Local Linear Logic for Locality Consciousness in Multiset Transformation

Hugh McEvoy and Pieter H. Hartel

Department of Computer Systems, University of Amsterdam,
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
{hugh,pieter}@fwi.uva.nl

Abstract. We use Girard's linear logic (LL) to produce a semantics for Gamma, a multiset transformation language. The semantics improves on the existing structured operational semantics (SOS) of the language by highlighting Gamma's inefficiencies, which were hidden by the SOS. We propose a new logic called local linear logic (Local LL), which adds locality-consciousness to the resource-consciousness of linear logic. As a case study, we use this logic to propose a new semantics for Gamma. The new semantics suggests an annotation of Gamma which increases its efficiency without compromising its programming style. We show how the new semantics also gives us a better understanding of parallel Gamma and its implementation, and offers insight into the nature of chemical-reaction based computational models in general.

1 Introduction

Languages based upon the chemical reaction model combine terse expression of parallel programs with terrible efficiency problems. Gamma [9] is such a language. The tendency has been to view the efficiency problem as either a matter for implementors to overcome, or a reason to abandon such languages altogether. Abandoning such languages altogether would be a shame, as they possess some pleasing mathematical properties [22]. They are also showing themselves useful as abstract languages for describing physical and biological phenomena [28]. The persistence of Linda [17, 16] in the programming community also bears witness to the usefulness of the paradigm. We believe that instead of abandoning these languages, we should attempt to reap the benefits of these languages without reaping the whirlwind.

A traditional SOS of these languages doesn't help clarify the issue, because it captures possible execution traces while ignoring the cost of termination detection (linked to the cost of finding data). For example, extant SOS of Gamma [21, 15] use quantifiers to express the presence in the multiset of data with particular properties, which obscures that it is the cost of *finding* these data which cripples the implementations. This is unfortunate, for an SOS is otherwise an ideal vehicle for studying languages without the need to produce an implementation with all its associated clouding of the essential characteristics of the language for which it was written. We show that Girard's Linear Logic [18] can be used to provide a semantics for Gamma which highlights its parallelism, non-determinism and inefficiencies. We propose a locality-conscious variant of LL which enables us to reason about both the cost of reproducing data *and* the cost of relocating data, and show the benefits of this approach for understanding the costs of maintaining an ordered data structure and the partitioning of data on a parallel platform. This logic provides the theoretical underpinning for adding an annotation to Gamma to improve its efficiency without altering the basic, compelling, style of the language and of the programs written therein.

This paper is structured as follows. We first give a brief introduction to the language Gamma and to linear logic. The next two sections describe how we translate Gamma functions into linear logic formulae, and how the execution of these programs in Gamma correspond to steps in linear logic proof trees. Following that, we describe the shortcomings of the approach and suggest a solution, in the form of a refinement both of linear logic and of the Gamma model. We discuss the benefits yielded by these refinements. Finally, we draw our conclusions, survey related work, and discuss possibilities for future research.

2 Gamma

Gamma is a non-deterministic language for the transformation of multisets proposed by Le Métayer and Banâtre [9]. It combines a pleasing declarative programming style with the possibility of massively parallel implementations. Its abstract syntax is shown in Fig. 1. Gamma is designed as a general-purpose language, and has been used to describe a number of programs [10, 27]. The computational model is based upon a chemical reaction: an initial multiset is transformed repeatedly as functions are applied to its elements. The transformations continue, with the products of old reactions being re-used in new reactions, until no more reactions are possible. This occurs either when the multiset contains too few elements to provide enough arguments to any function, or when no permutation of the elements satisfies the boolean conditional of any function. The arguments of a function can be thought of as the reagents and the results as the products of a chemical reaction.

Multiple functions in a single program can be combined using two connectives. The first is written '$\circ$' and is a right-associative composition of two functions. `f1` $\circ$ `f2` performs `f2` as many times as possible and then `f1` as many times as possible. The second connective is written '+' and indicates the parallel interleaving of two functions. For example, `f1 + f2` will repeatedly apply either `f1` or `f2` to the set until neither can be applied. In the case that both functions can be applied, a choice between them is made non-deterministically.

An SOS-style semantics for Gamma, as given in [21], is shown in Fig. 2. A pair (R, A) is a function (with its name elided). R is the reaction condition, which is a boolean conditional corresponding to the 'if ... ' clause in the abstract syntax. A is the action, which is a function returning a multiset of expressions to be added to the multiset. Its syntax corresponds to '[exp [: type]]$^+$' in the abstract syntax. M is the multiset.

$$
\begin{array}{lcl}
\text{system} & = & \text{function } \{ \ [\text{exp}]^* \ \} \\
\\
\text{function} & = & \text{fn } (\ [\text{tup } [:: \text{type}]]^+ \ ) \Rightarrow [\text{exp } [: \text{type}]]^+ \text{ if exp} \\
 & | & \text{function} + \text{function} \\
 & | & \text{function} \circ \text{function} \\
\\
\text{tup} & = & \text{id} \\
 & | & \langle \ \text{tup, tup} \ \rangle
\end{array}
$$

Fig. 1. The essentials of the abstract syntax of Gamma. Expressions ('exp') are built out of constants, variables ('id') and operators. We do not allow function names ('fn') in expressions.

$$((R,A),M) \to ((R,A),(M - \{x_1 \dots x_n\} \cup A(x_1 \dots x_n))) \quad \text{if } \exists x_1 \dots x_n \in M.R(x_1 \dots x_n) \qquad (1)$$

$$((R,A),M) \to M \quad \text{if } \neg\exists x_1 \dots x_n \in M.R(x_1 \dots x_n) \qquad (2)$$

$$\frac{(P_2,M) \to M}{(P_1 \circ P_2, M) \to (P_1, M)} \quad (3) \qquad \frac{(P_2,M) \to (P_2',M')}{(P_1 \circ P_2, M) \to (P_1 \circ P_2', M')} \quad (4)$$

$$\frac{(P_2,M) \to (P_2',M')}{(P_1 + P_2, M) \to (P_1 + P_2', M')} \quad (5) \qquad \frac{(P_1,M) \to (P_1',M')}{(P_1 + P_2, M) \to (P_1' + P_2, M')} \quad (6)$$

$$\frac{(P_1,M) \to M \quad (P_2,M) \to M}{(P_1 + P_2, M) \to M} \quad (7)$$

Fig. 2. An SOS-style semantics for Gamma

The Gamma model is related to the models of UNITY [12], Action Systems [6, 7], Linda and the Chemical Abstract Machine of Berry and Boudol [11]. All four related formalisms use the metaphor of a chemical reaction as their computational model, although only the latter two use multisets (tuple spaces) as their only data types.

The chemical reaction metaphor allows the expression of algorithms which contain a high degree of data parallelism. For example, the Gamma implementations described in [13, 32, 20] all exhibit potential data parallelism up to the limits imposed by the hardware (the nature of the data may restrict this in practice). Furthermore, in Gamma, multiple functions can be applied simultaneously to arbitrary disjoint subsets of the multiset. This latter property might incline one to believe that parallel implementations of Gamma could achieve good scalability. Unfortunately, this has transpired not to be the case. We discuss why this is not the case in due course.

2.1 An Example Gamma Program

An example Gamma program 'addup' is given in Fig. 3. It finds the sum of a multiset of numbers, by repeatedly replacing each pair of numbers with their sum. It continues until there are not enough elements in the set to provide it with its arguments: this single element is the sum of all the elements in the original multiset. We can 'execute' this program using the SOS rules of Fig. 2. A possible execution trace is shown in Fig. 3.

$$\text{addup } (x, y) \Rightarrow x + y \text{ if True } \{\ldots\}$$

$$\begin{array}{cl}
((\text{True}, x+y), \{1,2,3,4\}) & \\ \hline
((\text{True}, x+y), \{3,3,4\}) & (\mathbf{1}) \\ \hline
((\text{True}, x+y), \{6,4\}) & (\mathbf{1}) \\ \hline
((\text{True}, x+y), \{10\}) & (\mathbf{1}) \\ \hline
\{10\} & (\mathbf{2})
\end{array}$$

Fig. 3. The program 'addup' (on the left) and an example execution according to the SOS (on the right). The boldface numbers on the right-hand side indicate which SOS rule was used for each step. Many other execution trees are possible, as the SOS is non-deterministic.

