DNA Tiles, Wang Tiles and Combinators

Marco Bellia and M. Eugenia Occhiuto

Dipartimento di Informatica, Università di Pisa, Italy
{bellia,occhiuto}@di.unipi.it

Abstract. In this paper we explore the relation between Wang Tiles and Schonfinkel Combinators in order to investigate Functional Combinators as an programming language for Self-assembly and DNA computing. We show: How any combinatorial program can be expressed in terms of Wang Tiles, and again, how any computation of the program fits into a grid of tiles of a suitable finite, tile set, and finally, how a program for Self-assembly DNA computing can be obtained. The result is a general methodology that, given any computable function, allows to define a Self-assembly program that can be used to construct the computations of the function

1 Introduction

In the last decade, one of the emerging approaches [1] to DNA Computing, is Self-Assembly [2]. It describes a computation in terms of a process in which small components, autonomously and automatically, assemble into larger, more complex, structures [3–5]. The assembly is based on the Watson-Crick complementary law and is effectively governed by various bio-chemical techniques [6]. However, in terms of computable functions, in the Self-Assembly computation process, it is possible to recognize:

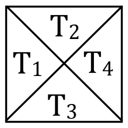
(a) The computed application. The computed application is expressed by the small components to be assembled. In particular, these components include a representation for the function arguments, if any, i.e. the inputs of the application, and a representation for the function to be applied.
(b) The computation. The larger and more complex structures, that result at the end of the SelfAssembly process, form the effective computations. Each of such structures can be read as the complete trace of a computation, from its start to its end.

Various kinds of DNA Tiles has been introduced, in the years, in the various proposals, to be used as the small components of point (a) [7, 8]. In [9] the relation between DNA Tiles (TX, triple crossover, molecules) and Wang Tiles has been used to show how to simulate finite state automata with output, i.e. a transducers, in Wang Tiles. Moreover, by using compositions of transducers and the relation with Wang Tiles, [9] shows how the computation of general recursive functions can be expressed using self-assembly. This allow to use the formalism of general recursive functions as a programming language for DNA computing. With the same aims, [10] introduced DSL as language for programming with the DNA Tiles of the aTAM model [11].

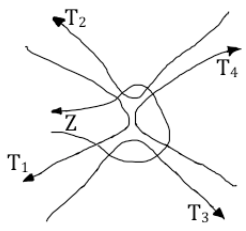
In this paper we explore the relation between Wang Tiles [12] and SKI combinators [13, 14] in order to investigate Functional Combinators [15, 16] as an High-/Intermediate level, programming language for Self-Assembly computations. The result is the definition of a language for Self-Assembly, SKI-Tiles, and of a general methodology that, given any computable function, allows to define a program, in SKI-Tiles, that compute each application of the function, by using Self-Assembly computations.

2 Wang Tiles

Wang Tiles [12] were introduced in 1961. It is a formal system based on the notion of tile. A tile may be graphically represented by a unit square with colored sides from a (possibly, denumerable) set T of distinct colors. Figure 1.a shows the form of a tile such that: *West* side has color T_1, *north* has color T_2, *south* has color T_3 and *east* has color T_4. Tiles must be arranged side by side on the plane (*computation grid*) in a way that adjacent tiles must have the adjacent side of the same color, see Figure 5: We will name this operation *Wang-arrangement*. The interest is on the set $\mathcal{F}$ of all the finite sets of distinct tiles: What tile sets of $\mathcal{F}$, can cover the infinite plane by using Wang-arrangement on copies of the tiles of the set, obtained by translation (no rotation, no reflection). In 1963, Wang showed that to each Turing Machine M corresponds a finite set $T_M \in \mathcal{F}$ such that the computation of M on a tape D can be emulated by a covering, with the (copies of) tiles of T_M, of a plane containing an initial row of tiles that describes D. Finally, Wang proved that the halting problem of Turing Machines can be reduced to the undecidability, for finite tile sets, of covering the infinite plane.



a. Wang Tile: A unit square with the sides colored by T1, T2, T3, T4



b. DNA Tile: 4 strands of DNA, T1, T2, T3, T4, are kept together by a suitable DNA structure, Z.

