A Geometry of Interaction Machine for Gödel's System T

Ian Mackie

Abstract. Gödel's System T is the simply typed lambda calculus extended with numbers and an iterator. The higher-order nature of the language gives it enormous expressive power—the language can represent all the primitive recursive functions and beyond, for instance Ackermann's function. In this paper we use System T as a minimalistic functional language. We give an interpretation using a data-flow model that incorporates ideas from the geometry of interaction and game semantics. The contribution is a reversible model of higher-order computation which can also serve as a novel compilation technique.

1 Introduction

We present a data-flow model of functional computation, where a single token (the run-time system) travels around a fixed network (the program). Computation begins with an empty token at the root of the network, and it ends when the token returns back to the root with the result. The token is deterministic (at any choice point in the network the token has enough information to proceed in a unique way) and reversible (the token can turn back on itself, to re-trace its steps exactly to undo the computation done).

Gödel's System T (see e.g., [6]) is an applied typed λ-calculus. It is a functional programming language supporting higher-order functions, pairs and projections, numbers and an iterator. It can express all the primitive recursive functions, and up to the so-called ϵ_0 functions. Ackermann's function is included in this set, so its expressive power is sufficiently large to make it interesting. We present a simpler version of System T, called Linear System T [2] with the same expressive power but with a syntax more suited to our needs.

The data-flow interpretation uses ideas developed for linear logic [4]. Specifically, the geometry of interaction (GOI), which models the dynamics of the logic using paths in networks. The GOI machine [9] was a concrete realisation of this idea, originally given for PCF, where an ad-hoc solution was given for base types. To make the data-flow idea work for System T, we need to provide a general reversible interpretation of base types (numbers), which are the data constructors, and also iterators. The need to provide a solution to these points distinguishes the work from others that are related to this, for example [1] gives a general theory of reversible computation through reversible combinators but does not deal with base types in this way. Our approach is also quite different from other reversible functional (which are often first-order), or higher-order, languages, for instance the reversible SECD machine [8]. To summarise, the main contributions are:

- A (reversible) GOI-style model of computation for Gödel's System T.
- An implementation of the model that is a direct compilation into current hardware (essentially directly to assembly language).
- An interesting side effect of this work is an implementation technique that uses an exceptionally small (in terms of space) run-time system in some cases, thus can open up applications to embedded systems, for instance.

Overview. In the next section we give some background material on System T and the geometry of interaction. Section 3 gives the definitions of the token and network structure needed to model System T and a compilation into these networks. In Section 4 we briefly look at some properties, and in Section 5 we discuss some implementation aspects. Finally, we conclude in Section 6.

2 Background

We use a specific version of System T, which is equivalent to the standard presentation (see e.g., [6]), but has a linear syntax. We assume familiarity with the λ-calculus, and refer the reader to [3, 7] for standard notations and concepts. In [2] it was shown that System T can be presented using the linear λ-calculus without losing any computational power. This linear System T was called System L. Essentially, that work illustrates that it is possible to duplicate and erase in System T either using the λ-calculus or the iterator. We simplify the presentation with a variant of System L, that includes numbers as primitives, and gives a simple reversible model. The set of terms is given by the following grammar:

$$t, u ::= x \mid \lambda x.t \mid tu \mid \langle t, u\rangle \mid \mathbf{let}\ \langle x, y\rangle = t\ \mathbf{in}\ u \mid n \mid \mathsf{S}t \mid \mathsf{iter}\ t\ u\ v$$

where n ranges over natural numbers, and x ranges over a finite set of variables. The **let** construct is a way of splitting the pair so that we have access to both components. Numbers are included, so that we can write n for $\mathsf{S}^n 0$. The typing rules (Figure 1) show the valid terms; note that the type system captures the linearity constraints, for example x must occur in t in the abstraction rule. We

$$\frac{}{x : A \vdash x : A}\ (\mathsf{Axiom})$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}\ (\to \mathsf{Intro}) \qquad \frac{\Gamma \vdash t : A \multimap B \qquad \Delta \vdash u : A}{\Gamma, \Delta \vdash tu : B}\ (\to \mathsf{Elim})$$

$$\frac{\Gamma \vdash t : A \qquad \Delta \vdash u : B}{\Gamma, \Delta \vdash \langle t, u\rangle : A \otimes B}\ (\otimes\mathsf{Intro}) \qquad \frac{\Gamma \vdash t : A \otimes B \qquad x : A, y : B, \Delta \vdash u : C}{\Gamma, \Delta \vdash \mathbf{let}\ \langle x, y\rangle = t\ \mathbf{in}\ u : C}\ (\otimes\mathsf{Elim})$$