The SOS shows the result of executing the program, without having to implement the language. However, the SOS fails to highlight Gamma's inefficiency, for the presence of quantifiers in the SOS hides the cost of finding data with the required properties for a function to be applied. We show how linear logic can be used to generate a semantics which highlights this information, as well as a number of other interesting properties which were not immediately obvious from the SOS given above. The SOS is also an interleaving semantics rather than a parallel semantics. It may well be possible to address the first of these criticisms by refining the SOS, but the second criticism cannot be so addressed.

3 Linear Logic

Linear logic was proposed as an alternative to conventional intuitionistic and classical logics, wherein the structural rules of weakening and contraction were removed and replaced with more restricted variants. The result is a logic of resources: the effect of using a formula in a derivation is that the formula is destroyed. If it is desired that a particular formula be copied or discarded, it must be labelled with an 'exponential' symbol to indicate that this is the case. Otherwise, no copying or discarding of formulae is allowed. If the mantra for classical logic is ‘truth is free’, then that for linear logic is ‘computation is not free’. The current work is not intended to be an introduction to linear logic. Excellent introductions are to be found in many of the papers cited above, and in [31, 4].

The contribution of linear logic, from the point of view of computer science, is that it allows reasoning about the number of storage operations required and formulae which must be discarded in order that a particular result be generated. In functional languages in particular, this allows reasoning about the costs of particular evaluation strategies and has inspired work in type systems [24, 8, 33, 1].

We use a somewhat nonstandard set of rules for our presentation of classical linear logic. We use a combination of Girard’s original notation for the multiplicative fragment of the logic, Yetter’s κ rule [35], from his Cyclic LL, and variants of the polynomially-bounded exponentials from Girard, Scedrov and Scott’s Bounded LL [19]. We also extend the latter notion by allowing a special case of an exponential bounded by ω. This is because we wish to be able to produce infinitely deep proofs for non-terminating computations (either ones which cycle endlessly or those which continue to generate new results forever). We require all our sequences of wff (an abbreviation of ‘Well-Formed Formulae’) to be finite, so we have no use for Gentzen’s Ω-rule.

We use completely non-standard exchange laws. The reason for this is that we wish to allow our one-dimensional sequence of wff to mimic the multiset, which contains elements of different types. From the point of view of efficiency, it is beneficial to partition the multiset into a number of subsets; one for each type. We therefore allow elements of different types to ‘pass over’ each other as if all elements of different types were invisible to each other. Although this was already possible with the original LL exchange law, we wish to distinguish this use of exchange from other uses. The reasons for this will become clear when we discuss the inefficiencies of the Gamma model. However, it is easy to see that the union of these exchange laws apply to precisely the same set of formulae as that to which the normal exchange rule applied.

The sequent calculus for the multiplicative fragment of our version of Classical Linear Logic is shown in Fig. 5. The reader is referred to Fig. 4 for the readings of the symbols. Some useful equivalences in LL are given in Fig. 6 (following [1]). The τ in our exchange laws stands for ‘type-different’, indicating that it applies to elements of different types. The ς stands for ‘stirring’, indicating that it applies to elements of the same type and corresponds to the magic-stirring mechanism (explained in section 6). The κ exchange laws apply only to formulae prefixed by a κ, and allow us to make further distinctions in our uses of exchange laws.

4 Casting Gamma into LL

Gamma possesses a notion of resource-consciousness. That is, data are destroyed when a function is successfully applied to them. To have multiple copies of a

$\otimes$	= multiplicative conjunction	$\parr$	= multiplicative disjunction
$\multimap$	= multiplicative linear implication	$\mathbf{1}$	= multiplicative truth
$\perp$	= multiplicative falsehood	$!, ?$	= exponentials
$\forall, \exists$	= quantifiers	$a, b, \ldots$	= terms
$A, B, \ldots$	= types	$x, y, \ldots$	= variables
$\emptyset$	= empty set of wff	**A**	= Axiom rule
E	= Exchange rule	**C**	= Contraction rule
CD	= Contraction-Dereliction rule	**W**	= Weakening rule

Fig. 4. The readings of the symbols of the multiplicative fragment of LL. The notation follows Girard. The last five entries are names of rules. The rules themselves are shown in Fig. 5.

datum, it must be explicitly copied. In this sense, Gamma is linear. Furthermore, Gamma possesses several properties which make it simple to translate into LL, and which we use to minimise the size of the formulae generated:

- It is first order.
- It is eager.
- It has no recursive data types: only tuples may be constructed.

These properties serve to make the translation extremely terse: so terse that the statements of linear logic produced in the translation are often no longer than the Gamma functions from which they were translated. This enables us to reason about real Gamma programs, even by hand, and gives a better appreciation of the simplicity and compelling nature of the chemical-reaction model, and of some of its disadvantages. However, a possible difficulty looms:

- Gamma is non-deterministic.

Fortunately our logic-based semantics is inherently non-deterministic, and allows us to capture the non-determinism in the language without difficulty. Indeed, it would be more difficult to capture determinism than to capture non-determinism: we would have to ensure that only one logical rule be applicable to the sequence of wff at any one time.

4.1 The Translation of Gamma Programs into LL

We can translate Gamma programs into formulae of linear logic, following the rules shown in Fig. 7. Note that we choose to translate functions of the form 'if A then B' into the unconventional $A^{\perp} \parr B$ instead of the usual $A \multimap B$. These two forms are equivalent, as shown in Fig. 6. We choose to adopt the unconventional rendering for reasons which will become apparent later on.

$$\frac{}{a{:}A \vdash a{:}A}\ \mathbf{A} \qquad \frac{\Gamma_1, \Gamma_2 \vdash \Delta}{\Gamma_1, \mathbf{1}, \Gamma_2 \vdash \Delta}\ \mathbf{1L} \qquad \frac{}{\emptyset \vdash \mathbf{1}}\ \mathbf{1R}$$

$$\frac{\Gamma_1, t{:}A, b[t/x]{:}B, \Gamma_2 \vdash \Delta}{\Gamma_1, t{:}A, \forall(x{:}A).b{:}B, \Gamma_2 \vdash \Delta}\ \forall\mathbf{L} \qquad \frac{\Gamma_1, \Gamma_2 \vdash a{:}A, \Delta}{\Gamma_1, a^{\perp}{:}A^{\perp}, \Gamma_2 \vdash \Delta}\ {}^{\perp}\mathbf{L} \qquad \frac{\Gamma_1, a{:}A, \Gamma_2 \vdash \Delta}{\Gamma_1, \Gamma_2 \vdash a^{\perp}{:}A^{\perp}, \Delta}\ {}^{\perp}\mathbf{R}$$

$$\frac{\Gamma_1, !_n a{:}A, !_m a{:}A, \Gamma_2 \vdash \Delta}{\Gamma_1, !_{n+m} a{:}A, \Gamma_2 \vdash \Delta}\ \mathbf{C} \text{ if } n > 1 \text{ and } m > 1 \qquad \frac{\Gamma_1, a{:}A, !_n a{:}A, \Gamma_2 \vdash \Delta}{\Gamma_1, !_{1+n<\omega} a{:}A, \Gamma_2 \vdash \Delta}\ \mathbf{CD} \text{ if } n > 0 \qquad \frac{\Gamma_1, \Gamma_2 \vdash \Delta}{\Gamma_1, !_n a{:}A, \Gamma_2 \vdash \Delta}\ \mathbf{W}$$

$$\frac{\Gamma_1, !_z a{:}A, !_\omega a{:}A, \Gamma_2 \vdash \Delta}{\Gamma_1, !_\omega a{:}A, \Gamma_2 \vdash \Delta}\ \mathbf{C}_\omega \text{ if } z > 1 \qquad \frac{\Gamma_1, a{:}A, !_\omega a{:}A, \Gamma_2 \vdash \Delta}{\Gamma_1, !_\omega a{:}A, \Gamma_2 \vdash \Delta}\ \mathbf{CD}_\omega \qquad \frac{\Gamma_1, \Gamma_2 \vdash \Delta}{\Gamma_1, !_\omega a{:}A, \Gamma_2 \vdash \Delta}\ \mathbf{W}_\omega$$