Fig. 1. Wang Tiles and DNA Tiles

3 SKI Combinators

3.1 The monoid SKI

SKI Combinators [14] is a formal system that expresses all the computable functions[1] without requiring any (bound) variable and by using only one operation: the monadic, functional *application*. Hence, it is the monoid[2] Σ, below:

$$\Sigma = S|K|I|\Pi|X|\Sigma\Sigma$$

where the application is represented by juxtaposition of a (left) term, representing a monadic function, to a the (right) term, representing the argument. Currying, higher order functions, and left associativity of application are provided for non-monadic functions. The symbols S, K, I are combinators (but other ones could be added in [15]), X is a set of (free) variable symbols, Π is a set of constant symbols. The terms of Σ are also called *combinatorial* terms, and the terms built by using the application operator, namely those in $\Sigma\Sigma$, are called (combinatorial) *application* term. Combinators obey to the following *application* laws, for $a, b, c \in \Sigma$:

$$\begin{aligned}&I\ a == a\\ &K\ a\ b == a\\ &S\ a\ b\ c == a\ c\ (b\ c)\end{aligned}$$

3.2 Bracket Abstraction and Bound Variables

The combinators S, K, I express the bracket abstraction in the following way (other characterizations are in [15]). Let $a \in \Sigma$ be any term, possibly containing a (free) variable $x \in X$. Then, we define the bracket abstraction of $a \in \Sigma$ with x, written $[x]a$, be the term $b \in \Sigma$ such that: $b\ x = a$. Such a term[3] always exists in Σ and can be obtained by using the following rules:

$$\begin{aligned}&[x]x = I\\ &[x]u = Ku, \text{ for } u \in \{S, K, I\} \cup \Pi \cup X \text{ and } u \neq x\\ &[x](a\ b) = S([x]a)([x]b)\end{aligned}$$

Hence, all the closed terms of the calculus are all the terms of Σ that do not contain variables.

3.3 Program, Computation, Recursion

Noting that in the application $\Sigma\Sigma$, there is no distinction between the terms that are functions (driving the computation to be done) and those that are arguments (forming the values). Any term becomes the function to be applied, when it is

[1] in its original formulation, in 1924, by Moses I. Schonfinkel, [13], the combinator "I", which could be expressed through SKK, was replaced by the combinator "U", in order to express first order predicates without the use of bound variables.

[2] Also Wang Tiles is a monoid, on Tiles as terms, with Wang-arrangement as the only operation

[3] moreover, for all terms $c \in \Sigma$, we have $b\ c = a[x \leftarrow c]$, i.e. b behaves like one λ-abstraction and when applied to c reduces according to Church's β-axiom [17]

in the left side, whilst it behaves as a value when it is in the right side of the application. A (combinatorial) program is any term of Σ. A program computes according according to the *application* laws of the SKI calculus. In order to obtain a notion of computation, we can encapsulate the application laws into the reduction system obtained by the binary relation on combinatorial terms, $\rightarrow$, defined in the following way. Relation $\rightarrow$ is called *combinatorial reduction.*

Definition 1 ($\rightarrow^*$). *Relation $\rightarrow^*$ is the reflexive and transitive closure of $\rightarrow^*$.*

$$\frac{\mathrm{I\ a == a}}{\mathrm{I\ a \rightarrow a}} \qquad \frac{\mathrm{K\ a\ b == a}}{\mathrm{K\ a\ b \rightarrow a}} \qquad \frac{\mathrm{S\ a\ b\ c == a\ c\ (b\ c)}}{\mathrm{S\ a\ b\ c \rightarrow a\ c\ (b\ c)}}$$

$$\frac{\mathrm{a \rightarrow a'}}{\mathrm{a\ \ b \rightarrow a'\ b}} \qquad \frac{\mathrm{b \rightarrow b'}}{\mathrm{a\ \ b \rightarrow a\ b'}}$$

Given a program a, a computation of a is any sequence, for $n \geq 0$[4]:

$a \rightarrow a_1 \rightarrow ... \rightarrow a_n$

Whenever a_n is such that for no $b \in \Sigma$, $a_n \rightarrow b$, then we say that: Program a has one *terminating* computation; $a \rightarrow a_1 \rightarrow ... \rightarrow a_n$ is a *terminating* computation of a; Program a computes a_n, or equally, a_n is the "value" computed by a. Relation $\rightarrow$ has Church-Rosser confluence property, since if $a \rightarrow a_1 \rightarrow ... \rightarrow a_n$ is a *terminating* computation of a then $a_n \equiv b_m$ for any other *terminating* computation $a \rightarrow b_1 \rightarrow ... \rightarrow b_m$ [18]. However, Σ contains *nonterminating* programs. As a matter of the fact consider the term Ψ of Definition2.

Definition 2 (The Kleene fixed-point combinatorial program calculator, Ψ). *Let $R \equiv S(S(KS)(S(KK)I))(K(SII))$. Then, $\Psi \equiv SRR$ is a combinatorial program. Moreover, Ψ is such that, for all pairs of terms $G, a \in \Sigma$, the following holds:*

(*) $\quad \Psi\ G\ a \equiv G\ (\Psi\ G)\ a$

The proof of (*) is a trivial exercise. Ψ points out the elegance with which Schonfinkel monoid expresses the computable functions. In particular, Ψ introduces *recursively defined* terms on one hand, and computes the least fixed point of them, on the other hand. However, in Section 6, we use term equations for dealing with recursive definitions, because Self-assembly computation has a notion of term replacement that already support recursive definitions.

