let d (al:update (quote a) (; val (al:lookup (quote a) d)) d))
   (let val (. r))
   (let r (, r))
   (let d (al:update (quote b) (; val (al:lookup (quote b) d)) d))
   (let r (; (. (al:lookup (quote a) d)) r))
   (let r (; (quote ()) r))
   (let arg1 (. r))
```

```
   (let arg2 (. (, r)))
   (let r (, (, r)))
   (let res (= arg1 arg2))
   (let r (; res r))
   (let perm (. r))
   (let r (, r))
   (if perm 1 0))

(1 (let r (; (. (al:lookup (quote b) d)) r))
   (let d (al:update (quote a) (, (al:lookup (quote a) d)) d))
   (let d (al:update (quote b) (, (al:lookup (quote b) d)) d))
   (return (list d
                 r
                 (quote ())
                 (quote ())
                 (quote ((apd (name a
                               name b
                               lookup a
                               const ()
                               eq
                               select (lookup b)
                                      (lookup b
                                       lookup a
                                       cdr
                                        plookup apd
                                        apply
                                        lookup a
                                        car
                                        cons)
                               forget a
                               forget b)))))))

(0 (let r (; (. (al:lookup (quote b) d)) r))
   (let r (; (. (al:lookup (quote a) d)) r))
   (let arg1 (. r))
   (let r (, r))
   (let res (, arg1))
   (let r (; res r))
   (let drc (call 2))
   (let d (. drc))
   (let r (. (, drc)))
   (let pr (. (, (, drc))))
   (let pd (. (, (, (, drc)))))
   (let r (; (. (al:lookup (quote a) d)) r))
   (let arg1 (. r))
   (let r (, r))
   (let res (. arg1))
   (let r (; res r))
   (let arg1 (. r))
   (let arg2 (. (, r)))
   (let r (, (, r)))
   (let res (; arg1 arg2))
   (let r (; res r))
   (let d (al:update (quote a) (, (al:lookup (quote a) d)) d))
```

```
   (let d (al:update (quote b) (, (al:lookup (quote b) d)) d))
   (return (list d r (quote ()) pr pd)))
)
```

E.g. when given ⌜(((q w) (1 2 3)))⌝ as arguments yields ⌜(q w 1 2 3)⌝.

In order to achieve such compilation, we had to modify the following:

a) "functionalize the C stack" κ – *i.e.* on ⌜APPLY⌝, instead of appending new code κ' onto κ, it calls another instance of itself with κ' instead of κ.

b) introduced separate δ and ρ registers [not stacks!] for procedures.

The following listing contains this modified $DRC\langle\mathcal{SE}\rangle$ emulator in $FCL^*\langle\mathcal{SE}\rangle$.

```
((program inputs)
 init

 (init (let D (gen ()))
       (let R inputs)
       (let C program)
       (let PR ())
       (let PD ())
       (let DRC (call step D R C PR PD))
       (return (. (. (, DRC))))) { i.e. (. R) }

 (step (if (= C ()) end step2))

 (end (return (list D R C PR PD))) { w sumie zawsze C==()... }

 (step2 (let cmd (. C))
        (let C (, C))
        (goto check-const))

 (check-const   (if (= cmd 'CONST) do-const check-proc))
 (check-proc    (if (= cmd 'PROC) do-proc check-lookup))
 (check-lookup  (if (= cmd 'LOOKUP) do-lookup check-plookup))
 (check-plookup (if (= cmd 'PLOOKUP) do-plookup check-name))
 (check-name    (if (= cmd 'NAME) do-name check-pname))
 (check-pname   (if (= cmd 'PNAME) do-pname check-forget))
 (check-forget  (if (= cmd 'FORGET) do-forget check-pforget))
 (check-pforget (if (= cmd 'PFORGET) do-pforget check-select))
 (check-select  (if (= cmd 'SELECT) do-select check-apply))
 (check-apply   (if (= cmd 'APPLY) do-apply check-cons))
 (check-cons (if (= cmd 'CONS) do-cons check-car))
 (check-car  (if (= cmd 'CAR) do-car check-cdr))
 (check-cdr  (if (= cmd 'CDR) do-cdr check-eq))
 (check-eq   (if (= cmd 'EQ) do-eq check-num))
 (check-num  (if (= cmd 'NUM) do-num check-atom))
 (check-atom (if (= cmd 'ATOM) do-atom check-add))
 (check-add  (if (= cmd 'ADD) do-add check-sub))
 (check-sub  (if (= cmd 'SUB) do-sub check-mul))
 (check-mul  (if (= cmd 'MUL) do-mul check-div))
 (check-div  (if (= cmd 'DIV) do-div check-mod))
 (check-mod  (if (= cmd 'MOD) do-mod check-gt))
 (check-gt   (if (= cmd 'GT) do-gt check-lt))
 (check-lt   (if (= cmd 'LT) do-lt err))