$$\frac{\Gamma_1, a{:}A, b{:}B, \Gamma_2 \vdash \Delta}{\Gamma_1, a \otimes b{:}A \otimes B, \Gamma_2 \vdash \Delta}\ \otimes\mathbf{L} \qquad \frac{\Gamma_1 \vdash \Delta_1, a{:}A, \Delta_2 \quad \Gamma_2 \vdash \Delta_3, b{:}B, \Delta_4}{\Gamma_1, \Gamma_2 \vdash \Delta_3, \Delta_1, a \otimes b{:}A \otimes B, \Delta_4, \Delta_2}\ \otimes\mathbf{R}$$
$$\text{if } \Delta_2 = \Delta_3 = \emptyset \text{ or } \Delta_2 = \Gamma_1 = \emptyset \text{ or } \Delta_3 = \Gamma_2 = \emptyset$$

$$\frac{\Gamma \vdash \Delta_1, a{:}A, b{:}B, \Delta_2}{\Gamma \vdash \Delta_1, a \parr b{:}A \parr B, \Delta_2}\ \parr\mathbf{R} \qquad \frac{\Gamma_1, a{:}A, \Gamma_2 \vdash \Delta_1 \quad \Gamma_3, b{:}B, \Gamma_4 \vdash \Delta_2}{\Gamma_3, \Gamma_1, a \parr b{:}A \parr B, \Gamma_4, \Gamma_2 \vdash \Delta_1, \Delta_2}\ \parr\mathbf{L}$$
$$\text{if } \Gamma_2 = \Gamma_3 = \emptyset \text{ or } \Gamma_2 = \Delta_1 = \emptyset \text{ or } \Gamma_3 = \Delta_2 = \emptyset$$

$$\frac{\Gamma_1, a{:}A, \Gamma_2 \vdash \Delta}{\Gamma_1, \kappa a{:}A, \Gamma_2 \vdash \Delta}\ \kappa \qquad \frac{\Gamma_1, b{:}B, a{:}A, \Gamma_2 \vdash \Delta}{\Gamma_1, a{:}A, b{:}B, \Gamma_2 \vdash \Delta}\ \mathbf{E}_\tau \text{ if } A \neq B \qquad \frac{\Gamma_1, b{:}A, a{:}A, \Gamma_2 \vdash \Delta}{\Gamma_1, a{:}A, b{:}A, \Gamma_2 \vdash \Delta}\ \mathbf{E}_\varsigma$$

$$\frac{\Gamma_1, \kappa b{:}B, a{:}A, \Gamma_2 \vdash \Delta}{\Gamma_1, a{:}A, \kappa b{:}B, \Gamma_2 \vdash \Delta}\ \mathbf{E}_\kappa(\text{left}) \quad \frac{\Gamma_1, b{:}B, \kappa a{:}A, \Gamma_2 \vdash \Delta}{\Gamma_1, \kappa a{:}A, b{:}B, \Gamma_2 \vdash \Delta}\ \mathbf{E}_\kappa(\text{right})$$

Fig. 5. The multiplicative fragment of the sequent calculus for classical linear logic, showing both types and formulae. **C** is slightly modified from the traditional presentation of contraction, and **CD** is a combination of the traditional contraction and dereliction laws. In the structural rules, n and m are polynomials.z may be a polynomial or ω.

The reader's attention is called to the translation of functions which throw away their arguments. An example would be the function 'pointless', defined thus:

$$\text{pointless } (x, y) \Rightarrow \{\} \text{ if True;}$$

As it is not possible to use uncontrolled applications of the Weakening rule in linear logic, it would appear that such functions are untranslatable. However, Fig. 8 shows that this view is mistaken, and that translation of such functions is possible. Part of the reason for this is that our translation rules in Fig. 7 render functions in a form equivalent to $!(A \multimap B)$ rather than the more conventional $(!A) \multimap B$.

$$\mathbf{1}^{\perp} \overset{\text{def}}{=} \perp \qquad \perp^{\perp} \overset{\text{def}}{=} \mathbf{1} \qquad (\forall x.a)^{\perp} \overset{\text{def}}{=} \exists x.(a^{\perp}) \qquad (!a)^{\perp} \overset{\text{def}}{=} ?(a^{\perp})$$

$$\perp \parr a = a \qquad \mathbf{1} \parr a = \mathbf{1} \qquad \perp \otimes a = \perp \qquad \mathbf{1} \otimes a = a$$

$$a^{\perp\perp} = a \qquad a \multimap b \overset{\text{def}}{=} a^{\perp} \parr b \qquad (a \otimes b)^{\perp} = a^{\perp} \parr b^{\perp}$$

Fig. 6. Some important equivalences in Linear Logic. The last four entries in the table indicate the distinguished elements of each multiplicative connective. $\otimes$ and $\parr$ are commutative.

$$\begin{array}{lll}
\mathcal{T}[\![f_1 + f_2]\!] & = \mathcal{T}[\![f_1]\!], \mathcal{T}[\![f_2]\!] & (1) \\
\mathcal{T}[\![f\ \{e^+\}]\!] & = \mathcal{T}[\![e^+]\!], \mathcal{T}[\![f]\!] & (2) \\
\mathcal{T}[\![fn(x_1 \ldots x_n) \to e^+ \text{ if } e']\!] = & & (3) \\
\multicolumn{3}{r}{!_{\infty}\kappa\forall_{i=1}^{n}\mathcal{T}[\![x_{n-i+1}]\!].(\mathcal{T}[\![e']\!])^{\perp} \parr ((\bigotimes_{j=1}^{n}\mathcal{T}[\![x_j]\!])^{\perp} \parr \mathcal{T}[\![e^+]\!])} \\
\mathcal{T}[\![e_1, e_2]\!] & = \mathcal{T}[\![e_1]\!] \otimes \mathcal{T}[\![e_2]\!] & (4) \\
\mathcal{T}[\![\langle e_1, e_2\rangle]\!] & = \mathcal{T}[\![e_1]\!] \parr \mathcal{T}[\![e_2]\!] & (5) \\
\mathcal{T}[\![True]\!] & = \mathbf{1} & (6) \\
\mathcal{T}[\![False]\!] & = \perp & (7) \\
\mathcal{T}[\![e_1 \wedge e_2]\!] & = \mathcal{T}[\![e_1]\!] \otimes \mathcal{T}[\![e_2]\!] & (8) \\
\mathcal{T}[\![e_1 \vee e_2]\!] & = \mathcal{T}[\![e_1]\!] \parr \mathcal{T}[\![e_2]\!] & (9) \\
\mathcal{T}[\![\neg e]\!] & = \mathcal{T}[\![e]\!]^{\perp} & (10) \\
\mathcal{T}[\![x]\!] & = x & (11) \\
\mathcal{T}[\![e_1 + e_2]\!] & = e_1 + e_2 & (12) \\
 & \vdots & \\
\mathcal{T}[\![e_1 \bmod e_2]\!] & = e_1 \bmod e_2 & (13) \\
\mathcal{T}[\![\{\}]\!] & = \perp & (14)
\end{array}$$

Fig. 7. The translation rules from Gamma programs to LL. Angle brackets indicate tupling. Note that there is no translation of "∘" in this table; an explanation of this is given in the text. One- and many-shot functions can be translated used a trivial variant of (3), not shown.

Bounded modalities allow us to give an upper bound to the number of times a function can be applied. In other words, we can identify the minimum number of times a function must be applied to give a particular answer. The mechanism can be used to allow one-shot (or n-shot) functions, if so desired, and functions which may be applied an ω times. A one-shot function, by definition, may only be applied *at most* once; it is then discarded. An n-shot function is applied *at most* n times before being discarded. Programming experience has shown that these are a useful addition to the language, but we do not use them in the examples presented in this paper. They are also useful for calculating complexity measures,

$$
\begin{aligned}
&\mathcal{T}[\![\text{pointless }(x,y) \Rightarrow \{\} \text{ if } \mathit{True}]\!]\\
=&\ \{(3)\}\\
&!_\omega\kappa\forall\mathcal{T}[\![y]\!],\mathcal{T}[\![x]\!].(\mathcal{T}[\![\mathit{True}]\!])^\perp⅋((\mathcal{T}[\![x]\!]\otimes\mathcal{T}[\![y]\!])^\perp⅋\mathcal{T}[\![\{\}]\!])\\
=&\ \{(11),\ (11),\ (6),\ (11),\ (11),\ (14)\}\\
&!_\omega\kappa\forall y,x.\mathbf{1}^\perp⅋((x\otimes y)^\perp⅋\bot)\\
=&\ \{\text{equivalence rules}\}\\
&!_\omega\kappa\forall y,x.\bot⅋((x\otimes y)^\perp⅋\bot)\\
=&\ \{\text{equivalence rules, commutativity}\}\\
&!_\omega\kappa\forall y,x.(x\otimes y)^\perp
\end{aligned}
$$

Fig. 8. The translation of 'pointless'.

as Girard et al. have shown [19].