4 The Approach

We start introducing the structures and the properties that the Wang tiles must have in order to be used for expressing the combinatorial terms and their computation. Then, we show how to use such structures in order to get the definition and the computation of any combinatorial program.

[4] Obviously, $n = 0$ means that for no $b \in \Sigma$, $a \rightarrow b$

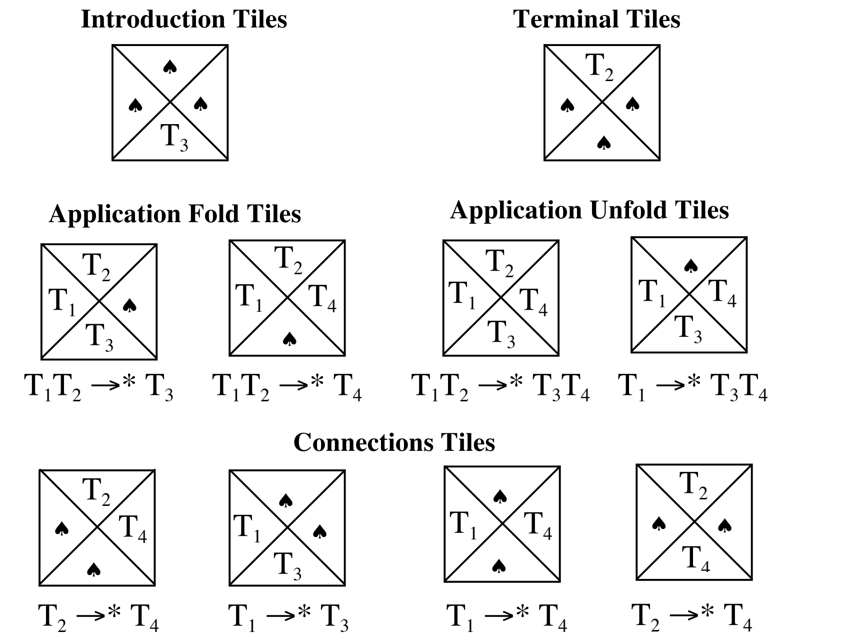
4.1 SKI-Tiles: A formalism of Wang Tiles for combinatorial programs

The colors that may occur in the tiles, are the combinatorial terms (of Σ): Different terms are different colors. In addition, a special color $\spadesuit$ is used for combining the terms within a tile and for arranging the tiles in the computation grids. The sides of a tile may be colored with an input (i.e. the right part of an application term) or with a function (i.e. the left part of an application term) or with an output (the result of an application) or finally, with a connection term (which allows to arrange together distinct tiles and distinct parts of the computation grid). When more different colors occur in a tile, their arrangement in the tile sides obeys properties based on the combinatorial reduction. According to how colors are used in the tile sides, the tiles fall in one of the following five classes, shown in Figure 2.

– **Introduction Tiles** are the tiles that introduce the components, namely function and arguments, see Figure 2, of the computation to be made. These tiles may occur in the top line of a computation grid. No, specific, property is required to the color used in the tile.
– **Terminal Tiles** are the tiles that collect the result of a computation, see Figure 2. These tiles may occur in the bottom line of a computation grid. No, specific, property is required to the color used in the tile.
– **Application Fold-tiles** deal with the reduction of applicative terms that do not require any reduction on their subterms. Color T_1 is used for the function, color T_2 for the argument, whilst colors T_3 or T_4 for the reduced term: It obeys the properties that are indicated at the bottom of the tile in Figure 2.
– **Application Unfold-tiles** deal with the reduction of applicative terms that require some subterm reduction. Color T_1 is used for the function and color T_2, if any, for the argument, exactly as in the fold-tile s, but the reduced term is an application T_3 T_4. This tile structure allows to use two distinct tiles, one for reducing the color T_3 and one for reducing the color T_4, separately. Constraints on the colors are indicated at the bottom of the tile in Figure 2.
– **Connection Tiles** they furnish tiles that are suitable to connect different parts of the computation grids and in some cases they may involve simple term reductions. Constraints on the colors are indicated at the bottom of the tile in Figure 2.

4.2 Soundness of SKI-Tiles

Apart from the introduction and the terminal tiles, all other tiles of SKI-Tiles, are combinatorial term reductions of $\rightarrow^*$. The Wang-arrangement operation corresponds to the (reflection and) transitivity of $\rightarrow^*$. Hence, computation grids contain only sound reductions on the combinatorial terms that are involved in the tiles, and in particular from the terms of the introduction tiles up to the term of the terminal tile of the grid.



