UPPSALA
UNIVERSITET

IT 11 038
Examensarbete 15 hp
December 2011

COMBILOG as a Basis for Visualizing Programming in a Computer-Supported Collaborative Learning Intervention

Mikael Fors

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

COMBILOG as a Basis for Visualizing Programming in a CSCL Intervention

Mikael Fors

We consider the outline of a Computer-Supported Collaborative Learning (CSCL) intervention to be implemented in a university-level module on introductory programming.

Through study of the possibilities in visualizing COMBILOG - a variable-free compositional-relational language suggested by Hamfelt and Nilsson - we investigate the outline of an intervention focusing on general ideas rather than explicit syntax. By careful consideration of the intricate didactical challenges imposed by any such endeavour, we design a target intervention with the aim of supportive integration within a module rather than replacing any pre-existing construct.

Handledare: Andreas Hamfelt
Ämnesgranskare: Lars Oestreicher
Examinator: Anders Jansson
IT 11 038
Tryckt av: Reprocentralen ITC

Contents

1 Introduction **1**
- 1.1 Aim . . . 2
- 1.2 Demarcations . . . 2
- 1.3 Method . . . 3

2 COMBILOG **4**
- 2.1 Combinatory Logic . . . 4
- 2.2 COMBILOG . . . 5
  - 2.2.1 Aritypes . . . 5
  - 2.2.2 Operators . . . 6
  - 2.2.3 Pre-defined Predicates and Combinators . . . 6
- 2.3 Implementation in a Meta-logic Context . . . 7
- 2.4 Visualization . . . 8
  - 2.4.1 Visualization by Håkansson et al. – ViCoLL . . . 11
  - 2.4.2 Visualization by Zetterström . . . 11

3 CSCL **13**
- 3.1 Meaning-making . . . 13
  - 3.1.1 On Long-term Goals . . . 14
- 3.2 Community of Practice . . . 16
- 3.3 Collaboration . . . 16
  - 3.3.1 Communication . . . 17
  - 3.3.2 Collaboration Integration . . . 18
  - 3.3.3 A Clear and Explicit Long-term Goal . . . 20
- 3.4 Learning Process . . . 21
- 3.5 Computer-Supported Design . . . 21
- 3.6 Evaluation . . . 22
  - 3.6.1 Data Logging . . . 22
  - 3.6.2 Questionnaires . . . 23
  - 3.6.3 Outcome . . . 23

4 Proposed Intervention **24**
- 4.1 Outline . . . 24
- 4.2 Proposed Intervention . . . 25
  - 4.2.1 To support a module in higher education . . . 25
  - 4.2.2 Implicitly enforcing collaboration . . . 25
  - 4.2.3 Visualizing COMBILOG in the Intervention . . . 26
  - 4.2.4 Module Integration . . . 28
  - 4.2.5 Evaluation . . . 29

5 Discussion **31**

6 Conclusion **32**

1 Introduction

Consider the intricate process of learning programming for an individual with no prior knowledge of any programming language. For anyone with some experience of the matter, it will seem natural that the first problem – out of many, one might add – that is implicitly introduced is that of syntax[7]. Whether it is intended or not, it is a natural reaction of the pupil to regard the descriptive nature of a new language as a first obstacle to overcome. With the introduction of the first very basic syntax, usually in the neighbourhood of 'hello world', the student is facing the problem of reproducing working software. The tutor or textbook gives an example. The student copies the code with some possible minor alterations. Focus is always on how to *write* the code, not why or what one is actually doing. Noting the very nature of the somewhat mandatory introductory 'hello world' example is evidence enough of the explicit focus on syntax. However, considering the ease with which new programming languages can be learned by users with prior programming experience, it is implicitly implied that syntactic notions are subordinate to the main component of programming. That is, the generic method of approaching a problem space, abstracting it and finding a solution which is then applied through programming language specific syntax.

In this thesis we build on this notion, namely by asking why it is that we focus so much on language in modules teaching introductory programming and not on the generic skills associated with problem solving techniques. Further, we consider the apparent utility of collaboration, which is extremely common when it comes to professional programming application, directly from the get-go. This is done through means of designing and evaluation a proposed intervention, based on didactic theory and ideas central to computer-supported collaborative learning. By intervention we refer to a construct such that it intervenes in the meaning-making process, i.e. an additional supportive portion of a module. Therefore an intervention should not only be viewed as a static construct, in a traditional sense, but also as a subprocess that adds "value" to the total outline. We will focus on the apparent utility of utilizing machines in the meaning-making process that is teaching, taking particular care to restrict our intentions to further emphasize the importance of a result-driven process. Especially, we will consider methods of implicitly enforcing meaning-making through the introduction of long-term goals which will further help guide the student in the module process. Again, we will rely on theory in the field of didactics and as a direct result of our intention of providing a dynamic environment – easily supported through computer-supported means – evaluation methodologies. The focus on evaluation is of great importance when considering the aim of any intervention, namely to produce meaning and should this process be subpar, it *must* be improved considering the target audience and the impact on society in large. If a student is not obtaining sufficient knowledge through the utilization of the proposed intervention we will consider not only the intervention as a failure, but also the entire module. As such, it is obvious that success is crucial.

As made apparent in the title of this thesis, focus is also on COMBILOG which is a combinatory logic language proposed by Hamfelt and Nilsson [9]. The strength of this methodology is that it is variable-free, viz. there is no data being stored apart from programs running. This means that we must regard the programming process as a means of describing

a process, rather than a collection of states. As will be made apparent, this can be quite beneficial should we consider an aim of focusing solely on the generic programming components previously described. Further, additional benefits such as the painless introduction of multi-threading capabilities may be central to future programming needs; considering the explosion of multicore CPUs on the consumer market. Another key aspect of COMBILOG is the ongoing effort of realising visual programming. Since there are no variables, we may disregard any intricate problems with visually representing them. Describing processes through visual means is already common practice within the model languages, e.g. UML. It is our belief that visually representing programs may not only allow easier overview, it can also support a collaborative process as ideas can more easily be conveyed. This idea will be central to our proposed intervention.

1.1 Aim

The aim of this thesis is to evaluate possible visualization methods regarding COMBILOG and how said visualization could be utilized in a computer-supported collaborative learning intervention that is to support the meaning-making process involved in introductory programming. This aim implicitly defines several subquestions that need to be explored:

- What is COMBILOG and what inherit strengths does it possess?
- How could COMBILOG be visualized?
- What are the key points present in CSCL (Computer-Supported Collaborative Learning) and how do they relate to the outlining of a proposed intervention?
- Considering the target module – one focusing on introductory programming – what aspects of the meaning-making process are of particular interest when designing the supporting software capabilities?
- How do we evaluate the proposed intervention?
- What possible problems may arise?

1.2 Demarcations

While the question of whether or not COMBILOG is an optimal choice as a basis for the proposed area of study, viz. the visualization of programming in a variable-free context, is an interesting one; it is far too complex and difficult to be treated in this thesis. We shall, however, briefly explain the benefits of the language in a context with a strong mathematical focus. That is, while the benefits and possibilities are of great interest from a theoretical point of view, it is quite possible that any actual realization would result in practical problems. For instance, there may be performance issues which could possibly render another option – viz. language – more viable in any actual implementation. We acknowledge this possibility, but refrain from considering it further in this thesis.

Further, it should be pointed out that we focus on proposing a theoretical intervention, viz. it is not actually developed in any form. This limitation implies that any issues closely related to the implementation or actual utilization are only considered from a theoretical perspective. It should therefore be concluded that while we certainly do consider possible problems with the proposed intervention, they are by no means the result of any empirical study but should rather be seen as highly probable scenarios. The incompleteness of the possible problems considered follow as a direct result of previously mentioned limitations.

1.3 Method

Considering the questions listed in the aim section, a large portion of the thesis is concerned with the study of previous results by researchers in fields tangent to the topic of this thesis. More explicitly, we shall thoroughly consider the works by Hamfelt and Nilsson as well as look at the visualization proposition by Zetterström. In addition to this, we will consider work done by others in similar situations; specifically the visualization possibilites and utilization of visual programming in teaching programming. Obviously we will also study the intricate details and issues of CSCL as well as some evaluations and outcomes of the implementation and utilization of said techniques in actual modules of various sorts.

The second part of the thesis will deal with the outline of a proposed intervention, based on the ideas and concepts central to the litterature study conducted in part one. Special care will be taken to utilize the didactical theories central to CSCL in order to ensure that the final suggestion will be as dynamic, user-friendly and strong as possible with regards to the meaning-making process.

2 COMBILOG

In this section we give a rough introduction to combinatory logic which is the basis of COMBILOG suggested by Hamfelt and Nilsson. Further, we describe the general direction of the outline of COMBILOG and consider some visualization propositions suggested. Due to the continuous effort in this area, viz. visualization of COMBILOG, we will mainly focus on the work of Zetterström as a basis for our consideration as it contains several interesting aspects central to our cause.

2.1 Combinatory Logic

Combinatory logic is a type of logic first suggested by Schönfinkel [25] and later refined by Curry [3][4][5], in which variables have been eliminated and replaced by application of combinators. A combinator is a function of higher order that is obtained through composition of function application and previously defined combinators. Analogous to any function, a combinator defines an output from its input in the form of arguments.

A combinatory term is given by $\mathsf{T} := x \mid P \mid (\mathsf{T}_1\mathsf{T}_2)$, where x is a variable, P one of the primitive functions and $(\mathsf{T}_1\mathsf{T}_2)$ is the application of combinatory terms T_1 and T_2. Note that the binary composition of application is usually not explicity stated, but rather implicitly enforced with "sloppy" notation $(\mathsf{T}_1 \cdots \mathsf{T}_n)$. Further, it should be pointed out that primitive functions are themselves combinators, viz. when considered from a classical logic perspective all their variables are bound. For every primitive combinator P there is a reduction rule Pe; $(Px_1 \cdots x_n) \vdash X$ where X is a term containing only variables from $\{x_i \mid 1 \leq i \leq n\}$.

The most basic of combinator is the identify combinator **Id**: $(\forall x : \mathsf{T})(\mathbf{Id}\, x) \vdash x$. Another example is the constant combinator, in classical logic defined as $C_n^m(p_1 \cdots p_m) = n$ (with $n \leq m$), **K** which is defined as: $(\forall x_1, x_2 : \mathsf{T})(\mathbf{K}\, x_1\, x_2) \vdash x_1$ (so it is basically projection). Note the ease with which we may extend our combinators regarding arity without actually having to change our notation, a direct result of the implicit enforcement of binary application.