We have not translated the "$\circ$" operator. Looking at the SOS for Gamma in Fig. 2, we see that a dot requires that the first function terminates before the second is applied. It is interesting to note that Hankin and Le Métayer have both intimated, during private conversations, that they do not feel that the sequential composition of Gamma fits well with the rest of the language. It is certainly the case that sequential composition requires global knowledge of the state of the computation. This contrasts with the spirit of the model, which allows functions to have only local knowledge of the multiset. This is the very reason why the operator is difficult to encode in LL: it needs to know something about the sequence of wff of which it is a member, viz. whether any of the other functions in the sequence can successfully be applied. We hope to provide a translation for "$\circ$" in the future. However, even without dot composition, we can translate a large number of programs. In fact, Gamma is perhaps unique in possessing a sequential composition operator: other related formalisms such as UNITY and Action Systems have no such operator. Certainly it is possible to code a class IV cellular automata ([34]) in Gamma without sequential composition, thus showing that Gamma without sequential composition is capable of universal computation.

5 Executing Gamma Programs in LL

The eagerness of Gamma allows us to write an extremely simple set of expression reduction laws, as shown in Fig. 9. The eagerness implies that expression reduction occurs immediately after the variables in an expression are bound. We shall see this in our later derivations.

We can now translate 'addup' into an LL wff.

$$\mathcal{T}[\![\mathit{addup}]\!]\ =\ !_\omega\kappa\forall y{:}\mathit{Int},x{:}\mathit{Int}.((x\otimes y)^\perp⅋((x+y){:}\mathit{Int}))$$

An example 'execution' of the program, with an initial multiset, is given in Fig. 10. Notice that function application is realised as an application of **CD**

$$\begin{array}{rlrl}
[\![constant]\!] = & constant & [\![e_1 > e_2]\!] = \mathbf{1} & \text{if } e_1 > e_2 \\
[\![e_1 + e_2]\!] = & [\![e_1]\!] + [\![e_2]\!] & [\![e_1 > e_2]\!] = \bot & \text{if } e_1 \not> e_2 \\
& \vdots & \vdots &
\end{array}$$

Fig. 9. The reduction laws for expressions.

or $\mathbf{CD}_\omega$ followed by (in a bottom-up reading) an application of the expression reduction laws (if applicable), neutral element laws and $\wp L$. There may be instances of the contraction rules applied to different formulae between these, however. The neutral element laws were given in Fig. 6. The four phases of function application: variable binding, condition evaluation, expression evaluation and storage reclamation, are thus clearly distinguished. We can therefore analyse the effects on each of these caused by changes in the order of the wff in the sequence, or by changes in the order of application of rules of LL.

Comparing the LL proof in Fig. 10 with the SOS execution in Fig. 3, we see that the results of executing the program are the same, as we would expect. In fact, we can show that the set of possible results is always the same for both the LL semantics and the SOS, by an easy argument. The main difference is that LL proof trees highlight the possibilities for parallel function application which are not visible in the SOS semantics, because the former is a true concurrency semantics whereas the latter is an interleaving semantics.

Examination of the proof trees generated for this 'program' with different numbers of data, gives us a number of insights. We first give some intuitions about how these trees should be read.

- Contraction corresponds to copying a function for parallel execution. Several functions could be created in this way, all of which move to their data *before* the application of the $\forall$ rule. Several $\forall$ rules can then be applied together. This implements parallel composition of functions on disjoint subsets of the multiset, as can be seen at the top of the proof tree.
- Applications of Weakening correspond to discarding of unwanted functions, as can be seen towards the bottom of the tree.
- Bounded exponentials can be used to determine a lower bound on the number of function applications needed to ensure termination of a computation, as can be seen at the bottom of the proof tree.
- We can identify trees with partially-applied functions at their leaves with trees in which such functions were discarded *before* they were partially applied. For Gamma does not return partial functions as values of computations, and our semantics must reflect the fact that functions will only be applied when all of their arguments are present. There is no danger in our identification, except that it allows functions to be wasted in an attempted application when they lack sufficient data. This feature highlights the necessity, in an implementation, of checking the availability of a function's

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{1\vdash 1}\,\mathbf{A} \qquad \dfrac{}{2\vdash 2}\,\mathbf{A}}{1,2\vdash 1\otimes 2}\,\otimes\mathbf{R}}{1,2,(1\otimes 2)^{\perp}\vdash\emptyset}\,{}^{\perp}\mathbf{L} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{3\vdash 3}\,\mathbf{A} \qquad \dfrac{}{7\vdash 7}\,\mathbf{A}}{3,7\vdash 3\otimes 7}\,\otimes\mathbf{R}}{3,7,(3\otimes 7)^{\perp}\vdash\emptyset}\,{}^{\perp}\mathbf{L} \qquad \dfrac{}{10\vdash 10}\,\mathbf{A}}{3,7,((3\otimes 7)^{\perp}\parr 10)\vdash 10}\,\parr\mathbf{L}}{3,7,\forall y,x.((x\otimes y)^{\perp}\parr(x+y))\vdash 10}\,\forall\mathbf{L},\forall\mathbf{L},\mathbf{Arith}.}{3,4,(3\otimes 4)^{\perp},3,7,\forall y,x.((x\otimes y)^{\perp}\parr(x+y))\vdash 10}\,\parr\mathbf{L}}{1,2,((1\otimes 2)^{\perp}\parr 3),7,\forall y,x.((x\otimes y)^{\perp}\parr(x+y))\vdash 10}\,\parr\mathbf{L}}{1,2,(1\otimes 2)^{\perp}\parr(1+2),7,\forall y,x.((x\otimes y)^{\perp}\parr(x+y))\vdash 10}\,\mathbf{Arith}.}{1,2,\kappa\forall y,x.((x\otimes y)^{\perp}\parr(x+y)),7,\kappa\forall y,x.((x\otimes y)^{\perp}\parr(x+y))\vdash 10}\,\kappa*2,\forall\mathbf{L},\forall\mathbf{L}}{1,2,7,\kappa\forall y,x.(\cdots\parr\cdots),\kappa\forall y,x.(\cdots\parr\cdots),\kappa\forall y,x.(\cdots\parr\cdots),!_{n-2}\forall\cdots\vdash 10}\,\mathbf{W}}{1,2,7,\kappa\forall y,x.((x\otimes y)^{\perp}\parr(x+y)),\kappa\forall y,x.((x\otimes y)^{\perp}\parr(x+y)),!_{n-2}\forall\cdots\vdash 10}\,\mathbf{E}_{\tau},\mathbf{E}_{\tau}}{1,2,7,!_{n}\kappa\forall y,x.((x\otimes y)^{\perp}\parr(x+y))\vdash 10}\,\mathbf{CD},\mathbf{CD}
$$

Fig. 10. A proof showing the execution of 'addup'. '**Arithmetic**' corresponds to the use of expression-rewrite rules and neutral element laws.

arguments *before* it is applied.

For quantitative results, we note the equivalences shown in Fig. 11. For 'addup', these equivalences allow us to derive the results shown in Fig. 12. All of these may be gained by easy arguments.

Proof sketch for the case of '# functions applied': Define 'initial sequence' as the root of the derivation tree. Our functions are the only elements in our initial sequence which contain quantifiers. No rule introduces quantifiers in a premise except the structural rules, so no quantifiers can be present in any sequence unless they are present in wff unchanged during the derivation, or are present in wff generated through applications of **CD**, **C**, $\mathbf{CD}_{\omega}$, $\mathbf{C}_{\omega}$. Therefore, all of the quantifiers originate in functions in the initial sequence. Quantified variables can only be instantiated to numbers in this proof, because of the type constraints on the $\forall L$ rule. Therefore, the number of quantified variables which can be instantiated in the proof is equal to the number of numbers in the proof (assuming that we have enough copies of the function). For every pair of numbers added together, a new number (their sum) is created. Therefore, if our initial sequence of wff contains n numbers, we generate a binary tree of numbers with $2n-1$ nodes. However, at the root the function cannot be successfully applied because it does not have enough arguments. Therefore, there are $2n-2$ successful function applications. □

The number of elements permuted reflects the attempt to gain the maximal parallelism of the program. Were the reductions done sequentially, the number of elements to which the exchange laws were applied would have been zero.