$$\frac{}{\vdash n : \mathbb{N}}\ (\mathsf{Num}) \qquad \frac{\Gamma \vdash t : \mathbb{N}}{\Gamma \vdash \mathsf{S}\ t : \mathbb{N}}\ (\mathsf{Succ}) \qquad \frac{\Gamma \vdash t : \mathbb{N} \quad \Theta \vdash u : A \multimap A \quad \Delta \vdash v : A}{\Gamma, \Theta, \Delta \vdash \mathsf{iter}\ t\ u\ v : A}\ (\mathsf{Rec})$$

Fig. 1. Type System

can now write simple functions, for example: $\mathsf{add} = \lambda mn.\mathsf{iter}\ m\ (\lambda x.\mathsf{S}x)\ n$, $\mathsf{mult} = \lambda mn.\mathsf{iter}\ m\ (\mathsf{add}\ n)\ 0$, $\mathsf{two} = \lambda fx.\mathsf{iter}\ 2\ f\ x$ and finally Ackermann's function: $\mathsf{ack} = \lambda mn.(\mathsf{iter}\ m\ (\lambda gu.\mathsf{iter}\ (\mathsf{S}u)\ g\ 1)\ (\lambda x.\mathsf{S}x))n$.

For reference, we define the reduction rules, thus giving an operational semantics to the language. This is also useful for a correctness result of the token interpretation that we give later.

Definition 1 (Reduction). *Reduction can take place in any context:*

Name	*Reduction*		*Condition*
Succ	$\mathsf{S}n$	$\to n+1$	
Beta	$(\lambda x.t)v$	$\to t[v/x]$	$\mathsf{fv}(v) = \varnothing$
Let	$\mathbf{let}\ \langle x, y\rangle = \langle t, u\rangle\ \mathbf{in}\ v$	$\to (v[t/x])[u/y]$	$\mathsf{fv}(t) = \mathsf{fv}(u) = \varnothing$
Rec1	$\mathsf{iter}\ (n+1)\ v\ u$	$\to v(\mathsf{iter}\ n\ v\ u)$	$\mathsf{fv}(v) = \varnothing$
Rec2	$\mathsf{iter}\ 0\ v\ u$	$\to u$	$\mathsf{fv}(v) = \varnothing$

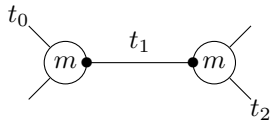
For simplicity, we assume substitution $t[v/x]$ is a meta-operation corresponding to the explicit substitution defined in [2]. We only consider evaluation of programs, that is closed terms of base type. In this way, all programs give a number as a result. The following main properties that we need for this paper (see [2] for the proofs) are:

Proposition 1.
- *Adequacy: If* $\vdash t : \mathbb{N}$*, then* $t \to^* n$*, for some* n*.*
- *Subject Reduction. If* $\Gamma \vdash t : A$ *and* $t \to u$ *then* $\Gamma \vdash u : A$*.*
- *Strong Normalisation: if* $\Gamma \vdash t : A$*, then* t *is strongly normalisable.*

Data-flow and the geometry of interaction. The starting point for the data-flow model comes from the geometry of interaction, which was first set up as a semantics for linear logic [5]. In [9] this is used as an implementation technique—essentially mapping the model to assembly language. This idea was extended to cover the language PCF. Here we use some similar ideas, but our focus is on building a simple data-flow model based on the linear calculus.

We build networks out of nodes and edges. Each node is labelled and has a fixed arity that specifies how many edges can be connected to it. The point of connection between the edge and the node is called a port. The ports of a node are ordered, and we use the label $\bullet$ to give the position of the first port to make clear the orientation. Edges connect two ports together (potentially on the same node) with the constraint that only one edge can be connected to each port. Edges may also connect to one port only in which case we say that it is a free edge. The networks that we build for our programs always have one unique free edge that we call the root of the network.