Legenda. T_1, T_2, T_3, T_4 are colors for combinatorial terms; →* is the reflexive, transitive closure of the combinatorial reduction.; The colors must obey the property, if any, that is put below the tile.

Fig. 2. The classes of tiles of SKI-Tiles for the Combinatorial Terms

4.3 The Computation Grids of S, K and I in SKI-Tiles

The combinators are completely defined in the Wang Tile formalism by the computation grids in Figure 3, for I and K, and in Figure 4, for S. The grid for I consists of only one fold-tile that switches the input on the output. The grid for K consists of 4 tiles: The tile on the left top corner is a fold-tile that collects the first argument and has "a" as output. The tiles on the right top and the left bottom corners are connection tiles. They are used for connecting the fold-tile on the bottom right corner of the grid. The latter tile contains, as output, the output of the grid. Actually, for S we give two grids of 9 tiles: Both are correct. The two grids differ for the tile on the right bottom corner. Both contains the same fold-tile on left top corner, and the same 7 connection tiles. The other tile is a fold-tile in the left grid, whilst it is an unfold-tile in the right one. The choice of the right grid may depend on the input terms, if a and b do not require any reduction then the left grid may be the best grid to be used. The grids in Figure

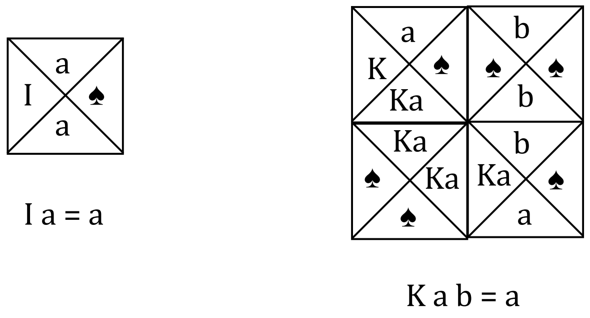


Fig. 3. The computations of K and I in the SKI-Tiles formalism

3, and in Figure 4 are defined for being used as grid components, hence do not contain any introduction or terminal tile.

Theorem 1. *The SKI calculus can be expressed in Wang Tiles*

Proof. The proof is easily obtained by induction on the pure combinatorial terms (i.e. $\Sigma - (\Pi + X)$) and by using suitable grid compositions. The extension to the entire Σ comes immediately since each symbol in $\Pi + X$ is an uninterpreted symbol.

The Theorem above is not surprising since Wang's result [12], but the theorem furnishes a constructive proof and a concrete way to do it. The next section shows how the approach effectively applies in a computation.

5 Applications and Examples

The section shows how the approach effectively applies in a computation. Consider the function $Proj_2^4$ that selects the second argument, from a sequence of four arguments. We write a program that, given four arbitrary terms, c_1, c_2, c_3, c_4, as inputs, computes c_2 as output. In combinatorial programming, the program can be obtained two different ways, according to a use of combinatory programming as an intermediate level or as a higher level programming language. We consider both view and for each of them we show the corresponding computation grid in the tile formalism of the previous section.

5.1 Combinatorial programming at an Intermediate Level

This way of programming is widely influenced by the use of combinators in the implementation of functional languages [19]. In order to obtain a combinatorial term for $Proj_2^4$, we start giving a formulation of $Proj_2^4$ in a functional language. In this case, we can express it by the lambda term:

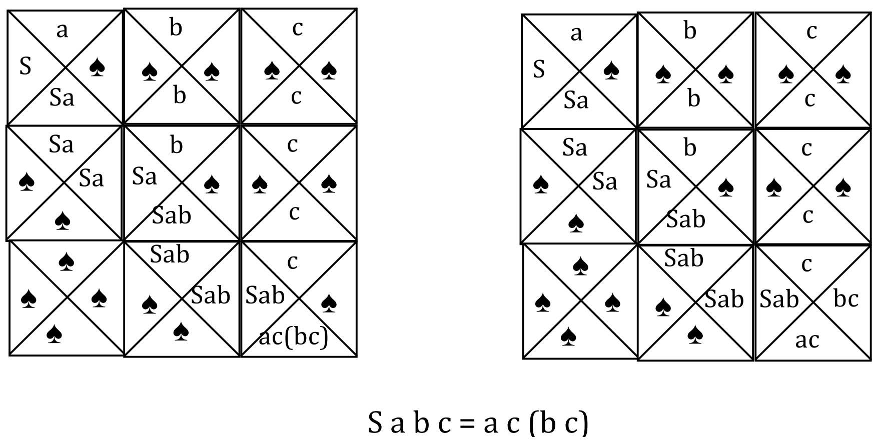


Fig. 4. Two different computations of S in the SKI-Tiles formalism

$\lambda\ x_1.\lambda\ x_2.\lambda\ x_3.\lambda\ x_4.\ x_2.$

Then, by using the technique for removing bound variables from lambda terms[5], we obtain the combinatorial term:

$K(S(KK)(S(KK)I)).$

Eventually, we have the combinatorial term T and its computation grid, in Figure5.