Runtime information	LL proof tree property
# functions successfully applied	$\#\mathbf{CD} + \#\mathbf{CD}_\omega$
# tuples reclaimed	$\#^\perp\mathbf{L}$
# elements permuted	$\#\mathbf{Exchange}_\varsigma$
max. parallelism	max. $\#\forall$ on a line $\div$ fn. arity

Fig. 11. Some correspondences between features of LL proof trees for the `addup` program and features of an execution of the corresponding program. 'Elements' in the above table indicates one-tuples, not wff. By 'permuted' , we do *not* mean the flowing over each other of elements with different types, but only applications of exchange on pairs of adjacent elements with the same type.

Runtime information	Number of occurrences	
	This run	Per n numbers
# functions successfully applied	2	$2n - 2$
# tuples reclaimed	2	$2n - 2$
max. parallelism	1	n div 2

Fig. 12. Results from the LL derivation of 'addup' with an initial set of three elements.

5.1 Storage Reclamation of Used Data

We translated our functions in an unconventional way, using $A^\perp \mathbin{⅋} B$ instead of the more usual, and equivalent, $A \multimap B$. Examining the derivation tree of 'addup' shows why this particular translation was chosen. When a function is applied, it produces 'anti-information' (negated-wff), whose job it is to track down and annihilate one of the arguments of the function. In LL, anti-information allows us to apply the **A** rule, thus removing from the multiset the wff which is the negated counterpart of the anti-wff. We therefore see the operational interpretation of the negated formulae in the translation of a function: they correspond to the notion of storage reclamation of unwanted data. Notice that this reading implies that the storage reclamation need not take place when the wff first reacts, but is only required to take place at some point in the future *before* the wff of which it is the anti-wff is re-used in another reaction (for if that happened, then linearity would be violated). Similarly, it does not matter if the actual wff is destroyed by the anti-wff, only that a wff with the same type and value is destroyed by the anti-wff. This opens the possibility for very intelligent storage reclamation, which should be further explored. Using $A \multimap B$ for functions removes this freedom.

6 Magic Stirring: Gamma's Efficiency Haemorrhage

Detecting termination of a Gamma computation requires checking that every datum performs all the reactions of which it is capable. Such behaviour requires an $\mathrm{O}\left(\frac{n!}{(n-m)!}\right)$ search of all possible permutations of the multiset, where n is the cardinality of the multiset and m is the largest arity of any function. Clearly, this affects both sequential and parallel implementations adversely. The implementations in [13, 32] 'solve' the problem by partitioning the multiset into individual values, which move around the network and interact when they meet at a node (and certain conditions are satisfied). Obviously, the communication costs involved in transporting so many tiny packets of data (single multiset elements) are prohibitively high.

Consider the following example; an asynchronous one-dimensional cellular automaton. At each function application, the value at a point on a one-dimensional surface is updated, depending on the values of its neighbours. The Gamma program is shown in Fig. 13. The translation into LL of part of this program, is shown in Fig. 14.

$f + g\ \{\ldots\}$

$f(\langle x', y'\rangle, \langle x, y\rangle, \langle x'', y''\rangle)$
$\quad \Rightarrow \langle x, (y' \vee y \vee y'')\rangle, \langle x', y'\rangle, \langle x'', y''\rangle$
$\qquad \text{if } (x \bmod 2 = 0) \wedge (x' = x - 1) \wedge (x'' = x + 1)$

$g(\langle x', y'\rangle, \langle x, y\rangle, \langle x'', y''\rangle)$
$\quad \Rightarrow \langle x, y\rangle, \langle x', y'\rangle, \langle x'', y''\rangle$
$\qquad \text{if } (x \bmod 2 \neq 0) \vee (x' \neq x - 1) \vee (x'' \neq x + 1)$

Fig. 13. 1-D Cellular automaton in Gamma

We can derive a proof tree for the LL translation of the program, as shown in Fig. 15. We use the notation $\mathbf{E}_{\varsigma}^{O(n!)}$ to indicate the application of $\mathrm{O}(n!)$ permutations to order the data, where n is the cardinality of the largest set whose type appears in the argument list of the function. We see that such reorderings of the data occur every time a function has been successfully applied. This is clearly

$$
\begin{array}{l}
!_{\omega}\kappa\forall(x''\parr y''),(x\parr y),(x'\parr y').\\
\quad(((x \bmod 2=0)\otimes(x'=x-1)\otimes(x''=x+1))^{\perp}\parr\\
\qquad(((x\parr y)\otimes(x'\parr y')\otimes(x''\parr y''))^{\perp}\\
\qquad\quad\parr((x\parr(y'\parr y\parr y''))\otimes(x'\parr y')\otimes(x''\parr y''))))
\end{array}
$$

Fig. 14. LL translation of function f from the 1-D Cellular automaton program into linear logic. Function g is similar, and has been elided. It is not used in the later proofs.

the major source of inefficiencies in Gamma. We shall address this problem in the next section.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c}\vdots\\ (1\parr 1),(2\parr 0),(3\parr 0)\vdash(1\parr 1)\otimes(2\parr 0)\otimes(3\parr 0)\end{array}}{(1\parr 1),(2\parr 0),(3\parr 0),((1\parr 1)\otimes(2\parr 0)\otimes(3\parr 0))^{\perp}\vdash\emptyset}\,{}^{\perp}\mathbf{L}\qquad \dfrac{\dfrac{\begin{array}{c}\vdots\\ \cdots,(1\parr 1),(2\parr 1),(3\parr 0),!_{\omega}\kappa\forall\cdots\vdash\Delta\end{array}}{\cdots,(2\parr 1),(1\parr 1),(3\parr 0),!_{\omega}\kappa\forall\cdots\vdash\Delta}\,\mathbf{E}_{\varsigma}^{O(n!)}}{\cdots,((2\parr 1)\otimes(1\parr 1)\otimes(3\parr 0)),!_{\omega}\kappa\forall\cdots\vdash\Delta}\,\otimes\mathbf{L}}{(1\parr 1),(2\parr 0),(3\parr 0),((1\parr 1)\otimes(2\parr 0)\otimes(3\parr 0))^{\perp}\parr((2\parr 1)\otimes(1\parr 1)\otimes(3\parr 0)),!_{\omega}\kappa\forall\cdots\vdash\Delta}\,\parr\mathbf{L}}{\cdots(1\parr 1),(2\parr 0),(3\parr 0),((1 \bmod 2=0)\otimes(1=2-1)\otimes(3=2+1))^{\perp}\parr(\cdots\parr\cdots),!_{\omega}\kappa\forall\cdots\vdash\Delta}\,\mathbf{Arith.}}{\cdots,(1\parr 1),(2\parr 0),(3\parr 0),\forall(x''\parr y''),(x\parr y),(x'\parr y').(\cdots\parr\cdots),!_{\omega}\kappa\forall\cdots\vdash\Delta}\,\forall\mathbf{L}*3}{\cdots,(1\parr 1),(2\parr 0),(3\parr 0),!_{\omega}\kappa\forall(x''\parr y''),(x\parr y),(x'\parr y').(\cdots\parr\cdots)\vdash\Delta}\,\mathbf{CD},\kappa}{\cdots,(2\parr 0),\cdots,(1\parr 1),\cdots,(3\parr 0),!_{\omega}\kappa\forall(x''\parr y''),(x\parr y),(x'\parr y').(\cdots\parr\cdots)\vdash\Delta}\,\mathbf{E}_{\varsigma}^{O(n!)}
$$

Fig. 15. Fragment of execution of the automaton program in Linear Logic. **Arith.** is an abbreviation for **Arithmetic**. $\mathbf{E}_{\varsigma}^{O(n!)}$ is an abbreviation for $O(n!)$ occurrences of $\mathbf{E}_{\varsigma}$.

Recent work [20] has shown that Gamma can be implemented more efficiently on a shared memory machine, with the generation of permutations replaced by a PROLOG-like search strategy. It is not yet clear whether or not their approach will carry over well onto a distributed memory machine, but their reliance on a homogeneous data set would seem to indicate otherwise: backtracking search across a distributed data set suffers from the same efficiency shortcomings as permutation generation.