A third very important combinator which is central to COMBILOG is the application combinator, usually denoted **S**. It says $(\forall x_1, x_2, x_3 : \mathsf{T})(\mathbf{S}\, x_1\, x_2\, x_3) \vdash (x_1\, x_3\, (x_2\, x_3))$. Note that **Id** is defined from **K** and **S** by composition. Indeed, the two main axioms of combinatory logic are **K** and **S** which can be stated as:

$$\begin{array}{ll} (K) & \phi \to (\psi \to \phi) \\ (S) & (\phi \to (\psi \to \theta)) \to ((\phi \to \psi) \to (\phi \to \theta)) \end{array}$$

Should we add the rule $((\phi \to \bot) \to \bot) \to \phi$ and the derivation of modus ponens, we have a complete Hilbert-style deduction system, assuring us of the solid basis of the entire logic system discussed. Finally we mention a very important aspect that is frequent in tangent fields as well, such as type theory (e.g. Martin-Löf [19]), namely the distinction of definitional and propositional equality. We say that two terms are definitionaly equal – denoted by $\bullet \equiv \bullet$ – if they produce the same output for all input (extensional equality).

Two terms are propositionaly equal if they have the same **Id**, viz. since they have proof objects [21] we may reason about their equality (intensional equality). Let us illustrate this distinction with a simple example (where $S(n) = n + 1$). Let

$$\begin{array}{rcl} f(0) & := & 0 \\ f(S(n)) & := & S(f(n)) \end{array} \qquad\qquad g(n) := n$$

We ask ourselves whether $f = g$. Clearly $(\forall n : \mathbb{N})(f(n) = g(n))$ implies that they are extensionally equal. But should we consider computability we note that $f(3) = f(2) + 1 = f(1) + 1 + 1 = f(0) + 1 + 1 + 1 = 0 + 1 + 1 + 1 = 3$ while $g(3) = 3$. So clearly they are intentionally different in that they define different algorithms. It is crucial to differentiate these equalities because intensional equality is decidable whereas extensional equality is not. Consider[1] functions f, g given by

$$f(n) = 0 \qquad\qquad g(n) = \begin{cases} 1 & \text{if } 2n + 4 \text{ is not the sum of two primes} \\ 0 & \text{otherwise} \end{cases}$$

If extensional equality was decidable, we could prove or disprove *Goldbach's Conjecture*[2]. We may obviously replace said conjecture with anything and prove or disprove it directly, an impossibility. This distinction further enhances the implication of equality as being something we consider in terms of computability rather than in terms of the output.

2.2 COMBILOG

COMBILOG distinguishes between single terms $t := x \,|\, c \,|\, cons$, that is, variables x, constants c and compound terms defined by recursion of the list constructor, and predicate terms φ. We consider predicate terms to be predicate identifiers, predicate variables and compound combinator terms $\xi(\varphi_1, \cdots, \varphi_n)$ where ξ is a combinator.

2.2.1 Aritypes

No individual constants or category mappings are explicitly present in COMBILOG programs, but are rather implicitly defined through the definitions of pre-defined predicates. Hamfelt and Nilsson [9] show that COMBILOG can be embedded as an object language in a metalogic programming environment through the utilization of an *apply* functionality. Predicate terms are supplied as arguments to *apply* which results in application of a predicate term to its arguments. For this to work we need to impose some weak type restrictions on the predicates regarding arity, refered to as an *aritype*. Consider the predicate

$$\xi(\varphi_1, \cdots, \varphi_m)$$

[1] We recite a proof used by Olov Wilander in a course on Applied Logic.
[2] See for instance [18].

It would have an aritype $((d_1)\cdots(d_m)d_0)$ with $d_i \in \mathbb{N}$ such that φ_i is of arity d_i. The $d_0 \in \mathbb{N}$ denotes the arity of the resulting predicate term, viz. the number of term arguments accepted by the predicate term.

2.2.2 Operators

Two distinct types of operators are considered: logical and manipulatory. Structure outline is to be described through logical operators, that is e.g. $\wedge, \vee$ and $\exists$. We remind the reader of the basic introduction rules for these operators, adjusted slightly for our framework:

$$\frac{P(X_1,\cdots,X_n)}{\mathbf{or}(P,Q)(X_1,\cdots,X_n)}\vee i_1 \qquad \frac{Q(X_1,\cdots,X_n)}{\mathbf{or}(P,Q)(X_1,\cdots,X_n)}\vee i_2$$

$$\frac{P(X_1,\cdots,X_n) \qquad Q(X_1,\cdots,X_n)}{\mathbf{and}(P,Q)(X_1,\cdots,X_n)}\wedge i \qquad \frac{P(X_1,\cdots,X_n)}{\mathbf{ex}(P)(X_2,\cdots,X_n)}\exists i$$

Both $\wedge i$ and $\vee i$ are defined "as usual", however, the rule for $\exists i$ is somewhat different in that we express the existensial quantification with the variable X_1. We implicitly enforce that the desired aritypes exist, e.g. **or** is overloaded with aritypes of the form $((i)(i)i)$ with $0 \leq i \leq n$ where $n \in \mathbb{N}$ is the maximum arity to be utilized in the desired implementation.

The second type of operators – manipulatory – are used to rearrange the arguments of a predicate. These are essentially the functionality of projection and composition. We list a few examples given by Hamfelt and Nilsson to illustrate the main ideas:

$$\begin{aligned} P(X_1,X_2,X_3,\cdots,X_n) &\vdash \mathbf{inv}(P)(X_2,X_1,X_3,\cdots,X_n) \\ P(X_1,\cdots X_n)\wedge \mathbf{id}(X_1,X_2) &\vdash \mathbf{ref}(P)(X_1,X_3,\cdots,X_n) \\ P(X_1,\cdots X_n) &\vdash \mathbf{rotl}(P)(X_2,\cdots,X_n,X_1) \end{aligned}$$

Hamfelt and Nilsson introduce a generic m-fold indexed *make* operator to generate the operators *ex, inv, ref* and *rotl* (among others) discussed so far. The idea is that

$$Q(X_1,\cdots,X_n) \vdash \mathbf{make}[\mu_1,\cdots,\mu_m](Q)(X_{\mu_1},\cdots,X_{\mu_m})$$

where μ_i $(i \in \mathbf{N}^+)$ are distinct positive index numbers. Clearly

$$\begin{aligned} \mathbf{ex} &= \mathbf{make}[2,\cdots,n](P) \\ \mathbf{inv} &= \mathbf{make}[2,1,\cdots,n](P) \\ \mathbf{ref} &= \mathbf{make}[1,3,\cdots,n](\mathbf{and}(P,\mathbf{make}[1,2,3,\cdots,n](\mathbf{id}))) \end{aligned}$$

2.2.3 Pre-defined Predicates and Combinators

As previously mentioned, we require some pre-defined predicates and combinators to ensure the possibility of composition. In addition to the already discussed operators *or, and* and *make* there are

- The identity predicate $\mathbf{id}(X,X)$.

- The list construction predicate $\mathbf{cons}(X, Y, l(X, Y))$.
- The $\top$ predicate $\mathbf{true}()$.
- The constant predicate, defined for all constants utilized in a considered program, $\mathbf{const}_c(c)$.

We are now ready to define a predicate term φ in its entirety:

$$\begin{aligned} \varphi &:= P \mid C \mid Q \\ P &:= \mathbf{id} \mid \mathbf{cons} \mid \mathbf{true} \\ C &:= \mathbf{const}_c \\ Q &:= (\mathbf{or} \mid \mathbf{and} \mid \mathbf{make})(\phi_1, \cdots, \phi_n) \\ \phi_i &:= \varphi \end{aligned}$$

2.3 Implementation in a Meta-logic Context

COMBILOG can be implemented as a meta-logic environment inside PROLOG. One utilizes a meta-predicate apply, taking as arguments a program and a set of arguments which are to be applied to said program, to ensure the structure previously defined. So a simple program $f(x)$ would be written as $apply(f,\ [X])$ and a typical PROLOG clause

$$f(X) \text{ :- } g(X), h(X).$$

would be written as

$$apply(f,\ [X]) \text{ :- } apply(and(g, h),\ [X]).$$

Note that the composition scheme implicitly enforces the necessity of a projection function, viz. make, if the arity of any of g or h would be different from that of f. Syntax-wise, this is done accordingly, where g has a unary arity but f is binary:

$$apply(f,\ [X_1,\ X_2]) \text{ :- } apply(make([1,\ 2],\ g),\ [X_1,\ X_2]).$$

In this case we simply do not utilize the second argument in our call to g, through the projection functionality of make.

We consider some suggestions proposed by Zetterström [29] regarding how one can implement "syntactic sugar", viz. meta-syntax, which eliminates some of the more intricate situations in COMBILOG.

For instance, we must define all constants explicitly through $apply(const\_a,\ [a])$, which is an obvious problem as any real program would require several constants. Zetterström suggests that any such declaration be hidden from the programmer, as the extensive use of constants would require a significant typing effort of applications (apply of constants) that are trivial. The benefits of this approach are easily illustrated through an example (originally by Zetterström):

1 apply(p_0, [a_1, a_2]) :-
2 apply(p_1, [a_1, a]).

Note in particular that while the composition scheme appears to be broken, seeing as how a_2 is not part of the RHS (right-hand side), the actual implementation would be

$$apply(and(make([1,2],\ p_1),\ make([2,1],\ const\_a)),\ [a_1, a_2]).$$

where *const_a* is defined as previously mentioned.

Zetterström considers how one could further simplify the syntax for combinations with mixed arities and suggests hiding projection when adding unbound arguments directly. For instance, should we have a program f of unary arity and wish to call a second program g with binary arity where one of the arguments passed is known to be a constant, we would like to write

$$apply(f,\ [X_1])\ \text{:-}\ apply(g,\ [X_1,\ a]).$$

directly, rather than invoke projection and write a third function h which we would wrap in f. Again, while it appears as if we have broken the composition scheme, we remind the reader that this is simply meta-syntax and that the actual implementation will adhere to arity restrictions. As pointed out by Zetterström [29] these small changes to the "interface" syntax simplify the task of the programmer significantly as many of the instances where one would have to write wrapper programs are eliminated. This implies fewer lines of code and more comprehendible and thus easier-to-read programs.