5.2 Combinatorial programming at an High Level

This way of programming uses the possibility of introducing new combinators and super combinators [16] in order to obtain a more expressive and neat solution to a possibly, more general problem than the given one. In this case, the problem may be solved by using a family, $Proj = \{f_n \colon D^n \rightarrow D\}$, of curried functions, each function being indexed by the arity. We can express each function of $Proj$ by the following combinatorial term: $T_p = K^{i-1}(WIK^{n-i})$, where n is the arity of f_n, $0 < i \leq n$ is the position of the argument to be selected, I is the corresponding combinator of SKI calculus. Finally, $K^m g = K(K^{m-1}g)$ is a variant of combinator K (for $m > 1$), whilst W is an additional combinator that obeys the following application law: $Wabc = b(ac)$. Then, the combinatorial program is now expressed by $T = (((K(WIK^2)c_1)c_2)c_3)c_4$, and its computation grid can be obtained by using the same methodology of Section 5.1.

[5] it roughly corresponds [15, 19] to the computation of $[x_1]([x_2]([x_3]([x_4]x_2)))$

6 Self-assembly Computations with SKI-Tiles

This section discusses the formalism of SKI-Tiles in the context of the Self-Assembly programming and extends the formalism with the notions of program and of computation of the Self-Assembly programming paradigm.

6.1 Wang Tiles vs. Self-Assembly

Wang Tiles and Self-Assembly share the same fundamental operation for connecting the tiles: Wang-arrangement. Nevertheless, there is a subtle but relevant

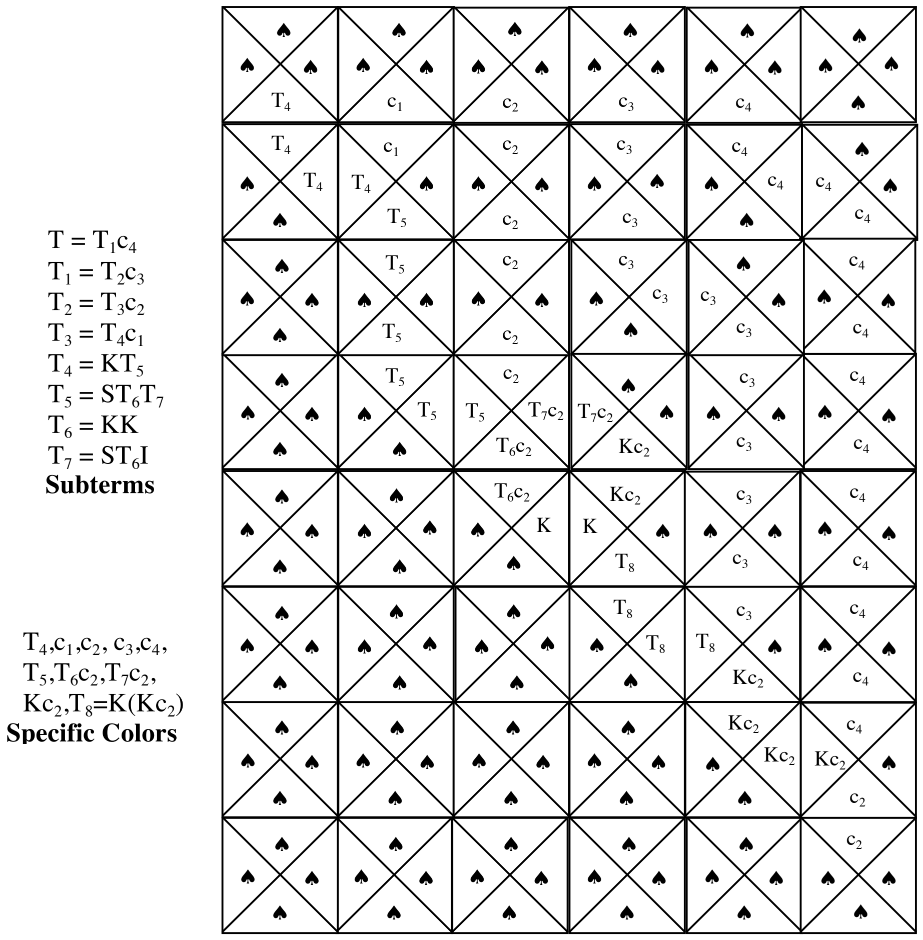


Fig. 5. The computation grid of $T = (((K(S(KK)(S(KK)I))c_1)c_2)c_3)c_4$

difference. The Wang formalism neither has a notion of program nor of computation: The aim is the construction of some computation grid that must be assembled with the tiles of a given tile set. Differently, Self-assembly is a programming paradigm with a notion of program, semantics and computation, that consider all the grids that can be assembled by applying Wang-arrangement to the tiles of the program.