A token travelling on this network moves from port-to-port along the edges, and is transformed and directed by the nodes: the nodes are routing devices. Consider the following example network that uses two occurrences of a node m of arity three, connected as shown. In this example, t_0 is the starting token, and it is moving to the right. The node m will transform the token to t_1 and direct it towards the second node. The second m node will transform the token to t_2 and direct it along the edge as shown:



Depending on the port of arrival, the token will be inspected by the node, then modified and re-directed. In this example, t_0 is directed along the edge as shown, and it will be modified to remember where it came from: the right-hand edge of the m node. Token t_1 can now change direction and because it has information about where it came from, it can return to t_0 and forget the information it had. Equally, t_1 could continue moving to the right, and this time the second m node will do exactly the same transformation and use the information to direct it to t_2. A simple stack of left (l) and right (r) labels can be stored in the token to achieve this: when arriving from the left or right, push l or r on the stack. When there is a choice, go in the direction indicated by the top of the stack, and pop the stack. In the example above, if t_0 is the empty stack, t_1 will hold r and the token t_2 is again the empty stack. This travel is deterministic, and reversible.

In the next section we define the structure of the token, and define the different nodes that we need to encode System T. Each node is defined by giving the transformation and re-directions of the token.

3 Encoding

Here we define the nodes for the network together with the token interpretation. We also give a compilation of terms into networks, and computation is then a flow over the network with a token that stores the current state.

Definition 2 (Token). *A token (m, i, s, d) is defined by the following components:*

- *m is a stack that contains the elements l and r. This stack is used to navigate the lambda, application and pairs.*
- *i is a stack which keeps information about the iterator. It can contain elements that are either numbers or pairs of numbers.*
- *s is an array of stacks which keeps temporary information about the different iterators. The size of the array is known at compile time, as it depends on the number of iterators.*
- *d is the data stack that contains numbers and an additional element $*$.*

We write $\square$ for the empty stack/array, and use the notation $n : s$ for an element n pushed onto the stack s.

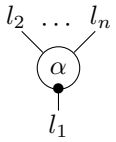
Each node has a flow associated with it, and operations that change the token. We will make this explicit when we define the nodes. The token contains information that is used to redirect it though the nodes of the graph in a deterministic way. Information stored in the token is the only information that is

used for this. The data stack stores numbers, but also $*$ which can be thought of as a place-holder for a number that will be found later. When the data stack has a $*$ at the top of the stack, then this is a question: the token is looking for an answer to this question. When the token has a number at the top, then the token has an answer for the last un-answered question. When the initial question is answered, computation is complete. Some nodes will use the question/answer information to direct the node.

For a program, starting with the token $(\Box, \Box, \Box, * : \Box)$, every run will end with $(\Box, \Box, \Box, n : \Box)$, where n is the result. Any part of the computation can be reversed, and in particular, the whole computation can be reversed: a computation starting with $(\Box, \Box, \Box, n : \Box)$, will end with $(\Box, \Box, \Box, * : \Box)$.

Definition 3 (Network). *A network is a (not necessarily connected) graph built from a set of nodes and edges. A node has a name, a fixed number of ports, and in some cases also a value. The collection of nodes is fixed and defined below.*

We next define the nodes and the transformations for our networks. For each node α of arity n we label the ports $l_1, \ldots, l_n$:

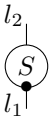


We then describe the transformation by the convention that arriving from l_i and leaving from $\hat{l}_j$ (i.e., the port that is connected to the other end of l_j). For each operation f, we have the reverse operation f^*. We write these two functions as $ft = t'f^*$, which is just an abbreviation of $ft = t'$ and $f^*t' = t$.

Value nodes. A number n is represented as the following network:

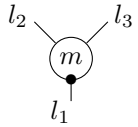
The data-flow for this node is from l to $\hat{l}$. There are two cases: if the token is a question it collects the value n, otherwise, the token is an answer, and it drops off the value n. We define operations n and n^* which are defined as $n(m, i, s, * : d) = (m, i, s, n : d)n^*$ to do these transformations.

Unary functions. The successor node is represented as the net:



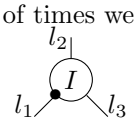
There are two data-flows for this node, and two cases for each depending on whether the token is a question or an answer. We explain two flows, and the other two are the reverses. If the token is a question and arrives at l_2, then we ask another question to find the argument of the function, thus we push $*$ onto the data stack and the token travels along the edge to $\hat{l}_1$. We define an operation s_1 to do this transformation. If the token is an answer and arrives at l_1, then we need to continue to $\hat{l}_2$, and apply the successor function. The operation s_2 does this transformation. These two operations, and inverses, are defined as: $s_1(m, i, s, * : d) = (m, i, s, * : * : d)s_1^*$ and $s_2(m, i, s, n : * : d) = (m, i, n{+}1 : d)s_2^*$.