In addition to these changes, Zetterström argues that one should allow for more user-friendly recursion operators that to a large degree would hide fold calls, making natural recursion function more intuitive. He also discusses several methods of hiding projections, as they are in a sense very technical and will have little to no impact on the solution strategy (other than being a tedious necessity). We do not consider these in detail, as they do not really affect the topic of this thesis. However, we will point out – in accordance with what Zetterström mentions – that negation is a problem. While Zetterström chooses to define negation as failure, viz. if P fails then $\neg P$ succeeds, we point out the danger in such a strategy and refer to the section of combinatory logic (consider especially the more versatile definition $\neg P \equiv P \to \bot$ and decidability[3]).

2.4 Visualization

The main utility of having a functional language is the ease at which one can generate multi-threaded programs [1][14]; a task becoming increasingly important with the introduction and utilization of multicore CPUs [22]. An obvious problem, however, is the difficult nature of COMBILOG. Not only can it be challenging to imagine the outline of otherwise easily grasped programs; the task of writing them can be difficult as well. Mainly, an issue is the rather non-expressive nature of the language. Writing compositions requiring a vast number of **make** operators will not seem worthwhile. Therefore alternative approaches seem desirable.

[3] $P \to \bot$ is a function! [21]

We consider especially the approach of visualization, as it allows one to get an overview of general program composition without having to deal with the intricate compositional issues present in COMBILOG. Zetterström [29] discusses basic visualization issues and strategies. He points out several key issues that are central to the degree of utility that can be obtained through visualization. Essentially, as will be discussed more in detail in the sections covering CSCL, there is a trade-off between complexity and utility. Therefore we must find an acceptable and thus "balanced" level regarding expressibility while at the same time preserving a visualization which enables one to easily and rather swiftly get a good overview of the program one is viewing.

Hanna [10] explores how to visualize datatypes, such as lists. These are shown as linked lists consisting of "cons head tail" with the tail visually being a pointer to the next link. The strength of this visualization is the apparent ease at which the user can view stored data and thus gain a better insight into what is actually happening when running recursive functions. While we find this approach desirable, we will not look deeper into how – and for that matter *if* – data could and should be visualized.

We consider the following important properties of general compositional-relational programming and how to visualize them:

- Programs
- Arguments
- Composition
- Relations
- Recursion

We need to be able to define distinct programs in an easy fashion, since these essentially compose the major portion of the entire programmatical task in COMBILOG. With this in mind, there are a few points that we need to consider prior to discussing the outline of the actual visualization. First, we need to be able to distinguish between a collection of programs, viz. there must be some sort of identifier associated with each program. In a textual form this is handled through the naming convention, i.e. we assign a name to each program. An easy solution to this issue is to reuse the concept of naming. That is, we simply allow the user to name the visual representations of programs. This is not only beneficial from the perspective of easy recognition and distinction, but also has the added benefit of supporting a large collection of possible identification values. Further, this allows for the programmer to assign *meaningful* identificators to selected portions of the total outline, substantially simplifying the human reading process.

We also must consider the necessity of representing arguments in a meaningful way. Essentially this task requires us to on the one hand assign identificators to each argument and on the other hand allow for arguments to be assigned to explicit programs. Zetterström solves this problem by allowing textual identification of arguments and then visually showing what

program they are associated with. Such a strategy is obviously very useful, as it reduces the risk of screen cluttering.

As already made quite clear, a key strategy utilized in the design of COMBILOG programs is the composition scheme. There are two major issues associated with this, when it comes to visualization:

- Connective composition
- Containment composition

By connective composition we refer to the task of grouping a collection of programs into a single one through the utilization of connectives such as $\wedge$ and $\vee$, e.g.

$$apply(f,\ [X])\ \text{:-}\ apply(and(apply(or(a,\ b),\ [X]),\ c),\ [X]).$$

Such composition is very different to what we call containment composition, viz. letting a program f wrap programs g and h. We must be able to represent both these methods in a meaningful way. For instance, with containment composition we must expect very deep compositional structures, i.e. each sub-program could also be generated by containment composition. This is not the case of connective composition, where we expect a flat structure; obviously with the possibility of individual program terms being composed by containment.

Recursion is a heavily used scheme in many program implementations, and it should therefore be easy to distinguish recursive programs from non-recursive ones. After all, we want the end result to be an inductive structure with meaning: one should – possibly with some effort – be able to "read" the visual representation. All efforts that imply support of this process are thus desirable.

Finally, we must consider the problems associated with relations and how to represent them visually. We know for certain that there needs to be a relation between the arguments of a program and how they are projected on possible sub-programs. Further, when considering recursive programs we must ensure that the relation between the inductive base and the induction step is made clear.

With all the issues outlined, note the utility of several of the proposed meta-level syntax additions by Zetterström. Said "syntactic-sugar" allow us to disregard multiple issues that would otherwise result in unnecessary visual complexity. As an example, consider the concept of "dummy" arguments – required when dealing directly with projection and the composition scheme – and the difficulty of visually denoting them. After all, one would have to distinguish between "dummy" arguments and ones actually utilized. Obviously this issue also implicitly enforces a requirement of more sophisticated tools when generating the visual representations, viz. a more complex development environment. Also, recall our desire to make the visual representations as easy to "read" as possible; cluttering the screen with implicit arguments does not render this task easier.

We look at two different approaches to the visualization of COMBILOG from a perspective where we in particular consider the problems outlined above. Recall that there is yet to be an actual implementation of the techniques discussed and as such we are rather limited in our evaluation of them. However, considering our area of application, viz. implementation withing a computer-supported collaborative learning intervention, we have a rather explicit set of requirements regarding utility.

2.4.1 Visualization by Håkansson et al. – ViCoLL

Håkansson et al. suggest a visualization of COMBILOG which is heavily influenced by Euler-Venn diagrams [11]. One of the main issues with visualization in this manner is the apparent risk of cluttering the screen as a direct result of displaying conjunction and disjunction in the traditional sense, i.e. through the overlapping property. To solve the problem, Håkansson et al. suggest replacing said spatial indicator by logical connectives *and* and *or*. Further, they suggest representation in $\mathbb{R}^3$ as a way of ensuring the limitation of any spatial cluttering.

The actual visual representation of programs is in the form of spheres that are connected by their relationships. The end result is quite analogous to molecules; consisting of atoms and bonds. Names are to be given individual programs to further enhance readability and the make operator is to be made implicit in the interface. That way, one can ensure a working environment for the programmer that does not require too much effort regarding intricate details of COMBILOG. Since the make operator is implicit, one needs to be able to explicitly state the arity of individual programs: Håkansson et al. suggest this be done through textual means directly on the spheres.

We know that every individual COMBILOG program can be constructed through composition and this fact results in a high risk of extensive information being visual, should we decide to "show" programs as they really are. Håkansson et al. avoid this problem by only showing the "inside" of programs should the user explicitly ask for it. There is thus a sort of unfold mechanism to the visualization in that one may unfold individual programs to view intricate details regarding their structure and composition.

One problem with this approach, as pointed out by Zetterström [29] is that many questions are left unanswered. Especially since the actual mapping between the visualization and the underlying language is not explicitly defined. Also, while the structures generated surely would represent programs and their composition, we may ask whether or not the visualization shares anything with the task, viz. the connection between "molecular" structures and programs.

2.4.2 Visualization by Zetterström

By representing each program as a rectangle with a textual name, Zetterström allows easy partition of program composition. Further, he lets placement of argument "boxes" indicate association in addition to the distinction of internal and external arguments. In figure 1 we see the outline of a simple program taking two arguments. Note in particular that Zetter-

ström argues that every program should function as a black box, viz. when looked at from a broader view the explicit functioning of the program is irrelevant; rather we should only concern ourselves with the mapping functionality (output vs. input). However, it should still be possible to view the inside of the box, but only in a context where we actually work with the functionality of said program and not when working on a higher "level". He also allows for each program (box) to have documentation – a good idea when we consider the otherwise difficult task of adding comments to the project.

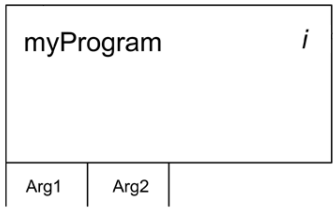

Figure 1: A program box by Zetterström.

In figure 2 it is depicted how Zetterström imagines connective combination. We also see how inner arguments would be positioned – key here is that the empty list utilized inside the program is not visible from the outside. As we cannot alter said list from the outside, there is no reason for us to view it from that context either. This ensures that each component, i.e. program, is separate in the sense that they each handle their own functionality. The conjuncture connective utilized in the depicted example is placed in such a way that it becomes obvious which sub-programs are arguments to it. Since we can always nest binary connectives, we may design connectives of higher arity rather easily. As it might be beneficial to generate such structures in one level, we suggest the introduction of a connective "block" which can be linked to programs in a similar fashion as the arguments are linked to programs in the example by Zetterström. This idea is also shown in figure 2.

Finally, when dealing with recursion, Zetterström introduces – see figure 3 – a box with a zigzag top allowing the user to easily identify which programs have a recursive nature. The $L : List$ argument is also marked as recursive, since we need to explicitly state which argument is decreasing to ensure that the computation will finish. While it is quite possible to let the environment "discover" which argument it is doing recursion on, conflict may arise when we have multiple recursive arguments. A similar situation is evident in this Coq[4] structure:

```
1  Fixpoint plus (n m: nat) {struct m} : nat :=
2        match m with
3              |0 => n
4              |S p => S (plus n p)
5        end.
```

[4]Coq is a formal proof management system, see `http://coq.inria.fr`.