 (err (return (list 'ERROR 'UNKNOWN 'COMMAND cmd)))
```

```
(do-const (let exp (. C))
          (let C (, C))
          (let R (; exp R))
          (goto step))

(do-proc (let proc (. C))
         (let C (, C))
         (let PR (; proc PR)) {!}
         (goto step))

(do-lookup (let var (. C))
           (let C (, C))
           (let R (; (. (AL:lookup var D)) R))
           (goto step))

(do-name (let var (. C))
         (let val (. R))
         (let C (, C))
         (let R (, R))
         (let D (AL:update var (; val (AL:lookup var D)) D))
         (goto step))

(do-forget (let var (. C))
           (let C (, C))
           (let D (AL:update var (, (AL:lookup var D)) D))
           (goto step))

(do-plookup (let var (. C))
            (let C (, C))
            (let PR (; (. (AL:lookup var PD)) PR))
            (goto step))

(do-pname (let var (. C))
          (let proc (. PR))
          (let C (, C))
          (let PR (, PR)) {!}
          (let PD (AL:update var (; proc (AL:lookup var PD)) PD))
          (goto step))

(do-pforget (let var (. C))
            (let C (, C))
            (let PD (; (AL:update var (, (AL:lookup var PD)) PD) ()))
            (goto step))

(do-select (let perm (. R))
           (let R (, R))
           (let concl (. C))
           (let alt (. (, C)))
           (let C (, (, C)))
           (if perm do-concl do-alt))

(do-concl (let C (append concl C))
          (goto step))

(do-alt (let C (append alt C))
        (goto step))

(do-apply (let proc (. PR))
          (let PR (, PR))
          (let oC C)
          (let C proc)
          (let DRC (call step D R C PR PD))
```

```
          (let D (. DRC))
          (let R (. (, DRC)))
          (let PR (. (, (, DRC))))
          (let PD (. (, (, (, DRC)))))
          (let C oC)
          (goto step))

(do-cons (let arg1 (. R))
         (let arg2 (. (, R)))
         (let R (, (, R)))
         (let res (; arg1 arg2))
         (let R (; res R))
         (goto step))

(do-car (let arg1 (. R))
        (let R (, R))
        (let res (. arg1))
        (let R (; res R))
        (goto step))

(do-cdr (let arg1 (. R))
        (let R (, R))
        (let res (, arg1))
        (let R (; res R))
        (goto step))

(do-eq (let arg1 (. R))
       (let arg2 (. (, R)))
       (let R (, (, R)))
       (let res (= arg1 arg2))
       (let R (; res R))
       (goto step))

(do-num (let arg1 (. R))
        (let R (, R))
        (let res (# arg1))
        (let R (; res R))
        (goto step))

(do-atom (let arg1 (. R))
         (let R (, R))
         (let res (@ arg1))
         (let R (; res R))
         (goto step))

(do-add (let arg1 (. R))
        (let arg2 (. (, R)))
        (let R (, (, R)))
        (let res (+ arg1 arg2))
        (let R (; res R))
        (goto step))

(do-sub (let arg1 (. R))
        (let arg2 (. (, R)))
        (let R (, (, R)))
        (let res (- arg1 arg2))
        (let R (; res R))
        (goto step))

(do-mul (let arg1 (. R))
        (let arg2 (. (, R)))
        (let R (, (, R)))
```

```
         (let res (* arg1 arg2))
         (let R (; res R))
         (goto step))

 (do-div (let arg1 (. R))
         (let arg2 (. (, R)))
         (let R (, (, R)))
         (let res (/ arg1 arg2))
         (let R (; res R))
         (goto step))

 (do-mod (let arg1 (. R))
         (let arg2 (. (, R)))
         (let R (, (, R)))
         (let res (% arg1 arg2))
         (let R (; res R))
         (goto step))