6.1 Locality-Conscious Multisets

We believe that the inherent inefficiencies in Gamma cannot be overcome by clever implementations. We believe that progress will only be made by moving the problem into the semantic domain and solving it there. Therefore, we outline a more radical alternative to those discussed above. We claim that the Gamma model conflates two issues, which we believe should be separated. Solutions (multisets) in Gamma are constantly permuted to ensure that all reagents come into contact with one another. Yet there are really *two* notions here. The first is the notion of locality: that reagents cannot react unless they are in contact with one another. The second is the notion of permutation generation: all elements come into contact with one another sooner or later (if they don't react with something else first). These two notions are confused, both in the quantifier-laden SOS of Gamma, and in the permutation-driven implementations. Both neglect to distinguish data which *have* locality and data which *change* locality, and account for the inefficiency of the language.

Therefore, we distinguish the locality of reagents from their motion. We make locality *primitive* to the model, and allow re-orderings of the data only when they do not violate the locality constraints. We shall discuss the details of this in the next section, and explain how arbitrary permutations of the data can be generated, if they are required. This return to, and refinement of, the original chemical-reaction metaphor, is the insight behind local linear logic and Local Gamma.

7 Local LL and Local Gamma

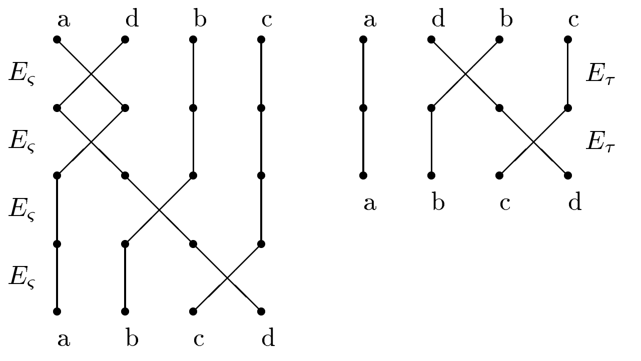
LL (and particularly Bounded LL) gave us a more fine-grained notion of resource-consciousness. Local linear logic extends this notion with locality-consciousness, so we offer the mantra 'movement is not free', to complete a trio. Local LL is linear logic with the normal exchange rule replaced with those shown in Fig. 16. The logic gives us a notion of locality by giving indices to elements in the sequence of wff and ensuring that the sequence remains ordered relative to some metric. Extending LL with a notion of order allows us to reason about structured data and partitioning of data on parallel machines. When used as a semantics for Gamma, we therefore have an improvement over the SOS of Gamma given earlier in that we have specific information at the semantic level concerning the costs of relocating data. We will use this power to show how the efficiency of Gamma can be improved. An illustration of the difference between the old and the new exchange laws is shown in Fig. 17. The laws of Local LL always move an incorrectly-ordered element *towards* the correct place in the sequence. The best that the traditional exchange rule could manage was to generate permutations of the solution until the correct ordering appeared. The traditional exchange rule of LL corresponds to the permutation generation of multiset languages. In both cases, it can be difficult to perform it intelligently when it is automated. Notice that the new exchange laws apply to a strict subset of the sequences to which the exchange law of Classical LL applied.

Yetter [35] and Abrusci [2, 3] effectively proposed a topology of the sequence of wff. We generalise the topology of the sequence to a multi-dimensional, open, surface. That is, formulae have (multi-dimensional) neighbourhoods within which they can move freely, but may only move outside of these neighbourhoods in certain circumstances. Thus, at a local level—when we are only concerned with one place—the behaviour of wff is the same as in LL. In a wider context, however, movement cannot be performed freely and, in some cases, not at all (see section 7 for a discussion).

$$\frac{\Gamma_1, b_j{:}A, a_i{:}A, \Gamma_2 \vdash \Delta}{\Gamma_1, a_i{:}A, b_j{:}A, \Gamma_2 \vdash \Delta}\,\mathbf{E}_\epsilon \quad \text{if } i > j \qquad \frac{\Gamma_1, b_j{:}A, a_i{:}A, \Gamma_2 \vdash \Delta}{\Gamma_1, a_i{:}A, b_j{:}A, \Gamma_2 \vdash \Delta}\,\mathbf{E}_\lambda \quad \text{if } i = j \qquad \frac{\Gamma_1, b_j{:}B, a_i{:}A, \Gamma_2 \vdash \Delta}{\Gamma_1, a_i{:}A, b_j{:}B, \Gamma_2 \vdash \Delta}\,\mathbf{E}_\tau \quad \text{if } A \neq B$$

$$\frac{\Gamma_1, \kappa b_j{:}B, a_i{:}A, \Gamma_2 \vdash \Delta}{\Gamma_1, a_i{:}A, \kappa b_j{:}B, \Gamma_2 \vdash \Delta}\,\mathbf{E}_\kappa(\text{left}) \qquad \frac{\Gamma_1, b_j{:}B, \kappa a_i{:}A, \Gamma_2 \vdash \Delta}{\Gamma_1, \kappa a_i{:}A, b_j{:}B, \Gamma_2 \vdash \Delta}\,\mathbf{E}_\kappa(\text{right})$$

Fig. 16. The exchange rules in Local LL.



(a) Magic stirring (b) Order preservation

Fig. 17. The difference between the exchange rules of LL and of Local LL.

Formulae of LL can be translated into formulae of Local LL with the same behaviour in two ways: either by prefixing them with a κ, or by giving every wff in a sequence the same index. However, by giving all formulae indices, then we restrict the possible derivations which can be made from the formulae. This is achieved because the rules given in Fig. 5 require the formulae to be in certain spatial configurations in order that they apply. In Local LL, then, we can generally only obtain a subset of the proofs available in LL. We aim to show that this subset is interesting.

Local Gamma is a variant of Gamma which replaces the unordered multiset with an ordered multiset. The syntax and translation rules given for Gamma in Fig. 1 are therefore unchanged. However, we add an annotation which indicates whether a particular expression should be regarded as an index on some ordering. Any ordering can be expressed which defines a metric space. Adding this annotation requires more translation rules, of the following form:

$$\mathcal{T}[\![\langle \widehat{x_1 \ldots x_n}, y\rangle]\!] = \mathcal{T}[\![y]\!]_{(x_1 \ldots x_n)}$$

In other words, the annotated vector $x_1 \ldots x_n$ is used as the index of the translated formula.

As in Local LL, elements can be permuted freely if they are at the same location. Elements at different locations can only move in such a way that they preserve the ordering of the multiset. Thus, new elements created at runtime can only move *towards* the position in the multiset dictated by their index. An example Local Gamma program is given in Fig. 18. Notice that we do not have to indicate the values of indices, only their presence. We believe that adding this annotation to the language will not result in programming errors, for the programmer should know which elements of which tuples are to be regarded as indices. In fact, forcing the programmer to be explicit about this should be regarded as good Local Gamma programming style.

$$order(x) \Rightarrow \langle \hat{x}, x\rangle$$

Fig. 18. An example local Gamma program. The ^ is the annotation indicating an element of a tuple which should be regarded as an index.

Programs whose data ordering is captured in this way become more efficient, in the general case. Of course, it is always possible to write a program (such as a permutation generator) whose behaviour is unaffected. However, in the author's experience such pathological programs are seldom written. An example of a program whose efficiency improves is the one-dimensional cellular automaton, whose time complexity reduces drastically. The Local Gamma version of the program is shown in Fig. 19, its translation into Local LL is shown in Fig. 20, and its 'execution' in Fig. 21.

The derivation tree for the cellular automaton in Local Linear Logic yields a quantity of interesting information. Firstly, we give the intuitive readings of the trees.