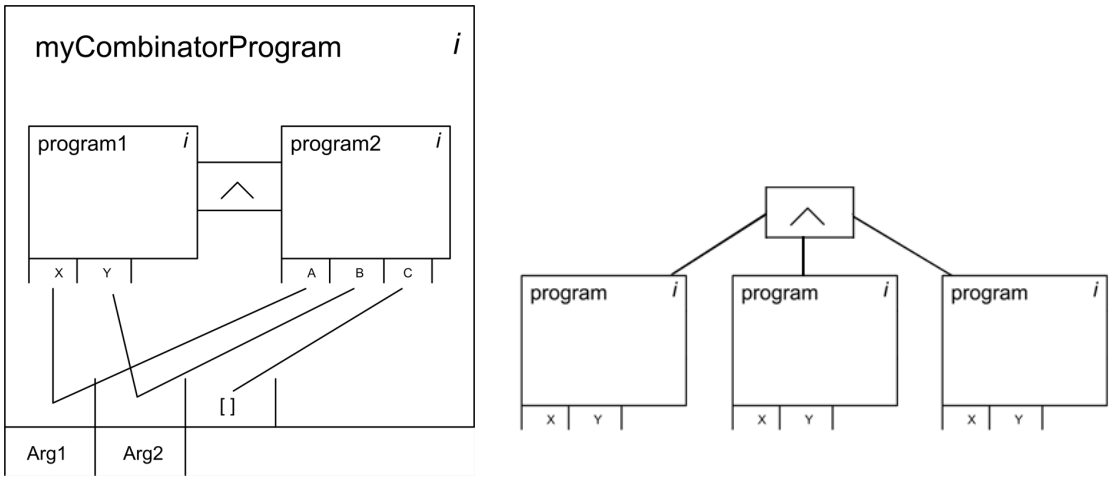

Figure 2: Left: Combination by Zetterström. Right: Multi-Conjecture suggestion.

In Coq, the assistant will try to figure out which argument is decreasing. However, it is not "well-defined" unless we explicitly define the decreasing argument through "struct m". While Zetterström implicitly deals with this issue, we suggest a more clear distinction between recursive elements (in the form of arguments) and decreasing arguments in recursion. In a strict mathematical sense, we would have to indicate a lot of arguments as being recursive, e.g. $N \subset \mathbb{N}$ [2], but that is not equivalent to what Zetterström defines the term as. To avoid confusion it might therefore be easier to let the zigzag-indicator be present on all recursive terms and denote decreasing elements of the recursion operator through some other visual indicator, e.g. an R symbol. This would also allow us to construct inductive or fixpoint structures for arguments in a more explicit sense, opening up many possibilities for advanced constructs.

Note also how a "jump" arrow is used to indicate the accumulator transition between recursive calls. Such a strategy allows us to easier comprehend the process that the diagram is depicting. As previously stated, our goal is to allow easy comprehension of the programs designed, which this method strongly supports. There is also a clear distinction between the recursive base and recursive step, each represented by its own box. These are clearly marked and have a similar outline to that of a program for easier comprehension of the structure.

3 CSCL

In this section we present the ideas and concepts central to Computer-Supported Collaborative Learning. We will pay particular care to notions tangent to our aim and field of interest.

3.1 Meaning-making

The idea of "learning something" is quite vague, in the sense that it is not explicit what it constitutes. For instance, we would not consider learning a fact by heart and *understanding*

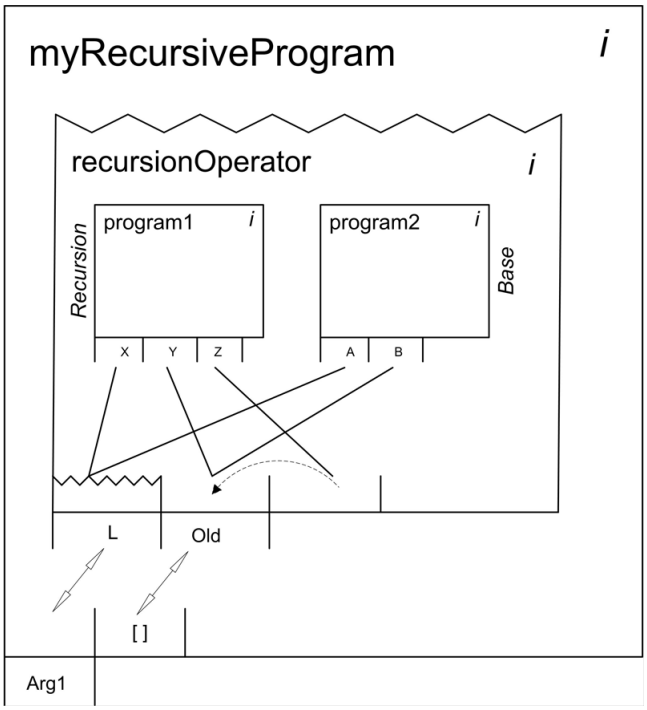

Figure 3: Recursion by Zetterström.

the underlying details central to said fact as equivalent. Indeed, *understanding* and *knowing* imply quite different scenarios. While knowing facts are certainly important, they have little impact in our daily lives unless we can associate them with meaning, viz. understanding the context in which they fit in. With this in mind, we define *meaning-making* as the process in which facts and notions are connected in a manner that allows context grouping. Quite analogous to partitioning a set into equivalence classes.

Said process, viz. meaning-making, is closely related to the work being done in a given module. That is, while the teacher provides facts and ideas, the utilization of them is left to the pupil. Since each person is an individual, the meaning-making process cannot be viewed as static but should rather be seen as a dynamic process; slightly modified for each pupil. This renders any explicit approach towards meaning-making useless: we have to be general. Unfortunately it is usually more difficult to design general approaches than explicit ones. Given that our target pupils are adults, we assume that the pupils' level of responsibility allow us to be rather general in our outline without loss of quality.

3.1.1 On Long-term Goals

When one considers learning or meaning-making in general, the notion of *why* is rarely the first question one asks. However, it is possibly the most important question to ask since if there is not an apparent reason behind the learning, viz. a field or scenario of application, then the entire learning process is a waste of everyones time and effort. With this in mind, we begin by answering the previously mentioned question: why should one learn programming and – to a further extent – what is the domain of application?

In order to answer this question, thereby outlining a possible long-term goal for the meaning-making process covered, we restrict our target pupil domain. In this thesis we will cover an intervention designed for students that partake in university-level courses and as such we implicitly expect a sound level of individual responsibility as well as a real interest in the subject covered, viz. computer science. With the target domain defined, we can more explicitly define the domain of application. We imagine that students are interested in taking a course in programming for any non-empty subset of the following set of reasons:

- **Utility:** The student has a desire to utilize programming in conjuncture with other skills. As an example we may imagine a physics student using his or her knowledge of programming to develop a model of some physical phenomena.
- **Understanding:** The student has an implicit desire to understand the ideas central to that of programming or possibly what it means to program. We may consider, as an example, someone who wishes to work in a field that is tangent to programming, e.g. a managerial position in information technology.
- **Skill:** The student has an explicit desire to work directly with programming or possibly in the computer science field, thus requiring a solid basis of knowledge regarding programming.

We must be careful when considering these listed reasons, as they all implicitly define not only the reason(s) for the student taking a programming module, but also provide important information regarding the extent of interest. For instance, one may assume that a student who has a 'Skill' reason for learning programming has a more keen interest in the underlying theory than that of a student which only requires an understanding of the field. Obviously assuming so is a generalization, but considering the domain of application it is a reasonable assumption. We note the findings of a study conducted by Latterell [16], which describe how students mainly interested in a field as utility to support their primary field of interest altered their focus. That is, it is possible for a student to discover that the field which is only tangent to their main area of study is more interesting to them. Motivation and interest are key elements when it comes to success in most fields, e.g. mathematics [13], so one can imagine a lower rate of failure for motivated students, viz. those who have 'Skill' and 'Utility' reasons.

Regarding the question of *why* one wants to learn programming, we may conclude that it is somewhat dependent on the field of application. However, we can abstract the general reason rather easily by noting that the aim of programming as far as application is concerned is closely related to the notion of problem-solving and – possibly more importantly – problem description. By problem description we refer to the capability of explicitly describing a problem space, e.g. the Maxwell's relations. Said capability is of importance not only to those fields in which programming has application but also as a means in itself, viz. as a way of relating to how one may describe an abstracted situation. With this in mind we, somewhat generally, conclude that one studies programming to gain understanding of how to describe explicit situations and relations in conjuncture with the desire to obtain tools that enable one to solve – or at the very least reduce – said situations or relations.

The concept of a long-term goal as an engine implicitly enforcing the aim of the module is a powerful one. It allows us to place the module in a larger context, enabling us to consider the module outline not only as a collection of concepts that should support the short-term aim of the module itself, but as an integral part of an education. We note the importance of having a solid understanding of said long-term goal as it allows us to determine what process we are to support with our intervention.

3.2 Community of Practice

When considering the design of a computer-supported collaborative learning (CSCL) intervention it is vital to first determine the *community of practice* (CoP) [17] central to the subject area in order to ensure that there is an explicit reasoning for the underlying aim of said intervention. A CoP is said to be defined by three aspects: the domain of the practice, the community in which it is practiced and the shared practice that holds it together. Knowledge is shaped and shared within the CoP, so it is of great interest that it is well known.

3.3 Collaboration

With the reason for studying programming roughly defined, we consider the notion of collaboration and how it can be incorporated in the meaning-making process as far as learning programming is concerned. We previously noted that theory should be included on some level in an intervention hoping to target the field and the defined target group. The reason for this is that on a higher level, viz. education level, theory becomes more important as it allows one to fully understand what one is doing – an important lesson when dealing with nonhomogeneous problems. In such a situation, the notion of the solution is of greater importance than the explicit steps. This can be though of as the underlying theory being more general than any specific algorithm. However, at a first glance it might seem difficult to incorporate this in an intervention; after all, theory is closely related to the old-school of accumulative learning [26]. Obviously one cannot neglect the importance of reading theory, but one can enforce the notion of relating to said theory, viz. discussing it or – better yet – applying it to a problem space of some kind. This is the key introductory point of collaboration when it comes to programming.

The goal of collaboration is thus, as far as an intervention is concerned, to allow the student to explore the theory and the utility of it through social means. This is supported by Stahl [27], who argues that the exploration of issues is a central concept in the study of natural sciences. We note that there are several key issues that need to be discussed and covered when it comes to successfully designing an intervention that allows meaning-making in a social context. These are listed in figure 4 and are treated in the subsections that follow.

- Communication
  - Personal
  - Programming
- Collaboration Integration
  - Collaboration should be *implicit*
  - Enforce individual contribution
  - Enforce the neccessity of idea exchange
- A clear and explicit long-term goal

Figure 4: Key issues for the intervention design.

3.3.1 Communication

Since we are dealing with a social situation, it is important that we allow communication to occur in a natural and supportive way. Not only do we need to consider how to provide such channels of communication regarding what we have denoted *personal communication*, by which we refer to inter-personal communication, but we must also consider how to allow programming-related ideas to be exchanged efficiently. The latter of these points is vital to the success of any intervention dealing with the target subject, because if said communication of programming-related ideas through adequate means is not properly supported, the users will be frustrated [8]. Such a scenario would implicate a poor experience and possibly lead to the failure of the intervention.