Multiplexing. We use a multiplexing node for abstraction, application, the pair and the let constructs.



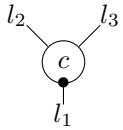
There are four different data-flows for m, and all the operations only alter the m component of the token. Arriving at l_1, we need to decide which way to go. If the top of the stack m stack is l, we apply the l^* transformation and travel along the edge to $\hat{l}_2$. Otherwise, the top of the stack is r, so we apply the r^* transformation and travel along the edge to $\hat{l}_3$. Arriving at l_2 or l_3, the token travels along the edge to $\hat{l}_1$, and we need to remember which side we came from. The operations l and r do this transformation, respectively. These functions are defined as: $l(m, i, s, d) = (l : m, i, s, d)l^*$ and $r(m, i, s, d) = (r : m, i, s, d)r^*$.

Iterator. We next need the nodes that will allow for the encoding of the iterator. The first one will find the number of times we need to iterate.



There are four different data-flows for this node, and the operations alter the d and i components of the token. Arriving at l_1, there are two cases. If the token is an answer, then the token travels along the edge to $\hat{l}_3$ and we apply an operation called i_2. Otherwise, the token is a question, and it travels along the edge to $\hat{l}_2$ and we apply the i_1^* operation. Arriving at l_2, the token travels along the edge to $\hat{l}_1$, and we apply the i_1 operation. Finally, arriving at l_3, the token travels along the edge to $\hat{l}_1$, and we apply the i_2^* operation. These functions are defined as $i_1(m, i, s, d) = (m, i, s, * : d)i_1^*$ and $i_2(m, i, s, n : d) = (m, n : i, s, d)i_2^*$.

Counter. To model the iteration process in a reversible way, we need something similar to the multiplexing node, but having the ability to count.



There are four different data-flows for this node, and the operations only alter the i component of the token. Arriving at l_1, if the top of the i stack is 0, we apply the z^* transformation and travel along the edge to $\hat{l}_2$. Otherwise, the top of the stack is non-zero, so we apply the s^* transformation and travel along the edge to $\hat{l}_3$. Arriving at l_2 or l_3 the token travels along the edge to $\hat{l}_1$, and we apply the operation z or s, respectively. Counters will always come in pairs: we call them c and c', and the prime operations are essentially the same, but operate on the other component of the pair. These functions are defined as:

$$\begin{array}{ll} z(m, c : i, s, d) & = (m, (0, c) : i, s, d)z^* \\ z'(m, c : i, s, d) & = (m, (c, 0) : i, s, d)z'^* \\ s(m, (c_1, c_2) : i, s, d) & = (m, (c_1 + 1, c_2) : i, s, d)s^* \\ s'(m, (c_1, c_2) : i, s, d) & = (m, (c_1, c_2 + 1) : i, s, d)s'^* \end{array}$$

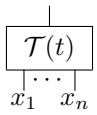
The final node that we need captures the scope of an iterator.

There are two data-flows possible for this kind of node. Arriving at l_1, the token travels towards $\hat{l}_2$, and we apply the operation t^*. Arriving at l_2, the token travels along the edge to $\hat{l}_1$, and we apply the operation t. The collection of functions are defined as $t_j(m, (c_1, c_2) : i, s, d) = (m, i, \mathit{push}(j, s, (c_1, c_2)), d)t_j^*$, where the operation *push* updates the array of stacks at position j $(1 \leq j \leq n)$:

$$\mathit{push}(j, [s_1, \ldots, s_n], (c_1, c_2)) = [s_1, \ldots, (c_1, c_2) : s_j, \ldots, s_n]$$

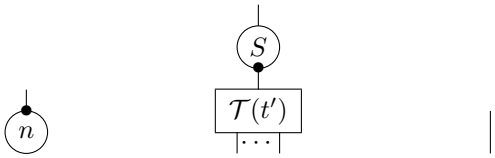
This completes the definition of the nodes that we need for our networks.

Compilation. The compilation $\mathcal{T}(\cdot)$ of terms into networks is given using the nodes introduced above. A term t with $\mathsf{fv}(t) = \{x_1, \ldots, x_n\}$ will be translated as a network $\mathcal{T}(t)$ with the root edge at the top, and n free edges corresponding to the free variables of the term.