 (do-gt (let arg1 (. R))
        (let arg2 (. (, R)))
        (let R (, (, R)))
        (let res (> arg1 arg2))
        (let R (; res R))
        (goto step))

 (do-lt (let arg1 (. R))
        (let arg2 (. (, R)))
        (let R (, (, R)))
        (let res (< arg1 arg2))
        (let R (; res R))
        (goto step))
)
```

3.6 Conclusions and future work.

We hope to have shown some fundamentals of expressing computations from abstract and uniform point of view. Our central goal was reached if the reader got convinced that with the use of mathematical methods we can better understand both what we currently do, and what could be done with mechanical computing devices.

Of course the *drczlang* project will continue. We plan *i.a.* to:

a) experiment with specialization projections (as in [6]),

b) try out techniques of propagating more informations while driving (*e.g.* as in [5]),

c) try out techniques of ensuring termination of driving (*e.g.* homeomorphic embeddings, or as Turchin called them *whistles* ([16]), and author's idea of *role-annotations for variables*),

d) devise supercompiler for $DRC\langle\mathcal{SE}\rangle$ machine (*i.e.* "$DRC\langle\mathcal{SE}\rangle$ meta-machine" – this is already work in progress),

e) try to develop algebraic theory of program interpretation, by first endowing the class of all possible interpreters (in $DRC\langle\mathcal{SE}\rangle$ machine language) with lattice structure – we suspect this could be done, and should clarify the relations between various computation spaces (*e.g.* how they emulate [embed in] each other).

Any results will be reported at our [20].

References

[1] Andreas Blass , Yuri Gurevich , *"Algorithms: A quest for absolute definitions"*, Bulletin of the European Association for Theoretical Computer Science, 2003.

[2] George S. Boolos and Richard J. Jeffrey, *"Computability and Logic (fifth edition)"*, Cambridge University Press, 2007.

[3] Olivier Danvy, Lasse R. Nielsen *"Defunctionalization at work"*, PPDP '01 Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming Pages 162 - 174, ACM New York, 2001.

[4] Yoshihiko Futamura, *"Partial Evaluation of Computation Process—an Approach to a Compiler-Compiler"*, "Systems.Computers.Controls", Volume 2, Number 5, 1971.

[5] Robert Glück, A. V. Klimov, *"Occam's Razor in Metacomputation: the Notion of a Perfect Process Tree"*. Static Analysis 1993, 1993.

[6] Robert Glück, Jesper Jørgensen, *"Generating Transformers for Deforestation and Supercompilation"*. Static Analysis 1994, 1994.

[7] Robert Glück, *"A self-applicable online partial evaluator for recursive flowchart languages"*, John Wiley & Sons, Ltd, 20 JUN 2011.

[8] Andrzej Grzegorczyk *"Zarys logiki matematycznej"*, PWN, 1984.

[9] John Hatcliff, *"An Introduction to Online and Offline Partial Evaluation using a Simple Flowchart Language"*, Partial Evaluation - Practice and Theory, DIKU 1998 International Summer School (Pages 20 - 82), 1998.

[10] N.D. Jones, C.K. Gomard, and P. Sestoft, *"Partial Evaluation and Automatic Program Generation."*, Prentice Hall International, June 1993.

[11] Peter J. Landin, *"The mechanical evaluation of expressions"*, Computer Journal, 6(4):308-320, January 1964.

[12] Peter J. Landin, *"The Next 700 Programming Languages"*, CACM 9(3):157–65, March 1966.

[13] John McCarthy, *"Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I"*, Massachusetts Institute of Technology, Cambridge, Mass. 1, April 1960.

[14] John C. Reynolds *"Definitional interpreters for higher-order programming languages"*, Reprinted from the proceedings of the 25th ACM National Conference: 717–740,1972,

[15] Alfred Tarski *"Badania nad rachunkiem zdań"*, Pisma logiczno-filozoficzne tom 2: Metalogika, PWN 2001,

[16] Valentin F. Turchin, *"The concept of a supercompiler"*. ACM Transactions on Programming Languages and Systems (New York: ACM) 8 (3): 292–325, July 1986.

[17] Alan M. Turing, *"On Computable Numbers, with an application to the Entscheidungsproblem"*, Proceedings of the London Mathematical Society, 1936.

[18] Guy Lewis Steele, Jr. and Gerald Jay Sussman, *"The Art of the Interpreter of, the Modularity Complex (Parts Zero, One, and Two)"*. MIT AI Lab. AI Lab Memo AIM-453, May 1978

[19] `http://plato.stanford.edu/entries/church-turing/` - *"The Philosophy of Computer Science"*, Stanford Encyclopedia of Philosophy, 2008.

[20] `http://wacpawstb.blogspot.com/` - author's project blog.

[21] `https://github.com/drcz/drczlang/` - author's project repository.

Appendix A – on $DRC\langle\mathcal{SE}\rangle$ machine implementation

Data structures

We have picked an obvious representation of *S-expressions* with [integer] numerals and [string] identifiers:

```
typedef enum {NUM,SYM,CONS} SEtype;
typedef struct SE SE;

struct SE {
  SEtype type;
  union {
    int num;
    char *str;
    SE *cons[2];
  };
};

#define NIL ((void *)0)

#define type(E) (E)->type
#define numval(E) (E)->num
#define symval(E) (E)->str
#define car(E) (E)->cons[0]
#define cdr(E) (E)->cons[1]
```

Main loop

The whole $DRC\langle\mathcal{SE}\rangle$ emulator works in loop, whose each iteration:

1) checks whether $.C$ contains any more commands (halts if it doesn't),

2) switches on first command, performing it on $.D$, $.R$ and $.C$ 'registers',

3) goes back to (1).

We have implemented (1)-(3) loop by one big switch with goto to its top at the end of each case.

Memory management

Since asking the system for millions of small (m)allocs and frees leads quickly to memory defragmentation and poor performance, we decided to allocate one big heap (mempool) once, divide it into sizeof(SE) chunks, and store pointers to them on stack (memstack). *Cf.* `https://github.com/drcz/drczlang/blob/master/c-src/mempool.c`.

Garbage collector

Each main loops case – implementation of particular opcode – "cleans up" after itself in that it removes unneeded partial results from $.R$ and conses from $.C$ (*cf.* `voidcode_next_*()` procedures in DRCZ.c). The environment look-up copies referenced expressions, so that they can get removed along other partial results.

On tail recursion optimization

When SELECT or APPLY don't have succeeding commands [but sometimes a bunch of FORGETs], saving empty remaining list on $.C$ seems redundant, therefore we can omit it; for this purpose we use two alternative operators SRELECT and TRAPPLY:

```
6')   < D, (t)::R, (srelect B1 B2) > => < D, R, B1 >,
7')  < D, (())::R, (srelect B1 B2) > => < D, R, B2 >,
8')         < D, (P)::R, (trapply) > => < D, R, P >,
```

tr-eliminate.drcz1 program transforms DRC source into equivalent one using SRELECT and TRAPPLY whenever possible. The translation consists of tracking tail recursive calls/conditionals, and then moving whole FORGET-block before the actual call (right-before TRAPPLY or to the end of SRELECTs branches).

Assembler

To avoid string comparison overhead we translate the $DRC\langle\mathcal{SE}\rangle$ code to its "numeric" representation: we translate operators mnemonics for opcodes, and variable names to addresses (their number on occurrence list). We also annotate code with number of variables used in program, so that the machine knows how many stacks should $.D$ contain.

Short manual

```
-= INSTALLATION =-

  cd c-src/
  make clean
  make all
  make install

Everything you need goes to the-thing/.



To use emacs mode for drcz1 just add the following to your .emacs file:
```

```
--------------------------------------------------------------------------------
  (setq scheme-program-name "<path to your drczlang install>/the-thing/drcz1")

  (add-to-list 'auto-mode-alist '("\\.drcz1\\'" . drcz-generic-mode))
  (add-to-list 'auto-mode-alist '("\\.drcz0\\'" . drcz-generic-mode))
  (global-set-key [(f6)] 'run-scheme) ; for example...
  (global-set-key [(f3)] 'drcz-generic-mode) ; ...
  (setq show-paren-delay 0
        show-paren-style 'parenthesis)
  (show-paren-mode 1)
  (require 'generic-x)
  (define-generic-mode 'drcz-generic-mode
    '(( "{" . "}"))
    '()
    '(("[()]" . font-lock-type-face))
    '()
      nil
     "drczX mode")
--------------------------------------------------------------------------------

-= USAGE EXAMPLES =-

  cd the-thing/

Make sure to chmod +x the following files: comp-d0, comp-d1, desugar, compile,
trelim, drcz1, inliner1, compile-inline, comp-d1-inline.