Regarding personal communication, i.e. person-to-person communication that is not of programming nature, we need to consider the possible purpose of such interaction and then look at the different communication means available, e.g. textual, video, audio or a combination of these. It is important to convey a natural environment in which the participants feel confident while at the same time not restricted by the communication capabilities available. However, the level of freedom made available regarding means of personal communication depend heavily on the purpose of the intervention, viz. whether participants ought to know with whom they are working or not. For the target audience assumed in this thesis, we intend the extended social context made available to be a complement to the regular module layout and as such there is no reason to consider possible restrictions to the communication channels that are part of said intervention. Since we see the presented intervention as a complement to ordinary[5] teaching, we stress the notion that the intervention should provide additional benefits that are not possible to provide in a non-CSCL environment.

In accordance with our intention, we note that textual communication, with an addition of a possibility of audio support, should suffice. Due to the nature of the general coursework

[5] As in non-CSCL

that is part of any typical programming module, we find the notion of restricting the possibility of face-to-face communication in addition to the utilization of the intervention to be counter-productive. After all, a large portion of programming deals with the concept of applying known solutions or solution strategies to more explicit problem domains. With this in mind, it is obvious that there is no harm in conveying ideas and thoughts amongst students, as long as there is individual progress regarding understanding and application mastery.

As previously mentioned, it is vital to provide users with a possibility to convey programmatical ideas in their natural context. How this is done needs to be carefully considered as any limitations would surely cause some, albeit possibily limited, frustration among the users. Recall that the intervention is to *support* a meaning-making process, not dictate the conditions for it. With this in mind, we must consider the problem of width versus complexity. Should we decide to provide a wide, and thus less restricted, range of input we must accept the complexity that comes with it. If we do not desire complexity, we must settle for a more restricted input language. Since a restricted language is in itself an issue, we must accept that there will be some level of complexity associated with the input support. Recall the central ideas of the visualization of COMBILOG presented in the previous section and how the compositional nature of the underlying language is reflected in both ViCoLL and the outline by Zetterström. While we thus have a rather clear picture of the nature of our target means of visually communicating, we must explicitly explore tangent issues. This, however, is done on an intervention-specific basis and is thus covered when we discuss our proposed intervention.

3.3.2 Collaboration Integration

We have previously discussed the means of communication our intervention should support. Communication is only one expression that has its origins in the notion of collaboration. By carefully regarding the various elements of collaboration, such as we envision it within the field of meaning-making within programming, we explicitly define properties central to our design.

In figure 4 we have stated that collaboration should be implicit. By this we refer to the idea of collaboration as a hidden layer of the intervention. It is an important aspect, but it should not be the aim of the intervention to center around collaboration. Instead, collaboration should be a natural part of the intervention, viz. from the context of the intervention, collaboration should be the natural choice of action – an implicit choice. This is based on the previously mentioned reason for learning programming: to learn and master a skill that can be applied to a field of interest. Since we wish to support this aim, we cannot introduce collaboration for any other purpose.

Collaboration is based around the notion of individual contribution and should individuals not contribute, the entire intervention will fail as the social context is rendered incomplete. This is especially true when we consider the idea of sharing, and thus testing the strength of ideas and reasoning central to programming. If an individual does not provide any feedback on the ideas expressed by others, then there is no exhange of ideas and nothing is generated

in the collaborative sense. Therefore it is important to enforce individual contribution to ensure the exhange of ideas and thus promote the notion of a collaborative environment in which the individual is a non-distinctive part. This is a very central concept that defines the ideas present in computer-supported collaborative learning [28].

To ensure said contribution of all individuals, one should incorporate a strategy that renders lack of contribution meaningless. For instance, if one explicitly imposes that participation is mandatory, there would be no incentive to not participate as that would result in failure of the module. However, one could easily enforce participation implicitly by designing the intervention in such a fashion to require participation. If we design our intervention to center around problem-solving there would need to be some grading mechanism, be it manual or automatic. Should we require all participants that are part of a group to approve any work prior to submission, everyone shares the responsibility of said input. If the submitted work is then used as a basis for the students final grade, there is an apparent incentive to ensure correctness. Obviously even such elaborated design strategies cannot ensure that a student will contribute to the submission. Even with more enhanced submission mechanisms that require participation to be confirmed by other members of the group, one cannot avoid the possibility of non-contributing students. However, considering the target user for the intervention discussed, we – at some point – have to rely on the notion that the participants have a desire to learn and accept the workload that is an integral part of the course.

Closely related to the idea of enforcing individual contribution is the notion of enforcing the necessity of idea exhange, a concept central to any intervention based around the concept of collaboration and social meaning-making. As with enforcing individual contribution, a well-designed intervention should provide an environment in which the exhange of ideas comes naturally. Designing such an intervention requires one to impose a strategy that incorporates implicit benefits of working together and sharing ideas to conquer a common goal. It should never be a good strategy for the individual user to not cooperate with others.

Further, we wish for the exchange of ideas to trigger the individual to reflect on theory. The utility of this concept is easily illustrated through an example involving standard problem-solving, which is a common aspect of any programming module. One is to discover the utility, and thus the application, of the theory through problem sets that consist of problems with elements closely related to the theory covered. The typical university-level textbook in programming will provide partial solutions or output answers to some portion of the problems given in the book. Consider a situation where one verifies the solution one has obtained through application of theory with that of the answer in the book. Should they match, one is assured of the correctness of one's abilities; which is a good thing, seeing as how one can then conclude that the theory covered is understood. However, should the solution in the book not match the output one has produced, a quite different situation arises. First, one is typically inclined to verify the soundness of the initial approach one has taken to the problem. Then the entirety of the calculations and their underlying reasoning are questioned, typically by reflecting on the theory covered. This process results in re-reading the theory from a different viewpoint than before, possibly with a greater insight into the application. Further, one is inclined to "convince oneself" of the soundness of the reasoning

found within the initial, and thus wrong, solution. This process of returning to the theory and verifying one's ideas and reasoning is often extremely rewarding as it does – hopefully – not only result in one solving the problem correctly, but also allows one to gain additional knowledge regarding how to approach problems in general. Goos et al. [6] point out that a similar approach can be utilized from the get-go, i.e. initial failure to implement a solution can trigger a verification step analogous to the process mentioned above.

The question at hand is thus how to design the intervention in such a manner that it implicitly promotes reflection on ideas presented by oneself and others. This concept is closely related to the notion of communication as one has to be able to communicate the reflections made so that others can verify them or, possibly, provide evidence of their incorrectness. Such communication should not only be possible, it should be encouraged. As with the other concepts discussed in this subsection, we note that it is probably easier to design the intervention to promote such critical thinking, rather than having the teacher impose it on the students. In all fairness, it would be very difficult to verify that the students have indeed criticized each others work. However, the entire problem can be reduced greatly by imposing verification prior to submission, viz. all group members have to accept a solution before submission. There should be feedback on the work submitted, either through final grades or by requiring the group to redo subpar submissions. One could also include tasks that require more detailed explanation and justification, for instance a short paper on how to solve a problem algorithmically rather than just submit a program.

3.3.3 A Clear and Explicit Long-term Goal

We have previously argued for feedback regarding any work being submitted or generated through the use of the intervention. The reasoning behind this position is that through implicit incentives we increase the likelihood of individual commitment as well as the exchange of ideas and reasoning central to programming. Since this was concluded to be essentially the purpose of the intervention, it logically follows that the necessity of designing an intervention such that it is utilized in a context where the actual work, viz. demonstration of social meaning-making, affects the grading of the entire module, is absolute. For obvious reasons we cannot enforce any particular utilization. We can, however, recommend a utilization and design our intervention around the notion that any assumptions taken are indeed fulfilled.

Prince [23] states that small teams are a good idea when it comes to active learning but that it is imperative to provide direction. The reasoning being that if one does not know what is being examined it is difficult to plan and thus generate the knowledge through social meaning-making. A situation analogous to having to plan a trip without knowing the destination. This is a solid argument, given that we want the users to explore problem spaces by reasoning and applying theory in a manner that reflects their reasoning for taking the module.

3.4 Learning Process

Given that an intervention is to support a meaning-making process, we must understand what it means to learn something. This is covered in didactic theory, which is both extensive and – unfortunately – an inexact science. With this in mind, one should not expect to find a silver bullet outline regarding what learning and the meaning-making process is, but rather be satisified with eliminating aspects which are *not* central to it.

As discussed briefly in the section on meaning-making, we have a strong desire to outline a general approach when it comes to learning. That is, we do not impose any explicit method, but rather present information which the pupil then has to organize and group in a meaningful context by him or herself. Romero and Barberá [24] argue that not only is the amount of time spent learning positively correlated to the outcome, but also the *quality* of the time. Pupils that are unable to focus on the task at hand, due to other abstractions, tend to have poor results considering the time they invest learning. Romero and Barberá are also strong advocates of allowing flexibility, should the pupils be highly motivated: which we may assume given our target audience.

The use of reflective prompting, viz. asking students about the underlying reasoning behind their input, is a commonly researched area. One such study, by Krause and Stark [15], show that the time spent on learning increased when reflective prompting was introduced to a group of university students: the mean time spent per task went from from 68.47 min (SD 29.03 min) to 74.78 min (SD 27.33 min). However, the time spent did not correlate with test performance. It did increase the reflective outcomes, but due to the nature of the study these were of personal nature (self-reflection). The authors argue that it is difficult to discard the utility of reflective prompting, as the social context was not taken into account; maybe reflection implies greater success in cooperative tasks.

We have strong reason to consider the utility of reflection, flexibility and time (both quantitative- and quality-wise) within a general outline. Further, it should be noted that while these results are encouraging, one cannot evaluate the impact unless the implemented intervention is tested within a module environment.

3.5 Computer-Supported Design

Stahl et al. [28] promote the notion of paying careful attention to the computer-supported portion of CSCL. They point at the importance of utilizing the possibilities enabled by technology and thus construct a learning environment which enables possibilities otherwise not supportable. More explicitly, they state that one should strive to generate new opportunities and not just mimic other means that can be accomplished without the technological tools available. This is a sound argument, as there should be an explicit reasoning for utilizing information technology in teaching. Information technology in itself does not equal "better": it all relates to how it is used.

We must therefore consider what aspects could be beneficial to the study of programming such that their realization rely on technological support. One example provided by Stahl et al. is the idea that information technology allows reconfigurable environments which can offer the user a dynamic experience. Another concept strongly linked to this idea is the notion of automation: allowing direct feedback on input at any time.