6.2 The SKI-Tiles language for Self-Assembly programming

The section formalizes the notions of program and of computation in order to make SKI-Tiles a language for Self-Assembly programming. Then, it introduces a (combinatorial) formulation of conditional, booleans and numbers for the use of programs, for arithmetic programming, in SKI-Tiles.

Chemical Context Let $H \equiv \{T, g, \tau\}$ be triple defining the physics of molecular self-assembly [5] of the programs. We assume that for all programs, the set of color T, the binding strength function g and the temperature parameter τ are chosen in a way that Wang-arrangement can apply always and only when the tiles abut on sides that are colored by a same color.

Programs. A program is a finite sequence of quadruples of the form (T_1, T_2, T_3, T_4). The use of quadruples introduces a convenient, linear notation for tiles [4], in particular the quadruple (T_1, T_2, T_3, T_4) corresponds to the tiles in Figure 1 provided that T_1, T_2, T_3, T_4 are colors of the SKI-Tiles formalism.

Semantics. Let P be a program. The semantics of P is the set of all *sound* computation grids that can be obtained from P by τ-stable derivation.

Seed and τ-stable Derivation. Let P be a program. Let s be the seed tile of A_0, i.e. the only tile of the grid A_0. Then, $A_0 \rightarrow_P ... \rightarrow_P A_n$ is a computation. Moreover, $\rightarrow_P$ is the τ-stable Derivation (of P in H) and is such that $A \rightarrow_P B$ if and only if B is obtained from A by Wang-arrangement, with a (copy of a) tile of P, which satisfies the chemical context H.

Sound Computation Grid. Unfortunately, the Wang-arrangement does not always produce meaningful computation grids when unfold tiles are admitted. Hence, a computation grid is said sound if and only if the property hold:

- The topmost row contains only introduction tiles and only one of them, the seed, has color $T_3 \neq \spadesuit$, *and*
- The bottom row, if any, contains only terminal tiles and only one of them has color $T_2 \neq \spadesuit$, *and*
- The leftmost column, if any, contains only tiles with a $\spadesuit$ as east side, and
- The rightmost column, if any, contains only tiles with a $\spadesuit$ as west side, and
- No unfold-tile occurs in the grid, *or*
- The unfold-tiles satisfy the sub-grid property.

Definition 3 (Quasi-grids.). *A quasi-grid is a $n \times m$ grid of tiles with $n, m > 1$ and such that: The tiles of the first column, exception for the top tile, have a $\spadesuit$ as west side; The tiles of the first raw, exception for the leftmost one, have a $\spadesuit$ as north side; The tiles of the last column, exception for the bottom tile, have a $\spadesuit$ as east side; Finally, the tiles of last raw, exception for the rightmost one, have a $\spadesuit$ as south side.*

Definition 4 (Sub-grid Property.). *Let G be a computation grid and A be an unfold-tile of G. Then A satisfies the sub-grid property if A is the left top corner of a quasi-grid of G.*

It is worth noting, that the unfold-tiles involve the reduction of combinatorial terms of the form $a\ b$ with the aim of reducing, firstly, a to some a' and b to some b', separately, and then, of reducing $a'\ b'$. Hence, this leads to a sub-computation that behaves like a quasi-grid. As an example, the tile $A \equiv (T_5, c_2, T_6c_2, T_7c_2)$ (4th tile from the top, of the 3rd column, from the left) of computation grid in Figure 5, is an unfold-tile which satisfies the sub-grid property: In particular, the tile is the left top corner of a quasi-grid of 4 tiles. Moreover, even if the tile $B \equiv (T_7c_2, c_3, c_2, \spadesuit)$ was in the program, the sub-grid property would forbid to put it on the east side of A, i.e. the replacing of $(T_7c_2, \spadesuit, Kc_2, \spadesuit)$ with B.