To run drcz1 interpreter:
  ./drcz1
or
  ./DRCZ d1.bc

To run drcz1 with bigger ,,heap'' use -p <size> option, e.g.:
  ./DRCZ -p 3000000 d1.bc
The size is in CONS cells, not bytes. To check the size of CONS cell on your
system try:
  ./DRCZ -c
Please do remember that memory pool uses stack for storing CONS-cell addresses,
which again takes <size>*sizeof(SE *) bytes, and that currently we still use
malloc/free for strings.

To compile drcz1 program:
  ./compile <source filename>

To compile with naive inliner:
  ./compile-and-inline <source filename>

The former might take quite a lot of memory (does code duplication and does not
preform dead code removal yet).

If all goes well they both produce file named <source filename>.bc; to run it:
  ./DRCZ [-p <size>] <source filename>.bc
```

Appendix B – manual for some of $DRC\langle\mathcal{SE}\rangle$ programs

```
To run the DRC-machine emulator with program <p> and initial R stack <r>, one needs to
first compile it:

 $ cd cd/drcz1-src/
 $ compile DRCZmachine.drcz1

and then simply run the emulator on [real] DRC machine (type in <p> [hit enter], then
type in <r> [enter]):

 $ DRCZ DRCZmachine.drcz1.bc
 Initializing mempool (1048576 cells)...
 Loading code from DRCZmachine.drcz1.bc...
 Allocating 31 env-slot(s) for environments...
 Running the machine.
 code:
 >(NAME x NAME y LOOKUP x LOOKUP x MUL LOOKUP y LOOKUP y MUL ADD)
 stack:
 >(2 3)
 (((x . 2)) (3) ((name y lookup x lookup x mul lookup y lookup y mul add)))
 (((y . 3) (x . 2)) () ((lookup x lookup x mul lookup y lookup y mul add)))
 (((y . 3) (x . 2)) (2) ((lookup x mul lookup y lookup y mul add)))
 (((y . 3) (x . 2)) (2 2) ((mul lookup y lookup y mul add)))
 (((y . 3) (x . 2)) (4) ((lookup y lookup y mul add)))
 (((y . 3) (x . 2)) (3 4) ((lookup y mul add)))
 (((y . 3) (x . 2)) (3 3 4) ((mul add)))
 (((y . 3) (x . 2)) (9 4) ((add)))
 (((y . 3) (x . 2)) (13) (()))
 Result: 13

 Auf wiedersehen!
--------------------------------------------------------------------------------

To use the FCL* interpreter, compile it and run:

 $ cd drcz1-src/FCL/
 $ compile fcl-rec-interpreter.drcz1
 $ DRCZ fcl-rec-interpreter.drcz1.bc
 Initializing mempool (1048576 cells)...
 Loading code from fcl-rec-interpreter.drcz1.bc...
 Allocating 150 env-slot(s) for environments...
 Running the machine.
 program:
 >( (a b) init (init (return (+ (* a a) (* b b)))) )
 input:
 >( 2 3 )
 Result: 13

 Auf wiedersehen!
```

Appendix C – manual for $FCL^*\langle\mathcal{SE}\rangle$ specializer

```
To get started you need to compile the following tools (you should add
the-thing/ directory to your PATH before doing this):

 $ cd drcz1-src/
 $ compile fcl-rec-interpreter.drcz1
 $ compile live-var-analysis.drcz1
 $ compile fcl-onmix.drcz1
 $ compile obierak.drcz1

We first show how to do things manually. When not debugging or examining
the information propagation, one could automate this process by using
-s[ilent] flag of DRCZ machine and pipes.