3.6 Evaluation

Considering the area of application, viz. higher education, it is imperative that we include an evaluation process to better adapt to any external changes and possible suboptimal elements included in the covered intervention. Supporting a module is a difficult task, especially when considering our particular interest in providing dynamic possibilities through the utilization of information technology. It can be difficult to evaluate a proposed intervention prior to implementation as one has to consider the outcome from the student's perspective. Aspects that might seem very reasonable and understandable from the designers point of view may be viewed as non-logical and hard to grasp by students in the actual implementation. Instead of outlining a set of steps that need to be taken to evaluate every key aspect of a proposed intervention, we consider the apparent strength of information technology: the ease with which we may support a dynamic environment. That is, given an explicit method of evaluation we may alter our intervention in a desired direction as a means of iterative improvement.

While such a strategy, viz. iterative development (improvement), has many apparent benefits there is also one major problem that needs to be overcome: how to evaluate the intervention. There are many suggestions on how this can be done, but essentially there is no silver bullet. Instead, one has to focus on the aim of the intervention and carefully consider the context one is operating in. Since we have already explicitly stated several limitations to our proposed intervention in the aim of this thesis, we refrain from considering methods of evaluation that do not apply.

3.6.1 Data Logging

Holst and Holmer [12] consider the utility of using a continous evaluation method of web-based cooperative learning in order to capture relevant data as soon as possible. They point out the difficulty in trying to evaluate dynamic interventions where variables are constantly changing. To handle the problematic nature of such an intervention, they propose the use of extensive data logging which can be analyzed and summarized to tutors, providing them with a rough overview of how the learning is taking place.

The use of data logging becomes even more apparent when we consider the possibilities of generating complex and explicit graphs of outcome over time. Obviously this would require us to properly determine what outcome would be desirable, should we want to assign any meaning to the output. This is in itself a difficult problem, but assuming that the difficulties associated with it can be overcome, the utility is quite apparent.

Even more interesting are the long-term possibilities of extensive data logging. With enough data one is able to perform basic statistical analysis, allowing rough conclusions to be drawn regarding correlation of predetermined variables and outcome. This is highly desirable, as it fits the idea of an iterative module development process.

3.6.2 Questionnaires

In addition to extensive data logging, Holst and Holmer [12] also suggest the utilization of questionnaires. These can be presented by electronic means to the pupils after key events in the learning process to provide deeper insight into the meaning-making process associated with the module. Said questionnaires could be generated from a database of questions in such a fashion that they properly reflect the current state of progress. This would allow questions to cover critical incidents which may recently have occurred.

While situational questionnaires seem reasonable, seeing as how they enable us to generate a picture of learning as a process over time rather than a static variable, we note the warning of Martinez et al. [20]. They point out that one should pay careful attention not to simplify the evaluation process, given that only considering quantitative data as a basis for the evaluation is rather limiting. In fact, they suggest, one should always strive to complement any quantitative data with a qualitative counterpart. With this in mind, we note the importance of outlining any questionnaires used within the module so that the pupil may submit personal reflections. Utilizing such a strategy allows us to always get a "bigger picture" regarding the overview of the module progression. For instance, poor performance measured through quantitative means would not provide as much insight alone as it would should we also know that the students found the directions for a given assignment hard to comprehend.

3.6.3 Outcome

Measuring the success of learning directly correlated to the intervention is important, but one should not forget about the actual outcome as far as performance is concerned. Such performance is typically measured through a written exam at the end of the module. If the module is offered prior to the introduction of the proposed intervention, comparison of results will be easy. However, one might have to take into account the possibility of the students obtaining other skills through the intervention than those measured on a written exam. Thus it logically follows that it is possible to note equivalent, or at the very least comparable, results before and after the introduction of said intervention and at the same time have students that have gained something out of the introduction, e.g. social skills and confidence in discussing ideas central to programming. These are examples of results that would typically be labeled as desirable, so one should possibly complement the results of a written exam with a final questionnaire where the students have an opportunity to evaluate their personal growth. Prince [23] states that it is hard to measure lifelong effects of problem-based learning, which should also be taken into account.

4 Proposed Intervention

In this section we define the outline of our proposed intervention; one which is to support a module in introductory programming at a higher education level with COMBILOG as basis. Note in particular that it is our desire to hide the underlying technical details of the language and instead focus on the "main" ideas central to programming. While it is indeed the case that we need to limit ourselves regarding programming methodology, our aim is to provide a "timeless" – i.e. not restricted to any explicit language or framework – experience for the user. That way, any knowledge obtained can to a large extent be viewed as generalized and is thus easily applied to any specific area of application.

We begin by explicitly stating the key aspects of our desired intervention, as outlined by the previous sections. Our proposed intervention is then presented and discussed from the framework outlined in the section on CSCL. Particular care is taken to provide intricate details on the underlying didactical ideas, with focus on the aim of utilizing the dynamics made possible through the use of information technology.

4.1 Outline

We list the key aspects of our desired intervention. Focus is on providing a well designed and easily integrated intervention that can support the meaning-making process. Further, we take particular care to consider any probable issues that may arise as a result of utilizing the proposed intervention in a module.

- The intervention should be outlined to support our aim, viz. support a module in introductory programming given in a higher education context.
- The intervention should focus on relationships and schematic outline, rather than intricate and language specific details.
- The intervention should support a collaborative environment. Specifically, we consider the communicatory task central to collaboration and as such we wish to design the intervention in a manner that enables the participants to discuss ideas central to programming.
- The intervention should implicitly enforce the need for collaboration and the benefits of working in small teams.
- There should be an explicit evaluation process to ensure module quality.
- The dynamic possibilities of information technology should be thoroughly utilized throughout the outline.
- Limitations of the intervention should be evaluated to better understand any difficulties regarding integration within a module. Further, the intervention should aim to support and not replace any meaning-making process.

4.2 Proposed Intervention

We are now ready to present our proposed intervention. It will be done by considering the key aspects previously outlined, thus ensuring that our aim is supported. Given that said outline also serves as a means of partitioning the construct, we divide our presentation into several subsections for an easier overview. Each such subsection will target one specific aspects.

4.2.1 To support a module in higher education

First, we consider what it means to support a module in introductory programming in a higher education context. This is important as it allows us to gain an insight into what the pupils will be expecting and also what type of dedication we can expect.

One of the major challenges, as experienced by the author of this thesis while working as a teacher in a module on introductory programming, is that the group of students taking a module is rarely homogeneous. It is very likely that a subset of the group has previous knowledge of programming and may thus demand more difficult tasks combined with more in-depth theory. This is diffcult to support while at the same time provide the "basics" for the ordinary students without prior knowledge. Obviously this situation supports the idea of a general outline, allowing those with prior knowledge to investigate more intricate details of the theory covered.

Possibly the most important notion, however, is that there should be an explicit reason behind the module outline. This outline is not directly linked to the design of our intervention, but should still be considered. That is, we should design the intervention to enhance the idea of underlying reason behind the different tasks supported.

4.2.2 Implicitly enforcing collaboration

One of the main issues with providing a computer-supported collaborative learning intervention is the notion of implicitly enforcing collaboration. For instance, should the intervention be integrated in a poor fashion within the target module, we are likely to end up with a situation in which collaboration is an explicit requirement rather than a means of reaching a target goal. Recall that it is our desire to design an intervention in which collaboration is an integral part. Therefore it would be counter-productive to not carefully outline the design to implicitly enforce the exchange of ideas. If we are successful, the users should choose to collaborate without having to explicitly state it as a formal requirement within the module.

There are several strategies we may choose to follow in order to fulfill this aim. Common to most of them is the idea of defining a set of goals which the pupils have to reach throughout the course of the module. The theory is then that if we outline said goals to be easier to reach with cooperation among the students, we are implicitly enforcing collaboration. Somewhat tanget to this is the idea of an explicit long-term goal, which is to provide the pupil with motivation for reaching target sub-goals during the module. While this certainly sounds obvious and easy, it can be quite challenging to design goals that adhere to said standards.

For our desired intervention, we choose to include four assignments. These are to partition the module into logical portions which each cover different aspects of the programming experience. Note that the actual number of assignments is not central but the notion of module partitioning is. The reason behind this is that one has to allow the pupils to learn from their mistakes, and should we only allow a very limited number of tries, there is little to no room for the students to evaluate their own performance and thus improve. Essentially, this is no different from any other set of problems utilized in module outlines.

Said assignments will not be trivial, in the sense that we will require a decent amount of work by each team in order to pass. However, since we are dealing with collaboration this work effort is analogous to *team* effort rather than individual effort. While some of said effort will be spent on coordinating the team rather than working towards the goal, one can expect several benefits as a result of synergy. To further enhance the notion of the importance of the team process, we introduce an explicit requirement of presenting the work process through textual means. Said report does not have to be extensive, but should rather – in addition to the previously mentioned enhancing notion – be seen as a way of allowing the students to reflect on their problem approach.

We also pay particular care to the nature of our chosen assignment problems, as we do not want to enforce any restrictions to the solution domain. Ambitious teams should, to a large extent, be able to produce extended solutions that cover additional subproblems associated with the main problem space. In accordance with the notions previously presented regarding implicitly enforcing collaboration, we will require all team members to verify the final submission. Also, we will assume that submissions do affect grade outcome.

4.2.3 Visualizing COMBILOG in the Intervention

We find the ideas regarding visualization presented by Zetterström [29] to be viable for our desired aim. While there are certain aspects of his proposed visualization that may be questionable and incomplete, it is still "good enough" in the sense that there is a direct link between the visualization and the code. Any limitations would thus become apparent in an actual implementation. In figure 5 we depict our proposed interface, allowing the users to share ideas visually in real-time. Since we have stated that the means of communication supported should not be limiting, there exists a chat functionality. We refrain from extending this further, but the actual utilization will not differ should one choose to support VoIP communication as a complement to the textual version depicted.

One apparent design problem is how to allow the users to coordinate their different applications, especially since there may be several versions of each. A solution could involve the use of XML[6]. Since the structure of the code in its visual form is composite, we could easily convert it into a tree form. That is, we let the universe – as in the entire picture – be the root and then all programs would be branches. That way, the student could easily share and save their work at other locations. It would also be easy to let the interface require an input XML if the users do not decide to start a new project. Further, one could allow the merging of

[6]Zetterström investigates this notion briefly.