Booleans, Conditional, Numbers in Functional Programming. We list some usefull functional structures for arithmetic calculus, including Barendregt numbers [17] and use them in writing arithmetic programs in functional programming[6]:

- $True \equiv \lambda x.\lambda y.\ x$
- $False \equiv \lambda x.\lambda y.\ y$
- Conditional is implicitly expressed by $True$ and $False$
- $Pair \equiv \lambda x.\lambda y.\lambda z.\ z\ x\ y$
- The number 0 is $[0] = Pair\ True\ (Pred\ [0])$ [7]
- The successor of n is $[n+1] = Pair\ False\ [n]$, for $n > 0$
- Program for Test on 0: $Zero = \lambda x.\ x\ True$
- Program for Predecessor: $Pred = \lambda x.\ x\ False$
- Program for Addition: $Add = \lambda x.\lambda y.\ (Zero\ x)\ y\ (Pair\ False\ (Add\ (Pred\ x)\ y))$
- Program for Product: $Prod = \lambda x.\lambda y.\ (Zero\ x)\ x\ (Add\ (Prod\ (Pred\ x)\ y)\ y)$
- Program for Factorial: $Fact = \lambda x.\ (Zero\ x)\ [1]\ (Prod\ x\ (Fact\ (Pred\ x)))$

Additional Combinators for SKI-Tiles This section extends the set of combinators, to include some combinators, C, B, P, that are of general use in combinatorial programming [15], and some other that are convenient in expressing, in SKI-Tiles, the programs listed above.

[6] We use λ-notation to express the terms: In particular application is term juxtaposition, is left associative, and has precedence on abstraction. Finally, recursive definitions use equations of the form $x = E$, where E is an abstraction and x is a functional variable that cannot occur bound in E

[7] In the original formulation [17], [0] is $Pair\ True\ False$. Here, we extend the domain of the numbers with the *undefined* value, $Pred[0]$.

- Left application combinator is B: $B\ a\ b\ c == a\ c\ b$
- Right application combinator is C: $C\ a\ b\ c == a\ (b\ c)$
- Combinator for Pair is: $P\ a\ b\ c == c\ a\ b$
- Combinator for True is: $T_b\ a\ b == a$
- Combinator for False is: $F_b\ a\ b == b$
- Combinator for Pred is: $P_r\ a == a\ F_b$
- Combinator for test on 0 is: $Z\ a == a\ T_b$
- Combinatorial term for 0 is: $[0] = P\ T_b\ (P_r[0])$
- Combinatorial term for $n+1$ (with $n > 0$) is: $[n+1] = P\ F_b\ [n]$
- Combinatorial Program for Addition:
  $+ = S(CS(B(CC(CZI))I))(C(C(PF_b))(B(CC(C+(CP_rI)))I))$
- Combinatorial Program for Product:
  $\star = S(BC(S(CZI)I))(B(CS(C(C+)(CC(C\star(CP_rI)))))I)$
- Combinatorial Program for Factorial: $F_t = S(B(CZI)[1])(S(C\star I)(CF_t(CP_rI)))$

Let $\mathcal{N}$ be the the minimal set such that $\mathcal{N} = \{[0], PF_b[n] \mid [n] \in \mathcal{N}\}$. Then, $\mathcal{N}$ is the set of (the combinatorial terms for) numbers, whilst $\mathcal{B} \equiv \{T_b, F_b\}$ is the set of terms for booleans.

Self-Assembly Programs in SKI-Tiles Programs in SKI-Tile, for the predecessor, the addition, and the factorial, have the listing in Figure 6: The listing contains only the application tiles. Each program must be completed adding (as by default) the suitable, connection tiles, introduction tiles, and terminal tiles. About the connection tiles, each program includes connection tiles of whatever kind but that involve only one of the program colors (the program colors are all the colors, but ♠, that occur in the program). For instance, the connection tile $(+(Pr\ [2])m, ♠, +(Pr\ [2])m, ♠)$ is included, but $(+(Pr\ [2])m, ♠, +[1]m, ♠)$ is not, in the program for +. About the terminal tiles, these programs compute numbers, hence numbers are the only colors that can be contained in a terminal tile to be included in the programs. Finally, the introduction tiles must contain only colors for numbers and for the name of the program.
In SKI-Tiles, the colors are the combinatorial terms that occur in the program tiles. But the terms occurring in the tiles of the programs in Figure6are not always combinatorial terms because of the the symbols n, m, b. Symbols n ad m are variables ranging on a finite subset of $\mathcal{N}$, whilst b is ranging over $\mathcal{N}$, and the tiles of the programs are in fact, tile schemata.
Finally, note that the program P_r has no computation grid for computing $P_r[0]$.

7 Conclusions

We have investigated three computation formalisms, Wang Tiles, Schonfinkel Combinators and Self-Assembly Programming, in order to define a high level programming language for Self-assembly and DNA computing. We have defined the formalism SKI-Tiles: It states the structures and the properties that the Wang tiles must have in order to express combinatorial terms and the computation of combinatorial programs in the Wang Tiles formalism. We have discussed the soundness of SKI-Tiles. We have used the formalism SKI-Tiles as the kernel of a language for Self-Assembly programming. In order to do it we have revised the notion of computation and introduced the sound computation grid. We called this language the SKI-Tiles language. We have shown programs for Self-Assembly programming that are written in the SKI-Tiles language. These programs compute a partial function for predecessor on naturals, and functions for addition and factorial.