Consider the following small fcl* program that calculates power function:

  ((a b)
   init

   (init (let res 1)
         (goto test))

   (test (if (= b 0) stop loop))

   (loop (let res (* a res))
         (let b (- b 1))
         (goto test))

   (stop (return res)) )

One can run it with fcl-rec-interpreter.drcz1:

 $ DRCZ fcl-rec-interpreter.drcz1.bc
 Initializing mempool (1048576 cells)...
 Loading code from fcl-rec-interpreter.drcz1.bc...
 Allocating 150 env-slot(s) for environments...
 Running the machine.
 program:
 >((a b)
  init
  (init (let res 1)
        (goto test))
  (test (if (= b 0) stop loop))
  (loop (let res (* a res))
        (let b (- b 1))
        (goto test))
  (stop (return res)) )
 input:
 >(2 3)
 Result: 8

 Auf wiedersehen!
```

```
Suppose we would like to specialize it wrt b=3. We need to compute its
live-variables map [actually we could incorporate live variables
analysis into specializer, but they remain separate for ,,historical''
reasons]:

 $ DRCZ live-var-analysis.drcz1.bc
 Initializing mempool (1048576 cells)...
 Loading code from live-var-analysis.drcz1.bc...
 Allocating 152 env-slot(s) for environments...
 Running the machine.
 >((a b)
  init
  (init (let res 1)
        (goto test))
  (test (if (= b 0) stop loop))
  (loop (let res (* a res))
        (let b (- b 1))
        (goto test))
  (stop (return res)) )

 Result: ((init a b) (test a res b) (loop a res b) (stop res))
-----------------------------------------------------------------------
Now we run onmix and supply it with code, division [known variables'
names], init-vals [resp.] and live-map:

$ DRCZ -p 9000111 fcl-onmix.drcz1.bc
Initializing mempool (9000111 cells)...
Loading code from fcl-onmix.drcz1.bc...
Allocating 170 env-slot(s) for environments...
Running the machine.
program:
>((a b)
 init
 (init (let res 1)
       (goto test))
 (test (if (= b 0) stop loop))
 (loop (let res (* a res))
       (let b (- b 1))
       (goto test))
 (stop (return res)) )

division:
>(b)
init-vals:
>(3)
livemap:
>((init a b) (test a res b) (loop a res b) (stop res))
Result: ((a) (init (b . 3)) ((init (b . 3)) (let res (* a (quote 1)))
                                            (let res (* a res))
                                            (let res (* a res))
                                            (return res)))
-----------------------------------------------------------------------
```

```
The output program uses Cons[program-point-label, store] as labels, we
need to replace them with atomic ones (in our case, numerals):

$ DRCZ obierak.drcz1.bc
Initializing mempool (1048576 cells)...
Loading code from obierak.drcz1.bc...
Allocating 140 env-slot(s) for environments...
Running the machine.
>((a) (init (b . 3)) ((init (b . 3)) (let res (* a (quote 1)))
                                     (let res (* a res))
                                     (let res (* a res))
                                     (return res)))
Result: ((a) 0 (0 (let res (* a (quote 1)))
                  (let res (* a res))
                  (let res (* a res))
                  (return res)))
----------------------------------------------------------------------

As one can see, it doesn't perform any ,algebraic' simplifications.
Nevertheless its outputs can surprise. There remain some termination
issues, eg when one tries to specialize the power program wrt static a
(and dynamic b) - these we plan to solve using homeomorphic embedding
criterion (,,the Russian whistle'').

We can use onmix for producing [generating!] generating extensions.
For example, consider the Ackermann function implementation in ack.fcl
file. Suppose we would like to get a generating extension for Ackermann
projections wrt fixed first [m] argument. All we need is to compute
live-maps for ack.fcl and onmix.fcl
(the former one takes a while):

 $ cat ack.fcl | DRCZ -s live-var-analysis.drcz1.bc > ack-live.tmp
 $ cat onmix.fcl | DRCZ -s live-var-analysis.drcz1.bc > onmix-live.tmp

Now we only need to supply [drcz1's] onmix with [fcl*'s] onmix source,
proper division (with only init-vals dynamic) and the initial values:
ack source, (m) division and ack livemap:

 $ echo "(program division live-map) " > div.tmp
 $ echo "(m) " > div2.tmp
 $ cat onmix.fcl div.tmp lpar ack.fcl div2.tmp ack-live.tmp \
       rpar onmix-live.tmp | DRCZ -sp 9111222 fcl-onmix.drcz1.bc

The resulting program, after peeling off with obierak.drcz1, can get
executed via fcl-rec-interpreter.drcz1; notice that init-vals has
list type, therefore arguments list for the extension has list of
list type, eg. ((3)).

Some more meta-programming experiments you can find in futamura/
directory.

Have fun, good luck!
```