$f + g\ \{\ldots\}$

$$\begin{aligned}
&f(\langle \hat{x}', y'\rangle, \langle \hat{x}, y\rangle, \langle \hat{x}'', y''\rangle)\\
&\qquad \Rightarrow \langle \hat{x}, (y' \vee y \vee y'')\rangle, \langle \hat{x}', y'\rangle, \langle \hat{x}'', y''\rangle\\
&\qquad\qquad \text{if } (x \bmod 2 = 0) \wedge (x' = x - 1) \wedge (x'' = x + 1)
\end{aligned}$$

$$\begin{aligned}
&g(\langle \hat{x}', y'\rangle, \langle \hat{x}, y\rangle, \langle \hat{x}'', y''\rangle) \Rightarrow \langle \hat{x}, y\rangle, \langle \hat{x}', y'\rangle, \langle \hat{x}'', y''\rangle\\
&\qquad\qquad \text{if } (x \bmod 2 \neq 0) \vee (x' \neq x - 1) \vee (x'' \neq x + 1)
\end{aligned}$$

Fig. 19. 1-D Cellular automaton in Local Gamma.

$$\begin{aligned}
&!_{\omega}\kappa \forall y''_{x''}, y_x, y'_{x'}.(((x \bmod 2 = 0) \otimes (x' = x - 1) \otimes (x'' = x + 1))^{\perp} \parr\\
&\qquad\qquad ((y_x \otimes y'_{x'} \otimes y''_{x''})^{\perp} \parr ((y' \parr y \parr y'')_x \otimes y'_{x'} \otimes y''_{x''})))
\end{aligned}$$

Fig. 20. LL translation of function f from the 1-D Cellular automaton program into Local LL.

- Instances of Exchange$_{\epsilon}$ correspond to the preservation of ordering through moving new elements to their correct position in the solution.
- We may wish to examine the effect of partitioning the data over a distributed-memory machine. Ordering the set using indices gives the most natural way of partitioning data: all the elements whose indices are between certain bounds are in the same partition. For example, in a cellular automaton we know that with the Margolus or Von Neumann neighbourhoods ([30]) we need the values of only those elements within one unit's distance of the cell in question. By examining possible partitioning strategies, we can investigate their effects in terms of the lengths of the borders and the relative sizes of the partitions.
- In a parallel implementation of the language, we have a measure of the number of elements which will be communicated from one partition of the set to another, given a particular program. For example, if the number of the node is given as an index to a formula, then the number of formula for which that value changes during a single generation of the automata is exactly equal to the number of elements which have to be communicated from one place to another.

Comparing the LL version of the automaton with the Local LL version highlights the reduction in the number of permutations which have been generated by introducing the locality awareness: the exponential searches of the data space are replaced by O(n) steps for each new element. An actual implementation of the language would probably use a range-tree ([23]) or similar to reduce this to

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c}\vdots\\ 1_1,0_2,0_3 \vdash 1_1\otimes 0_2\otimes 0_3\end{array}}{1_1,0_2,0_3,(1_1\otimes 0_2\otimes 0_3)^{\perp}\vdash\emptyset}\;{}^{\perp}\mathbf{L}
\qquad
\dfrac{\dfrac{\begin{array}{c}\vdots\\ \cdots,1_1,1_2,0_3,!_{\omega}\kappa\forall\cdots\end{array}}{\cdots,1_2,1_1,0_3,!_{\omega}\kappa\forall\cdots\vdash\varDelta}\;\mathbf{E}_{\epsilon}^{O(n)}}{\cdots,(1_2\otimes 1_1\otimes 0_3),!_{\omega}\kappa\forall\cdots\vdash\varDelta}\;\otimes\mathbf{L}}{1_1,0_2,0_3,(1_1\otimes 0_2\otimes 0_3)^{\perp}\parr(1_2\otimes 1_1\otimes 0_3),!_{\omega}\kappa\forall\cdots\vdash\varDelta}\;\parr\mathbf{L}}{\cdots 1_1,0_2,0_3,((1 \bmod 2=0)\otimes(1=2\text{–}1)\otimes(3=2+1))^{\perp}\parr(\cdots\parr\cdots),!_{\omega}\kappa\forall\cdots\vdash\varDelta}\;\mathbf{Arith}.}{\cdots,1_1,0_2,0_3,\forall y''_{x''},y_x,y'_{x'}.(\cdots\parr\cdots),!_{\omega}\kappa\forall\cdots\vdash\varDelta}\;\forall\mathbf{L}*3}{\cdots,1_1,0_2,0_3,!_{\omega}\kappa\forall y''_{x''},y_x,y'_{x'}.(\cdots\parr\cdots)\vdash\varDelta}\;\mathbf{CD},\kappa}{\cdots,0_2,\cdots,1_1,\cdots,0_3,!_{\omega}\kappa\forall y''_{x''},y_x,y'_{x'}.(\cdots\parr\cdots)\vdash\varDelta}\;\mathbf{E}_{\epsilon}^{O(n)}
$$

Fig. 21. Fragment of execution of the automaton program in Local Linear Logic. Note that $\mathbf{E}_\epsilon$ has replaced $\mathbf{E}_\varsigma$.

O(log n). The quantitative results for one iteration of the automaton are shown in Fig. 22, and are obtained by easy arguments.

Runtime information	Number of occurrences per function application	
	LL	Local LL
# tuples reclaimed	3	3
# elements reordered	$n!$	$3n$

Fig. 22. Results comparing the automaton program in LL and Local LL, for a single iteration of an automaton with three elements. n is the number of tuples of type Num $\parr$ Num in the multiset. Notice how much more efficient the Local LL version is.

8 Related Work

In LL, Girard recognised that resources are not infinite—but he did not take into account that in many circumstances (for example on a distributed-memory machine), not all resources are equally easy to access. So while LL went a long way towards a logic of computation, it did not address issues of data access. To adequately reflect the behaviour of a real system, we must factor in the cost of accessing or relocating remote data. This is what Local LL does. So as well as charging the customer for the product, we must charge her for the delivery too.

Yetter [35] proposed a variant of LL which he called Cyclic LL. In Cyclic LL, wff may not be consumed except in the order in which they are presented. Cyclic permutations of the wff *are* allowed, however. Yetter's work amounts to introducing a weak notion of locality to the sequence of wff: every wff is in a different location on a one-dimensional torus. In order to embed LL in Cyclic LL, Yetter allowed formulae prefixed with a κ to move around freely in the sequence. We have adopted his idea.

Abrusci described a completely non-commutative linear logic [2, 3]: that is, formulae are not allowed to move around at all. This suggestion is tantamount to locating each formula at a different place on a one-dimensional, open, surface.

Guo [14] has presented a mechanism for translating terms of the lambda calculus into formulae of linear logic. Guo's system is more powerful than that here presented, for it allows the embedding of the entire lambda calculus and the description of graph reduction, laziness etc. However, his system has the disadvantage, due to its power, that in the general case, translation of complete functional programs yields unwieldy formulae. The links between these formulae and the lambda terms from which they were translated, are not always clear. Nevertheless, Guo's work has been a major inspiration for the current work.

Abramsky has produced a linear variant of the Chemical Abstract Machine [1], whose operation he describes in detail. However, our translation offers us the advantage that pieces of Gamma code can be translated into short logical sequences.

Andreoli [5] has examined a certain class of LL proof trees—focusing proofs—from the point of view of efficient proof search for a logic programming language. In the current paper, we have not concerned ourselves with focusing proofs, although an examination of these in the light of Local LL might be interesting. At this stage, we are not interested in the question of automatically executing Gamma programs in LL, and we assume the usual intelligence on the part of the prover. Providing an executable semantics is obviously an interesting goal, but we leave it for future research.

Meseguer [29] has presented a unifying framework for a number of models of concurrency, based upon category-theoretic considerations. The essential difference between our approach and his is that we are interested in an operational semantics of a rewriting language which is sufficiently low-level to give insights into the implementation of the language on a parallel architecture.

Yves Lafont [25] has proposed Interaction Nets as a generalisation of Girard's Proof Nets [18] and as a new sort of programming language in their own right. There may well be close connections between our LL semantics of Gamma and (a class of) Interaction Nets, but we have not yet explored this avenue.

Lincoln [26] has examined the complexity of LL fragments. An examination of the complexity of fragments of Local LL would also be an interesting area for further study.

9 Future Work

There are a number of directions for future work. The first is to provide a full proof- and model- theoretic examination of Local LL and an investigation of its relationship to other extant flavours of LL. The second direction is to examine in greater detail the ways in which Local LL proof trees mimic the general behaviour of an single program, multiple data (SPMD) system, by examining a number of programs and parallelisation strategies, in an attempt to extract quantitative information. Thirdly, an attempt should be undertaken to translate sequential composition. Fourthly, the definition of Local LL and the translations of Gamma programs to Local LL expressions could be used used to build an abstract machine for Local Gamma. The abstract machine instructions can be built from sequences of Local LL rule applications. From there an implementation could be produced, both for parallel and sequential machines. These implementations would provide interesting results concerning the efficiency and scalability of implementations based upon the 'locality-sensitive chemical reactions' model.