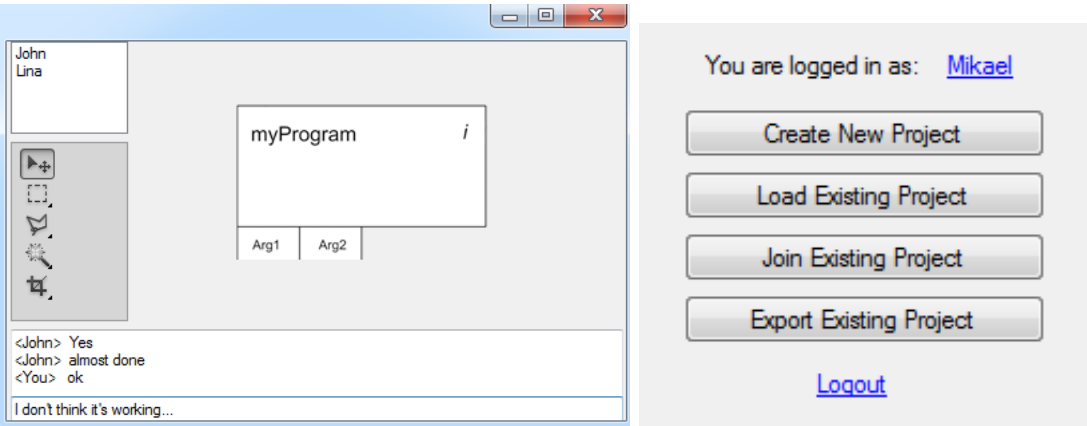

Figure 5: Left: Proposed interface, Right: Initial Options.

branches into one single tree; given that no contradictions emerge. Such functionality would allow us to support a continuous progression model throughout the module, viz. students could "build" upon previous code.

As a direct result of supporting collaboration and using a very modular language, we must expect users to work on different parts of the program simultaneously. However, we also wish to allow them to share their views with each other while keeping track of their own work "place". To solve this problem, we propose a split screen functionality that becomes activated once a team member indicates that he or she wish to share something. In conjuncture with this, we also consider the utility of having markers that indicate user "positions", viz. which program a user is currently viewing. In a sense this would allow implicit allocation of work, as it would become obvious who is working with what subprogram. We also implement a strict position limitation scheme, outlined below:

- A program P can only be the position of one user at a time.
- A subprogram $p \in P$ can only be the position of a user U_1 if either:
  - P is not occupied.
  - P is occupied by U_1.
  - P is occupied by U_2 who allows U_1 to change position to p.
- All programs P can be visited by any number of users, but a visitor has a read-only permission.

This scheme restricts alterations and is in place to reduce the risk of deadlocks and cooperation problems as a result of users altering the same code at the same time.

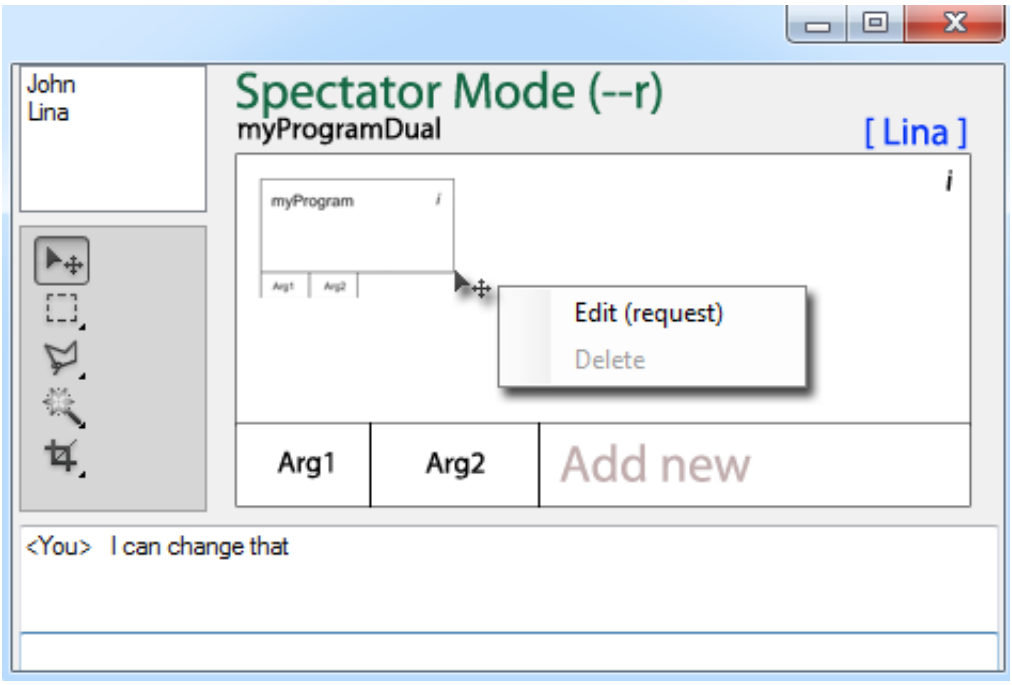

Figure 6: Viewing a program in spectator mode: we see the owner (Lina) as well as the options available to her (add new argument). To edit the subprogram myProgramDual::myProgram a permission request is required.

4.2.4 Module Integration

As previously mentioned, we need to ensure that the intervention *supports* the outline of the target module rather than replaces it. With this in mind we are somewhat restricted unless we design the intervention to be dynamic. Given that we have refrained from explicitly stating any details concerning the course work, other than the number of assignments, we have ensured a dynamic outcome. Further, as an added benefit of allowing program exportation through XML schemes, we have in no way restricted how the intervention is to be used. One could easily imagine using other tools in conjuncture with the intervention suggested.

An additional benefit of COMBILOG, and in particular the visualization of it, is that we have refrained from letting the user deal with syntax specific problems directly. The suggestions of "syntactic-sugar" by Zetterström further enhances this notion. Our aim is to provide an insight into the mental process involved in programming, rather than focusing on a specific environment or framework. Obviously the fact that COMBILOG has a very mathematical nature to it, further supports this aim.

With all this in mind, one should rather easily be able to integrate the proposed intervention in any module targeting introductory programming, with the explicit requirement that the focus is on theoretical approach rather than specific practical knowledge.

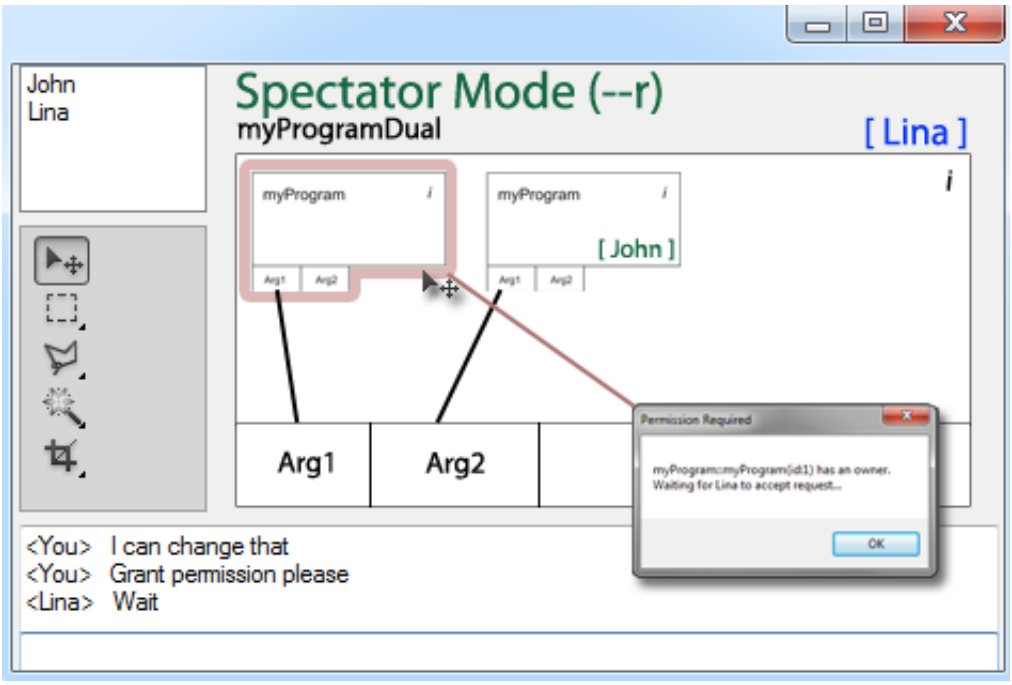

Figure 7: An edit request in progress (to owner Lina). The other instance of myProgram is currently owned by John. We see that Lina has connected some of the outer arguments to the inner programs.

4.2.5 Evaluation

Finally, we must – in accordance with the theory covered in the CSCL section – consider how to evaluate the intervention. Our aim is to ensure continous improvement to the outline and implementation. We recall the benefits of data logging and utilizing questionnaires. One problem outlined was the difficulty of defining what constitutes a desirable outcome, viz. what results we want. Instead of providing an explicit list of outcomes we define a set of "fuzzy" goals. These are used as a rough guideline to what we wish our outcome to be. Note that by not having explicit criteria for these, we ensure an element of non-quantitative nature in our evaluation.

- The student should understand the benefits of a modular solution design.
- The student should understand the importance of coordination when solving a larger problem.
- The student should understand the value of being able to explain a solution approach.
- The student should understand the necessity of criticism.
- The student should accept that problems are usually difficult.
- The student should value approach planning.

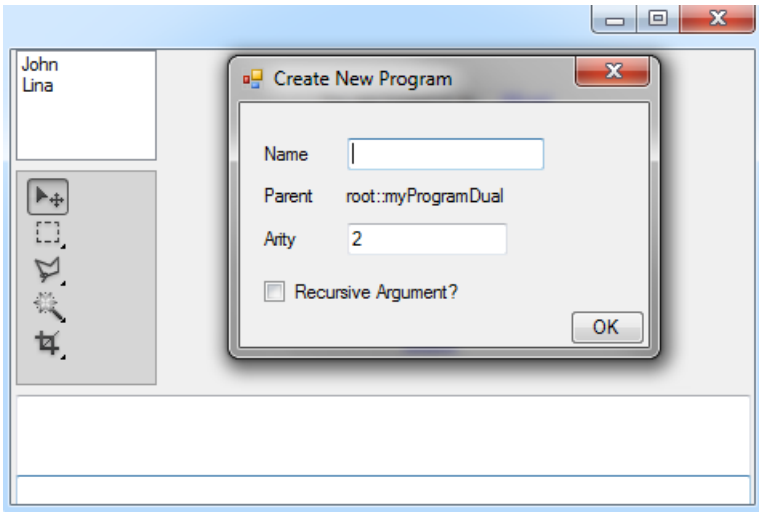

Figure 8: Creating a new program. Note that the visual "size" can be altered by the user later, it has no impact on performance. We allow the arity to be edited later on. By selecting a recursive argument on creation we generate accumulator boxes directly.