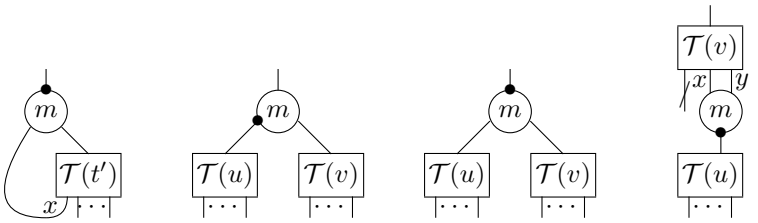


The labelling of free edges is just for convenience, and is not part of the system. We proceed by induction over the structure of the term being translated.

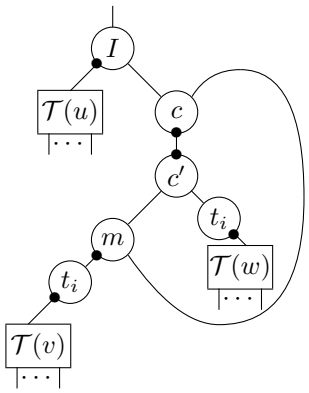
Numbers, Functions and Variables. When t is a number, say n, then $\mathcal{T}(n)$ is given by the first network below. When t is the unary function $\mathsf{S}\ t'$, then $\mathcal{T}(t)$ is given by the middle diagram, where we use node S. Finally, when t is a variable, say x, then $\mathcal{T}(t)$ is translated into a wire, as shown right-most below.



Abstraction, Application, Pairs and Let. If t is an abstraction, say $\lambda x.t'$, then $\mathcal{T}(t)$ is translated into the network on the left below. We connect the (necessarily unique) occurrence of the variable x to the binding λ (m node). We have assumed, without loss of generality that x is the leftmost free variable of the term t'. If t is uv, then $\mathcal{T}(t)$ is given by the second diagram below. If t is $\langle u, v\rangle$, then $\mathcal{T}(t)$ is given by the third network below. Because of the linearity constraints, there are no common free variables in u and v. Finally, if t is **let** $\langle x, y\rangle = u$ **in** v, then $\mathcal{T}(t)$ is given by the network on the right—we have assumed that x and y are the right-most two free variables in v. The other free variables of v are represented as a line struck through.



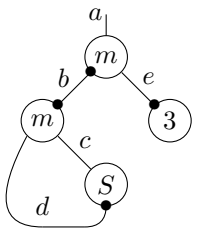
Recursor. If t is iter u v w, then $\mathcal{T}(t)$ is given by the following network.



The compilation of the iterator uses several different nodes. The two counter nodes work together to maintain the iterations. The I node is responsible for navigating the token to the number, then passing that value to the counters. The t_i nodes are used to record the state of the iterators if the token leaves the scope of the iterator, where a new index i is associated to each iterator.

There are a number of ways that a network can be simplified at compile time, but we leave the details for another occasion.

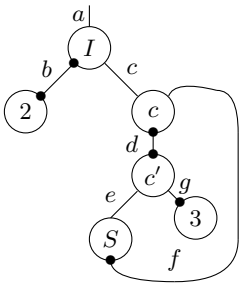
Examples. We show several examples to illustrate how this style of computation works. The first one is the program $(\lambda x.\mathsf{S}x)3$, thus the successor function applied to 3. The network generated through the compilation of this term is given below, where we have labelled the edges for reference. The initial token (which is the same for any program of base type) is $(\square, \square, \square, * : \square)$, and the program counter starts in the network at the root (labelled with a). The token then travels along the sequence b, c, d, etc. We show the edge together with the token at that place.



a $(\square, \square, \square, * : \square)$
b $(l : \square, \square, \square, * : \square)$
c $(\square, \square, \square, * : \square)$
d $(\square, \square, \square, * : * : \square)$
b $(r : \square, \square, \square, * : * : \square)$
e $(\square, \square, \square, * : * : \square)$
e $(\square, \square, \square, 3 : * : \square)$
b $(r : \square, \square, \square, 3 : * : \square)$
d $(\square, \square, \square, 3 : * : \square)$
c $(\square, \square, \square, 4 : \square)$
b $(l : \square, \square, \square, 4 : \square)$
a $(\square, \square, \square, 4 : \square)$

The token arrives back to the root with the answer 4. At any point during the computation we can use the information in the token to turn back and re-trace the computation—deterministically—to the start.