We are already working on a parallel implementation of Local Gamma. Initial results are promising but inconclusive, as only small programs can so far be compiled. More work needs to be done.

10 Conclusions

The continued existence of arrays as data structures attests to the presence in most algorithms of a notion of data locality. However, multiset transformation languages such as Gamma lack this notion. This accounts for their terrible efficiency problems. Unfortunately, Gamma's traditional SOS hid this difficulty, leading to a belief that the efficiency problem should be solved through clever implementations. We used Girard's resource-conscious linear logic (LL) to provide an alternative semantics for Gamma, which captured Gamma's non-determinism and potential parallelism through a straightforward translation scheme from Gamma programs into terms of LL. This semantics is more fine-grained than the traditional SOS of Gamma in that it highlighted the language's inefficiencies. We extended linear logic, making it locality-conscious. This logic we called Local LL. It does for data locality what LL does for resource allocation and discarding. We used the new logic to suggest an alternative Gamma in which the multiset is ordered according to some metric. We showed that the Gamma programming style is essentially unchanged, but that the inefficiencies are removed. Our semantics also gives us clues for efficient data partitioning for a parallel implementation, and yields information on the maximum parallelism we can attain from a Gamma program.

11 Acknowledgments

We would like to thank Anne Troelstra, Yi-Ke Guo, Marcel Beemster and Jon Mountjoy for proof reading this paper and providing a number of comments.

Thanks also go to Walter Hoffman, Eelco Visser, Wim Vree, Peter Sloot, Jaap Kaandorp and Jeroen Voogt for several helpful discussions. We would also like to thank the anonymous PLILP referees for their helpful suggestions.

References

1. S. Abramsky. Computational interpretations of linear logic. *Theoretical computer science*, 111(3):3–57, Oct 1993.
2. V. M. Abrusci. Non-commutative intuitionistic linear logic. *Zeitschrift fur Mathematische Logik und Grundlagen der Mathematik*, 36(1):297–318, 1990.
3. V. M. Abrusci. Phase semantics and sequent calculus for pure non-commutative classical linear propositional logic. *J. Symbolic Logic*, 56(4):1403–1451, Dec 91.
4. V. Alexiev. Applications of linear logic to computation. Technical Report TR93-18, University of Alberta, Saskatchewan, Canada, 1993.
5. J.-M. Andreoli. Logic programming with focusing proofs in linear logic. *J. Logic and Computation*, 2(3):297–347, 1992.
6. R. J. R. Back and R. Kurki-Suonio. Decentralisation of process nets with centralised control. In *2nd SIGACT-SIGOPS Symp. on Principles of Distr. Computing (PODC)*, pages 131–142, Montreal, Canada, 1983. Springer Verlag, Berlin.
7. R. J. R. Back and R. Kurki-Suonio. Distributed co-operation with action systems. *ACM transactions on programming languages and systems*, 10(4):513–554, 1988.
8. H. G. Baker Jr. Lively linear Lisp – "look ma, no garbage!". *ACM SIGPLAN notices*, 27(8):89–98, Aug 1992.
9. J.-P. Banâtre, A. Coutant, and D. le Métayer. Parallel machines for multiset transformations and their programming style. *Informationstechnik, Oldenbourg Verlag*, 30(2):99–109, 1988.
10. J.-P. Banâtre and D. le Métayer. Programming by multiset transformation. Research report PI 522, IRISA, Rennes, France, Mar 1990.
11. G. Berry and G. Boudol. The chemical abstract machine. In *17th Principles of programming languages*, pages 81–94, San Fransisco, California, Jan 1990. ACM, New York.
12. K. M. Chandy and J. Misra. *Parallel program design: A foundation.* Addison Wesley, Reading, Massachusetts, 1988.
13. C. Creveuil and G. Moguerou. Dérivation systématique d'un algorithme de segmentation d'images - un exemple d'application du formalisme Gamma. Research report 1049, INRIA Rocquencourt, France, Jun 1989.
14. J. Darlington and Y. Guo. Reduction as deduction. In J. R. W. Glauert, editor, *6th Implementation of Functional Languages*, pages 10.1–10.10. School of Information Systems, Univ. of East Anglia, Norwich, UK, Sep 1994.
15. L. Errington, C. Hankin, and T. Jensen. A congruence for gamma programs. In P. Cousot, M. Falaschi, G. Filè, and A. Rauzy, editors, *3rd Static Analysis (WSA), LNCS 724*, pages 242–253, Padova, Italy, 1993. Springer Verlag, Berlin.
16. D. Gelernter. Generative communication in linda. *ACM transactions on programming languages and systems*, 7(1):80–112, 1985.
17. D. Gelernter and A. J. Bernstein. Distributed communication via global buffer. In *1st SIGACT-SIGOPS Symp. on Principles of Distr. Computing (PODC)*, pages 10–18, Ottawa, Canada, 1982. ACM, New York.
18. J.-Y. Girard. Linear logic. *Theoretical computer science*, 50(1):1–102, 1987.

19. J.-Y. Girard, A. Scedrov, and P. J. Scott. Bounded linear logic: a modular approach to polynomial-time computability. *Theoretical computer science*, 97:1–66, 1992.
20. K. Gladitz and H. Kuchen. Shared memory implementation of the Gamma-operation. In J. R. W. Glauert, editor, *6th Implementation of Functional Languages*, pages 26.1–26.13. School of Information Systems, Univ. of East Anglia, Norwich, UK, Sep 1994.
21. C. L. Hankin, D. le Métayer, and D. Sands. A calculus of Gamma programs. Research Report 92/22, Dept. of Computing, Imperial College London, Jul 1992.
22. C. L. Hankin, D. le Métayer, and D. Sands. Transformation of Gamma programs. In M. Billaud, P. Castéran, M-M. Corsini, K. Musumbu, and A. Rauzy, editors, *Static Analysis (WSA 92)*, pages 12–19, Bordeaux, France, Sep 1992. BIGRE, 81-82.
23. P. H. Hartel, M. H. M. Smid, L. Torenvliet, and W. G. Vree. A parallel functional implementation of range queries. In P. G. M. Apers, D. Bosman, and J. van Leeuwen, editors, *Computing science in The Netherlands*, pages 173–189, Utrecht, The Netherlands, Nov 1989. CWI, Amsterdam.
24. Y. Lafont. The linear abstract machine. *Theoretical Computer Science*, 59:157–180, 1988.
25. Y. Lafont. Interaction nets. In *17th Principles of programming languages*, pages 95–108, San Fransisco, California, Jan 1990. ACM, New York.
26. P. Lincoln and T. Winkler. Constant-only multiplicative linear logic is NP-complete. *Theoretical computer science*, 135(1):155–169, 1994.
27. H. McEvoy. Gamma, chromatic typing and vegetation. Technical report in preperation, Dept. of Comp. Sys, Univ. of Amsterdam; Presented at the ESPRIT 9102 coordination meeting, Geneva, Dec 1994.
28. H. McEvoy and J. Kaandorp. *Multisets and their transformers as models for environmentally-sensitive growth.* Dept. Comp. Sys., Univ. of Amsterdam, 1995.
29. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical computer science*, 96(1):73–155, 1992.
30. T. Toffoli and N. Margolus. *Cellular automata machines.* MIT press, Cambridge, Massachusetts, 1991.
31. A. S. Troelstra. Lectures on linear logic. Lecture notes 29, Centre for the Study of Language and Information, Leyland Stanford Junior University, California, 1992.
32. M. Vieillot. Premiers pas de Gamma avec une PAM. Rapport de stage, IFSIC, IRISA, Univ. de Rennes, France, 1992.
33. P. L. Wadler. Linear types can change the world! In M. Broy and C. B. Jones, editors, *Programming concepts and methods*, pages 561–581, Sea of Gallilee, Israel, Apr 1990. North Holland, Amsterdam.
34. S. Wolfram. *Cellular automata and complexity.* Addison Wesley, Reading, Massachusetts, 1994.
35. D. Yetter. Quantales and (noncommutative) linear logic. *J. Symbolic logic*, 55(1):41–64, 1990.

This article was processed using the LaTeX macro package with LLNCS style