- The student should understand problem structures.
- The student should feel confident about reasoning with others.
- The students should feel safe in approaching new problems.

Note in particular our desire to include collaborative outcomes, e.g. confidence in reasoning, understanding the necessity fo criticism and approach planning. These are valuable skills when we consider the tasks involved in actual programming work.

We aim to supply questionnaires to target key events, in our case this is equivalent to assignment submissions. We define three types of questions:

- What worked well?
- What did not work?
- What are your thoughts on the assignment problem?

These are all subjective questions used to complement the quantitative data that can be obtained through the grading of the submissions. We have refrained from making questions that target specific aspects to ensure that we are given the "bigger picture". Details are good, to a certain degree, but we are more interested in the total experience.

In addition to questionnaires being given after submissions, we introduce a final questionnaire to be taken at the end of the module. We can then target specific details, as by then we will already have an overview gained from the other questionnaires and quantitative data obtained through submissions. This allows us to target minor flaws in the implementation

in addition to major ones. The teacher is to generate the final questionnaire in our proposal but obviously one can imagine automating it, given sufficient data. All of the questionnaires will be mandatory to ensure sufficient input.

5 Discussion

With the proposed outline we have covered the main points central to our original aim. However, as have become somewhat apparent within the outline construction, there are several aspects that are not explicit enough. The reason for this is the rather theoretical nature of the area along with a lack of actual implementation regarding visualization. The implementation inside a meta-logical context illustrates the capabilities of COMBILOG but fail to convey all the benefits discussed briefly within this thesis. Therefore it is difficult do discuss possible evaluation regarding actual success as far as utilization goes. Mathematically, it makes sense, but one must recall that we are essentially dealing with consumer response. Whether said consumer is a student or a professional is irrelevant to a large extent, given the aim of the language.

When it comes to proposed intervention, especially the visualization, we note several problems that need to be investigated further:

- How can one ensure an easy overview of the total structure? Imagine a multi-layered universe with several programs present as level 1 nodes. We must consider how said nodes can be navigated with ease, as the picture (i.e. the visualization) would become large. Further, how can a student easily find explicit portions of the total structure, e.g. a level k program inside a specific branch?

- Are there viable alternatives to the restriction scheme presented? If so, how do we test them in an actual implementation? Maybe it is best to allow the utilization of several schemes and thus let the users decide upon project initialization?

- It seems probable that one would still want to be able to access the underlying code during visual work. For instance if we want to "translate" LISP code into a COMBILOG counterpart.

- How do we allow proof of correctness? This is one of the major strengths of the logical basis; but it needs to be carefully considered. Having proof-like capabilities like those present in proof assistants would be very beneficial. If by adhering to type theoretical limitations we could allow visual programming of reasoning, imagine the benefits. Overall, there are a lot of interesting aspects within this area that need to be investigated further.

- Would the utilization of the proposed intervention be more beneficial should it be part of a more advanced course? That is, should we require specific prerequisit knowledge?

In additional to said problems described above, we note that the nature of the target problem is rather extensive and as such – as mentioned in the demarcations section – we

have only generated a theoretical outline that can at best be described as basic. However, given the restrictions imposed, we feel that this is a good start and hopefully others will continue with this work.

6 Conclusion

We have illustrated, through litterature study along with careful consideration of applicable didactic theory, that COMBILOG can be visualized in a manner that could be incorporated in a computer-supported collaborative learning intervention to provide the users with additional communicatory possibilities regarding concepts central to programming. Further, we note the utility of providing a module which has a distinct focus on generalizable – and thus "timeless" – knowledge, in the form of learning how algorithmic outlining can be achieved in various problem spaces. We have stressed the neccesity of a dynamic progression throughout the utilization of the proposed intervention, especially regarding the actual implementation within a pre-existing module.

It has been our desire to approach the target problems from a theoretical viewpoint, providing an insight into the issues central to the subject. Additional research is required regarding practical visualization of COMBILOG to support meaningful experimentation. The goal, obviously, being to produce target models which could be tested and thus evaluated more accurately. With the introduction of the communication possibilities provided by the internet, we consider the development of more intricate and meaningful meaning-making platforms to be inevitable. There is a strong demand for these types of constructs, considering the recent rise in location-independent courses provided by several universities. Never before have the possibilities been greater for implementing powerful supportive tools within offered modules, it all boils down to a question of *how*.

References

[1] Al Zain, A.D.I. Hammond, K. Berthold, J. Trinder, P. Michaelson, G. and Aswad, Mustafa. *Low-pain, high-gain multicore programming in Haskell: coordinating irregular symbolic computations on multicore architectures*. Proceedings of the 4th workshop on Declarative aspects of multicore programming, pp. 25–36, 2008.

[2] Cori, R. and Lascar, D. *Mathematical Logic Part II – Recursion Theory, Gödel's Theorems, Set Theory, Model Theory*. Oxford University Press, Oxford, 2001.

[3] Curry, H. B. *Grundlagen der kombinatorischen Logik. Teil I.* American Journal of Mathematics LII, 1930, pp. 509–536.

[4] Curry, H. B. *Grundlagen der kombinatorischen Logik. Teil II.* American Journal of Mathematics LII, 1930, pp. 789–834.

[5] Curry, H. B. and Feys, R. *Combinatory Logic, volume I.* Studies in Logic and the Foundations of Mathematics. North-Holland, 1958.

[6] Goos, M. Galbraith, P. and Renshaw, P. *A Money Problem: A Source of Insight Into Problem Solving Action*. International Jounral for Mathematics Teaching and Learning, 2000.

[7] Gudmundsen, D. Olivieri, L. and Sarawagi, N. *Reducing the learning curve in an introductory programming course*. J. Comput. Sci. Coll, 27(3), pp. 158–159, 2012.

[8] Guibert, N. Girard, P. and Guittet, L. *Example-based Programming: a pertinent visual approach for learning to program*. Proceedings of the working conference on Advanced visual interfaces, pp. 358–361, 2004.

[9] Hamfelt, A. Nilsson, J.F. and Vitoria, A. *A Combinatory Form of Pure Logic Programs and its Compositional Semantics*. Unpublished, `http://www.anst.uu.se/andhamlt/pub/Combilog.ps`. 1998.

[10] Hanna, K. *Interactive visual functional programming*. SIGPLAN Not., 37(9), pp. 145–156, 2002.

[11] Håkansson, A. Oestreicher, L. Jonsson, T. and Hamfelt, A. *ViCoLL – a Visual Compositional Logic Language*. Human-Centric Computing Languages and Environments, 2001. Proceedings IEEE Symposia, 2001, pp. 394–395.

[12] Holst, S. and Holmer, T. *Continuous Evaluation of Web-based Cooperative Learning: The Conception and Development of an Evaluation Toolkit*. Proceedings of the Conference on Computer Support for Collaborative Learning: Foundations for a CSCL Community, pp. 646–647, 2002.

[13] Korpershoek, H. Kuyper, H. van der Werf, G. and Bosker, R. *Who succeeds in advanced mathematics and science courses?*. British Educational Research Journal. 37(3), pp. 357–380, 2011.

[14] Kraus, J.M. and Kestler, H.A. *Multi-core parallelization in Clojure: a case study*. Proceedings of the 6th European Lisp Workshop, pp. 8–17, 2009.

[15] Krause, U. and Stark, R. *Reflection in example- and problem-based learning: effects of reflection prompts, feedback and cooperative learning*. Evaluation & Research in Education, 23(4), pp. 255–272, 2010.

[16] Latterell, C. *Four Women's Motivation for Obtaining Graduate Degrees in Mathematics*. International Journal for Mathematics Teaching and Learning, 2005.

[17] Lave, J. and Wenger, E. *Situated Learning: Legitimate Peripheral Participation*. Cambridge: Cambridge University Press, 1991.

[18] Li, H. *The Exceptional Set of Goldbach Numbers*. Quarterly Journal of Mathematics, (50), pp. 471–482, 1999.

[19] Martin-Löf, P. *Intuitionistic Type Theory – Notes by Giovanni Sambin*. Bibliopolis, 1984.

[20] Martinez, A. Dimitriadis, Y. Rubia, B. Gomez, E. Garrachon, I. and Marcos, J.A. *Studying Social Aspects of Computer-Supported Collaboration with a Mixed Evaluation Approach*. Proceedings of CSCL'02, pp.631–632, 2002.

[21] Nordström, B. Petersson, K. and Smith, J. M. *Programming in Martin-Löf's Type Theory*. Oxford University Press, Oxford, 1990. (`http://www.cse.chalmers.se/research/group/logic/book/book.pdf`).

[22] Patterson, D. and Hennessy, J.*Computer Organization and Design - The Hardware/Software Interface*. Morgan Kaufmann Publishers, Burlington, USA, 4th ed. 2009.

[23] Prince, M. *Does Active Learning Work? A Review of the Research*. J. Engr. Education, 93(3), pp. 223–231, 2004.

[24] Romero, M. and Barberá, E. *Quality of Learner's Time and Learning Performance Beyond Quantitative Time-On-Task*. International Review of Research in Open & Distance Learning, 12(5), pp. 125–137, 2011.

[25] Schönfinkel, M. *Über die Bausteine der mathematischen Logik*. Mathematische Annalen 92, 1924, pp. 305–316.

[26] Sfard, A. *On Two Methaphors for Learning and the Dangers of Choosing Just One*. Educational Researcher, 27(2), pp.4–13. 1998.

[27] Stahl, G. *Computer Mediation of Collaborative Mathematical Exploration*. Proceedings of the 9th International Conference of the Learning Sciences, (2), pp. 30–33, 2010.

[28] Stahl, G. Koschmann, T. and Suthers, D. *Computer-supported Collborative Learning: An Historical Perspective*. Cambridge handbook of the Learning Sciences, pp. 409–426, 2006.

[29] Zetterström, A. *Visual Compositional–Relational Programming*. Master's Thesis, Uppsala University, 2010.