The next example is $\mathsf{iter}\ 2\ (\lambda x.\mathsf{S}\ x)\ 3$, which computes the addition of 2 and 3. (This results in a simplified version of $\mathsf{add}\ 2\ 3$, without the need for abstraction and application; it is the same term when the (linear) β-reductions have been done). The resulting network is shown below. Looking at the structure of the execution trace reveals a symmetry, reflecting that the computation is reversible.



a $(\square, \square, \square, * : \square)$
b $(\square, \square, \square, * : * : \square)$
b $(\square, \square, \square, 2 : * : \square)$
c $(\square, 2 : \square, \square, * : \square)$
d $(\square, (0, 2) : \square, \square, * : \square)$
e $(\square, (0, 1) : \square, \square, * : \square)$
f $(\square, (0, 1) : \square, \square, * : * : \square)$
d $(\square, (1, 1) : \square, \square, * : * : \square)$
e $(\square, (1, 0) : \square, \square, * : * : \square)$
f $(\square, (1, 0) : \square, \square, * : * : * : \square)$
d $(\square, (2, 0) : \square, \square, * : * : * : \square)$
g $(\square, 2 : \square, \square, * : * : * : \square)$
g $(\square, 2 : \square, \square, 3 : * : * : \square)$
d $(\square, (2, 0) : \square, \square, 3 : * : * : \square)$
f $(\square, (1, 0) : \square, \square, 3 : * : * : \square)$
e $(\square, (1, 0) : \square, \square, 4 : * : \square)$
d $(\square, (1, 1) : \square, \square, 4 : * : \square)$
f $(\square, (0, 1) : \square, \square, 4 : * : \square)$
e $(\square, (0, 1) : \square, \square, 5 : \square)$
d $(\square, (0, 2) : \square, \square, 5 : \square)$
c $(\square, 2 : \square, \square, 5 : \square)$
b $(\square, \square, \square, 2 : 5 : \square)$
b $(\square, \square, \square, * : 5 : \square)$
a $(\square, \square, \square, 5 : \square)$

The final example, shown in Figure 2, gives the compilation of Ackermann's function applied to two arguments: *ack* 2 3. We have applied several optimisations to this example which remove some of the nodes that are not needed, and we have also performed a linear β-reduction. We will not attempt a trace of this example, as the number of operations is quite large (see next section). The example is just to show what a more elaborate program looks like in this setting.

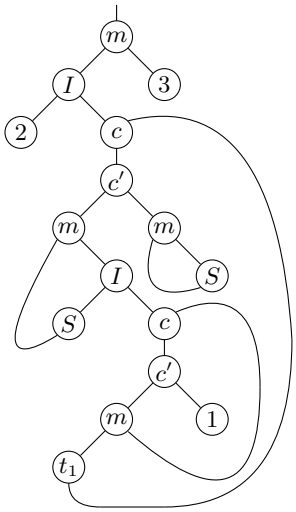


Fig. 2. Ackermann's function

4 Properties

Here we show some important results about this method of computing. Full details will be included in a longer version of this paper.

Lemma 1 (Determinism). *For a closed term $t : \mathbb{N}$, computation is bidirectional and deterministic.*

This result holds because each transformation has this property, and we are able to show that the token will always have enough information to progress. Computation is therefore reversible: at any point we can turn back and undo the computation. For instance, the second example in the last section can be started with the token $(\Box, \Box, \Box, 4 : \Box)$, and the computation will be performed in a deterministic way to give the final result $(\Box, \Box, \Box, * : \Box)$ as required. To show that this notion of computation is correct, then we need a way of relating the token flow with an operational semantics.

Theorem 1 (Correctness). *Let $t : \mathbb{N}$ be a closed term. $t \to^* n$ iff there is a run in the network $\mathcal{T}(t)$: $(\Box, \Box, \Box, * : \Box) \to^* (\Box, \Box, \Box, n : \Box)$.*

Using this result we can give the main result about reduction:

Theorem 2. *For each reduction rule $t \to u$, the initial and final states of the token are the same.*

5 Discussion: implementation

We give a few implementation details for our data-flow model on traditional hardware by giving a compilation for System T directly to assembly language. The data stack d can be separated out as a number stack and a question stack. The question stack can then be implemented using a register, using 0 for question and 1 for answer. To push and pop values on this stack, we can then use simple register shifts. Knowing if the current operation is a question or an answer is then just a bit test. We can also reduce the size of the question stack in some cases. For instance, when we ask for the result of a successor applied to an argument, then we can use the same question for the argument as the result. The other stacks can be mapped onto memory in standard ways. Depending on the number of iterators (known at compile time), we can map some of the structure to registers or blocks of memory.