(P_r, [0], P_r, [0])

(P_r, PF_b n, n, ♠)

P_r: A Program for Predecessor

(F_t, n, (Z n)[1](∗n(F_t(P_r n))), ♠)

(F_t, n, (Z n)[1], ∗n(F_t(P_r n)))

(Z n [1], ♠, Z n, [1])

(Z n, ♠, Z, n)

(Z, n, b, ♠)

(b, [1], b [1], ♠)

(T_b [1], ∗n(F_t(P_r n)), [1], ♠)

(F_b [1], m, m, ♠)

(∗n(F_t(P_r n)), ♠, ∗n, F_t(P_r n))

(P_r n, ♠, P_r, n)

F_t : A Program for factorial

(+, n, T, ♠)

(T, m, Z n m, T_1)

(Z n m, ♠, Z n, m)

(Z n, ♠, Z, n)

(Z, n, b, ♠)

(b, m, b m, ♠)

(T_b m, T_1, ♠)

(F_b m, T_1, PF_b, +(P_r n)m)

(+(P_r n)m, ♠, +(P_r n), m)

(+(P_r n), ♠, +, P_r n)

(P_r n, ♠, P_r, n)

legenda:
T≡S(C(Z n)I)(C(PF_b)(C(+(P_r n))I))
T_1≡ PF_b(+(P_r n)m)

+: A Program for addition

Legenda: The Tiles are schemata where n, m are ranging on a finite subset of N and b is ranging on B. Programs specify only the application tiles (The other tiles may be added, by default).

Fig. 6. Self-Assembly Programs for Predecessor, Addition and Factorial in SKI-Tile

References

1. Doty, D.: Theory of Algorithmic Self-Assembly. Comm. ACM **55**(12) (2012)
2. Winfree, E.: On the Ccomputational Power of DNA Annealing and Ligation. In: 2^{th} DIMACS Meeting on DNA Based Computers. (June 1996)
3. Winfree, E., Liu, F., Wenzler, L.A., Seeman, N.C.: Design and Self-Assembly of Two-Dimensional DNA Crystals. Nature **394** (1998) 539–544
4. Adleman, L.: Towards a mathematical theory of self-assembly. Technical report 00-722, Department of Computer Science, University of Southern California (2000)
5. Rothemund, P., Winfree, E.: The Program Size Complexity of Self-Assembled Squares - extended abstract. In: ACM Symposium on Theory of Computing. (2000) 459468
6. Rothemund, P.: Using lateral capillary forces to compute by self-assembly. PNAS **97**(3) (2000) 984–989
7. LaBean, T., Yan, H., Kopatsch, J., Liu, F., Winfree, E., Reif, J., Seeman, N.: The construction, analysis, ligation and self-assembly of dna triple crossover complexes. J. Am. Chem. Soc. **122** (2000) 1848–1860
8. Mao, C., LaBean, T., Reif, J., Seeman, N.: Logical computation using algorithmic self-assembly of DNA triple-crossover molecules. Nature **407** (2000) 493–496
9. Jonoska, N., Liao, S., Seeman, N.: Transducers with programmable input by dna self-assembly. In: Molecular Computing. LNCS 2950 (2004) 219240
10. Doty, D., Patitz, M.: A domain-specific language for programming in the tile assembly model. In: Proceedings of DNA. (2009) 2534
11. Winfree, E.: Simulations of Computing by Self-Assembly. In: 4^{th} DIMACS Meeting on DNA Based Computer. (June 1998)
12. Robinson, R.M.: The Undecidability and Nonperiodicity for Tilings of the Plane. Inventiones math. **12** (1972) 177–209
13. Schonfinkel, M.: On the Building Blocks of Mathematical Logic. in From Frege to Gdel - A Source Book in Mathematical Logic 1879-1931Harvard University Press, 1967 (1924)
14. H.B.Curry, R.Feys: Combinatory Logic. North-Holland Publishing Company, Amsterdam (1956)
15. Turner, D.: Another algorithm for bracket abstraction. The Journal of Symbolic logic **44**(2) (1979)
16. Hughes, J.: Graph reductions with super-combinators. Technical monograph prg-28, Oxford University Computing Laboratory, Programming Research Group (1982)
17. Barendregt, H.P.: Functional Programming and Lambda Calculus. in Handbook of Theoretical Computer Science: Formal Models and Semantics, Elsevier-The MIT Press (1990)
18. Huet, G.: Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems: Abstract Properties and Applications to Term Rewriting Systems. J. ACM **27**(4) (1980) 797–821
19. Jones, S.L.P.: The Implementation of Functional Programming Languages. International Series in Computer Science, Prentice-Hall (1987)