The compilation procedure is to build a network, then compile this network to instructions. This is done by compiling each node as a block of code. We show the example below for the case of the m nodes, which are the simplest case (we refer the reader to the diagram of the m nodes with the labels to help understand the labels, and we use a register $R0$ to hold the m stack). Edges are then just linking information, instructing how to get to the next block of code.

```
l2 : lsl R0          l3 : lsl R0          l1 : lsr R0
     br l1^               inc R0               be l2^
                          br l1^               br l3^
```

Thus for each node we need to generate instructions that will implement the appropriate collection of functions associated to it. In some cases, this will mean interrogating the question stack, and then doing an operation depending on that value. All these functions however map quite directly to assembly level.

There are many improvements and optimisations that can be made to the compilation. For example, if the number of times a function is being iterated is known, then we do not need to push the value on the stack if we leave through a t node when compiling the base value, as we know that this value will always be the same—a single stack location is all that is needed. This can be identified statically. There are other little improvements that can be made also, but it is not yet clear if any of these have a significant impact on performance.

We have implemented all the ideas in this paper, and it is worth mentioning that the execution times are long, but the run-time memory usage can be surprisingly small. We give a couple of illustrative examples in the table below.

Program	Result	i	d	jumps
two two two two $(\lambda x.\mathsf{S}x)$ 0	65536	16	2	92M
ack 3 2	29	3	15	2.4M
ack 2 24	51	2	26	537M
ack 3 3	61	3	31	4.7M

We show the example program (using the terms given in Section 2) together with the result, the maximum height of the iterator stacks, and the maximum

height of the data stack. Since two integers are needed for each element of the iterator stack, and each data element just one, this means that $2i + d$ is the total number of memory space used. The table also shows the (approximate) number of jumps (units are millions) in the network. Computation can take a long time, but little memory is used.

6 Conclusion

We have given a geometry of interaction style data-flow implementation of a simple language built from the linear λ-calculus extended with a recursor operator. Depending on how we constrain the use of the recursor, this language is rich enough to capture all primitive recursive functions or more generally Gödel's System T. Current work in this area includes developing the ideas to richer languages (in particular to include other data-structures), and developing new compilation techniques. Various program transformations can be applied to change the kind of runs in interesting ways, and computations can be significantly shortened by jumping in the network. The work has been implemented, and benchmark tests show that significant computations can be performed using very little run-time memory, thus there is potential for application in embedded systems in addition to giving a reversible implementation of a seemingly non-reversible language.

References

1. Abramsky, S.: A structural approach to reversible computation. Theoretical Computer Science 347, 441–464 (2005)
2. Alves, S., Fernández, M., Florido, M., Mackie, I.: Gödel's system $\mathcal{T}$ revisited. Theoretical Computer Science 411(11-13), 1484–1500 (2010)
3. Barendregt, H.P.: The Lambda Calculus: Its Syntax and Semantics, Studies in Logic and the Foundations of Mathematics, vol. 103. North-Holland Publishing Company, second, revised edn. (1984)
4. Girard, J.Y.: Linear Logic. Theoretical Computer Science 50(1), 1–102 (1987)
5. Girard, J.Y.: Geometry of interaction 1: Interpretation of System F. In: Ferro, R., Bonotto, C., Valentini, S., Zanardo, A. (eds.) Logic Colloquium 88, Studies in Logic and the Foundations of Mathematics, vol. 127, pp. 221–260. North Holland Publishing Company, Amsterdam (1989)
6. Girard, J.Y., Taylor, P., Lafont, Y.: Proofs and Types. Cambridge University Press, New York, NY, USA (1989)
7. Hankin, C.: An Introduction to Lambda Calculi for Computer Scientists, Texts in Computing, vol. 2. King's College Publications (2004)
8. Kluge, W.: A reversible se(m)cd machine. In: Koopman, P., Clack, C. (eds.) Implementation of Functional Languages, Lecture Notes in Computer Science, vol. 1868, pp. 95–113. Springer Berlin / Heidelberg (2000)
9. Mackie, I.: The geometry of interaction machine. In: Proceedings of the 22nd ACM Symposium on Principles of Programming Languages (POPL'95). pp. 198–208. ACM Press (January 1995)